Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Heering, J. Sidi, A. Verhoog (eds)

Generation of interactive programming environments - GIPE
Intermediate report

Computer Science/Department of Software Technology Report CS-R8620 May

Generation of Interactive Programming Environments

— GIPE —

Intermediate Report

Editors

Jan Heering
CWI
Amsterdam
The Netherlands

Jacqueline Sidi
SEMA METRA
Montrouge
France

Ard Verhoog
BSO/Automation Technology bv
Utrecht
The Netherlands

The objective of the GIPE-project is to realize a prototype system for generating interactive programming environments from formal language definitions. Partners in this five-year project, which has started in November 1984, are BSO/Automation Technology (Utrecht), CWI (Amsterdam), INRIA (Rocquencourt/-Sophia-Antipolis), and SEMA (Montrouge). In this intermediate report we describe a common development environment, various language definition formalisms, and the environment generator itself.

69 D21
69 D26
69 D41
69 D44
[illegible]

1986 CR Categories: D.2.1 [**Software Engineering**]: Requirements/Specifications - *Languages;* D.2.6 [**Software Engineering**]: Programming Environments; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.4 [**Programming Languages**]: Processors; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages.

1980 Mathematics Subject Classification: 68B99 [**Software**].

Key Words & Phrases: generation of programming environments, language definition, syntax definition, type checking, inference rule semantics, structured operational semantics, algebraic semantics.

Note: Partial support received from the European Communities under ESPRIT project 348.

Report CS-R8620
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Participants

D. Clement (SEMA)
J. Despeyroux (INRIA - Sophia-Antipolis)
T. Despeyroux (INRIA - Sophia-Antipolis)
N.W.P. van Diepen (CWI)
L. Gallot (INRIA - Sophia-Antipolis)
L. Hascoët (INRIA - Sophia-Antipolis)
J. Heering (CWI)
P.R.H. Hendriks (CWI)
G. Kahn (INRIA - Sophia-Antipolis)
P. Klint (CWI/University of Amsterdam)
B. Lang (INRIA - Rocquencourt)
J. Sidi (SEMA)
J. Symes (BSO/Automation Technology)
G.P. Verhoog (BSO/Automation Technology)

External consultants

J.A. Bergstra (University of Amsterdam/University of Utrecht)
P. Borras (SEMA)
M. Devin (INRIA/Bull)
V. Pascual-Migot (INRIA - Sophia-Antipolis)

BSO/Automation Technology bv
Kon. Wilhelminalaan 3
3527 LA UTRECHT
The Netherlands

INRIA
Centre de Sophia-Antipolis
Route des Lucioles
06565 VALBONNE CEDEX
France

CWI
Kruislaan 413
1098 SJ AMSTERDAM
The Netherlands

SEMA METRA
16-18, Rue Barbès
92126 MONTROUGE CEDEX
France

INRIA
Centre de Rocquencourt
Domaine de Voluceau
Rocquencourt
78153 LE CHESNAY CEDEX
France

OUTLINE OF THE GIPE PROJECT

Generation of Interactive Programming Environments
Outline of the GIPE Project

J. Heering (CWI)
G. Kahn (INRIA)
P. Klint (CWI)
B. Lang (INRIA)

1. INTRODUCTION

1.1. Objectives of the GIPE project

The objectives of the project are:

- To investigate the possibilities for automatically generating interactive programming environments from language specifications. A "programming environment" is here understood as a set of integrated tools for the incremental creation, manipulation and transformation of structured, formalized objects such as programs in a programming language, specifications in a specification language, or formalized technical documents.
- To create a software environment that allows experimenting with formalisms for specifying various aspects of structured, formalized objects.
- To evaluate the adequacy of various formalisms (some of them existing, some of them to be designed as part of the project) for the specification of various aspects of (programming) languages (such as concrete and abstract syntax, type checking rules, dynamic semantics) and interactive programming environments (syntax-directed editing, pretty printing, program manipulation and transformation).
- To create appropriate interfaces and a software architecture for the integration of selected language specification formalisms in a single system.
- To manipulate large formal specifications (which may even use combinations of different formalisms), to incrementally maintain their consistency, and to compile such specifications into executable programs.
- To design and implement a prototype system for the generation of interactive programming environments from language specifications.

1.2. Advances

Many difficult problems have to be solved in order to achieve these objectives. Advances are needed in three directions. First, most problems involved have only been solved in a batch-oriented setting but not in an interactive setting. For example, the problem of generating parsers from BNF grammars can be considered to be solved (if we exclude error recovery). However, the parsing problem is far from solved if we want to parse incrementally, i.e., when editing and parsing are being combined as is the case in a syntax-directed editor. Secondly, some problems have not been dealt with at all, such as, for instance, the simultaneous use of several semantic formalisms. Thirdly, none of the approaches so far has yielded an integrated system capable of handling large formal definitions.

For the sake of completeness, we give here a list of partly solved and partly open questions:

- Specification of lexical and concrete syntax of programming languages and the derivation of lexical scanners and parsers from these specifications is largely solved. However, the incremental parsing problem is almost entirely open.
- Specification of abstract syntax and the automatic derivation of functions for the construction of abstract syntax trees is solved. The treatment of comments — which is trivial in the case of batch-oriented compilation — is not yet very well understood.
- Specification of type checking rules and the generation of incremental type checkers is a subject of current research.
- Specification of pretty printing and automatic derivation of pretty printers has only been solved for simple cases. Open problems are the specification of flexible output formats, of multiple views of the same program, and of more advanced (e.g. graphical) output techniques.
- Combination of parsers or pretty printers to handle multiple formalism documents is still an unexplored area.
- Specification of dynamic semantics and generation of language processors from such specifications is a subject of current research. Some experimental systems exist in this area. Major problems to be solved are: balancing the (theoretical) adequacy of the modeling power of specification formalisms with their descriptive convenience, modularity of specifications, specifications using multiple formalisms, and compilation techniques for obtaining acceptable performance of the generated language processors.
- Some results are available in the area of automatic derivation of syntax-directed editors from language specifications. Many problems remain to be solved: specification of dialogues, multiple input devices, etc.

We expect that the proposed project will lead to advances in most (if not all) of the above-mentioned areas.

2. POSITION OF GIPE WITHIN THE ESPRIT PROGRAMME

This project carries out research on specification techniques for programming languages and interactive programming environments. It involves primarily the subjects mentioned in ESPRIT area 2.4. The project aims at developing expertise that is essential for the development of future, more flexible and advanced, software development environments. As such it can be considered as a long term complement to project 32 “Portable Common Tool Environment (PCTE)”. The prototype programming environment generator will be developed under Unix. As a consequence, it is very likely that the resulting prototype system can be interfaced with the PCTE, and we are making efforts in this direction.

The project does not depend heavily on the results of other ESPRIT projects, but the resulting Interactive Programming Environment Generator may very well be used for maintaining and processing the many kinds of formal specifications (based on many different formalisms) that will emerge from other ESPRIT projects in widely different areas, such as formal semantics of interfaces, specifications of office systems, office document languages, formal descriptions in Computer Integrated Manufacturing, etc.

On a more global, longer-term scale, the results of the project lead to an order-of-magnitude reduction in the costs of developing programming environments for existing (e.g. Ada, Chill) and next generation languages (e.g. Concurrent Prolog, functional programming languages). This cost reduction will be the consequence of several aspects of the proposed Interactive Programming Environment Generator:

- Factorisation of the design, development, and maintenance efforts and costs for all components of interactive programming environments that are language independent.

- Reduction in design, development, and maintenance costs for all language dependent aspects of these environments, resulting from the use of higher-level, specialized, and thus more tractable specification formalisms, together with the assistance from specialized design tools. The use of high-level, mathematically well understood specification formalisms should ultimately provide an additional reduction of maintenance costs resulting from the replacement of traditional benchmark validation by more reliable formal certification techniques [37].
- Uniformity of the generated interactive environment as to conceptual structure and user interface for all programming or specification languages used. This will reduce the costs of training users, and allow a greater mobility of users between sites and/or languages.

In addition, the ease and low cost of environment generation should reduce the design costs of new languages by permitting rapid prototyping and experimentation.

3. STATE OF THE ART

3.1. Introduction

One of the most successful paradigms in Computer Science consists in isolating a subclass of problems that can be completely formalized. On the basis of this formalisation, one may then build general purpose tools so that solving a problem in the given class is reduced to presenting and debugging its formal specification. The best known examples of that paradigm are lexical analysis and parsing: lexical analyser generators and parser generators have been the subject of very intense study since the pioneering work of Brooker and Morris [2] and Ross [34]. First, a great deal of attention was given to the theoretical framework — finite automata, regular expressions, context-free languages, LR(k) languages — followed by considerations of efficiency and flexibility of use in various computing contexts. Today, Lex [24] and Yacc [18] are standard tools in the Unix system.

Lexical analysis and parsing represent but a small fraction of the computing task performed by a translator. To push compiler generation further, research has proceeded along two different lines:

(1) The first strategy consisted in theoretical research on the semantics of programming languages. Advocated early by C. Strachey and the authors of the Vienna Definition Language, this approach resulted in a flurry of interesting results during the seventies. In particular, the ideas coming from Denotational Semantics have cristallized into a beautiful experimental system created by P. Mosses at Oxford. His Semantics Implementation System (SIS) is able to interpret a formal description of a programming language, say L, and execute on that basis a program in L.

(2) The second strategy consisted in analyzing in greater detail what is going on computationally in certain phases of a compiler. As a result, D. Knuth introduced attribute grammars, which immediately appealed to less theoretically inclined researchers, even though, of course, they gave rise to quite interesting theoretical problems. Systems allowing the processing of large attribute grammar descriptions were built and used in the seventies in many places. A typical example is the GAG system of Kastens c.s. [20] used in Karlsruhe for the construction of an Ada compiler.

As these developments were taking place, use of computers in a time-shared, interactive fashion spread throughout research and industry, culminating today in the concept of an individual workstation. As a consequence, it became abundantly clear that a programmer was not working with a single — albeit sophisticated — language processor, but rather within a programming environment including a wealth of computing tools. Experimental systems have explored different approaches and ideas that should, in one form or another, be incorporated in such systems. Clearly, in such environments, many features are directly linked to the syntactic and semantic nature of the language under consideration, and much of the behavior of the interactive system must be derived from a formal characterisation of the various aspects of this language. In a nutshell, this is what the present proposal is trying to achieve.

3.2. Language-dependent programming environments

The creation and maintenance of software is becoming increasingly expensive. To improve upon this situation several software tools and language-specific programming environments have been proposed, implemented, and have come into use, some of them with considerable success.

Programming environments are based on the premise that the productivity of a programmer can be increased by relieving him from the burden of many administrative and clerical aspects of programming. It is the task of a programming environment to take care of these aspects automatically, and, evidently, a programming environment can provide more assistance if it contains more knowledge about the programming language being used. Typical services provided by such *language-dependent* programming environments are: syntax-directed editing, debugging, pretty printing, separate compilation, maintenance of libraries of programs, and incremental dataflow analysis. Systems that fall into this category are: Interlisp [36], Smalltalk [11], and the Cornell Program Synthesizer [35].

The implementation of a programming environment dedicated to a particular language (e.g. Pascal, Ada, Chill) requires a very substantial design and implementation effort. This is exemplified by the current efforts to build Ada Programming Support Environments (APSEs).

3.3. Language-independent programming environments

If one takes a more general point of view, it becomes clear that all these language-specific programming environments have many traits in common. These can, in principle, be factored out by developing *language-independent* programming environments, which can be tailored towards a particular language by entering a definition of that language into the system. Another, even more important, argument in favor of language-independent programming environments is that they provide a uniform user interface: programmers using different languages can still work with similar if not identical programming environments for the various languages.

Existing systems in this category are: Mentor [5,6], the Synthesizer Generator [31], Gandalf [25] and CEYX [16]. These systems are still under development and address only parts of the problems of language-independent programming environments.

These systems have several properties in common. They are all language-independent and use similar notions to structure the definitions of new languages. The use of the word "language" is somewhat misleading and restrictive here: these systems are, in fact, all dedicated to the manipulation of hierarchically structured information in general. Programs in a programming language are just one example of such information. Other examples are systems for document preparation, for VLSI design, and for proof checking. All these systems use *trees* as their primary data structure for representing the objects that are being manipulated.

A definition for a new language can globally be subdivided in definitions for:

lexical syntax:
which defines the tokens of the language, i.e., keywords, identifiers, punctuation marks, etc.

concrete syntax (also: context-free syntax):
which defines the textual form of programs, i.e., the sequences of tokens that constitute a legal program.

abstract syntax:
which defines the abstract tree structure underlying the concrete (textual) form of programs.

tree construction:
which specifies the mapping from parse-tree to abstract syntax tree.

unparsing (also: pretty printing):
which defines the mapping of a program from its abstract syntactic form to its written representation.

static semantics:
which defines certain constraints on programs that can be verified without executing them, i.e., constraints that do not depend on input data. For instance, in a "legal" program all variables

should have been declared, all expressions should be type consistent, etc.

dynamic semantics:
which defines the meaning of a program, i.e., the relation between its input and output data.

This list is not exhaustive; one could also include documentation, correctness proof systems, etc.

Starting from a language definition, these systems process programs in the defined language in similar ways. The definition of lexical and concrete syntax contains sufficient information to create a parser for the newly defined language and to build a parse-tree. A parse-tree typically contains non-terminals as nodes and terminals as leaves. Subsequently, this parse-tree is transformed into an abstract syntax tree. An abstract syntax tree typically contains semantic notions as nodes and only constants and identifiers as leaves. The abstract syntax tree can be built directly and independently of the parse-tree, when a program is created during syntax-directed editing. If desired, the inverse operation can be carried out: the abstract syntax tree can be transformed into source text by means of unparsing (pretty printing). In fact, "unparsing" is a misnomer since the operations of both the scanner, the parser, and the abstract syntax tree constructor have to be inverted in order to transform an abstract syntax tree back into source text.

A final similarity between these systems is that they all allow syntax-directed editing for each new language with a standard user interface.

Many problems are not or only partly addressed by existing systems. We mention only a few: specifying dynamic semantics, language-independent debugging, language-independent tools for flow analysis, and general, language-independent, optimization techniques.

3.4. Language specifications

It is clear that most of the unsolved problems in language-independent programming environments have to do with the specification of static and dynamic semantics. For instance, how can one — during syntax-directed editing — enforce the immediate, incremental checking of the constraints imposed by the static semantics of a language? How can one reverse the direction of computing to support *undo* facilities in editors and debuggers? In the following two subsections we briefly survey two specification formalisms that are particularly suited to our purposes: specifications based on inference rules and on abstract algebra.

3.5. Specifications based on inference rules

Much effort has been invested in the development of attribute grammars (for specifying static semantics) and denotational semantics (for specifying dynamic semantics). However, both formalisms have their deficiencies.

The major deficiencies of attribute grammars are:

- Specifications based on attribute grammars often result in heavy — seemingly very low level — notations.
- Attributes are attached to single tree nodes rather than tree patterns; as a consequence structural information obscures attributes.
- The formalism seems more appropriate for static calculations rather than for dynamic execution.
- Semantic analysis of attribute grammars is difficult due to the low level of the formalism.

The major deficiencies of denotational semantics are:

- For static semantics, denotational semantics equations are clumsy and the ways to specify tree traversal are not very elegant.
- It is difficult to describe parallel constructs in the dynamic semantics of a language.
- Pure denotational semantic definitions may result in overspecifications.

As a result, an approach advocated by Gordon Plotkin in his lecture notes *A Structural Approach to Operational Semantics* [29] will be a focus of investigation for the project. This approach consists in presenting an axiomatization — via axioms and inference rules — of an abstract machine. In this method, the best aspects of earlier methods are retained:

- Semantics is defined recursively on the structure of the formalism (as is the case in denotational semantics).
- The definition is declarative (Attribute Grammars, Predicate logic).
- Extensive use of pattern matching and overloading (SIS [26], Hope [3]).

Furthermore, progress is made on several key points:

- The definitions are short, readable, elegant.
- Several concepts from attribute grammars can be recovered (such as incremental computation).
- Interfacing a definition of this kind with recursive semantic equations or abstract algebraic specifications seems feasible.
- Specifying concurrent behaviour is easy and natural.
- Static semantics and translation can easily be expressed.
- Constraints — in particular those occurring in software engineering and specifications of man-machine interfaces — are likely to be easily expressible as well.

One difficult aspect deserves to be mentioned however: even though this approach dates back to Gentzen, and has been extensively used in theoretical work (Barendregt [1]), not a great deal of meta-theory is known today. We expect that significant theoretical advances in this area will take place as the project proceeds. It is clear that this topic, and we shall see this again later on, is directly linked to studies in compiling Prolog programs.

3.6. Algebraic specifications

Another promising method for specifying the static as well as dynamic semantics of programming languages is based on abstract algebra. The major use of this method has been for the specification of abstract data types. Most of the research in this area has concentrated on fully understanding the mathematical properties of specifications of relatively simple data types. It is clear, however, that the method can also be used for the specification of more complex data types and for specifying various aspects of programming languages and programming environments. There is currently a lot of activity in this field. The salient points are briefly summarized below.

- The so-called initial algebra semantics of algebraic specifications is rather well understood [8,9].
- The theory of the semantics of parametrized or otherwise incomplete specification modules and the composition of such modules is developing rapidly. Needless to say, libraries of modules and module composition operators will be of prime importance in an environment tailored towards algebraic specification [10,32].
- For (incremental) compilation and consistency checking of specification modules see the next section.
- Although increasingly large specifications are being produced — specifications currently exist for simple languages, editors, compilation schemes, etc. — the limits to what can profitably be specified algebraically are not yet clear.

3.7. Compilation of specifications

The idea of directly compiling and executing specifications presented via axioms and inference rules is new. The first problem is to define a machine processable version of the inference rule formalism. This is actually a language design activity and it requires many decisions: what is an adequate type structure, how does one make wise use of overloading, how does one separate structural conditions from other conditions that trigger the applicability of inference rules, how does one incorporate a form of modularity. With regard to compilation, a possible initial strategy is to produce Prolog code, which is then executed. The compilation process involves of course type checking of the inference rules and generating clauses containing control information that insulate the "inference rule programmer" from considerations of order in rules, or order in the premises of a rule. Very little work in this direction has been done so far, but initial attempts look extremely promising [4]. In terms of efficiency, it is likely that extremely efficient Prolog interpreters (and machines) will become available as the project proceeds. These interpreters will accommodate interaction with more conventional languages, such as Lisp or Pascal, so that inference rule specifications can be interfaced with other forms of specifications on the one hand, and with system programming tools (in particular graphics) on the other.

The development of specification languages is a natural continuation of the development of high-level programming languages in the past. Although compilation of specifications may be viewed in this light, this does not mean that conventional compilation techniques are adequate. Instead, theorem proving techniques are required and certain restrictions must be imposed on the formalisms used in order to make compilation possible. Furthermore, especially when the specifications are large, incremental compilation and proper modularization are an absolute must.

Prolog is an example where imposing restrictions on a formalism, viz. predicate logic, can lead to easily executable specifications [22]. Similarly, certain restrictions can be imposed on the equations of an algebraic specification so as to make their compilation to rewrite rules easy [13]. In general, however, more sophisticated methods are needed to obtain executable specifications. The algorithms needed have much in common with the algorithms used for Prolog-like formalisms and some work is being done to integrate both approaches (see for example [17]). The incremental compilation problem is largely open.

Not only (incremental) compilation of specifications is important in a programming environment, but also (incremental) consistency checking of specification modules [27,12,33,15]. Like the incremental compilation problem, the incremental type checking problem is largely open.

4. DESCRIPTION OF THE PROJECT

4.1. Phases of the project

The project is subdivided in the following phases:

- Construction of a shared software environment as a point of departure for experimenting with and making comparisons between language specification techniques. The necessary elements of this — Unix-based — software environment are: efficient and mutually compatible implementations of Lisp and Prolog, a parser generator, general purpose algorithms for syntax-directed editing and pretty printing, software packages for window management and graphics, etc. Most of these elements are already available or can be obtained elsewhere; the main initial effort will be to integrate these components into one reliable, shared software environment.
- A series of experiments that amount to developing sample specifications — based on different language specification formalisms, but initially based on inference rules and universal algebra — for a set of selected examples in the domain of programming languages, software engineering and man-machine interaction.
- Construction of a set of tools on top of the shared environment to carry out the above experiments. It will be necessary to create, manipulate, and check (parts of) language specifications and to compile them into executable programs. These tools draw heavily upon techniques used in object oriented programming (for manipulation of abstract syntax trees), automatic theorem proving (for inferring properties from given specifications to check their consistency and select potential compilation methods), expert systems (to organize the increasing number of facts that become known about a given specification) and advanced information processing in general (man-machine interfaces, general inference techniques, maintenance and propagation of constraints, etc.).
- The above experiments will indicate which of the chosen formalisms is most appropriate for characterizing various aspects of programming languages and interactive programming environments. These insights will be used in constructing a prototype system for deriving programming environments from language specifications. The envisioned "programming

environment generator" consists of an integrated set of tools and an adequate man-machine interface for the incremental creation, consistency checking, manipulation and compilation of language specifications.

A more detailed workplan is given in the next section.

4.2. Workplan

The project consists of two phases:

Phase I (years 1 and 2)
Establishment of a common software environment, selection of a set of representative examples, specification of selected examples in two specification formalisms, software tools for checking the specified examples, global design of a programming environment generator. This report gives the status of the project after the first year.

Phase II (years 3, 4 and 5)
Evaluation and integration of the specification formalisms and related tools, optimization of specifications, design of a man-machine interface, application to tough example.

At the end of years 2 and 5 we envisage a review of the results of the project by external (scientific) experts. In addition to this we will make use of highly qualified external consultants in order to assess the strong/weak points of the emerging system and, when necessary, to get advice on specific, scientific or technical, problems.

Here follows a complete list of all tasks in the project. The total estimated manpower for each task is given between square brackets.

T0 [0.2] — *Familiarization*
Familiarization of new team members.

T1 [1.8] — *Establishment of a common environment*
Construction, installation, and documentation of a common software development environment. See section D1 of this report.

T2 [1.0] — *Definition of common interfaces*
Definition of common software interfaces and internal representations of objects to be used during implementation of various tools that will be built on top of the standard software environment. See section D2 of this report.

T3 [0.5] — *Porting the common environment to workstations*
Transportation of common environment to workstations.

T4 [1.5] — *Proposal for the language definition formalisms to be used and selection of test cases*
Definition of the specification formalisms to be used and selection of a representative set of examples for comparing them. See section D4 of this report.

T5 [1.0] — *Global outline of an environment generator*
Global outline of the programming environment generator to be developed. See section D5 of this report.

T6 [0.1] — *Prepare demonstration*
Prepare demonstration of common environment.

T7 [2.5] — *Specify selected examples*
Specify the selected examples using the proposed specification formalisms.

T8 [2.2] — *Experimental software for checking examples*
Design and implementation of *experimental* software tools for constructing, consistency checking, and processing of specified examples.

T9 [1.0] — *Preliminary design of generator*
Preparatory study for the architecture of the (prototype) programming environment generator.

T10 [0.2] — *Prepare demonstration*
Prepare demonstration of the software tools for checking and processing of specified examples.

T11 [0.2] — *Assessment*
Assessment of the results obtained in the first phase of the project.

T12 [4.0] — *Integration of formalisms*
Integration of the two language specification methods.

T13 [5.0] — *Integration of tools*
Integration of the corresponding software tools.

T14 [3.8] — *Man-machine interface design*
Design and formal specification of a man-machine interface for the integrated set of language specification tools.

T15 [6.0] — *Optimization*
Design and implementation of compilation and/or optimization techniques for the two language specification techniques. Techniques currently envisaged are: transformation of equations into rewrite rules, compilation of equations or rewrite rules to Prolog or Lisp, compilation of Prolog, application of modern theorem proving techniques for deriving properties of specifications.

T16 [2.0] — *Tough example*
Exploration of the limits of the approach by constructing a small but "tough" example (i.e., an example that is difficult due to the problems involved — such as concurrency or advanced man-machine interfacing — but not due to its sheer size) requiring the use of the specification formalisms developed in the project.

T17 [1.5] — *Strategic plan*
Development of a strategic plan for the introduction of the technology developed into an industrial setting.

LITERATURE

1. Barendregt, H., The type free lambda calculus, in: Barwise, J., ed., *Handbook of Mathematical Logic*, North-Holland, 1977, pp. 1091-1132.
2. Brooker, R., Morris, D., A general translation program for phrase structure languages, *Journal of the ACM*, **9** (1962), pp. 1-10.
3. Burstall, R.M., MacQueen, D.B., Sannella, D.T., HOPE: An experimental applicative language, Internal Report, Department of Computer Science, University of Edinburgh, May 1980.
4. Despeyroux, T., Executable specification of static semantics, in: *Symposium on Semantics of Data Types*, Lecture Notes in Computer Science, Springer-Verlag, Vol. 173, 1984, pp. 215-233.
5. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B. & Lévy, J.J., A structure oriented program editor: a first step toward computer assisted programming, *International Computing Symposium*, North-Holland, 1975.
6. Donzeau-Gouge, V., Huet, G., Kahn, G. & Lang, B., Programming environments based on structured editors: the Mentor experience, INRIA Research Report, No. 26, 1980.
7. Fages, F., Formes canoniques dans les algèbres booléennes, et applications à la démonstration automatique en logique de premier ordre, Thesis, University of Paris 6, 1983.
8. Goguen, J.A., Thatcher, J.W., Wagner, E.G., An initial algebra approach to the specification, correctness, and implementation of abstract data types, in: Yeh, R., ed., *General Trends in Programming Methodology*, Prentice-Hall, 1978, pp. 80-149.
9. Meseguer, J., Goguen, J.A., Initiality, induction, and computability, in: Nivat, M., Reynolds, J., eds, *Algebraic methods in Semantics*, Cambridge University Press, 1986.
10. Goguen, J.A., Meseguer, J., Programming with parametrized abstract objects in OBJ, in: Ferrari, D. & Goguen, J.A., eds, *Theory and Practice of Software Engineering*, North-Holland, 1983, pp. 163-193.
11. Goldberg, A., Robson, D., *Smalltalk-80, The Language and its Implementation*, Addison-Wesley, 1983.

12. Guttag, J.V., Horning, J.J., Preliminary report on the Larch shared language, Report MIT/LCS/TR-307, MIT, October 1983.

13. Hoffman, C.M., O'Donnell, M.J., Programming with equations, *ACM Transactions on Programming Languages and Systems*, **4** (1982), 1, pp. 83-112.

14. Huet, G., Oppen, D.C., Equations and rewrite rules: a survey, in: Book, R., ed., *Formal Languages: Perspectives and Open Problems*, Academic Press, 1980, pp. 349-405.

15. Huet, G., Hullot, J.-M., Proofs by induction in equational theories with constructors, *Journal of Computer and System Sciences*, **25** (1982), pp. 239-266.

16. Hullot, J.-M., CEYX - a multiformalism programming environment, *Proceedings of IFIP 83*, North-Holland, 1983.

17. Hsiang, J., Topics in automated theorem proving and program generation, Report R-82-1113, University of Illinois at Urbana-Champaign, Department of Computer Science, 1982.

18. Johnson, S.C., Yacc - Yet Another Compiler-Compiler, Computer Science Technical Report No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

19. Jouannaud, J.-P., Kirchner, H., Completion of a set of rules modulo a set of equations, CSL Technical Note, SRI international, April 1984.

20. Kastens, U., Hutt, B., Zimmermann, E., GAG: a practical compiler generator, Lecture Notes in Computer Science, Vol. 141, Springer-Verlag.

21. Knuth, D.E., Semantics of context free languages, *Mathematical Systems Theory*, **2** (1968), 2, pp. 127-145.

22. R. Kowalski, *Logic for problem solving*, North-Holland, 1979.

23. Lescanne, P., Computer experiments with the REVE rewriting system generator, in: *Conference Record of the 10th ACM Symposium on Principles of Programming Languages*, ACM, 1983, pp. 99-108.

24. Lesk, M.E., Lex - a lexical analyzer generator, Computer Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

25. Special issue devoted to the GANDALF system, *The Journal of Systems and Software*, **5** (1985), 2.

26. Mosses, P., SIS-Semantics Implementation System: reference manual and user guide, Technical Report DAIMI MD-30, Computer Science Department, Aarhus University, Aarhus, Denmark, August 1979.

27. Nakajima, R., Yuasa, T., eds, *The IOTA Programming System*, Springer-Verlag, Berlin, 1983.

28. Paulson, L., A compiler generator for semantic grammars, Ph.D. Dissertation, Computer Science Department, Stanford University, December 1981.

29. Plotkin, G., A structural approach to operational semantics, DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

30. Reps, T., Teitelbaum, T., Demers, A., Incremental context-dependent analysis for language-based editors, *ACM Transactions on Programming Languages and Systems*, **5** (1983), 3, pp. 449-477.

31. Reps, T., *Generating Language Based Environments*, MIT Press, 1984.

32. Sannella, D., Wirsing, M., A kernel language for algebraic specification and implementation, in: Karpinski, M., ed., *Foundations of Computation Theory*, Lecture Notes in Computer Science, Vol. 158, Springer-Verlag, pp. 413- 427.

33. Shapiro, E.Y., *Algorithmic Program Debugging*, MIT Press, 1982.

34. Ross, D., The AED approach to generalized computer-aided design, *Communications of the ACM*, **10** (1967), pp. 367-385.

35. Teitelbaum, T., Reps, T., The Cornell program synthesizer: a syntax directed programming environment, *Communications of the ACM*, **24** (1981), 9, pp. 563-573.

36. Teitelman, W., *Interlisp Reference Manual*, Xerox, Palo Alto Research Center, 1978.

37. EEC, Study on Ada compiler validation in Europe, Final Report to the Commission, November 1982.

D1 - ESTABLISHMENT OF A COMMON ENVIRONMENT

D1.A1 - THE EFFICIENCY OF THE EQUATION INTERPRETER COMPARED WITH THE UNH PROLOG INTERPRETER

D1.A2 - A COMPARISON OF TWO WINDOW SYSTEMS

D1.A3 - SYNTAX DIRECTED EDITING OF LE_LISP

GIPE : CEC 348/A/T1/3
version 5.1 January 1986

Establishment of a Common Environment

Deliverable D1 of Task T1 — Second Review —

A. Verhoog (BSO)
G. Kahn (INRIA)

The present status of T1 is described in this document. It lists components for the Common Environment with rationales for each. The environment is constructed around the Unix operating system. Other components selected so far are: Le_Lisp, Ceyx, C-Prolog and Mentor.

This document (D1) is an intermediate report for the Second Review of January 1986 and has three annexes (D1.A1, D1.A2, D1.A3).
Task T1 is planned to finish May 1986 (2.1 My).

1. General Considerations

Task T1 consists in building a shared software environment that will be used as a point of departure for experimenting with language specification techniques.
In constructing this Common (Software) Environment the main guidelines to be followed are:

- Portability of the environment.
  That the software developed in the GIPE project be portable is a goal induced by the diverse hardware the Consortium is planning to use. The environment resulting from Task T1 will be such that compatibility between the partners exists on this "higher" level of software utilities and components, the Common Software Environment. Application programs using bitmap displays must also be portable.
- A suitable programming language.
  The programming language we are to use should have good facilities for object oriented programming and dynamically defining functions.
- Minimising the programming effort.
  Since relatively little manpower is available to develop various software packages and tools ourselves, we will use existing software as much as possible.

2. List of Components

2.1. Hardware

The Consortium does not strive for common (or even compatible) hardware in the first two years.

- Rationale: due to government policies it is not possible to select the most suitable hardware for the GIPE project freely. However, this is not felt to be a major constraint since it means that the software components of the Common Environment must be portable, which is a goal anyway. Another reason to choose one's own equipment is compatibility with hardware already present (easy support and maintenance).
- Problem areas: bitmap displays form a serious problem. Compatibility is not easily achieved for them; therefore a "virtual bitmap" is envisaged together with a solution on a higher software level (cf. **2.6**).

Current Status
Hardware in use: VAX, Bull SPS 7 (SM90), Bull SPS 9 (Ridge 32).
Workstation hardware selected by CWI-BSO is Sun2/50 and Sun3. Workstation hardware selected by INRIA-SEMA is Bull SPS 7, Bull SPS 9, Sun2 and Sun3.

2.2. UNIX

The operating system component of the Common Environment is UNIX.

- Rationale: all partners have extensive experience with Unix. It is a portable operating system supported by all major hardware manufacturers and numerous minor companies. This is an ideal base for a modern software system to be built on. General compatibility is easily achieved in spite of incompatible hardware: the software developed will have a common basis in Unix-based language implementations.
- Problem areas: Unix is not a fully standardised system yet. This may cause problems when software tools rely on certain operating system utilities. Special care will be taken to isolate system dependencies that may vary from one Unix to another. Document processing is another area of concern: both Troff and $\TeX$ are used. These are incompatible formatting systems.

Current Status
Unix versions in use by the Consortium are Unix BSD4.2 and Unix System-V.

2.3. LISP

The Lisp programming language has been chosen as the primary implementation and prototyping language (a secondary one being Prolog). From the major Lisp versions (like: Common Lisp, Maclisp, Interlisp, Franz Lisp, LE_LISP, Zetalisp) the one developed at INRIA was selected for the project; this Lisp is called LE_LISP [1].

- Rationale: Lisp is an interpretive and interactive language. Compilation facilities are available when efficiency is mandatory. An important feature needed in the concept of *generating* a programming environment is the ability to define and evaluate functions dynamically. Lisp has this facility (as opposed to languages like Pascal, C, Ada).
  LE_LISP was chosen because it was readily available to the Consortium, as it is a development of INRIA itself. This also implies easy maintainability and direct support. Moreover, at INRIA a layer on top of LE_LISP was constructed to facilitate object oriented programming. This layer is called CEYX [1] and includes various libraries one of which is a pretty-print package. An interface to the Unix YACC Compiler Compiler also exists (CXYACC [1]).
  Another important feature of LE_LISP is that it is a portable Lisp implementation. Based on a low level virtual machine language (LLM3), LE_LISP has been implemented on hardware like: VAX, Honeywell-Bull 68, Perkin Elmer 32, IBM 30xx, Sel 32, PR1ME, Norsk Data, Bull SPS 7 and Bull SPS 9, and a variety of MC68000 based workstations (including Sun2; porting to Sun3 remains to be done).
- Problem areas: experiments at CWI (parsing) and INRIA (tree construction) show that some of the facilities of CEYX (namely dynamically typed objects) impose a substantial overhead in

time and space (cf. Deliverable D2).

Current Status
At both sites (CWI-BSO and INRIA-SEMA), LE_LISP, CEYX and CXYACC have been implemented and are operational on the available hardware (cf. **2.1**) running Unix BSD4.2 and Unix System-V.

2.4. PROLOG

One aspect of generating an environment from formal specifications is to compile these specifications into executable code so that they become operational. Experiments in this area have shown that Prolog can be used as a target language for the compilation of specifications.
Several Prolog systems exist, eg.: C-Prolog, UNH-Prolog, Quintus Prolog, Prolog II. It has been decided to use C-Prolog from Edinburgh University.

- Rationale: At CWI, two target languages have been investigated that can be used for compiling specification formalisms (Annexe D1.A1). (Note that the compilation process was done by hand.) One target language is the one used by the Equation Interpreter [2]. Execution of "compiled" examples in some specification formalism with this system was somewhat slower than execution via the other target language which was Prolog. However, the Equation Interpreter does extensive preprocessing which is unacceptable since formal specifications will have to be executed many times during their development. The (interpreted!) Prolog route does not have preprocessing.
  At INRIA, experiments with the Typol language (Annexe D4.A2) are carried out and include a compiler to Prolog. The compilation process, while not completely trivial, is made easy by the organisation of Prolog programs into separate clauses that match the organisation of Typol programs into inference rules. The Prolog programs produced can be arranged in such a way that their execution is as efficient as Prolog permits, by a suitable reordering of clauses. Of course, for maximum efficiency, it would be very pleasant to make use of a Prolog compiler.

+ Requirements: it is necessary that (internal data structures of) Prolog can be used and accessed from within the Lisp language and vice-versa. Communication between LE_LISP and C-Prolog has been implemented (see Deliverable D2).

Current Status
At INRIA, extensive experiments with C-Prolog [3] have shown this system to be an extremely good candidate. It is easy to port to any Unix machine, because it is written in C with due care for machine independence. It is reasonably efficient as a Prolog interpreter. C-Prolog can be integrated into other software as a subroutine, with excellent communication in both directions (towards Prolog and from Prolog to the calling system), and even clean treatment of signal handling. Finally, it proved possible to prepare input to Prolog directly in parsed form rather than through the Prolog reader routine, resulting in an improvement in speed of problem submission by a factor of 100. This indicates that C-Prolog is a well written piece of software that has been brought under control.
Two technical difficulties may still come our way: one is the absence of an occur-check possibility. The second is the lack of some features of Prolog II (*freeze, diff*). It is not clear yet whether these features are absolutely necessary, given their cost in terms of efficiency.

2.5. MENTOR; Syntax Directed Editing

Mentor is a system for generating syntax directed editors from BNF-like specifications and was developed at INRIA [4]. It is included in the Common Environment for the purpose of prototyping.

- Rationale: Its main use is to generate syntax directed editors for the various experimental versions of specification formalisms (cf. Deliverable D4) under development in the project. It allows easy implementation of compilers to other high level formalisms such as Prolog.

Current Status
Mentor is operational at CWI and INRIA.
An initial prototype Typol environment has been constructed with Mentor. A Typol to TeX pretty-printer has been developed. The compiler from Typol to Prolog has been developed under

Mentor.
A complete LE_LISP/CEYX environment has been developed with Mentor. To achieve this an abstract syntax had to be defined for LE_LISP and CEYX. This task turned out to involve delicate engineering decisions in two areas:

- what Lisp primitives should be made into operators of the language?
- should the abstract syntax reflect the concrete layout of Lisp programs or should it take more semantic notions into account?

The technical details and the conclusions of this experiment may be found in Annexe D1.A3.

2.6. Window Management

The selection of a Window System is currently under investigation. Special attention is being paid to bitmap displays and a common set of window routines. To achieve portability, we envisage a suitable Window Interface level for GIPE applications, and a virtual bitmap device for easy implementation of the selected Window System on different hardware.
Possible candidates are: The Brown Workstation Environment [5] which includes a virtual bitmap device, the Lucasfilm package [6], and SunWindows.
See Annexe D1.A2 for a comparison of the BWE and Lucasfilm systems.

Current Status
The Brown Workstation Environment has proven to be a portable and robust system. It is available on Apollo Computers and Sun Workstations and has been ported to the SPS 7 and SPS 9. BWE consists of several layers of which only its lowest one, ASH (handling of screen primitives), seems useful for the project.

2.7. Summary

Apart from Prolog and Mentor, the overall structure of the Common Environment is shown is the following figure in which the double horizontal line indicates the boundary with application software yet to be developed.

Application		
Lisp-bitmap	LE_LISP & CEYX	
Virtual-bitmap	UNIX	LLM3
Bitmap display	processor	

3. References

[1] *Le_Lisp & Ceyx*

- J. Chailloux, "Le_Lisp de l'INRIA, Le Manuel de Reference" (version 15), INRIA Report (to be published), February 1985.
- J.M. Hullot, "Ceyx - Version 15, II: Programmer en Ceyx," INRIA Technical Report no. 45, February 1985.
- G. Berry, B. Serlet, "CXYACC et LEX-KIT version 2.1," INRIA Report, March 1984.

[2] *Equation Interpreter*

- C.M. Hoffmann, M.J. O'Donnell, "Programming with equations," ACM Transactions on Programming Languages and Systems, 4(1982) 1, 83-112.

[3] *C-Prolog*

- F. Pereira, D. Warren, D. Bowen, L. Byrd, L. Pereira, "C-Prolog User's Manual, Version 1.5," EdCAAD, Department of Architecture, University of Edinburgh, February 1984.

[4] *Mentor*

- V. Donzeau-Gouge, B. Lang, B. Melese, "Practical Applications of a Syntax Directed Program Manipulation Environment," Proceedings of the 7th Int. Conf. on Software Eng., Orlando Florida, March 1984.
- B. Melese, V. Migot, D. Verove, "The Mentor - V5 Documentation," INRIA Technical Report no. 43, January 1985.

[5] *BWE*

- J.N. Pato, S.P. Reiss, M.H. Brown, "The Brown workstation environment," Brown University CS-84-03, October 1983.

[6] *Lucasfilm*

- M.J. Hawley, "So you've got a new Sun II, *or*, Writing programs for the Sun window system — My Way," Lucasfilm Ltd, Technical Report #117, August 1984.

GIPE : CEC 348/A/T1/1
version 2.2 May 1986

The Efficiency of the Equation Interpreter Compared with the UNH Prolog Interpreter

Annexe D1.A1 of Deliverable D1 — Second Review —

J. Heering (CWI)
P. Klint (CWI)

There are several alternatives for transforming algebraic specifications into executable prototypes. In this note the Equation Interpreter (a rewrite rule interpreter) and the University of New Hampshire Prolog interpreter are viewed as target systems for executing prototypes. The efficiencies of these systems are compared with each other.

Note: The measurements reported in this paper apply to Distribution 1 (5/16/83) of the Equation Interpreter. Since then a faster version (Distribution 2, 9/9/85) has become available which we have not yet tested.

Note: This paper has been issued separately as CWI report CS-R8509. It has been published in *SIGPLAN Notices*, **21** (1986), 2, pp. 18-21.

1. Motivation

Transforming formal specifications into executable prototypes has several applications: one can either use the executable prototype to validate the specification or one may be interested in using the prototype system itself. Two alternatives for deriving executable prototypes from *algebraic* specifications are

(1) transform the specification into a complete (conditional) term rewriting system and execute it by means of an existing rewrite rule interpreter;

(2) transform the specification into a set of Horn clauses and use an existing Prolog system for their execution.

Here, we are interested in the relative efficiency of the end products which can be obtained along these two lines using the Equation Interpreter [HOD82a, HOD82b] and the UNH Prolog interpreter from the University of New Hampshire, respectively.

Two issues will *not* be addressed:

(1) The way in which an algebraic specification can be transformed into either a term rewriting system or Horn clauses.

(2) The relative merits of either the Equation Interpreter or Prolog as programming systems *per se*.

We restrict ourselves to the relative efficiency of both systems considered as (abstract) computing machines.

In the remainder of this note the measurement method and the measurements themselves are described and some conclusions are drawn. The appendices give detailed information on the programs used for the measurements.

2. Measurement method

The efficiency of the Equation Interpreter and Prolog have been compared by executing a series of examples using both systems. Each example consists of a program and input for that program. The listings of the programs, the input, and resulting output are given in the appendices. In choosing the examples we had to avoid violating implementation limitations of the systems involved. We have avoided, for instance, very long input expressions (which cause overflow of the parse stack used in the Equation Interpreter), input expressions using too many different variables (a restriction of the Prolog interpreter), or too many user defined symbols (a restriction of the Equation Interpreter). Any of these limitations could have been removed by increasing the relevant parameter in each system, but we decided not to do that and to use the standard version.

The examples are now described in more detail. The first program (EMPTY) is the empty program. It serves to measure the initialisation times for both systems.

The second program (REV) performs list reversal. It reads a list of 7 elements from input and replicates it 16 times. The resulting list of 112 elements is reversed two times and finally its length is determined. This program serves to measure the processing of large data structures.

The third program (ACK) computes Ackermann's function for the value (3,2). This program serves to measure the speed of recursion and integer arithmetic.

The fourth program (ALPHA) is actually a series of programs of increasing size. These programs define an alphabet of N characters with an equality predicate. Each program defines the Boolean functions `and` and `or`, the conditional function `if`, and the successor (`succ`) and equality (`eq_INTEGER`) functions on natural numbers. For given N, each program defines N constants (representing the characters in the alphabet), a function `ord` that injects these constants in the integers, and an equality function on characters (`eq_CHAR`) that is defined by means of `ord` and `eq_INTEGER`. The input for each program is a conditional expression containing fifteen applications of `eq_CHAR` with the fifteen last characters in the alphabet as argument; this conditional expression returns the last character in the alphabet as value. This program has as purpose to measure the effect of an increasing number of equations on the time needed for preprocessing and for execution.

Measurements have been performed on a VAX11/780 with Berkeley Unix Version 4.2. We used the first distribution of the Equation Interpreter dated 5-16-83 and version 1.3 of UNH Prolog from the University of New Hampshire.

Initial experiments showed that the timing of the Equation Interpreter presented problems due to the fact that it has been implemented as a pipeline of five concurrent processes: two preprocessors, the actual interpreter and two postprocessors. This organisation makes the timing highly sensitive to the scheduling of the individual processes in the pipeline. To avoid these fluctuations, we have replaced the pipeline by a sequence of five processes. This causes a slight increase in the execution times measured, but we observed that the execution time of the whole system is completely dominated by the execution time of the actual interpreter (this accounts for more than 95% of the total execution time).

3. Measurements

The results of the experiment are summarized in Table I. Preprocessing times have been measured 5 times. Execution times have been measured 10 times. The table gives the averages of these measurements in seconds. The standard deviation, expressed as percentage of the average of each series of measurements, never exceeded 6%.

Preprocessing times will be denoted by t and execution times by T. The total preprocessing time t_E of the Equation Interpreter includes syntactic and semantic checking of the input program, generation of an equivalent Pascal program (which includes tables for fast pattern matching of terms at execution time) and compilation of this program. This compilation time varies between 95 and 160 seconds in the above examples. T_E indicates the execution time of the Equation Interpreter.

The Prolog system does no preprocessing, i.e. $t_P = 0$. The execution times T_P given include the time needed by the Prolog system to read the example programs.

Example	Equation Interpreter		Prolog
	Preprocessing time t_E	Execution time T_E	Execution time T_P
EMPTY	136.5	2.1	0.2
REV	172.6	61.4	50.5
ACK	155.5	18.5	3.6
ALPHA (N=15)	331.9	7.8	4.3
ALPHA (N=20)	428.5	10.7	7.3
ALPHA (N=25)	528.3	13.7	9.7
ALPHA (N=30)	688.0	16.3	12.9
ALPHA (N=40)	911.2	22.1	19.7
ALPHA (N=50)	1278.4	28.5	27.5
ALPHA (N=60)	1681.1	34.1	33.3
ALPHA (N=70)	2168.4	39.8	41.7
ALPHA (N=80)	2668.1	45.5	54.4
ALPHA (N=90)	3277.0	50.8	62.7

Table I. Summary of measurements.

4. Conclusions

(1) It is surprising that a system without preprocessing performs so well as compared with a system with extensive preprocessing.

(2) The preprocessing time t_E of the Equation Interpreter tends to become prohibitive. The trends in the measurements suggest that the Equation Interpreter outperforms Prolog on large sets of equations. It depends on the particular application which system should be chosen. In the case of prototyping the same program will probably only be executed a few times. In that case, the disadvantage of considerable preprocessing time outweighs the advantage of the shorter execution time. If the number of executions is larger than $n_0 = \frac{t_E - t_P}{T_P - T_E}$ the large preprocessing time of the Equation Interpreter starts to pay off. In example ALPHA, $n_0 = 1141$, 300 and 275 for $N=70$, 80, 90, respectively.

(3) All Prolog programs in the measurements were *interpreted* and not compiled. If compilation instead of interpretation will be used one may expect a speed up of the execution time by a factor between 5 and 15.

5. References

[HOD82a] Hoffmann, C.M. & O'Donnell, M.J., "Programming with equations", *ACM Transactions on Programming Languages and Systems*, 4 (1982)1, 83-112.

[HOD82b] Hoffmann, C.M. & O'Donnell, M.J., "Pattern matching in trees", *Journal of the ACM*, **29** (1982), 68-95.

[OD77] O'Donnell, M.J., *Computing in Systems Described by Equations*, Lecture Notes in Computer Science **58**, Springer-Verlag, Berlin, 1977.

Appendix I: EMPTY

I.1 Equational program

```
Symbols
        a:0;
        noop:1.
For all x:
        noop(x) = x.
```

I.2. Input

```
a
```

I.3. Output

```
a
```

I.4. Prolog program

```
(* empty program *)
```

I.5. Input

Note: all Prolog programs are assumed to reside on the file "prodef".

```
[prodef].
```

I.6. Output

```
                    --- UNH Prolog 1.3 ---

| ?-
[ prodef consulted ]


yes
| ?-
```

Appendix II: REV

II.1 Equational program

```
Symbols
        cons:   2;
        nil:    0;
        rev:    1;
        append: 2;
        repl2:  1;
        repl4:  1;
        repl16: 1;
        length: 1;
        job:    1;
        add:    2;
        include atomic_symbols;
        include integer_numerals.

For all x, y, z, h, t, l:

        include addint;

        append(nil, x)          = cons(x, nil);
        append(cons(x, y), z)   = cons(x, append(y, z));
        rev(nil)                = nil;
        rev(cons(x, y))         = append(rev(y), x);
        repl2(nil)              = nil;
        repl2(cons(h, t))       = cons(h, cons(h, repl2(t)));
        repl4(l)                = repl2(repl2(l));
        repl16(l)               = repl4(repl4(l));
        length(nil)             = 0;
        length(cons(x, y))      = add(length(y), 1);
        job(l)                  = length(rev(rev(repl16(l)))).
```

II.2. Input

```
job(cons(a,cons(b,cons(c,cons(d, cons(e, cons(f, cons(g, nil))))))))
```

II.3. Output

```
112
```

II.4. Prolog program

```
append(nil, L, L).
append(cons(X, L1), L2, cons(X,L3)) :- append(L1, L2, L3).

rev(nil, nil).
rev(cons(H,T), L) :- rev(T,Z), append(Z, cons(H, nil), L).

repl2(nil, nil).
repl2(cons(H,T1), cons(H, cons(H, T2))) :- repl2(T1, T2).

repl4(X, Y) :- repl2(X, Z), repl2(Z, Y).
repl16(X,Y) :- repl4(X,Z), repl4(Z,Y).

len(nil,0).
len(cons(H, T), N) :- len(T, M), N is M+1.

job(L, R) :- repl16(L, X), rev(X, Y), rev(Y, Z), len(Z, R).
```

II.5. Input

```
[prodef].
job(cons(a,cons(b,cons(c,cons(d,cons(e,cons(f,cons(g,nil))))))), N).
```

II.6. Output

Note: in all following Prolog output we have removed irrelevant system messages and have only retained essential information.

```
N = 112
```

Appendix III: ACK

III.1 Equational program

```
Symbols
        add:    2;
        subtract:       2;
        equ:    2;
        if:     3;
        ack:    2;
        include atomic_symbols;
        include truth_values;
        include integer_numerals.
For all m, n:
        include addint, subint, equint;
        if(true, m, n)  = m;
        if(false, m, n) = n;
        ack(m, n)       = if(equ(m, 0), add(n,1),
                             if(equ(n, 0),ack(subtract(m, 1), 1),
                                ack(subtract(m,1), ack(m, subtract(n,1))))).
```

III.2. Input

```
ack(3,2)
```

III.3. Output

```
29
```

III.4. Prolog program

```
ack(0, N, R) :- R is N+1.
ack(M, 0, R) :- M1 is M-1, ack(M1, 1, R).
ack(M, N, R) :- M1 is M-1, N1 is  N-1, ack(M, N1, R1), ack(M1, R1, R).
```

III.5. Input

```
[prodef].
ack(3, 2, R).
```

III.6. Output

```
R = 29
```

Appendix IV: ALPHA

Note: we only show the ALPHA example for the case N = 15.

IV.1 Equational program

```
Symbols
        char_0: 0;
        char_1: 0;
        char_2: 0;
        char_3: 0;
        char_4: 0;
        char_5: 0;
        char_6: 0;
        char_7: 0;
        char_8: 0;
        char_9: 0;
        char_10: 0;
        char_11: 0;
        char_12: 0;
        char_13: 0;
        char_14: 0;
        char_15: 0;
        eq_INTEGER: 2;
        eq_CHAR: 2;
        TRUE: 0;
        FALSE: 0;
        AND: 2;
        IF: 3;
        succ: 1;
        ord: 1;
        include integer_numerals;
        include atomic_symbols.
For all x, y, c1, c2:
AND(TRUE, TRUE) = TRUE;
AND(TRUE, FALSE) = FALSE;
AND(FALSE, TRUE) = FALSE;
AND(FALSE, FALSE) = FALSE;
IF(TRUE, x, y) = x;
IF(FALSE, x, y) = y;
eq_INTEGER(0, 0)        = TRUE;
eq_INTEGER(succ(x), succ(y)) = eq_INTEGER(x, y);
eq_INTEGER(0, succ(x)) = FALSE;
eq_INTEGER(succ(x), 0) = FALSE;
eq_CHAR(c1, c2) = eq_INTEGER(ord(c1), ord(c2));
ord(char_0) = 0;
ord(char_1) = succ(succ(succ(succ(succ(ord(char_0))))));
ord(char_2) = succ(succ(succ(succ(succ(ord(char_1))))));
ord(char_3) = succ(succ(succ(succ(succ(ord(char_2))))));
ord(char_4) = succ(succ(succ(succ(succ(ord(char_3))))));
ord(char_5) = succ(succ(succ(succ(succ(ord(char_4))))));
ord(char_6) = succ(succ(succ(succ(succ(ord(char_5))))));
ord(char_7) = succ(succ(succ(succ(succ(ord(char_6))))));
ord(char_8) = succ(succ(succ(succ(succ(ord(char_7))))));
ord(char_9) = succ(succ(succ(succ(succ(ord(char_8))))));
ord(char_10) = succ(succ(succ(succ(succ(ord(char_9))))));
ord(char_11) = succ(succ(succ(succ(succ(ord(char_10))))));
ord(char_12) = succ(succ(succ(succ(succ(ord(char_11))))));
ord(char_13) = succ(succ(succ(succ(succ(ord(char_12))))));
ord(char_14) = succ(succ(succ(succ(succ(ord(char_13))))));
ord(char_15) = succ(succ(succ(succ(succ(ord(char_14)))))).
```

IV.2. Input

```
IF(AND(eq_CHAR(char_0, char_0),
AND(eq_CHAR(char_1, char_1),
AND(eq_CHAR(char_2, char_2),
AND(eq_CHAR(char_3, char_3),
AND(eq_CHAR(char_4, char_4),
AND(eq_CHAR(char_5, char_5),
AND(eq_CHAR(char_6, char_6),
AND(eq_CHAR(char_7, char_7),
AND(eq_CHAR(char_8, char_8),
AND(eq_CHAR(char_9, char_9),
AND(eq_CHAR(char_10, char_10),
AND(eq_CHAR(char_11, char_11),
AND(eq_CHAR(char_12, char_12),
AND(eq_CHAR(char_13, char_13),
AND(eq_CHAR(char_14, char_14),
    eq_CHAR(char_15, char_15))))))))))))))))),
char_15, FALSE)
```

IV.3. Output

```
char_15
```

IV.4. Prolog program

```
and(true, true, true).
and(true, false, false).
and(false, true, false).
and(false, false, false).
if(true, X, Y, X).
if(false, X, Y, Y).
eq_INTEGER(0, 0, true).
eq_INTEGER(succ(X), succ(Y), R) :- eq_INTEGER(X, Y, R).
eq_INTEGER(0, succ(X), false).
eq_INTEGER(succ(X), 0, false).
eq_CHAR(C1, C2, R) :- ord(C1, N1), ord(C2, N2), eq_INTEGER(N1, N2, R).
ord(char_0, 0).
ord(char_1, succ(succ(succ(succ(succ(R)))))) :- ord(char_0, R).
ord(char_2, succ(succ(succ(succ(succ(R)))))) :- ord(char_1, R).
ord(char_3, succ(succ(succ(succ(succ(R)))))) :- ord(char_2, R).
ord(char_4, succ(succ(succ(succ(succ(R)))))) :- ord(char_3, R).
ord(char_5, succ(succ(succ(succ(succ(R)))))) :- ord(char_4, R).
ord(char_6, succ(succ(succ(succ(succ(R)))))) :- ord(char_5, R).
ord(char_7, succ(succ(succ(succ(succ(R)))))) :- ord(char_6, R).
ord(char_8, succ(succ(succ(succ(succ(R)))))) :- ord(char_7, R).
ord(char_9, succ(succ(succ(succ(succ(R)))))) :- ord(char_8, R).
ord(char_10, succ(succ(succ(succ(succ(R)))))) :- ord(char_9, R).
ord(char_11, succ(succ(succ(succ(succ(R)))))) :- ord(char_10, R).
ord(char_12, succ(succ(succ(succ(succ(R)))))) :- ord(char_11, R).
ord(char_13, succ(succ(succ(succ(succ(R)))))) :- ord(char_12, R).
ord(char_14, succ(succ(succ(succ(succ(R)))))) :- ord(char_13, R).
ord(char_15, succ(succ(succ(succ(succ(R)))))) :- ord(char_14, R).
job(T) :- eq_CHAR(char_0, char_0, R0),
and(R0, R0, T0),
eq_CHAR(char_1, char_1, R1),
and(T0, R1, T1),
eq_CHAR(char_2, char_2, R2),
and(T1, R2, T2),
eq_CHAR(char_3, char_3, R3),
and(T2, R3, T3),
eq_CHAR(char_4, char_4, R4),
and(T3, R4, T4),
eq_CHAR(char_5, char_5, R5),
and(T4, R5, T5),
eq_CHAR(char_6, char_6, R6),
and(T5, R6, T6),
eq_CHAR(char_7, char_7, R7),
and(T6, R7, T7),
eq_CHAR(char_8, char_8, R8),
and(T7, R8, T8),
eq_CHAR(char_9, char_9, R9),
and(T8, R9, T9),
eq_CHAR(char_10, char_10, R10),
and(T9, R10, T10),
eq_CHAR(char_11, char_11, R11),
and(T10, R11, T11),
eq_CHAR(char_12, char_12, R12),
and(T11, R12, T12),
eq_CHAR(char_13, char_13, R13),
and(T12, R13, T13),
eq_CHAR(char_14, char_14, R14),
and(T13, R14, T14),
eq_CHAR(char_15, char_15, R15),
and(T14, R15, T15),
if(T15, char_15, false, T).
```

IV.5. Input

```
[prodef].
job(T).
```

IV.6. Output

```
T = char_15
```

GIPE : CEC 348/A/T1/6
version 1.2 03-02-86

A Comparison of Two Window Systems

Deliverable D1.A2 of Task T1 — Second Review —

A. Verhoog (BSO)

Window systems are becoming standard components of applications with high-level user interfaces. At the same time, these window systems form a serious obstacle in terms of availability, portability, efficiency and the architecture of the procedure call interface. Two of the many window packages currently in existence have been investigated for suitability within the GIPE project. A major conclusion of this investigation is that a "higher" level window interface should be defined. To achieve this Task T1 will be extended by 3 man months.

1. Introduction

This Annexe describes two window systems which are candidates for use within the GIPE project. From the many systems currently available (BWE, Lucasfilm, the User Interface of PCTE, which is being developed under Esprit Project 32, WM, X-System, SunWindows, ...), only the first two have been investigated. *BWE* is the Brown Workstation Environment from Brown University, Providence, Rhode Island - USA. It consists of several packages such as ASH (A Screen Handler) and MAPLE (a menu package). *Lucasfilm* comes from the Computer Division of Lucasfilm Ltd., San Rafael, California - USA and was specifically designed for use with the window package SunWindows from Sun Microsystems Inc., Mountain View, California - USA. These two window systems were readily available to the Consortium and both look promising candidates. They are of a rather different nature, however. Other systems have not been looked at in-depth.

The different natures of BWE (or in fact *ASH*, being the part of BWE we are actually dealing with) and Lucasfilm can be summarised as follows: ASH is a relatively *low* level, but very complete, set of routines, while Lucasfilm is of a relatively *high* level, but has some shortcomings. The fact that neither system suffices as a window system interface on the application level, has led to the conclusion that a *higher level* interface must be defined. This "Window Interface for GIPE" *WIG*, will resemble the set of functions in Lucasfilm, but will also include various indispensable ASH aspects. WIG will/should enable the porting of applications to hardware supporting other window systems, like the Apple Macintosh.
In the Appendices an example is presented in both the BWE and Lucasfilm setting. It also includes complete lists of the routines present in the two systems, each with a short description.

2. BWE/ASH in a nutshell

BWE is an integrated toolkit of portable software components allowing applications to use bitmap-based workstations.

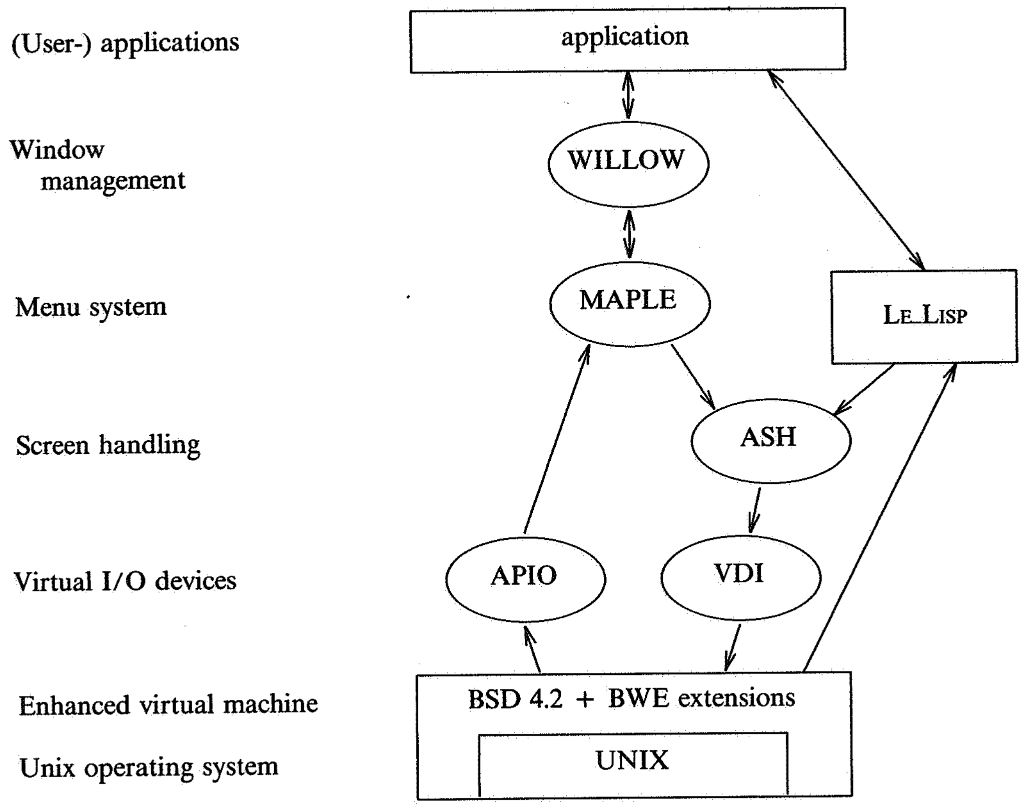
The tools, in the form of libraries of subroutines, are such that application programs can effectively use the available hardware with a minimum of effort.

The facilities offered include:
- menu-based graphical input (keyboard and locators)
- sophisticated graphical output (multiple, overlapping windows)
- text and graphics editing

- user-level window management.

The design and implementation of the toolkit emphasizes *Extensibility* (a layered structure of the components), *Flexibility* (tailoring the components to specific application needs) and *Portability* (C coding for UNIX environments; bitmap hardware dependencies isolated in virtual device interfaces).

2.1. Overview of BWE



BWE packages & LE_LISP

The BWE components are:

- Enhanced Virtual Machine:
  Virtual device drivers, offering the minimum functionality necessary for BWE; facilities manufacturers may offer are taken into account; adaptions to UNIX to make it compatible with BSD 4.2 (as far as needed).

- VDI - Virtual Device Interface:
  A device independent *output* device driver for basic graphical operations (display and bitmap). VDI is based on the emerging ANSI-Standard Virtual Device Metafile and supports fonts, pixels, strings, disk I/O, points, lines, (filled) polygons, multiple colours and clipping.

- APIO - Apollo Input Only Package:
  A low level *input* driver (to be replaced by a general I/O VDI) providing input from keyboard and locator devices. The acquired input is passed on to the rest of BWE as a stream of events.
  (Operations: keyboard input, keystroke mapping, locator (*mouse*) position and buttons, locator

simulation possible)

- ASH - A Screen Handler:
  A low level machine-independent window manager. Handles displaying text and graphics on 'logical screens' (bitmaps, of which several classes are supported) in full, in part (i.e. a partial view of the logical screen (= *window*) or not at all).
  (Operations: window hierarchies & stacks, colour, integer coordinates, multiple views of same window, user or ASH refresh, window moving/push&pop/finding/sensing)

- MAPLE - A Menu Package:
  A high level sophisticated user interface for interactive graphics programs. The (user) application is run by menu manipulation. Various types of menus are offered including buttons (text or icons) which are highlighted upon selection and have (application) action routines associated with them. Input is via ASH, output via VDI, so MAPLE is a device independent menu package.

- WILLOW - Wonderful Integrated Language for Laying-out Windows:
  A user interface for managing windows within a system. It has three built-in window manipulation methods, others can readily be defined. To the user (and ASH) WILLOW is analogous to the Shell in UNIX. Communication with ASH is via 'messages' (this is also directly usable from the application level).
  WILLOW can define cq. knows about: window name, window size (minimal, maximal, optimal), foreground/background colour, fill pattern, compress into/expand from icon, association of windows and buttons with application routines, hardcopy, save/set-up files, history with undo possibility.

2.2. Only ASH

Experiments with BWE soon revealed that the MAPLE package is probably too sophisticated and also rather big and inefficient. The ASH level, however, has turned out to be a reasonable set of "screen handling routines", of which 75% has been interfaced with LE_LISP. WILLOW has not really been investigated, as it was/is too unstable, apart from its expected large size and inefficiency.

3. Lucasfilm in a nutshell

In essence, *Lucasfilm* provides (high level) *bitmap* graphics in a window supplied by SunWindows. M.J. Hawley of Lucasfilm Ltd. designed and made it because using SunWindows directly proved inadequate for the applications he had in mind. His work owes much to the work of R. Pike for the Blit terminal (Olivetti DMD 5620) and L. Cardelli (among others), both from Bell Laboratories.

Since SunTools (which is Sun's own window manager on top of SunWindows) was not satisfactory either, a Lucasfilm *Suntools* version was developed as well. The routines we are dealing with are in the library *-lsun*. They can be subdivided into various classes, such as mouse routines, menu routines, keyboard routines etc. (cf. Appendix 3). The library *-lsuntools* implements the Suntools environment.

Apart from the window libraries, Lucasfilm also has a collection of useful *tools*, such as a bitmap editor, a browse program to display bitmaps, a font editor and various programs dealing with *mail* and *news* in a graphical way. Moreover, a large amount of bitmaps are available such as many (mouse) cursors, icons, textures and just pictures.

3.1. Design aspects of Lucasfilm

By defining appropriate structures like *Point*, *Rectangle*, *Bitmap* and new user functions, e.g. for menus and input processing, Lucasfilm shields the programmer from SunWindows.

A Rectangle for instance is identified by two points, the upper left-hand corner (*origin*), and lower right-hand corner (*corner*). The Points themselves consist of an x and y component, which are coordinates relative to the window's origin.

A Bitmap structure contains a rectangle and a pointer to another structure (a *pixrect*) containing information of the underlying SunWindows world. The actual bitmap bits (*pixels*) are hidden deeply in this pixrect structure but can be manipulated.

A typical Lucasfilm application looks as follows:

```
#include <sun.h>

main(possible arguments) {
        InitDisplay();
        InitDevices(devices);
        . . . -- This is where the application initializes ...
        Go();
}

Input() {
        int i = Poll(devices);

        if (i&KBD) {
                -- application reacts to keyboard typing
        }
        if (i&MOUSE) {
                -- application reacts to mouse movements or button clicks
        }

        -- other input handling (user defined devices)
}

Timeout() { ... }
```

In the main routine, the display bitmap is initialized (it is the *window* in which the program runs) and the necessary devices are set up. Then a "start-up" follows where the bitmap of the application is filled with, say, sticky menus or other graphics, and initialisations are done for user input or timer run-outs. Finally, the whole display is actually put onto the screen as a self-contained (Suntools) window from which point the application is "driven" by the Input and Timeout routines. In the Input routine, the devices set up for user input are polled and appropriate actions taken, e.g. to handle menu input when some mouse button is depressed *and* the mouse happens to be in the menu. One does not, however, code the menu behaviour oneself, merely a menu function is called which returns the selected menu item.

4. ASH vs. Lucasfilm

The two window systems *ASH* and *Lucasfilm* differ in many respects which essentially can be summarised as:

- With ASH:	one can do (almost) anything, but often with a major programming effort.
- With Lucasfilm:	one can do many things (but not *all* we want) nicely, with a minor programming effort.

In the following sections the important differences are further elaborated.

4.1. Managerial aspects

Sources of both systems are available, so bugs encountered can in principle be repaired and necessary adaptations be made.

Also, of both systems documentation exists, but the ASH manual is not particularly clear; often small experiments have to be done to grasp the exact meaning of a routine's functioning (in fact, this applies more or less to all BWE manuals). The Lucasfilm documentation is good except for small shortcomings which can easily be resolved by inspection of the sources, which are more comprehensible than those of ASH.

We have to do our own maintenance since it cannot be expected from Brown University, nor from Lucasfilm: both systems come "as is." However, at CWI, a relatively good contact exists with Lucasfilm, so that we may expect to receive occasional updates.

As for portability of a (window package) interface, one can look at two levels: Portability on the level that defines the package interface, the *applications* level and portability on the (lower) level that *implements* the package (on existing software/hardware). We are concerned with both these portability levels. The applications level is normally referred to as "the interface" of the package, in this case a windows package. A higher interface level (Lucasfilm approach) gives a better application portability (i.e. portability to *another* windows system), while a lower interface level (ASH) reduces this portability, since the low-level functions are too specific and unlikely to have counterparts in another package.
Naturally, portability on the *implementation* level is an even more desired property. A port of the windows system implies far less conversion of the applications. In this respect ASH is a winner. However, such a port may introduce inefficiencies when routines of the implementation level are "hooked to" the target on too high a level (as is the case with the ASH implementation on the Sun). Lower level interface packages are generally more portable to other targets than a higher level interface. Lucasfilm is too intertwined with Sun software so that its implementation level is vague. Although porting requires a larger effort it does not lead to loss of efficiency, since the target system's facilities may be fully utilized.

Indeed, ASH has been ported to different bitmap hardware without too much effort. Implementations are now available on Sun2, SPS7 or SM90, SPS9 or Ridge (with Numelec bitmap displays). Outside GIPE, work is in progress on a Macintosh implementation of ASH.

The aspects described above are summarised in a table in which the signs have the following meanings:

+ :	good (or present/fulfilled)
~ :	questionable, or possible with effort
− :	not so good (or absent).

Managerial Aspects		
	ASH	Lucas-film
Availability	source	source
Documentation	~	+
Maintenance	own	own
Porting of window system	+	~
Efficiency of ported package	~	~/+
Portability of applications	−	+

4.2. Architecture aspects

From the programmer's point of view, developing an application for Lucasfilm is straightforward for the following reasons:

- A good tutorial, with instructive examples.
- Classification of the routines in classes (general window management, menus, mouse, text I/O, graphics, "points", "rectangles", miscellaneous).
- The routines form an intelligible, rather *small* interface with clear names and arguments.
- The intricacies of window management are shielded from the programmer.

The following trivial program puts a window on the screen and exits when a mouse button is pressed.

```
#include <sun.h>

main() {
        InitDisplay();
        InitDevices(MOUSE);
        Go();
}

Input() { WaitButtons(); exit(); }
```

This clearly shows the high level approach of Lucasfilm. A program in ASH with the same functionality looks less attractive.

```
#include "ash.h"
#include "apio.h"

main() {
        APIO_EVENT event;
        int x, y, butns; char ch;

        ASHinit(ASH_MODE_WINDOW);
        ASHcursor(1);                /* enable mouse cursor */
        for (;;) {
          APIOget (&event, &x, &y, &butns, &ch);
          if (event == APIO_EVENT_TPAD && (butns&(1|2|4))) exit();
        }
}
```

Strictly speaking, ASH deals only with *output*. A routine from another BWE package (*APIO*) has to be called to handle the mouse (and keyboard) *input**.
It may be seen from the above example that using ASH (and APIO for input events) is not quite straightforward. A few reasons are:

- No tutorial documentation, no examples.
- ASH is a single *big* collection of routines, which is difficult to subdivide into classes (which indeed is not done).
- Often the routines require parameters of an unclear nature (or there are just "a lot of them")
- Programs soon become complicated because more routines/statements are needed, generally speaking.

See Appendix 1 for another example. Note that more pronounced differences appear when dealing with, say, menus. Even when using MAPLE (the BWE menu package) for such an application with ASH, the benefits of a Lucasfilm solution are obvious, let alone when menu facilities were programmed with ASH as part of the program.

In the following table, the differences between ASH and Lucasfilm are summarized:

* In ASH-Le_Lisp input is done using new C routines that may be called from Le_Lisp.

Architectural Aspects		
	ASH	Lucasfilm
As a window package ●	"big", rather complex	"small" and clean
●	needs other BWE packages (APIO, MAPLE) (or extra user programming)	one integrated package
Number of functions	140 (*all* of ASH, *with* APIO; MAPLE another 110)	115 (*incl.* menus)
Sources: #lines (C code + include files)	10900 (incl. 18% APIO)	5800
#chars	237k	136k
Application program size (bytes) (cf. App.1)	~300k	~150k
Defaults handling	many implicit defaults (=> program behaviour not so obvious)	explicit attribute parameters (=> code is more clear)

5. WIG: Window Interface for GIPE — rationale

With ASH, we have done quite a few experiments, both directly in C and with the LE_LISP interface of ASH (see Deliverable D2.A3). One application, for instance, is a TEX viewing program. It enables the user to preview formatted TEX output on a bitmap screen. Output of up to six TEX pages are kept in separate invisible (ASH) windows. Per window only a *part*, an ASH *view*, is actually displayed. At the bottom and right sides of such a view are so-called elevators, small horizontal or vertical bars, which indicate the part of the window on the screen. These elevators can be moved by mouse operations which results in "viewing" another part of the TEX page.

The above described viewing mechanism with elevators is a typical example of a higher level function that we like to have in a windows interface library. Other functions that should be available "off the shelf" include: menu routines of fixed appearance, as well as of user defined form and operation, input routines to enter text or various kinds of graphics (lines, circles, boxes, ellipses, ...). ASH does not have any of these functions. However, one can *draw* graphics, but not *interactively*, while Lucasfilm supplies four kinds of menus, although not user definable, and has a text input routine. It may become clear that with ASH all such high level window routines could be implemented (as indeed is done in the MAPLE package of BWE), but are not presently available. Lucasfilm goes a long way towards a *WIG*, but has a few serious deficiencies:

- It is built on Sun software, so porting is not easy.
- There is no provision for (hierarchic/independent) multiple "windows".
- No multiple *views* per window.

To overcome the problems and deficiencies/discrepancies of ASH and Lucasfilm, a definition of a higher level windows system package is needed. In the sequel of Task T1, we will develop a suitable set of window routines that will serve as an interface between GIPE applications and the window system (-primitives) used. Note that "the window system" can be ASH itself, or SunWindows (as with Lucasfilm), or yet another window system, like the one available on the Macintosh. The set of routines suggested above is indeed an *interface*, implementing a layer that shields the application from the window system used. In this way, we achieve a high level of portability which is, especially for bitmap applications, essential for further successful software development in the project.

The table below lists the main requirements for WIG. The relative merits of these requirements are judged for ASH and Lucasfilm, with meanings as described in section 4.1.

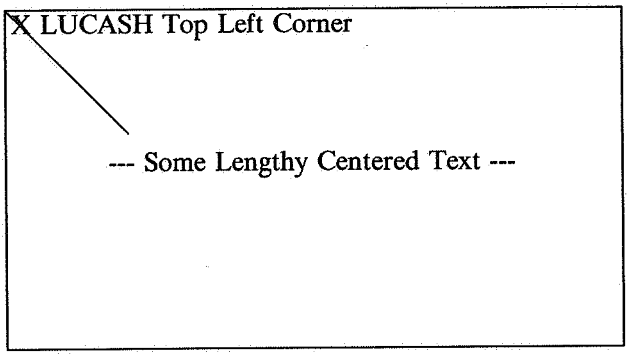
WIG Requirements		
	ASH	Lucas-film
Hierarchical and/or multiple—		
windows	+	−
menus	−	+
Multiple views	+	−
Graphics		
elementary	+	+
interactive	−	~/+
Fonts *	~	~
Efficient interface with LE_LISP	+	−
Portability of		
application	−	+
window system	+	~

* Both ASH and Lucasfilm have several fonts included (and are incompatible), but these have not been experimented with. It seems that those of Lucasfilm can be used without trouble.

Appendix 1 — an example

A simple example
(without menus)

In both the BWE/ASH and Lucasfilm environments, a simple example is shown which makes the following picture in a "stolen"* window (ASH), or in a newly created window (Lucasfilm).



The examples.

=== ASH example ===

```
/* cc -I/pro/include vb1.c /pro/lib/prolib.a */
#include "ash.h"
#include "apio.h"

#define button123()     (butns&(1|2|4)?1:0)
int butns;

main () {
      ASH_WINDOW w;
      APIO_EVENT event;
      int lx, by, rx, ty, x, y; char ch, s[256];

      w = ASHinit(ASH_MODE_WINDOW);

      ASHcursor(1); /* enable mouse cursor */
      ASHline (0,0,100,100); /* ie. where is Window's origin */
      ASHtext (0,16,"X ASH Top Left Corner");  /* 0,0: inappropriate... */
      ASHinq_size (ASH_SIZE_WINDOW, &lx, &by, &rx, &ty);
      strcpy(s,"--- Some Lengthy Centered Text ---");
      ASHinq_text_extent(s, &x, &y);
      ASHtext ((rx-lx-x)/2, (by-ty)/2, s);
      for (;;) {
        APIOget (&event, &x, &y, &butns, &ch);
        if (event == APIO_EVENT_TPAD)
          if (button123()) exit();
      }
}
```

* *Stolen* means: the ASH application runs in the window area on the screen in which the program happens to have been invoked (rather than creating a *new* window, like Lucasfilm does).

= = = Lucasfilm example = = =

```
/* cc -I/usr/sun/include vb1.c -lsun -lsuntool -lsunwindow -lpixrect */
#include <sun.h>

main() {
     char *s, *StrSave();
     Point cp;

     InitDisplay();
     InitDevices(MOUSE);
     segment (Display, Dr.o, Pt(100,100), F_STORE);
     string(Display, Dr.o, "X LUCAS Top Left Corner.", defont, F_STORE);
     s=StrSave("--- Some Lengthy Centered Text ---");
     cp = Center(Dr); /* "cp" necessary... */
     string(Display,
       Pt( (HS(Dr)-StrHS(defont,s))/2, cp.y ),
       s,
       defont,
       F_STORE);
     Go();
}

Input() { if (Poll(MOUSE) && button123()) exit(); }
```

Appendix 2 — list of ASH routines

ASH routines
as present in the LE_LISP interface

The 92 listed ASH routines comprise the ones defined for the LE_LISP interface. The calling sequences are in C (as defined in BWE).

1 **ASHinit** `(mode[,stream])`

Initialize ASH (only necessary for some special mode).

2 **ASHtrace** `(level)`

Enable various levels of tracing.

3 **ASHpush_state** `()`

Save current window with drawing information.

4 **ASHpop_state** `()`

Restore a window with drawing information.

5 **ASHcreate** `(parent_lx,parent_by,lx,by,rx,ty,border_id,flgs)   -- ASH_WINDOW`

Create a window of given size within the current (parent) window. Various attributes may be specified (eg. sensitivity, visibility).

6 **ASHselect** `(window)`

Make the window current.

7 **ASHvisible** `(flag)`

Enable/disable the current window's visibility.

8 **ASHpop** `()`

Pop current window to top of the screen.

9 **ASHpush** `()`

Push current window to bottom of the screen.

10 **ASHuncover** `(x,y)`

Push top window at given point to bottom.

11 **ASHfind_window** `(x,y)   -- ASH_WINDOW`

Return window on top at (x,y).

12 **ASHinq_under** `(window,x,y) -- ASH_WINDOW`

Find window directly under given window at point (x,y).

13 **ASHinq_rectangle** `(x,y,&lx,&by,&rx,&ty) -- ASH_WINDOW`

Return coordinates of a completely visible rectangle containing point (x,y). The corresponding window is also returned.

14 **ASHinq_region_visible** `(window,lx,by,rx,ty) -- int`

Determine if a given region is completely visible in the window.

15 **ASHhitable** `(hittype)`

Set hittability of current window.

16 **ASHhit** `(x,y,&window_x,&window_y) -- ASH_WINDOW`

A hittable top window at (x,y) is returned (or NULL). Hit coordinates are returned as well, if applicable.

17 **ASHhitwindow** `(window)`

Mark window as the current hit window.

18 **ASHmap** `(from_window,from_x,from_y,to_window,&to_x,&to_y) -- int`

Map a point in the from window to coordinates in the to window (if possible) and return the new coordinates via the last two parameters.

19 **ASHremove** `(window)`

Remove the window.

21 **ASHview** `(parent_lx,parent_by,lx,by,rx,ty)`

Reset the current window's view and locate it in the parent at the specified point.

22 **ASHnewview** `(window,p_lx,p_by,lx,by,rx,ty,b_id,flgs) -- ASH_WINDOW`

Create another view of window, yielding a window akin to ASHcreate.

23 **ASHnewframe** `()`

Refresh the screen.

24 **ASHresize** `(lx,by,rx,ty)`

Resize the current window, according to the given coordinates.

25 **ASHpar_resize** `(plx,pby,lx,by,rx,ty)`

Idem as ASHresize, but the bitmap in the parent window is relocated as well.

26 **ASHquickmove** `(flag)`

Set/reset a mode specially for dragging windows on the screen.

27 **ASHinq_size** `(type,&lx,&by,&rx,&ty)`

Inquire the size of various box types associated with the current window.

28 **ASHinq_top** `()  -- ASH_WINDOW`

Return window for actual screen.

29 **ASHinq_window** `()  -- ASH_WINDOW`

Return current window.

30 **ASHinq_parent** `()  -- ASH_WINDOW`

Return window of current window's parent.

33 **ASHinq_border_size** `(id,&left,&bottom,&right,&top)`

Return border size of specified border id.

34 **ASHset_window_name** `(name)`

Put name in the current window's border.

35 **ASHinq_window_name** `()  -- char *`

Get the border name.

36 **ASHset_userdata** `(data)`

Associate a (pointer) value with a window.

37 **ASHinq_userdata** `()  -- char *`

Read back the (pointer) value.

45 **ASHpush_window** `()`

Save selected window in a stack.

46 **ASHpop_window** `()`

Restore a window.

54 **ASHline** `(x0,y0,x1,y1)`

Draw a line in current window, with current combination rule.

55 **ASHpolyline** `(numpts,x[],y[])`

Draw sequence of connected lines.

56 **ASHpoint** `(x,y)`

Draw a point in current window.

57 **ASHpolypoint** `(numpts,x[],y[])`

Draw a collection of points.

59 **ASHconvex_polygon** `(numpts,x[],y[])`

Draw and fill a (convex) polygon.

60 **ASHgeneral_polygon** `(numpts,x[],y[])`

Draw and fill a general polygon.

61 **ASHrectangle** `(x0,y0,x1,y1)`

Draw a filled box.

62 **ASHround_rectangle** `(x0,y0,x1,y1,radius)`

Draw a filled box with rounded corners.

63 **ASHbox** `(x0,y0,x1,y1)`

Draw a rectangular box outline.

64 **ASHround_box** `(x0,y0,x1,y1,radius)`

Draw a rectangular box outline with rounded corners.

65 **ASHcircle** `(x,y,r)`

Draw a circle outline with center at (x,y) and radius r.

66 **ASHfilled_circle** `(x,y,r)`

Draw a filled circle.

67 **ASHellipse** `(x,y,rx,ry)`

Draw an ellipse outline at (x,y) with radii rx and ry.

68 **ASHfilled_ellipse** `(x,y,rx,ry)`

Draw a filled ellipse.

69 **ASHclear** `()`

Clear current window.

70 **ASHtext** `(x,y,text)`

Display text at given point.

71 **ASHcenter_text** `(text,lx,by,rx,ty)`

Center the text inside the given rectangle.

74 **ASHfill** `(pattern_id)`

Set active fill pattern for current window.

75 **ASHline_style** `(pattern_id)`

Set active line style.

76 **ASHcombination_rule** `(rule)`

Set combination rule.

77 **ASHfont** `(font)`

Set font.

81 **ASHclip** `(flag)`

Enable/disable clipping.

82 **ASHclip_region** `(lx,by,rx,ty)`

Define clipping region.

83 **ASHinq_blt** `(lx,by,rx,ty,dlx,dby)   -- int`

Determine whether the blt operation can be done successfully.

84 **ASHblt** `(lx,by,rx,ty,dlx,dby)`

Perform a *raster-op* of the given rectangle in the current window to the destination window given by the last two parameters.

85 **ASHzoom_blt** `(lx,by,rx,ty,dlx,dby,zoomx,zoomy,gapx,gapy)`

Like ASHblt, but the image is enlarged during the operation.

86 **ASHread_pixels** `(lx,by,rx,ty,pixels)`

Read back saved pixels from the array into the bitmap rectangle.

87 **ASHwrite_pixels** `(lx,by,rx,ty,pixels)`

Save the bitmap pixels in an array.

88 **ASHsave_bitmap** `(file)`

Save the current window's bitmap in a portable file format (*BRIM*).

91 **ASHload_bitmap** `(file)`

Restore a bitmap from a file.

92 **ASHsource** `(window)`

Set source window for current window (for use with ASHblt).

95 **ASHloadfont** `(name)   -- int`

Load a font by it's name; a font id (int) is returned.

97 **ASHinq_text_extent** `(str,&x,&y)`

Return "x" and "y" size of the text in str.

98 **ASHinq_text_offset** `(str,&x,&y)`

Return lower left box offsets for use with ASHtext.

99 **ASHinq_text_next** `(str,&x,&y)`

Return increments for placing str next to (or under) a previously written text.

100 **ASHinq_font_info** `(name,hor_sp,vert_sp,space_sz,width,down,up)   -- int`

Return data on the given font (for the whole font, and for each character in it).

101 **ASHinq_font** `()   -- int`

Return current font in current window.

104 **ASHinq_fill** `()` `-- int`

Return current fill pattern.

105 **ASHinq_line_style** `()` `-- int`

Return current line style.

106 **ASHinq_combination_rule** `()` `-- int`

Return current combination rule.

109 **ASHpush_drawinfo** `()`

Save all drawing parameters of current window; reset them to default values.

110 **ASHpop_drawinfo** `()`

Restore previously saved drawing parameters.

111 **ASHcopy_drawinfo** `(source)`

Copy the drawing information from given window to current one.

112 **ASHcursor_move** `(x,y)`

Move cursor to given point in current window.

113 **ASHcursor** `(flag)`

Turn the mouse cursor on or off.

114 **ASHcursor_load** `(id)`

Load a cursor identified by id.

115 **ASHcursor_restore** `()`

Get original cursor again.

116 **ASHinq_cursor** `(&id)` `-- int`

Inquire whether cursor is being displayed; optionally return its id type.

117 **ASHpush_cursor** `(fg)`

Save cursor's type and mode, possibly enabling/disabling it afterwards.

118 **ASHpop_cursor** `()`

Restore a saved cursor.

119 **ASHcursor_define** `(id,ch,font,x,y,xor_flag)`

Dynamically define a new cursor with its hot spot.

120 **ASHsensitive_area** `(lx,by,rx,ty,type)` `-- ASH_SENSE`

Define and return a reference to a sensitive rectangle, possibly inverting bits when hit with ASHhit.

121 **ASHsensitive_remove** `(sense)`

Remove a sensitive area.

122 **ASHsensitive_remove_all ( )**

Remove all sensitive areas of the current window.

123 **ASHinq_sensitive ( )** `-- ASH_SENSE`

Return the sensitive area directly under the coordinates of the last ASHhit call.

124 **ASHbell ( )**

Ring a bell.

Appendix 3 — list of Lucasfilm routines

Lucasfilm window routines

This is an extensive overview of the Lucasfilm library *"-lsun"* which provides a clean set of routines for writing interactive graphics programs on the Sun II workstation.

For the *suntools* environment: (InitDisplay(3))
High level routines for window management, device initialisation and polling, timing.

InitDisplay `()`

Obligatory routine to initialize the Display bitmap, which becomes a window on the screen.

InitDevices `(devices)`

Initialize the used input devices.

`int` **DefineDevice** `(fd, func)`

Define input file fd as an input device, for which func will be called whenever there is input available. The returned device should be polled.

UndefineDevice `(fd)`

Undefine a device fd.

`int` **Poll** `(devices)`

Return a mask of those devices that have changed their state.

Go `()`

Actually start displaying the Display on the screen. Arrange for timeout and input routines.

SetInput `(func)`

Call func instead of the default Input().

SetTimeout `(func, seconds, microseconds)`

Call func when timer runs out.

SetExit `(func)`

Call func when program exits normally.

`int` **Iconic** `()`

True if in an *iconic* state.

SetIcon (b, s)

Define an icon b (a small *bitmap*) with label s for the program.

SetDisplayRect (r)

Take rectangle r as the Display rectangle (rather than the default size).

SetNamestripe (s)

Give the program's window a name.

DisplayLock (r)

Lock a rectangle r on the display.

DisplayUnlock ()

Unlock a rectangle r on the display.

Menus: (Hit(3))
Routines for defining menus; 4 types: string, bitmap, slider or switch. Each can be sticky or popup.

int **Hit** (m, buttons)

Wait for one of the mouse buttons to be pressed when in a menu, and return the selected menu item.

Menu ***NewMenu** (arguments)

Return a menu made up from string arguments. Menu hierarchies are possible.

Menu ***NewMenuArray** (char **items)

Like NewMenu, but the string items are in one array.

BMenu ***NewBMenu** (x, y, items)

Make a menu of bitmap items, which are arranged in x rows and y columns.

Slider ***NewSlider** (value, min, max, title, orient, Tic, tic)

Make a slider which lets the user input a value within a specified range. The range may be annotated with tick marks.

SetSlider (s, value)

Set and update the slider's value.

Switch ***NewSwitch** (title, s1, s2, s3)

Create and return a two or three state switch.

int **AddMenu** (m, s, max, scroll)

Add a string item s to the bottom of a menu and scroll if applicable; return true if successful.

`int` **DelMenu** `(m, s, max)`

Delete item s from menu; return true if successful.

`Rectangle` **DrawPopup** `(m, p)`

Draw a popup menu (`Menu, BMenu, Switch` or `Slider`) at point p.

UndrawPopup `(m)`

Undraw menu m.

Mouse: (mouse(3))
Mouse related routines. Inquiring for pressed mouse switches (buttons), manipulating the mouse cursor.

`int` **button** `(i)`

True if any button of the given combination is pressed on the mouse (`i` = 1, 2, 3, 12, 13, 23, 123).

`int` **button*i*** `()`

Same as button(i).

`int` **buttons** `(i)`

True if all buttons of the given combination are depressed.

WaitButtons `()`

Poll the mouse until the buttons change.

`int` **MouseButton** `()`

Return the currently depressed mouse button.

SetMouse `(p)`

Position the cursor at p.

`Cursor` ***SetCursor** `(Cursor *c)`

Set the cursor to a new cursor c and return the current one.

CursorOn `()`

Turn the cursor on.

CursorOff `()`

Turn the cursor off.

DeclareCursor `(name, hx, hy, code, bits)`

Declare a cursor by name and point out its hot-spot.

`Cursor` ***ReadCursor** `(fname)`

Read a cursor from file and return it.

`int` **WriteCursor** `(Cursor *c, fname)`

Write cursor c to a file, and report success or failure.

Keyboard & Text: (kbdchar(3))
Keyboard I/O and displaying typed text on the screen.

kbdchar `()`

Return the next character typed at the keyboard.

`int` **keydown** `(c)`

Return true if key c is currently down.

kqueue `(c)`

Put c in the input character queue.

`Point` **stringf** `(b, p, font, f, fmt, args)`

Make up a string from the args according to fmt and draw it in bitmap b at point p. Return the position where the next character should be drawn.

ReadString `(prompt, s, maxlen, r, p)`

Popup a rectangle with a prompt and read a string, displayed starting at point p.

`Point` **PutChar** `(c, font, r, p, f)`

Draw one character at point p (in the given font and code f).

`Point` **PutString** `(s, font, r, p, f)`

Draw a string at point p (in the given font and code f; wrap within ractangle r if applicable).

`Point` **GetString** `(s, maxlen, font, r, p)`

Read a string into s typed at the keyboard. Input is done within the rectangle r.

`Point` **BoldString** `(b, p, s, font, f)`

Display string s in a boldish fashion.

`Rectangle` **TextRect** `(font, s)`

Return a rectangle large enough to hold s in the given font.

Graphics drawing primitives: (balloc(3))
Low level bitmap routines for bitmap allocation, *bitblt*, drawing points, text, fill patterns, line segments.

`Bitmap` ***balloc** `(r)`

Allocate a Bitmap with the size of r and return it, or 0 if no memory is left.

bfree `(b)`

Free up any space used by b.

bitblt `(db, p, sb, r, f)`

Bit block transfer: copy bits from area r in source sb to point p in destination db using code f (store, or, xor, set, clr).

rect `(b, r, f)`

Draw rectangle r in bitmap b using code f.

texture `(db, r, sb, f)`

Paint in destination bitmap dp the given texture sb.

segment `(b, p1, p2, f)`

Draw a line segment in bitmap b from p1 to p2.

`Point` **string** `(b, p, s, font, f)`

Draw a string at point p (in the given font and code f; return point for the next character).

`int` **point** `(b, p, f)`

Draw the pixel at p in b using code f and return its value.

getpoint `(b, p)`

Return the pixel value of p in bitmap b (0 or 1).

Clear `(b)`

Clear all pixels in bitmap b.

DeclareBitmap `(name, x, y, bits)`

Declare a Bitmap structure known by name, consisting of a rectangle x wide and y high, where bits define the initial pixels.

`short` **realbits** `(b) []`

Via this array one can access the current pixels of a bitmap b.

For `Point` **arithmetic:** (Pt(3))
Routines to manipulate 'Points' and inquiring whether a point is within a distance of *gap* from a line or segment.

`Point` **Pt** `(x, y)`

Make a Point structure with fields x and y, and return it.

`int` **eqpt** `(p1, p2)`

True if p1 and p2 are the same point.

`Point` **PtCopy** `(p1, p2)`

Point p1 becomes point p2.

`Point` **add** `(p1, p2)`

Return Pt(p1.x+p2.x, p1.y+p2.y).

`Point` **sub** `(p1, p2)`

Return Pt(p1.x-p2.x, p1.y-p2.y).

`Point` **mul** `(p1, p2)`

Return Pt(p1.x*p2.x, p1.y*p2.y).

`Point` **div** `(p1, p2)`

Return Pt(p1.x/p2.x, p1.y/p2.y).

`Point` **shift** `(p, n)`

Return Pt(p.x << n, p.y << n); n<0 is right shift.

ponline `(p, p1, p2, gap)`

True if p is within a distance of gap from the line p1...p2.

ponsegment `(p, p1, p2, gap)`

True if p is within a distance of gap from segment p1...p2.

For `Rectangle` **arithmetic:** (Rect(3))
Rectangle manipulation and inquiries of sizes, relationships between rectangles and points.

`Rectangle` **Rect** `(xo, yo, xc, yc)`

Make a rectangle structure out of points (xo,yo) and (xc,yc) with fields o (origin) and c (corner).

`int` **eqrect** `(r1, r2)`

True if r1 and r2 are the same.

`int` **rXr** `(r1, r2)`

True if rectangles r1 and r2 intersect.

`Rectangle` **Rpt** `(p1, p2)`

Make a rectangle out of points p1 and p2.

`Rectangle` **Raddp** `(r, p)`

Return Rpt(add(r.o, p), add(r.c, p)).

`Rectangle` **Rsubp** `(r, p)`

Return Rpt(sub(r.o, p), sub(r.c, p)).

`Rectangle` **inset** `(r, n)`

Return a rectangle inset n pixels from the border of rectangle r.

`Rectangle` **RectCanon** `(p1, p2)`

Returns a canonical rectangle made from two points where p1 and p2 are not necessarily origin and corner.

`int` **pinr** `(p, r)`

True if p is contained in r.

`int` **ponr** `(p, r, gap)`

True if p is within a distance of gap from the border of r.

`int` **rinr** `(r1, r2)`

True if r1 is contained in r2 (inclusive).

UpdateRect `(oldr, newr)`

Undraw the old rectangle and draw the new one.

RectClear `(r)`

Clear the pixels in rectangle r.

RectInvert `(r)`

Invert the pixels in r.

RectFlash `(r)`

Invert the pixels in r *twice.*

`int` **InRpt** `(p, p1, p2)`

True if point p lies in the rectangle Rpt(p1,p2).

`int` **HS** `(r)`

Return the horizontal size of r.

`int` **VS** `(r)`

Return the vertical size of r.

Other: (sunutil(3))
All kinds of miscellaneous routines, e.g. higher level graphics (circles, ellipses); user input of points, rectangles, confirmation; font manipulation and I/O, bitmap I/O.

border `(b, r, n, f)`

Draw a border in bitmap b starting at rectangle r and extending inward n pixels.

DoubleBox `(b, r, f)`

Draw a double box (border) in rectangle r in bitmap b.

DrawShadow `(r)`

Draw a shadowlike grey texture of some thickness at the right and bottom sides of r.

circle `(b, p, r, f)`

Draw a circle of radius r with center at p.

disc `(b, p, r, f)`

Draw a disc with radius r centered at p in bitmap b. The code f is applied to the whole circle area (disc).

ellipse `(b, p, a, b, f)`

Draw an ellipse centered at p with half-axes a,b in bitmap b.

`int` **Confirm** `(button)`

True if button was depressed by the user as a confirmation.

`Point` **GetPoint** `(button)`

Get a point from the user and return it.

`Rectangle` **GetRect** `(button)`

Let the user sweep out a rectangle, and return it when user releases button.

`Rectangle` **TrackRect** `(r)`

By moving the mouse, slide r; fix it when a button is pressed and return the new rectangle.

nap `(n)`

Sleep for n Hz.

move `(p)`

Move to point p in the display which becomes the new current point (DisplayPt). Used in conjunction with line().

line `(p, f)`

Draw a line on the display from the current point to p.

spline `(b, Point *p, n, f)`

Draw a splined curve from p[0] to p[n-1], in mode f.

`int` **Between** `(x, min, max)`

True if min $\leqslant$ x $\leqslant$ max.

`int` **Clip** `(x, min, max)`

Return value x clipped into range min...max (inclusive).

`int` **Transform** `(x, p1, p2)`

Transform x from range p1 (ie. p1.x .. p1.y) to range p2 (ie. p2.x .. p2.y) and return it.

`Point` **PtTransform** `(p, c1, c2)`

Transform p from coordinate frame c1 to c2 and return it.

`Rectangle` **RectTransform** `(r, c1, c2)`

Transform r from coordinate frame c1 to c2 and return it.

`Point` **Center** `(r)`

Return the center point in rectangle r.

`Bitmap` ***readBitmap** `(fp, fname)`

Read a bitmap from file fp (a `FILE` pointer) and return it.

`Bitmap` ***ReadBitmap** `(fname)`

Read a bitmap from the named file and return it.

writeBitmap `(b, fp, fname)`

Append bitmap b to file fp (in format *bitmap(5)*).

`int` **WriteBitmap** `(b, fname)`

Write bitmap b to the named file and report success or failure.

`Font` ***ReadFont** `(fname)`

Read and return the font in file fname.

`int` **WriteFont** `(font, fname)`

Write the font on file fname (in format *sunfont(5)*).

FontFree `(font)`

Free the allocated font.

`int` **CharHS** `(font, c)`

Return the horizontal size of character c in the given font.

`int` **CharVS** `(font, c)`

Return the vertical size of character c in the given font.

`int` **StrHS** `(font, s)`

Return the horizontal size of string s in the given font.

`int` **StrVS** `(font, s)`

Return the vertical size of string s in the given font.

`int` **FontVS** `(font)`

Return the default vertical size of the font.

`int` **FileExists** `(fname)`

True in file fname is readable.

`char` ***SearchPath** `(path, fname)`

Look for file fname in every directory in path until found, and return the pathname.

`char` ***PicturePath** `()`

Returns the path in which to search for bitmap pictures.

`char` ***FontPath** `()`

Returns the path in which to search for Sun fonts.

`char` ***StrSave** `(s)`

Save s in memory somewhere and return a pointer to it.

GIPE : **CEC 348/S/T1/4**
version 1.2 December 1985

Syntax Directed Editing of LeLisp

Annexe D1.A3 of Deliverable D1 — Second Review —

D. Clement (SEMA)

1. Introduction

The Lisp language is known to be a quite flexible programming language. The Lisp programmer has the possibility to define his own programming style, even to a certain extent his own programming language. There is no Lisp predefined construct, and users may not only define their own functions, but also system functions such as *DE* or *EVAL* (although results may be surprising).

On the other hand, the concrete syntax of Lisp is rather simple. Lisp objects are atoms or lists of objects. Parsing Lisp programs is merely reduced to lexical analysis of identifiers and construction of list of terms. In this context, the design of a syntax directed editor for Lisp with the Mentor system exhibits unusual difficulties.

We shall point out main aspects of Lisp that are relevant in the context of syntax directed editing. Then we will describe the general principles followed in the design of a Mentor editor to deal with the LeLisp/Ceyx language. Experiments carried out on a large variety of Lisp programs have shown that these principles were insufficient. Interactions between the editor and the LeLisp environment were found to be too weak to make the system practical. However, experience gained during this task convinced us that a Lisp syntax directed editor is feasible in the context of the environment generator currently under development.

2. Requirements for a Lisp Editor

A Lisp environment provides a large number of functions, among which Lisp programmers will select their favourite constructs. In fact the Lisp system itself is built from a small collection of basic primitives, and it is easy to augment it with a private collection of primitives, i.e. to build a special purpose environment. In all generality a Lisp editor should allow to define, not only new functions, but also new abstract syntax trees and new parsing as well as unparsing rules. This means that the description of the Lisp formalism in use should be extensible by the user of the editor. To be more precise, users should be able to:

- describe the abstract syntax they want to associate to their new constructs.
- describe the pretty-printing rules for the corresponding abstract trees.
- describe parsing rules, i.e. the concrete syntax and tree building actions, for their new constructs.

Such functionalities are indeed very attractive. Their implementation raises several technical problems. The first question put by these remarks regards the definition of a formalism to describe abstract syntax, pretty-printing, and parsing. One should allow changes in one of these three aspects independently of the other two. Users may wish to modify only the pretty-printing of functions, or to change both abstract syntax and pretty-printing, or to change only parsing rules. So, clearly the definition should not be monolithic. We do not pretend to define here such a formalism, but to separately enumerate the main problems raised by each one of these three points.

2.1. Abstract Syntax

Extending the abstract syntax of a formalism does not present intrinsic difficulties. It is only necessary to check that new abstract constructs are consistent with the previous description of the formalism. Problems arise with remanent storage. The brute force consists in a kind of *dump* on external memory of the state of the editor. This solution leads to huge files, and is not acceptable.

When trees are saved on external memory, information on the language must be coded in an appropriate manner. In the case of a formalism that has been fixed once and for all, this coding mechanism takes advantage of the uniqueness of the description of that formalism to optimise code generation. Furthermore, the description of the abstract syntax itself can be compiled into an efficient manageable form. Such an optimisation and compilation are no longer feasible when the formalism changes with the current *state* of the editor. Then new techniques will have to be developed to save abstract trees without redundancy on remanent storage.

2.2. Pretty Printing

Pretty-printing of programs is merely a matter of style and it depends heavily on the programmer's choices. This is more true with LISP than with other programming languages. It requires to describe the pretty-printing of structured data with a formal language. Once more, remark that this description has to be separated from the abstract syntax and parsing definitions.

The design of a pretty-printing language does not present difficulties. For example, with each operator of an abstract syntax one associates instructions describing its concrete layout. These instructions may be interpreted by a special purpose display processor. Then it is not difficult to handle dynamic formatting descriptions of abstract syntax trees. Unfortunately, experiments carried out in that area have shown that such a display *interpreter* is too slow to be useful.

In fact, to be of real use, a pretty-printing language has to be compiled. For example it is possible to automatically translate pretty-printing instructions written in a pretty-printing language into a LELISP program. Then this LELISP program may be compiled, as any other LELISP program. Of course the resulting unparser of abstract trees is far more efficient than any interpreter of pretty-printing instructions.

As for remanent storage optimisation, this compilation takes advantage of the completeness of the description of the formalism. We know all possible cases for every abstract operator, as well as every operator in a phylum. This is no longer the case when the description of the formalism may be updated by the user.

2.3. Parsing

In fact the most delicate point is parsing. Concrete description of programming languages are usually expressed with BNF like rules. As it is, this formalism allows, with some care, to add new rules or to modify old rules. We may design an interpreter of such a BNF like language able to parse LISP programs containing new parsing rules for new constructs.

But we must keep in mind that the whole parsing process of a program depends heavily on the size of that program and tends to be time consuming. On the other hand, in an interactive environment, the parsing time of programs must be as short as possible. One more time we must compile the concrete syntax description. Usually, a parser generator compiles BNF like rules into an equivalent automaton. This compilation may be time consuming but it is only done once and for all. And the resulting automaton is usually quite efficient. When concrete rules are dynamically extended by the user we cannot any longer generate such an automaton.

As a matter of conclusion, we must admit that while we know how to design a very flexible environment generator, we do not know how to implement it efficiently, i.e. with compilers rather than interpreters. Nevertheless we may also wonder whether such a programming tool is really desirable. One of the aims of structured editors is to preserve programmers from syntax errors and to facilitate the coding of programs. We are not convinced that all LISP programming practices mentioned above are well in accordance with these basic software engineering rules. We rather like the idea of designing a syntax directed editor for a well defined LISP programming style. More practically, we will assume for the moment that an abstract syntax definition of LELISP has been chosen. LISP constructs that do not belong to that definition will not be associated to abstract operators.

3. Abstract Syntax Definition of LELISP/CEYX

The Mentor system for generating syntax directed editors from BNF-like specifications was used to develop a LELISP/CEYX environment. The User Manual of LELISP Version 15 was taken as a reference. Then, both a concrete syntax and an abstract syntax for LELISP and CEYX were defined. We will now take under consideration only the LELISP component. Problems with CEYX are very similar.

3.1. Concrete Syntax

Before talking about abstract syntax, we must make some remarks on the concrete syntax definition of LELISP. As indicated above, a concrete syntax of LELISP as atoms and lists of objects is irrelevant in the context of structural editing. In tools such as Mentor abstract trees are built during the parsing of programs. Tree building instructions are associated, as actions, to concrete rules. Hence, we need derivation rules describing how LISP primitives are built from subterms of the grammar.

For example, each one of the LELISP function definition primitives, *de, df, dm* have to be defined by a production rule in the BNF-like description of LELISP:

```
<named_expr> ::= (de <symbol> <obj_list> )

<named_expr> ::= (df <symbol> <obj_list> )

<named_expr> ::= (dm <symbol> <obj_list> )
```

For example, if a binary operator is associated to the *de* function, a tree building action may be associated to the first rule as follows:

```
<named_expr> ::= (de <symbol> <obj_list> ) ;
        de(<symbol>,<obj_list>)
```

Such rules turn identifiers *de, df, dm* into *keywords*, i.e. *reserved* identifiers: the user is no longer able to redefine them. We know that such a constraint may be too restrictive*, and we provide an escape mechanism (but not very elegant): if a keyword is preceded by a neutral character, for example a blank, it returns to the status of identifier.

* The identifier *tag* is the name of a LELISP primitive; as such, it plays the role of a keyword. Often, it is also used as the name of the first parameter of error recovery functions (*de catch* (*tag val*) ...), where it plays the role of an identifier.

3.2. Abstract Syntax Operators

We have now to define an abstract syntax for LeLisp/Ceyx. There are about five hundred functions in the LeLisp manual. Some of these functions differ only by the value they return, for example *progn, prog1, prog2*. Some are special cases of a general construct, for example *if, ifn, when, unless*. Then what should be the abstract syntax? We are faced with two delicate engineering decisions:

i) what LeLisp primitives should be chosen as operators of the language?

ii) should the abstract syntax reflect the concrete layout of Lisp functions or should it take more semantic notions into account?

The first problem is motivated by the number of LeLisp primitives, and by unusual constructs such as the *progn*s. In LeLisp, sequential evaluation of expressions is done via a standard grouping primitive *progn*, merely a kind of *begin end* construct. Expressions inside the *progn* are evaluated sequentially and the whole construct, i.e. the *progn* function itself, returns the value of the last expression. Similarly, the function *prog1*, respectively *prog2*, results in sequential evaluation of expressions, but returns the value of the first, respectively the second expression. Due to our Lisp inexperience at the time of this design and to reduce the number of Lisp primitives associated to abstract trees, we have retained only the *progn* construct.

Let us discuss the second point in the case of function definitions. In LeLisp, functions are defined using *lambda, flambda,* or *mlambda.* For example a function may be defined by:

(lambda <var_list> <s_expr1> .... <s_exprn>)

and may be assigned to a symbol by a function definition:

(de <symbol> <var_list> <s_expr1> .... <s_exprn>)

If we decide to stay close to the concrete form, these two expressions may be described by the following abstract syntax trees:

lambda: **ARGS x BODY -> SEXPR**

de: **IDENT x ARGS x BODY -> SEXPR**

On the other hand, the semantics of a *de* expression is to assign a *lambda* expression to an identifier, the semantics of a *df* is to assign an *flambda* expression to an identifier, and so on. We may decide to use that fact in the abstract syntax description, and define the following abstract tree:

defun : **IDENT FUNCTION -> SEXPR**

where FUNCTION stands for all possible kinds of function definitions, i.e. *lambda, flambda,* and *mlambda.*

It it clear that this kind of tuning may be done for many LeLisp primitives. When designing an abstract syntax for LeLisp/Ceyx we found this technique more satisfactory and providing a rather elegant abstract syntax. But results are sometime rather surprising.

For example, the four conditional functions *if, ifn, when,* and *unless* may be represented by a unique *if* abstract tree. In LeLisp the conditional instruction *if* is not symmetrical. Its concrete form is:

```
(if <expr1> <expr2> <expr3> .... <exprn>)
```

with the meaning:

- if the value of *<expr1>* is *true*, i.e. not nil, then the *if* returns the value of *<expr2>*.
- if the value of *<expr1>* is *false*, i.e. nil, then the remaining expressions *<expr2>* to *<exprn>* are evaluated sequentially and the *if* returns the value of the last evaluated *<exprn>*. Remark that this is nothing else than a *progn* like construct.

Three other conditional primitives are available in LeLisp: *ifn, when,* and *unless*.

```
        (ifn <expr1> <expr2> <expr3> .... <exprn>)
```
stands for *if <expr1> then <expr3> ... <exprn> else <expr2>*.

```
        (when <expr1> <expr2> .... <exprn>)
```
stands for *if <expr1> then <expr2> ... <exprn> else nothing.*

```
        (unless <expr1> <expr2> .... <exprn>)
```
stands for *if <expr1> then nothing else <expr2> ... <exprn>*.

It is clear that we may define a unique ternary operator

if : **SEXPR** x **SEXPR_S** x **SEXPR_S** -> **SEXPR**

with the usual *meaning*:

if SEXPR then SEXPR_S else SEXPR_S

In this way, the four LeLisp functions *if, ifn, when,* and *unless* are reduced to fictitious constructs: they are just a different layout of the same operator according to the following unparsing rules:

1) if the second son is a list of at least two elements and if the third son is reduced to a single element, then unparse the tree as a LeLisp *ifn*.
2) if the third son is empty, then unparse the tree as a LeLisp *when*.
3) if the second son is empty, then unparse the tree as a LeLisp *unless*.
4) in all other cases, unparse the tree as a LeLisp *if*, with eventually a *progn* surrounding the expressions of the true case.

Some users may be surprised by such an *optimisation* of their if constructs: a deletion of the third son of an *if* expression turns it into a *when*.

3.3. Conclusion

The design of a LeLisp/Ceyx syntax directed editor appears to be a rather delicate task. We have seen that one of the main problems to deal with was in the area of concrete syntax (some identifiers being used as keywords or as identifiers). We would like to mention an alternate approach for this problem.

In the Mentor system, abstract syntax descriptions and parsing rules are well dissociated. But a powerful tree-pattern-matching-based mechanism is provided to allow to build trees in accordance with the desired abstract syntax. In the case of LeLisp, this facility can be used to reduce the concrete description to its basic form. Let us assume that the so-called Lisp *s_expr* is described by the following concrete syntax:

An *s_expr* is:

either an *identifier*

<s_expr> ::= <ident>

or a *list* of s_exprs enclosed in parentheses

<s_expr> ::= (<s_expr_list>)

A list of s_exprs is either an empty list or a list followed by an s_expr. With that kind of concrete syntax definition we do not have any parsing problems! (The terminal <ident> stands for identifiers.)

Let us see now how we may build abstract trees. Consider the following abstract syntax:

sorts

```
        SEXPR, IDENT, SEXPRS, FUNCTION
```

subsorts

```
        SEXPR > IDENT, FUNCTION
```

functions

```
        ident    :                        -> IDENT
        list     :  SEXPR*                -> SEXPR
        defun    :  IDENT x FUNCTION      -> SEXPR
        tag      :  IDENT x SEXPR_S       -> SEXPR
        lambda   :  SEXPR x SEXPR_S       -> FUNCTION
        sexpr_s  :  SEXPR*                -> SEXPR_S
```

As noted in the footnote on page 3, the identifier *tag* may be used as the name of a LE_LISP primitive:

(tag <ident> <s_expr_list>)

or as *the first parameter* of so-called *lock* functions. A lock function must have two parameters and must be either a *lambda* or a *de*:

(lambda (tag <ident>) <s_expr_list>)

(de <ident> (tag <ident>) <s_expr_list>)

In Mentor, the following pattern-matching instructions may be associated to concrete rules:

```
        case <s_expr_list>
             when sexprs[ident("lambda").sexprs[sexprs[ident("tag"),V].Y]]
             => lambda(list[ident("tag"),V],Y)
             when sexprs[ident("de").sexprs[X.sexprs[sexprs[ident("tag"),V].Y]]]
             => defun(X,lambda(list[ident("tag"),V],Y)
             when sexprs[ident("tag").sexprs[X.Y]]
             => tag(X,Y)
```

It is clear that with this kind of mechanism we are able to catch tag identifiers that are parameters of lock functions.

4. Perspectives

As indicated above, experiments have shown that the abstract syntax must be as complete as possible. All LISP primitives must be, in one way or another associated to abstract operators. We must admit that we left out a rather important aspect in the context of LISP editing: the level of interaction between the editor and the LE_LISP environment. LISP programmers are used to check functions quite often, and thus they switch frequently from the LISP top-level to the editor: this process must be as "user friendly" as possible. With the Pascal version of the Mentor system we cannot satisfy such a constraint. In fact the interface between the LE_LISP environment and Mentor is quite time consuming. As a consequence this environment has not been used as expected during the coding of the Virtual Tree Processor.

We are convinced that the main reason for the failure of that first attempt of a LISP syntax directed editor consists in the poor user interface. In the context of our project, experiments gained with the Mentor LISP editor will be very useful to develop a more complete version with tools currently under development. The kernel of the GIPE System is written in LE_LISP. Its interface is clear and efficient. We believe that in this context, the interface difficulty will be resolved.

Appendix

Abstract Syntax of Le_Lisp and Ceyx

sorts

CEYX, SEXPR, MODE, FIELD, FIELDS, OGETQ, MODEL, NAME, DEFABBREV, SYMB, ASSIGN, SETQ, STEP, ARGS, COND, CLAUSE, DEC_WITH, DECW, DEC_FLET, DECF, DEC_LET, DECL, FUNCTIONS, LAMBDA_EXPR, ARG_LIST, SEXPRS, LL_LIST, ID

subsorts

CEYX>
  OGETQ, DEFABBREV
SEXPR>
  CEYX, SETQ, COND, SEXPRS, LAMBDA_EXPR, LL_LIST, SYMB
MODE>
  SYMB
MODEL>
  SYMB
NAME>
  DEFABBREV, SYMB
SYMB>
  ID
FUNCTIONS>
  LAMBDA_EXPR
ARG_LIST>
  LL_LIST, SYMB

functions

'Basic Lisp Objects'

id	:		→	ID
number	:		→	SEXPR
string	:		→	SEXPR
ll_list	:	SEXPR*	→	LL_LIST
vector	:	SEXPR*	→	SEXPR

'Evaluation Functions'

sexprs	:	SEXPR*	→	SEXPRS
quote	:	SEXPR	→	SEXPR

'Application Functions'

lambda	:	ARG_LIST×SEXPRS	→	LAMBDA_EXPR
flambda	:	ARG_LIST×SEXPRS	→	LAMBDA_EXPR
mlambda	:	ARG_LIST×SEXPRS	→	LAMBDA_EXPR
dlambda	:	SEXPR×SEXPRS	→	FUNCTIONS
apply	:	SEXPR×ARG_LIST	→	SEXPR
funcall	:	SEXPR×SEXPRS	→	SEXPR

'Functions that modify the Environment'

ll_let	:	DEC_LET×SEXPRS	→	SEXPR
slet	:	DEC_LET×SEXPRS	→	SEXPR
letn	:	SYMB×DEC_LET×SEXPRS	→	SEXPR
letv	:	SEXPR×SEXPR×SEXPRS	→	SEXPR
letvq	:	SEXPR×SEXPR×SEXPRS	→	SEXPR
dec_let	:	DECL*	→	DEC_LET
decl	:	SEXPR×SEXPR	→	DECL

'Definition Functions'

defun	:	SYMB×FUNCTIONS	→	SEXPR
flet	:	DEC_FLET×SEXPRS	→	SEXPR
ll_with	:	DEC_WITH×SEXPRS	→	SEXPR
dec_flet	:	DECF*	→	DEC_FLET
decf	:	SYMB×SEXPR×SEXPRS	→	DECF
dec_with	:	DECW*	→	DEC_WITH
decw	:	SYMB×SEXPRS	→	DECW

'Control Functions'

if	:	SEXPR×SEXPRS×SEXPRS	→	SEXPR
or	:	SEXPR*	→	SEXPR
and	:	SEXPR*	→	SEXPR
cond	:	CLAUSE*	→	COND
selectq	:	SEXPR×COND	→	SEXPR
while	:	SEXPR×SEXPRS	→	SEXPR
until	:	SEXPR×SEXPRS	→	SEXPR
repeat	:	SEXPR×SEXPRS	→	SEXPR
for	:	ARGS×SEXPRS	→	SEXPR
clause	:	SEXPR×SEXPRS	→	CLAUSE
args	:	SYMB×STEP×SEXPRS	→	ARGS
step	:	SEXPR×SEXPR×SEXPR	→	STEP

'Exception Functions'

tag	:	SYMB×SEXPRS	→	SEXPR
untilexit	:	SYMB×SEXPRS	→	SEXPR
lock	:	SEXPR×SEXPRS	→	SEXPR
protect	:	SEXPR×SEXPRS	→	SEXPR

'List Constructors'

mcons	:	SEXPR*	→	SEXPR
mlist	:	SEXPR*	→	SEXPR
append	:	SEXPR*	→	SEXPR

'Assignment Functions'

setq	:	ASSIGN*	→	SETQ
assign	:	SEXPR×SEXPR	→	ASSIGN
synonymq	:	SYMB×SYMB	→	SEXPR

'Character Macros'

brace	:	SYMB	→	SYMB
colon	:	SYMB×ID	→	SYMB
sharp	:	SEXPR	→	SEXPR

'Ceyx Models'

defmodel	:	NAME×MODEL	→	CEYX
defabbrev	:	SYMB×SYMB	→	DEFABBREV
ceyx_field	:	SYMB×MODEL×SEXPR	→	MODEL
ceyx_predicate	:	SYMB×SEXPR	→	MODEL
ceyx_list	:	MODEL	→	MODEL
ceyx_cons	:	MODEL×MODEL	→	MODEL
ceyx_vector	:	MODEL*	→	MODEL

'Ceyx Semantics'

omatchq	:	SYMB×SEXPR	→	CEYX
ogetq	:	SYMB×SYMB×SEXPR	→	OGETQ
oputq	:	OGETQ×SEXPR	→	CEYX
omakeq	:	SYMB×SETQ	→	CEYX
defaccess	:	SYMB×SEXPRS	→	CEYX
defmake	:	SYMB×SYMB×LL_LIST	→	CEYX

'Ceyx Types'

deftype	:	NAME×MODEL	→	CEYX
tcons	:	SYMB×SEXPR	→	CEYX
type	:	SEXPR	→	CEYX
sendq	:	SYMB×SEXPRS	→	CEYX

'Ceyx Structures'

defrecord	:	NAME×FIELDS	→	CEYX
deftrecord	:	NAME×FIELDS	→	CEYX
defclass	:	NAME×FIELDS	→	CEYX
deftclass	:	NAME×FIELDS	→	CEYX
deftree	:	NAME×FIELDS	→	CEYX
defcons	:	NAME×FIELDS	→	CEYX
fields	:	FIELD^+	→	FIELDS
field	:	MODE×SEXPR	→	FIELD
mode	:	SYMB×MODEL	→	MODE

D2 - DEFINITION OF COMMON INTERFACES

D2.A1 - INTERFACES BETWEEN LISP AND PROLOG

D2.A2 - INTERFACES BETWEEN LISP AND ASH

D2.A3 - THE VIRTUAL TREE PROCESSOR

GIPE : **CEC 348/S/T2/1**
version 2.1 December 1985

Definition of Common Interfaces

Deliverable D2 of Task T2

D. Clement (SEMA)
T. Despeyroux (INRIA)
M. Devin (INRIA)
L. Gallot (INRIA)
B. Lang (INRIA)

The present status of T2 is described in this document. It describes the interfaces for each component of the Common Environment which have been listed in the Deliverable D1. The resulting Common Environment will integrate these tools.
This document is the final report for the Second Review of February 1986.

1. Introduction

To build the Common Environment we have not only to choose various software components but also to define communication protocols between these components. Task T2 consists in designing the appropriate interfaces for each component.

In such a software design process, we could separate interfaces in at least two levels of dependencies:

- the system level, describing how components are connected with the operating system in use.
- the component level, describing how components are connected with other components.

The whole project being based on the use of the UNIX operating system, we are not concerned with the first level. We are mostly concerned with the use in our interfaces of concepts provided by that operating system. Hence, interfaces will be expressed in terms of *how to use* a given application or *how to make* a more complete application from different parts.

The next aspect to deal with is whether or not we use an already existing component or a new one. In the context of this project we have to consider both of these cases. Interface description will be of two kinds:

a) Those involving existing software, such as LE_LISP, C-Prolog, and Ash, for which we emphasize extensions made to their respective interfaces.

b) Those involving new components, such as the virtual tree processor, for which a more precise description of the architecture will be given.

Some components, for example LE_LISP and C-Prolog, are already in wide use on a variety of machines. On the other hand, we must pay more attention to window management, because there is no well-accepted, portable window management subsystem. High resolution graphics devices are not standardised in any way, so that we must insist on interfacing them through a virtual device specification. The component that we have experimented with, ASH, which comes from Brown University, is organised in that way. ASH *per se* knows only of a virtual device that has capabilities such as clipping, line and polygon drawing etc... An interface exists on the Sun workstation, on the Apollo workstation. In spite of scarce documentation, it was possible to adapt it completely to the SPS 7 bitmap device at INRIA. Clearly, these three bit-map displays have extremely different hardware capabilities.

2. Interfaces between Lisp and Prolog

Experiments have shown that Prolog can be used as a target language for the compilation of specifications. In the context of an interactive programming environment implemented in Lisp, using Prolog as an inference system implies that Prolog may be used and accessed from within the Lisp language, and in turn, that Lisp may be called from within the Prolog system. Furthermore, it must be pointed out that one should be able to call both Lisp and Prolog *recursively*.

The description of an interface between LE_LISP [1] and C-Prolog [3], as well as some examples, may be found in Annexe D2.A1.

3. Interface between Lisp and Ash

Ash [2] is a low level screen handler for use by graphics applications. It provides facilities for the user's programs to create and manipulate a number of virtual bitmaps on a raster display. It provides a variety of operations on text or graphics by means of primitives: text output, drawing lines and polygons (simple, dotted, dashed, ...), filling rectangles and so on. All these operations are done with reasonable speed.

On the other hand LE_LISP provides both the interactive facilities of an interpreter and a device independent input-output interface.

Ash is implemented in C. Once transported to any hardware running under UNIX, it is used as a library of primitives to which user programs may be linked. Thus the simplest solution is to extend the LE_LISP environment with Ash primitives. Then we obtain a new LE_LISP environment allowing the user to manipulate windows through Ash. The description of this environment may be found in Annexe D2.A2.

4. The Virtual Tree Processor

The kernel of a modern Common Environment has to embody some fundamental principles. It must be modular, extensible and portable. Furthermore it must be open to and from outer systems, providing a uniform interface. In view of its use in a Programming Environment Generator it must also be parameterised by the language to be manipulated and allow both concurrent manipulation of several languages *and* handling of multilingual documents.

The basic data structuring concepts to achieve these goals are:

- Abstract tree representation of structured objects. All formalisms will be specified in terms of their **abstract syntax**.
- *Annotations* and *gates*. Two distinct mechanisms, respectively called annotations and gates will allow mixing formalisms within a single document.
- Metadescription of each formalism. The metalanguage for describing formalisms will be the key to the modularity and extensibility of the system.

The detailed description of a virtual tree processor may be found in Annexe D2.A3.

5. References

[1] *LE_LISP & CEYX*

- J. Chailloux, "LE_LISP de l'INRIA, Le Manuel de Reference" (version 15), INRIA Report (to be published), February 1985.
- J.M. Hullot, "CEYX - Version 15, II: Programmer en CEYX," INRIA Technical Report no. 45, February 1985.
- G. Berry, B. Serlet, "CXYACC et LEX-KIT version 2.1," INRIA Report, March 1984.

[2] *BWE*

- J.N. Pato, S.P. Reiss, M.H. Brown, "The Brown workstation environment," Brown University CS-84-03, October 1983.

[3] *C-Prolog*

- F. Pereira, D. Warren, D. Bowen, L. Byrd, L. Pereira, "C-Prolog User's Manual, Version 1.5," EdCAAD, Department of Architecture, University of Edinburgh, U.K.(1983).

[4] *Mentor*

- V. Donzeau-Gouge, B. Lang, B. Melese, "Practical Applications of a Syntax Directed Program Manipulation Environment," Proceedings of the 7th Int. Conf. on Software Eng., Orlando Florida, March 1984.
- B. Melese, V. Migot, D. Verove, "The Mentor - V5 Documentation," INRIA Technical Report no. 43, January 1985.

GIPE : **CEC 348/S/T2/2**
version 2.1 December 1985

Interfaces between LISP and Prolog

Deliverable D2.A1 of Task T2

D. Clement (SEMA)
T. Despeyroux (INRIA)

1. Introduction

This document describes the specification of an interface between LE_LISP and C-Prolog. The proposal is based on understanding gained in experiments carried out with the Typol language on Multics (HB68), interfacing Mentor [4] with Prolog/P, a Prolog system written in Pascal, then on the SM90 (Bull SPS7), and interfacing Mentor with C-Prolog.

We will examine successively both sides of this interface.

2. Interfacing LE_LISP with C

The need to call efficient C routines from LE_LISP is shared by many applications. Hence this facility exists in LE_LISP and it is one of the attractive features of this system.

The need to invoke LE_LISP functions from C has been felt more recently, in particular within our project. Close contact with the implementors of LE_LISP has allowed a common design to emerge quickly.

2.1. Calling C from within LE_LISP

It is possible to import a C routine into the LE_LISP environment as a new LE_LISP function, using the following syntax:

(DEFEXTERN <proc> (<types>) <result_type>)

In this definition, the name *<proc>* of the C procedure must be prefixed by an underscore, due to Unix conventions. Conversion between C objects and LE_LISP objects (and conversely) is taken care of automatically, so that portability is not impaired.

The types of the arguments, and the type of the result may be one of the following:

fix for integer.
float for reals.
string for strings.
vector for vectors of fix numbers.
external for external pointers.
pointer or *t* for LISP pointers.

In general, the C routines that one may wish to call from LISP must be linked with the LE_LISP system. This may be done either statically or dynamically. When object modules produced by the C compiler are linked statically with the LE_LISP system, the primitive *defextern* may be used for every routine defined in these modules. This technique leads unfortunately to a plethora of specialised instances of LE_LISP. So it it recommended to incorporate object modules dynamically with the loader primitive:

(CLOAD <string>)

where the string argument contains names of object files. It is also possible to load a library, *with the convention of the C loader.*

2.2. Calling LE_LISP from within C

To make it possible to call LE_LISP functions from within C, a group of three procedures was designed and implemented. With the protocol adopted, recursive calls between the two systems, LE_LISP and C, may be performed efficiently. Furthermore a C declaration of LE_LISP basic objects is available. This provides a way to coerce LE_LISP objects into C objects.

2.2.1. Protocol description

To use the protocol, you must proceed as follows:

a) Get the symbolic address of a function with the

getsym(<string>);

primitive, where <string> is the LE_LISP name of the function. The same primitive may also be used to get the address of a LE_LISP atom. The LISP function may or may not be compiled.

b) Before calling the function its arguments must be pushed on top of the LE_LISP execution stack. Arguments are pushed one at a time using:

pusharg(<type>, <val>);

where <val> is the value of an argument, and <type> its type.

c) Finally, the LE_LISP function may be called with the primitive

lispcall(<type>, <nargs>, <atom-addr>);

where <type> is the type of the resulting value, <nargs> is the number of arguments of the LE_LISP function and <atom-addr> is a pointer to a LE_LISP atom. Typically, <atom-addr> is a value returned by an earlier call to getsym.

As is the case with *defextern*, <type> may be one of the following basic types, for which automatic conversions between C objects and LE_LISP objects are performed.

fix for fix numbers, 16-bits masked.

float for real numbers.

string for strings.

vector for vectors of fix numbers.

t for LE_LISP objects

2.2.2. Example

The fibonacci function is a commonly used benchmark for LISP systems, and we give a definition where computations alternate between a LE_LISP fibonacci function and a C fibonacci function. The efficiency is reduced by a factor of 2, which seems quite reasonable.

Definition of FIB (On the LE_LISP side)

```
(_cload "fib.o")              ; the object module is dynamically linked
(defextern _fib (fix) fix)    ; and defines the external _fib function

        (de fib (number)
            (if (eq 0 number)
                ;then return 1
                1
                ;else
                (if (eq 1 number)
                    ;then call the external _fib
                    (_fib 1)
                    ;else alternating calls to fib and _fib
                    (+ (fib (- number 1)) (_fib (- number 2)))
                )
            )
        )
```

Definition of FIB (On the C side)

The types LL_SYMBOL and LL_OBJECT are defined in the include file lispcall.h, shown in Appendix I.

```
        int fib(number)
        int number;
        {
            LL_SYMBOL llfib;
            LL_OBJECT llres;

            if (number == 0) return 1;
            if (number == 1) return 1;
            llfib = getsym("fib");          /* Get the address of the LE_LISP
                                               fib function */
            pusharg(LLT_FIX, number - 1); /* Push an argument of type fix
                                               number */
            llres = lispcall(LLT_FIX, 1, llfib); /*  Call the LE_LISP fib
                                                     function with 1 parameter */
            return ((int)llres + fib(number - 2));
        }
```

Calls of *getsym("fib")* are not necessary, and a better solution would be to establish the link to the LE_LISP function fib only once. Let us define an initialisation primitive:

```
        lisp_init_fib(a)
        LL_SYMBOL *a;
        {
           llfib = a;
        }
```

Then we can use it inside the LE_LISP system to assign to "llfib" the current LE_LISP fibonacci function. For example:

```
        (defextern _lisp_init_fib (t) fix)
        (_lisp_init_fib 'fib)
```

3. Interfacing C-Prolog with LE_LISP

The C-Prolog system was not designed to be used as a subroutine from other software, and we had to modify it slightly. A careful study has confirmed that the implementation of C-Prolog can be brought under control. It is clearly a well designed and flexible piece of software.

For our purposes, the main program of the C-Prolog system had to be modified:

1) to be able to ask for a clause resolution without going through the usual interactive top level of C-Prolog.

2) to be able to define new special purpose predefined predicates

3.1. Calling C-Prolog from within LE_LISP

Prolog may be called from within LE_LISP like any other collection of C procedures. But at this stage, we only require resolution of a clause in a somewhat "background-like" manner. But inside a clause resolution it may be necessary to call a LE_LISP function that will in turn ask for a Prolog resolution of another clause. To allow such a completely recursive mechanism, we use the break level facility of the C-Prolog system. Each call of Prolog from LE_LISP increases that level, which is decreased upon returning to the caller.

When Prolog is called for the first time it builds its environment, reading a "startup" file, as usual. Later calls will find Prolog exactly as it was left after returning from the previous call, so that C-Prolog and LE_LISP may be seen as two *coroutines*.

To communicate requests, we allow the C-Prolog top-level to look for messages in a memory resident buffer, the so-called Prolog mailbox. Requests to Prolog may be sent to the mailbox, via the primitive:

(mailtoprolog <string> <number>)

where the <string> argument contains the message, and the <number> is the length of the string.

Then an entry point in C-Prolog may be activated that:

- fetches its goal from the Prolog mailbox rather than from some file.
- silently resolves its goal, without a plethora of spurious messages.
- then returns to its caller.

The full mechanism is encapsulated in one LE_LISP function:

({prolog}:send <string>)

The usual entry point, *prolog*, is still reachable and it will normally read its input from the standard input, print the familiar messages on standard output etc. This "standard" mode may be very useful for debugging. This mode may be obtained with the LE_LISP function:

(prolog)

which switches from LE_LISP to Prolog:

```
 :- PROLOG_PREDICATES.          C-Prolog top-level
 :- halt.                       We stop using Prolog, and return to LE_LISP
```

3.2. Calling LE_LISP from within C-Prolog

It is necessary to call LE_LISP from within Prolog, when evaluable predicates use functions written in LE_LISP. These evaluable predicates call LE_LISP via a small collection of predefined predicates that give access to the LE_LISP interface. For efficiency reasons, we had to extend these general purpose predicates with more specialised predicates corresponding to functions of the Virtual Tree Processor interface.

Large amounts of data may have to be transmitted between LISP and Prolog and this appears to be a rather common activity. One solution is to use the Prolog mailbox. Experience has shown this method to be intolerably slow: the Prolog reader routine is not efficient enough. Instead, using a collection of evaluable predicates, one improves transmission time by a factor of 100.

3.2.1. Basic predicates

The first collection of evaluable predicates corresponds to the standard interface between C and LE_LISP.

1) **getsym(arg1,arg2)** <=> arg2 = getsym(arg1)

arg1 must be an atom denoting a LE_LISP function, and arg2 a Prolog variable. In case of success arg2 will contain the same result as getsym(arg1).

2) **pusharg(type, val)** <=> pusharg(type, val)

where val is a value of a type allowed by the C to LE_LISP interface, i.e. fix, float, string, vector, and pointer.

3) **lispcall(type, nargs, ll_name, res)** <=> res = lispcall(type, nargs, ll_name)

where res is a Prolog variable that will contain the result of the LE_LISP function.

3.2.2. Special purpose predicates

With the previous three predicates, it is possible to call any LE_LISP function from within C-Prolog. But as indicated before, we have implemented special purpose predicates for communication between the Virtual Tree Processor and the C-Prolog system.

- **getvar(atom, value)**

  Gets the value of a LE_LISP variable *atom*. The result is in *value*.

  Calling conditions: *atom* is a Prolog atom, *value* is a Prolog variable.

- **gettree(lang, tree, term)**

  Gets the V.T.P. tree denoted by *tree*, which is in language *lang*. The result will be the corresponding Prolog *term*.

  Calling conditions: *lang* is a Prolog atom, *tree* is the result of a previous call to getvar, and *term* is a Prolog variable.

- **sendtree(lang, term, tree)**

  Builds the **abstract tree** corresponding to the Prolog term *term*, which is in language *lang* within the LE_LISP environment. The resulting *tree* will be a reference to this abstract tree.

  Calling conditions: *lang* is a Prolog atom, *term* is a Prolog term, and *tree* is a Prolog variable.

- **sendvar(lang, atom, tree)**

  Assigns the V.T.P. tree denoted by *tree*, which is in language *lang*, to the LE_LISP variable named *atom*.

  Calling conditions: *lang* and *atom* are Prolog atoms, and *tree* is the result of a previous call to sendtree.

3.2.3. Example

Let us assume that we have to define a new predicate to get information on all operators belonging to a given language. This information is stored in a LE_LISP structure and is accessible via a LE_LISP primitive {operators}:list-off, taking the name of the language as a parameter. The corresponding C-Prolog predicate will be

operators-list-off(arg1, arg2)

where arg1 is an atom and arg2 a Prolog variable that will receive the result of the call to {operators}:list-off.

In such a case there is only one way to exchange data between LE_LISP and Prolog, by using the mailbox. The LE_LISP function {operators}:list-off will send its result to the mailbox, using the *mail-toprolog* primitive.

We first define the auxiliary function *call-lisp-list-off*, using predefined predicates:

```
call-lisp-list-off(Lang) :-
        getsymb('#:operators:list-off', LISPaddr),
        type('LLT_STRING', S),
        pusharg(S, Lang),
        type('LLT_FIX', F),
        lispcall(F, 1, LISPaddr, Res).   Puts messages in the mailbox
```

And now the definition of *operators-list-off* will be:

```
operators-list-off(Lang, Opers) :-
        atom(Lang),                        Checks on arguments types
        var(Opers),
        call-lisp-list-off(Lang),          send messages to the mailbox
        see($mailbox),                     that is read to get the
        read(Opers),                       desired result
        seen.
```

From the LE_LISP system one may invoke the Prolog resolution of the predicate by calling the {prolog}:send function:

```
({prolog}:send ":-operators-list-off(pascal, Opers).")
```

4. References

[1] *LE_LISP & CEYX*

- J. Chailloux, "LE_LISP de l'INRIA, Le Manuel de Reference" (version 15), INRIA Report, February 1985.
- J.M. Hullot, "CEYX - Version 15, II: Programmer en CEYX," INRIA Technical Report no. 45, February 1985.
- G. Berry, B. Serlet, "CXYACC et LEX-KIT version 2.1," INRIA Report, March 1984.

[2] *C-Prolog*

- F. Pereira, D. Warren, D. Bowen, L. Byrd, L. Pereira, "C-Prolog User's Manual, Version 1.5," EdCAAD, Department of Architecture, University of Edinburgh, U.K.(1983).

[3] *Mentor*

- V. Donzeau-Gouge, B. Lang, B. Melese, "Practical Applications of a Syntax Directed Program Manipulation Environment," Proceedings of the 7th Int. Conf. on Software Eng., Orlando Florida, March 1984.
- B. Melese, V. Migot, D. Verove, "The Mentor - V5 Documentation," INRIA Technical Report no. 43, January 1985.

5. APPENDIX I: C-LE_LISP structures

We give below a complete listing of C declarations for every basic LE_LISP data type. These declarations are included in the file *lispcall.h* which is provided with the C-LE_LISP interface.

5.1. TYPE DECLARATIONS

```
typedef int LL_FIX;

typedef double LL_FLOAT;
```

A LE_LISP object may be of type SYMBOL, CONS, STRING, VECTOR, FIX or FLOAT. We would like to define the type of a LE_LISP object as a union of these types, but this is not possible in the C language.

```
/*
typedef union {
     struct LL_SYMBOL *ll_symbol;
     struct LL_CONS   *ll_cons;
     struct LL_STRING *ll_string;
     struct LL_VECTOR *ll_vector;
            LL_FIX     ll_fix;
            LL_FLOAT  *ll_float;
} LL_OBJECT;
*/
typedef char *LL_OBJECT;

struct LL_SYMBOL {
     LL_OBJECT  ll_cval;
     LL_OBJECT  ll_plist;
     LL_OBJECT  ll_fval;
     LL_OBJECT  ll_alink;
     LL_OBJECT  ll_pkgc;
     LL_OBJECT  ll_oval;
     char             ll_ftype;
     char             ll_ptype;
     short            ll_pad;
     LL_OBJECT  ll_pname;
};

struct LL_CONS {
     LL_OBJECT  ll_car;
     LL_OBJECT  ll_cdr;
};

struct LL_STRING {
     struct {
        struct LL_STRING *ll_strarr;
        int               strsize;
        char              ll_strfil;
     }  *ll_strobj;
};

struct LL_VECTOR {
     struct{
          struct LL_VECTOR *ll_vecarr;
          int             vecsize;
          LL_OBJECT    ll_vecfil;
     } *ll_vecobj;
};
```

5.2. SYMBOLIC TYPES OF PARAMETERS

```
#define LLT_T 0        /* LE-LISP Object */
#define LLT_FIX 1      /* Integer */
#define LLT_FLOAT 2    /* Real */
#define LLT_STRING 3 /* String */
#define LLT_VECTOR 4 /* Vector */
```

5.3. FUNCTION DECLARATIONS

```
/*  struct LL_SYMBOL *getsym (pname) char *pname; */
struct LL_SYMBOL *getsym();

/*  void pusharg (ll_type, value) int ll_type; any value; */
void pusharg();

/*  LL_OBJECT lispcall (ll_type, narg, symbol)
         int ll_type, narg; struct LL_SYMBOL *symbol;  */

LL_OBJECT lispcall();
```

6. APPENDIX II: An example

We describe an example using the interface between C and LE_LISP to implement a tree coercion primitive from LISP objects to Prolog terms. We use some primitives of the Virtual Tree Processor described in Annexe D2.A3. We therefore also explain how to use some primitives of the Virtual Tree Processor interface.

6.1. Coercing LISP trees to Prolog terms

At the present time, one of the most frequent operations in the environment we are building is a coercion from Virtual Tree objects to Prolog terms. Basically this primitive does a tree traversal and calls appropriate C-Prolog constructors on every node of the tree. A Prolog term being either an atom or a functor, we use only two C-Prolog constructors:

({prolog}:lookup <string>) -> <entry>

returns the entry point associated with an atom. This entry point, external to the LE_LISP system, has the type *t* of the C-LE_LISP interface.

({prolog}:apply <entry> <nargs> <entries>) -> <entry>

returns the entry point associated with a functor. The first parameter is the entry associated with the name of the functor, the second one is the arity of the functor, and the last one is an array or vector of entries associated to each sub-term.

Following the general principles of the Virtual Tree Processor, we defined the coercion primitive as a new primitive, {tree}:prolog associated with class tree.

```
(de {tree}:prolog (tree)
   (lets ((operator ({tree}:operator tree))
          (node-name (catenate ({formalism}:name
                                       ({operator}:formalism operator))
                               "$"
                               ({operator}:name operator)))
          (vector (if (numberp ({operator}:arity operator))
                      (makearray ({operator}:arity operator) ())
                      (makearray 2 ())
                  );end if
          )
         )
         (selectq ({operator}:arity operator)
           (0 (classcall ({operator}:class operator) 'prolog tree)
           ) ; end of nullary operator
           ((+ *) ({tree}:list-prolog tree node-name vector)
           ) ; end of list+ or list* operator
           (t ({tree}:node-prolog tree node-name vector)
           ) ; end of fixed arity operator
         )
      ) ; end let
) ; end of {tree}:prolog
```

For trees of arbitrary arity, i.e. lists, we use an auxiliary primitive. The two functions {tree}:head and {tree}:tail return respectively the first element of a list and the tail of a list.

```
(de {tree}:list-prolog (tree node-name vector)
    (vset vector 0 ({tree}:prolog ({tree}:head tree)))
    (if (eq 1 ({tree}:sons-number tree))
        ; the list has only one element
          (vset vector 1 ({prolog}:apply
                                             ({prolog}:lookup node-name) 1
                                             (vector ({prolog}:lookup "nil"))))
        ; else the list has more than one element
        (vset vector 1 ({tree}:prolog ({tree}:tail tree)))
    );endif
    ({prolog}:apply ({prolog}:lookup node-name) 2 vector)
);end {tree}:list-prolog
```

An auxiliary function for trees of fixed arity is:

```
(de {tree}:node-prolog (tree node-name vector)
    (for (pos 0 1 (-1 ({tree}:arity tree)))
       (vset vector pos ({tree}:prolog ({tree}:down tree pos)))
    ); endfor
    ({prolog}:apply ({prolog}:lookup node-name)
                    ({tree}:arity tree) vector)
);end {tree}:node-prolog
```

6.2. Efficiency considerations

Although the code for the {tree}:prolog function is transparent, some remarks are in order:

1) Using the LE_LISP function *catenate* for building the name associated with an operator belonging to a formalism leads to the generation of strings in the LE_LISP string area. The garbage collector will be called every time this area becomes full. This technique is undoubtedly wasteful.

2) The use of the LE_LISP function *makevector* for getting a local vector leads also to frequent calls of the garbage collector.

3) The use of primitives such as {tree}:down is not well suited to the writing of our coercion function.

To obtain a faster version, we find a solution for each of the three previous problems.

1) Instead of using *catenate*, define an auxiliary C function which always uses the same string area.

   ({prolog}:functor lang:<string> operator-name:<string>) -> <string>

2) Define a global vector of size N. N should correspond to the maximum arity of operators in a formalism. All parameters to C functions are deposited in this global vector.

3) Use the general purpose {tree}:for-all-sons iterator.

Furthermore, the name of the formalism of a V.T.P tree obtained by ({tree}:formalism tree) is given as a parameter to avoid recomputing it for each operator. If the type of a nullary operator is not a symbol, string, integer, or singleton, the coercion function from objects of that type to Prolog terms must be called (without the name of the current formalism, which may be different).

```
(de {tree}:prolog (lang tree)
   (let ((operator ({tree}:operator tree))
        )
        (selectq ({operator}:arity operator)
           (0 (selectq ({operator}:class operator)
               ; fast version of classcall
                  (symbol ({symbol}:prolog lang tree))
                  (string  ({string}:prolog lang tree))
                  (integer ({integer}:prolog lang tree))
                  (singleton ({singleton}:prolog lang tree))
                  (t (classcall ({operator}:class operator)
                           'prolog tree))
              );endselectq
           ) ; end of nullary operator
           ((+ *) ({tree}:list-prolog lang
                                      tree
                                      ({prolog}:functor lang
                                           ({operator}:name operator)))
           ) ; end of list+ or list* operator
           (t ({tree}:node-prolog lang ({operator}:arity operator)
                                  tree
                                  ({prolog}:functor lang
                                       ({operator}:name operator))
               )
           ) ; end of fixed arity operator
        )
   ) ; end let
) ; end of {tree}:prolog
```

To build the Prolog term associated with a list, the list traversal has to be done backwards. The last parameter of the general purpose function {tree}:for-all-sons is given as *backward.*

```
(de {tree}:list-prolog (lang tree node-name)
    (let ((first 't)
          (prev ())
          (s1 ())
          (s2 ())
         )
     ({tree}:for-all-sons tree
                 (lambda (son)
                       (setq s1 ({tree}:prolog lang son))
                       (setq s2
                       (if first
                          ({prolog}:apply node-name 1 {prolog}:nil)
                          prev))
                       (vset {prolog}:globalvector 0 s1)
                       (vset {prolog}:globalvector 1 s2)
                       (setq prev ({prolog}:apply node-name 2 {prolog}:globalvector))
                       (setq first nil)
                 )
                 backward)
       prev
))
```

For fixed arity trees, the size of the global vector must be greater than or equal to the arity of the tree. Otherwise a local vector is used.

```
(de {tree}:node-prolog (lang arity tree node-name)
    (let ((s0) (s1)
          (count 0))
         (if (le arity 3)
           (progn
              ({tree}:for-all-sons tree
               (lambda (son)
                (selectq count
                 (0 (setq s0 ({tree}:prolog lang son)))
                 (1 (setq s1 ({tree}:prolog lang son)))
                 (2 (vset {prolog}:globalvector 2 ({tree}:prolog lang son)))
                )
                (incr count)))
              (vset {prolog}:globalvector 0 s0)
              (vset {prolog}:globalvector 1 s1)
              ({prolog}:apply node-name arity {prolog}:globalvector)) ;end progn
            ; else arity > 3
              (setq s1 (makevector arity ())) ; we use s1 as local vector
              ({tree}:for-all-sons tree
               (lambda (son)
                  (vset s1 count ({tree}:prolog lang son))
                  (incr count)))
              ({prolog}:apply node-name arity s1)
         );end if arity
    );endlet
);end {tree}:node-prolog
```

6.3. Coercion primitives for basic classes

For every basic class of the Virtual Tree Processor, and ultimately for every user defined class, there must exist a coercion primitive.

Class Symbol

```
(de {symbol}:prolog (lang tree)
    (ifn ({tree}:atom_value tree)
         ({prolog}:functor lang ({operator}:name ({tree}:operator tree)))
         (vset {prolog}:globalvector 0 ({prolog}:lookup ({tree}:atom_value tree)))
         ({prolog}:apply ({prolog}:functor lang
                                       ({operator}:name ({tree}:operator tree)))
                         1
                         {prolog}:globalvector)
   );endifn
)
```

Class String

```
(de {string}:prolog (lang tree)
    (if (eq 0 (slength ({tree}:atom_value tree)))
       ({prolog}:functor lang ({operator}:name ({tree}:operator tree)))
       (vset {prolog}:globalvector 0
                                  ({prolog}:lookup ({tree}:atom_value tree)))
```

```
        ({prolog}:apply ({prolog}:functor lang
                                            ({operator}:name ({tree}:operator tree)))
                        1
                        {prolog}:globalvector)
    );endif
)
```

Class Char

```
(de {char}:prolog (lang tree)
    (if (eq 0 (slength ({tree}:atom_value tree)))
        ({prolog}:functor lang ({operator}:name ({tree}:operator tree)))
        (vset {prolog}:globalvector 0
                                          ({prolog}:lookup ({tree}:atom_value tree)))
        ({prolog}:apply ({prolog}:functor lang
                                            ({operator}:name ({tree}:operator tree)))
                        1
                        {prolog}:globalvector)
    );endif
)
```

Class Integer

```
(de {integer}:prolog (lang tree)
    (vset {prolog}:globalvector 0 ({prolog}:consint ({integer}:fix
                                      ({integer}:name ({tree}:atom_value tree)))
                                  ))
    ({prolog}:apply ({prolog}:functor lang
                                            ({operator}:name ({tree}:operator tree)))
                    1
                    {prolog}:globalvector)
)
```

Class Fix

```
(de {fix}:prolog (lang tree)
  (vset {prolog}:globalvector 0 ({prolog}:consint ({tree}:atom_value tree)))
  ({prolog}:apply ({prolog}:functor lang
                                          ({operator}:name ({tree}:operator tree)))
                  1
                  {prolog}:globalvector)
)
```

Class Singleton

```
(de {singleton}:prolog (lang tree)
    ({prolog}:functor lang ({operator}:name ({tree}:operator tree)))
)
```

GIPE : CEC 348/S/T2/3
version 2.1 December 1985

Interfaces between LISP and Ash

Deliverable D2.A2 of Task T2

M. Devin (INRIA)
L. Gallot (INRIA)

1. Introduction

Ash provides its own interface [1], so that the situation is fairly typical. The first step is accomplished by linking Ash and LE_LISP codes either with static linking or dynamic linking. Notice that the use of static linking provides a less time consuming call of the whole system Ash-LE_LISP. Then to realise an effective connection one has to map Ash entry points to LE_LISP functions. This is done with the DEFASH primitive of LeLISP (which uses a built-in primitive DEFEXTERN [2]).

Thus the DEFASH function is the essential part of our interface. Any call to DEFASH takes the following form:

(DEFASH <name> (<type1>...<typen>) <restype>)

where

<**restype**> is optional and should be omitted when the ASH primitive is a procedure.

<**name**> is the name of the entry point which is to be mapped. This entry point should not begin with the four letters "_ASH". Those are provided automatically.

(<**typ1**> ... <**typn**>) are N type descriptors, describing the type of the corresponding arguments of the primitive. If the primitive has no arguments this should be the empty list.

<**restype**> is the type of the result returned by the entry point.

The call returns the name of the LISP function mapping the C entry point. This name is a symbol in the *ash* package. For example *#:ash:create* corresponds to the ASH entry point *ASHcreate.*

Type descriptors can be chosen from among the following symbols:
- fix (for 16-bits fixnums)
- float (for reals)
- string
- vector (for vectors of fixnums)
- external (for external pointers)
- t (for LISP pointers).

The correspondence between C-types and LISP descriptors is as follows:

LISP descriptor	*C type*
fix	int (but 16-bits masked)
float	double
string	char *
vector	int * (elements are 16-bits masked)
external	char *, WINDOW *, (any pointer)
t	int *, (any pointer)

The *external* type descriptor will be used for objects that are outside the LISP world, such as WINDOW pointers returned by the ASHcreate primitive.

The *t* type descriptor will be used for LISP objects which must be given by reference to C routines (see the ASHhit declaration below).

1.1. Selected examples

We explain below how to use DEFASH with the most important primitives. The complete list of Ash primitives is given in the Appendix.

The windows are organised into a hierarchy, i.e. windows may have subwindows, and so forth. Subwindows are displayed relative to their parents, but are otherwise independent.

To handle windows we have to create them. To represent the external view of a bitmap Ash defines the data type ASH_WINDOW and provides the routine to create such a window:

ASHcreate(parent_lx,parent_by,lx,by,rx,ty,border_id,flags) : <ASH_WINDOW>

(DEFASH create (fix fix fix fix fix fix fix fix) external)

To place this window we have to give the location of its lower left-hand corner within its parent window. This location will be at the point (parent_lx,parent_by) in the coordinate system of its parent window. The parameters lx, by, rx, and ty are the lower left-hand corner and the upper right-hand corner coordinates in the coordinate system of the created window.

The border_id parameter indicates what kind of border will be associated with the window. (For example, a border of type BORDER_NONE is an empty border.) Flags allow attaching a given attribute to the window. (For example WINDOW_VISIBLE will make the window immediately visible.) The list of symbolic constants such as BORDER_NONE and WINDOW_VISIBLE is given in the Appendix.

Once in the LE_LISP environment the following function call:

```
?(:create 10 500 0 400 400 0 :border_none :window_visible)
=(10 . 364)
```

will result in the display on the bitmap device of a window of 400*400 pixels located at the point (10,500) of its parent (the previous current window). This new window becomes the current window. The result (10 . 364) is just the LISP representation of an external pointer.

Many primitives use the notion of a *current* window. So another useful primitive makes the given window the current window.

ASHselect(window)

(DEFASH select (external))

This call defines the Lisp function *select* which takes one external pointer as argument and returns no significant result.

This function can be used in the following way:

```
?(setq f1
        (:create 10 500 0 400 400 0 :border-none :window-visible))
=(10 . 364)
?(setq f2
        (:create 100 100 0 50 400 0 :border-thin :window-visible))
=(10 . 452)
?(:select f1)
=1342
```

But we do not always know the top window on the screen at a given location. The routine ASHhit returns the hittable top window at the coordinate (x,y) if such a window exists.

ASHhit(x,y,&window_x,&window_y) : <ASH_WINDOW>

(DEFASH hit (fix fix t t) external)

The parameters window_x and window_y are pointers to locations. In case of success the coordinates of the hit in the returned window are placed in the corresponding locations. The Lisp function needs quoted symbols here, whose value will be modified by the call.

For example:

```
?(:hit 100 200 'x 'y)
=(10 . 364)
?x
= 54
?y
= 62
```

Now we know how to create windows and select them. But we can also output something on them, using for example ASHtext.

ASHtext(x,y,text)

(DEFASH text (fix fix string))

The given text string will be output at the given coordinates (x,y) in the current window.

1.2. Pop-Up menu implementation

We give an example of an implementation in Le_Lisp/Ceyx of a Pop-Up menu with manipulation primitives.

```
(defrecord button area action) ; a button is a sensitive area in a menu
                               ; an action is associated with it

(defrecord menu frame buttons~(List button)); a menu is a frame and a list
                                             ;of buttons

(defvar BUTTON-HEIGHT 20)     ; button's height in a menu
(defvar MENU-WIDTH 110)       ; button's width in a menu

(de {menu}:create (title button_names button_actions)
  (let((button_number (length button_names))) ; to compute the menu's  height
     (#:ash:push_window)           ; push current window in the windows' stack
     (#:ash:select(#:ash:inq_top))      ; select the screen
     (let ((pop-up (omakeq menu)))      ; create a new menu
          ({menu}:frame pop-up          ; new menu's frame
             (#:ash:create 0 0                       ; immaterial position
                       0                             ; left x
                  (* button_number BUTTON-HEIGHT)  ; bottom y: menu's height
                  MENU-WIDTH                         ; right x: menu's width
                       0                             ; top y
                  #:ash:border_window                ; border style
                  ( logor                            ; attributes
                    #:ash:window_invisible           ; keep the menu invisible
                    #:ash:window_hit_use)            ; make the menu hitable
                  ))
             (#:ash:set_window_name title)    ; title will appear in the band
                                              ; at the top of the menu
             (for (i 1 1 button_number)              ; create all buttons
                 (let ((button (omakeq button)))     ; a new button
                      ({button}:area button          ; a button is a
                        (#:ash:sensitive_area        ; rectangle  sensitive
                                                     ; area in the menu
                           1 (* i BUTTON-HEIGHT)     ; bottom left
                           (1- MENU-WIDTH)           ; right
                           (* (- i 1) BUTTON-HEIGHT) ; top
                           1                         ; invert when hit
                           ))
                      ({button}:action button (nextl button_actions))
                      ({menu}:buttons pop-up          ; add new button
                        (cons button ({menu}:buttons pop-up))     ; to list
                      )
                      (#:ash:center_text
                          (nextl button_names)        ; text string in button
                          1                           ; rectangle area
                          (1- (* i BUTTON-HEIGHT))    ;    to center
                          (1- MENU-WIDTH)             ;    button's name
                          (+ 1 (* (- i 1) BUTTON-HEIGHT))
                      )
                      (#:ash:line 0                ; to separate the  buttons
                                (* i BUTTON-HEIGHT)
                                MENU-WIDTH
                                (* i BUTTON-HEIGHT))

                 );endlet
             );endfor
```

```
            (#:ash:pop_window) ; get current window at the top of the
                               ; windows' stack
            pop-up)            ; return the new menu
    ); endlet
)
```

The "{menu}:create" function allows the user to create a pop-up menu. A menu is a window and a set of buttons. Each button is associated with an action, which is invoked when the button is selected. A pop-up menu is a menu that appears temporarily on the screen and then disappears. The arguments of this function are:

- the title string
- the list of button names
- the list of actions associated with the buttons.

The function creates a Ceyx object, the type of which is "menu"; it is a record with two fields:

- the frame field is an ASH_WINDOW
- the buttons field is the list of the created menu's buttons.

Each button consists of a rectangular area in the menu's window and an action. The function returns the created menu, which remains invisible and will only appear when "{menu}:visible" is called.

```
(de {menu}:visible (menu posx posy)         ; make menu visible at (posx, posy)
  (let ((button_number
        (length ({menu}:buttons menu))))    ; how many buttons in the menu ?
      (#:ash:push_window)                   ; push current window
      (#:ash:select({menu}:frame menu))     ; select the menu's frame
      (#:ash:view posx posy                 ; menu's position in the screen
      0 (* button_number BUTTON-HEIGHT) MENU-WIDTH 0) ; menu's viewing area
      (#:ash:visible 1)                     ; to make the menu visible
      (#:ash:pop_window))                   ; get current window
); end
```

To make a pop-up menu visible on the screen one calls the function "{menu}:visible". Its arguments are:

- the menu one wants to make visible (it is the result of an earlier call of the {menu}:create function).
- the x and y coordinates on the screen for placing the menu's bottom left-hand point.

The menu will remain on the the screen until the user chooses a button in it or cancels the menu's call. Suppose, for example, we are working on a system with a mouse. We can call the {mouse}:visible function by pushing down the (a) button of the mouse in a given area of the screen. The menu will appear at the location pointed to by the mouse. Then one can track the mouse until its button is released and at that time call the {menu}:click function which detects which (if any) button has been chosen and applies the action associated with it.

```
(de {menu}:click (menu posx posy)        ; current menu and position on screen
    (let (( hit_window
          (#:ash:hit posx posy 0 0)))    ; window hit at (posx,posy)
        (if(equal hit_window ({menu}:frame menu))   ; the hit window is
                                                    ; the current menu ?
          (let((hit_button (#:ash:inq_sensitive))   ; which area is hit ?
                 (l ({menu}:buttons menu))          ; menu's buttons' list
                 (bl)
                )
                (while l                            ; visit buttons' list
                       (setq bl (nextl l))          ; current button
                       (if(equal ({button}:area bl)
                                 hit_button)        ; is current button hit?
                         (funcall({button}:action bl)); if so: apply current
                                                      ; buttons' action

                       ); endif
                ); endwhile
             ); endlet
          ); endif
        ); endlet
        (#:ash:push_window)                           ; push current window
        (#:ash:select({menu}:frame menu))             ; select current menu
        (#:ash:visible 0)                             ; make it invisible
        (#:ash:pop_window)                            ; pop current window
); end
```

The function {menu}:click is called with the following arguments

- a menu (the latest pop-up menu appearing on the screen for example)
- a location on the screen (the location of the mouse when its button has been released).

This function first decides if the given location lies inside the menu. If so, it then determines what button is selected and applies the associated action. Finally, in all cases (whether the menu is pointed to or not), the function makes the menu invisible.

1.3. Locator device

The locator device is managed by LE_LISP as a soft interrupt called each time the user clicks the mouse, for example. In the present system this interrupt has to be raised by a physical action on the locator device. This could be changed later on, however.

To handle the device the user has to define its own *mouse* function with two arguments: the location pointed at and the button number. Here is a (strictly pedagogical) example:

```
(de mouse (location button)
    (let ((x) (y))
         (:select (:hit (car location) (cdr location) 'x 'y))
         (:text x y "You hit me")))
```

2. References

[1] *LE_LISP & CEYX*

- J. Chailloux, "LE_LISP de l'INRIA, Le Manuel de Reference" (version 15), INRIA Report (to be published), February 1985.
- J.M. Hullot, "CEYX - Version 15, II: Programmer en CEYX," INRIA Technical Report no. 45, February 1985.

[2] *BWE*

- J.N. Pato, S.P. Reiss, M.H. Brown, "The Brown workstation environment," Brown University CS-84-03, October 1983.

3. APPENDIX: ASH

We give below a complete listing of ASH constants and ASH entry points as declared in the file *defash.ll* that is provided with the AshLe_Lisp system.

In order to use them one should only type
```
? (libload defash)
```
under the AshLe_Lisp top level.

We also provide some examples and demos that can be run with the function
```
? (ashdemo)
```

3.1. ASH CONSTANTS

3.1.1. WINDOW BORDERS

```
(defvar #:ash:border_none       0)
(defvar #:ash:border_thin       1)
(defvar #:ash:border_window     2)
(defvar #:ash:border_sensitive  3)
(defvar #:ash:border_tab        4)
(defvar #:ash:border_tabsense   5)
(defvar #:ash:border_flip       6)
```

3.1.2. TYPES OF WINDOWS

```
(defvar #:ash:window_visible        #$1)
(defvar #:ash:window_invisible      #$2)
(defvar #:ash:window_hit_use        #$4)
(defvar #:ash:window_hit_parent     #$8)
(defvar #:ash:window_frame         #$10)
(defvar #:ash:window_courteous     #$20)
(defvar #:ash:window_nosave        #$40)
(defvar #:ash:window_dependent     #$80)
(defvar #:ash:window_transparent  #$100)
(defvar #:ash:window_independent  #$200)
```

3.1.3. LINE STYLES

```
(defvar #:ash:solidl    0)
(defvar #:ash:blank     1)
(defvar #:ash:dotted    2)
(defvar #:ash:dashed    3)
(defvar #:ash:dotdash   4)
(defvar #:ash:double    5)
```

3.1.4. FILLING MODES

```
(defvar #:ash:solid0   10)
(defvar #:ash:pattern0 20)
(defvar #:ash:patternl 21)
```

```
(defvar #:ash:pattern2 22)
(defvar #:ash:pattern3 23)
(defvar #:ash:pattern4 24)
(defvar #:ash:pattern5 25)
(defvar #:ash:pattern6 26)
(defvar #:ash:pattern7 27)
(defvar #:ash:pattern8 28)
(defvar #:ash:pattern9 29)
(defvar #:ash:hatch0   30)
(defvar #:ash:hatch1   31)
(defvar #:ash:hatch2   32)
(defvar #:ash:hatch3   33)
(defvar #:ash:hatch4   34)
(defvar #:ash:hatch5   35)
(defvar #:ash:hatch6   36)
(defvar #:ash:hatch7   37)
(defvar #:ash:hatch8   38)
(defvar #:ash:hatch9   39)
```

3.2. ASH ENTRY POINTS

All calls to DEFASH define functions in the ASH package. The definitions are in the same order as in the ASH documentation.

```
(defash init (fix))
(defash trace (fix))
(defash push_state ())
(defash pop_state ())
(defash create (fix fix fix fix fix fix fix fix) external)
(defash select (external))
(defash visible (fix))
(defash pop ())
(defash push ())
(defash uncover (fix fix))
(defash find_window (fix fix) external)
(defash inq_under (external fix fix) external)
(defash inq_rectangle (fix fix t t t t) external)
(defash inq_region_visible (external fix fix fix fix) fix)
(defash hitable (fix))
(defash hit (fix fix t t) external)
(defash hitwindow (external))
(defash map (external fix fix external t t))
(defash remove (external))
(defash view (fix fix fix fix fix fix))
(defash newview (external fix fix fix fix fix fix fix fix) external)
(defash newframe ())
(defash resize (fix fix fix fix))
(defash par_resize (fix fix fix fix fix fix))
(defash quickmove (fix))
(defash inq_size (fix t t t t))
(defash inq_top () external)
(defash inq_window () external)
(defash inq_parent ())
```

```
(defash inq_border_size (fix t t t t))
(defash set_window_name (string))
(defash inq_window_name () string)
(defash set_user_data (t))
(defash inq_user_data () t)
(defash push_window ())
(defash pop_window ())
(defash line (fix fix fix fix))
(defash polyline (fix vector vector))
(defash point (fix fix))
(defash polypoint (fix vector vector))
(defash convex_polygon (fix vector vector))
(defash general_polygon (fix vector vector))
(defash rectangle (fix fix fix fix))
(defash round_rectangle (fix fix fix fix fix))
(defash box (fix fix fix fix))
(defash round_box (fix fix fix fix fix))
(defash circle (fix fix fix))
(defash filled_circle (fix fix fix))
(defash ellipse (fix fix fix fix))
(defash filled_ellipse (fix fix fix fix))
(defash clear ())
(defash text (fix fix string))
(defash center_text (string fix fix fix fix))
(defash fill (fix))
(defash line_style (fix))
(defash combination_rule (fix))
(defash font (fix))
(defash clip (fix))
(defash clip_region (fix fix fix fix))
(defash inq_blt (fix fix fix fix fix fix) fix)
(defash blt (fix fix fix fix fix fix))
(defash zoom_blt (fix fix fix fix fix fix fix fix fix fix))
(defash read_pixels (fix fix fix fix external))
(defash write_pixels (fix fix fix fix external))
(defash save_bitmap (string))
(defash load_bitmap (string))
(defash source (external))
(defash loadfont (string) fix)
(defash inq_text_extent (string t t))
(defash inq_text_offset (string t t))
(defash inq_text_next (string t t))
(defash inq_font_info (string t t t string string string) fix)
(defash inq_font () fix)
(defash inq_fill () fix)
(defash inq_line_style () fix)
(defash inq_combination_rule () fix)
(defash push_drawinfo ())
(defash pop_draw_info ())
(defash copy_draw_info (external))
(defash cursor_move (fix fix))
(defash cursor (fix))
(defash cursor_load (fix))
(defash cursor_restore ())
(defash inq_cursor (t) fix)
```

```
(defash push_cursor (fix))
(defash pop_cursor ())
(defash cursor_define (fix fix fix fix fix fix))
(defash sensitive_area (fix fix fix fix fix) fix)
(defash sensitive_remove (fix))
(defash sensitive_remove_all ())
(defash inq_sensitive () fix)
(defash bell ())
(defash terminate ())
```

3.3. ASH DEMOS

Below we give the listing of some AshLe_Lisp programs to show how to use ASH from within Lisp. Every identifier beginning with a colon is in the ASH package.

3.3.1. BITBLIT DEMO

```
(de :demo:blit ()
    (let ((x 30) (y 30) (l 90) (pas 5) (maxpas 12))
         (:demo:blit:init x y)
         (repeat 4
            (repeat 30 (:demo:blit:show x y (incr x pas) y))
            (repeat 30 (:demo:blit:show x y x (incr y pas)))
            (repeat 30 (:demo:blit:show x y (decr x pas) y))
            (repeat 30 (:demo:blit:show x y x (decr y pas)))
            (incr pas 2))))

(de :demo:blit:init (x y)
    (:combination_rule 0)
    (:blt 0 (+ 1 maxpas maxpas) (+ 1 maxpas maxpas) 0
          (- x maxpas) (+ y 1 maxpas ))
    (:fill 23)
    (:combination_rule 3)
    (:rectangle x y (+ x 1) (+ y 1)))

(de :demo:blit:show (ox oy x y)
    (:blt (- ox pas) (+ oy 1 pas) (+ ox 1 pas) (- oy pas)
          (- x pas) (+ y 1 pas)))
```

3.3.2. RECTANGLE FILLING DEMO

```
(de :demo:tunnel ()
    (:combination_rule 3)
    (let ((x1 210) (y1 175) (x2 750) (y2 525) (patt 20))
      (:clear)
         (while (and (< x1 x2) (< y1 y2))
                (:fill (incr patt))
                (:rectangle x1 y1 x2 y2)
                (incr x1 20)
          (incr y1 15)
          (decr x2 20)
          (decr y2 15))
    ))
```

3.3.3. POLYLINES DEMO

```
(de :demo:lignes (s)
    (:clear)
    (:line_style s)
    (let ((x (makevector 50 0))
          (y (makevector 50 0))
          (zx (makevector 3 0))
          (zy (makevector 3 0)))
         (vset x 0 0)
         (vset y 0 350)
         (for (i 1 1 (sub1 (vlength x)))
              (vset x i (min (add (vref x (sub1 i)) (random 0 50))
                             1024))
              (vset y i (rem (add (vref y 0) (random -50 51)) 789)))
         (:polyline (vlength x) x y)
         (repeat (quo (vlength x) 5)
              (let ((i (mul 2 (random 1 (div (sub1 (vlength y)) 2)))))
                   (bltvector zx 0 x (sub1 i) 3)
                   (bltvector zy 0 y (sub1 i) 3)
                   (vset zy 1
                         (rem (sub (vref zy 1) (random 150 400)) 789))
                   (repeat 10
                      (vset zy 1
                            (rem (add (vref zy 1) (random 30 80)) 789))
                      (:polyline 3 zx zy))))))
```

3.3.4. RANDOM RECTANGLES AND BOXES DRAWING

```
(de :demo:rect (n type rule)
     (repeat n
        (:combination_rule rule)
        (repeat 8
           (let ((x (random -100 800))
                 (y (random 100 700)))
                (repeat (random 3 7)
                   (let ((lx (add x (random 0 50)))
                         (rx (add x (random 50 200)))
                         (ty (sub y (random 20 200))))
                        (cond ((eq type 'rectangle)
                               (:fill :pattern0)
                               (:rectangle (sub lx 5) (sub ty 5) (sub rx 5) y)
                               (:fill (random 21 38))
                               (:rectangle lx ty rx y))
                              ((eq type 'box)
                               (:box (add lx 5) y (add rx 5) (sub ty 5))
                               (:box lx y rx ty))))
                   (incr x (random 50 100)))))
        (repeat 2 (repeat 32767))
     (tycls)
        (:combination_rule 3)))
```

3.3.5. POLYLINES DRAWING

```
(de :demo:traits ()
    (:push_window)
    (let  ((f1 (progn (:select (:inq_top))
                      (:create 90 600 0 0 400 400 1 1)))
           (f2 (progn (:select (:inq_top))
                      (:create 440 650 0 0 400 400 2 1))))
         (protect
            (progn
               (for (i 0 1 60)
                    (:select f1)
                    (:line 0 (* i 10) 400 (- 400 (* i 10)))
                    (:select f2)
                    (:line (* i 10) 0 (- 400 (* i 10)) 400))
               (:center_text "Que Le_ASH soit avec vous!"
                             50 200 350 220)
               (repeat 6
                   (repeat 10000)
                   (:select f1)
                   (:pop)
                   (repeat 10000)
                   (:select f2)
                   (:pop)))
            (:remove f1)
            (:remove f2)
            (:pop_window))))
```

3.3.6. DEPENDENT WINDOWS, MOVING WINDOWS

```
(de :demo:coeur (pas)
    (unless (fixp pas)
            (syserror ':demo:coeur 'errnia pas))
    (:push_window)
    (let ((w (progn (:select (:inq_top))
                    (:create 100 600 0 0 400 400 2 1))))
         (protect
          (progn
           (:set_window_name "LE_LISP + ASH")
           (for (i 30 30 400)
                (:line 0 i 400 i)
                (:line i 0 i 400))
      (:center_text "Matthieu + Laurence" 100 180 300 220)
         (let ((x (:create 100 100 0 0 200 200 0 1)))
             (protect
               (progn
               (for (i 0 5 205)
                    (:line 0 i 200 (+ 200 i))
                    (:line 0 i 200 (- 200 i)))
               (let ((boutx 300) (bouty 300))
                (let ((dx)(dy)(x 100)(y 100))
                     (while t
                          (setq dx (random (- pas) (1+ pas))
                                dy (random  (- pas) (1+ pas)))
(while (< (add (abs dx) (abs dy)) pas)
                (setq dx (random (- pas) (1+ pas))
                      dy (random  (- pas) (1+ pas))))
         (untilexit bump
               (setq x (+ x dx)
                     y (+ y dy))
               (cond
                  ((<= x -50)
                   (setq x -50)
                   (exit bump))
                  ((>= x (add boutx 50))
                   (setq x (add boutx 50))
                   (exit bump))
                  ((<= y 0)
                   (setq y 0)
                   (exit bump))
                  ((>= y bouty)
                   (setq y bouty)
                   (exit bump)))
              (:view  x y 50 50 150 150))))))
   (:remove x))))
 (:remove w)
(:pop_window))))
```

GIPE : CEC 348/R/T2/2
version 2.1 February 1986

The Virtual Tree Processor

Deliverable D2.A3 of Task T2

B. Lang (INRIA)

1. INTRODUCTION

The organization of the Virtual Tree Processor (VTP) is object oriented, i.e. it is structured as a collection of **classes** (in the Simula-67 or Smalltalk sense).

Each class is characterized by:

- an abstract domain of values (either pure values or modifiable objects).
- a collection of primitive functions and procedures operating on the values of the class, either alone or in conjunction with values of other classes.
- a specification of an internal representation (in remanent or persistant memory) of the values belonging to the class.
- a unique **class name**.

The internal representation in central memory of the values belonging to the class is not part of the specification, and it may change from implementation to implementation. However, the existing implementation in LE_LISP and CEYX will be sometimes mentioned since it shares some of the abstract concepts and notations on which the specification was built (and remains very close to the specification), and also as a basis for efficiency considerations.

The remanent representations of the values belonging to a class have to be part of the specification of that class in order to preserve compatibility between distinct implementations of the VTP. However, knowledge of these representations is needed only by implementors of a new VTP implementation, and by users of the VTP who define new classes with remanent values. The remanent representations for existing classes in the VTP will be described in a separate document, together with the techniques to be used to define consistently the remanent representations for new classes.

A class could be specified formally by any appropriate mathematical technique (e.g. abstract algebras), but we have not attempted to do this here. Only an informal description in English is given for each class (completed by its actual implementation in LE_LISP and CEYX).

The existence of two representations (central and remanent) of the values of a class implies the implementation of two functions that perform the translation between these two representations (see functions *save* and *restore*).

Although a textual representation of the values of each class is usually provided (cf. the functions *name* and *write* in class schema *universal*), it is more regarded as a convenience for developers than as a standard since we do not wish to make assumptions about the user interface.

The primitive functions provided for each class are usually very redundant. This is intended both as a convenience for the user of the VTP and as a simple means of achieving a better code efficiency. For example the function *{tree}:change_son* is redundant with the combination of *{tree}:down* and *{tree}:replace*, but the use of the former function is faster than the use of the latter two. We hope to be able to provide means to handle automatically these efficiency problems in the future, but this will require much more complex developments.

1.1. Implementation techniques

A section like this should not appear in a specification. However, the present specification has been expressed in a notation that is close to the language being used for the existing implementation of the VTP. We hope that this will make the specification more readily usable, and also easier to read because of the small amount of new notations and conventions. It is also true that the potential practical uses of the VTP are more apparent with some knowledge of its implementation.

The implementation of the VTP has been done in LE_LISP and CEYX. Knowledge of these two languages is not strictly necessary to read this document, but it is nevertheless helpful (and it cannot be dispensed with if one intends to actually use the VTP). Explanations are provided below concerning the notations that are unusual with respect to classical Lisp's.

The classes of the VTP are untagged CEYX classes, i.e. essentially a structure definition (the central memory representation) and a collection of primitive functions placed in a LE_LISP package.

For each class, the name of the associated LE_LISP package is the name of the class. Thus, following the syntax of LE_LISP and CEYX, a function *foo* defined for the class *myclass* is actually designated by *{myclass}:foo.* The syntax of a function call is the traditional syntax of Lisp, for example:

```
({myclass}:foo arg1 arg2 arg3)
```

if {myclass}:foo takes three arguments. In this case, the first argument, if it exists, is expected to belong to the class *myclass*, but this is not compulsory.

Classes are themselves values belonging to the class called **class**. The class of the value of a variable *xx* may be itself a run-time value contained in a variable *cc*. To apply the function *foo* (which may be defined for many classes) to the value of *xx*, one may use the special function **{class}:call** as follows:

```
({class}:call cc 'foo xx arg2 arg3)
```

If the current value of *cc* is the class called *myclass*, then the above call is equivalent to

```
({myclass}:foo xx arg2 arg3)
```

We note that, when using *{class}:call*, the name of the primitive function could itself be a computed value, since it has to be quoted when given explicitly.

The CEYX definition of the central memory representation of the values belonging to a class is not given here since it may be changed without notice. Thus users of the VTP should rely exclusively on the functions and other primitives entities given in the present specification, for which upward compatibility will be guaranteed (unless the contrary is explicitly mentioned).

The knowledgeable reader will remark that very little is used of the existing object oriented facilities available in CEYX. In particular we do not use tagged objects, i.e. objects that carry their class with them at run-time. The reason is that, although the systematic use of tagged objects is considerably easier and safer, it entails a space overhead (and to a lesser extent a time overhead) that has been considered unacceptable due to the very large size of the objects we intend to manipulate (e.g. the text of a complete program).

The user of the VTP must be able to determine from the context of its use, or from the way it was computed, the class of any value manipulated. Possibly the class itself is computed and stored in some location. For circumstances where tagged values are necessary, a special class **tagval** is provided. A value of class *tagval* is essentially a pair consisting of a value belonging to any class and a value representing that class. The function *foo* of its class may be applied to the untagged value contained in the *tagval* value of a variable *xx* with the following call:

```
({tagval}:call 'foo xx arg1 arg2)
```

Thus any value of any class may be used both in untagged and in tagged form (see section on class *tagval* for more details). This combination is not permitted by the standard CEYX constructions at the time of this writing.

It may also be noted that the inheritance mechanisms of CEYX are not used (except may be as an implementation device). In the specification and the use of the VTP, inheritance and subclassing appear only as abstract concepts with class schemata (see below), but not as actual operational devices.

1.2. Class Schemata

In the realization of the VTP, we have chosen not to use concepts such as subclassing and inheritance which are not yet cleanly defined and implemented in available languages. They are, however, useful concepts for the specification of the VTP since they offer a means of factorizing common aspects of differents classes (though the factorization cannot be carried trivially into the implementation). Thus we introduce for the purpose of this specification the concept of a **class schema.**

A class schema is abstractedly similar to a class, the main difference being the absence of a specific implementation model. In fact a class schema may have several implementation models available at the same time, each corresponding to a different class. A class is said to be an **instance** of a class schema if it is an implementation model of the class schema.

Since the present specification is rather informal, we only mean by an implementation model of a class schema a class that has all the primitive operations of the class schema, with the same properties (informally) defined in English. Following the usually accepted terminology, we shall say that an instance of a class schema **inherits** all primitives and properties of the class schema. A more formal approach would also associate more formally defined properties (axioms) to both classes and class schemata, and require the preservation of the properties through some appropriately defined homomorphisms between them.

A class schema S1 is a **subclass** of a class schema S2 if and only if all instances of S1 are instances of S2. We also say that S2 is a **superclass** of S1. We will also say that a class is a subclass of a class schema if and only if it is an instance of that class schema.

We will sometimes keep in a class schema a primitive that, under the same name, has a somewhat different (though related) behaviour in the various instances of the class schema. This is merely to have an opportunity to underline the differences.

For each class specified in this document, we indicate under the heading *"Superclasses:"* the class schemata of which it is an instance. The primitives already described in the class schemata are not described again in the instance class, unless additional information specific to the instance class is to be supplied.

1.3. Exceptions

Calls to the functions of the VTP may fail, for example when requesting a computation that is not meaningful with the supplied data (e.g. requesting the first son of an atomic node of a tree), or when requesting a modification of the manipulated structures that would cause them to become inconsistent (e.g. a tree transformation not consistent with the syntax of the represented language).

Signaling and handling such failures is done via the LE_LISP exception primitives such as *tag*, *exit*, *lock* and others.

A failure tag is associated to each such failure of the VTP functions. These tags are specified in the present document, with the following two restrictions:

- the choice of tag names may change in future versions,
- the type of the value associated with the tag when an exception is raised, is not yet specified.

These restrictions are motivated by the fact that the impact of the organization of exception names is not yet explored fully enough to make a final commitment.

1.4. Notations

Each class or class schema is specified in a separate section of this document, which is devoted exclusively to this class or class schema. Each such section has the title **Class** or **Class schema**, followed by the name of the class (schema) it specifies. The contents of the section describes (in English) the role of the class, i.e. the intended abstract domain, and the primitive functions (also called **semantic** functions) defined for the class, unless they have already been completely specified for a superclass. The specification of the internal representations in remanent memory of values belonging to the class will be specified in a future document.

Class names are often enclosed in angle brackets (e.g. <a_class>) when used alone, but the brackets may be omitted when there is no ambiguity. A semantic function called *foo* defined for the class <a_class> is denoted by the name of the class in braces, followed by a colon, followed the name of the function, as in the following example:

{a_class}:foo

However, *within this specification only,* the class prefix of a function name may be (and will usually be) omitted within the section devoted to that class. A function name without class prefix may either mean that the function is a semantic function of the class described in the current section, or that the function is a general utility function not attached to any class. The distinction will always be clear from the context. The class prefixes are required when the VTP functions are used in actual Lisp code (unless they are used by means of the function *{class}:call*).

A function description such as:

(**foo** a1:<class1> a2:<class2> [a3:<class3>]) --> <class4>

specifies a function named *foo*, or *{a_class}:foo* when it occurs in a section devoted to the specification of class *a_class*. This function takes two arguments belonging to classes <class1> and <class2>, a third optional argument belonging to class <class3> and returns a result that belongs to <class4>. The identifiers a1:, a2: and a3: are formal names for the arguments, and have no other role than helping to improve the readability of the specification. *Optional arguments* may be specified between square brackets, usually at the end of the argument list. In any other position, an optional argument that is omitted in a function call must be replaced by ().

If the last formal argument is followed by three dots "...", it will indicate that any non-null number of such arguments may be supplied in an actual call (see for example the function *{class}:call*).

If the above specification of the function *foo* appears in a section devoted to class <a_class>, it actually specifies a semantic function *{a_class}:foo* of that class. If it appears in a section devoted to a class schema, it specifies the semantic function for all class instances of that class schema. In any other case, it just specifies a normal Lisp function not attached to any class, and does not require a class prefix when used.

1.5. Conversions

To keep down the size of the terminology in the VTP, a function that may be considered as a conversion function from a class <class1> to a class <class2> is usually denoted by *{class1}:class2*. A common example is the function *name* defined for all classes, that returns a name, i.e. a <name> value, associated with each object of any class. Such a conversion function may take additional arguments, usually indicating the context in which the conversion is to be done. For example, the function *{name}:operator* converts a name into the operator bearing that name in the language supplied as second argument.

An obvious conversion function from class *class1* to class *class2* is often simply specified by mentioning *class2* after the heading *"Standard conversion to:"* at the beginning of the section devoted to the specification of *class1*.

2. BASIC CLASSES

This section contains all basic classes on top of which the VTP is constructed. The class schema *universal* describes the properties common to all classes. The class *class* allows manipulation of classes themselves as objects. The class *tagval* is used for run-time manipulation of values tagged with their class. All other classes defined here are the usual elementary classes of value, with some consideration for implementation and portability problems.

2.1. Class schema: universal

This class schema covers all classes. Thus the primitives defined in this class schema should exist for all classes to be defined. This will not be strictly true. All those primitives will be meaningful and implementable for all classes, but this implementation will be available only when deemed practically useful.

(**equal** u1:<universal> u2:<universal>) --> <boolean>

Tests if the two arguments are equal. Thus equality has to be explicitly defined for each class. Note that equality does not have to mean that both arguments are the same object. In particular, in classes of non pure values that may be modified, two equal objects may become unequal when one of them is modified.

(**copy** u:<universal>) --> <universal>

This function returns a copy of its argument. This copy must be equal to the argument in the sense of the primitive function *equal.* For classes of pure values, i.e. objects which cannot be modified, the function *copy* is the identity. When objects may be modified, the function *copy* returns an equal but different object.

(**name** u:<universal>) --> <name>

Returns the <name> of the arguments. This notion is class dependent. For classes with complex values, the result is often a <name> giving an abbreviated (incomplete) information about the argument value. In some cases it may even be just the name of the class to which the value belongs.

(**write** u:<universal>) --> <universal>

As a function, it is the identity. However it has the important side-effect of printing the value of its argument, in a class dependent format, on the current text output channel.

(**save** f:<file-name> u:<universal>) --> <universal>

As a function, it returns its second argument u:. However it has the important side-effect of storing this argument (in remanent internal representation, not in textual format) in the file designated by the first argument f:. This destroys any information that may have been previously in that file.

(**restore** f:<file-name> [p:<search-path>]) --> <universal>

The result is a value read from the file named f:, if the class of this value agrees with the class qualifying the function *restore.* For example, the call *({tree}:restore 'mytree)* must find a <tree> value in the file named 'mytree'. Values retrieved with the function *restore* must have been stored in the file (in remanent internal representation) with the function *save*. The optional argument p: denotes a search-path to be used when identifying the file named f:. Search-paths may not have been defined for all implementations.

The functions *save* and *restore* allow only one (possibly very complex) value to be stored in a file. Other primitives are intended to allow storing several values in succession on the same file. Higher level primitives for remanent objects are still very much in the research and design phase.

2.2. Class: class

Superclasses: universal.

This is the class of class values. Each class used in the system is itself (described by) an object belonging to the class <class>. Such an object may be dynamically computed and used. Application of a function according to a run-time computed class value for one of its arguments is performed by the function *{class}:call.*

({**name**}:**class** n:<name>) --> <class>

Each <class> has a unique name associated with it. This name is returned by the function *name*, or printed by the function *write*. The function *{name}:class* returns the class denoted by the name n: passed as argument.

(**call** c:<class> f:<symbol> [v:<any> ...]) --> <any>

The function *{class}:call* applies the function named by the Lisp symbol f:, as defined for the <class> c: to the subsequent list of arguments. The function f: must be a function that evaluates its arguments (i.e. not a fexpr nor a macro in Lisp terminology).

Two classes play a special role in the VTP:

- the class <**tagval**>: its values are pairs consisting of a class value and a value belonging to that class.
- the class <**any**>: it is a pseudo-class used to indicate that a value may belong to any class.

2.3. Class: tagval

Superclasses: universal.

The values contained in class *tagval* are pairs composed of a <class> value and another value belonging to that class. It allows run-time manipulation of values whose type cannot be determined statically or from the computational context. The <class> tag is never *tagval.*

This class is very similar to the class *free_context*, the main difference being that one cannot change the components of a <tagval> value. However, the value component of a <tagval> value may itself have its components modified when allowed by its class.

(**make** c:<class> v:<any>) --> <tagval>

This function creates a <tagval> value from its constituents. It is the user's responsibility to ensure that the class of the value of the second argument v: is the value of the first argument c:. If the value of c: is <tagval> then the result of the function is the value of the argument v:, since the value component is never a <tagval> value.

(**class** tv:<tagval>) --> <class>

This function returns the <class> component of the argument tv:. The result is never *tagval.*

(**value** tv:<tagval>) --> <any>

This function returns the value that is tagged with its class in the argument tv:.

(**call** tv:<tagval> f:<symbol> [v:<any> ...]) --> <any>

This function calls the semantic function f: defined for the class which is the class component of the first argument tv:. This semantic function must take the value component of tv: as its first argument (cf. *{class}:call*). Undefined behaviour will result when the function f: has not been so defined.

When applicable, this is equivalent to

({class}:call ({tagval}:tag tv:) f: ({tagval}:value tv:) ...)

2.4. Class: any

The class <any> is mainly used in the specification of some functions to indicate that the effective class of some argument, or of the result, is dependent on the context of its use. This is typically the case for the result of the function *{class}:call* and for several primitive functions in the class <tagval>.

The class value <any> may also be used effectively, when developing an application with the VTP, to indicate that a class is left unspecified. Whenever the pseudo-class <any> is used to type a function or any other entity, it is the VTP user's responsibility to know the actual class of the values manipulated by any appropriate means, and to process these values accordingly.

Thus any value of any class may be considered of class *any*. Conversely, any value specified to be of class *any* may be used as a value of another class under the programmer's responsibility.

No semantic function is associated with the class *any* itself. As a consequence, the function *{class}:call* may not be called with <any> as its first argument.

2.5. Native Lisp classes

Several classes are defined here that correspond to standard LE_LISP data types. The role of these "native" classes is to define smoothly the correspondence between native Lisp types and the classes of the VTP, in particular the conversion functions, and to allow a well defined use of Lisp literals. The mixing of Lisp concrete structures with the abstract classes of the VTP is necessary due to the lack of a sufficient abstraction mechanism in the Lisp language itself.

Naturally, all standard LE_LISP functions for Lisp types may be used with values taken in the corresponding VTP classes, without any class prefix. Prefixes must be used only for functions defined in the present document.

2.5.1. Class: symbol

Superclasses: universal.

These are the standard Lisp symbols recognized by the LE_LISP predicate function *symbolp* (cf. *name*).

2.5.2. Class: string

Superclasses: universal.

Standard conversion to: symbol, name, integer, number.

This class contains the Lisp string values, recognized by the LE_LISP predicate function *stringp*.

2.5.3. Class: integer

Superclasses: universal.

Standard conversion to: number, string.

This class contains the Lisp fixed precision integer values, recognized by the LE_LISP predicate function *fixp* (cf. *number*).

2.5.4. Class: character

Superclasses: universal.

Standard conversion to: string, symbol, name.

This class contains the Lisp character code values, i.e. all one character symbols as produced by the LE_LISP function *ascii.*

2.5.5. Class: boolean

Superclasses: universal.

This class contains the usual logical values *true* and *false*. Following the Lisp convention, the literal () may be used for *false*, and any other Lisp object to represent true.

2.6. Other elementary classes

This is a collection of elementary classes completing the "native" classes of LE_LISP. Some of them seem redundant with native classes: they have been introduced mainly to ensure independence from the LE_LISP implementation and should be preferred to the corresponding native class.

2.6.1. Class: name

Superclasses: universal.

Standard conversion to: symbol, string.

This class is similar to the native Lisp class *symbol.*

In the current implementation, these classes are identical. The intent of this new class is to prepare for an implementation of *names* independent of the native symbol implementation, thus avoiding the space overhead due to the role of symbols as identifiers in Lisp.

Lisp *symbol* literals and values may be supplied in place of *names* to the functions of the VTP. However users of the VTP should be careful to appropriately use conversion functions in their own code, so as to remain compatible with future implementation changes.

2.6.2. Class: number

Superclasses: universal.

Standard conversion to: integer, string.

This class contains values corresponding to all the non-negative integers. They are not intended for computation, but only to provide a syntactic representation for all integers, independently of any implementation limitation. Thus no arithmetic primitive functions are supplied. In a future version this class shall be implemented with the arbitrary precision rational package of LE_LISP, and will then be usable for arithmetic computations.

2.6.3. Class: singleton

Superclasses: universal.

This class contains only one value, which is represented by any Lisp value. Its role is to allow a uniform definitions. For example, the definition of 0-ary operators of a formalism requires the specification of a class for associated values. When there is no significant associated value, the class *singleton* is specified.

3. CONTEXTS

In the VTP, a **context** is a place within an object that may hold a modifiable value. The <context> value must not be confused with the value it contains. For example, when considering a subtree T1 of a tree T, we may be interested in T1 as a <tree> value, i.e. a tree in its own right, or as a <subtree> value, i.e. the place in T which holds T1. The class <subtree> is an example of the class schema <context>.

As a standard rule, any function that changes a value within a context returns the value that was formerly in that context. This rule, chosen to minimise accesses that may sometimes be expensive, is contrary to the usual Lisp rule of returning always the new value.

The role of contexts is essential in the VTP since its purpose is to provide a basis for a document and structure manipulation system, and thus to deal primarily with means of modifying those documents and structures, i.e. of changing subparts within their context.

Contexts are also a way to control substructure sharing. For example a <tree> value has a unique *main context*, which is the larger tree of which it is a subpart. This enforced uniqueness of the *main context* guarantees that we do not create dags†. Other references to the same <tree> value may be stored in *secondary* contexts.

3.1. Class schema: context

The general primitives and properties of contexts are described by the class schema *context.* Instances of this class schema include:

- the class <subtree> of positions of subtrees in surrounding trees.
- the class <annotation> of annotation positions within a tree, i.e. places that may hold a value annotating a node (a subtree) of a tree.
- the class <gate> of the place within each atom (i.e. a <tree> leaf) that holds the value associated with the atom.
- the class <sublist> of values denoting the position of a sublist within the list of the sons of a list-node of a tree.
- the class <free_context> used to create artificially a context for some values.

3.1.1. Creation and destruction of context values

Context values may be created in a variety of ways, depending on the kind (class) of the objects in which they denote a value holder. In some cases, these objects may be modified in such a way that some contexts cease to have a meaning. For example, a context may be a place in a list of values with variable length. A new context is created when a new value is inserted in the list. Changing a value in one place of the list is understood as changing the value contained in the corresponding context. Removing a value from the list is understood as deleting the context that

† *dag*: *d*irected *a*cyclic graph
(trees may have shared subtrees)

contains that value. Then any reference to that context is no longer meaningful.

Destruction of a <context> may be a side-effect of some primitive function operating on the object in which it is defined, or it may be explicitly required by a *delete* function. To test whether a <context> is meaningful, one uses the function *valid.*

There are also <context> values that are always valid, i.e. meaningful, though they have no actual existence. This may be the case when a context is actually created only when there is a 'useful' value to place in it, although it may validly be referenced as a context before then. Such a <context> may remain valid, though without actual existence, even after being given as an argument to the function *delete.*

(**delete** c:<context>) --> <any>

The exact effect of this function varies for each class instance of the class schema <context>. It may range between:

- actual destruction of the <context> c:, making it an invalid <context> in the sense of the function *valid* (e.g. <subtree> context associated with a list element).
- destruction of the value contained in c:, making it uninitialized, or equivalently removing the context from actual to virtual existence. The <context>, however, remains a valid one. In addition to the semantic uses of undefined values, virtualization of a context may also be useful to reclaim its storage when it no longer contains a useful value (e.g. <annotation> contexts).
- no effect at all, except maybe the raising of an exception if requested by the VTP user (e.g. <gate> context).

(**valid** c:<context>) --> <boolean>

The result of this function is true if and only if c: is a meaningful context as explained above.

(**actual** c:<context>) --> <boolean>

The result of this function is true if the <context> c: actually exists. This always implies that the <context> c: is valid.
Note: this function is only included tentatively in this version of the VTP and may be removed in the future.

Any function other than *valid* or *delete* applied to an invalid <context> will cause the raising of the exception *invalid_context.*

3.1.2. Reading or writing in a context

The purpose of a <context> is to contain a value. To each <context> is associated a <class> which is the class of values that are permitted in this context.

(**class** c:<context>) --> <class>

This function returns the <class> of values permitted in the <context> c:.

(**value** c:<context>) --> <any>

This function returns the value currently contained in the <context> c:. This value belongs to the class required by the context c:. If the context c: is valid but not actually created or initialized, the exception *uninitialized_context* is raised.

(**tagval** c:<context>) --> <tagval>

This function returns a <tagval> value, which is either the value returned by the function *value* if its class is <tagval>, or this value tagged with its class otherwise. If the context c: is valid but not actually created or initialized, the exception *uninitialized_context* is raised.

(**replace** c:<context> v:<any> [k:<class>]) -->. <any>

Like the function *value*, this function returns the value previously contained in the <context> c:. The result is undefined when c: had no previous actual existence, or had not been properly initialized.
The second argument v: becomes the new value contained in the context c:. Both values must belong to the class required by the context c:. The optional third argument k:, when present, must indicate the class of the supplied value v:, and thus may play two roles:

- it allows conversion between tagged and untagged values, depending on v: and on the class required by the context k:.
- it allows run-time check of the class correctness of the arguments supplied with respect to the type required by the context k:.

3.2. Class: free_context

Superclasses: universal, context.

Standard conversion to: tagval.

A free context is a context that is not within a larger structure and is created for the only purpose of containing a value. One main use of *free_contexts* is to share a changing value between different structures by accessing the value through this context.

Another use of *free_contexts* is to artificially create a context in some circumstances. For example, most <tree> values have a context which is their place in a larger tree. Some trees, not being part of a larger one, have no such context. For uniformity it is sometimes convenient to create artificially a context for these trees by means of a free_context.

The class of values that may be contained in a free_context is fixed when the free_context is created. This class may be <any> or <tagval>.

The class *free_context* is very similar to the class *tagval*, the main difference being that one can change the value placed in a free_context.

(**make** c:<class> v:<any>) --> <free_context>

This function creates a <tagval> value from its constituents. It is the user's responsibility to ensure that the class of the value of the second argument v: is the value of the first argument c:.

The function *delete* is ineffective for *free_contexts*.

The functions *valid* and *actual* always return true for *free_contexts*.

4. TREE STRUCTURES

The intent of the VTP is to provide a structured representation of documents as labelled trees. The specification of these tree structures is the object of the present chapter. This chapter makes few assumptions about the syntactic constraints that must be obeyed when building or modifying trees (only the distinction between atomic nodes — leaves — and other nodes is essential). The definition and enforcement of syntactic constraints on labelled trees (abstract syntax) is the object of the next chapter.

The aim of this separation, reflected in the implementation, is to leave open the possibility of experimenting with different definitions of abstract syntax. It is also to allow manipulation of the tree structures independently of any syntactic check.

4.1. Class schema: arborescent

This class schema covers (is a super-class of) several actual classes corresponding to tree structured objects, including <tree> <subtree> and <sublist>.

The functions available in this class schema are divided into several categories:

- local navigation
- modification of non-atomic nodes
- modification of list nodes
- access to, and modification of atomic nodes
- pattern matching
- conversion functions
- miscellaneous primitives

Modification primitives are used to change tree structures by insertion, deletion or replacement of tree subparts. A general convention is that these functions return any <tree> value that they remove from its context (usually a surrounding tree).

In the description of these functions, for every argument t:<arborescent>, we use the expression "<tree> value corresponding to t:" with the following meaning:

- when t: is a <tree> value: it means t: itself;
- when t: is a <subtree> value: it means the conversion of t: to a <tree> value, i.e. ({*subtree*}*:tree t:*);
- when t: is a <sublist> value: it means a virtual (not actually existing) tree *tt* constructed as follows:
  * the root of *tt* is labelled by the same operator as the list in which t: is defined;
  * the list elements in t: are sons of this root;
  * the tree *tt* is considered a son of the list in which t: is defined in place of the elements selected by t:.

When the interpretation of a semantic function is not obvious for a specific class instance of the class schema *arborescent,* this semantic function is covered again in the section devoted to that class instance.

4.1.1. Local navigation primitives

These primitives are used for explicit local motions in the tree structure. They do not modify the structure and have no side-effect.

Whenever the requested motion is not possible, the exception *navigation* is raised.

(**up** t:<arborescent> n:<integer>) --> <arborescent>

Returns the <arborescent> value that is n: levels higher in the tree structure than the argument t:. If there are not that many levels above t:, then the exception *navigation* is raised. A negative argument n: requests the highest <arborescent> value above t:.

(**down** t:<arborescent> n:<integer>) --> <arborescent>

Returns the <arborescent> value that is the n:-th son of the argument t:. It raises the exception *navigation* if there is no such son. Sons are numbered from the left, the first one having rank 1. When the argument n: is negative, the numbering is taken from right to left and the absolute value of n: is used.

(**left** t:<arborescent> n:<integer>) --> <arborescent>

Returns the <arborescent> value that is the n:-th brother on the left of the argument t:. When n: is 0, the argument t: itself is returned. When n: is less than 0, the result is the brother whose rank, counted from left to right, is the absolute value of n:. The exception *navigation* is raised when there is no applicable brother.

(**right** t:<arborescent> n:<integer>) --> <arborescent>

Returns the <arborescent> value that is the n:-th brother on the right of the argument t:. When n: is 0, the argument t: itself is returned. When n: is less than 0, the result is the brother whose rank, counted from right to left, is the absolute value of n:. The exception *navigation* is raised when there is no applicable brother.

(**next** t:<arborescent>) --> <tree>

Returns the <arborescent> value that follows the argument t: in a preorder traversal of the tree containing t:. The exception *navigation* is raised when the <tree> value associated with t: is the rightmost leaf of a tree that is not a subtree of a larger one.

4.1.2. Modification of non-atomic nodes

(**change_son** t:<arborescent> t1:<tree> n:<integer>) --> <tree>

Places the argument t1: as the n:-th son of the tree-structure designated by t:. The result is the <tree> value that was formerly in that position. The argument n: must be interpreted in the same way as in the function *down.*

(**replace** t:<arborescent> t1:<tree>) --> <tree>

Returns the <tree> value corresponding to t:, after replacing it by the <tree> t1: within its context.

(**erase** t:<arborescent>) --> <tree>

The <tree> value corresponding to the argument t: is returned as a result. It is replaced in the tree structure by an atomic tree with the operator 'metavariable', standing for an undefined sub-tree. The name of the metavariable is that of the phylum corresponding to the location where the sub-tree is being replaced. This function should not be confused with the functions *delete* and *nullify*.

(**nullify** t:<arborescent>) --> <tree>

The <tree> value corresponding to the argument t: is returned as a result. It is replaced in the tree structure by a "null" tree constructed in a standard way from an operator belonging to the intersection of the phylum associated with the position of the argument t: and of a special phylum of null-tree operators specially defined for each formalism (precisely the phylum returned by the call (*{formalism}:null_tree_operator ff:*), where ff: is the formalism of the argument t:). If there is more than one such operator, one of them is chosen according to an unspecified algorithm. The semantic intent is to perform a replacement by a tree representing a null object. (The realization of this functionality may change slightly in the future.) This function should not be confused with the functions *delete* and *erase*.

4.1.3. Modification of list nodes

The following primitive functions are used to create or delete sons in list nodes of a tree. They change the number of sons of these list nodes. Any attempt to use them that would result in changing the number of sons of a fixed-arity node results in raising the exception *not_a_list_node.*

When the corresponding checks are enabled and meaningful for the formalism used, any attempt to delete the last son of a list that may not be empty results in raising the exception *non_empty_list*.

(**adopt** t:<arborescent> t1:<tree> n:<integer>) --> <tree>

Insert a new son t1: in the <tree> value corresponding to t: after the n:-th existing son of t:. The head operator of t: must be a list operator. The argument n: must be interpreted as in the function *down*. When n: is 0, tree t1: is inserted as the first son. The result returned is t1: (cf. *precede* and *follow*).

(**disown** t:<arborescent> n:<integer>) --> <tree>

The head operator of the <tree> value corresponding to t: must be a list operator. The position of the n:-th son of the head node in the list is deleted, thereby reducing the number of sons by one. The result is the <tree> value of the son that has been removed (cf. *delete*).

(**precede** t:<arborescent> t1:<tree>) --> <tree>

Inserts t1: as an adjacent left brother of the <tree> value corresponding to t:, which must be a son of a list node. The result returned is t1: (cf. *adopt* and *follow*).

(**follow** t:<arborescent> t1:<tree>) --> <tree>

Inserts t1: as an adjacent right brother of the <tree> value corresponding to t:, which must be a son of a list node. The result returned is t1: (cf. *adopt* and *precede*).

(**delete** t:<arborescent>) --> <tree>

The <tree> value corresponding to the argument t: must be the son of a list node of a tree structure. The effect of the call is to remove the position of this son from the list, thereby reducing the number of sons by one. The result is the <tree> value of the son that has been removed. This function should not be confused with the functions *erase* and *nullify* (cf. *disown*).

4.1.4. Atomic nodes access and modification primitives

Access to the internal structure of atoms is done by considering them as *gates*, i.e. the *context* values that contain the values associated with atoms. However, some (redundant) primitive functions are provided to access directly this internal structure.

The exception *non_atomic* is raised when one of these functions is applied to an <arborescent> value that does not correspond to an atomic tree.

(**gate** t:<arborescent>) --> <gate>

The <tree> value corresponding to t: must be atomic, i.e. have a 0-ary operator. The result of this conversion function is the <gate> value associated with this leaf node.

(**atom_class** t:<arborescent>) --> <class>

This function is equivalent to the application of *{gate}:class* to the result of the conversion of t: into a <gate> value.

(**atom_value** t:<arborescent>) --> <any>

This function is equivalent to the application of *{gate}:value* to the result of the conversion of t: into a <gate> value.

(**atom_tagval** t:<arborescent>) --> <tagval>

This function is equivalent to the application of *{gate}:tagval* to the result of the conversion of t: into a <gate> value.

(**atom_replace** t:<arborescent> v:<any> [k:<class>]) --> <any>

This function is equivalent to the application of *{gate}:replace* to the result of the conversion of t: into a <gate> value.

4.1.5. Pattern matching based primitives

These primitives are a collection of functions based on the use of tree pattern matching. In particular, they include simple pattern matching, global search of a tree pattern, and application of a function to all occurrences of a pattern. See the chapter on pattern matching for details about the construction of tree schemata (i.e. tree patterns) and the pattern matching process itself.

Essentially, tree patterns are normal trees where some subtrees have been replaced by a special atomic node called a **metavariable**. Each metavariable has a name. A tree is an instance of a tree pattern if they are equal except for occurrences of metavariables in the pattern.

(**match** t:<arborescent> s:<tree> [st:<store>]) --> <boolean>

This function returns *true* if and only if the <tree> value corresponding to t: is an instance of the tree pattern s:. The optional argument st: is a **store** function, i.e. a function that takes three arguments

- a <name> value,
- another value in any class,
- a <class> value which must be the class of the second argument.

One could define the functional class <store> as follows:

$$<store> = <name> \times <any> \times <class> \rightarrow <any>$$

The result of the functional argument st: is immaterial here. The normal aim of such a store function is to replace in some environment the value associated with *name* (first argument) by the value specified by the second argument. The third argument is necessary to dynamically keep track of classes.

When the match succeeds, the function st: (if provided) is called once for each metavariable of the pattern s:. For each such call, the first argument is the name of the metavariable and the second is (in simple cases) the part of the tree t: that is replaced by the metavariable in s:.

(**find** t:<arborescent> s:<tree> [st:<store>] [l:<tree>]) --> <arborescent>

This function searches the <tree> value *tt* associated with t: to find a part of it which is an instance of the tree pattern s:. If such an instance is found, the optional argument st: is applied in the same manner as described above with the *match* function. The search proceeds in prefix order. If *tt* is itself a part of a larger tree, and if no instance of s: is found in *tt*, then the search continues in the right brothers and uncles of *tt* in this larger tree. The prefix search may be limited by an optional last argument l: which must be a <tree> value encountered in the search. The search is terminated as soon as it finishes exploring part or all of that tree. To limit the search to the tree *tt*, if is sufficient to give it as last the argument. The exception *no_match* is raised when no instance of the pattern is found within the assigned limits.

(**for_all_instances** t:<arborescent> s:<tree> foo:<tree→any> [st:<store>]
[l:<tree>]) --> <boolean>

This function searches the <tree> value *tt* associated with t: to find all parts of it that are instances of the tree pattern s:. For each such an instance that is found, the optional argument st: is applied as in the semantic function *match*, and then the function foo: is called with the found instance of s: as an argument. The search proceeds in prefix order. If *tt* is itself a part of a larger tree, then the search continues in the right brothers and uncles of *tt* in this larger tree. This prefix search may be limited by an optional last argument l: which must be a <tree> value encountered in the search. The search is terminated as soon as it finishes exploring part or all of that tree. The result of the call is *true* when an instance of s: is encountered and *false* otherwise.

4.1.6. Miscellaneous primitives

This category includes in particular a collection of conversion primitives.

(**operator** t:<arborescent>) --> <operator>

It returns the <operator> of the head node of the <tree> value corresponding to t:.

(**son_phylum** t:<arborescent> n:<integer>) --> <phylum>

It returns the <phylum> of the n:-th son position of the head operator of the <tree> value corresponding to t:. It raises the exception *operator_structure* when there cannot be an n:-th son.

(**formalism** t:<arborescent>) --> <formalism>

It returns the <formalism> to which the head operator of the <tree> value corresponding to t: belongs.

(**for_all_sons** t:<arborescent> foo:<arborescent→any>) --> <any>

This function is an iterator that applies the function foo: successively from left to right, to each <arborescent> value corresponding to a son of t:. The result is that of the last call to foo:.

(**length** t:<arborescent>) --> <integer>

It returns the number of sons of the head node of the <tree> value corresponding to t:. It is equal to the arity of the operator of that node for non-list nodes.

(**rank** t:<arborescent>) --> <integer>

This function returns the position of the <tree> value associated with t: in the list of its father's sons. The first son has rank 1. When it does not have a father, the exception *navigation* is raised.

(**slice** i1:<integer> i2:<integer> t:<arborescent>) --> <sublist>

If the <tree> value corresponding to the argument t: has a list operator labelling its root, then the result is the <sublist> value referencing the sublist of this root starting at the i1:-th son and ending at the i2:-th son. The result is an empty list when i2: is equal to (i1: - 1). The result is an undefined <sublist> value when i2: is less than (i1: - 1), but no exception is raised. The exception *not_a_list_node* is raised if the label of the root of t: is not a list operator. The son indexes i1: and i2: may have the values (n + 1) and 0 respectively, where n is the length of t:, so as to be able to slice empty lists. Otherwise, any value of the indexes that is not the rank of a son of t: causes the raising of the exception *navigation*.

4.2. Class: tree

Superclasses: universal, arborescent.

A tree is a recursive structure composed of labelled nodes with sons that are themselves trees, except for so-called **atomic** nodes or **leaves** that have no sons, but have an associated value. A tree may not be shared by two larger tree (no dags), nor be a son of one of its own nodes. Thus a tree *tt* usually occurs at most once within the context of a larger tree of which it is a part. The place where it so occurs is a <subtree> value which is by definition the **main context** of the tree *tt*. When a <tree> value is not part of a larger tree, it may have no main context. However, any <context> value that may hold a tree may be assigned to it as its main context with the function *{tree}:set_context*.

A <tree> value may have at most one main context, which is returned by applying to it the function {*tree*}*:context*. However, the same <tree> value may be stored in other contexts that are considered as secondary.

(**equal** t1:<tree> t2:<tree>) --> <boolean>

Two <tree> values are equal if and only if they correspond to trees having the same structure and labels, with the same values associated with each corresponding atomic node. They may not be the same tree, i.e. they do not remain equal when one of them is modified.

(**copy** t:<tree>) --> <tree>

This function returns a new <tree> value that is equal to its argument t:.

(**make** o:<operator> [t:<tree> ...]) --> <tree>

This function returns a <tree> value. The head operator (i.e. the label of the root) is the first argument o:. The sons of the root are the subsequent arguments in the given order.

(**tree** t:<tree>) --> <tree>

This is the identity in class *tree*.

(**subtree** t:<tree>) --> <subtree>

This function returns the <subtree> value which is the place occupied by the argument t: within a larger tree. The exception *arborescent_conversion* is raised if t: is not part of a larger tree.

(**context** t:<tree>) --> <any>

This function returns the *main*context of the <tree> value t:.

(**set_context** t:<tree> c:<any> cl:<class>) --> <any>

The argument c: must be a <context> value that may contain a <tree> value. The last argument cl: is the class of that context. The <tree> value t: is stored in that context, as with the function *replace*, and the result is the same as if *replace* had been called. The additional effect is to assign the context c: as the *main context* of tree t:.

(**sublist** t:<tree>) --> <any>

When t: is a <tree> value that is the son of a list node, the result is the sublist of that node that contains only that tree. Otherwise the exception *arborescent_conversion* is raised.

4.3. Class: subtree

Superclasses: universal, context, arborescent.

A <subtree> value is the place that holds a subtree within a larger tree. It is a context.

Since a tree may occur only once in a larger tree (no dags), there is a natural conversion between <tree> and <subtree> values. The difference between a <tree> value *t1* and the corresponding <subtree> value *st* becomes apparent when the larger tree is modified by replacing *t1* by another <tree> value *t2* at the place indicated by *st*. After such a modification, the tree *t1* no longer has an associated subtree, while the tree associated with *st* is now *t2*.

The distinction between *trees* and *subtrees* is also apparent in the semantic functions *equal* and *copy*.

A <subtree> value may become **undefined** if it denotes a place in a list node which is later destroyed with the function *delete*. An *undefined* subtree never becomes defined again.

(**equal** t1:<subtree> t2:<subtree>) --> <boolean>

The result is *true* when t1: and t2: are the same <subtree> values, i.e. the same place in the same tree. Thus their equality is not affected by tree modifications.

(**copy** t:<subtree>) --> <subtree>

This is an identity function in the class *subtree*.

(**tree** t:<subtree>) --> <tree>

This function returns the <tree> value occurring as subpart of a larger tree at the place which is the value of the argument t:.

(**subtree** t:<subtree>) --> <subtree>

This is an identity function in the class *subtree*.

(**sublist** t:<subtree>) --> <any>

When t: is the place of a tree as son of a list node, the result is the sublist of that node that contains only that tree. Otherwise the exception *arborescent_conversion* is raised.

(**valid** st:<subtree>) --> <boolean>

The result is *true* if the sublist st: is not *undefined* according to the above definition.

(**actual** st:<subtree>) --> <boolean>

This function gives the same result as the function *valid*.

4.4. Class: sublist

Superclasses: universal, context, arborescent.

A <sublist> value denotes a sequence of adjacent sons of a same node within a tree. A sublist may be **empty**, either because it is created empty or because all sons within the sublist have been destroyed with the function sons of the same list-node.

If a new son is created for a list-node between two sons of that node, any empty sublist that may have existed between these same two sons becomes **undefined**, because one does not know whether it is placed before or after the new son. An undefined sublist is essentially an empty list with an undefined position. An undefined sublist may be made defined again when sons are removed from the list in which it occurs (cf. function *slice*).

(**equal** sl1:<sublist> sl2:<sublist>) --> <boolean>

The result is *true* when sl1: and sl2: are the same <sublist> value, i.e. they denote the same son positions for the same list node of the same tree. Thus their equality is not affected by tree modifications.

(**copy** sl:<sublist>) --> <sublist>

This is an identity function in the class *sublist*.

(**tree** sl:<sublist>) --> <tree>

If the argument sl: is a sublist that contains exactly one son of a list node, then the result is that son. Otherwise, the exception *arborescent_conversion* is raised.

(**subtree** sl:<sublist>) --> <subtree>

If the argument sl: is a sublist that contains exactly one son of a list node, then the result is the <subtree> value which is the place where that son occurs. Otherwise, the exception *arbores-*

cent_conversion is raised.

(**sublist** t:<sublist>) --> <any>

This is an identity function in the class *sublist*.

(**delete** sl:<sublist>) --> <any>

The effect of this function is the same as if the function *{tree}:delete* had been applied to each tree in the sublist.

(**valid** sl:<sublist>) --> <boolean>

The result is *true* if the sublist sl: is not *undefined* according to the above definition.

(**actual** sl:<sublist>) --> <boolean>

This function gives the same result as the function *valid.*

All navigation functions, and all son indexing, must be understood as if the <sublist> value were an actual node, having the trees it contains as sons, and being itself a son of the parent list node in their place. When the result is specified as <arborescent> in the specification of class schema *arborescent*, it must be here a <tree> value.

We detail a few of these functions as examples:

(**up** sl:<sublist> n:<integer>) --> <tree>

Returns the <tree> value that is n: lsevel higher in the tree structure than the argument sl:. When n: is equal to 1, the result is the <tree> value having as root the list node in the sons of which the sublist sl: occurs. If there are not that many levels above sl:, then the exception *navigation* is raised. A negative argument n: requests the highest <tree> value above sl:.

(**down** sl:<sublist> n:<integer>) --> <tree>

Returns the <tree> value that is the n:-th son of the sublist sl:. It raises the exception *navigation* if there is no such son. Sons are numbered from left to right, starting at 1. When the argument n: is negative, the numbering is taken from right to left and the absolute value of n: is used.

(**left** sl:<sublist> n:<integer>) --> <tree>

Returns the <tree> value that is the n:-th brother on the left of the leftmost element of the sublist sl:. When n: is 0, this leftmost element itself is returned. When the list is empty, the adjacent list son on the right of the sublist replaces the leftmost element of the sublist. When n: is less than 0, the result is the brother whose rank, counted from left to right, is the absolute value of n:. The exception *navigation* is raised when there is no such brother.

4.5. Class: gate

Superclasses: universal, context.

A value of class <gate> corresponds to the place in a tree atom that holds the value associated with that atom. It is a *context.* New <gate> values may be obtained only by application of the function *gate* to an <arborescent> value corresponding to a tree atom. The function *delete* is without effect on gates. The function *valid* always returns the value true.

5. FORMALISMS AND ABSTRACT SYNTAX

A **formalism** is a structure that defines a set of trees. In its most elementary form, a formalism only specifies a collection of **operators**, i.e. of labels, to be used on the nodes of trees belonging to the formalism. A distinction is made at least between atomic operators (reserved for leaf nodes) and non-atomic operators (reserved for non-leaf nodes).

The set of trees in a given formalism may be restricted by an **abstract syntax**, a collection of rules specifying the number of sons permitted to each node according to its label, and the labels permitted on these sons. Furthermore, a <class> value must be associated with each atomic operator to specify the class of the values to be associated with leaf nodes labelled with this operator.

In the VTP, the abstract syntax of a formalism is characterized by the definition of its operators and phyla. A **phylum** is a set of operators belonging to the formalism. Each operator is characterized by its arity, i.e. the number of sons permitted to a node it labels, and by the phyla to which the operators of these sons must belong.

Tree construction or manipulation primitives provided by the VTP may or may not check correctness with respect to the abstract syntax. This checking may be enabled or disabled by the user of the VTP.

Creation of a formalism should be done by compilation of a specification of the formalism in a special purpose language. One such language, **Metal**, is already implemented, and its use is presented in a separate document. Here we assume that all formalisms already exist in the user environment, and may be referenced as needed (in fact they are automatically loaded from remanent storage when referenced).

Primitive functions for the creation of formalisms shall be provided, to be used only for the implementation of formalism specification languages and compilers. The use of these primitives for dynamic modification of an existing formalism should be avoided so as to maintain compatibility between the structures created within this formalism.

5.1. Class: formalism

Superclasses: universal.

A formalism is characterized by a name, a dialect and a version number. This is necessary to allow coexistence in the same environment of different dialects of the 'same' formalism, and to distinguish the level of evolution of each dialect. Dialects and versions are common with programming languages, whose manipulation is one of the aims of the VTP.

The exception *remanent_store* is raised when a needed formalism is not found in the user's environment.

({**name**}:**formalism** n:<name> [d:<name>] [v:<integer>]) --> <formalism>

This function returns the version v: of dialect d: of the <formalism> with name n:. The arguments d: and v: are optional. A default value is taken for them when necessary, i.e. when there are several versions or dialects.

(**name** f:<formalism>) --> <name>

The result is the name of <formalism> f:, without any indication of the version number or dialect used.

(**dialect** f:<formalism>) --> <name>

The result is the name of the dialect of <formalism> f:.

(**version** f:<formalism>) --> <integer>

The result is the version number of <formalism> f:.

(**for_all_operators** f:<formalism> foo:<operator→any>) --> <any>

This function is an iterator that applies the function foo: to each <operator> belonging to the <formalism> f:. The result is that of the last call to foo:. The order in which the operators are processed is undefined and may change from call to call.

(**for_all_phyla** f:<formalism> foo:<phylum→any>) --> <any>

This function is an iterator that applies the function foo: to each <phylum> belonging to the <formalism> f:. The result is that of the last call to foo:. The order in which the phyla are processed is undefined and may change from call to call.

5.2. Class: operator

Superclasses: universal.

An operator is characterized by its name and by the formalism it belongs to. Operators are statically defined by the formalism they belong to, and they are available whenever their formalism is. Thus no primitive is supplied for the creation of new operators independently of the creation of new formalisms.

(**{name}:operator** n:<name> f:<formalism>) --> <operator>

This function returns the <operator> named n: in the <formalism> f:. The exception *unknown_operator* is raised when there is no such operator.

(**formalism** o:<operator>) --> <formalism>

This function returns the <formalism> the operator o: belongs to.

(**atomic** o:<operator>) --> <boolean>

This function returns true if and only if the operator o: is atomic.

(**arity** o:<operator>) --> <arity>

This function returns the <arity> of the operator o:. The **arity** of its operator defines the number of sons permitted to a tree node. An <arity> value is either a non-negative integer or one of the Lisp symbols '* and '+ †. An integer arity indicates that a node must have exactly that number of sons. The other two arities are for so-called list nodes (or arbitrary nodes) that may have any number of sons ('*), or at least one son ('+).

(**phylum** o:<operator> n:<integer>) --> <phylum>

The result is the <phylum> of the permitted operators on the n:-th son of a tree node labelled with the operator o:. The exception *operator_structure* is raised when n: is negative or greater than the arity of o:, or when o: is an atomic operator (i.e. its arity is zero).

(**class** o:<operator>) --> <class>

When o: is an atomic operator (i.e. its arity is zero), this function returns the <class> of values that may be associated with tree leaves labelled by o:. Otherwise, the exception *operator_structure* is raised.

† (*quote* *) and (*quote* +).

(**belongs** o:<operator> p:<phylum>) --> <boolean>

The result is true if and only if the operator o: belongs to the phylum p:.

5.3. Class: phylum

Superclasses: universal.

A phylum is characterized by its name and the formalism it belongs to. New phyla may be dynamically created in a formalism, but this has no effect on the set of trees defined by the formalism since this is entirely controlled by operators. However, new phyla are sometimes useful to define special families of trees and/or operators within a formalism.

(**{name}:phylum** n:<name> f:<formalism>) --> <phylum>

The result is the <phylum> of name n: in the formalism f:. The exception *unknown_phylum* is raised when there is no such phylum.

(**formalism** p:<phylum>) --> <formalism>

The result is the <formalism> the <phylum> p: belongs to.

(**contains** p:<phylum> o:<operator>) --> <boolean>

The result is true when the <phylum> p: contains the <operator> o:. This function is the same as *{operator}:belongs* with an inversed order of arguments.

(**make** n:<name> f:<formalism> <operator>-list) --> <phylum>

This function creates a new phylum named n: in formalism f:, containing precisely the operators given as arguments after f:. Empty phyla are permitted.

(**for_all_operators** p:<phylum> foo:<operator→any>) --> <any>

This function is an iterator that applies the function foo: to each <operator> belonging to the <phylum> p:. The result is that of the last call to foo:. The order in which the operators are processed is undefined and may change from call to call.

5.4. Syntax checking

Structure constructing or modifying primitives may be used with or without checking that the abstract syntax is respected. Appropriate means shall be provided to enable or disable these checks.

Similarly, means shall be provided to enable or disable dynamic type checking in circumstances where static type checking is not possible.

[to be completed]

6. PATTERN-MATCHING

[to be completed]

7. ANNOTATIONS

[to be completed]

8. INDEXES

This section contains several distinct indexes for the various kinds of entities introduced in the present document:

- An index of functions defined for each class schema. The function name is prefixed with the name of the class between braces, in the usual notation.
- An index of the functions available in the actual classes implemented. These functions are normally defined in the section of this document describing the corresponding class. However, for each class we also include all the functions inherited from its superclasses. All function names are prefixed with the name of the class. The page references are either real references in sections dealing explicitly with the class, or inherited references from sections describing a superclass.
- An index of all functions, without any class prefix. These functions are either functions belonging to classes and can only be used with a proper class prefix, or general utility functions (still ?) independent of any class and are to be used without a prefix.
- An index of the exceptions that may be raised by the functions of the VTP.

The following font conventions for page references are used within the indexes:

boldface:

used for definitional references, i.e. pages where the indexed entity is defined. Boldface references are also used for other important occurrences (in boldface) of the entity name in the text.

Boldface is also used for *inherited* definitional or important references. Inherited references denote occurrences of entities belonging to a subclass, when they appear in the text in connection with a superclass of this subclass, rather than in connection with the subclass itself.

roman:

used for all other references.

8.1. Index of class schema functions

8.2. Index of class functions

8.3. General index of functions

8.4. Index of exceptions

9. TABLE OF CONTENTS

D4 - PROPOSAL FOR A LANGUAGE DEFINITION FORMALISM AND SELECTION OF TEST CASES

D4.A1 - USER DEFINABLE SYNTAX FOR SPECIFICATION LANGUAGES

D4.A2 - SPECIFICATIONS IN NATURAL SEMANTICS

D4.A3 - PROPOSAL FOR AN ALGEBRAIC SEMANTICS DEFINITION FORMALISM

GIPE : CEC 348/A/T4/1
version 2 November 1985

Proposal for a Language Definition Formalism and Selection of Test Cases

Deliverable D4 of Task T4 — Second Review —

J. Heering (CWI)
G. Kahn (INRIA)
P. Klint (CWI)

The goal of ESPRIT Project 348 is to generate interactive programming environments from formal language definitions. An initial version of the language definition formalism required has been developed as part of Task T4 of the project. The scope of this formalism and its various subformalisms is defined, and criteria for selecting examples to be used as test cases for them are identified. Proposals for specific (sub)formalisms to be investigated together with specific test cases are given in Annexes D4.A1, D4.A2, and D4.A3.

Actual specification of the selected examples falls outside the scope of Task T4 but will be carried out under Task T7.

1. INTRODUCTION

The goal of ESPRIT Project 348 is to generate interactive programming environments from formal language definitions. An initial version of the language definition formalism required has been developed as part of Task T4 of the project. This formalism and its various subformalisms will be tested and, if necessary, improved by applying them to a series of carefully selected examples. It is expected that the outcome of this phase will be a satisfactory language definition formalism for which a programming environment generator can be developed.

The purpose of this document and its annexes is twofold:

- Definition of the scope of the language definition formalism and identification of the various subformalisms required (§2). Proposals for specific (sub)formalisms to be investigated are given in the annexes.
- Identification of criteria for selecting examples that will be used as test cases for these (sub)formalisms (§3). Proposals for specific test cases are given in the annexes.

Although the actual specification of the test cases is scheduled as Task T7 ("Specification of Selected Examples") and falls outside the scope of this document, various examples will be given in the annexes for the purpose of illustrating the formalisms proposed.

2. THE LANGUAGE DEFINITION FORMALISM - GENERAL CONSIDERATIONS

In principle, a formal language definition need only contain a specification of the syntax and semantics of the language to be defined. As far as the language proper is concerned this is sufficient. If, however, a programming environment is to be derived from such a definition, it must contain additional information so as to enable the environment generator to produce language-dependent programming tools like a syntax-directed editor, type-checker, evaluator, symbolic tracer, and file system.

We currently envisage language definitions consisting of three parts:

(A) A *syntax section* containing a definition of the abstract and concrete syntax (including lexical syntax and pretty printing) of the language to be defined.
(From this part the programming environment generator has to derive a syntax-directed editor.)

(B) A *static constraints* section containing a definition of the *type constraints* or *static semantics* of the language.
(From this part the programming environment generator has to derive an incremental type-checker.)

(C) A definition of the *(dynamic) semantics* of the language.
(From this part the programming environment generator has to derive an incremental evaluator and, at a later stage, a compiler.)

Reflecting the above partitioning of language definitions, the overall language definition formalism we currently have in mind has three well-defined subformalisms for describing respectively the syntax, static constraints, and semantics of a language. Preliminary proposals for three corresponding subformalisms together with appropriate test cases are given respectively in Annexes D4.A1, D4.A2 and D4.A3. Of course, language definitions may require additional sections for other purposes, so their partitioning into three sections and the corresponding partitioning of the overall formalism is tentative at best.

The proposed subformalism for use in the syntax section of language definitions (Annexe D4.A1) can be used for defining both the concrete and abstract syntax of a language. As for the other two sections, INRIA is currently concentrating on the use of *structural operational semantics* (*inference rules*) [PLO81] for defining static constraints (Annexe D4.A2), while CWI is concentrating on *algebraic definition* [KLA83, EM85] of semantics (Annexe D4.A3). It is not obvious, however, that static constraints and semantics require different subformalisms, so INRIA will also investigate definitions of semantics in terms of inference rules, while CWI will try its hand at algebraic definition of static constraints. The theoretical relationship between inference rules and algebraic semantics will be investigated as part of Task T7.

Language definitions can easily become quite large and will therefore have to be built from smaller ones. A fourth subformalism will deal with composition of language definition modules and related matters. A preliminary formalism for this purpose is proposed in Annexe D4.A3 as part of the algebraic semantics definition subformalism. Because language definitions are composite objects containing sections in at least two or three different subformalisms, combining them is difficult. Work on composition of language definitions will be continued under Task T7.

The overall language definition formalism adopted must be such that a variety of programming and application languages can be defined in it. It must also have good mathematical properties, i.e. its semantics must be well understood and mathematically manageable. Perhaps the main question to be answered is: Why not try denotational semantics for static constraints and/or dynamic semantics? On the part of INRIA experience with denotational semantics has led to its rejection on the following grounds:

- For static semantics, denotational semantics equations are clumsy and the ways to specify tree traversal are not very elegant.
- It seems difficult to describe parallel constructs in the dynamic semantics of a language.
- Use of pure denotational semantics may lead to overspecification.

It must be noted that we are not yet convinced of the superiority of inference rules over denotational semantics in the case of dynamic semantics. The test cases will have to play a crucial role here.

On the part of CWI modularisation of language definitions and reuse of language definition modules have for some time been considered important issues. This has led to a "language definitions viewed as abstract data types" viewpoint which in turn has led to the choice of algebraic semantics. Although algebraic semantics may be viewed as a form of denotational semantics, for several reasons the associated style of specification is rather different from that used in ordinary denotational semantics:

- Much attention has been paid to the semantics of parameterised or otherwise incomplete algebraic specification modules and to the composition of such modules into larger ones. Modular specification is the rule rather than the exception. (This advantage is important but undoubtedly only temporary, because the development of modular denotational semantics is only a matter of time. Modular functional languages have recently started to appear.)
- Algebraic semantics - at least in its present form - is a first-order formalism. Models of (complete) algebraic specifications are first-order algebras.
- Apart from the fact that they have to be first-order, the equations allowed are more general than those allowed in ordinary denotational semantics. (In principle this is an advantage, but it may also lead to problems if the specifications have to be made executable.)

Whether algebraic semantics is powerful enough to satisfy our requirements remains to be seen. The problems associated with making algebraic specifications operational will be circumvented by imposing regularity restrictions on the (conditional) equations used [BK82]. If this is done, the resulting specifications can very simply be transformed into confluent term rewriting systems. These in turn can readily be compiled to PROLOG programs.

Our experiences with compiling general, i.e. unrestricted, algebraic specifications to term rewriting systems by means of the Knuth-Bendix algorithm (see for instance [OH80]) have been largely negative. The algorithm behaves unpredictably, often requires human intervention, and is totally unsuitable for large scale applications. We believe this approach to compiling algebraic specifications to be a dead end and do not intend to pursue it any further.

In its present state the overall formalism is not yet an integrated or homogeneous whole. Full integration (Task T9) can only be attempted after more experience has been gained with the various subformalisms by applying them to selected test cases (Task T7).

3. ASSESSMENT OF THE VARIOUS FORMALISMS - GENERAL CONSIDERATIONS

The overall formalism and its subformalisms are not put to the test in isolation, but their behaviour and performance will be compared *informally* with that of other formalisms with which we have experience:

- The syntax definition subformalism will be compared with the METAL language of the MENTOR system developed at INRIA [KLMM83], the LEX lexical scanner generator [JOH79] and the YACC parser generator [LS79] (both of them widely used), and the syntax formalism used in the OBJ2 system [FGJM85].
- The merits of inference rules will be compared with those of algebraic semantics both in the case of static constraints as well as dynamic semantics and the theoretical relationship between both formalisms will be investigated.
- Both inference rules and algebraic semantics will be compared with denotational semantics and, in the case of static constraints, with attribute grammars.

The test cases selected must cover the range of language features we want to be able to model in some reasonable way, which unfortunately is difficult to make precise. Special attention will be paid to:

Syntax definition

- Definition of lexical syntax and prettyprinting.
- The acceptability of any limitations the syntax definition formalism may impose on the kind of concrete to abstract syntax mappings that can be expressed in it.
- Derivation of incremental parsers/prettyprinters from non-incremental definitions.

Static constraints

- Definition of type constraints in case of overloading and/or polymorphism.
- Definition of type constraints in case of languages using declaration-less type-from-context systems.
- Derivation of incremental type checkers from non-incremental static constraints definitions.

Dynamic semantics

- Modelling of error and exception handling.
- Modelling of lazy evaluation.
- Derivation of incremental evaluators from non-incremental dynamic semantics definitions.

Additional examples will have to test the modularisation component of the language definition formalism. In this case the requirements to be met do not follow from language features to be modelled, but from the functional requirements to be met by the environment generation system itself.

Modularisation of language definitions

- Control of exported names.
- Parameterised modules.
- Composition of modules containing sections in different formalisms.

See Annexe D4.A2 and Annexe D4.A3 for the test cases chosen. Many of the proposed test cases not only test the semantics definition formalism but also the syntax definition formalism of Annexe D4.A1.

REFERENCES

[BK82] J.A. Bergstra & J.W. Klop, "Conditional rewrite rules: confluency and termination", Report IW 198/82, CWI, 1982.

[EM85] H. Ehrig & B. Mahr, *Fundamentals of Algebraic Specification*, Vol. I, *Equations and Initial Semantics*, Springer-Verlag, 1985.

[FGJM85] K. Futatsugi, J.A. Goguen, J.P. Jouannaud & J. Meseguer, "Principles of OBJ2", *Conf. Record 12th Ann. ACM Symp. Principles of Programming Languages*, ACM, 1985, pp. 52-66.

[JOH79] S.C. Johnson, "YACC: yet another compiler-compiler", in: *UNIX Programmer's Manual*, Vol. 2B, Bell Laboratories, 1979.

[KLA83] Klaeren, H.A., *Algebraische Spezifikation*, Springer-Verlag, 1983.

[KLMM83] G. Kahn, B. Lang, B. Mélèse & E. Morcos, "METAL: a formalism to specify formalisms", *Science of Computer Programming*, 3(1983), pp. 151-188.

[LS79] M.E. Lesk & E. Schmidt, "LEX - A lexical analyzer generator", in: *UNIX Programmer's Manual*, Vol. 2B, Bell Laboratories, 1979.

[OH80] D.C. Oppen & G. Huet, "Equations and rewrite rules", in: R. Book (ed.), *Formal Languages: Perspectives and Open Problems*, Academic Press, 1980.

[PLO81] G.D. Plotkin, "A structural approach to operational semantics", DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

GIPE : CEC 348/A/T4/9
version 4 26-05-86

User Definable Syntax for Specification Languages

Annexe D4.A1 of Deliverable D4

J. Heering (CWI)
P. Klint (CWI)

A new formalism is introduced (both informally and formally) which allows concrete and abstract syntax of specification (and other) languages to be defined simultaneously. The new formalism can be combined with a variety of specification languages. By doing so these obtain fully general user definable syntax. Examples of this are given in the context of algebraic specifications.

1. INTRODUCTION

It is common practice to make a distinction between the *parse tree* of a text in a programming or specification language *L* and its *abstract syntax tree.* The parse tree is the derivation showing how the text in question can be derived from the start symbol of the grammar of *L.* Its interior nodes are non-terminals of the grammar and its leaves are the lexical tokens, such as keywords, operator symbols, identifiers, etc., making up the original text. The abstract syntax tree only contains the essential information describing the text; its interior nodes are the constructors (also called operators) of the language and the direct descendants of each node are its operands. The leaves of the abstract syntax tree are identifiers, integer constants, etc.

Let us first of all summarize the arguments for using some structured representation of programs or specifications instead of the text itself:

(1) By using a structured representation one may abstract from the details of the text. A single structured representation may correspond to a whole class of variations of a text obtained by using "abstractly equivalent" but textually different lexical conventions or syntactic variations.

(2) Using the textual representation makes it difficult to give a semantics based on equations or rewrite rules, because in this case all semantic actions have to be expressed as string transformations. Such transformations easily become ambiguous, however: combinations of strings arising during processing may (erroneously) be interpreted as language constructs not occurring in the original text. These ambiguities are avoided by using a bracketed (and thus structured) representation of the text.

(3) A structured representation allows more efficient processing and editing since repeated parsing of source text can be avoided.

Which structured representation should one use? The main arguments in favour of using abstract syntax trees rather than parse trees are:

(1) Parse trees may contain chains of non-terminals that describe the intermediate steps necessary for deriving a string from a certain non-terminal. These derivation chains are mostly redundant and do not concisely express the essential structure of the string.

(2) The syntactic definition of (a) priority and associativity of operators, and (b) syntactic iteration, i.e. lists of items, introduce non-terminals which need not appear in the abstract syntax tree.

(3) The use of parser generators accepting restricted classes of grammars (e.g. LR(k), LALR(k),

LL(k), etc.) may also lead to the introduction of "unnecessary" non-terminals.

(4) For different semantic purposes one may wish to associate different abstract syntax trees with one parse tree.

The major part of the translation between parse trees and abstract syntax trees is typically devoted to the elimination of non-terminals introduced in (1), (2) and (3). This is one of the areas where we attempt to improve upon existing formalisms.

In most compiler generation systems a distinction is made between the definition of lexical syntax and construction rules for lexical tokens, the definition of context-free syntax, and the translation rules from parse trees to abstract syntax trees. The formalisms used for defining them are different (e.g. regular expressions vs. BNF-notation).

In this paper we develop a syntax definition formalism — called *SDF* — allowing the simultaneous definition of concrete and abstract syntax. It has the following properties:

(1) Abstract syntax trees are described by (first-order) signatures.

(2) The formalism implicitly defines a translation from abstract syntax trees to parse trees.

(3) The need for introducing "unnecessary" non-terminals is eliminated. This is achieved (a) by using subsorts to eliminate undesirable derivation chains; (b) by using a separate mechanism for defining priority and associativity of operators; (c) by not restricting the class of acceptable grammars.

(4) The *concrete* (i.e. lexical as well as context-free) syntax is described by the same formalism.

(5) It can be used to add user definable syntax to any formalism based on first-order signatures.

In section 2 we discuss related work and describe two existing syntax definition formalisms: LEX/YACC and METAL. In section 3 we give an informal description of *SDF.* Next, in section 4, three simple examples are given using LEX/YACC, METAL and *SDF.* In section 5 we give a formal definition of *SDF.* The combination of *SDF* with various specification formalisms is discussed in section 6. A syntax definition of Pascal has been added as an appendix.

2. EXISTING FORMALISMS FOR THE DEFINITION OF SYNTAX

2.1. General

There have been many attempts to introduce user-definable syntax in programming languages. These ideas have led to user-definable syntax for operators in various programming languages (SNOBOL4, ALGOL68, PROLOG), and to several styles of macro-definitions (PL/I, LISP). Around 1970, there was much interest in so-called *extensible languages* (see, for instance, [IR70], [WE70], [ST75]). The aim of this line of research was to define a small base language in combination with a syntax definition mechanism. New language constructs could then be added to the base language by defining their syntax and by describing their semantics in terms of the base language. For various reasons, however, the overall goal of full syntactic and semantic language extensibility has never been completely achieved. Although it does not have a syntactic extension mechanism[1], SMALLTALK-80 may be viewed as the most successful extensible language in existence.

A successful method for defining language constructs is by means of *syntax-directed translations* ([IR61], [AU72]). LEX/YACC and METAL (to be discussed in the next sections) fall into this category. The LITHE system [SAN82] combines syntax-directed translation with classes. Apart from the fact that its lexical syntax is fixed, LITHE has user-definable concrete syntax[2]. Notational specifications [ODO85] constitute an alternative to syntax-directed translations.

In Annexe D4.A3 we give a complete algebraic specification of syntax and semantics of a small programming language. We are, however, not satisfied with the algebraic specification of syntax as presented there; this has motivated the development of *SDF*. Most closely related to our work are the user-definable *distfix* operators in HOPE [BMS80] and OBJ2 [FGJM85]. Our notion of subsorts was inspired by the subsort mechanism of OBJ2, but we use only a very restricted form of subsorts.

The main contributions of our formalism are:

(1) Uniform definition of lexical and context-free syntax in combination with first-order signatures, and

(2) the use of subsorts to describe (and eliminate) derivation chains that can be part of a parse tree but should not appear in the corresponding abstract syntax tree.

2.2. LEX and YACC

LEX and YACC are, respectively, the scanner generator and parser generator of the UNIX system. We restrict our description of LEX and YACC to two small examples, since both systems have been described extensively in the literature ([JOH79], [LS79], [ASU85]).

LEX uses regular expressions to describe the syntax of lexical tokens. These are compiled to tables for a deterministic finite automaton. A typical LEX definition for recognizing identifiers is:

```
    [a-z][a-z0-9]*  { id(yytext); }
```

The left-hand part of this rule describes the syntax of identifiers (i.e. a letter followed by zero or more letters or digits) and the right-hand part describes the action to be performed when the rule matches. Actions are written in the C programming language. In the above example the procedure `id` is called. The global variable `yytext` is part of the LEX/C interface and has as value a character array containing the text of the current token.

YACC uses a BNF-like notation for defining grammars and compiles these definitions to

1. SMALLTALK-72 — a predecessor of SMALLTALK-80 — did support syntactic extensibility: each class had to parse the messages sent to it explicitly. This feature has been replaced by a more limited scheme of keyword parameters in SMALLTALK-80. This new scheme results in more readable programs and allows a more efficient implementation.
2. The user-defined syntax must be LR(1).

tables for a shift/reduce parser. Input grammars have to satisfy the LALR(1) restriction. A part of a typical YACC definition for defining a `while`-statement is:

```
%token WHILE, DO, ENDWHILE

 ... definitions for expr and series ...

while : WHILE expr DO series ENDWHILE
                { $$ = mk-while( $2, $4 ); }
```

The rule starting with `%token` declares the lexical tokens of the language (defined separately by means of LEX definitions). The second rule defines the syntax for a `while`-statement. Its left-hand part gives the syntax definition, and its right-hand part the code to be executed when a `while`-statement is recognized. The value of variable **$***i* is the value returned by the *i*-th component of the rule after matching. The action associated with the rule returns a value in variable **$$**.

Syntax rules may contain alternatives (which are separated by the `|`-operator) but there is no mechanism for expressing repetition.

2.3. METAL

METAL ([KLMM83]) is the syntax-definition formalism of the MENTOR system. A language definition in METAL specifies:

(1) the lexical tokens of the language,

(2) its context-free syntax, and

(3) its abstract syntax.

The METAL compiler translates METAL specifications to specifications for an existing scanner/parser generator. The UNIX implementation of MENTOR, for instance, compiles METAL into input for LEX and YACC.

The actual form of lexical tokens is left unspecified in METAL definitions: these are specified in the language of the host scanner generator.

BNF-notation is used to define context-free syntax. A typical rule in a METAL definition is:

```
<while_stat> ::= while <exp> do <series> od ;
        while(<exp>, <series>)
```

Non-terminals are between angle brackets and terminals are just written as they are. Characters in terminals that conflict with the syntax notation have to be escaped (using a sharp sign). The formalism allows neither alternation nor repetition in syntax rules.

The part of the rule following the semicolon specifies the abstract syntax tree to be built for the construct. In the above example a tree labeled with the operator "`while`" with the abstract trees corresponding to `<exp>` and `<series>` as descendants. The abstract syntax of a language is defined by means of *functions* and *sorts* (or *operators* and *phyla* in METAL terminology). Functions are the constructors of abstract syntax trees. Functions with fixed arity are allowed to have descendants of different sorts. Functions with arity zero are the leaves of the abstract tree and represent the atoms of the language. Varyadic functions (also called *list functions*) are only allowed to have descendants of the same sort.

A typical definition of a function is:

```
while   -> EXP SERIES;
```

which defines the function `while` with two arguments which are respectively of sorts `EXP` and `SERIES`. The output sort of functions is specified in a separate definition (see below). The formalism allows the definition of list functions:

```
series   -> STATEMENT + ... ;
```

This defines the function `series` which may have one or more arguments, all of sort `STATEMENT`. Constructors for the empty list (e.g. `series-list`) and for prepending (e.g. `series-pre`) and appending (e.g. `series-post`) items to an existing list are implicitly defined.

The output sorts of functions are defined by enumerating all functions with a given output sort. For instance,

```
STATEMENT ::= assign if while;
```

specifies that the functions `assign`, `if` and `while` all have `STATEMENT` as output sort. Hence, the complete specification of the input and output sorts of, for instance, the function `while` expressed in METAL is

```
while      -> EXP SERIES
STATEMENT ::= assign if while;
```

which is equivalent to

$$while : EXP \times SERIES \rightarrow STATEMENT$$

using conventional mathematical notation. In the sequel we will use # instead of $\times$ to indicate the Carthesian product.

3. INFORMAL DESCRIPTION OF THE SYNTAX DEFINITION FORMALISM

One may characterize *SDF* roughly as "signatures with an integrated concrete syntax definition mechanism": we start with signatures — being a convenient formalism for expressing abstract syntax — and add to them mechanisms for defining lexical and context-free syntax.

Signatures consist of definitions of sorts, subsorts and constants and functions over these sorts and subsorts (see, for instance, [KLA83], [EM85] and [GM85]). For each (sub)sort s in the signature, we define the derived sorts s^* and $s+$, denoting lists with elements of sort s (containing respectively zero or more and one or more elements).

Two examples will illustrate the flavour of the signatures we have in mind. First, we define the signature of the Booleans, with one sort `BOOL`, the constants `true` and `false` and the functions `not`, `or` and `orl` (the or-function on lists of Booleans of arbitrary length):

```
module Booleans
begin
   sorts BOOL

   functions
      true  :                -> BOOL
      false :                -> BOOL
      not   : BOOL           -> BOOL
      or    : BOOL # BOOL    -> BOOL
      orl   : BOOL*          -> BOOL

end Booleans
```

As a second example, we define the signature describing a stack of Booleans, with sorts `STACK` (representing both empty and non-empty stacks) and `NESTACK` (representing non-empty stacks), with constant `empty-stack` and with functions `push` and `pop`. The subsort relation between `NESTACK` and `STACK` (denoted by <) is used to express the fact that `push` is defined on both empty and non-empty stacks (and has a non-empty stack as a result), but that `pop` is only defined on non-empty stacks. Note that module `Booleans`, defined above, is imported in module `Stack-of-Booleans`.

```
module Stack-of-Booleans
begin
   sorts STACK, NESTACK
   subsorts NESTACK < STACK

   functions
      empty-stack :                -> STACK
      push        : BOOL # STACK -> NESTACK
      pop         : NESTACK      -> BOOL

   imports Booleans

end Stack-of-Booleans
```

Signatures are extended with a mechanism for the definition of concrete syntax: definitions of constants and functions will not only have to specify their input and output sorts but also their concrete syntactic form. This syntactic information is mixed with the description of the input sorts. In this manner, the function definitions

```
BOOL v BOOL         -> BOOL
push BOOL on STACK -> NESTACK
```

introduce, for instance, new syntactic forms for, respectively, the functions `or` and `push`. Instead of the prefix terms `or(true,false)` and `push(true,empty-stack)` one should now write `true v false` and `push true on empty-stack`, respectively. The complete syntax definition for `Stack-of-Booleans` is:

```
module Stack-of-Booleans
begin
   sorts STACK, NESTACK
   subsorts NESTACK < STACK

   functions
      empty-stack              -> STACK
      push BOOL on STACK       -> NESTACK
      pop  NESTACK             -> BOOL

   imports Booleans

end Stack-of-Booleans
```

For derived sorts two forms are available:

(1) s* (or s +) stands for zero or more (one or more) repetitions of s.

(2) $\{s\ \ t\}$* (or $\{s\ \ t\}$+) stands for zero or more (one or more) repetitions of s *separated by* the symbol t.

Several other aspects of the concrete syntactic form of functions may be defined:

(1) The priority of a function relative to other functions. The usual priorities of the arithmetic functions + (addition), - (subtraction), * (multiplication), / (division) and `exp` (exponentiation) can, for instance, be defined by the following priority declaration:

```
priority (+, -) < (*, /) < exp
```

(2) If the syntactic representation of a function may be surrounded by parentheses (in order to change its priority), the attribute `par` is added to its definition.

(3) If a function is associative the attribute `assoc` is added to its definition.

(4) If the syntactic form of a function is part of the lexical syntax, the attribute `lex` is added to its definition. In this case, no layout symbols are allowed between the constituents of its syntactic representation.

In sections 4 and 5 we give more examples of the syntax definition formalism.

4. THREE EXAMPLES

Three examples are now given to illustrate the descriptive power of our formalism. They demonstrate the syntax definition of expressions, statements and lexical constants. For each example we give

(1) the desired concrete and abstract syntax,

(2) a definition in LEX/YACC,

(3) a definition in METAL, and

(4) a definition in *SDF.*

4.1. Expressions

This example consists of a grammar for simple arithmetic expressions. It illustrates the definition of operator priorities in the various formalisms.

4.1.1. Concrete and abstract syntax

The concrete syntax for arithmetic expressions is:

```
expr   ::= term | expr "+" term
term   ::= factor | term "*" factor
factor ::= id | "(" expr ")"
```

The desired abstract syntax is:

```
add     : EXPR # EXPR -> EXPR
mul     : EXPR # EXPR -> EXPR
id      : ID          -> EXPR
```

4.1.2. Definition in YACC

```
%token ID
%left '+'
%left '*'
%%
expr    : expr '+' expr           { $$ = add($1, $3); }
        | expr '*' expr           { $$ = mul($1, $3); }
        | '(' expr ')'            { $$ = $2; }
        | ID                      { $$ = id($1); }
        ;
```

YACC allows the definition of priority and associativity of operators. The lines `%left '+'` and `%left '*'` define + and * as left-associative operators. Consecutive lines define operators with increasing priority. Hence, * has higher precedence than +. The definitions of the functions `add`, `mul` and `id` are not shown; they are written in C and construct nodes of the abstract syntax tree.

4.1.3. Definition in Metal

```
definition of EXPRESSIONS is

    chapter 'expr-rules'
        rules
            <exp>                 ::= <term> ;
                <term>
            <exp>                 ::= <exp> #+ <term> ;
                plus(<exp>,<term>)
            <term>                ::= <factor> ;
                <factor>
            <term>                ::= <term> #* <factor> ;
                mul(<term>,<factor>)
            <factor>              ::= <id> ;
                <id>
            <factor>              ::= #( <exp> #) ;
                <exp>
            <id>                  ::= %ID ;
                id-atom(%ID)
        abstract syntax
            plus                  -> EXP EXP;
            mul                   -> EXP EXP;
            id                    -> implemented as IDENT;

            EXP                   ::= plus mul;
            ID                    ::= id;
    end chapter;
end definition
```

4.1.4. Definition in *SDF*

```
sorts EXPR, ID
subsorts ID < EXPR
priority + < *
functions
   EXPR + EXPR -> EXPR   {par, assoc}
   EXPR * EXPR -> EXPR   {par, assoc}
```

4.2. Statements

The following grammar for simple statements illustrates the treatment of syntactic iteration, i.e. the description of lists of syntactic notions.

4.2.1. Concrete and abstract syntax

The concrete syntax for statements is:

```
if     ::= "if" expr "then" series "else" series "endif"
while  ::= "while" expr "do" series "endwhile"
stat   ::= if | while
series ::= { stat ";" }*
```

The notation `{ <stat> ';' }*` is used to indicate lists of `<stat>`s separated by `';'` and is equivalent to

```
series ::= "" | series1
series1::= stat | stat ";" series1
```

The desired abstract syntax for statements is:

```
if     : EXPR # SERIES # SERIES -> STAT
while  : EXPR # SERIES          -> STAT
series : STAT*                  -> SERIES
```

4.2.2. Definition in YACC

```
%token IF, THEN, ELSE, ENDIF, WHILE, DO, ENDWHILE
%%

stat    : if                    { $$ = $1; }
        | while                 { $$ = $1; }
        ;
if      : IF expr THEN series ELSE series ENDIF
                                { $$ = if($2, $4, $6); }
        ;
while   : WHILE expr DO series ENDWHILE
                                { $$ = while($2,$4); }
        ;
series  :                       { $$ = emptyseries; }
        | series1               { $$ = $1; }
        ;

series1: stat                   { $$ = series($1, emptyseries); }
        | stat ';' series1      { $$ = series($1, $3); }
        ;
```

4.2.3. Definition in Metal

```
definition of STATEMENTS is

    chapter 'stat-rules'
        rules
            <stat>              ::= <if> ;
                <if>
            <stat>              ::= <while> ;
                <while>
            <if>                ::= if <exp> then <series>
                                         else <series> endif ;
                if(<exp>,<series>.1,<series>.2)
            <while>             ::= while <exp> do <series> endwhile ;
                while(<exp>,<series>)
            <series>            ::=  ;
                series-list(())
            <series>            ::= <series1> ;
                <series1>
            <series1>           ::= <stat> ;
                series-list(<stat>)
            <series1>           ::= <stat> #; <series1> ;
                series-pre (<stat>,<series1>)
        abstract syntax
            if                  -> EXP SERIES SERIES;
            while               -> EXP SERIES;
            series              -> STAT * ... ;
            series1             -> STAT + ... ;

            SERIES              ::= series series1;
            STAT                ::= if while;
    end chapter;
end definition
```

4.2.4. Definition in *SDF*

```
sorts EXPR, STAT, SERIES
functions
   if EXPR then SERIES else SERIES endif          -> STAT
   while EXPR do SERIES endwhile                  -> STAT
   { STAT ; }*                                    -> SERIES
```

4.3. Lexical constants

This example illustrates the definition of natural numbers and identifiers as lexical constants.

4.3.1. Concrete and abstract syntax

The concrete syntax for natural numbers and identifiers is:

```
int       ::= digit+
id        ::= letter id-tail*
id-tail   ::= letter | digit
letter    ::= "a" | ... | "z"
digit     ::= "0" | ... | "9"
```

The desired abstract syntax is:

```
int : DIGIT+              -> INT
id  : LETTER # ID-TAIL* -> ID
a   :                     -> LETTER
 ...
z   :                     -> LETTER
0   :                     -> DIGIT
 ...
9   :                     -> DIGIT
i1  : LETTER              -> ID-TAIL
i2  : DIGIT               -> ID-TAIL
```

4.3.2. Definition in LEX

Both YACC and METAL rely on LEX for the definition of lexical syntax:

```
%%
[a-z][a-z0-9]*          { id(yytext); }
[0-9]+                  { int(yytext); }
%%
```

`id` and `int` are C functions (not shown here) which construct lexical tokens on the basis of the input text recognized.

4.3.3. Definition in *SDF*

```
sorts DIGIT, LETTER, INT, ID, ID-TAIL
subsorts (DIGIT, LETTER) < ID-TAIL
functions
   a                    -> LETTER {lex}
    ...
   z                    -> LETTER {lex}
   0                    -> DIGIT  {lex}
    ...
   9                    -> DIGIT  {lex}
   DIGIT+               -> INT    {lex}
   LETTER ID-TAIL*      -> ID     {lex}
```

The subsort mechanism is used to define a “union” of sorts, i.e. `ID-TAIL` represents both `DIGIT` and `LETTER`.

5. FORMAL DEFINITION OF SDF

We will now give a formal definition of *SDF*. Section 5.1 contains some preliminary definitions. Section 5.2 gives an overview of the definition and summarizes sections 5.3-5.9.

5.1. Preliminary definitions

First we introduce the notions of context-free grammar, parse tree, and reduced parse tree.

DEFINITION 5.1.1. A *context-free grammar* is a 4-tuple $(N,\Theta,PROD,R)$ where

(1) N is a finite set of *non-terminal symbols.*

(2) Θ is a finite set of *terminal symbols*, disjoint from N.

(3) $PROD$ is a finite subset of $N\times(N\cup\Theta)^*$. An element (α,β) in $PROD$ will be written $\alpha\Rightarrow\beta$ (or, alternatively, $\alpha::=\beta$) and is called a *production.*

(4) R is a symbol in N called the *root* or *start symbol.*

DEFINITION 5.1.2. The *language* $L(G)$ defined by a context-free grammar $G=(N,\Theta,PROD,R)$ is defined as $L(G) = \{\theta\in\Theta^* \mid R\overset{*}{\Rightarrow}\theta\}$.

DEFINITION 5.1.3. A *labeled ordered tree* over an alphabet A is either

(1) $R\in A$, or

(2) $<R\ Q_1\ \cdots\ Q_n>$ with $n\geq 1$, $R\in A$ and Q_i a labeled tree over A.

We define a function *label* on labeled ordered trees over an alphabet A as follows:

(1) $label(R)=R$, if $R\in A$, and

(2) $label(<R\ Q_1\ \cdots\ Q_n>)=R$.

DEFINITION 5.1.4. A *parse tree* P for a context-free grammar $G=(N,\Theta,PROD,R)$ is a labeled ordered tree over the alphabet $N\cup\Theta\cup\{\epsilon\}$ (ϵ denotes the empty string) such that

(1) $label(P)=R$.

(2) If $P=<R\ \epsilon>$ then $R::=\epsilon$ should be a rule in G.

(3) If $P=<R\ Q_1\ \cdots\ Q_n>$ $(n\geq 1)$ then $R::=label(Q_1)\ \cdots\ label(Q_n)$ should be a rule in G and either $Q_i\in\Theta$ or Q_i is a parse tree for the grammar $(N,\Theta,PROD,label(Q_i))$.

The set of all parse trees for a grammar G is denoted by $PT(G)$. A grammar $G=(N,\Theta,PROD,R)$ gives rise to grammars $G_Q=(N,\Theta,PROD,Q)$ for each $Q\in N$. We will always write $PT_Q(G)$ instead of $PT(G_Q)$.

DEFINITION 5.1.5. The *frontier relation* $\phi\subseteq PT(G)\times L(G)$ consists of the pairs (P,θ) with $P\in PT(G)$ and θ the string consisting of all terminal symbols (leaves) encountered during a preorder traversal of P.

Next, we introduce many-sorted signatures and abstract syntax trees over these signatures.

DEFINITION 5.1.6. A *many-sorted signature with variables* Σ is a 4-tuple $(S,OP,X,type)$ where

(1) S is a finite set of *sorts.*

(2) OP is a finite set of *constant/function symbols.*

(3) X is a finite set of *variables*, disjoint from OP.

(4) *type* is a pair of functions $(type_{OP},type_X)$ which assign types to the elements of OP and X respectively. The elements of OP are typed by $type_{OP} : OP \rightarrow S^*\times S$. We will usually write $f{:}s_1\#\ \cdots\ \#s_n{\rightarrow}s$ instead of $type_{OP}(f) = (s_1\cdots s_n,s)$ $(n\geq 0)$. Function symbols c with $c{:}{\rightarrow}s$ for some s are called *constant symbols* of sort s. The elements of X are typed by $type_X : X \rightarrow S$. We will usually write $x{:}{\rightarrow}s$ instead of $type_X(x)=s$.

The fact that $type_{OP}$ and $type_X$ are (single valued) functions and that $OP\cap X=\emptyset$ means that we do not allow overloaded function or variable symbols in signatures.

DEFINITION 5.1.7. An *abstract syntax tree* of sort s for a many-sorted signature with variables $\Sigma=(S,OP,X,type)$ is a labeled ordered tree over the alphabet $OP\cup X$ such that

(1) t is a constant symbol or variable of sort s.

(2) t is of the form $f(t_1, \ldots, t_n)$ with $f{:}s_1 \# \cdots \# s_n \to s$ for some $s_1, \ldots, s_n$ and with t_i an abstract syntax tree of sort s_i over Σ for all $1 \leqslant i \leqslant n$.

We further define $AT_s(\Sigma)$ as the set of all abstract syntax trees of sort s over Σ and $AT(\Sigma)$ as the set of all abstract syntax trees over Σ, i.e. $AT(\Sigma) = \bigcup\limits_{s \in S} AT_s(\Sigma)$.

5.2. Overview of the *SDF*-definition

To keep the size of the definition manageable, five (sub)formalisms $SDF^0, \ldots, SDF^4$ are introduced, each of which contains a new feature. SDF^0 is the simplest one and allows the basic association of concrete representations with abstract syntax trees over a many-sorted signature (section 5.3). The formalisms $SDF^i, i>0$, introduce the following features:

SDF^1: lexical syntax (section 5.4)
SDF^2: priorities (section 5.5)
SDF^3: subsorts (section 5.6)
SDF^4 ($= SDF$): lists (section 5.7)

The relationships between the various levels are shown in figure 1. A basic syntax definition D^0 gives rise to a signature on the one hand and a context-free grammar on the other hand. The corresponding sets $AT(D^0)$ and $PT(D^0)$ of abstract syntax trees and parse trees are isomorphic. In fact, a basic syntax definition is simply a context-free grammar (but with all rules written backward) in which a distinction is made between rules defining constants and functions and rules defining variables.

For each level i, $i>0$, we define a translation $\tau^i : SDF^i \to SDF^{i-1}$. For each syntax definition D^i at level i, $i \geqslant 0$, we define the corresponding set of abstract syntax trees AT^i, the corresponding frontier relation ϕ^i, and the corresponding language L^i. Using τ^i, the abstract syntax trees at all levels can ultimately be expressed as abstract syntax trees at level 0. AT^0, AT^1 and AT^2 are equal; they are not shown in figure 1.

5.3. SDF^0: basic syntax definitions

5.3.1. Definition of SDF^0

DEFINITION 5.3.1. A *basic syntax definition* D^0 is a 4-tuple (S, Θ, OP, X) where

(1) S is a finite set of *sorts*.

(2) Θ is a finite set of *terminal symbols*, disjoint from S.

(3) $OP \subseteq (S \cup \Theta)^* \times S$ is a finite set of *constant/function declarations*. We will usually write $a_0 s_1 a_1 \cdots s_n a_n \to s$ instead of $(a_0 s_1 a_1 \cdots s_n a_n, s)$ for elements of OP ($n \geqslant 0$, $a_i \in \Theta^*$, $s_i \in S$). Function declarations of the form $a \to s$ ($a \in \Theta^*$) are called *constant declarations.*

(4) $X \subseteq \Theta^* \times S$ is a finite set of *variable declarations*, disjoint from OP. We will usually write $a \to s$ instead of (a, s).

We will now define for each SDF^0-definition D^0:

(a) The derived signature $\Sigma(D^0)$ (defining a set of abstract syntax trees $AT(D^0) = AT(\Sigma(D^0))$).

(b) The derived context-free grammar $G(D^0)$ (defining a set of parse trees $PT(D^0) = PT(G(D^0))$, a language $L(D^0) = L(G(D^0))$, and a frontier relation $\phi^0 \subseteq PT(D^0) \times L(D^0)$).

(c) A bijective derivation function $\delta : AT(D^0) \to PT(D^0)$.

The relationships between these domains and functions are shown in figure 2.

DEFINITION 5.3.2. The *derived signature* $\Sigma(D^0)$ of a basic syntax definition $D^0 = (S, \Theta, OP, X)$ is the signature $(S, OP', X', type)$ with

(1) $OP' \subseteq (S \cup \Theta)^*$ such that $a_0 s_1 a_1 \cdots s_n a_n s \in OP'$ if $a_0 s_1 a_1 \cdots s_n a_n \to s \in OP$;

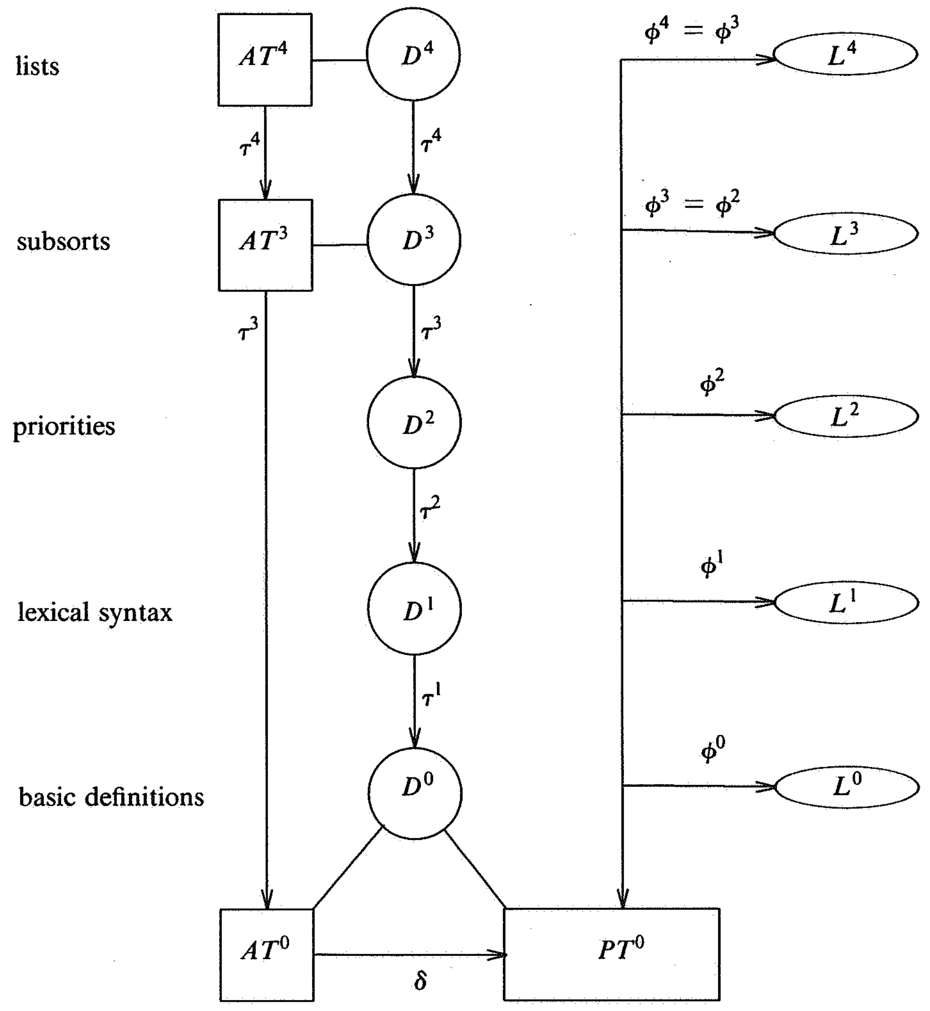


Figure 1. Overview of the formal definition of *SDF.*

(2) $X' \subseteq (S \cup \Theta)^*$ such that $as \in X'$ if $a \rightarrow s \in X$;

(3) $type = (type_{OP'}, type_{X'})$ such that

(a) $type_{OP'}(a_0 s_1 a_1 \cdots s_n a_n s) = s_1 \# \cdots \# s_n \rightarrow s$ $(n \geqslant 0)$,

(b) $type_{X'}(as) = \rightarrow s$.

We will usually write $\overline{a_0 s_1 a_1 \cdots s_n a_n \rightarrow s}$ instead of $a_0 s_1 a_1 \cdots s_n a_n s$, i.e. $\overline{a_0 s_1 a_1 \cdots s_n a_n \rightarrow s} \in OP'$ corresponds to $a_0 s_1 a_1 \cdots s_n a_n \rightarrow s \in OP$.

Note that we generate somewhat bizarre names in the derived signature: a complete declaration as it appears in the basic syntax definition is used as a *name* in the derived signature. These names are only introduced in the definition and will not be visible to the user of *SDF.* A declaration occurring more than once in *OP* corresponds to a single name in *OP'*.

DEFINITION 5.3.3. For a given basic syntax definition D^0 with derived signature $\Sigma(D^0)$, we define the set of *abstract syntax trees of sort s* $AT_s(D^0)$ as $AT_s(\Sigma(D^0))$ and the set of all abstract syntax trees $AT(D^0)$ as $AT(\Sigma(D^0))$ (see section 5.1).

DEFINITION 5.3.4. The *derived context-free grammar* $G(D^0)$ of a basic syntax definition $D^0 = (S, \Theta, OP, X)$ is the context-free grammar $(S \cup \{R\}, \Theta, PROD, R)$ where

(1) R is a new (start)symbol, i.e. $R \notin S \cup \Theta$.

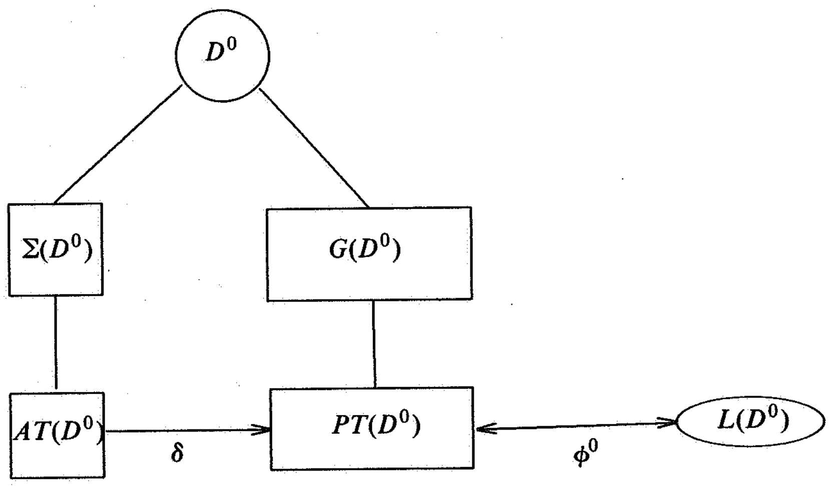


Figure 2. Relationships between the various domains and functions associated with an SDF^0-definition D^0.

(2) *PROD* consists of

(a) productions $R::=s$ for each sort $s \in S$;

(b) productions $s::=a_0 s_1 a_1 \cdots s_n a_n$ for each constant/function declaration $a_0 s_1 a_1 \cdots s_n a_n \rightarrow s$ in OP;

(c) productions $s::=a$ for each variable declaration $a \rightarrow s$ in X.

Essentially, $G(D^0)$ is obtained from D^0 by "reversing all rules".

DEFINITION 5.3.5. For a basic syntax definition $D^0=(S,\Theta,OP,X)$ the *frontier relation* $\phi^0 \subseteq PT(D^0) \times \Theta^*$ is defined as $\phi^0 = \phi$ (def. 5.1.5).

DEFINITION 5.3.6. The *language* $L(D^0)$ of a basic syntax definition D^0 with derived grammar G^0 is defined as $L(D^0)=L(G^0)$ (def. 5.1.2.).

THEOREM 5.3.1. For a basic syntax definition D^0:

$$\theta \in L(D^0) \Leftrightarrow \phi^0\ (p,\theta) \text{ for some } p \in PT(D^0).$$

PROOF. Apply theorem 2.11 of [AU72] with $G=G(D^0)$.

DEFINITION 5.3.7. For a given basic syntax definition $D^0 = (S,\Theta,OP,X)$ with derived signature $\Sigma^0 = (S,OP',X',type)$ and derived grammar $G^0 = (S \cup \{R\},\Theta,PROD,R)$ the *derivation function* $\delta_s : AT_s(\Sigma^0) \rightarrow PT_s(G^0)$ $(s \in S)$ is defined as follows

(1) For all $x \in X'$ with $x=\overline{a \rightarrow s}$ define: $\delta_s(x) = <s\ a>$;

(2) For all $f \in OP'$ with $f=\overline{a \rightarrow s}$ define: $\delta_s(f) = <s\ a>$;

(3) For all $f(t_1, \ldots, t_n)$ with $f=\overline{a_0 s_1 a_1 \cdots s_n a_n \rightarrow s}$ $(n \geq 1)$ define:

$$\delta_s(f(t_1, \ldots, t_n)) = <s\ a_0\ \delta_{s_1}(t_1)\ a_1\ \cdots\ \delta_{s_n}(t_n)\ a_n>.$$

The derivation function $\delta : AT(\Sigma^0) \rightarrow PT(G^0)$ is defined by $\delta(t) = <R\ \delta_s(t)>$ for $t \in AT_s(\Sigma^0)$.

THEOREM 5.3.2. The derivation function δ corresponding to a basic syntax definition D^0 is bijective.

PROOF. Assume $D^0 = (S,\Theta,OP,X)$ and let $\Sigma^0 = (S,OP',X',type)$ and $G^0 = (S \cup \{R\},\Theta,PROD,R)$ be respectively the derived signature and the derived grammar of D^0. Construct γ_s and γ which are the inverse functions of δ_s and δ respectively.

If $<s\ Q_1 \cdots Q_n> \in PT_s(G^0)$ then

$s::=label(Q_1) \cdots label(Q_n) \in G^0$,
$label(Q_1) \cdots label(Q_n) \rightarrow s \in OP$, and $\overline{label(Q_1) \cdots label(Q_n) \rightarrow s} \in OP'$.

Define $\gamma_s : PT_s(G^0) \rightarrow AT_s(\Sigma^0)$ $(s \in S)$ as follows

$$\gamma_s(<s\ Q_1\ \cdots\ Q_n>) = \overline{label(Q_1) \cdots label(Q_n) \rightarrow s}(\gamma_{label(Q_{i_1})}(Q_{i_1}), \cdots, \gamma_{label(Q_{i_k})}(Q_{i_k}))$$

with $1 \leqslant i_1 < \cdots < i_k \leqslant n$ $(k \geqslant 0)$ such that $label(Q_{i_j}) \in S$ $(1 \leqslant j \leqslant k)$ and $label(Q_i) \in \Theta$ $(i \neq i_j)$.
(k is the arity of $\overline{label(Q_1) \cdots label(Q_n) \rightarrow s}$.)
We now have $\delta_s \gamma_s = id_{PT_s(G^0)}$ and $\gamma_s \delta_s = id_{AT_s(\Sigma^0)}$.

For all parse trees of the form $<R\ Q_1\ \cdots\ Q_n> \in PT(G^0)$ we have $n=1$ and $label(Q_1) \in S$ (this is due to the particular form of the derived grammar G^0 in which all rules for the startsymbol R have the form $R::=Q$). Define γ as follows: $\gamma(<R\ Q>) = \gamma_{label(Q)}(Q)$ with $R::=label(Q) \in G^0$ and $label(Q) \in S$. We now have $\delta\gamma = id_{PT(G^0)}$ and $\gamma\delta = id_{AT(\Sigma^0)}$. □

Our final concern is the *expressive power* of basic syntax definitions. For each basic syntax definition we construct a translation to abstract syntax trees (this is achieved by δ^{-1}). It is natural to compare the translation δ^{-1} with syntax-directed translations as, for instance, defined in [AU72]. Syntax-directed translations are recipes for string-to-string translations. With each non-terminal in the grammar two rules are associated: a *recognition rule* (consisting of a sequence of terminals and non-terminals) and a *translation rule* (consisting of a sequence of arbitrary terminals and a permutation of the non-terminals in the recognition rule). In *simple syntax-directed translations*, the non-terminals in recognition and translation rule occur in the same order. Without proof we state that the translation δ^{-1} as generated for each basic syntax definition, is a simple syntax-directed translation.

5.3.2. Concrete representation of basic syntax definitions

We will use a concrete representation for syntax definitions in accordance with the following rules:

(1) All names of sorts are listed after the keyword `sorts`. The names of sorts are in upper case letters.

(2) The set Θ of terminals is not given explicitly, but can be derived from the function and variable declarations. In cases of ambiguity, terminals may be surrounded by double quotes.

(3) The set OP is given by listing all constant/function declarations after the keyword `functions`.

(4) The set X is given by listing all variable declarations after the keyword `variables`.

EXAMPLE 5.3.1. Consider the following basic syntax definition:

```
sorts BOOL, STACK
functions
        true                            -> BOOL
        false                           -> BOOL
        BOOL v BOOL                     -> BOOL
        ~ BOOL                          -> BOOL
        empty                           -> STACK
        push BOOL on STACK              -> STACK
```

The derived signature is :

```
sorts BOOL, STACK
function
```

$$\begin{array}{llll}
\overline{\texttt{true -> BOOL}} & : & & \texttt{-> BOOL} \\
\overline{\texttt{false -> BOOL}} & : & & \texttt{-> BOOL} \\
\overline{\texttt{BOOL v BOOL -> BOOL}} & : & \texttt{BOOL \# BOOL} & \texttt{-> BOOL} \\
\overline{\sim\texttt{ BOOL -> BOOL}} & : & \texttt{BOOL} & \texttt{-> BOOL} \\
\overline{\texttt{empty -> STACK}} & : & & \texttt{-> STACK} \\
\overline{\texttt{push BOOL on STACK -> STACK}} & : & \texttt{BOOL \# STACK} & \texttt{-> STACK}
\end{array}$$

Some abstract syntax trees over the derived signature are:

$$\overline{\texttt{true -> BOOL}}$$
$$\overline{\texttt{BOOL v BOOL -> BOOL}}(\overline{\sim\texttt{ BOOL -> BOOL}}(\overline{\texttt{false -> BOOL}}), \overline{\texttt{true -> BOOL}})$$
$$\overline{\texttt{push BOOL on STACK -> STACK}}(\overline{\texttt{true -> BOOL}}, \overline{\texttt{empty -> STACK}}).$$

The derived grammar is:

```
R     ::= BOOL
R     ::= STACK
BOOL  ::= true
BOOL  ::= false
BOOL  ::= BOOL v BOOL
BOOL  ::= ~ BOOL
STACK ::= empty
STACK ::= push BOOL on STACK
```

Examples of parse trees for this grammar are:

```
<R <BOOL true>>,

<R <BOOL <BOOL ~ <BOOL false>> v <BOOL true>>>

<R <STACK <push <BOOL true> on <STACK empty>>>>
```

These same parse trees are displayed graphically in figure 3. We also have, for instance,

$$\delta(\overline{\texttt{BOOL v BOOL -> BOOL}}(\overline{\sim\texttt{ BOOL -> BOOL}}(\overline{\texttt{false -> BOOL}}), \overline{\texttt{true -> BOOL}}))$$
$$= \texttt{<R <BOOL <BOOL }\sim\texttt{ <BOOL false>> v <BOOL true>>>}$$

and

$$\phi^0 \; (\texttt{<R <BOOL <BOOL }\sim\texttt{ <BOOL false>> v <BOOL true>>>, }\sim\texttt{ false v true}).$$

Hence,

$$\phi^0\ (\delta(\overline{\texttt{BOOL v BOOL -> BOOL}}(\overline{\texttt{\textasciitilde BOOL -> BOOL}}(\overline{\texttt{false -> BOOL}}),\overline{\texttt{true -> BOOL}})),$$
$$\texttt{\textasciitilde false v true }).$$

In this particular case, the concrete representation is ambiguous, because

$$\overline{\texttt{\textasciitilde BOOL -> BOOL}}(\overline{\texttt{BOOL v BOOL -> BOOL}}(\overline{\texttt{false -> BOOL}},\overline{\texttt{true -> BOOL}}))$$

has the same concrete representation. This is due to ϕ^0 because, according to theorem 5.3.2, δ is a bijection.

5.4. SDF^1: syntax definitions with layout

In this section we add a lexical syntax definition facility to SDF^0 by allowing an additional alphabet Λ of layout symbols and an attribute **lex** to all declarations in the syntax definition that correspond to lexical tokens. In the textual representation of these functions no layout symbols may occur, while these are allowed in the textual representation of all other functions.

DEFINITION 5.4.1. A *syntax definition with layout* D^1 is a 6-tuple $(S,\Theta,OP,X,\Lambda,\Pi)$ such that (S,Θ,OP,X) is a basic syntax definition, Λ is a finite set of layout symbols (disjoint from Θ and S), and $\Pi : OP \cup X \rightarrow 2^{\{\mathbf{lex}\}}$ is a function that may associate the attribute **lex** with each element of OP and X.

Note that we use the powerset notation $2^{\{\mathbf{lex}\}}$ to indicate either $\{\mathbf{lex}\}$ or the empty set. This anticipates the introduction of several other attributes in the subformalism SDF^2.

DEFINITION 5.4.2. The *translation* $\tau^1 : SDF^1 \rightarrow SDF^0$ is defined as follows: let $D^1=(S,\Theta,OP,X,\Lambda,\Pi)$, then $\tau^1(D^1) = (S,\Theta,OP,X)$.

DEFINITION 5.4.3. The *abstract syntax trees* and *parse trees* for an SDF^1-definition D^1 are defined as $AT(D^1) = AT(\tau^1(D^1))$, and $PT(D^1) = PT(\tau^1(D^1))$.

DEFINITION 5.4.4. For an SDF^1-definition $D^1 = (S,\Theta,OP,X,\Lambda,\Pi)$ the *layout relation* $layout_s \subseteq PT_s(D^1)\times(\Theta\cup\Lambda)^*$ is defined as follows. Let $p = <s\ Q_1 \cdots Q_n>$ be some parse tree in $PT_s(D^1)$ and let $f = label(Q_1) \cdots label(Q_n) \rightarrow s \in OP$ be the corresponding declaration. Now define $layout_s$ as follows:

(1) If $\mathbf{lex} \in \Pi(f)$ then $layout_s(p,\ \alpha_1 \cdots \alpha_n)$ where $\alpha_i = Q_i$ if $Q_i \in \Theta$ and $\phi^0\ (Q_i,\alpha_i)$ otherwise.

(2) If $\mathbf{lex} \notin \Pi(f)$ then $layout_s(p,\ \lambda_0\alpha_1\lambda_1 \cdots \alpha_n\lambda_n)$ where $\lambda_i \in \Lambda^*$ and $\alpha_i = Q_i$ if $Q_i \in \Theta$ and $layout_{s_i}(Q_i,\alpha_i)$ otherwise (where $Q_i = <s_i\gamma_1 \cdots \gamma_{n_i}>$).

DEFINITION 5.4.5. For an SDF^1-definition $D^1 = (S,\Theta,OP,X,\Lambda,\Pi)$, a parse tree $p \in PT(D^1)$ and a subtree $q \in PT_s(D^1)$ of p, q is a *maximal lexical subtree* of p if and only if

(1) the function declaration corresponding to q has attribute **lex**;

(2) q is not a (direct or indirect) descendant of a subtree of p corresponding to a function declaration with attribute **lex**.

DEFINITION 5.4.6. For an SDF^1-definition $D^1 = (S,\Theta,OP,X,\Lambda,\Pi)$, a parse tree $p \in PT_s(D^1)$, and a string $\theta \in (\Theta\cup\Lambda)^*$ with $layout_s(p,\theta)$ we say that p represents a *lexically longest match* of θ ($llm(p,\theta)$) if for each maximal lexical subtree $q \in PT_t(D^1)$ of p the following holds:

(1) $layout_t(q,\beta)$ and $\theta = \alpha\beta\gamma$, with $\alpha,\beta,\gamma \in (\Theta\cup\Lambda)^*$.

(2) no other parse tree $q' \in PT_{t'}(D^1)$ exists such that

  (a) $layout_{t'}(q',\beta\beta')$, with $\beta' \in (\Theta\cup\Lambda)+$;

  (b) β' is a prefix of γ;

  (c) q' corresponds to a function declaration with attribute **lex**.

DEFINITION 5.4.7. For an SDF^1-definition $D^1 = (S,\Theta,OP,X,\Lambda,\Pi)$ the *frontier relation*

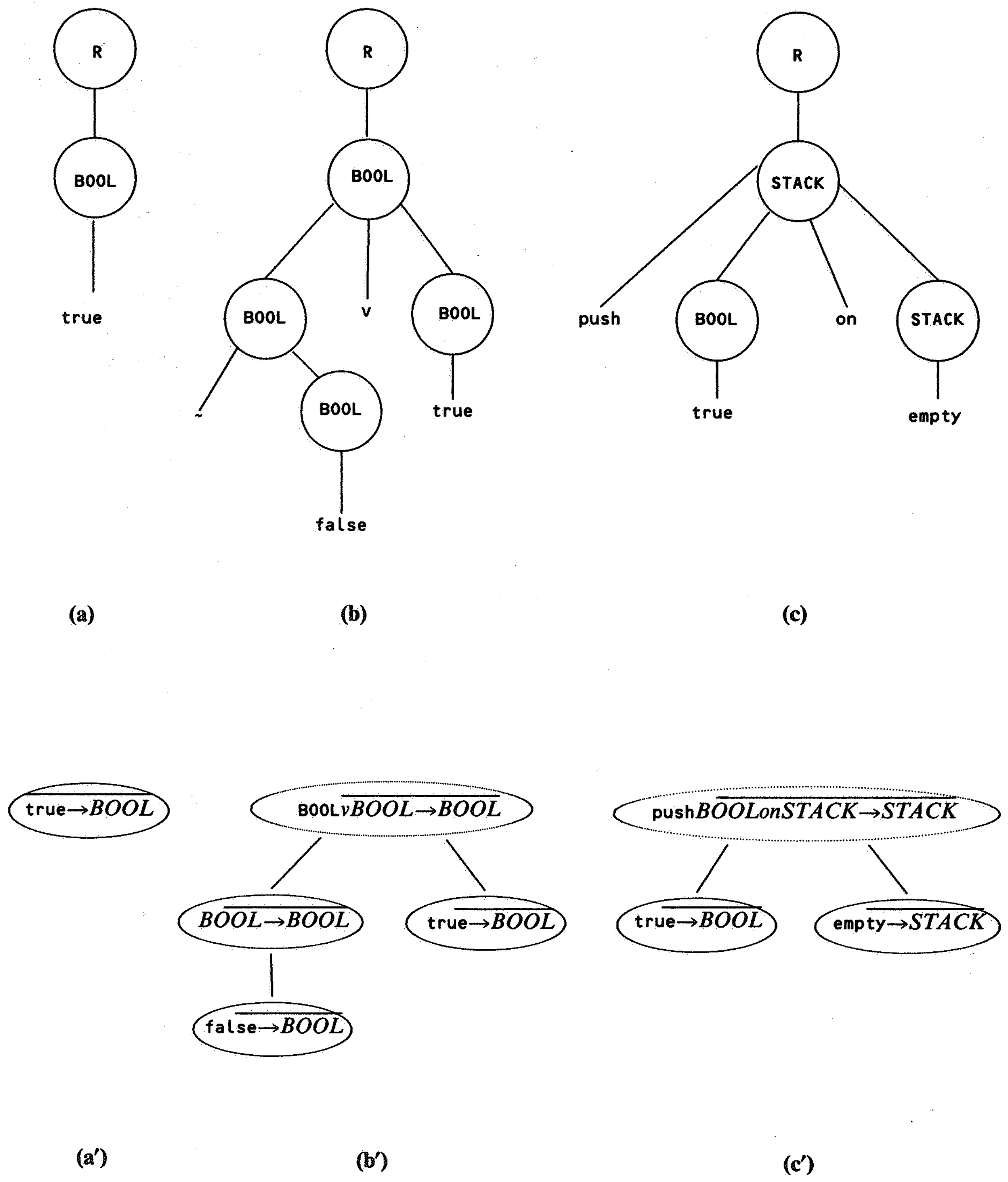


Figure 3. Examples of parse trees and corresponding abstract syntax trees.

$\phi^1 \subseteq PT(D^1)\times(\Theta\cup\Lambda)^*$ is defined by

$$\phi^1\ (p,\theta)\Leftrightarrow layout_s(p,\theta)\wedge llm(p,\theta)$$

for some $p\in PT_s(D^1)$ $(s\in S)$ and $\theta\in(\Theta\cup\Lambda)^*$.

DEFINITION 5.4.8. The *language* $L(D^1)$ of an SDF^1-definition $D^1 = (S,\Theta,OP,X,\Lambda,\Pi)$ is the subset of $(\Theta\cup\Lambda)^*$ such that

$\theta \in L(D^1) \Leftrightarrow \phi^1\ (p,\theta)$ for some $p \in PT(D^1)$.

EXAMPLE 5.4.1. See section 4.3.3. for an SDF^1-definition that defines identifiers and integer constants.

EXAMPLE 5.4.2. The following SDF^1-definition specifies two-bit binary numbers:

```
sorts BIT, BNUM
layout " "
functions
        0               -> BIT  {lex}
        1               -> BIT  {lex}
        BIT             -> BNUM {lex}
        BIT BIT         -> BNUM {lex}
        BNUM + BNUM     -> BNUM
        BNUM BNUM       -> BNUM
```

Binary numbers consist of one or two bits; two operations are defined: addition (denoted by `+`) and multiplication (denoted by the juxtaposition of two binary numbers). The layout alphabet Λ consists of the space character. A legal abstract syntax tree over the derived signature of this syntax definition is, for instance,

$$\begin{aligned}&\overline{\texttt{BNUM + BNUM -> BNUM}}(\overline{\texttt{BIT BIT -> BNUM}}(\overline{\texttt{1 -> BIT}},\overline{\texttt{1 -> BIT}}),\\ &\qquad\qquad\overline{\texttt{BIT BIT -> BNUM}}(\overline{\texttt{0 -> BIT}},\overline{\texttt{0 -> BIT}})).\end{aligned}$$

$\phi^1\ \delta$ associates, among other ones, the following concrete representations with this abstract syntax tree: `11+00`, `11 +00`, `11  +00`, `11+ 00`, `11+  00` or `11 + 00` but it will never associate strings like `1 1+00` or `11+0 0` with it. Note that, for instance, `10` will always be interpreted as a two-bit, binary number and not as a binary multiplication with operands `1` and `0`. This is due to the preference of a lexically longest match. Of course, `1 0` can *only* be interpreted as a binary multiplication.

EXAMPLE 5.4.3. The following syntax definition with layout illustrates that keywords and identifiers may coincide:

```
sorts EXP, STAT, ID
layout " "
functions
        if EXP then STAT        -> STAT
        ID := EXP               -> STAT
        x                       -> ID
        y                       -> ID
        if                      -> ID
        then                    -> ID
```

If we assume that `EXP` represents arithmetic expressions, then the following strings are acceptable and unambiguous:

```
x := 2 * y
if := then / 4
if if / then = 0 then then := 3 * if
```

This shows that we can express the use of keywords as identifiers as in, for instance, PL/I and FORTRAN. This scheme was chosen to accommodate the composition of *SDF*-definitions (see section 6.3). We can express a reserved word strategy (i.e. certain identifiers may not be used, they are reserved as keywords) as in, for instance, PASCAL and C, by using priority declarations (see section 5.5). We are *not* able to express concisely, identifiers and keywords with embedded layout characters as in, for instance, FORTRAN and ALGOL68.

5.5. SDF^2: syntax definitions with priorities

In this section, we introduce *priority relations* between the declarations in a syntax definition. These serve the purpose of resolving parsing conflicts (and thus ambiguities) between rules.

DEFINITION 5.5.1. A *syntax definition with priorities* D^2 is a syntax definition with layout $(S,\Theta,OP,X,\Lambda,\Pi)$ in which

(1) a partial order is defined on the declarations in OP (this ordering relation is denoted by $<$);

(2) Θ contains the distinguished symbols (and) denoting opening and closing parenthesis;

(3) $\Pi : OP \cup X \rightarrow 2^{\{\mathbf{lex},\mathbf{assoc},\mathbf{left-assoc},\mathbf{right-assoc},\mathbf{par}\}}$ is a function that may associate the attributes **lex** (lexical syntax), **assoc** (associative function), **left−assoc** (left-associative function), **right−assoc** (right-associative function), and **par** (concrete representation of function may be surrounded by parentheses) to elements of OP and X. The attributes **assoc**, **left−assoc**, and **right−assoc** are mutually exclusive.

DEFINITION 5.5.2. The *translation* $\tau^2 : SDF^2 \rightarrow SDF^1$ is defined as follows: let $D^2 = (S,\Theta,OP,X,\Lambda,\Pi)$, then $\tau^2(D^1) = (S,\Theta,OP,X,\Lambda,\Pi')$, where $\Pi'(f) = \Pi(f) \cap \{\mathbf{lex}\}$, $f \in OP \cup X$.

DEFINITION 5.5.3. The *abstract syntax trees* and *parse trees* for an SDF^2-definition D^2 are defined by $AT(D^2) = AT(\tau^2(D^2))$, and $PT(D^2) = PT(\tau^2(D^2))$.

DEFINITION 5.5.4. Given an SDF^2-definition $D^2 = (S,\Theta,OP,X,\Lambda,\Pi)$, two parse trees $p = <s\ u_1 \cdots u_k> \in PT_s(D^2)$ and $q = <s'\ v_1 \cdots v_l> \in PT_{s'}(D^2)$, and corresponding declarations $f = label(u_1) \cdots label(u_k) \rightarrow s \in OP$ and $g = label(v_1) \cdots label(v_l) \rightarrow s' \in OP$. Define the *extended priority relation* $\equiv$ as follows

$<s\ u_1 \cdots u_k> \ \equiv \ <s'\ v_1 \cdots v_l> \ =$

true if $f = g$ and $u_i \equiv v_i$, $i = 1,...,k$.

true if $f = g$, $k = 2$, $\mathbf{assoc} \in \Pi(f)$, and there exist a,b,c such that either

$$u_1 \equiv a,\ u_2 \equiv <s\ b\ c>,$$
$$v_1 \equiv <s\ a\ b>,\ v_2 \equiv c, \text{ or}$$

$$u_1 \equiv <s\ a\ b>,\ u_2 \equiv c,$$
$$v_1 \equiv a,\ v_2 \equiv <s\ b\ c>.$$

false otherwise.

DEFINITION 5.5.5. Given an SDF^2-definition $D^2 = (S,\Theta,OP,X,\Lambda,\Pi)$, two parse trees $p = <s\ u_1 \cdots u_k> \in PT_s(D^2)$ and $q = <s'\ v_1 \cdots v_l> \in PT_{s'}(D^2)$, and corresponding declarations $f = label(u_1) \cdots label(u_k) \rightarrow s \in OP$ and $g = label(v_1) \cdots label(v_l) \rightarrow s' \in OP$. Define the *extended priority relation* $<$ as follows

$<s\ u_1 \cdots u_k> \ < \ <s'\ v_1 \cdots v_l> \ =$

true if there exists a parse tree $r \in PT_s(D^2)$ such that $p \equiv r$ and $r < q$.

true	if $f = g$, and there exist a $j \in 1, \ldots, k$ such that $u_j < v_j$ and for all $i \neq j : u_i \equiv v_i$ or $u_i < v_i$.
true	if $f = g$, $k = 2$, **left−assoc**$\in \Pi(f)$, and there exist a,b,c such that $u_1 \equiv a$, $u_2 \equiv <s\ b\ c>$, $v_1 \equiv <s\ a\ b>$, $v_2 \equiv c$.
true	if $f = g$, $k = 2$, **right−assoc**$\in \Pi(f)$, and there exist a,b,c such that $u_1 \equiv <s\ a\ b>$, $u_2 \equiv c$, $v_1 \equiv a$, $v_2 \equiv <s\ b\ c>$.
true	if $g < f$.
false	otherwise.

DEFINITION 5.5.6. For an SDF^2-definition $D^2 = (S,\Theta,OP,X,\Lambda,\Pi)$ the *frontier relation* $\phi^2 \subseteq PT(D^2) \times (\Theta \cup \Lambda)^*$ is defined as follows. Let $p = <s\ Q_1 \cdots Q_n>$ be some parse tree in $PT_s(D^2)$ and let $f = label(Q_1) \cdots label(Q_n) \rightarrow s \in OP$ be the corresponding function declaration. Now define ϕ^2 as follows:

(1) For all $\alpha \in (\Theta \cup \Lambda)^*$ with $\phi^1\ (p,\alpha)$ for which no $p' \in PT_s(D^2)$ exists with $p < p'$ define: $\phi^2\ (p,\alpha)$.

(2) If **par**$\in \Pi(f)$ then for all $\alpha \in (\Theta \cup \Lambda)^*$ with $\phi^2\ (p,\alpha)$ define: $\phi^2\ (p,(\ \alpha\ ))$.

DEFINITION 5.5.7. The *language* $L(D^2)$ of an SDF^2-definition $D^2 = (S,\Theta,OP,X,\Lambda,\Pi)$ is the subset of $(\Theta \cup \Lambda)^*$ such that

$$\theta \in L(D^2) \Leftrightarrow \phi^2\ (p,\theta) \text{ for some } p \in PT(D^2).$$

The declaration of priority and associativity of rules in a syntax definition serves the purpose of eliminating ambiguities from its derived grammar. If ϕ^2 relates, for instance, two parse trees p_1 and p_2 with the same concrete representation, then the priority declarations may be used to eliminate this ambiguity by selecting either p_1 or p_2. Clearly this is not possible if the priority declarations are "incomplete". Continuing this same example, if p_1 and p_2 correspond to the same rule of the grammar and this rule is associative, then we consider them to be equivalent, i.e. $p_1 \equiv p_2$. This notion of disambiguation is expressed in the following definition.

DEFINITION 5.5.8. An SDF^i-definition D^i $(i = 0,1)$ is *unambiguous* iff

$$\forall \theta \in L(D^i),\ \forall p_1,p_2 \in PT(D^i) : \phi^i(p_1,\theta) \wedge \phi^i(p_2,\theta) \Rightarrow p_1 = p_2$$

An SDF^i-definition D^i $(i \geqslant 2)$ is *unambiguous* iff

$$\forall \theta \in L(D^i),\ \forall p_1,p_2 \in PT(D^i) : \phi^i(p_1,\theta) \wedge \phi^i(p_2,\theta) \Rightarrow p_1 \equiv p_2$$

Priorities are declared by stating a priority relation between two declarations in the syntax definition. In principle, the complete function definition should be given in such a declaration. In the concrete representation of syntax definitions we allow the use of *abbreviated function definitions* in order to avoid unnecessary repetition. An abbreviated function definition contains as many elements of the intended function definition as are needed to identify it uniquely. We will use the following abbreviation rules:

(1) either the *complete* keyword skeleton (i.e. all terminals occurring in the function definition) is given, or

(2) a (consecutive) substring of terminals and sorts of the function definition is given.

EXAMPLE 5.5.1. The syntax definition with priorities

```
sorts BOOL
priorities v < & < ~
functions
        true            -> BOOL
        false           -> BOOL
        BOOL v BOOL     -> BOOL {par,assoc}
        BOOL & BOOL     -> BOOL {par,assoc}
        ~ BOOL          -> BOOL {par}
```

defines, for instance, the following (unambiguous) strings

```
true v false v true
true & false v true
true & (false v true)
~ true v false.
```

Compare this with the ambiguities occurring in example 5.3.1. Note how "keyword skeletons" such as, for instance, `&` are used to identify function declarations in the syntax definition.

EXAMPLE 5.5.2. Given the SDF^2-definition

```
sorts N
priority + < *
functions
        N + N -> N
        N * N -> N
        n     -> N
```

we show how the extended priority relations resolve the ambiguity of the sentence `n+n+n*n`. This sentence has the following five parses (using parentheses to distinguish the various parses)

$p1$	`((n+n)+n)*n`
$p2$	`(n+(n+n))*n`
$p3$	`n+((n+n)*n)`
$p4$	`(n+n)+(n*n)`
$p5$	`n+(n+(n*n))`

and we have the following priority relations between the parse trees corresponding to the above parses:

$$p1, p2 < p3, p4, p5,$$
$$p3 < p5.$$

Hence, no choice can be made between $p4$ and $p5$. If, extending the above example, `+` and `*` have the property **assoc**, then $p4 \equiv p5$ holds and $p4$ and $p5$ represent the same parse. If, on the other hand, `+` and `*` have property **left−assoc** (or **right−assoc**) then $p5 < p4$ (respectively $p4 < p5$) holds and $p4$ (respectively $p5$) are the preferred parse.

EXAMPLE 5.5.3. The following syntax definition with priorities solves the "dangling else" problem by giving a higher priority to the `if-then-else`-construct. Each `else` matches the nearest enclosing `if`.

```
sorts BOOL, STAT, EXPR
priority if then < if then else
functions
        if BOOL then STAT           -> STAT
        if BOOL then STAT else STAT -> STAT
```

In this example essential use is made of the requirement that a keyword skeleton in an abbreviated declaration should be complete.

5.6. *SDF*^{3}: syntax definitions with subsorts

Now we introduce the notion of *subsorts*. These are mostly used to describe derivation chains in parse trees and to eliminate these chains from the corresponding abstract syntax trees. Subsorts define a partial order on the set of sorts. We define abstract syntax trees AT^3 using subsorts in such a way that at all argument positions in abstract syntax trees where a certain sort s is required also abstract syntax trees of any subsort of s are allowed. We also define a reduction τ^3 that reduces these syntax trees to AT^2-syntax trees by inserting injection functions where a subsort relation is used in an AT^3-term.

DEFINITION 5.6.1. A *syntax definition with subsorts* D^3 is a syntax definition with priorities $(S,\Theta,OP,X,\Lambda,\Pi)$ in which a partial order is defined on the set of sorts S. This ordering relation is denoted by $\subseteq$. If a syntax definition with subsorts contains two sorts s_1 and s_2 with $s_1 \subseteq s_2$, then it may not also contain a function declaration of the form $s_1 \rightarrow s_2$.

Note that the restriction in definition 5.6.1 is necessary to ensure the correctness of the translation function τ^3, which introduces an injection function $s_1 \rightarrow s_2$.

DEFINITION 5.6.2. The *translation* $\tau^3 : SDF^3 \rightarrow SDF^2$ is defined as follows. Let $D^3 = (S,\Theta,OP,X,\Lambda,\Pi)$. For each sort add an injection function to all its enclosing sorts, i.e. add an injection function $s_1 \rightarrow s_2$ to OP for each pair of sorts $s_1,s_2 \in S$ with $s_1 \subseteq s_2$. This gives a new set of functions OP'. Now define $\tau^3(D^3)=(S,\Theta,OP'',X,\Lambda,\Pi)$.

DEFINITION 5.6.3. Given a syntax definition with subsorts $D^3=(S,\Theta,OP,X,\Lambda,\Pi)$. The *abstract syntax trees* AT_s^3 of sort s are labeled ordered trees over the alphabet $OP \cup X$ such that

(1) $t \in AT_s^3$ if t is a variable of sort s;

(2) $f(t_1, \ldots, t_n) \in AT_s^3$ for all function declarations $\overline{f = a_0 s_1 a_1 \cdots s_n a_n \rightarrow s}$ in OP and all abstract syntax trees $t_i \in AT^0_{s'_i}$ such that either $s'_i = s_i$ or $s'_i \subseteq s_i$.

(3) There are no further abstract syntax trees of sort s in AT_s^3.

DEFINITION 5.6.4. The *parse trees* for an SDF^3-definition D^3 are defined by $PT(D^3)=PT(\tau^3(D^3))$.

DEFINITION 5.6.5. The *translation* $\tau^3 : AT^3 \rightarrow AT^2$ is defined as follows. Let $D^3 = (S,\Theta,OP,X,\Lambda,\Pi)$ and let $t \in AT_s^3$, then

(a) if $t \in X_s$, $\tau^3(t) = t$;

(b) if $t = f(t_1, \ldots, t_n) \in OP$ and $f = \overline{a_0 s_1 a_1 \cdots s_n a_n \rightarrow s,}\ \underline{\ \ }\tau^3(f(t_1, \ldots, t_n)) = f(u_1, \ldots, u_n)$ such that for each $t_i \in AT^3_{s_i'}$: $u_i = \tau^3(t_i)$ iff $s_i'=s_i$ or $u_i = \overline{s_i' s_i}(\tau^3(t_i))$ iff $s_i' \subseteq s_i$.

DEFINITION 5.6.6. Given an SDF^3-definition D^3 and two abstract syntax trees $f(u_1, \ldots, u_k) \in AT_s^3(D^3)$ and $g(v_1, \ldots, v_l) \in AT^3_{s'}(D^3)$. Define the *extended subsort relation* $\subseteq$ on abstract syntax trees as follows: $f(u_1, \ldots, u_k) < g(v_1, \ldots, v_l)=$

true	if $s \subseteq s'$.
true	if $f=g$, $k=l$, and there exists a $j \in 1, \ldots, k$ such that $u_j \subseteq v_j$ and for all $i \neq j$: $u_i=v_i$ or $u_i \subseteq v_i$.
false	otherwise.

DEFINITION 5.6.7. Given an SDF^3-definition D^3 and two parse trees $p_1 \in PTs(D^3)$ and

$p_2 \in PT_{s'}(D^3)$. Define the *extended subsort relation* $\subseteq$ on parse trees as follows:

$$p_1 \subseteq p_2 \Leftrightarrow \exists t_1 \in AT^3_s(D^3), t_2 \in AT^3_{s'}(D^3): t_1 \subseteq t_2, \tau^3(t_1)=p_1, \tau^3(t_2)=p_2.$$

DEFINITION 5.6.8. Given an SDF^3-definition $D^3=(S,\Theta,OP,X,\Lambda,\Pi)$. The *frontier relation* $\phi^3 \subseteq PT(G^3)\times(\Theta\cup\Lambda)^*$ is then defined as follows. For all $p \in PTs(D^3)$ and $\alpha\in(\Theta\cup\Lambda)^*$ with ϕ^2 (p,α) for which no $p' \in PT_{s'}(D^3)$ exists with $p' \subseteq p$ define ϕ^3 (p,α).

DEFINITION 5.6.9. The *language* $L(D^3)$ of an SDF^3-definition $D^3 = (S,\Theta,OP,X,\Lambda,\Pi)$ is the subset of $(\Theta\cup\Lambda)^*$ such that

$$\theta \in L(D^3) \Leftrightarrow \phi^3\ (p,\theta) \text{ for some } p \in PT(D^3).$$

EXAMPLE 5.6.1. Consider the following syntax definition with subsorts:

```
sorts  BOOL, NESTACK, STACK
subsorts NESTACK < STACK
functions
        true                    -> BOOL
        false                   -> BOOL
        BOOL & BOOL             -> BOOL
        empty                   -> STACK
        push BOOL on STACK      -> NESTACK
        pop NESTACK             -> BOOL
```

The reduction τ^3 will add the injection `NESTACK -> STACK` to *OP*.

EXAMPLE 5.6.2. Given the following definition:

```
sorts N, R, C
subsorts N < R < C
priorities + < *
functions
        N + N -> N        {par, left-assoc}
        R + R -> R        {par, left-assoc}
        C + C -> C        {par, left-assoc}
        N * N -> N        {par, left-assoc}
        R * R -> R        {par, left-assoc}
        C * C -> C        {par, left-assoc}
        n     -> N
        r     -> R
        c     -> C
```

The disambiguation rules always select a parse that is compatible with the priority and associativity of operators and is "most precise", i.e. it leads to an abstract syntax tree of the smallest sort. This can be seen by considering the selected parse, the sort of the corresponding abstract syntax tree and the types selected for the various operators in the following sentences:

sentence	parse	sort	op1	op2
n+n	n+n	N	N+N->N	
n+c	n+c	C	C+C->C	
r+n	r+n	R	R+R->R	
r+n+n	(r+n)+n	R	R+R->R	R+R->R
r+n*n	r+(n*n)	R	R+R->R	N*N->N
r+n+c	(r+n)+c	C	R+R->R	C+C->C
r+c*n	r+(c*n)	C	C+C->C	C*C->C
r*c+n	(r*c)+n	C	C*C->C	C+C->C

5.7. SDF^4: syntax definitions with lists

Syntactic iteration will now be introduced, i.e. a mechanism to describe the abstract and concrete representation of lists of syntactic notions such as "lists of statements separated by semicolons" or a "list of zero or more booleans", etc. This is achieved by introducing for each sort s the derived sorts $s*$ and $s+$ standing for lists of zero (one) or more elements of sort s and by defining AT^4-syntax trees using these derived sorts. We also define a reduction τ^4 that reduces AT^4-syntax trees to AT^3-syntax trees by replacing all occurrences of syntax trees of a derived sort by syntax trees consisting of (explicitly generated) binary list constructors.

DEFINITION 5.7.1. A *syntax definition with lists* D^4 is a syntax definition with subsorts $(S,\Theta,OP,X,\Lambda,\Pi)$ in which

(1) $OP \subseteq (S \cup \Theta \cup LS)^* \times S$, and

(2) $X \subseteq \Theta^* \times (S \cup LS)$,

where $LS = S \times (\Theta \cup \epsilon) \times \{+,*\}$ is the domain of *list sorts*. We will respectively write $\{s\ a\}*$, $\{s\ a\}+$, $s*$ or $s+$ for the elements $(s,a,*)$, $(s,a,+)$, $(s,\epsilon,*)$ or $(s,\epsilon,+)$ of LS.

DEFINITION 5.7.2. The *translation* $\tau^4 : SDF^4 \rightarrow SDF^3$ is defined as follows. Let $D^4 = (S,\Theta,OP,X,\Lambda,\Pi)$.

For each sort $s = (\sigma,a,*)$ or $s = (\sigma,a,+)$ in LS that occurs in some definition $f \in OP$ or $x \in X$ add the sorts $LIST0_\sigma$, $LIST1_\sigma$, $LIST0_{\sigma,a}$ and $LIST1_{\sigma,a}$ to S such that the following subsort relations hold:

$$LIST1_{\sigma,a} \subseteq LIST0_{\sigma,a},$$
$$LIST1_\sigma \subseteq LIST0_\sigma,$$
$$LIST0_\sigma \subseteq LIST0_{\sigma,a} \text{ and}$$
$$LIST1_\sigma \subseteq LIST1_{\sigma,a}.$$

Also add the new functions

$$\rightarrow LIST0_{\sigma,a}$$
$$\sigma \rightarrow LIST1_{\sigma,a}$$
$$\sigma\ a\ LIST1_{\sigma,a} \rightarrow LIST1_{\sigma,a}$$

to OP. This gives a new set of function definitions OP'. Now we define $\tau^4(D^4) = (S,\Theta,OP',X,\Lambda,\Pi)$.

DEFINITION 5.7.3. The *abstract syntax trees* AT_s^4 of sort $s \in S \cup LS$ are labeled ordered trees over the alphabet $OP \cup X$ such that:

(1) $t \in AT_s^4$ if t is a variable of sort s;

(2) $f(t_1, \ldots, t_n) \in AT_s^4$ for all function definitions $f = a_0 s_1 a_1 \cdots s_n a_n \rightarrow s$ in OP and all abstract syntax trees $t_i \in AT_{s_i'}^4$, $i = 1, \ldots, n$ such that either $s_i' = s_i$ or $s_i' \subseteq s_i$;

(3) $list_{\sigma,a}(t_1, \ldots, t_n) \in AT_s^4$ for all $\sigma \in S$, $a \in \Theta$, $t_i \in AT_\sigma^4$ and $n \geq 0$ (if $s = \{\sigma\ a\}*$ or $s = \sigma*$) or $n \geq 1$ (if $s = \{\sigma\ a\}+$ or $s = \sigma+$).

(4) There are no further abstract syntax trees of sort s in AT_s^4.

DEFINITION 5.7.4. The *translation* $\tau^4 : AT^4 \rightarrow AT^3$ is defined as follows.

(a) $\tau^4(x) = x$ for all $x \in X$;

(b) $\tau^4(f(t_1, \ldots, t_n)) = f(u_1, \ldots, u_n)$ for all $f = a_0 s_1 a_1 \cdots s_n a_n \rightarrow s$ in OP, $t_i \in T_{s_i}^3$, and $u_i = \tau^4(t_i)$;

(c) $\tau^4(list_{\sigma,a}()) = l_0$, where $l_0 = \overline{\rightarrow LIST0_{\sigma,a}}$;

(d) $\tau^4(list_{\sigma,a}(t_1)) = l_1(\tau^4(t_1))$, where $l_1 = \overline{\sigma \rightarrow LIST1_{\sigma,a}}$;

(e) $\tau^4(list_{\sigma,a}(t_1, \underline{\cdots, t_n)) = l}_2(\tau^4(t_1), l_2(\underline{\tau^4(t_2), l_2(\cdots, l_2(\tau^4(t_{n-1})}, {}_1(\tau^4(t_n))) \cdots))),$
where $l_1 = \sigma \rightarrow LIST1_{\sigma,a}$ and $l_2 = \sigma\ a\ LIST1_{\sigma,a} \rightarrow LIST1_{\sigma,a}$.

DEFINITION 5.7.5. Let D^4 be an SDF^4-definition $D^4 = (S, \Theta, OP, X, \Lambda, \Pi)$. The *frontier relation* $\phi^4 \subseteq PT(G^4) \times (\Theta \cup \Lambda)^*$ is then defined by $\phi^4 = \phi^3$.

DEFINITION 5.7.6. The *language* $L(D^4)$ of an SDF^4-definition $D^4 = (S, \Theta, OP, X, \Lambda, \Pi)$ is the subset of $(\Theta \cup \Lambda)^*$ such that

$$\theta \in L(D^4) \Leftrightarrow \phi^4\ (p, \theta) \text{ for some } p \in PT(D^4).$$

EXAMPLE 5.7.1. Given the following syntax definition with lists:

```
sorts EXP, STAT, SERIES
        if EXP then SERIES else SERIES endif -> STAT
        while EXP do SERIES endwhile         -> STAT
        { STAT ; }*                          -> SERIES
```

This syntax definition will be reduced by τ^4 to

```
sorts  EXP, STAT, SERIES,
       LIST0-STAT, LIST1-STAT,
       LIST0-STAT-SEMI, LIST1-STAT-SEMI
subsorts
       LIST1-STAT-SEMI < LIST0-STAT-SEMI,
       LIST1-STAT  < LIST0-STAT,
       LIST0-STAT < LIST0-STAT-SEMI,
       LIST1-STAT < LIST1-STAT-SEMI
functions
       if EXP then SERIES else SERIES endif -> STAT
       while EXP do SERIES endwhile         -> STAT
       LIST0-STAT-SEMI                      -> SERIES
                                            -> LIST0-STAT-SEMI
       STAT                                 -> LIST1-STAT-SEMI
       STAT ; LIST1-STAT-SEMI               -> LIST1-STAT-SEMI
```

EXAMPLE 5.7.2. Here we define lexical, concrete and abstract syntax of the simple programming language PICO as defined in [Annexe D4.A3].

```
sorts    PICO-PROGRAM, DECLS, EXP, STAT, SERIES,
         ID, ID-TYPE, PICO-TYPE,
         NAT-CON, STR-CON,
         CHAR, DIGIT, LETTER, LETTER-OR-DIGIT
subsorts
         (DIGIT, LETTER) < LETTER-OR-DIGIT < CHAR,
         ID < EXPR, NAT-CON < (EXP, NAT)
priorities
         + < ||
functions
       begin DECLS SERIES end              -> PICO-PROGRAM
       declare {ID-TYPE , }* ;             -> DECLS
       ID : PICO-TYPE                      -> ID-TYPE
       { STAT ; }*                         -> SERIES
       ID := EXP                           -> STAT
       if EXP then SERIES else SERIES fi   -> STAT
       while EXP do SERIES od              -> STAT
       EXP + EXP                           -> EXP
       EXP || EXP                          -> EXP
       DIGIT+                              -> NAT-CON {lex}

       LETTER LETTER-OR-DIGIT*             -> ID      {lex}
```

5.8. The concrete representation of *SDF*

The complete grammar for *SDF* is:

```
SDF-definition    ::= sorts subsorts priorities layout functions

sorts             ::= "sorts" { sort "," }+
sort              ::= id

subsorts          ::= "subsorts" { subsort-def  "," }+ | empty
subsort-def       ::= { sort-names ">" }+ | { sort-names "<" }+
sort-names        ::= id | "(" { id "," }+ ")" .

priorities        ::= "priorities" { prio-def "," }+  | empty
prio-def          ::= { rules ">" }+ | { rules "<" }+
rules             ::= rule | "(" { rule "," }+ ")"

layout            ::= "layout" { literal "," }+ | empty

functions         ::= "functions" function-def+  | empty
function-def      ::= rule "->" sort attributes.
rule              ::= rule-elem*
rule-elem         ::= literal | sort | sort replicator |
                        "{" sort literal "}" replicator
replicator        ::= "+" | "*"
attributes        ::= "{" { attribute "," }+ "}" | empty
attribute         ::= "lex" | "par" | "assoc"
empty             ::= ""
```

The notions `id` and `literal` are defined as follows:

(1) an `id` consists of a letter followed by zero or more letters, digits and hyphens.

(2) a `literal` consists of zero or more ASCII characters surrounded by double quote characters ". In a literal, the double quote character must be preceeded by a backslash \. In addition to this, the following escape sequences may be used: \n (newline), \r (carriage return), \b (backspace), \t (horizontal tab) and \\ (backslash). The double quotes that surround the characters in the literal may be omitted if this does not introduce ambiguities in the parsing of the *SDF*-definition.

5.9. Construction of a syntax definition for given a grammar

We intend to use *SDF* for the definition of existing and of newly designed formalisms and languages. When defining an existing language, such as e.g. Pascal, one is faced with the problem of constructing an *SDF*-definition from a given context-free grammar. Clearly, many equivalent *SDF*-definitions can be constructed for a given grammar. This poses the problem of constructing an "optimal" *SDF*-definition for a given grammar, but it is not obvious how such a notion of optimality should be formulated. In this section, we discuss some of the transformations one could apply, during this construction.

First we show a trivial, but unsatisfactory, construction from context-free grammar to *SDF*-definition.

ALGORITHM 5.9.1.
Given a context-free grammar $G=(N,\Theta,PROD,R)$, we construct a *basic syntax definition* $D=(S,\Theta,OP,\varnothing)$ as follows:

(1) [*Define sorts*]: For each non-terminal $Y \in N$ add a sort Y to S.

(2) [*Define functions*]: For each rule $Y::=a_0 Y_1 a_1 \cdots Y_n a_n$ in *PROD*, add a function definition $a_0 Y_1 a_1 \cdots Y_n a_n \rightarrow Y$ to *OP*.

We will now describe a series of possible transformation steps of a given grammar. The following grammar will be used as a starting point:

```
EXP    ::= EXP - TERM | TERM
TERM   ::= TERM / FACTOR | FACTOR
FACTOR ::= ( EXP ) | ID | ID ( EXPS )
ID     ::= a | b
EXPS   ::= EXP | EXP , EXPS
```

Step 1. Apply algorithm 5.9.1. This leads to the construction of the following basic syntax definition:

```
sorts EXP, EXPS, TERM, FACTOR, ID
functions
        EXP - TERM      -> EXP
        TERM            -> EXP
        TERM / FACTOR   -> TERM
        FACTOR          -> TERM
        ( EXP )         -> FACTOR
        ID              -> FACTOR
        ID ( EXPS )     -> FACTOR
        a               -> ID
        b               -> ID
        EXP             -> EXPS
        EXP , EXPS      -> EXPS
```

Step 2. Replace sorts and functions whose sole purpose is to define priorities between function definitions by equivalent priority definitions. This gives:

```
sorts EXP, EXPS, ID
priorities - < /
functions
        EXP - EXP       -> EXP {left-assoc}
        EXP / EXP       -> EXP {left-assoc}
        ( EXP )         -> EXP
        ID              -> EXP
        ID ( EXPS )     -> EXP
        a               -> ID
        b               -> ID
        EXP             -> EXPS
        EXP , EXPS      -> EXPS
```

Step 3. Eliminate all functions of the form `( S )` for some sort `S` by equivalent `par` attributes. This gives:

```
sorts EXP, EXPS, ID
priorities - < /
functions
        EXP - EXP       -> EXP {left-assoc,par}
        EXP / EXP       -> EXP {left-assoc,par}
        ID              -> EXP {par}
        ID ( EXPS )     -> EXP {par}
        a               -> ID
        b               -> ID
        EXP             -> EXPS
        EXP , EXPS      -> EXPS
```

Step 4. Eliminate all sorts and functions which are used for defining lists and replace them by equivalent list sorts. This gives:

```
sorts EXP, ID
priorities - < /
functions
        EXP - EXP                -> EXP {left-assoc,par}
        EXP / EXP                -> EXP {left-assoc,par}
        ID                       -> EXP {par}
        ID ( { EXP   , }+ )      -> EXP {par}
        a                        -> ID
        b                        -> ID
```

Step 5. Eliminate all injections of the form `S -> T` (which correspond to chain rules in the original grammar) by equivalent subsort declarations. Note that chain rules can, in many cases, also be eliminated by repeated substitutions in the given grammar. Introduction of subsorts in our example syntax definition gives:

```
sorts EXP, ID
subsorts ID < EXP
priorities - < /
functions
        EXP - EXP                -> EXP {left-assoc,par}
        EXP / EXP                -> EXP {left-assoc,par}
        ID ( { EXP   , }+ )      -> EXP {par}
        a                        -> ID
        b                        -> ID
```

Two comments can be made on the above construction:

(1) The constructed *SDF*-definition is "too large" in the sense that the original grammar had one startsymbol (in the example: `EXP`), while the constructed *SDF*-definition defines the sorts `EXP` and `ID`. One essentially needs a mechanism for name hiding (see section 6.) to obtain a syntax definition which is completely equivalent to the original grammar.

(2) Ambiguities in the original context-free grammar are carried over to the constructed *SDF*-definition. In the case of the given Pascal-grammar (see appendix) ambiguities exist which can only be eliminated by type checking.

6. RELATION WITH LOGIC SPECIFICATIONS

6.1. Combination of *SDF* with other formalisms

In the previous sections we have only been concerned with mappings between concrete representations and abstract syntax trees and have disregarded the semantics of these syntax trees. Now we will address the question of assigning meanings to syntax trees, or in other words, of combining *SDF* with an arbitrary first-order formalism *F*. We will do so only in a global and informal way.

Two steps are necessary to combine *SDF* with a formalism *F*:

(1) The concrete representation of the well-formed formulae of *F* has to be defined. This amounts to combining the concrete representations of terms (as defined by the syntax definition) with the connectives that may further occur in the formulae of *F*. Choosing equational logic for *F*, one may, for instance, write

$$x * succ(y) = x + x * y$$

instead of

$$mul(x,\ succ(y)) = add(x,\ mul(x,\ y)).$$

Or choosing first-order logic for *F*, one may write

$$\forall x\colon x>0 \rightarrow succ(x)>0$$

instead of

$$\forall x\colon greater(x,\ 0) \rightarrow greater(succ(x),\ 0).$$

(2) The modularization constructs of *F* (if any) have to be extended to cover combination and renaming of function definitions in syntax definitions. On *import* of one module in another one, the underlying grammars of both modules have to be joined. On *export* of sorts and functions from a module, the grammar of the "exported language" of that module is defined by all exported sorts and by all definitions of exported functions with an exported sort as result domain. When a *parameterized module* is bound to an actual parameter module, the underlying grammars of both modules have to be renamed and combined.

6.2. Additional requirements due to composition of definitions

SDF has been designed from the point of view that only *one* language is being defined. When combining *SDF* with modularization constructs several additional questions and constraints arize:

(1) When combining modules one can not use a reserved word strategy for identifiers and keywords, since identifiers in one module may conflict with keywords in another module. This motivates the choice of some of our lexical conventions in section 5.4.

(2) How should the layout alphabets of two modules be combined? Choosing the union of these alphabets may introduce undesirable ambiguities, but choosing one general layout alphabet may be too restrictive.

(3) How should the comment conventions of two modules be combined? Typically, language L_1 serves as comment in the definition of language L_2. Using parameterized modules, a language definition could be parameterized with, for instance, a comment language.

(4) When two modules are combined the resulting language may be ambiguous. This can be seen as incompleteness of the combined priority definitions of the two modules. This suggests that the construct for module composition should allow for the definition of additional priorities.

(5) Modular language definitions may lead to completely different decompositions of a language definition. For instance, instead of specifying one, monolithic, syntax, separate modules can specify different syntactical categories such as expressions, statements, declarations, etc.

(6) Current parser generator technology uses extensive preprocessing of a given grammar to

produce an efficient parser. The implementation of modular language definitions requires techniques that preserve the compositionality of preprocessed modules at the implementation level.

6.3. Interaction between syntax and semantics

Syntax definitions, as developed in this paper, do not depend on semantics. They only define a concrete representation of abstract syntax trees over some signature. However, once syntax definitions have been combined with a specific formalism, additional constraints can be imposed on abstract syntax trees. In this manner, constraints can be expressed on abstract syntax trees that are otherwise not expressible in *SDF*. In this way, the expressive power of *SDF* (which is equivalent to simple syntax-directed translations, see section 5.3.3) can be increased.

6.4. Limitations and further work

Our syntax definition formalism allows the definition of arbitrary, context-free grammars. It has, however, some restrictions:

(1) The **par** attribute, as defined, only allows a fixed pair of parentheses. It should be generalized to allow arbitrary pairs of parentheses.

(2) In many languages, comment is defined at the lexical level and may even occur inside lexical tokens. We are only capable of expressing comments as syntactic entities.

(3) *SDF* does not address the *pretty printing* of concrete representations.

These problems have to be solved, but we also envisage other subjects for further research:

(1) Formal definition of the combination of syntax definitions with algebraic specifications.

(2) Study the combination with other formalisms such as, for instance, TYPOL [Annexe D4.A2].

(3) Study techniques for implementing syntax definitions. A full implementation will require the use of parsing techniques for arbitrary, context-free grammars. It may also be worthwhile to investigate the possibility of applying restricted, but more efficient parsing techniques.

REFERENCES

[ASU85] A.V. Aho, R. Sethi & J.D. Ullman, *Compilers. Principles, Techniques and Tools*, Addison-Wesley, 1985.

[AU72] A.V. Aho & J.D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Volumes I and II, Prentice-Hall, 1972.

[BMS80] R.M. Burstall, D. MacQueen & D. Sanella, "HOPE: an experimental applicative language", *Conf. Record of the 1980 LISP Conference*, Stanford University, 1980, pp. 136-143.

[EM85] H. Ehrig & B. Mahr, *Fundamentals of Algebraic Specifications 1*, EATCS Monographs in Theoretical Computer Science, Springer-Verlag, 1985.

[FGJM85] K. Futatsugi, J.A. Goguen, J.P. Jouannaud & J. Meseguer, "Principles of OBJ2", *Conf. Record 12th Ann. ACM Symp. Principles of Programming Languages*, ACM, 1985, pp. 52-66.

[GM85] J.A. Goguen & J. Meseguer, "Order-Sorted Algebra I: Partial and Overloaded Operators, Errors and Inheritance", (to appear).

[IR61] E.T. Irons, " A syntax directed compiler for Algol 60" Communications of the ACM **4** (1961) 1, pp. 51-55.

[IR70] E.T. Irons, "Experience with an Extensible Language", Communications of the ACM **13** (1970) 1, pp. 31-40.

[JOH79] S.C. Johnson, "YACC: yet another compiler-compiler", in: *UNIX Programmer's Manual*, Vol. 2B, Bell Laboratories, 1979.

[JW] K. Jensen, N. Wirth, (revised by A.B. Mickel & J.F. Miner), *PASCAL User* *manual and Report*, Third Edition, Springer-Verlag.

[KLMM83] G. Kahn, B. Lang, B. Mélèse & E. Morcos, "METAL: a formalism to specify formalisms", *Science of Computer Programming*, **3**(1983), pp. 151-188.

[KLA83] Klaeren, H.A., *Algebraische Spezifikation: Eine Einführung*, Springer Verlag, 1983.

[LS79] M.E. Lesk & E. Schmidt, "LEX - A lexical analyzer generator", in: *UNIX Programmer's Manual*, Vol. 2B, Bell Laboratories, 1979.

[ODO85] M.J. O'Donnell, *Equational Logic as a Programming Language*, MIT Press, 1985.

[SAN82] D. Sandberg, "LITHE: A language combining a flexible syntax and classes", *Conf. Record 9th Ann. ACM Symp. Principles of Programming Languages*, ACM, 1982, pp. 142-145.

[ST75] T.A. Standish, "Extensibility in Programming Language Design" SIGPLAN Notices **10** (1975) 7, pp. 18-21.

[WE70] B. Wegbreit, *Studies in Extensible Programming Languages*, dissertation, Harvard, 1970 (reprinted by Garland Publishing, 1980).

Appendix: Syntax definition of Pascal

In this appendix we give a definition of the lexical and concrete syntax of Pascal as described in [JW]. This definition also establishes an abstract syntax for that language.

```
sorts

   ACTUAL-PARAMETER, ACTUAL-PARAMETER-LIST, ANY-CHAR-BUT-APOSTROPHE, ARRAY-TYPE,
   BLOCK, BUFFER-VARIABLE, CASE, CHARACTER-STRING, COMPONENT-VARIABLE,
   COMPOUND-STATEMENT, CONDITIONAL-STATEMENT, CONFORMANT-ARRAY-SCHEMA, CONSTANT,
   CONSTANT-DEFINITION, CONSTANT-DEFINITION-PART, DIGIT, ELEMENT-DESCRIPTION,
   ENUMERATED-TYPE, EXPRESSION, FIELD-DESIGNATOR, FIELD-LIST,
   FILE-TYPE, FORMAL-PARAMETER-LIST, FORMAL-PARAMETER-SECTION,
   FUNCTION-DECLARATION, FUNCTION-DESIGNATOR, FUNCTION-HEADING,
   FUNCTION-IDENTIFICATION, IDENTIFIED-VARIABLE, IDENTIFIER, IDENTIFIER-LIST,
   INDEX-TYPE-SPECIFICATION, INDEXED-VARIABLE, LABEL, LABEL-DECLARATION-PART,
   LETTER, LETTER-OR-DIGIT, NIL, ORDINAL-TYPE, POINTER-TYPE,
   PROCEDURE-AND-FUNCTION-DECLARATION-PART, PROCEDURE-DECLARATION,
   PROCEDURE-HEADING, PROCEDURE-IDENTIFICATION, PROCEDURE-OR-FUNCTION-DECLARATION,
   PROGRAM, PROGRAM-HEADING, RECORD-SECTION, RECORD-TYPE, REPETITIVE-STATEMENT,
   SCALEFACTOR, SET-CONSTRUCTOR, SET-TYPE, SIMPLE-EXPRESSION, SIMPLE-STATEMENT,
   SIMPLE-TYPE, STATEMENT, STRING-ELEMENT, STRUCTURED-STATEMENT, STRUCTURED-TYPE,
   SUBRANGE-TYPE, TYPE, TYPE-DEFINITION, TYPE-DEFINITION-PART,
   UNPACKED-STRUCTURED-TYPE, UNSIGNED-CONSTANT, UNSIGNED-INTEGER, UNSIGNED-NUMBER,
   UNSIGNED-REAL, VALUE-PARAMETER-SPECIFICATION, VARIABLE, VARIABLE-DECLARATION,
   VARIABLE-DECLARATION-PART, VARIABLE-PARAMETER-SPECIFICATION, VARIANT,
   VARIANT-PART, VARIANT-SELECTOR, WITH-STATEMENT, WRITE-PARAMETER,
   WRITE-PARAMETER-LIST

subsorts
  LETTER-OR-DIGIT >
        (LETTER, DIGIT)
  UNSIGNED-NUMBER >
        (UNSIGNED-INTEGER, UNSIGNED-REAL)
  SCALEFACTOR >
        UNSIGNED-INTEGER
  STRING-ELEMENT >
        ANY-CHAR-BUT-APOSTROPHE
  UNSIGNED-CONSTANT >
        (UNSIGNED-NUMBER, CHARACTER-STRING, IDENTIFIER, NIL)
  VARIABLE >
        (IDENTIFIER, COMPONENT-VARIABLE, IDENTIFIED-VARIABLE, BUFFER-VARIABLE)
  COMPONENT-VARIABLE >
        (INDEXED-VARIABLE, FIELD-DESIGNATOR)
  EXPRESSION >
        (SIMPLE-EXPRESSION, UNSIGNED-CONSTANT, IDENTIFIER, VARIABLE,
        SET-CONSTRUCTOR, FUNCTION-DESIGNATOR)
  FIELD-DESIGNATOR >
        IDENTIFIER
  ELEMENT-DESCRIPTION >
        EXPRESSION
  ACTUAL-PARAMETER >
        (EXPRESSION, VARIABLE)
  WRITE-PARAMETER >
        EXPRESSION
  FORMAL-PARAMETER-SECTION >
        (VALUE-PARAMETER-SPECIFICATION, VARIABLE-PARAMETER-SPECIFICATION,
        PROCEDURE-HEADING, FUNCTION-HEADING)
  FUNCTION-DESIGNATOR >
        IDENTIFIER
  PROCEDURE-OR-FUNCTION-DECLARATION >
        (PROCEDURE-DECLARATION, FUNCTION-DECLARATION)
  ORDINAL-TYPE >
        (ENUMERATED-TYPE, SUBRANGE-TYPE, IDENTIFIER)
  TYPE >
        (SIMPLE-TYPE, STRUCTURED-TYPE, POINTER-TYPE)
  SIMPLE-TYPE >
        (ORDINAL-TYPE, IDENTIFIER)
  STRUCTURED-TYPE >
        (UNPACKED-STRUCTURED-TYPE, IDENTIFIER)
  UNPACKED-STRUCTURED-TYPE >
        (ARRAY-TYPE, RECORD-TYPE, SET-TYPE, FILE-TYPE)
  FIELD-LIST >
        VARIANT-PART
  VARIANT-SELECTOR >
        IDENTIFIER
  POINTER-TYPE >
        IDENTIFIER
  SIMPLE-STATEMENT >
        IDENTIFIER
  STRUCTURED-STATEMENT >
        (COMPOUND-STATEMENT, CONDITIONAL-STATEMENT, REPETITIVE-STATEMENT,
         WITH-STATEMENT)
  STATEMENT >
        (SIMPLE-STATEMENT, STRUCTURED-STATEMENT)

priorities

   (- EXPRESSION, + EXPRESSION, not EXPRESSION) >
   (*, /, div, mod, and) >
   (EXPRESSION + EXPRESSION, EXPRESSION - EXPRESSION, or) >
   (=, <>, <, <=, >, >=, in)

functions

   a                                             -> LETTER
   ...
   z                                             -> LETTER
   0                                             -> DIGIT
   ...
   9                                             -> DIGIT
   LETTER LETTER-OR-DIGIT*                       -> IDENTIFIER         {lex}
   DIGIT+                                        -> UNSIGNED-INTEGER   {lex}
   UNSIGNED-INTEGER . DIGIT+                     -> UNSIGNED-REAL      {lex}
   UNSIGNED-INTEGER . DIGIT+ e SCALEFACTOR       -> UNSIGNED-REAL      {lex}
   UNSIGNED-INTEGER e SCALEFACTOR                -> UNSIGNED-REAL      {lex}
   + UNSIGNED-INTEGER                            -> SCALEFACTOR        {lex}
   - UNSIGNED-INTEGER                            -> SCALEFACTOR        {lex}
   ' STRING-ELEMENT* '                           -> CHARACTER-STRING   {lex}
   ''                                            -> STRING-ELEMENT     {lex}
```

```
nil                                          -> NIL
+ UNSIGNED-NUMBER                            -> CONSTANT
- IDENTIFIER                                 -> CONSTANT
IDENTIFIER = CONSTANT                        -> CONSTANT-DEFINITION
const { CONSTANT-DEFINITION ; }+ ;           -> CONSTANT-DEFINITION-PART
                                             -> CONSTANT-DEFINITION-PART
VARIABLE ^                                   -> IDENTIFIED-VARIABLE
VARIABLE ^                                   -> BUFFER-VARIABLE
{ IDENTIFIER , }+ : TYPE                     -> VARIABLE-DECLARATION
var { VARIABLE-DECLARATION ; }+ ;            -> VARIABLE-DECLARATION-PART
                                             -> VARIABLE-DECLARATION-PART
VARIABLE [ { EXPRESSION , }+ ]               -> INDEXED-VARIABLE
VARIABLE . IDENTIFIER                        -> FIELD-DESIGNATOR
[ { ELEMENT-DESCRIPTION , }* ]               -> SET-CONSTRUCTOR
EXPRESSION .. EXPRESSION                     -> ELEMENT-DESCRIPTION

SIMPLE-EXPRESSION * SIMPLE-EXPRESSION        -> SIMPLE-EXPRESSION {par,left-assoc}
SIMPLE-EXPRESSION / SIMPLE-EXPRESSION        -> SIMPLE-EXPRESSION {par,left-assoc}
SIMPLE-EXPRESSION div SIMPLE-EXPRESSION      -> SIMPLE-EXPRESSION {par,left-assoc}
SIMPLE-EXPRESSION mod SIMPLE-EXPRESSION      -> SIMPLE-EXPRESSION {par,left-assoc}
SIMPLE-EXPRESSION +  SIMPLE-EXPRESSION       -> SIMPLE-EXPRESSION {par,left-assoc}
SIMPLE-EXPRESSION -  SIMPLE-EXPRESSION       -> SIMPLE-EXPRESSION {par,left-assoc}
- SIMPLE-EXPRESSION                          -> SIMPLE-EXPRESSION {par}
+ SIMPLE-EXPRESSION                          -> SIMPLE-EXPRESSION {par}
not SIMPLE-EXPRESSION                        -> SIMPLE-EXPRESSION {par}

SIMPLE-EXPRESSION and SIMPLE-EXPRESSION      -> EXPRESSION
SIMPLE-EXPRESSION or SIMPLE-EXPRESSION       -> EXPRESSION
SIMPLE-EXPRESSION =  SIMPLE-EXPRESSION       -> EXPRESSION
SIMPLE-EXPRESSION <> SIMPLE-EXPRESSION       -> EXPRESSION
SIMPLE-EXPRESSION <  SIMPLE-EXPRESSION       -> EXPRESSION
SIMPLE-EXPRESSION <= SIMPLE-EXPRESSION       -> EXPRESSION
SIMPLE-EXPRESSION >  SIMPLE-EXPRESSION       -> EXPRESSION
SIMPLE-EXPRESSION >= SIMPLE-EXPRESSION       -> EXPRESSION
SIMPLE-EXPRESSION in SIMPLE-EXPRESSION       -> EXPRESSION

( EXPRESSION )                               -> SIMPLE-EXPRESSION

( { ACTUAL-PARAMETER , }* )                  -> ACTUAL-PARAMETER-LIST

EXPRESSION : EXPRESSION                      -> WRITE-PARAMETER
EXPRESSION : EXPRESSION : EXPRESSION         -> WRITE-PARAMETER
( VARIABLE )                                 -> WRITE-PARAMETER-LIST
( VARIABLE , { WRITE-PARAMETER , }+ )        -> WRITE-PARAMETER-LIST
( { WRITE-PARAMETER , }+ )                   -> WRITE-PARAMETER-LIST

IDENTIFIER .. IDENTIFIER : IDENTIFIER        -> INDEX-TYPE-SPECIFICATION
packed array [ INDEX-TYPE-SPECIFICATION ] of IDENTIFIER
                                             -> CONFORMANT-ARRAY-SCHEMA
array [ { INDEX-TYPE-SPECIFICATION ; }+ ] of IDENTIFIER
                                             -> CONFORMANT-ARRAY-SCHEMA
array [ { INDEX-TYPE-SPECIFICATION ; }+ ] of CONFORMANT-ARRAY-SCHEMA
                                             -> CONFORMANT-ARRAY-SCHEMA
( { FORMAL-PARAMETER-SECTION ; }+ )          -> FORMAL-PARAMETER-LIST
IDENTIFIER-LIST : IDENTIFIER                 -> VALUE-PARAMETER-SPECIFICATION
```

```
IDENTIFIER-LIST : CONFORMANT-ARRAY-SCHEMA    -> VALUE-PARAMETER-SPECIFICATION
var IDENTIFIER-LIST : IDENTIFIER             -> VARIABLE-PARAMETER-SPECIFICATION
var IDENTIFIER-LIST : CONFORMANT-ARRAY-SCHEMA
                                             -> VARIABLE-PARAMETER-SPECIFICATION

function IDENTIFIER : IDENTIFIER             -> FUNCTION-HEADING
function IDENTIFIER FORMAL-PARAMETER-LIST : IDENTIFIER
                                             -> FUNCTION-HEADING
FUNCTION-HEADING ; BLOCK                     -> FUNCTION-DECLARATION
FUNCTION-HEADING ; IDENTIFIER                -> FUNCTION-DECLARATION
FUNCTION-IDENTIFICATION ; BLOCK              -> FUNCTION-DECLARATION
function IDENTIFIER                          -> FUNCTION-IDENTIFICATION
IDENTIFIER ACTUAL-PARAMETER-LIST             -> FUNCTION-DESIGNATOR

procedure IDENTIFIER FORMAL-PARAMETER-LIST   -> PROCEDURE-HEADING
procedure IDENTIFIER                         -> PROCEDURE-HEADING
procedure IDENTIFIER                         -> PROCEDURE-IDENTIFICATION
PROCEDURE-HEADING ; BLOCK                    -> PROCEDURE-DECLARATION
PROCEDURE-HEADING ; IDENTIFIER               -> PROCEDURE-DECLARATION
PROCEDURE-IDENTIFICATION ; BLOCK             -> PROCEDURE-DECLARATION

{PROCEDURE-OR-FUNCTION-DECLARATION ; }+ ;    -> PROCEDURE-AND-FUNCTION-DECLARATION-PART
                                             -> PROCEDURE-AND-FUNCTION-DECLARATION-PART

PROGRAM-HEADING ; BLOCK                      -> PROGRAM
program IDENTIFIER                           -> PROGRAM-HEADING
program IDENTIFIER ( { IDENTIFIER , }+ )     -> PROGRAM-HEADING

( IDENTIFIER-LIST )                          -> ENUMERATED-TYPE
CONSTANT .. CONSTANT                         -> SUBRANGE-TYPE
packed UNPACKED-STRUCTURED-TYPE              -> STRUCTURED-TYPE
array [ { ORDINAL-TYPE , }+ ] of TYPE        -> ARRAY-TYPE
record FIELD-LIST end                        -> RECORD-TYPE
IDENTIFIER-LIST : TYPE                       -> RECORD-SECTION

FieldList:                                   -> FIELD-LIST
{ RECORD-SECTION ;}+                         -> FIELD-LIST
{ RECORD-SECTION ;}+ ;                       -> FIELD-LIST
{ RECORD-SECTION ;}+ ; VARIANT-PART          -> FIELD-LIST

case VARIANT-SELECTOR of { VARIANT ; }+      -> VARIANT-PART
IDENTIFIER : IDENTIFIER                      -> VARIANT-SELECTOR
{ CONSTANT ,}+ : ( FIELD-LIST )              -> VARIANT

set of ORDINAL-TYPE                          -> SET-TYPE
file of TYPE                                 -> FILE-TYPE
^IDENTIFIER                                  -> POINTER-TYPE

IDENTIFIER = TYPE                            -> TYPE-DEFINITION
type { TYPE-DEFINITION ; }+ ;                -> TYPE-DEFINITION-PART
                                             -> TYPE-DEFINITION-PART


                                             -> SIMPLE-STATEMENT
VARIABLE := EXPRESSION                       -> SIMPLE-STATEMENT
IDENTIFIER := EXPRESSION                     -> SIMPLE-STATEMENT
IDENTIFIER ACTUAL-PARAMETER-LIST               -> SIMPLE-STATEMENT
IDENTIFIER WRITE-PARAMETER-LIST                -> SIMPLE-STATEMENT
goto LABEL                                     -> SIMPLE-STATEMENT

begin { STATEMENT ; }+ end                     -> COMPOUND-STATEMENT

if EXPRESSION then STATEMENT                   -> CONDITIONAL-STATEMENT
if EXPRESSION then STATEMENT else STATEMENT    -> CONDITIONAL-STATEMENT
{ CONSTANT , }+ : STATEMENT                    -> CASE
case EXPRESSION of { CASE ; }+ end             -> CONDITIONAL-STATEMENT
case EXPRESSION of { CASE ; }+ end ;           -> CONDITIONAL-STATEMENT

while EXPRESSION do STATEMENT                  -> REPETITIVE-STATEMENT
repeat { STATEMENT ; }+ until EXPRESSION       -> REPETITIVE-STATEMENT
for VARIABLE := EXPRESSION to EXPRESSION
     do STATEMENT                              -> REPETITIVE-STATEMENT
for VARIABLE := EXPRESSION downto EXPRESSION
     do STATEMENT                              -> REPETITIVE-STATEMENT

with { VARIABLE , }+ do STATEMENT              -> WITH-STATEMENT

LABEL : STATEMENT                              -> STATEMENT
{ DIGIT }+                                     -> LABEL                   {lex}
label { LABEL , }+ ;                           -> LABEL-DECLARATION-PART
                                               -> LABEL-DECLARATION-PART

LABEL-DECLARATION-PART CONSTANT-DEFINITION-PART
TYPE-DEFINITION-PART VARIABLE-DECLARATION-PART
PROCEDURE-AND-FUNCTION-DECLARATION-PART COMPOUND-STATEMENT
                                               -> BLOCK
```

GIPE : CEC 348/S/T4/12
version 2 20-12-85

Specifications in Natural Semantics

Deliverable D4 of task T4, Annex 2

D. Clement (SEMA)
J. Despeyroux (INRIA)
T. Despeyroux (INRIA)
L. Hascoet (INRIA)
G. Kahn (INRIA)

This Annex describes several experiments in describing programming languages carried out in the style of Natural Semantics. The examples are small but contain the typical difficulties in programming language semantics.

1. Introduction and overview

Inspired by early work of Plotkin, we have developed a formalism called Natural Semantics for describing the various aspects of programming language semantics[NS,JD]. This work has involved designing a prototype language, TYPOL and its compiler to Prolog. Specifications written in TYPOL can be executed, so that a higher degree of confidence in their validity can be reached. In this paper, we assume general familiarity with Natural Semantics and TYPOL.

Sample specifications have been developed in three areas: static semantics, dynamic semantics and translation.

In the area of **static semantics** we discuss:

- An Algol-68 like language, ASPLE. The significant point here is that type expressions are constrained in a non trivial way (coercions).
- A skeleton ML with polymorphic types and type inference. The main interest of this work is to show that unification in the *meta-system* can advantageously replace explicit calls to unification in the traditional polymorphic type-checking algorithm. This is a general fact rather than a fortuitous one.
- The AMBER language, designed by Cardelli. This language provides an opportunity to discuss multiple inheritance in the context of a static type-checking system.
- The ESTEREL language designed by Berry. It is used as an example of a non-toy language to work with. This is also developed as a future context in which we could test ideas on incremental type-checking.

In the area of **translation**, we elaborate two examples:

- Translating ASPLE to a simple stack machine language. Translation seems very easy in this context. The method allows generating code that contains free variables, a strategy to define relocatable code. A simple form of optimisation is obtained as well.

- Translating a reduced version of ML to a clever machine code, called CAM, especially designed for applicative languages. This translation serves as a base for a correctness proof in a forthcoming paper [JD].

Another form of translation occurs, usually intimately connected to type-checking: translating from *surface* abstract syntax to *deep* abstract syntax. This is done with ASPLE for example, to solve overloading of operators. These translations are very simple, so that we do not need to discuss them at length.
Finally, we also consider pretty-printing as a particular kind of translation. For example, we have designed a pretty-printer of TYPOL that generates TeX input following this idea. General pretty-printing is not discussed here, but it will be discussed in detail in future documents.

In **Dynamic Semantics**, we treat several examples:

- Semantics of SML, a low level machine language. Of interest here is the treatment of *jumps* and *input/output*.
- Semantics of CAM, the above-mentioned abstract machine code for applicative languages. Here, treatment of the environment as a datum is interesting.
- Semantics of ASPLE. This is a rather trivial example, since difficulties have been dealt with at type-checking time. But *input/output* has to be done carefully. Having both a semantics for ASPLE and SML, we explain next how to execute partially compiled programs, i.e. ASPLE programs in which some fragment may have been replaced by its compiled version.
- Semantics of a reduced version of ML. The resulting specification is compact and particularly pleasing. The equations describing the Mini-ML language were studied from the angle of mixing interpreted and compiled execution. Due to the presence of *closures* in Mini-ML, the problem is particularly difficult and interesting.
- Semantics of Standard ML. We have worked directly from a formal specification written by Milner[STML]. Two traits of this language make it interesting for us: exceptions and pattern matching.

Experimenting with these semantic definitions has already led us to revise the TYPOL formalism and its compiler to Prolog. The abstract syntax of TYPOL was extended to incorporate conditions and actions. The compiler was completed with a mode that is particularly suitable to tracing inference steps, following work by A. Porto in Lisbon. A type-checker for TYPOL was designed and implemented, but it is not in a sufficiently finished state to be presented here. It was felt that substantial effort deserved to be put in the design and implementation of the TYPOL pretty-printer: formal specifications should be as close to conventional mathematical definitions as possible.

In the future, it is believed that the examples shown below make a good collection of test cases for the project. Of course, a few more examples might be needed as well; in particular TYPOL itself, as a prototypical logic programming language.

2. A standard example: ASPLE

This example is in the tradition of Algol-like languages, even with an Algol-68 flavor [DKL]. As we can see in the abstract syntax of this language, an Asple program consists of two parts. The first part is a declarative part where all variables of the program must be declared. The second part is a list of statements.

Objects in Asple are constants (integer and boolean), and variables. These variables may have mode int (for integer), bool (for boolean), or reference to an object.

Additional operators are used to describe the deep abstract syntax of Asple (i.e. the abstract syntax on which semantics is defined). After type-checking, identifiers in expressions are prefixed by the proper number of explicit dereferencing, and the mode of arguments in a read/write statement is also made explicit. This introduces three more operators in the abstract syntax (deref, tinput, touput) and one extra sort (VAR). Transforming the abstract tree during type-checking, keeping some type information in the abstract syntax tree, will make the specification of dynamic semantics or translation easier.

Abstract Syntax of ASPLE

sorts

VAR, ID, OP, EXP, STM, IDLIST, MODE, DECL, PROGRAM, STMS, DECLS

subsorts

EXP> VAR> ID

operators

'Programs in ASPLE'

```
program  :  DECLS×STMS   →  PROGRAM
decls    :  DECL*        →  DECLS
stms     :  STM*         →  STMS
```

'Declarations'

```
decl    :  MODE×IDLIST  →  DECL
idlist  :  ID+          →  IDLIST
bool    :               →  MODE
int     :               →  MODE
ref     :  MODE         →  MODE
```

'Statements'

```
assign  :  ID×EXP          →  STM
if      :  EXP×STMS×STMS   →  STM
while   :  EXP×STMS        →  STM
input   :  ID              →  STM
output  :  EXP             →  STM
```

'Expressions'

```
bop        :  EXP×OP×EXP  →  EXP
plus       :              →  OP
times      :              →  OP
equal      :              →  OP
different  :              →  OP
and        :              →  OP
or         :              →  OP
```

'Constants and Identifiers'

id	:		→	ID
number	:		→	EXP
boolean	:		→	EXP

'Additional Operators'

deref	:	VAR	→	VAR
tinput	:	VAR×MODE	→	STM
toutput	:	EXP×MODE	→	STM

2.1. Static semantics

Rules describing the static semantics of Asple are given in the ASPLE_TC program.

We use three main predicates. Predicate ":" may be read as "has type". Predicate "→" may be read as "produces". It is used while building environment. The third predicate is not named. It may be read as "this piece of program is well typed".

The first rule explains that the declarative part of an Asple program is used to build an environment which then allows to type-check the statement part. It can be paraphrased as follow: "An Asple program is well types if, given the empty environment, the declarative part produces an environment in which the statement part is well typed". The empty environment is noted $\rho_\emptyset$.

Rule number 2 expresses the fact that an empty list of declarations does not affect the environment. Notice that to avoid any ambiguity, empty lists are not pretty-printed using concrete syntax.

In the third rule, we can see that the elaboration of declarations is linear. The first declaration produces an environment which is used to elaborate the following declarations.

Identifiers declared with a certain mode are entered in the environment with a mode which is a reference to the declared mode. This can be seen in rule number 4. So, a boolean (or an integer) will be assigned the mode bool (or int) (see rules 15 & 16), and a variable declared as int will be assigned the mode ref(int).

Rule 6 uses the set of rules DECLARE to increment the environment with the couple (ID,μ), where ID is an identifier, and μ its mode. In set DECLARE, rule 21 expresses the fact that an identifier can't be declared twice.

In rule 8, we can remark that all statements are typed using the same environment.

Rules 9, 10 and 11 need constraints on modes. These constraints (in fact two predicates) are defined in the set TYPE_COERCION which defines an order on type expressions. Predicate IS_BOOLEAN is defined using this order (see rule 25).

Rule 14 finds the mode of an identifier in the environment, using the set TYPE_OF. Notice that because an identifier may be introduced in the environment only once, (see condition on rule 21), no condition is needed here before recursion in rule 22.

Rules 17 to 19 compute the mode of expression. The contraints on OP, in rule 19, make these three rules exclusive. The set RESULT_TYPE computes the result type of an expression. It is an integer if both subexpressions are basically integer, and boolean if both subexpressions are basically boolean. The result type is ignored if the operator is a test operator.

program ASPLE_TC **is**

use ASPLE

$\rho, \rho_1, \rho_2 : \mathrm{ENV}$

$\mu, \mu_1, \mu_2, \mu_3 : \mathrm{MODE}$

$$\frac{\rho_\emptyset \vdash \mathrm{DECLS} \rightarrow \rho \qquad \rho \vdash \mathrm{STMS}}{\vdash \mathbf{begin}\ \mathrm{DECLS}\ \mathrm{STMS}\ \mathbf{end}} \qquad (1)$$

$$\rho \vdash \text{decls}[\,] \longrightarrow \rho \tag{2}$$

$$\frac{\rho \vdash \text{DECL} \longrightarrow \rho_1 \qquad \rho_1 \vdash \text{DECLS} \longrightarrow \rho_2}{\rho \vdash \text{DECL}; \text{DECLS} \longrightarrow \rho_2} \tag{3}$$

$$\frac{\rho \vdash \text{IDLIST}, \text{ref}(\text{MODE}) \longrightarrow \rho_1}{\rho \vdash \text{MODE} : \text{IDLIST} \longrightarrow \rho_1} \tag{4}$$

$$\rho \vdash \text{idlist}[\,], \mu \longrightarrow \rho \tag{5}$$

$$\frac{\rho \overset{\text{declare}}{\vdash} \text{ID}, \mu \longrightarrow \rho_1 \qquad \rho_1 \vdash \text{IDLIST}, \mu \longrightarrow \rho_2}{\rho \vdash \text{ID}, \text{IDLIST}, \mu \longrightarrow \rho_2} \tag{6}$$

$$\rho \vdash \text{stms}[\,] \tag{7}$$

$$\frac{\rho \vdash \text{STM} \qquad \rho \vdash \text{STMS}}{\rho \vdash \text{STM}; \text{STMS}} \tag{8}$$

$$\frac{\rho \vdash \text{ID} : \mu_1 \qquad \rho \vdash \text{EXP} : \mu_2 \qquad \overset{\text{type\_coercion}}{\vdash} \mu_1 \leq \text{ref}(\mu_2)}{\rho \vdash \text{ID} := \text{EXP}} \tag{9}$$

$$\frac{\rho \vdash \text{EXP} : \mu \qquad \overset{\text{type\_coercion}}{\vdash} \mathit{IS\_BOOLEAN}(\mu) \qquad \rho \vdash \text{STMS}_1 \qquad \rho \vdash \text{STMS}_2}{\rho \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi}} \tag{10}$$

$$\frac{\rho \vdash \text{EXP} : \mu \qquad \overset{\text{type\_coercion}}{\vdash} \mathit{IS\_BOOLEAN}(\mu) \qquad \rho \vdash \text{STMS}}{\rho \vdash \text{while EXP do STMS end}} \tag{11}$$

$$\frac{\rho \vdash \text{ID} : \mu}{\rho \vdash \text{input ID}} \tag{12}$$

$$\frac{\rho \vdash \text{EXP} : \mu}{\rho \vdash \text{output EXP}} \tag{13}$$

$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{id}\,\text{X} : \mu}{\rho \vdash \text{id}\,\text{X} : \mu} \tag{14}$$

$$\rho \vdash \text{boolean}\,\text{X} : \text{bool} \tag{15}$$

$$\rho \vdash \text{number}\,\text{X} : \text{int} \tag{16}$$

$$\frac{\rho \vdash \text{EXP}_1 : \mu_1 \qquad \rho \vdash \text{EXP}_2 : \mu_2 \qquad \overset{\text{result\_type}}{\vdash} \mu_1, \mu_2 : \mu_3}{\rho \vdash \text{EXP}_1 \neq \text{EXP}_2 : \text{bool}} \tag{17}$$

$$\frac{\rho \vdash \text{EXP}_1 : \mu_1 \qquad \rho \vdash \text{EXP}_2 : \mu_2 \qquad \overset{\text{result\_type}}{\vdash} \quad \mu_1, \mu_2 : \mu_3}{\rho \vdash \text{EXP}_1 \ = \ \text{EXP}_2 : \text{bool}} \tag{18}$$

$$\frac{\rho \vdash \text{EXP}_1 : \mu_1 \qquad \rho \vdash \text{EXP}_2 : \mu_2 \qquad \overset{\text{result\_type}}{\vdash} \quad \mu_1, \mu_2 : \mu}{\rho \vdash \text{EXP}_1 \text{ OP } \text{EXP}_2 : \mu} \quad (\text{OP} \neq \text{'='}, \text{OP} \neq \text{'}\neq\text{'})$$

set DECLARE is

μ, μ' : MODE

x, y : ID

$$\rho_\emptyset \vdash x, \mu \rightarrow x : \mu \tag{20}$$

$$\frac{\text{L} \vdash x, \mu \rightarrow \text{L}_1}{y : \mu' \cdot \text{L} \vdash x, \mu \rightarrow y : \mu' \cdot \text{L}_1} \quad (x \neq y) \tag{21}$$

end DECLARE;

set TYPE_OF is

μ : MODE

x : ID

$$x : \mu \cdot \text{L} \vdash x : \mu \tag{21}$$

$$\frac{\text{L} \vdash x : \mu}{\text{E} \cdot \text{L} \vdash x : \mu} \tag{22}$$

end TYPE_OF;

set TYPE_COERCION is

μ, μ' : MODE

$$\vdash \mu \leq \mu \tag{23}$$

$$\frac{\vdash \mu \leq \mu'}{\vdash \mu \leq \text{ref}(\mu')} \tag{24}$$

$$\frac{\vdash \text{bool} \leq \mu}{\vdash \mathit{IS\_BOOLEAN}(\mu)} \tag{25}$$

end TYPE_COERCION;

set RESULT_TYPE is

μ, μ', μ_1, μ_2 : MODE

$$\frac{\vdash \mu_1, \mu_2 : \mu}{\vdash \mathrm{ref}(\mu_1), \mathrm{ref}(\mu_2) : \mu} \tag{26}$$

$$\frac{\vdash \mu', \mu : \mu}{\vdash \mathrm{ref}(\mu'), \mu : \mu} \tag{27}$$

$$\frac{\vdash \mu, \mu' : \mu}{\vdash \mu, \mathrm{ref}(\mu') : \mu} \tag{28}$$

$$\vdash \mu, \mu : \mu \tag{29}$$

end RESULT_TYPE;

end ASPLE_TC

2.2. Translation to deep abstract syntax

Translation to deep abstract syntax is very similar to type-checking. But a new abstract tree is build, in which all rereferencing is explicit. This translation is specified in the ASPLE_STAT program. In fact, ASPLE_STAT can be seen as an other way to type-check Asple programs.

Sets TYPE_OF and TYPE_COERCION are imported from the ASPLE_TC program. We also import the ASPLE_TC program itself.

ASPLE_STAT defined only one predicate ("→"). Its meaning is "can be rewritten as". In some places, it involves an extra parameter which is the mode of the rebuilt tree.

The first rule explains how to rewrite a complete Asple program. The environment is built exactly as it was in ASPLE_TC. So we make an explicit call to this set of rules. Rewriting the statement part is done using the current (ASPLE_STAT) set of rules. Because the declarative part is not touched by the translation, it is taken as it was before in the initial abstract tree.

Rules 2 and 3 are straight-forward. Rule 4 is the first rule of interest. It expresses that after rewriting, the mode of the left side of the assignment must be exactly a reference to the mode of the right side. In fact, the left side is not affected by the translation. Only the right hand side must be dereferenced the proper number of times, if necessary (This is done in rule 11 and 12).

Rules 5 to 10 are not very difficult. The only important point is that constraints on modes are very strict. After rewriting, the mode of the expression in an if statement must be exactly bool, and not greater than bool as in ASPLE_TC. This remark holds for the while, input and output statements as well. Notice that input and output nodes are translated in tinput and touput nodes, in which type information is recorded, to allow easier dynamic semantics definition.

Rule 11 finds type information in the environment. Rule 12 expresses the fact that an identifier may be dereferenced a certain number of times, depending on its declared mode.

The last rules are straight-forward.

program ASPLE_STAT is

use ASPLE

import TYPE_OF, TYPE_COERCION **from** ASPLE_TC

import ASPLE_TC

ρ, ρ_1, ρ_2 : ENV

μ, μ' : MODE

$$\frac{\rho\emptyset \overset{\text{asple\_tc}}{\vdash} \text{DECLS} \rightarrow \rho \qquad \rho \vdash \text{STMS} \rightarrow \text{STMS}'}{\vdash \textbf{begin}\ \text{DECLS STMS}\ \textbf{end} \rightarrow \textbf{begin}\ \text{DECLS STMS}'\ \textbf{end}} \tag{1}$$

$$\rho \vdash \text{stms}[] \rightarrow \text{stms}[] \tag{2}$$

$$\frac{\rho \vdash \text{STM} \rightarrow \text{STM}' \qquad \rho \vdash \text{STMS} \rightarrow \text{STMS}'}{\rho \vdash \text{STM; STMS} \rightarrow \text{STM}'; \text{STMS}'} \tag{3}$$

$$\frac{\rho \vdash \text{ID} \rightarrow \text{ref}(\mu), \text{ID}' \qquad \rho \vdash \text{EXP} \rightarrow \mu, \text{EXP}'}{\rho \vdash \text{ID} := \text{EXP} \rightarrow \text{ID}' := \text{EXP}'} \tag{4}$$

$$\frac{\rho \vdash \text{EXP} \rightarrow \text{bool}, \text{EXP}' \qquad \rho \vdash \text{STMS}_1 \rightarrow \text{STMS}'_1 \qquad \rho \vdash \text{STMS}_2 \rightarrow \text{STMS}'_2}{\rho \vdash \textbf{if}\ \text{EXP}\ \textbf{then}\ \text{STMS}_1\ \textbf{else}\ \text{STMS}_2\ \textbf{fi} \rightarrow \textbf{if}\ \text{EXP}'\ \textbf{then}\ \text{STMS}'_1\ \textbf{else}\ \text{STMS}'_2\ \textbf{fi}} \tag{5}$$

$$\frac{\rho \vdash \text{EXP} \rightarrow \text{bool}, \text{EXP}' \qquad \rho \vdash \text{STMS} \rightarrow \text{STMS}'}{\rho \vdash \textbf{while}\ \text{EXP}\ \textbf{do}\ \text{STMS}\ \textbf{end} \rightarrow \textbf{while}\ \text{EXP}'\ \textbf{do}\ \text{STMS}'\ \textbf{end}} \tag{6}$$

$$\frac{\rho \vdash \text{ID} \rightarrow \text{ref}(\text{int}), \text{ID}'}{\rho \vdash \textbf{input}\ \text{ID} \rightarrow \textit{tinput}\ \text{ID}'\ \text{int}} \tag{7}$$

$$\frac{\rho \vdash \text{ID} \rightarrow \text{ref}(\text{bool}), \text{ID}'}{\rho \vdash \textbf{input}\ \text{ID} \rightarrow \textit{tinput}\ \text{ID}'\ \text{bool}} \tag{8}$$

$$\frac{\rho \vdash \text{EXP} \rightarrow \text{int}, \text{EXP}'}{\rho \vdash \textbf{output}\ \text{EXP} \rightarrow \textit{toutput}\ \text{EXP}'\ \text{int}} \tag{9}$$

$$\frac{\rho \vdash \text{EXP} \rightarrow \text{bool}, \text{EXP}'}{\rho \vdash \textbf{output}\ \text{EXP} \rightarrow \textit{toutput}\ \text{EXP}'\ \text{bool}} \tag{10}$$

$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{id}\,\text{x} : \mu}{\rho \vdash \text{id}\,\text{x} \rightarrow \mu, \text{id}\,\text{x}} \tag{11}$$

$$\frac{\rho \overset{\text{type\_of}}{\vdash} \text{id}\,\text{x} : \mu' \qquad \overset{\text{type\_coercion}}{\vdash} \mu \leq \mu' \qquad \rho \vdash \text{id}\,\text{x} \rightarrow \text{ref}(\mu), \text{VAR}}{\rho \vdash \text{id}\,\text{x} \rightarrow \mu, \text{deref VAR}} \tag{12}$$

$$\rho \vdash \text{boolean}\,\text{x} \rightarrow \text{bool}, \text{boolean}\,\text{x} \tag{13}$$

$$\rho \vdash \text{number}\,\text{x} \rightarrow \text{int}, \text{number}\,\text{x} \tag{14}$$

$$\frac{\rho \vdash \text{EXP}_1 \rightarrow \text{int}, \text{EXP}'_1 \qquad \rho \vdash \text{EXP}_2 \rightarrow \text{int}, \text{EXP}'_2}{\rho \vdash \text{EXP}_1 \neq \text{EXP}_2 \rightarrow \text{bool}, \text{EXP}'_1 \neq \text{EXP}'_2} \tag{15}$$

$$\frac{\rho \vdash \text{EXP}_1 \rightarrow \text{int}, \text{EXP}'_1 \qquad \rho \vdash \text{EXP}_2 \rightarrow \text{int}, \text{EXP}'_2}{\rho \vdash \text{EXP}_1 = \text{EXP}_2 \rightarrow \text{bool}, \text{EXP}'_1 = \text{EXP}'_2} \tag{16}$$

$$\frac{\rho \vdash \text{EXP}_1 \rightarrow \text{int}, \text{EXP}'_1 \qquad \rho \vdash \text{EXP}_2 \rightarrow \text{int}, \text{EXP}'_2}{\rho \vdash \text{EXP}_1 \text{ OP } \text{EXP}_2 \rightarrow \text{int}, \text{EXP}'_1 \text{ OP } \text{EXP}'_2} \quad (\text{OP} \neq \text{'='}, \text{OP} \neq \text{'}\neq\text{'})$$

$$\frac{\rho \vdash \text{EXP}_1 \rightarrow \text{bool}, \text{EXP}'_1 \qquad \rho \vdash \text{EXP}_2 \rightarrow \text{bool}, \text{EXP}'_2}{\rho \vdash \text{EXP}_1 + \text{EXP}_2 \rightarrow \text{bool}, \text{EXP}'_1 \text{ and } \text{EXP}'_2} \tag{18}$$

$$\frac{\rho \vdash \text{EXP}_1 \rightarrow \text{bool}, \text{EXP}'_1 \qquad \rho \vdash \text{EXP}_2 \rightarrow \text{bool}, \text{EXP}'_2}{\rho \vdash \text{EXP}_1 \times \text{EXP}_2 \rightarrow \text{bool}, \text{EXP}'_1 \text{ or } \text{EXP}'_2} \tag{19}$$

$$\frac{\rho \vdash \text{EXP}_1 \rightarrow \text{bool}, \text{EXP}'_1 \qquad \rho \vdash \text{EXP}_2 \rightarrow \text{bool}, \text{EXP}'_2}{\rho \vdash \text{EXP}_1 \text{ OP } \text{EXP}_2 \rightarrow \text{bool}, \text{EXP}'_1 \text{ OP } \text{EXP}'_2} \quad (\text{OP} \neq \text{'}\times\text{'}, \text{OP} \neq \text{'+'})$$

end ASPLE_STAT

2.3. Dynamic semantics of ASPLE

The dynamic semantics of our little imperative language is described by the following formal system where sequents have three different forms:

- For a declaration d:

$$s \vdash d : s'$$

may be read as "elaborating a declaration d in the store s produces a new store s'".

- For a statement *stm*:

$$s, i \vdash stm : s', o$$

means "the execution of a statement *stm* in a store s reads an input i and yields a new store s' and an output o". As there are no procedures in Asple, a store is directly a mapping from identifiers to values.

- For an expression e:

$$s \vdash e : v$$

means "in store s expression e evaluates to v".

The semantic domain contains integers $\mathbb{N}$ and truth values *true*, *false*. Constants $o_\emptyset$, $s_\emptyset$, denote the empty output, store, ...

program ASPLE_DS is

use ASPLE

use *STORE*

x, x_1, x_2, v, v_1 : *VALUE*;

s, s_1, s_2 : *STORE*;

i, i_1 : *INPUT*;

o, o_1 : *OUTPUT*;

$$\frac{s_\emptyset \vdash \text{DECLS} : s_1 \qquad s_1, i \vdash \text{STMS} : s_2, o}{i \vdash \textbf{begin}\ \text{DECLS STMS}\ \textbf{end} : o} \tag{1}$$

$$s \vdash \text{decls}[] : s \tag{2}$$

$$\frac{s \vdash \text{DECL} : s_1 \qquad s_1 \vdash \text{DECLS} : s_2}{s \vdash \text{DECL}; \text{DECLS} : s_2} \tag{3}$$

$$\frac{s \vdash \text{IDLIST} : s_1}{s \vdash \text{MODE} : \text{IDLIST} : s_1} \tag{4}$$

$$\frac{s \overset{\text{allocate}}{\vdash} s\_id\,\text{ID} : s_1}{s \vdash \text{id}\,\text{ID} : s_1} \tag{5}$$

$$\frac{s \overset{\text{allocate}}{\vdash} s\_id\,\text{ID} : s_1 \qquad s_1 \vdash \text{IDLIST} : s_2}{s \vdash \text{id}\,\text{ID}, \text{IDLIST} : s_2} \tag{6}$$

$$s, i_{\emptyset} \vdash \text{stms}[\,] : s, o_{\emptyset} \tag{7}$$

$$\frac{s \vdash \text{VAR} : x \qquad s \overset{\text{update}}{\vdash} x, v : s_1 \qquad s_1, i \vdash \text{STMS} : o}{s, v \cdot i \vdash \textit{tinput}\ \text{VAR MODE; STMS} : s_1, o} \tag{8}$$

$$\frac{s \vdash \text{EXP} : v \qquad s, i \vdash \text{STMS} : s, o}{s, i \vdash \textit{toutput}\ \text{EXP MODE; STMS} : s, v \cdot o} \tag{9}$$

$$\frac{s, i \vdash \text{STM} : s_1, o \qquad s_1, i_1 \vdash \text{STMS} : s_2, o_1}{s, i \cdot i_1 \vdash \text{STM; STMS} : s_2, o \cdot o_1} \tag{10}$$

$$\frac{s \vdash \text{ID} : x \qquad s \vdash \text{EXP} : v \qquad s \overset{\text{update}}{\vdash} x, v : s_1}{s, i \vdash \text{ID} := \text{EXP} : s_1, o} \tag{11}$$

$$\frac{s \vdash \text{EXP} : \textit{true} \qquad s, i \vdash \text{STMS}_1 : s_1, o}{s, i \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi} : s_1, o} \tag{12}$$

$$\frac{s \vdash \text{EXP} : \textit{false} \qquad s, i \vdash \text{STMS}_2 : s_1, o}{s, i \vdash \text{if EXP then STMS}_1 \text{ else STMS}_2 \text{ fi} : s_1, o} \tag{13}$$

$$\frac{s \vdash \text{EXP} : \textit{true} \qquad s, i \vdash \text{STMS} : s_1, o \qquad s_1, i_1 \vdash \text{while EXP do STMS end} : s_2, o_1}{s, i \cdot i_1 \vdash \text{while EXP do STMS end} : s_2, o \cdot o_1} \tag{14}$$

$$\frac{s \vdash \text{EXP} : \textit{false}}{s, i \vdash \text{while EXP do STMS end} : s, o_{\emptyset}} \tag{15}$$

$$s \vdash \text{id}\,\text{x} : s\_id\,\text{x} \tag{16}$$

$$\frac{s \vdash \text{EXP} : x \qquad s \overset{\text{store}}{\vdash} x \mapsto v}{s \vdash \text{deref EXP} : v} \tag{17}$$

$$s \vdash \text{boolean}\,\text{X} : \text{X} \tag{18}$$

$$s \vdash \text{number}\,\text{X} : \text{X} \tag{19}$$

$$\frac{s \vdash \text{EXP}_1 : x_1 \qquad s \vdash \text{EXP}_2 : x_2 \qquad \overset{\text{eval}}{\vdash} \text{OP}, x_1, x_2 : x}{s \vdash \text{EXP}_1 \text{ OP EXP}_2 : x} \tag{20}$$

set ALLOCATE is

$$s \vdash \text{S\_ID} : \text{S\_ID} \mapsto \bot \cdot s \tag{1}$$

end ALLOCATE;

set UPDATE is

$$\text{S\_ID} \mapsto \text{V} \cdot \text{S} \vdash \text{S\_ID}, v_1 : \text{S\_ID} \mapsto v_1 \cdot \text{S} \tag{1}$$

$$\frac{\text{S} \vdash s\_id, v_1 : \text{S}_1}{\text{M} \cdot \text{S} \vdash s\_id, v_1 : \text{M} \cdot \text{S}_1} \tag{2}$$

end UPDATE;

set STORE is

$$\mathrm{S\_ID} \mapsto \mathrm{V} \cdot \mathrm{S} \vdash \mathrm{S\_ID} \mapsto \mathrm{V} \qquad (1)$$

$$\frac{\mathrm{S} \vdash s\_id \mapsto v_1}{\mathrm{M} \cdot \mathrm{S} \vdash s\_id \mapsto v_1} \qquad (2)$$

end STORE;

end ASPLE_DS

Comments on the formal definition:

Points of interest here are the treatment of input and output. Rule (9) says - in short - that if s produces an output o, then $write(v); s$ produces an output $v \cdot o$. The execution of a program p corresponds to the proof tree of $i \vdash p : o$ where o is a variable, instantiating a little bit more each time a *write* executes. From the implementation point of view, the idea is that a variable of type Output is treated in a special way: each times there is a substitution on it, the corresponding *write*-action is done. This is not yet implemented. Input is treated in the same way. As an input is an hypothesis of a sequent, it means that a program begins to execute with an input variable, reads its input values one by one, and produces the (input) hypothesis it needs to run.

To be consistent with the style adopted in the static semantics of Asple, rules (2) of the set update might have been written:

$$\frac{\mathrm{S} \vdash s\_id, v_1 : \mathrm{S}_1}{\mathrm{S\_ID} \mapsto \mathrm{V} \cdot \mathrm{S} \vdash s\_id, v_1 : \mathrm{S\_ID} \mapsto \mathrm{V} \cdot \mathrm{S}_1} \qquad (\mathrm{S\_ID} \neq s\_id)$$

But, as there are no double declarations in Asple, this would just be an optimisation of our semantics.

2.4. A Simple Machine Language: SML

SML is a "P-Code-like" machine language, mainly consisting in *load*, *store*, and *jump* instructions:

Abstract Syntax of SML

sorts

SML_ID, CONSTANT, PROGRAM, COMS, COM

subsorts

COM> COMS

operators

'program'

coms	:	COM*	→	COMS
program	:	COMS	→	PROGRAM
block	:	COMS	→	COM
ldo	:	SML_ID	→	COM
sro	:	SML_ID	→	COM
lao	:	SML_ID	→	COM
ind	:		→	COM
sto	:		→	COM
op	:		→	COM
ujp	:		→	COM
fjp	:		→	COM
tjp	:		→	COM
lbl	:		→	COM
nop	:		→	COM
ldci	:	CONSTANT	→	COM

s_read	:	$\rightarrow$	COM
s_write	:	$\rightarrow$	COM

'atoms'

sml_number	:	$\rightarrow$	CONSTANT
sml_boolean	:	$\rightarrow$	CONSTANT
sml_id	:	$\rightarrow$	SML_ID

2.4.1. Dynamic semantics of SML

The dynamic semantics of our Simple Machine Language is described by the following formal system where sequents have two different forms:

- For a command c which may cause a jump or an input/ouput:

$$\rho, \sigma, i \vdash c : \sigma', o$$

may be read as "executing a code c in the environment ρ and the state σ takes the state to σ', reads the input i and produces an output o". An environment ρ is a mapping from labels to continuations, a store σ is a pair $<s, k>$ of a store s and an evaluation stack k. A store is a mapping from addresses [1] to values.

- For other commands c:

$$\sigma \vdash c : \sigma'$$

means "executing a code c in the state σ take it to σ'".

The semantic domain contains integers $\mathbb{N}$ and truth values *true*, *false*.

program SML_DS **is**
use SML
use *STORE*
$x, v, v_1, v_2 : VALUE;$
$\sigma, \sigma_1, \sigma_2 : STATE;$
$s, s_1, s_2 : STORE;$
$k : STACK;$
$\rho, \rho_1 : ENV;$
$c, c_1 : SML;$
$i, i_1 : INPUT;$
$o, o_1 : OUTPUT;$

$$\frac{\rho_\emptyset, <s, k_\emptyset>, i \vdash \text{COMS} : \sigma_1, o}{s, i \vdash program(\text{COMS}) : o} \quad (1)$$

$$\rho, \sigma, i_\emptyset \vdash \text{coms}[] : \sigma, o_\emptyset \quad (2)$$

$$\frac{\rho \overset{\text{cont\_find}}{\vdash} \text{lbl}\,\text{L} \mapsto c_1 \qquad \rho, \sigma, i \vdash c_1 : \sigma_1, o}{\rho, \sigma, i \vdash \text{ujp}\,\text{L}; c : \sigma_1, o} \quad (3)$$

$$\frac{\rho, <s, k>, i \vdash c : \sigma, o}{\rho, <s, true \cdot k>, i \vdash \text{fjp}\,\text{L}; c : \sigma, o} \quad (4)$$

[1] For the sake of simplicity addresses are identifiers: the one-one mapping from Asple-identifiers to Sml-addresses will thus be just the identity.

$$\frac{\rho \overset{\text{cont\_find}}{\vdash} \text{lbl}\,\mathrm{L} \mapsto c_1 \qquad \rho, <s, k>, i \vdash c_1 : \sigma, o}{\rho, <s, \mathit{false} \cdot k>, i \vdash \text{fjp}\,\mathrm{L}; c : \sigma, o} \tag{5}$$

$$\frac{\rho \overset{\text{cont\_find}}{\vdash} \text{lbl}\,\mathrm{L} \mapsto c_1 \qquad \rho, <s, k>, i \vdash c_1 : \sigma, o}{\rho, <s, \mathit{true} \cdot k>, i \vdash \text{tjp}\,\mathrm{L}; c : \sigma, o} \tag{6}$$

$$\frac{\rho, <s, k>, i \vdash c : \sigma, o}{\rho, <s, \mathit{false} \cdot k>, i \vdash \text{tjp}\,\mathrm{L}; c : \sigma, o} \tag{7}$$

$$\frac{s \overset{\text{update}}{\vdash} x, v : s_1 \qquad \rho, <s_1, k>, i \vdash c : \sigma_1, o}{\rho, <s, x \cdot k>, v \cdot i \vdash \text{s\_read}; c : \sigma_1, o} \tag{8}$$

$$\frac{\rho, <s, k>, i \vdash c : \sigma_1, o}{\rho, <s, v \cdot k>, i \vdash \text{s\_write}; c : \sigma_1, v \cdot o} \tag{9}$$

$$\frac{\overset{\text{cons\_env}}{\vdash} \mathrm{COMS} : \rho_1 \qquad \rho_1, \sigma, i \vdash \mathrm{COMS} : \sigma_1, o \qquad \rho_1, \sigma_1, i_1 \vdash c : \sigma_2, o_1}{\rho, \sigma, i \cdot i_1 \vdash \text{block}(\mathrm{COMS}); c : \sigma_2, o \cdot o_1} \tag{10}$$

$$\frac{\sigma \vdash \mathrm{COM} : \sigma_1 \qquad \rho, \sigma_1, i \vdash \mathrm{COMS} : \sigma_2, o}{\rho, \sigma, i \vdash \mathrm{COM}; \mathrm{COMS} : \sigma_2, o} \tag{11}$$

$$\sigma \vdash \text{nop} : \sigma \tag{12}$$

$$<s, k> \vdash \text{ldci}(\text{sml\_number}\, v) : <s, v \cdot k> \tag{13}$$

$$<s, k> \vdash \text{ldci}(\text{sml\_boolean}\, v) : <s, v \cdot k> \tag{14}$$

$$\frac{s \overset{\text{get}}{\vdash} \mathit{s\_id}\, \mathrm{ID} \mapsto v}{<s, k> \vdash \text{ldo}(\text{sml\_id}\, \mathrm{ID}) : <s, v \cdot k>} \tag{15}$$

$$\frac{s \overset{\text{update}}{\vdash} \mathit{s\_id}\, \mathrm{ID}, v : s_1}{<s, v \cdot k> \vdash \text{sro}(\text{sml\_id}\, \mathrm{ID}) : <s_1, k>} \tag{16}$$

$$<s, k> \vdash \text{lao}(\text{sml\_id}\, \mathrm{ID}) : <s, \mathit{s\_id}\, \mathrm{ID} \cdot k> \tag{17}$$

$$\frac{s \overset{\text{get}}{\vdash} x \mapsto v}{<s, x \cdot k> \vdash \text{ind} : <s, v \cdot k>} \tag{18}$$

$$\frac{s \overset{\text{update}}{\vdash} x, v : s_1}{<s, v \cdot x \cdot k> \vdash \text{sto} : <s_1, k>} \tag{19}$$

$$\frac{\overset{\text{eval}}{\vdash} \mathrm{OP}, v_1, v_2 : v}{<s, v_1 \cdot v_2 \cdot k> \vdash \text{op}\, \mathrm{OP} : <s, v \cdot k>} \tag{20}$$

$$\sigma \vdash \text{lbl}\, \mathrm{L} : \sigma \tag{21}$$

set CONT_FIND is

$$\text{lbl}\, \mathrm{L} \mapsto \mathrm{CONT} \cdot \rho \vdash \text{lbl}\, \mathrm{L} \mapsto \mathrm{CONT} \tag{1}$$

$$\frac{\rho \vdash \mathrm{L} \mapsto \mathrm{CONT}}{\mathrm{PAIR} \cdot \rho \vdash \mathrm{L} \mapsto \mathrm{CONT}} \tag{2}$$

end CONT_FIND;

set CONS_ENV is

$$\vdash \text{coms}[\,] : \rho_\emptyset \tag{1}$$

$$\frac{\vdash \text{COMS} : \rho}{\vdash \text{lbl}_L; \text{COMS} : \text{lbl}_L \mapsto \text{COMS} \cdot \rho} \qquad (2)$$

$$\frac{\vdash \text{COMS} : \rho}{\vdash \text{COM}; \text{COMS} : \rho} \qquad (3)$$

end CONS_ENV;

set GET is

$$\text{X} \mapsto v \cdot s \vdash \text{X} \mapsto v \qquad (1)$$

$$\frac{s \vdash \text{X} \mapsto v}{\text{M} \cdot s \vdash \text{X} \mapsto v} \qquad (2)$$

end GET;

set UPDATE is

$$\text{X} \mapsto v_1 \cdot s \vdash \text{X}, v_2 : \text{X} \mapsto v_2 \cdot s \qquad (1)$$

$$\frac{s_1 \vdash \text{X}, v : s_2}{\text{M} \cdot s_1 \vdash \text{X}, v : \text{M} \cdot s_2} \qquad (2)$$

end UPDATE;

end SML_DS

Comments on the formal definition:

Points of interest here are the treatment of jump and input/output. Input/output are treated in the same way as in the dynamic semantics of Asple, with the little improvement that instructions which do not need input/output do not take them as parameter.

For jumps (*ujp*, *tjp*, ...), we follow the idea of the denotational continuations. The scope of a label is a *block*, so each time one enters into a block a mapping from labels to continuations (an environment ρ) is constructed (see the *cons_env* set). Then a jump searches for the right continuation in this environment. To that end, the environment is a parameter of each rule which eventually needs it.

2.5. Translation from ASPLE to SML

The translation from Asple to Sml is described by the following formal system where sequents mainly have two different forms:

- For a declaration d:

$$s \vdash d : s'$$

  may be read as "the translation of an Asple-declaration d with the store s produces a new store s'". The declaration part of an Asple program produces the initial store of the code.

- For statements s:

$$\vdash s \rightarrow c$$

  means "the translation of a statement s is the code c".

program ASPLE_SML is

use ASPLE

use *SML*

use STORE

c, c_1, c_2, c_3 : *SML*;

s, s_1, s_2 : *STORE*;

$$\frac{s_{\emptyset} \vdash \text{DECLS} : s \qquad \vdash \text{STMS} \rightarrow c}{\vdash \textit{program}(\text{DECLS}, \text{STMS}) \rightarrow \textit{program}(c), s} \tag{1}$$

$$s \vdash \text{decls}[] : s \tag{2}$$

$$\frac{s \vdash \text{DECL} : s_1 \qquad s_1 \vdash \text{DECLS} : s_2}{s \vdash \text{DECL}; \text{DECLS} : s_2} \tag{3}$$

$$\frac{s \vdash \text{IDLIST} : s_1}{s \vdash \text{MODE} : \text{IDLIST} : s_1} \tag{4}$$

$$\frac{s \overset{\text{allocate}}{\vdash} s\_id\,\text{ID} : s_1}{s \vdash \text{id}\,\text{ID} : s_1} \tag{5}$$

$$\frac{s \overset{\text{allocate}}{\vdash} s\_id\,\text{ID} : s_1 \qquad s_1 \vdash \text{IDLIST} : s_2}{s \vdash \text{id}\,\text{ID}, \text{IDLIST} : s_2} \tag{6}$$

$$\vdash \text{stms}[] \rightarrow \text{coms}[] \tag{7}$$

$$\frac{\vdash \text{STM} \rightarrow c_1 \qquad \vdash \text{STMS} \rightarrow c_2}{\vdash \text{STM}; \text{STMS} \rightarrow c_1; c_2} \tag{8}$$

$$\frac{\vdash \text{EXP} \rightarrow c}{\vdash \text{id}\,\text{ID} := \text{EXP} \rightarrow c; \text{sro}(\text{sml\_id}\,\text{ID})} \tag{9}$$

$$\frac{\vdash \text{EXP} \rightarrow c_1 \qquad \vdash \text{STMS} \rightarrow c_2}{\vdash \text{if}\,\text{EXP}\,\text{then}\,\text{STMS}\,\text{fi} \rightarrow \text{block}(c_1; \text{fjp}\,1; c_2; \text{lbl}\,1)} \tag{10}$$

$$\frac{\vdash \text{EXP} \rightarrow c_1 \qquad \vdash \text{STMS}_1 \rightarrow c_2 \qquad \vdash \text{STMS}_2 \rightarrow c_3}{\vdash \text{if}\,\text{EXP}\,\text{then}\,\text{STMS}_1\,\text{else}\,\text{STMS}_2\,\text{fi} \rightarrow \text{block}(c_1; \text{fjp}\,1; c_2; \text{ujp}\,2; \text{lbl}\,1; c_3; \text{lbl}\,2)} \tag{11}$$

$$\frac{\vdash \text{EXP} \rightarrow c_1 \qquad \vdash \text{STMS} \rightarrow c_2}{\vdash \text{while}\,\text{EXP}\,\text{do}\,\text{STMS}\,\text{end} \rightarrow \text{block}(\text{lbl}\,1; c_1; \text{fjp}\,2; c_2; \text{ujp}\,1; \text{lbl}\,2)} \tag{12}$$

$$\frac{\vdash \text{VAR} \rightarrow c}{\vdash \textit{tinput}\,\text{VAR}\,\text{MODE} \rightarrow c; \text{s\_read}} \tag{13}$$

$$\frac{\vdash \text{EXP} \rightarrow c}{\vdash \textit{toutput}\,\text{EXP}\,\text{MODE} \rightarrow c; \text{s\_write}} \tag{14}$$

$$\vdash \text{id}\,\text{x} \rightarrow \text{lao}(\text{sml\_id}\,\text{x}) \tag{15}$$

$$\vdash \text{deref}\,\text{id}\,\text{x} \rightarrow \text{ldo}(\text{sml\_id}\,\text{x}) \tag{16}$$

$$\frac{\vdash \text{EXP} \rightarrow c}{\vdash \text{deref}\,\text{EXP} \rightarrow c; \text{ind}} \tag{17}$$

$$\vdash \text{boolean}\,\text{V} \rightarrow \text{ldci}(\text{sml\_boolean}\,\text{V}) \tag{18}$$

$$\vdash \text{number}\,\text{N} \rightarrow \text{ldci}(\text{sml\_number}\,\text{N}) \tag{19}$$

$$\frac{\vdash \text{EXP}_1 \rightarrow c_1 \qquad \vdash \text{EXP}_2 \rightarrow c_2 \qquad \overset{\text{operator}}{\vdash} \text{OP} \rightarrow \text{OP}_1}{\vdash \text{EXP}_1\,\text{OP}\,\text{EXP}_2 \rightarrow c_1; c_2; \text{OP}_1} \tag{20}$$

set OPERATOR is

$$\vdash \text{plus} \rightarrow \text{op}\ \text{“adi”} \tag{1}$$

$$\vdash \text{times} \rightarrow \text{op}\ \text{“mpi”} \tag{2}$$

$$\vdash = \rightarrow \textbf{op} \ \text{“equi”} \tag{3}$$

$$\vdash \neq \rightarrow \textbf{op} \ \text{“neqi”} \tag{4}$$

$$\vdash \textbf{and} \rightarrow \textbf{op} \ \text{“land”} \tag{5}$$

$$\vdash \textbf{or} \rightarrow \textbf{op} \ \text{“lor”} \tag{6}$$

end OPERATOR;

set ALLOCATE is

$$s \vdash \text{S\_ID} : \text{S\_ID} \mapsto \perp \cdot s \tag{1}$$

end ALLOCATE;

end ASPLE_SML

Comments on the formal system:

This translation is rather straightforward. Let us just note that the translation of an *if* (or a *while*) is a block of commands, whose purpose is just to provide a scope to labels. A part from that there are two optimisations: the translation of an "if" with no "else"-part, and the translation of "deref id X", which is an optimisation of the translation of "deref EXP".

3. A Simple Applicative Language: Mini-ML

ML is a programming language with very interesting characteristics from the standpoint of static and dynamic semantics. ML is a strongly typed language but there is no type declaration, i.e. expressions are *implicitly* typed, and it allows *polymorphism*, i.e. to define functions which work uniformly on arguments of many types. ML allows the definition of higher-order functions, i.e. a value of an ML expression may be a *closure*.

ML typechecking is the object of numerous discussions in the literature, e.g. [DM], [Cardelli], [Reynolds], [TD], and the use of an ***inference system*** to describe ML typing is now widely accepted. On the other hand recent work of Curien and Cousineau [CAM] has shown how to *compile* ML into code for an abstract machine, the Categorical Abstract Machine (CAM). Thus it is a natural challenge for TYPOL to specify typechecking, dynamic semantics, and translation into CAM for a mini-ML language. The features of ML retained are those of a simple typed λ-calculus with constants, extended by products and recursive functions.

Outline: For the sake of clarity we first present the type inference system of Damas-Milner for the simple λ-calculus part and then show how this system, which is non-deterministic, may be transformed into a deterministic system. Then we show that the latter system may be written in Typol to produce automatically a Mini-ML typechecker. Finally we give rules that must be included to that Typol system to deal with product and *letrec*. The dynamic semantics for mini-ML is described by an inference system. The nicier points of this system are recursive function specification and management of products. Translation from mini-ML to CAM illustrates how scope rules of λ-calculus may be described with an appropriate environment definition. The CAM is a very simple abstract machine. Transitions of the machine are described quite naturally by a formal system.

Abstract Syntax of Mini-ML

The abstract syntax given bellow describes λ-calculus extended by products , letrec, and if. Now in $\lambda P.e$ P may be an identifier or a pattern. For example $\lambda(x, y).e$ is a valid expression. Notice that the function mlpair stands for product of expressions. The nullpat function is used for the unit object () (which is both a pattern and an expression).

sorts

EXP, IDENT, PAT, NULLPAT

subsorts

EXP$\succ$

NULLPAT, IDENT

PAT$\succ$

NULLPAT, IDENT

functions

'patterns'

pairpat	:	PAT×PAT	→	PAT
nullpat	:		→	NULLPAT

'expressions'

ident	:		→	IDENT
number	:		→	EXP
false	:		→	EXP
true	:		→	EXP
apply	:	EXP×EXP	→	EXP
lambda	:	PAT×EXP	→	EXP
let	:	PAT×EXP×EXP	→	EXP
letrec	:	PAT×EXP×EXP	→	EXP

if	:	EXP×EXP×EXP	$\rightarrow$	EXP
mlpair	:	EXP×EXP	$\rightarrow$	EXP

3.1. Static Semantics

To present typing as an inference system we start with the definition of the type language. In typed λ-calculus every object has a type. Thus the type language must be able to express *ground types* as well as *functions*. For example, the type of the successor function $\lambda x.x + 1$ is $int \rightarrow int$. In the same way the identity function $\lambda x.x$ for integers has type $int \rightarrow int$, but for booleans it has type $bool \rightarrow bool$. It is clear that the identity function may be defined without taking into account the type of its present parameter. The significant point is that the identity function has type $\alpha \rightarrow \alpha$ whatever the value of α is. To express this *abstraction* on the type of the parameter the *type variable* α is bounded by a quantifier. The polymorphic identity function has type $\forall\alpha.\alpha \rightarrow \alpha$.

The Type Language

The type language is defined by the following syntax:

Types τ

type variable α

ground types *int*, *bool*

functions $\tau \rightarrow \tau$

Type Schemes σ

type τ

type_scheme $\forall\alpha.\sigma$

A type expression in this language may have both *free* and *bound* variables. Let us write $FV(\sigma)$ and $BV(\sigma)$ for the sets of free and bound variables of a type expression σ. Next we define relations between type expressions that contain type variables.

Def. A type scheme σ' is called an *instance* of a type scheme σ if there exists a substitution S of types for free type variables such that:

$$\sigma' = S\sigma.$$

Instantiation acts on free variables: if S is written $[\alpha_i \leftarrow \tau_i]$ with $\alpha_i \in FV(\sigma)$ then $S\sigma$ is obtained by replacing each free occurrence of α_i in σ by τ_i (renaming the bound variables of σ if necessary). The *domain* of S is the set $D(S) = \{\alpha \in FV(\sigma) \mid \alpha \text{ is replaced by } \tau\}$.

Def. A type scheme $\sigma = \forall\alpha_1 \cdots \alpha_m.\tau$ has a *generic instance* $\sigma' = \forall\beta_1 \cdots \beta_n.\tau'$, and we shall write $\sigma \succeq \sigma'$, if:

- there exists a substitution S such that

$$\tau' = S\tau \quad \text{with} \quad D(S) \subseteq \{\alpha_1 \cdots \alpha_m\}$$

- the β_j are not free in σ, i.e.

$$\beta_i \notin FV(\sigma) \qquad 1 \le i \le n$$

Generic instantiation acts on bounded variables. Note that if $\sigma \succeq \sigma'$ then for every substitution S, $S\sigma \succeq S\sigma'$.

3.1.1. Damas-Milner Inference System

In programming languages the type of an expression depends upon the type of identifiers that occur free in it. In other words an expression e has type σ under a given set of assumptions A. In the following A is a list of assumption $x : \sigma$ and A_x stands for the result of removing any assumption about x from A. We say that the expression e is *typable* if

$$A \vdash e : \sigma$$

may be derived from the Damas-Milner inference system:

TAUT $$A \vdash x : \sigma \qquad (x : \sigma \in A)$$

INST $$\frac{A \vdash e : \sigma}{A \vdash e : \sigma'} \qquad (\sigma \succeq \sigma')$$

GEN $$\frac{A \vdash e : \sigma}{A \vdash e : \forall \alpha . \sigma} \qquad (\alpha \notin FV(A))$$

APP $$\frac{A \vdash e : \tau' \rightarrow \tau \qquad A \vdash e' : \tau'}{A \vdash e\, e' : \tau}$$

ABS $$\frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \rightarrow \tau}$$

LET $$\frac{A \vdash e : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e' : \tau'}{A \vdash \mathrm{let}\, x = e \,\mathrm{in}\, e' : \tau'}$$

Let us see now how this system may be used to show typings. For example, we can show that the identity function $\lambda x.x$ has type $\forall \alpha . \alpha \rightarrow \alpha$ by the following derivation tree:

$$\cfrac{\cfrac{\{x : \alpha\} \vdash x : \alpha \quad [\mathrm{TAUT}]}{\vdash \lambda x.x : \alpha \rightarrow \alpha \quad [\mathrm{ABS}]}}{\vdash \lambda x.x : \forall \alpha . \alpha \rightarrow \alpha \quad [\mathrm{GEN}]}$$

We can use that proof to get a specialized type for the identity function:

$$\frac{\vdash \lambda x.x : \forall \alpha . \alpha \rightarrow \alpha}{\vdash \lambda x.x : int \rightarrow int} \qquad [\mathrm{INST}]$$

More directly we have also the following proof tree:

$$\frac{\{x : int\} \vdash x : int \quad [\mathrm{TAUT}]}{\vdash \lambda x.x : int \rightarrow int \quad [\mathrm{ABS}]}$$

Now to ***compute*** the type of an expression with such an inference system we must find the ***order*** in which to apply inference rules. With the exception of GEN and INST, all rules are ***exclusive***: there is only one rule to express the typing of each syntactic construct. It is clear that the system is non-deterministic because of rules GEN and INST. The problem reduces to choosing when to use GEN and INST.

3.1.2. Deterministic Inference System

Consider the proof tree for the typing of $\mathrm{let}\, i = \lambda x.x \,\mathrm{in}\, i\, i$:

$$\cfrac{\cfrac{\cfrac{\{x : \alpha\} \vdash x : \alpha}{\vdash \lambda x.x : \alpha \rightarrow \alpha}}{\vdash \lambda x.x : \forall \alpha . \alpha \rightarrow \alpha} \quad \cfrac{\cfrac{\{i : \forall \alpha . \alpha \rightarrow \alpha\} \vdash i : \forall \alpha . \alpha \rightarrow \alpha}{\{i : \forall \alpha . \alpha \rightarrow \alpha\} \vdash i : (\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta)} \quad \cfrac{\{i : \forall \alpha . \alpha \rightarrow \alpha\} \vdash i : \forall \alpha . \alpha \rightarrow \alpha}{\{i : \forall \alpha . \alpha \rightarrow \alpha\} \vdash i : \beta \rightarrow \beta}}{\{i : \forall \alpha . \alpha \rightarrow \alpha\} \vdash i\, i : \beta \rightarrow \beta}}{\vdash \mathrm{let}\, i = \lambda x.x \,\mathrm{in}\, i\, i : \beta \rightarrow \beta}$$

In this example we see that:

1) APP, ABS, and LET rules the variables τ and τ' cannot be quantified.
2) to derive $A \vdash e : \sigma$ in the LET rule we may have to apply the GEN rule. With the rule TAUT, an identifier may be associated to a type scheme in the set A, the rule GEN is the only one that can introduce a quantified type scheme.
3) to derive $A_x \cup \{x : \sigma\} \vdash e' : \tau'$ in the LET rule we may have to use the INST rule after each use of TAUT for x and σ is quantified. For example to apply the APP rule in the above proof tree.

Consider the following slightly different version of the Damas-Milner system:

TAUT $$\frac{\overset{\text{inst}}{\vdash}\ \sigma : \tau}{A \vdash x : \tau} \quad (x : \sigma \in A)$$

APP $$\frac{A \vdash e : \tau' \rightarrow \tau \qquad A \vdash e' : \tau'}{A \vdash e\,e' : \tau}$$

ABS $$\frac{A_x \cup \{x : \tau'\} \vdash e : \tau}{A \vdash \lambda x.e : \tau' \rightarrow \tau}$$

LET′ $$\frac{A \vdash e : \tau \qquad A \overset{\text{gen}}{\vdash} \tau : \sigma \qquad A_x \cup \{x : \sigma\} \vdash e' : \tau'}{A \vdash \text{let}\, x = e \,\text{in}\, e' : \tau'}$$

where the two sequents, $\overset{\text{inst}}{\vdash}$ and $\overset{\text{gen}}{\vdash}$ are defined by the following systems:

$$\text{INST} \quad \begin{cases} \vdash \tau : \tau & \text{if } \tau \text{ is not quantified} \\ \vdash \forall \alpha_1 \cdots \alpha_n.\tau : \tau' & \text{with } \tau' = S\tau \text{ and } D(S) = \{\alpha_1 \cdots \alpha_n\} \end{cases}$$

Notice that quantifiers are stripped: the substitution S acts on all the quantified variables α_i.

$$\text{GEN} \quad \begin{cases} A \vdash \tau : \forall \alpha_1 \cdots \alpha_n.\tau & (FV(\tau) \setminus FV(A) = \{\alpha_1 \cdots \alpha_n\}) \\ A \vdash \tau : \tau & (FV(\tau) \setminus FV(A) = \emptyset) \end{cases}$$

In the same manner all generic variables are quantified.

This system is deterministic. It is clear that the order in which rules are applied is fully determined by the syntactic structure of the expression. To each syntactic operator corresponds only one rule. Given an expression this system describes all its possible typing. But we know that a lower unifier bound may be found by unification (when unifiers exist). Milner,[Milner], has shown that unification may be used to compute a principal type for every typable expression. Here unification is pushed in the *meta* level (i.e. in the formal system).

3.1.3. Type Inference

We now describe how the above system may be used to *find* the typing of an expression, i.e. how it works. The general idea consists in building a proof for $A \vdash e : \tau$ with τ not yet known. In case of success the value of τ is a principal type of the expression e. Due to the lack of type declarations in λ-calculus the set of assumptions on identifiers is unknown when we start the proof. We have to do *type inference.*

To deduce the type of an abstraction, $\lambda x.e$, we must find a proof for the sequent $A_x \cup \{x : \tau'\} \vdash e : \tau$. But we do not know what the value of τ' should be. Thus a type variable is initially associated to the λ bound identifier x. The next rules to apply depend only on the structure of the expression e. Some of them may require the types of two sub-expressions to be identical, for example the rule APP′. It is the unification we are using to build a proof that checks if these types are equal or if there exists a most general unifier that makes them equal. This results in a new instance of the proof tree we are building. Notice that this unification acts only on valid proof trees: at each unification step we can obtain the same goal tree but starting with the present instance of the list of assumptions. Typing succeeds if there exists a complete proof tree in the system.

What about polymorphism? With the exception of $\overset{\text{inst}}{\vdash}$ and $\overset{\text{gen}}{\vdash}$, it is clear that every occurrence of a λ-bound identifier x in an expression e will have the *same* type. For example the typing of the expression $\lambda x.\,\text{let}\, i = x \,\text{in}\, i + i$ may be described by the following goal tree:

$$\cfrac{\cfrac{\{x : \alpha\} \vdash x : \alpha \qquad \overset{\text{gen}}{\vdash} \alpha : \alpha \qquad \{i : \alpha\} \vdash i + i : \tau}{\{x : \alpha\} \vdash \text{let}\, i = x \,\text{in}\, i + i : \tau}}{\vdash \lambda x.\,\text{let}\, i = x \,\text{in}\, i + i : \alpha \rightarrow \tau}$$

where α is a type variable and τ is the type term to find. Assuming that the function $+$ has type $int * int \to int$, to apply the APP$'$ rule both α and τ must be equal to int. Notice that the type variable α was not quantified because it belongs to the set $FV(\{x : \alpha\})$. The typing succeeds with the (incomplete) proof tree:

$$\frac{\dfrac{\{x : int\} \vdash x : int \qquad \overset{\text{gen}}{\vdash}\ int : int \quad \{i : int\} \vdash i + i : int}{\{x : int\} \vdash \text{let}\, i = x \,\text{in}\, i + i : int}}{\vdash \lambda x.\, \text{let}\, i = x \,\text{in}\, i + i : int \to int}$$

Next consider what happends when the LET rule has to be applied. If there exists a proof tree for the sequent $A \vdash e : \tau$, with τ a type expression without quantifier, the sequent $\overset{\text{gen}}{\vdash}$ may transform this type into a type scheme σ which all generic variables are quantified (this corresponds to applying all the possible GEN rules in the Damas-Milner system). Now during the proof of $A_x \cup \{x : \sigma\} \vdash e' : \tau'$ every time we use TAUT$'$ for x and σ is quantified a general instance of σ must be find. Assuming that all quantified variables are changed by new type variables, a first substitution S (second rule of $\overset{\text{inst}}{\vdash}$ definition) is obtained by unification (this corresponds to applying an INST rule in the Damas-Milner system). To different occurrences of the same identifier are associated different substitutions S: it is clear that these different occurrences may have different typing. Later on in the proof building, unification will still act on these new variables, and at the end to every occurrence of $\overset{\text{inst}}{\vdash}$ in the proof tree correspond a precise generic instance σ' of σ. .

This computing mechanism may be illustrated with the $\text{let}\, i = \lambda x.x \,\text{in}\, i\, i$ example. At some stage of the inference, the current goal tree may be represented by the following diagram:

$$\frac{\dfrac{\dfrac{\overset{\text{inst}}{\vdash}\ \alpha : \alpha}{\{x : \alpha\} \vdash x : \alpha}\,[\text{TAUT}']}{\vdash \lambda x.x : \alpha \to \alpha\,[\text{ABS}']} \qquad \overset{\text{gen}}{\vdash}\ \alpha \to \alpha : \forall \alpha.\alpha \to \alpha \qquad \{i : \forall \alpha.\alpha \to \alpha\} \vdash i\, i : \tau}{\vdash \text{let}\, i = \lambda x.x \,\text{in}\, i\, i : \tau}$$

To prove $\{i : \forall \alpha.\alpha \to \alpha\} \vdash i\, i : \tau$ the APP$'$ rule must be used. This rule expresses arrow elimination, i.e. if the first occurrence of i has a type that matches $\tau' \to \tau$ then the second occurrence of i must have a type that matches τ', and the expression $i\, i$ has type τ. Here to show the rule TAUT$'$ for the first occurrence of i we have to find a general instance of $\forall \alpha.\alpha \to \alpha$ that matches a type expression $\tau' \to \tau$, with τ' and τ not yet known. Once the type variable α has been changed by a new type variable α_1 the matching succeeds with the following constraints:

$$\tau' = \alpha_1,$$

$$\tau = \alpha_1.$$

In the same manner, to use the rule TAUT$'$ for the second occurrence of i we have to find a general instance of $\forall \alpha.\alpha \to \alpha$ that matches the type expression τ'. Once the type variable α has been changed to a new type variable α_2 the matching succeeds with the following constraint:

$$\tau' = \alpha_2 \to \alpha_2$$

Now rule APP$'$ requires α_1 to be equal to $\alpha_2 \to \alpha_2$. This last constraint is consistent with the three previous ones and the complete proof tree is:

$$\frac{\begin{array}{c} \dfrac{\dfrac{\overset{\text{inst}}{\vdash}\ \forall \alpha.\alpha \to \alpha : \alpha_1 \to \alpha_1}{\{i : \forall \alpha.\alpha \to \alpha\} \vdash i : (\alpha_2 \to \alpha_2) \to (\alpha_2 \to \alpha_2)} \quad \dfrac{\overset{\text{inst}}{\vdash}\ \forall \alpha.\alpha \to \alpha : \alpha_2 \to \alpha_2}{\{i : \forall \alpha.\alpha \to \alpha\} \vdash i : \alpha_2 \to \alpha_2}}{\{i : \forall \alpha.\alpha \to \alpha\} \vdash i\, i : \alpha_2 \to \alpha_2} \\ \dfrac{\dfrac{\overset{\text{inst}}{\vdash}\ \alpha : \alpha}{\{x : \alpha\} \vdash x : \alpha}}{\vdash \lambda x.x : \alpha \to \alpha} \quad \overset{\text{gen}}{\vdash}\ \alpha \to \alpha : \forall \alpha.\alpha \to \alpha \end{array}}{\vdash \text{let}\, i = \lambda x.x \,\text{in}\, i\, i : \alpha_2 \to \alpha_2}$$

3.1.4. Typol Specification

The Typol specification of the deterministic system is straightforward except for union and find operations on sets of assumptions, and for the two sequents $\overset{gen}{\vdash}$ and $\overset{inst}{\vdash}$. Consider first the so-called environment manipulations. It is clear that they do not depend on any particular inference system. To emphasize this fact they are described with two specialized Typol sets DECLARE and TYPEOF.

Now the formula $A_x \cup \{x : \sigma\}$ is implemented by the set named DECLARE.

set DECLARE is

$$[] \vdash x, \sigma : [x : \sigma]$$

$$[x : \sigma' \cdot A] \vdash x, \sigma : [x : \sigma \cdot A]$$

$$\frac{A \vdash x, \sigma : A'}{[y : \sigma' \cdot A] \vdash x, \sigma : [y : \sigma' \cdot A']} \qquad y \neq x$$

end DECLARE;

To add a new assumption on an identifier x into a list of assumptions A we must consider two cases:

1) the list does not contain any assumption on x. This corresponds to adding an assumption into an empty list (rule 1).
2) the list contains an assumption on x that has to be changed. This corresponds to a list starting by an assumption on x (rule 2).

When the two previous statements are not directly satisfied we may have to iterate on the list. This is expressed by the third rule.

In the rule TAUT′ to show that the assumption $x : \sigma$ belongs to the current set of assumptions A we use the Typol set TYPEOF.

set TYPEOF is

$$[x : \sigma \cdot A] \vdash x : \sigma$$

$$\frac{A \vdash x : \sigma}{[y : \sigma' \cdot A] \vdash x : \sigma} \qquad y \neq x$$

end TYPEOF;

Then the deterministic system is written in Typol as follows:

set TYPE is

$$A \vdash \text{true} : \mathit{bool}$$

$$A \vdash \text{false} : \mathit{bool}$$

$$A \vdash \text{number N} : \mathit{int}$$

$$\frac{A \overset{typeof}{\vdash} \text{ident X} : \sigma \qquad \overset{inst}{\vdash} \sigma : \tau}{A \vdash \text{ident X} : \tau}$$

$$\frac{A \vdash \mathrm{E} : \tau' \rightarrow \tau \qquad A \vdash \mathrm{E}' : \tau'}{A \vdash \mathrm{E}\,\mathrm{E}' : \tau}$$

$$\frac{A \overset{\text{declare}}{\vdash} \mathrm{x}, \tau' : A' \qquad A' \vdash \mathrm{E} : \tau}{A \vdash \lambda \mathrm{x}.\mathrm{E} : \tau' \rightarrow \tau}$$

$$\frac{A \vdash \mathrm{E} : \tau \qquad A \overset{\text{gen}}{\vdash} \tau : \sigma \qquad A \overset{\text{declare}}{\vdash} \mathrm{x}, \sigma : A' \qquad A' \vdash \mathrm{E}' : \tau'}{A \vdash \mathrm{let}\,\mathrm{x} = \mathrm{E}\,\mathrm{in}\,\mathrm{E}' : \tau'}$$

end TYPE;

Remark: It is not clear that environment manipulations are best described by Typol sets. As noticed above we tend to think that removal of an assumption about an identifier x, A_x, and union of assumptions, $A \cup B$, should be defined as constructs in the Typol language. For example, the TAUT′ rule could be written in Typol as:

$$\frac{\overset{\text{inst}}{\vdash} \sigma : \tau}{A \cup \{x : \sigma\} \vdash x : \tau}$$

Next consider the two sequents that manipulate type schemes. The sequent $\overset{\text{gen}}{\vdash}$ may be described with two specialized Typol sets FREEVARS and SETMINUS. The set FREEVARS builds the list of free variables that occur in a term. To build $FV(A)$ we use FREEVARS for each assumption: we build the local list of free variables that occur in the type expression of that assumption. The whole list of free variables is obtained as the exclusive union of all these local lists. Then we must implement the set difference operation $FV(\sigma) \setminus FV(A)$. This is done with another specialized Typol set, SETMINUS. The resulting list contains all the free variables that can be quantified.

set GEN is

$$\frac{\overset{\text{freevars}}{\vdash} A : l \qquad \overset{\text{freevars}}{\vdash} \tau : l' \qquad \overset{\text{setminus}}{\vdash} l', l : l''}{A \vdash \tau : \forall l''.\tau}$$

end GEN;

The sequent $\overset{\text{inst}}{\vdash}$ describe how to find a generic instance τ of a quantified type expression σ. We replace each occurrence of a type variable that belongs to the $\forall$-list by the same fresh type variable. Then we use unification to match this type expression with τ.

set INST is

$$\frac{l \overset{\text{rename}}{\vdash} \tau' : \tau}{\vdash \forall l.\tau' : \tau}$$

$$\vdash \tau : \tau$$

end INST;

We may notice that the sequents $\overset{\text{gen}}{\vdash}$ and $\overset{\text{inst}}{\vdash}$ do not depend on the exact structure of type expressions. They express generic operations that are often used in natural semantics. The former is used whenever a term containing variables has to be generalized into a quantified term. The latter is used to obtain generic instances of a quantified term. As for operations on sets of assumptions, we believe that they have to be considered as *predefined* sequents in the TYPOL language.

Static Semantics of Mini-ML

Now we are ready to specify the typechecker of our mini-ML language. We begin with a few remarks about extensions to the simple λ-calculus.

To include *products* into expressions we must introduce *patterns* in place of an identifier. For example, we may now define $\lambda(x,y).x+y$. Although the typechecking is not modified by this new construct, we must check that patterns are well formed, i.e. that they do not contain the same identifier twice. An expression such that $\lambda(x,x).x+x$ is not valid. This can be viewed as a declaration rule. The set MAKE_ENV builds a local environment that contains assumptions about identifiers in a pattern. The whole list of assumptions is the union of all these local lists of assumptions.

Recursion is included by adding the polymorphic fixed-point operator

$$\text{fix} : \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha$$

and the conditional expression *if* is introduced as usual in programming languages.

Finally consider the following example of optimisation. It is well known that the *let* construct is syntactic sugar for $\lambda x.e'\ e$. But we have seen how this construct allows the typing of polymorphic functions, such as let $i = \lambda x.x$ in $i\ i$. Of course we would like to be able to typecheck its equivalent form $(\lambda i.i\ i)\ \lambda x.x$. To do this we have only to add a more specialized form of the APP$'$ rule.

The static semantics of mini-ML is described by the following TYPOL program:

program L_TC is

use L

p : *PATTERN*;

A, A' : *ENV*;

τ, τ' : *TYPES*;

σ : *TYPE_SCHEMES*;

set TYPE is

$$A \vdash \textbf{true} : \textit{bool}$$

$$A \vdash \textbf{false} : \textit{bool}$$

$$A \vdash \textbf{number}\ \mathrm{N} : \textit{int}$$

$$\frac{A \overset{\text{typeof}}{\vdash} \textbf{ident}\ \mathrm{X} : \sigma \qquad \overset{\text{inst}}{\vdash} \sigma : \tau}{A \vdash \textbf{ident}\ \mathrm{X} : \tau}$$

$$\frac{[]\ \overset{\text{make\_env}}{\vdash}\ \mathrm{P}, \tau' : A' \qquad A \overset{\text{union}}{\vdash} A' : A'' A'' \vdash \mathrm{E} : \tau}{A \vdash \lambda \mathrm{P.E} : \tau' \rightarrow \tau}$$

$$\frac{A \vdash \mathrm{E} : \tau \qquad A \overset{\text{gen}}{\vdash} \tau : \sigma \qquad []\ \overset{\text{make\_env}}{\vdash}\ \mathrm{P}, \sigma : A' \qquad A \overset{\text{union}}{\vdash} A' : A'' \qquad A'' \vdash \mathrm{E}' : \tau'}{A \vdash (\lambda \mathrm{P.E}')\mathrm{E} : \tau'}$$

$$\frac{A \vdash \mathrm{E} : \tau' \rightarrow \tau \qquad A \vdash \mathrm{E}' : \tau'}{A \vdash \mathrm{E}\ \mathrm{E}' : \tau}$$

$$\frac{A \vdash (\lambda \mathrm{P.E}')\mathrm{E} : \tau}{A \vdash \textbf{let}\ \mathrm{P} = \mathrm{E}\ \textbf{in}\ \mathrm{E}' : \tau}$$

$$\frac{A \vdash (\lambda \mathrm{P}.\mathrm{E}')(\mathit{fix}\ \lambda \mathrm{P}.\mathrm{E}) : \tau}{A \vdash \text{letrec}\ \mathrm{P} = \mathrm{E}\ \text{in}\ \mathrm{E}' : \tau}$$

$$\frac{A \vdash \mathrm{E} : \tau \rightarrow \tau}{A \vdash (\mathit{fix}\ \mathrm{E}) : \tau}$$

$$\frac{A \vdash \mathrm{E} : \mathit{bool} \qquad A \vdash \mathrm{E}' : \tau \qquad A \vdash \mathrm{E}'' : \tau}{A \vdash \text{if}\ \mathrm{E}\ \text{then}\ \mathrm{E}'\ \text{else}\ \mathrm{E}'' : \tau}$$

$$\frac{A \vdash \mathrm{E} : \tau \qquad A \vdash \mathrm{E}' : \tau'}{A \vdash (\mathrm{E}, \mathrm{E}') : \tau \times \tau'}$$

end TYPE;

3.2. Dynamic semantics of Mini-ML

As noticed before, functions in ML can be manipulated as any other object in the language. For example a function may be the parameter of another function: it is possible to define higher-order functions. Thus the semantic domain of ML values is slightly more complicated than for a less expressive language.

Mini-ML values

- integers: N
- boolean values : *true*, *false*
- closures: $[\![\lambda \mathrm{P}.\mathrm{E}, \rho]\!]$, where E is an expression and ρ is an environment. A closure is just a pair of a λ-expression and an environment.
- identifiers for predefined operators: *plus*, ...
- pairs of semantic value: (α, β) (which may in turn be pairs, so lists of semantic values may be constructed)

As expected the semantic value of an expression e depends upon the values of the identifiers occurring free within it. If ρ stands for a list of mappings $x \mapsto \alpha$ from identifiers to values, we shall say that the expression e has value α if the theorem

$$\rho \vdash e : \alpha$$

may be derived from the following formal system:

program L_DS **is**

use L

$\rho, \rho_1 : \mathit{ENV};$

$\alpha, \beta, \gamma : \mathit{VALUE};$

$$\frac{\mathit{init\_env} \vdash \text{L\_EXP} : \alpha}{\vdash \text{L\_EXP} : \alpha} \qquad (1)$$

$$\rho \vdash \text{number}\ \mathrm{N} : \mathrm{N} \qquad (2)$$

$$\rho \vdash \text{true} : \mathit{true} \qquad (3)$$

$$\rho \vdash \text{false} : \mathit{false} \qquad (4)$$

$$\frac{\rho \overset{\text{val\_of}}{\vdash} \text{ident}\ \mathrm{I} \mapsto \alpha}{\rho \vdash \text{ident}\ \mathrm{I} : \alpha} \qquad (5)$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathit{true} \qquad \rho \vdash \mathrm{E}_2 : \alpha}{\rho \vdash \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3 : \alpha} \qquad (6)$$

$$\frac{\rho \vdash \mathrm{E}_1 : \mathit{false} \qquad \rho \vdash \mathrm{E}_3 : \alpha}{\rho \vdash \text{if}\ \mathrm{E}_1\ \text{then}\ \mathrm{E}_2\ \text{else}\ \mathrm{E}_3 : \alpha} \qquad (7)$$

$$\frac{\rho \vdash \text{E}_1 : \alpha \qquad \rho \vdash \text{E}_2 : \beta}{\rho \vdash (\text{E}_1, \text{E}_2) : (\alpha, \beta)} \tag{8}$$

$$\frac{\rho \vdash \text{E}_1 : \alpha \qquad \text{P} \mapsto \alpha \cdot \rho \vdash \text{E}_2 : \beta}{\rho \vdash \text{let P} = \text{E}_1 \,\text{in}\, \text{E}_2 : \beta} \tag{9}$$

$$\frac{\rho_1 = \text{P} \mapsto [\![\text{E}_1, \rho_1]\!] \cdot \rho \qquad \rho_1 \vdash \text{E}_2 : \beta}{\rho \vdash \text{letrec P} = \text{E}_1 \,\text{in}\, \text{E}_2 : \beta} \tag{10}$$

$$\rho \vdash \lambda\text{P.E} : [\![\lambda\text{P.E}, \rho]\!] \tag{11}$$

$$\frac{\rho \vdash \text{E}_1 : \alpha \qquad \text{P} \mapsto \alpha \cdot \rho \vdash \text{E}_2 : \beta}{\rho \vdash \lambda\text{P.E}_2\, \text{E}_1 : \beta} \tag{12}$$

$$\frac{\rho \vdash \text{E}_1 : \text{ident OP} \qquad \rho \vdash \text{E}_2 : (\alpha, \beta) \qquad \overset{\text{eval}}{\vdash} \text{OP}, \alpha, \beta : \gamma}{\rho \vdash \text{E}_1\, \text{E}_2 : \gamma} \tag{13}$$

$$\frac{\rho \vdash \text{E}_2 : \alpha \qquad \rho \vdash \text{E}_1 : [\![\lambda\text{P.E}, \rho_1]\!] \qquad \text{P} \mapsto \alpha \cdot \rho_1 \vdash \text{E} : \beta}{\rho \vdash \text{E}_1\, \text{E}_2 : \beta} \tag{14}$$

set VAL_OF is

$$\text{ident I} \mapsto \alpha \cdot \rho \vdash \text{ident I} \mapsto \alpha \tag{1}$$

$$\frac{\text{P}_2 \mapsto \beta \cdot \text{P}_1 \mapsto \alpha \vdash \text{ident I} \mapsto \gamma}{(\text{P}_1, \text{P}_2) \mapsto (\alpha, \beta) \cdot \rho \vdash \text{ident I} \mapsto \gamma} \tag{2}$$

$$\frac{\text{P}_2 \mapsto [\![\text{E}_2, \rho_1]\!] \cdot \text{P}_1 \mapsto [\![\text{E}_1, \rho_1]\!] \vdash \text{ident I} \mapsto \gamma}{(\text{P}_1, \text{P}_2) \mapsto [\![(\text{E}_1, \text{E}_2), \rho_1]\!] \cdot \rho \vdash \text{ident I} \mapsto \gamma} \tag{3}$$

$$\frac{\rho \vdash \text{ident I} \mapsto \alpha}{\rho_1 \cdot \rho \vdash \text{ident I} \mapsto \alpha} \tag{4}$$

end VAL_OF;

end L_DS

Comments on the formal definition:

Rule 1 just says that the evaluation of an expression begins with an initial environment. This environment is not given here. It simply says that the semantic value of the identifier "plus" is *plus* (and likewise for all predefined operators). This enables us to treat in the same way the two expressions $plus(2,3)$ and let $x = plus$ in $x(2,3)$ in rule (13).

Rules 2, 3, 4, 5, and 11 are the axioms of the system. They give semantic values to "simple" expressions. The rule 5 is subject to a condition: the environment ρ must associate a value to the identifier I.

The addition of new mapping $x \mapsto \alpha$ in the environment is performed directly in the rule, for identifiers as well as for pairs (see the let rule (9) for example), while searching for the value corresponding to an identifier is made in a separate set of rules (VAL_OF) (see rule 5). Let us examine this Typol set in greater detail.

In the VAL_OF set, rules 1 and 4 describe how to find the value of an identifier (notice that they are equivalent to the rules 1 and 2 of the TYPEOF set used in type checking). But in rule 9 of L_DS we have allowed the addition of a mapping of pairs (lists) in ρ. So we have to say something like "a mapping of a pair is a pair of mappings". That is the purpose of rule 2.

Rule 10 of L_DS uses the same sort of idea: we add in ρ a mapping of a pair of identifiers to one single closure, which is a closure of a pair of λ-expressions. Rule 3 says that "closure of a pair is a pair of closures". This simple idea, together with the addition of mappings directly in the rule, and of course the possibility of defining a graph in the environment enables us to give a very compact rule for the letrec. We go back now to the main Typol set.

The letrec rule (10) describes in an uniform way the "simple" recursive fonctions and the mutually recursive one:

$$\begin{aligned}\text{letrec}(f,(g,h)) = (&\lambda x.\cdots f\cdots g\cdots h\cdots,\\ &(\lambda y.\cdots f\cdots g\cdots h\cdots,\\ &\ \lambda z.\cdots f\cdots g\cdots h\cdots))\end{aligned}$$

Rule 14 shows how to evaluate an application, when the operator of the application, E_1, is a function.

Rule 13 is a special case of rule 14, in the case where E_1 evaluates to a predefined operator.

Rule 12 is another special case which expresses the fact that $\lambda P.E_2\ E_1 = (\text{let}\, P = E_1 \,\text{in}\, E_2)$. This is to be considered as an optimisation of the apply rule 14. Remark that this semantic definition specifies a call-by-value mechanism.

3.3. Dynamic semantics of CAM

The Categorical Abstract Machine, [CAM 85] has its roots both in categories and in the De Bruijn's notation for lambda calculus. It is a very simple machine where categorical terms can be considered as code acting on a stack of values. Actions are essentially cons, car, push, ..., as well as a LISP's rplac to implement recursion. To handle abstractions, values may be closures.

Abstract Syntax

The abstract syntax of CAM is given by the following specifications:

sorts

VALUE, COM, PROGRAM, COMS

subsorts

COM≻

COMS

functions

'program'

program	:	COMS	→	PROGRAM
coms	:	COM*	→	COMS

'com'

quote	:	VALUE	→	COM
op	:		→	COM
car	:		→	COM
cdr	:		→	COM
cons	:		→	COM
push	:		→	COM
swap	:		→	COM
app	:		→	COM
rplac	:		→	COM
cur	:	COMS	→	COM
branch	:	COMS×COMS	→	COM

'value'

int	:	→	VALUE
bool	:	→	VALUE
null_value	:	→	VALUE

The formal semantics of CAM:

The state of the CAM machine is a stack, whose top element may be thought of as a register. Elements of this stack are semantic values :

- integers $\mathbb{N}$
- truth values: *true*, *false*
- closures of the form $[\![\mathrm{C}, \rho]\!]$, where C is a list of CAM-commands and ρ is a semantic value
- pairs of semantic values (which may in turn be pairs, so that lists may be constructed)

Except in the first rule, all sequents have the form

$$s \vdash c : s'$$

where c is CAM-code and s and s' are states of the CAM machine. The sequent $s \vdash c : s'$ may be read as "executing code c when the machine is in state s takes it to final state s'".

program CAM_DS **is**

use CAM

$s, s_1, s_2 : STACK;$

$\alpha, \beta, \gamma, v : VALUE;$

$\rho, \rho_1 : ENV;$

$$\frac{init\_stack \vdash \mathrm{COMS} : v \cdot \mathrm{s}}{\vdash program(\mathrm{COMS}) : v} \tag{1}$$

$$s \vdash \emptyset : s \tag{2}$$

$$\frac{s \vdash \mathrm{COM} : s_1 \qquad s_1 \vdash \mathrm{COMS} : s_2}{s \vdash \mathrm{COM; COMS} : s_2} \tag{3}$$

$$\alpha \cdot \mathrm{s} \vdash quote(\mathrm{x}) : \mathrm{x} \cdot \mathrm{s} \qquad (\,var(\mathrm{x})\,) \tag{4}$$

$$\alpha \cdot \mathrm{s} \vdash quote(int\,\mathrm{N}) : \mathrm{N} \cdot \mathrm{s} \tag{5}$$

$$\alpha \cdot \mathrm{s} \vdash quote(bool\,\mathrm{T}) : \mathrm{T} \cdot \mathrm{s} \tag{6}$$

$$(\alpha, \beta) \cdot \mathrm{s} \vdash car : \alpha \cdot \mathrm{s} \tag{7}$$

$$(\alpha, \beta) \cdot \mathrm{s} \vdash cdr : \beta \cdot \mathrm{s} \tag{8}$$

$$\alpha \cdot \beta \cdot \mathrm{s} \vdash cons : (\beta, \alpha) \cdot \mathrm{s} \tag{9}$$

$$\frac{\overset{\text{eval}}{\vdash} \mathrm{OP}, \alpha, \beta : \gamma}{(\alpha, \beta) \cdot \mathrm{s} \vdash op\,\mathrm{OP} : \gamma \cdot \mathrm{s}} \tag{10}$$

$$\alpha \cdot \mathrm{s} \vdash push : \alpha \cdot \alpha \cdot \mathrm{s} \tag{11}$$

$$\alpha \cdot \beta \cdot \mathrm{s} \vdash swap : \beta \cdot \alpha \cdot \mathrm{s} \tag{12}$$

$$\rho \cdot \mathrm{s} \vdash cur(\mathrm{C}) : [\![\mathrm{C}, \rho]\!] \cdot \mathrm{s} \tag{13}$$

$$\frac{(\rho, \alpha) \cdot \mathrm{s} \vdash \mathrm{C} : s}{([\![\mathrm{C}, \rho]\!], \alpha) \cdot \mathrm{s} \vdash \mathrm{app} : s} \tag{14}$$

$$\frac{\mathrm{V} = \rho_1}{(\rho, \mathrm{V}) \cdot \rho_1 \cdot \mathrm{s} \vdash rplac : (\rho, \rho_1) \cdot \mathrm{s}} \tag{15}$$

$$\frac{\text{S} \vdash \text{C}_1 : s_1}{\text{“true”} \cdot \text{S} \vdash \mathit{branch}(\text{C}_1, \text{C}_2) : s_1} \tag{16}$$

$$\frac{\text{S} \vdash \text{C}_2 : s_1}{\text{“false”} \cdot \text{S} \vdash \mathit{branch}(\text{C}_1, \text{C}_2) : s_1} \tag{17}$$

end CAM_DS

Pattern maching suffices to explain simple instructions : *quote*(V), *car*, *cdr*, *cons*, *push* and *swap*. The *op* instruction performs additions etc...

Rule 2 and 3 deal with sequences of commands. Rule 1 says that the evaluation of a program begins with an initial stack and ends with a value on top of the stack which is the semantic value of the program. The initial stack is not given here; it contains the list of closures corresponding to the predefined operators: for example, $[\![$*cdr*; *op* "plus" , *0*$]\!]$ is the closure corresponding to the operator "plus".

Rule 16 and 17 shows that the CAM contains a *branch* instruction which takes its (evaluated) condition on top of the stack. The *cur* instruction is described in Rule 13 : *cur*(C) builds a closure with the code C and the current environment (top of the stack) and puts it on top of the stack. Rule 14 says that the *app* instruction takes a closure and an environment on top of the stack, and executes the code of the closure in an environment containing the environment of the closure prefixing the current one.

The remaining rule, 15, is the less intuitive one. An *rplac* instruction takes a pair consisting of an environment ρ and a variable V, followed by an environment ρ_1 on the stack. It identifies V and ρ_1 and returns the pair (ρ, ρ_1) on the stack. Notice that each (possibly) occurrence of V in ρ_1 has been replaced by ρ_1. The use of this instruction will be explained by the translation of the *letrec* instruction [see rule 9 of L_CAM]. We give two alternatives to the *rplac* instruction : the *recf* instruction, building a graph, used in [Kahn et al. 85] (but it doesn't treat the case of mutually recursive functions), and the use of fixed point in the environment (see Section 3.8).

3.4. Translation from mini-ML to CAM

Again, except for rule 1, all the sequents have the form :

$$\rho \vdash e \rightarrow c$$

where ρ is an environment, e an L_expression and c is it's translation. So, the sequent may be read as "given a context ρ, expression e is compiled into code c"

program L_CAM is

use L

use *CAM*

$c, c_1, c_2, c_3 : CAM;$

$\rho, \rho_1 : ENV;$

$$\frac{\mathit{init\_pat} \vdash \text{L\_EXP} \rightarrow c}{\vdash \text{L\_EXP} \rightarrow \mathit{program}(c)} \tag{1}$$

$$\rho \vdash \text{number N} \rightarrow \mathit{quote}(\mathit{int}\ \text{N}) \tag{2}$$

$$\rho \vdash \text{true} \rightarrow \mathit{quote}(\mathit{bool}\ \text{“true”}) \tag{3}$$

$$\rho \vdash \text{false} \rightarrow \mathit{quote}(\mathit{bool}\ \text{“false”}) \tag{4}$$

$$\frac{\rho \overset{\text{access}}{\vdash} \text{ident I} : c}{\rho \vdash \text{ident I} \rightarrow c} \tag{5}$$

$$\frac{\rho \vdash \text{E}_1 \rightarrow c_1 \qquad \rho \vdash \text{E}_2 \rightarrow c_2 \qquad \rho \vdash \text{E}_3 \rightarrow c_3}{\rho \vdash \text{if } \text{E}_1 \text{ then } \text{E}_2 \text{ else } \text{E}_3 \rightarrow push; c_1; branch(c_2, c_3)} \tag{6}$$

$$\frac{\rho \vdash \text{E}_1 \rightarrow c_1 \qquad \rho \vdash \text{E}_2 \rightarrow c_2}{\rho \vdash (\text{E}_1, \text{E}_2) \rightarrow push; c_1; swap; c_2; cons} \tag{7}$$

$$\frac{\rho \vdash \text{E}_1 \rightarrow c_1 \qquad (\rho, \text{P}) \vdash \text{E}_2 \rightarrow c_2}{\rho \vdash \text{let } \text{P} = \text{E}_1 \text{ in } \text{E}_2 \rightarrow push; c_1; cons; c_2} \tag{8}$$

$$\frac{(\rho, \text{P}) \vdash \text{E}_1 \rightarrow c_1 \qquad (\rho, \text{P}) \vdash \text{E}_2 \rightarrow c_2}{\rho \vdash \text{letrec } \text{P} = \text{E}_1 \text{ in } \text{E}_2 \rightarrow push; quote(\rho_1); cons; push; c_1; swap; rplac; c_2} \tag{9}$$

$$\frac{(\rho, \text{P}) \vdash \text{E} \rightarrow c}{\rho \vdash \lambda \text{P.E} \rightarrow cur(c)} \tag{10}$$

$$\frac{\rho \vdash \text{E}_2 \rightarrow c_2 \qquad \overset{\text{trans\_const}}{\vdash} \quad \text{E}_1 \rightarrow c_1}{\rho \vdash \text{E}_1 \ \text{E}_2 \rightarrow c_2; c_1} \quad (\ is\_const(\text{E}_1)\ ) \tag{11}$$

$$\frac{\rho \vdash \text{E}_1 \rightarrow c_1 \qquad (\rho, \text{P}) \vdash \text{E}_2 \rightarrow c_2}{\rho \vdash \lambda \text{P.E}_2 \ \text{E}_1 \rightarrow push; c_1; cons; c_2} \tag{12}$$

$$\frac{\rho \vdash \text{E}_1 \rightarrow c_1 \qquad \rho \vdash \text{E}_2 \rightarrow c_2}{\rho \vdash \text{E}_1 \ \text{E}_2 \rightarrow push; c_1; swap; c_2; cons; app} \tag{13}$$

set ACCESS is

$$\text{ident } \text{X} \vdash \text{ident } \text{X} : \emptyset \tag{1}$$

$$\frac{\rho_2 \vdash \text{X} : c_2}{(\rho_1, \rho_2) \vdash \text{X} : cdr; c_2} \tag{2}$$

$$\frac{\rho_1 \vdash \text{X} : c_1}{(\rho_1, \rho_2) \vdash \text{X} : car; c_1} \tag{3}$$

end ACCESS;

set IS_CONST is

ident "plus"   ident "minus"   ident "times"

ident "equal"   ident "fst"   ident "snd"

end IS_CONST;

set TRANS_CONST is

$$\vdash \text{ident "fst"} \rightarrow car \tag{1}$$

$$\vdash \text{ident "snd"} \rightarrow cdr \tag{2}$$

$$\vdash \text{ident } \text{X} \rightarrow op\, \text{X} \tag{3}$$

end TRANS_CONST;

end L_CAM

The environment use here is a list of L_identifiers (more exactly a tree) : ((P,Q),R). The initial environment is ((..(_,ident "plus"),ident "minus")..). The use of environment is illustrated by rule 5 and the set ACCESS, which says that in a given environment ρ, an identifier is translated to a sequence of *car* and *cdr* which realises the access to this identifier in ρ and will realise the access to the corresponding value in the stack of the CAM.

Rules 2, 3 and 4 are straightforward, except they say that a result value is put in place of (and not on) the top of the stack in CAM. It is the reason why almost all translations begin with *push*. The translation of a pair (rule 7), for example, makes this point clear: the idea is that all CAM instructions will use an environment on top of the stack and replace it by the final result.

Changing of environment of translation is made each time there is a declaration : see rules 8 and 9 for *let* and *letrec*.

The translation of a λ-expression uses naturally the *cur* instruction.

There are three rules for translating the *apply*, as it was the case in the dynamic semantics of L (rules 11, 12 & 13).

Finally, rule 9 is the less intuitive one:

$$\frac{(\rho,\text{P}) \vdash \text{E}_1 \rightarrow c_1 \qquad (\rho,\text{P}) \vdash \text{E}_2 \rightarrow c_2}{\rho \vdash \text{letrec P} = \text{E}_1 \text{ in E}_2 \rightarrow push; quote(\rho_1); cons; push; c_1; swap; rplac; c_2}$$

ρ_1 is a free (new) variable of this rule. $quote(\rho_1)$ will result (in the CAM execution) in putting a free variable on top of the stack. After a *cons* and a *push*, the top of the stack has the form (ρ,ρ_1). Now, E_1 is a λ-exp. or a list of λ-exp. So c_1, its translation, will result in building a closure (or a list of closure) using the current environment on the top of the stack: (ρ,ρ_1). So this closure will contains "holes" of the form of a variable (ρ_1). *rplac* will feed holes with the desired graph, in saying that the variable ρ_1 has the value of the closure itself.

3.5. Equivalent semantics of mini-ML

We present here alternative semantics of L, where we use fix-points in the environment, instead of graphs.

Changes in *L_DS* concern the letrec rule and the set val_of:

The letrec rule:

$$\frac{\rho_1 = \text{P} \mapsto [\![\text{E}_1,\rho_1]\!] \cdot \rho \qquad \rho_1 \vdash \text{E}_2 : \beta}{\rho \vdash \text{letrec P} = \text{E}_1 \text{ in E}_2 : \beta}$$

is changed to

$$\frac{(\mathit{fix}\ \lambda x.\text{P} \mapsto [\![\text{E}_1, x]\!] \cdot r) \vdash \text{E}_2 : \beta}{r \vdash \text{letrec P} = \text{E}_1 \text{ in E}_2 : \beta}$$

We need one more rule in the set val_of, for explicitly enrooling the fix-point operator:

$$\frac{r \vdash \text{ident I} \mapsto \alpha \qquad \alpha \overset{\text{subst}}{\vdash} x, (\mathit{fix}\ \lambda x.r) : \alpha'}{(\mathit{fix}\ \lambda x.r) \cdot r_1 \vdash \text{ident I} \mapsto \alpha'}$$

where $\alpha \overset{\text{subst}}{\vdash} x, e : \alpha'$ simply means that $\alpha' = \alpha[e\backslash x]$.

4. Inheritance by Type Inclusion: Mini-Amber

New developments in programming language design have shown that the notion of type inclusion is useful. In all generality type inclusion theory seems to provide a uniform view of type systems that includes abstract data types, parametric polymorphism and multiple inheritance, [CW].

The type system retained in the current version of the Amber programming language [Amber] uses type inclusion to provide multiple inheritance combined with higher-order functions. We consider in the sequel only the kernel of this type system, i.e. the monomorphic typed λ-calculus extended with unordered cartesian products, called *records*, and unordered disjoint sums, called *variants.*

4.1. Overview of the Language

A complete description of Amber may be found in [Amber], but we would like to describe briefly both types and expressions we have retained for our Typol specifications. The abstract syntax of our Mini-Amber is given by the following:

Abstract Syntax of Mini-Amber

sorts

IDENT, EXP, FIELD, SELECTORS, SELECT, TYPE, COLON

subsorts

EXP>

IDENT

functions

'Types'

arrow	:	TYPE×TYPE	→	TYPE
variant_type	:	COLON*	→	TYPE
record_type	:	COLON*	→	TYPE
colon	:	IDENT×TYPE	→	COLON
int	:		→	TYPE
bool	:		→	TYPE

'Expressions'

apply	:	EXP×EXP	→	EXP
dot	:	EXP×IDENT	→	EXP
case	:	EXP×SELECTORS	→	EXP
cases	:	SELECT*	→	SELECTORS
select	:	IDENT×EXP	→	SELECT
fun	:	IDENT×TYPE×EXP	→	EXP
variant	:	IDENT×EXP	→	EXP
record	:	FIELD*	→	EXP
field	:	IDENT×EXP	→	FIELD
true	:		→	EXP
false	:		→	EXP
number	:		→	EXP

'Identifiers'

ident : → IDENT

For each kind of data type we give its corresponding constructor with some comments.

Ground Types

- *int* is the type of integer. Numerals are of that type.
- *bool* is the type of constants *true* and *false*.

Functional Types

- $\tau' \rightarrow \tau$ where τ' is the type of the function parameter. For example $\mathbf{fun}(x : int)x + 1$ defines the successor function which has type $int \rightarrow int$.

Records

- $\{a : int, b : bool\}$ is the type of the unordered, labeled sets of values $\{a = 0, b = true\}$, called a record. Access to a record field is provided by the notation $r.a$, which selects the a component of a record r.

Variants

- variant types are unordered, labeled set of types $[a : int, b : bool]$. A variant expression is a labeled value $[a = 0]$. Variants are used in the case statement:

$$\mathbf{case}\, e\, \mathbf{of}\, a_1 \Rightarrow f_1 \mid \cdots \mid a_n \Rightarrow f_n$$

where the f_i's are functions and the a_i are tags (this concrete syntax is slightly different of the syntax used in Amber). Although we are not concerned by evaluation let us remember the meaning of such a case statement: if the variant expression e has tag a_i then the function f_i is applied to the contents of e.

4.2. Type Inclusion

Type inclusion must be defined with respect to some ordering relation. In Amber the type inclusion is determined by the structure of type terms, and the ordering relation is that of subsets. A record type R $[a : int, b : bool]$ is considered as the set of all records with at least the two components $a : int$ and $b : bool$. Thus a record type with more fields, $[a : int, b : bool, c : int]$, is included, as a set, in R.

This type inclusion mechanism is described with the following Typol system:

$$\vdash \tau \preceq \tau \tag{1}$$

$$\frac{\vdash \sigma' \preceq \sigma \qquad \vdash \tau \preceq \tau'}{\vdash \sigma \rightarrow \tau \preceq \sigma' \rightarrow \tau'} \tag{2}$$

$$\frac{\vdash \tau' \preceq \tau}{\vdash \{\text{X} : \tau', \text{RECORD}\} \preceq \{\text{X} : \tau\}} \tag{3}$$

$$\frac{\vdash \text{RECORD} \preceq \{\text{X} : \tau\}}{\vdash \{\text{Y} : \sigma, \text{RECORD}\} \preceq \{\text{X} : \tau\}} \quad (\text{Y} \neq \text{X}) \tag{4}$$

$$\frac{\vdash \{\text{COLON}, \text{RECORD}'\} \preceq \{\text{X} : \tau\} \qquad \vdash \{\text{COLON}, \text{RECORD}'\} \preceq \{\text{X}' : \tau', \text{RECORD}\}}{\vdash \{\text{COLON}, \text{RECORD}'\} \preceq \{\text{X} : \tau, \text{X}' : \tau', \text{RECORD}\}} \tag{5}$$

$$\frac{\vdash \tau' \preceq \tau}{\vdash [\text{X} : \tau'] \preceq [\text{X} : \tau, \text{VARIANT}]} \tag{6}$$

$$\frac{\vdash [\text{X} : \tau'] \preceq \text{VARIANT}}{\vdash [\text{X} : \tau'] \preceq [\text{Y} : \tau, \text{VARIANT}]} \quad (\text{Y} \neq \text{X}) \tag{7}$$

$$\frac{\vdash [\text{X} : \tau] \preceq [\text{COLON}, \text{VARIANT}'] \qquad \vdash [\text{X}' : \tau', \text{VARIANT}] \preceq [\text{COLON}, \text{VARIANT}']}{\vdash [\text{X} : \tau, \text{X}' : \tau', \text{VARIANT}] \preceq [\text{COLON}, \text{VARIANT}']} \tag{8}$$

Rule 1 expresses that every type is included in itself.

Rule 2 expresses that for functions the domain shrinks, while the codomain expands.

Rules 3, 4, and 5 express type inclusion on records. A record is included in a record type with only one field provided that the label matches and the respective field types are included (Rules 3 and 4). Next to prove that a record type with more fields is included in a record type with fewer fields we prove that the former is included in all the one field records of the latter (Rule 5).

In a similar manner, rules 6, 7, and 8 express that a variant type with fewer fields is included in a variant type with more fields, provided that the labels match and the respective fields types are included.

Consider the following example due to L. Cardelli:

type $Point = \{x : int, y : int\}$

type $Frame = \{hor : int, ver : int\}$

type $Tile = \{x : int, y : int, hor : int, ver : int\}$ The two sequents

$$\vdash \{x : int, y : int, hor : int, ver : int\} \preceq \{x : int, y : int\}$$

and

$$\vdash \{x : int, y : int, hor : int, ver : int\} \preceq \{hor : int, ver : int\}$$

may be derived from the type inclusion system. Tile is a subtype of both Point and Frame, i.e. there is multiple inheritance.

4.3. Inheritance

Now the typechecking of our Mini-Amber may be specified by the following Typol program, where A stands for a list of assumptions $x : \tau$.

program AMBER_TC **is**

use AMBER

$\tau, \tau', \sigma, \sigma' : TYPE$;

set TYPE **is**

$$A \vdash \text{true} : \text{bool} \tag{1}$$

$$A \vdash \text{false} : \text{bool} \tag{2}$$

$$A \vdash \text{number}\,\textsc{n} : \text{int} \tag{3}$$

$$\frac{A \overset{\text{typeof}}{\vdash} \text{ident}\,\textsc{x} : \tau}{A \vdash \text{ident}\,\textsc{x} : \tau} \tag{4}$$

$$\frac{\vdash \textsc{type} : \tau \qquad A \overset{\text{declare}}{\vdash} \textsc{ident}, \tau : A_1 \qquad A_1 \vdash \textsc{exp} : \tau'}{A \vdash \text{fun}(\textsc{ident} : \textsc{type})\textsc{exp} : \tau \rightarrow \tau'} \tag{5}$$

$$\frac{A \vdash \textsc{exp} : \sigma \rightarrow \tau \qquad A \vdash \textsc{exp}' : \tau' \qquad \vdash \tau' \preceq \sigma}{A \vdash \textsc{exp}\;\textsc{exp}' : \tau} \tag{6}$$

$$\frac{A \vdash \textsc{exp} : \tau}{A \vdash \{\textsc{lab} = \textsc{exp}\} : \{\textsc{lab} : \tau\}} \tag{7}$$

$$\frac{A \vdash \textsc{exp} : \tau \qquad A \vdash \textsc{record} : \sigma \qquad \sigma \overset{\text{union}}{\vdash} \textsc{lab}, \tau : \sigma'}{A \vdash \{\textsc{lab} = \textsc{exp}, \textsc{record}\} : \sigma'} \tag{8}$$

$$\frac{A \vdash \text{EXP} : \sigma \qquad \vdash \sigma \preceq [\text{X} : \tau]}{A \vdash \text{EXP.X} : \tau} \tag{9}$$

$$\frac{A \vdash \text{EXP} : \tau}{A \vdash [\text{LAB} = \text{EXP}] : [\text{LAB} : \tau]} \tag{10}$$

$$\frac{A \vdash \text{EXP} : \sigma \qquad A \vdash \text{SELECTORS} : \sigma' \rightarrow \tau \qquad \vdash \sigma \preceq \sigma'}{A \vdash \textbf{case}\ \text{EXP}\ \textbf{of}\ \text{SELECTORS} : \tau} \tag{11}$$

$$\frac{A \vdash \text{EXP} : \tau' \rightarrow \tau}{A \vdash \text{LAB} \Rightarrow \text{EXP} : [\text{LAB} : \tau'] \rightarrow \tau} \tag{12}$$

$$\frac{A \vdash \text{EXP} : \tau' \rightarrow \tau \qquad A \vdash \text{SELECTORS} : \sigma \rightarrow \tau \qquad \sigma \overset{\text{union}}{\vdash} \text{LAB}, \tau' : \sigma'}{A \vdash \text{LAB} \Rightarrow \text{EXP} \mid \text{SELECTORS} : \sigma' \rightarrow \tau} \tag{13}$$

end TYPE;

Rules 1, 2, 3,and 4 are the axioms of the system. The rule 4 expresses the fact that the type of an identifier has to be recorded in the list of assumptions A.

In rule 5 we have to check that the type expression TYPE of the parameter IDENT is a well formed type term. For example a record type with the same label twice is not allowed. This is done by the sequent $\vdash$ TYPE : τ.

Rule 6 is a very interesting one. It asserts that the argument of a function of type $\sigma \rightarrow \tau$ must have a type τ' which is a subtype of the type σ of the domain of the function. Thus the function

$\text{fun}(p : \{x : int, y : int\})\ p.x$ can be used with an argument of type Tile:

$$(\text{fun}(p : \{x : int, y : int\})\ p.x)\{x = 0, y = 0, hor = 1, ver = 1\}$$

Rules 7 and 8 express that the type of a record expression is a record type. As for function parameter the record expression must have a valid record type. This is done by the Typol set UNION (that implements an exclusive union).

Rule 9 expresses the selection of a record field by the type inclusion mechanism.

A variant expression has a variant type (Rule 10).

Rules 11, 12, and 13 handle the case expression. The type of the list of cases, SELECTORS, is expressed as a function $\sigma' \rightarrow \tau$, where σ' is the union of variant types $[a_i : \tau_i]$, with a_i the label of a function f_i of type $\tau_i \rightarrow \tau$. The type σ of the expression EXP must be a subtype of that variant type σ'.

The following Typol rules insure that the type declaration of the parameter of a function is a well formed type term (Notice that the $\vdash$ operator is overloaded).

$$\vdash \text{bool} : \text{bool} \tag{1}$$

$$\vdash \text{int} : \text{int} \tag{2}$$

$$\frac{\vdash \text{TYPE} : \tau \qquad \vdash \text{TYPE}' : \tau'}{\vdash \text{TYPE} \rightarrow \text{TYPE}' : \tau \rightarrow \tau'} \tag{3}$$

$$\frac{\{\} \vdash \{\text{FIELD}, \text{RECORD}\} : \tau}{\vdash \{\text{FIELD}, \text{RECORD}\} : \tau} \tag{4}$$

$$\frac{\vdash \text{TYPE} : \tau \qquad \sigma \overset{\text{union}}{\vdash} \text{LAB}, \tau : \sigma'}{\sigma \vdash \{\text{LAB} : \text{TYPE}\} : \sigma'} \tag{5}$$

$$\frac{\vdash \text{TYPE} : \tau \qquad \sigma \overset{\text{union}}{\vdash} \text{LAB}, \tau : \sigma' \qquad \sigma' \vdash \text{RECORD} : \sigma''}{\sigma \vdash \{\text{LAB} : \text{TYPE}, \text{RECORD}\} : \sigma''} \tag{6}$$

$$\frac{\{\} \vdash [\text{COLON}, \text{VARIANT}] : \tau}{\vdash [\text{COLON}, \text{VARIANT}] : \tau} \tag{7}$$

$$\frac{\vdash \text{TYPE} : \tau \qquad \sigma \overset{\text{union}}{\vdash} \text{LAB}, \tau : \sigma'}{\sigma \vdash [\text{LAB} : \text{TYPE}] : \sigma'} \tag{8}$$

$$\frac{\vdash \text{TYPE} : \tau \qquad \sigma \overset{\text{union}}{\vdash} \text{LAB}, \tau : \sigma' \qquad \sigma' \vdash \text{VARIANT} : \sigma''}{\sigma \vdash [\text{LAB} : \text{TYPE}, \text{VARIANT}] : \sigma''} \tag{9}$$

Environment manipulations are implemented by the two Typol sets DECLARE and TYPEOF. Notice that the set DECLARE ensures exclusive union of mappings of identifiers and type expressions.

set DECLARE is

$$env[] \vdash \text{IDENT}, \tau : \text{IDENT} \mapsto \tau$$

$$\frac{\text{ENV} \vdash \text{IDENT}, \tau : e}{\text{IDENT}' \mapsto \tau' \cdot \text{ENV} \vdash \text{IDENT}, \tau : \text{IDENT}' \mapsto \tau' \cdot e} \qquad \text{IDENT}' \neq \text{IDENT}$$

end DECLARE ;

set TYPEOF is

$$\text{IDENT} \mapsto \tau \cdot \text{ENV} \vdash \text{IDENT} : \tau$$

$$\frac{\text{ENV} \vdash \text{IDENT} : \tau}{\text{IDENT}' \mapsto \tau' \cdot \text{ENV} \vdash \text{IDENT} : \tau} \qquad \text{IDENT}' \neq \text{IDENT}$$

end TYPEOF ;

The Typol set UNION used for record types and variant types implements also an exclusive union operation on elements. It is clear that the use of Typol sets to implement set union and set difference operations leads to a plethora of such "specialized" Typol sets. We investigate now what could be done with new Typol constructs A_x for set difference and $\cup$ for set union.

4.4. Improvements

Consider first rules 4 and 5 of the Typol set TYPE. They could be expressed as:

$$A \cup \{\text{ident } \text{X} : \tau\} \vdash \text{ident } \text{X} : \tau$$

where the set A contains only one assumption upon the identifier X because of the declaration rule:

$$\frac{\vdash \text{TYPE} : \tau \qquad A_x \cup \{\text{X} : \tau\} \vdash \text{EXP} : \tau'}{A \vdash \text{fun}(\text{X} : \text{TYPE})\text{EXP} : \tau \rightarrow \tau'} \tag{5}$$

Mini-Amber gives us the opportunity to take profit of the strong analogy existing between record types and environments (Both are unordered sets of mappings with the same kind of manipulation primitives). As remarked above rules 7 and 8 of the Typol system TYPE both express that the type of a record expression is a record type and that this record type is well formed, i.e. the same label identifier does not appear wice. That could be expressed with the following rule:

$$\frac{A \vdash \text{EXP} : \tau \qquad A \vdash \text{RECORD} : \sigma}{A \vdash \{\text{L} : \text{EXP}, \text{RECORD}\} : \sigma_l \cup \{\text{L} : \tau\}}$$

where σ_l stands for removing any assumption on the label identifier L. Of course such a rule would have to be compiled into at least two rules, one for a list of only one component and one for a list of more than one component. The same remark holds for rules 12 and 13 of that Typol set TYPE.

Next consider the Typol rules describing the sequent $\vdash$ TYPE : τ. Rules 5 and 6, (and in the same way rules 8 and 9) could be written as:

$$\frac{\vdash \text{TYPE} : \tau \qquad \sigma_l \cup \{\text{L} : \tau\} \vdash \text{RECORD} : \sigma'}{\sigma \vdash \{\text{L} : \text{TYPE}, \text{RECORD}\} : \sigma'}$$

Now in the type inclusion system we may use the fact that record types and variant types are unordered sets of components to write rules 3 and 4 as:

$$\frac{\vdash \tau' \preceq \tau}{\vdash \text{RECORD} \cup \{\text{L} : \tau'\} \preceq \{\text{L} : \tau\}}$$

5. Dynamic Semantics of Standard ML

We present a first version of a Typol specification of the core language of Standard ML. This implementation is based upon a private communication of R. Milner, [Milner], which describes the dynamic semantics of Core ML in a structural axiomatic style. We found this situation very useful: it gives us the opportunity to use Typol on a language currently in development.

As pointed out by Milner the Standard ML design is based upon simple and well understood ideas that have been experimented with in previous versions of ML or in other functional languages (Hope in particular). For example polymorphic references and assignment are omitted. On the other hand the ML *varstructs* are extended to include Hope *patterns* and exceptions are now more general. Exceptions may carry values of an arbitrary polymorphic type. Note that the present specification does not include input/output and separate compilation.

Outline: we found that, in the context of formal specifications, exceptions and pattern matching are the most difficult features of Standard ML to deal with. Although we have not yet solved them in a completely satisfactory manner we indicate what has been done to get a first executable specification. Because of exceptions, to every syntactic construct correspond two rules: one describing evaluations that "return" a value, and one describing what has to be done when a subexpression leads to an exception. We describe the mechanism retained in ML in the case of declarations, and point out that a "direct" compilation of these rules leads to an inefficient implementation. For pattern matching the problem is closely related to the notion of *failure* in an inference system. In a first approach this difficulty is solved by the use of a *diff* operator on identifiers. But we think that a better solution will be found.

Abstract Syntax Definitions

We present an abstract syntax which is very close to the syntax given by Milner. The principal syntax classes are defined in terms of the following primitive classes:

VAR for value variables.

CON for value constructors.

TYVAR for type variables.

TYCON for type constructors.

LAB for record labels.

EXN for exception names.

module ML_Syntax

sorts

EXN, LAB, TYCON, TYVAR, CON, VAR, EXP, MATCH, FIELD_EXP, HRULE, HANDLER, MRULE, PAT, FIELD_PAT, TY, FIELD_TYPE, TY_SEQ, EXCBIND, DATABIND, CONSTR, CONSTR_S, TYBIND, TYVAR_SEQ, VALBIND, DEC

subsorts

EXP>

VAR, CON

PAT>

CON, VAR

TY>

TYVAR

EXCBIND>

EXN

functions

'Declarations'

val	:	VALBIND	→	DEC
type	:	TYBIND	→	DEC
datatype	:	DATABIND	→	DEC
abstype	:	DATABIND×DEC	→	DEC
exception	:	EXCBIND	→	DEC
local	:	DEC×DEC	→	DEC
dec_seq	:	DEC$^+$	→	DEC

'Value Bindings'

simple_value	:	PAT×EXP	→	VALBIND
valbind_s	:	VALBIND$^+$	→	VALBIND
rec	:	VALBIND	→	VALBIND

'Type Bindings'

simple_type	:	TYVAR_SEQ×TYCON×TY	→	TYBIND
tybind_s	:	TYBIND$^+$	→	TYBIND
tyvar_seq	:	TYVAR*	→	TYVAR_SEQ

'Datatype Bindings'

rec_datatype	:	TYVAR_SEQ×TYCON×CONSTR_S	→	DATABIND
databind_s	:	DATABIND$^+$	→	DATABIND
constr_s	:	CONSTR$^+$	→	CONSTR_S
constr	:	CON×TY	→	CONSTR

'Exception Bindings'

simple_excbind	:	EXN×EXN	→	EXCBIND
excbind_s	:	EXCBIND$^+$	→	EXCBIND

'Types'

type_constr	:	TY_SEQ×TYCON	→	TY
function_type	:	TY×TY	→	TY
record_type	:	FIELD_TYPE$^+$	→	TY
field_type	:	LAB×TY	→	FIELD_TYPE
ty_seq	:	TY*	→	TY_SEQ

'Patterns'

record_pat	:	FIELD_PAT$^+$	→	PAT
construction	:	CON×PAT	→	PAT
layered	:	VAR×PAT	→	PAT
field_pat	:	LAB×PAT	→	FIELD_PAT
wildcard	:		→	PAT

'Expressions'

fun	:	MATCH	→	EXP
let	:	DEC×EXP	→	EXP
raise	:	EXN×EXP	→	EXP
handle	:	EXP×HANDLER	→	EXP
application	:	EXP×EXP	→	EXP
record_exp	:	FIELD_EXP$^+$	→	EXP
field_exp	:	LAB×EXP	→	FIELD_EXP
match	:	MRULE$^+$	→	MATCH
mrule	:	PAT×EXP	→	MRULE
handler	:	HRULE$^+$	→	HANDLER
with	:	EXN×MATCH	→	HRULE

trap	:	EXP	$\rightarrow$	HRULE

'Identifiers'

var	:	$\rightarrow$	VAR
con	:	$\rightarrow$	CON
number	:	$\rightarrow$	CON
tyvar	:	$\rightarrow$	TYVAR
tycon	:	$\rightarrow$	TYCON
lab	:	$\rightarrow$	LAB
exn	:	$\rightarrow$	EXN

Environments and Values

As for other functional languages, a function value is a partial function represented as a **closure**. A closure is a pair of a function body, i.e. a match, and of an environment. Thus we need to import the abstract syntax of ML to define the abstract syntax of values.

An **environment** has two components: a value environment and an exception environment. An exception is an object from which an exception identifier EXN may be recovered in the exception environment. A *packet* is a pair of an exception and a value.

A **store** has two components: a memory component that associates values to addresses, and a component that records exceptions. It is not clear now that such a component is necessary within a Typol specification.

module Value_Environments

sorts

VAL_PACK, VALENV_FAIL, ENV_PACK, FAIL, EXC, PACK, STORE, EXCS, MEM, EXCS_PAIR, MEM_PAIR, ENV, EXCENV, VALENV, EXCENV_PAIR, VALENV_PAIR, VAL, ADDR, FIELD_VAL

subsorts

VAL_PACK>

PACK, VAL

VALENV_FAIL>

FAIL, VALENV

ENV_PACK>

PACK, ENV

VAL>

VAR, ADDR, CON

functions

'Values'

addr	:		$\rightarrow$	ADDR
record_val	:	FIELD_VAL$^+$	$\rightarrow$	VAL
closure	:	MATCH$\times$ENV	$\rightarrow$	VAL
prod	:	CON$\times$VAL	$\rightarrow$	VAL
field_val	:	LAB$\times$VAL	$\rightarrow$	FIELD_VAL

'Environments'

valenv	:	VALENV_PAIR*	$\rightarrow$	VALENV
excenv	:	EXCENV_PAIR*	$\rightarrow$	EXCENV

valenv_pair	:	VAR×VAL	→	VALENV_PAIR
excenv_pair	:	EXN×EXC	→	EXCENV_PAIR
env	:	VALENV×EXCENV	→	ENV

'Stores'

mem	:	MEM_PAIR*	→	MEM
excs	:	EXCS_PAIR*	→	EXCS
mem_pair	:	ADDR×VAL	→	MEM_PAIR
excs_pair	:	EXC×EXN	→	EXCS_PAIR
store	:	MEM×EXCS	→	STORE

'Packets'

pack	:	EXC×VAL	→	PACK

'Other Classes'

exc	:	→	EXC
fail	:	→	FAIL

imports ML_Syntax

5.1. Dynamic semantics of ML

The dynamic semantics of ML is described by the following Typol specification. An ML program is a sequence of declarations that may be value declarations, exception declarations, abstract data type declarations, and local declarations (other alternatives, type and data type declarations, are irrelevant in dynamic semantics). Given an environment *e* and a store *s* a declaration evaluates to either an environment or a packet if the sequent:

$$e, s \vdash \text{DEC} : e'$$

respectively the sequent:

$$e, s \vdash \text{DEC} : pack$$

may be derived in the formal system. Note that the ⊢ operator is heavily overloaded in the present Typol specification (as it is in the Milner operational system). Remark: the braces have no particular meaning.

program ML_DS is

use ML

$$\frac{e, s \vdash \text{VALBIND} : ve, s' \qquad is\_valenv(ve)}{e, s \vdash \text{val}\,\text{VALBIND} : (ve, []), s'}$$

$$\frac{e, s \vdash \text{VALBIND} : pack, s'}{e, s \vdash \text{val}\,\text{VALBIND} : pack, s'}$$

$$\frac{e, s \vdash \text{EXCBIND} : ee, s'}{e, s \vdash \text{exception}\,\text{EXCBIND} : ([], ee), s'}$$

$$e, s \vdash \text{type}\,\text{TYBIND} : ([], []), s$$

$$e, s \vdash \text{datatype}\,\text{DATABIND} : ([], []), s$$

$$\frac{e, s \vdash \text{DEC} : e\_pack, s'}{e, s \vdash \text{abstype}\,\text{DATABIND}\,\text{with}\,\text{DEC}\,\text{end} : e\_pack, s'}$$

$$\frac{(ve, ee), s \vdash \text{DEC}_1 : (ve_1, ee_1), s_1 \qquad (ve_1 \cdot ve, ee_1 \cdot ee), s_1 \vdash \text{DEC}_2 : e\_pack, s_2}{(ve, ee), s \vdash \text{local}\,\text{DEC}_1\,\text{in}\,\text{DEC}_2\,\text{end} : e\_pack, s_2}$$

$$\frac{e, s \vdash \text{DEC}_1 : pack, s'}{e, s \vdash \text{local}\,\text{DEC}_1\,\text{in}\,\text{DEC}_2\,\text{end} : pack, s'}$$

$$\left\{\begin{array}{c}
\dfrac{e, s \vdash \text{DEC} :(ve', ee'), s'}{e, s \vdash \text{DEC} :(ve', ee'), s'} \\[2ex]
\dfrac{(ve, ee), s \vdash \text{DEC} :(ve', ee'), s' \qquad (ve' \cdot ve, ee' \cdot ee), s' \vdash \text{DEC\_SEQ} :(ve'', ee''), s''}{(ve, ee), s \vdash \text{DEC} ; \text{DEC\_SEQ} :(ve'' \cdot ve', ee'' \cdot ee'), s''} \\[2ex]
\dfrac{e, s \vdash \text{DEC} : pack, s'}{e, s \vdash \text{DEC} ; \text{DEC\_SEQ} : pack, s'} \\[2ex]
\dfrac{(ve, ee), s \vdash \text{DEC} :(ve', ee'), s' \qquad (ve' \cdot ve, ee' \cdot ee), s' \vdash \text{DEC\_SEQ} : pack, s''}{(ve, ee), s \vdash \text{DEC} ; \text{DEC\_SEQ} : pack, s''}
\end{array}\right.$$

$$\left\{\begin{array}{c}
\dfrac{e, s \vdash \text{EXP} : val, s' \qquad is\_val(val) \qquad s' \vdash \text{PAT}, val : ve, s'' \qquad is\_valenv(ve)}{e, s \vdash \text{PAT} = \text{EXP} : ve, s''} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : val, s' \qquad is\_val(val) \qquad s' \vdash \text{PAT}, val : fail, s''}{e, s \vdash \text{PAT} = \text{EXP} :<ebind, ()>, s''} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : pack, s'}{e, s \vdash \text{PAT} = \text{EXP} : pack, s'}
\end{array}\right.$$

$$\left\{\begin{array}{c}
\dfrac{(ve' \cdot ve, ee), s \vdash \text{VALBIND} : ve', s' \qquad is\_valenv(ve')}{(ve, ee), s \vdash \text{rec VALBIND} : ve', s'} \\[2ex]
\dfrac{(ve' \cdot ve, ee), s \vdash \text{VALBIND} : pack, s'}{(ve, ee), s \vdash \text{rec VALBIND} : pack, s'}
\end{array}\right.$$

$$\left\{\begin{array}{c}
\dfrac{e, s \vdash \text{VALBIND} : ve, s' \qquad is\_valenv(ve)}{e, s \vdash \text{VALBIND} : ve, s'} \\[2ex]
\dfrac{e, s \vdash \text{VALBIND} : ve, s' \qquad is\_valenv(ve) \qquad e, s' \vdash \text{VALBIND\_S} : ve', s'' \qquad is\_valenv(ve')}{e, s \vdash \text{VALBIND and VALBIND\_S} : ve' \cdot ve, s''} \\[2ex]
\dfrac{e, s \vdash \text{VALBIND} : ve, s' \qquad is\_valenv(ve) \qquad e, s' \vdash \text{VALBIND\_S} : pack, s''}{e, s \vdash \text{VALBIND and VALBIND\_S} : pack, s''} \\[2ex]
\dfrac{e, s \vdash \text{VALBIND} : pack, s'}{e, s \vdash \text{VALBIND and VALBIND\_S} : pack, s'}
\end{array}\right.$$

$$e, (mem, excs) \vdash \text{exn X} : \text{exn X} \mapsto exc, (mem, exc \mapsto \text{exn X} \cdot excs)$$

$$(ve, ee), s \vdash \text{exn X} = \text{exn X}' : \text{exn X} \mapsto exc, s \qquad (\, ee \overset{\text{exc\_env}}{\vdash} \text{exn X}' \mapsto exc \,)$$

$$\dfrac{e, s \vdash \text{EXCBIND} : ee, s'}{e, s \vdash \text{EXCBIND} : ee, s'}$$

$$\dfrac{e, s \vdash \text{EXCBIND} : ee, s' \qquad e, s' \vdash \text{EXCBIND\_S} : ee', s''}{e, s \vdash \text{EXCBIND and EXCBIND\_S} : ee' \cdot ee, s''}$$

$$\dfrac{ve \overset{\text{val\_env}}{\vdash} \text{var X} \mapsto val}{(ve, ee), s \vdash \text{var X} : val, s}$$

$$e, s \vdash \text{con X} : \text{con X}, s$$

$$e, s \vdash \text{number X} : \text{number X}, s$$

$$\left\{\begin{array}{c}
\dfrac{e, s \vdash \text{EXP} : val, s' \qquad is\_val(val)}{e, s \vdash \{\text{lab}\,\text{X} = \text{EXP}\} : \{\text{lab}\,\text{X} = val\}, s'} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : val, s' \qquad is\_val(val) \qquad e, s' \vdash \text{FIELD\_S} : \{\text{FIELD}, \text{F\_S}\}, s''}{e, s \vdash \{\text{lab}\,\text{X} = \text{EXP}, \text{FIELD\_S}\} : \{\text{lab}\,\text{X} = val, \text{FIELD}, \text{F\_S}\}, s''} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : val, s' \qquad is\_val(val) \qquad e, s' \vdash \text{FIELD\_S} : pack, s''}{e, s \vdash \{\text{lab}\,\text{X} = \text{EXP}, \text{FIELD\_S}\} : pack, s''} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : pack, s'}{e, s \vdash \{\text{lab}\,\text{X} = \text{EXP}, \text{FIELD\_S}\} : pack, s'}
\end{array}\right.$$

$$\left\{\begin{array}{c}
\dfrac{e, s \vdash \text{EXP} : \text{con}\,\text{X}, s' \qquad e, s' \vdash \text{EXP}' : val', s'' \qquad is\_val(val')}{e, s \vdash (\text{EXP}\ \text{EXP}') : \text{con}\,\text{X} \times val', s''} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : :=, s' \qquad e, s' \vdash \text{EXP}' : \text{addr}\,\text{X}, s'' \qquad e, s'' \vdash \text{EXP}'' : val, (mem, excs) \qquad is\_val(val)}{e, s \vdash ((\text{EXP}\ \text{EXP}')\ \text{EXP}'') : (), (\text{addr}\,\text{X} \mapsto val \cdot mem, excs)} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : ref, s' \qquad e, s' \vdash \text{EXP}' : val, (mem, excs) \qquad is\_val(val)}{e, s \vdash (\text{EXP}\ \text{EXP}') : \text{addr}\,\text{X}, (\text{addr}\,\text{X} \mapsto val \cdot mem, excs)} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : f, s' \qquad basfun(f) \qquad e, s' \vdash \text{EXP}' : val, s'' \qquad is\_val(val) \qquad \overset{\text{apply}}{\vdash} f, val : val'}{e, s \vdash (\text{EXP}\ \text{EXP}') : val', s''} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : \text{closure}(match, e'), s' \qquad e, s' \vdash \text{EXP}' : val', s'' \qquad is\_val(val') \qquad e', s'' \vdash match, val' : val\_pack, s'''}{e, s \vdash (\text{EXP}\ \text{EXP}') : val\_pack, s'''} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : val, s' \qquad is\_val(val) \qquad e, s' \vdash \text{EXP}' : pack, s''}{e, s \vdash (\text{EXP}\ \text{EXP}') : pack, s''} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : pack, s'}{e, s \vdash (\text{EXP}\ \text{EXP}') : pack, s'}
\end{array}\right.$$

$$\left\{\begin{array}{c}
\dfrac{(ve, ee), s \vdash \text{EXP} : val, s' \qquad is\_val(val)}{(ve, ee), s \vdash \text{raise exn}\,\text{X}\ \text{with}\ \text{EXP} : <exc, val>, s'} \qquad \left(ee \overset{\text{exc\_env}}{\vdash} \text{exn}\,\text{X} \mapsto exc \right) \\[2ex]
\dfrac{e, s \vdash \text{EXP} : pack, s'}{e, s \vdash \text{raise exn}\,\text{X}\ \text{with}\ \text{EXP} : pack, s'}
\end{array}\right.$$

$$\left\{\begin{array}{c}
\dfrac{(ve, ee), s \vdash \text{DEC} : (ve', ee'), s' \qquad (ve' \cdot ve, ee' \cdot ee), s' \vdash \text{EXP} : val\_pack, s''}{(ve, ee), s \vdash \text{let}\ \text{DEC}\ \text{in}\ \text{EXP}\ \text{end} : val\_pack, s''} \\[2ex]
\dfrac{e, s \vdash \text{DEC} : pack, s'}{e, s \vdash \text{let}\ \text{DEC}\ \text{in}\ \text{EXP}\ \text{end} : pack, s'}
\end{array}\right.$$

$$e, s \vdash \text{fun}\ \text{MATCH} : \text{closure}(\text{MATCH}, e), s$$

$$\left\{\begin{array}{c}
\dfrac{e, s \vdash \text{EXP} : val, s' \qquad is\_val(val)}{e, s \vdash \text{EXP}\ \text{handle}\ \text{HANDLER} : val, s'} \\[2ex]
\dfrac{e, s \vdash \text{EXP} : pack, s' \qquad e, s' \vdash \text{HANDLER}, pack : val\_pack, s''}{e, s \vdash \text{EXP}\ \text{handle}\ \text{HANDLER} : val\_pack, s''}
\end{array}\right.$$

$$\dfrac{(ve, ee), s \vdash \text{MATCH}, val : val\_pack, s'}{(ve, ee), s \vdash \text{exn}\,\text{X}\ \text{with}\ \text{MATCH}, <exc, val> : val\_pack, s'} \qquad \left(ee \overset{\text{exc\_env}}{\vdash} \text{exn}\,\text{X} : exc \right)$$

$$e, s \vdash \text{exn}\,\text{X}\ \text{with}\ \text{MATCH}, pack : fail$$

$$\dfrac{e, s \vdash \text{EXP} : val\_pack, s'}{e, s \vdash \| \ \text{EXP}, pack : val\_pack, s'}$$

$$\left\{\begin{array}{c}
\dfrac{e, s \vdash \textsc{hrule}, pack: val\_pack, s' \qquad is\_val\_pack(val\_pack)}{e, s \vdash \textsc{hrule} \parallel \textsc{hrule\_s}, pack: val\_pack, s'} \\[2ex]
\dfrac{e, s \vdash \textsc{hrule}, pack: fail, s' \qquad e, s' \vdash \textsc{hrule\_s}, pack: val\_pack, s''}{e, s \vdash \textsc{hrule} \parallel \textsc{hrule\_s}, pack: val\_pack, s''} \\[2ex]
e, s \vdash \text{handler}[], pack: pack, s
\end{array}\right.$$

$$\left\{\begin{array}{c}
\dfrac{s \vdash \textsc{pat}, val: ve', s' \qquad is\_valenv(ve') \qquad (ve' \cdot ve, ee), s' \vdash \textsc{exp}: val\_pack, s''}{(ve, ee), s \vdash \textsc{pat} \Rightarrow \textsc{exp}, val: val\_pack, s''} \\[2ex]
\dfrac{s \vdash \textsc{pat}, val: fail, s'}{e, s \vdash \textsc{pat} \Rightarrow \textsc{exp}, val: fail, s'}
\end{array}\right.$$

$$\left\{\begin{array}{c}
\dfrac{e, s \vdash \textsc{mrule}, val: val\_pack, s' \qquad is\_val\_pack(val\_pack)}{e, s \vdash \textsc{mrule} \mid \textsc{mrule\_s}, val: val\_pack, s'} \\[2ex]
\dfrac{e, s \vdash \textsc{mrule}, val: fail, s' \qquad e, s' \vdash \textsc{mrule\_s}, val: val\_pack, s''}{e, s \vdash \textsc{mrule} \mid \textsc{mrule\_s}, val: val\_pack, s''} \\[2ex]
e, s \vdash \text{match}[], val: <ematch, ()>, s
\end{array}\right.$$

$$\left\{\begin{array}{c}
s \vdash \_\,, val: [], s \\[1.5ex]
s \vdash \text{var}\,\textsc{x}, val: \text{var}\,\textsc{x} \mapsto val, s \\[1.5ex]
s \vdash \text{con}\,\textsc{x}, \text{con}\,\textsc{x}: [], s \\[1.5ex]
s \vdash \text{number}\,\textsc{x}, \text{number}\,\textsc{x}: [], s \\[1.5ex]
\dfrac{(\textsc{mem}, \textsc{excs}) \vdash \textsc{pat}, val: ve\_fail, s'}{(\textsc{mem}, \textsc{excs}) \vdash \text{con}\ con\ con\ \text{“ref”}(\textsc{pat}), \text{addr}\,\textsc{x}: ve\_fail, s'} \qquad (\textsc{mem} \overset{\text{mem}}{\vdash} \text{addr}\,\textsc{x} \mapsto val) \\[2ex]
\dfrac{s \vdash \textsc{pat}, \textsc{val}: ve\_fail, s'}{s \vdash con\,\textsc{x}(\textsc{pat}), con\,\textsc{x} \times \textsc{val}: ve\_fail, s'} \\[2ex]
\dfrac{s \vdash \textsc{pat}, val: ve, s' \qquad is\_valenv(ve)}{s \vdash \text{var}\,\textsc{x}\ \text{as}\,\textsc{pat}, val: ve \cdot \text{var}\,\textsc{x} \mapsto val, s'} \\[2ex]
s \vdash pat, val: fail, s
\end{array}\right.$$

set EXC_ENV is

$$\text{exn}\,\textsc{x} \mapsto exc \cdot \textsc{excenv} \vdash \text{exn}\,\textsc{x} \mapsto exc$$

$$\left\{\begin{array}{c}
\dfrac{\textsc{f} \vdash \text{exn}\,\textsc{x} \mapsto exc}{\textsc{f} \cdot \textsc{t} \vdash \text{exn}\,\textsc{x} \mapsto exc} \\[2ex]
\dfrac{\textsc{t} \vdash \text{exn}\,\textsc{x} \mapsto exc}{\textsc{f} \cdot \textsc{t} \vdash \text{exn}\,\textsc{x} \mapsto exc}
\end{array}\right.$$

end EXC_ENV;

set VAL_ENV is

$$\text{var}\,\textsc{x} \mapsto val \cdot \textsc{valenv} \vdash \text{var}\,\textsc{x} \mapsto val$$

$$\frac{\mathrm{T} \vdash \mathbf{var}\,\mathrm{X} \mapsto val}{\mathrm{F} \cdot \mathrm{T} \vdash \mathbf{var}\,\mathrm{X} \mapsto val} \qquad (\ isvar(\mathrm{F})\ )$$

$$\left\{ \begin{array}{c} \dfrac{\mathrm{F} \vdash \mathbf{var}\,\mathrm{X} \mapsto val}{\mathrm{F} \cdot \mathrm{T} \vdash \mathbf{var}\,\mathrm{X} \mapsto val} \\ \dfrac{\mathrm{T} \vdash \mathbf{var}\,\mathrm{X} \mapsto val}{\mathrm{F} \cdot \mathrm{T} \vdash \mathbf{var}\,\mathrm{X} \mapsto val} \end{array} \right.$$

end VAL_ENV;

set MEM is

$$\mathbf{addr}\,y \mapsto val \cdot \mathrm{VALENV} \vdash \mathbf{addr}\,x \mapsto val \qquad (\ samevar(x, y)\ )$$

$$\left\{ \begin{array}{c} \dfrac{\mathrm{F} \vdash \mathbf{addr}\,x \mapsto val}{\mathrm{F} \cdot \mathrm{T} \vdash \mathbf{addr}\,x \mapsto val} \\ \dfrac{\mathrm{T} \vdash \mathbf{addr}\,x \mapsto val}{\mathrm{F} \cdot \mathrm{T} \vdash \mathbf{addr}\,x \mapsto val} \end{array} \right.$$

end MEM;

set BASFUN is

$$+$$

$$-$$

$$*$$

$$\frac{val}{apply(val, val')}$$

end BASFUN;

set APPLY is

$$\frac{plus(\mathrm{X}, \mathrm{Y}, \mathrm{Z})}{\vdash apply(+, \mathbf{number}\,\mathrm{X}), \mathbf{number}\,\mathrm{Y} : \mathbf{number}\,\mathrm{Z}}$$

$$\frac{minus(\mathrm{X}, \mathrm{Y}, \mathrm{Z})}{\vdash apply(-, \mathbf{number}\,\mathrm{X}), \mathbf{number}\,\mathrm{Y} : \mathbf{number}\,\mathrm{Z}}$$

$$\frac{product(\mathrm{X}, \mathrm{Y}, \mathrm{Z})}{\vdash apply(*, \mathbf{number}\,\mathrm{X}), \mathbf{number}\,\mathrm{Y} : \mathbf{number}\,\mathrm{Z}}$$

$$\vdash f, val : apply(f, val)$$

end APPLY;

set IS_VALENV is

$$\frac{}{[]}$$

$$\mathrm{VE} \cdot \mathrm{VALENV}$$

end IS_VALENV;

set IS_VAL is

$$\mathit{con}\,\mathrm{X}$$

$$\mathrm{number}\,\mathrm{X}$$

$$\mathrm{CON} \times \mathrm{VAL}$$

$$\{\mathrm{FIELD}, \mathrm{RECORD}\}$$

$$\mathrm{addr}\,\mathrm{X}$$

$$\mathrm{closure}(\mathrm{MATCH}, \mathrm{E})$$

$$:=$$

$$\mathit{ref}$$

$$\frac{\mathit{basfun}(f)}{f}$$

end IS_VAL;

set IS_VAL_PACK is

$$<\mathit{exc}, \mathit{val}>$$

$$\frac{\mathit{is\_val}(\mathit{val})}{\mathit{val}}$$

end IS_VAL_PACK;

end ML_DS

5.2. Exceptions

The exception mechanism of ML is based upon textual scope of exception identifiers. After an exception identifier have been declared, an exception raised by a raise expression

raise *exn* **with** *exp*

may be handled by a **handler** expression

exn **with** *match*

which lies in the textual scope of the declaration. Now for every expression, except for a handler, whenever the result of a subexpression is a packet then no further subevaluation occur, and the packet is also the result of the main evaluation. Thus, except for the rule that specify the evaluation of a handle expression, rules are of two forms. Consider the case of ML declarations.

As noticed above a declaration evaluates to a new environment. For example a value declaration is specified by the following rule:

$$\frac{e, s \vdash \text{VALBIND} : ve, s'}{e, s \vdash \text{val VALBIND} : (ve, \emptyset), s'} \tag{1}$$

where the variable *ve* is of type VALENV. But the evaluation of the value binding VALBIND may be a packet $< ebind, () >$ where *ebind* stands for the exception associated to the predefined identifier *bind* in the exception environment. This is expressed by the second rule

$$\frac{e, s \vdash \text{VALBIND} : pack, s'}{e, s \vdash \text{val VALBIND} : pack, s'} \tag{2}$$

Declarations may be composed with a sequence operator ; and the resulting environment is the union of each local environment:

$$e, s \vdash \emptyset : \emptyset, s \tag{3}$$

$$\frac{(ve, ee), s \vdash \text{DEC} : (ve', ee'), s' \qquad (ve' \cdot ve, ee' \cdot ee), s' \vdash \text{DEC\_SEQ} : (ve'', ee''), s''}{(ve, ee), s \vdash \text{DEC; DEC\_SEQ} : (ve'' \cdot ve', ee'' \cdot ee'), s''} \tag{4}$$

Next we have to express what appends when a declaration returns a packet.

$$\frac{e, s \vdash \text{DEC} : pack, s'}{e, s \vdash \text{DEC; DEC\_SEQ} : pack, s'} \tag{5}$$

$$\frac{(ve, ee), s \vdash \text{DEC} : (ve', ee'), s' \qquad (ve' \cdot ve, ee' \cdot ee), s' \vdash \text{DEC\_SEQ} : pack, s''}{(ve, ee), s \vdash \text{DEC; DEC\_SEQ} : pack, s''} \tag{6}$$

Although these six rules express the semantics of a part of ML declarations their compilation into an efficient code is not so easy. Consider first the rules 1 and 2. They only differ by their premises, and more precisely by the type of the variables *ve* and *pack*. Such rules are not directly executable: they must be distinguable by a syntactic construct. This kind of rule selection is usually solved by pattern matching: we specialize a variable with the help of a tree pattern.

For example we may replace *pack* by the abstract tree *pack*(EXC, VAL). But now the more precise rule is rule 2: whenever a declaration has to be evaluated it is that rule which is used first. Now if there is no exception raised during evaluation this proof fails and the whole process starts again.

To avoid such a time consuming evaluation we may render rule 1 more precise. In the present version of Typol we cannot match a list pattern with only one rule: we must have a rule for the empty list and a rule for a list with at least one element. To avoid this useless repetition we prefer to use the auxiliary Typol set IS_VALENV to check that the value of the variable *ve* is of type VALENV. Now we first try to prove that the evaluation returns a value.

In fact the difficulty is not really solved and the situation is worse for a sequence of declarations. Rules 3 and 4 are used to prove that a sequence of declaration returns a new environment. But if one of them gives a proof of *pack* then the whole sequence has to be reevaluated with rules 5 and 6 (rule 4 cannot be used when a declaration returns a *packet*).

The difficulty we are faced with is not really a new one (for example it is already present with the if statement in ASPLE). But it is more evident here. In fact this problem is due to "a one rule at a time" Typol to Prolog compilation. We think that a better compilation strategy is possible, but we shall not discuss it there.

5.3. Matching of Values

A conditionnal expression:

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

is translated into the following core Ml expression:

$$\begin{pmatrix} \text{fun} & \text{true} & \Rightarrow & e_2 \\ & \text{false} & \Rightarrow & e_3 \end{pmatrix} \; e_1$$

the evaluation of which is described by the following rule:

$$\frac{e \vdash \text{EXP} : [\![\text{MATCH}, e']\!] \quad e \vdash \text{EXP}' : val' \quad e' \vdash \text{MATCH}, val' : val}{e \vdash \text{EXP EXP}' : val} \tag{6}$$

The sequent $e' \vdash \text{MATCH}, val' : val$ describes the application of a match (here the true and false branches of the conditional) to a value (here the value of the boolean expression e_1). The rules of this application are:

$$\frac{e, s \vdash \text{MRULE}, val : val\_pack, s' \qquad is\_val\_pack(val\_pack)}{e, s \vdash \text{MRULE} \mid \text{MRULE\_S}, val : val\_pack, s'} \tag{7}$$

$$\frac{e, s \vdash \text{MRULE}, val : fail, s' \qquad e, s' \vdash \text{MRULE\_S}, val : val\_pack, s''}{e, s \vdash \text{MRULE} \mid \text{MRULE\_S}, val : val\_pack, s''} \tag{8}$$

$$e, s \vdash \text{match}[], val : <ematch, ()>, s \tag{9}$$

where the matching of a pattern to a value is described by the following rules (the rules for records are omitted):

$$s \vdash \_, val : [], s$$

$$s \vdash \text{var}\,\text{X}, val : \text{var}\,\text{X} \mapsto val, s$$

$$s \vdash \text{con}\,\text{X}, \text{con}\,\text{X} : [], s$$

$$s \vdash \text{number}\,\text{X}, \text{number}\,\text{X} : [], s$$

$$\frac{(\text{MEM}, \text{EXCS}) \vdash \text{PAT}, val : ve\_fail, s'}{(\text{MEM}, \text{EXCS}) \vdash \text{con}\; con\; con\; \text{“ref”}(\text{PAT}), \text{addr}\,\text{X} : ve\_fail, s'} \quad (\text{MEM} \overset{\text{mem}}{\vdash} \text{addr}\,\text{X} \mapsto val)$$

$$\frac{s \vdash \text{PAT}, \text{VAL} : ve\_fail, s'}{s \vdash con\,\text{X}(\text{PAT}), con\,\text{X} \times \text{VAL} : ve\_fail, s'}$$

$$\frac{s \vdash \text{PAT}, val : ve, s' \qquad is\_valenv(ve)}{s \vdash \text{var}\,\text{X as PAT}, val : ve \cdot \text{var}\,\text{X} \mapsto val, s'}$$

But what is the meaning of *fail*? In its specifications Milner says that matching a pattern to a value evaluates either to a value environment or to fail. But the second case holds only when no evaluation can be inferred for the pair (*pat*, *val*) from the former rules. It is well known that it is difficult to specify such a negation by failure in inference systems. If we assume that rules may be conditioned by a negation operator $\neq$ (but this has to be proved) which is predefined in the Typol language, we may include in the previous collection of rules the following rules:

$$s \vdash \text{con}\,\text{X}, \text{con}\,\text{Y} : fail, s \qquad (\text{X} \neq \text{Y})$$

$$s \vdash \text{number}\,\text{X}, \text{number}\,\text{Y} : fail, s \qquad (\text{Y} \neq \text{X})$$

$$\frac{s \vdash \text{PAT}, val : fail, s'}{s \vdash \text{var}\,\text{X as PAT}, val : fail, s'}$$

We are not convinced that such a solution is elegant, but at least it works. We should prefer to avoid the use of such a failure value. But this implies that we are able to express rules that depend on the failure of their premisses (see rules 7 and 8).

6. The language ESTEREL

Esterel is a real time language developed at the "CMA-Ecole des Mines" in Sophia-Antipolis by G. Berry and colleagues. It is aimed to specify the behaviour of automatic systems (such as electric toy cars, video games, wrist watches and other reactive systems). One specifies in an imperative way what the system must do in answer to such or such signal. Signals come both from the outside world and from other parts of the system. Esterel includes most of the features of other Pascal-like languages, like case and conditional statements, plus parallelism, and new statements dealing with signals. In its present state, the Esterel system provides a type checker, controllers for certain static semantics properties, an interpreter and a compiler. The compiler produces a finite automaton corresponding to the specification in Esterel, which in turn may be compiled to run on computers or later "compiled" directly on chips.

Our interest for Esterel stems from several reasons: first, Esterel is not a toy-size language any more, and it was a challenge to try and explain it completely. For instance, it was our first big type checker written in Typol. Second some features like block structure or signals are interesting to type-check in Typol. So we wrote a first type-checker which is nothing but the specification of what is a well typed Esterel program. This is the one we present here.

Then we found other interests in Esterel. The Esterel type-checker directly derived from the specification is a little too dumb, and we tried to improve it in various ways. First, it's a pity the type-checking process should stop on the first error reached, so we tried to make it go on anyway. Second, since it went on, we had to have it recover from these errors, and react as cleverly as possible. Third, we tried to make it incremental, which was an easier thing to do as soon as we had a good error recovery mechanism. Incremental type-checking is only useful for real-size programs written in real-size languages. We are now trying to design an automatic way of going from the "dumb" type-checker" to the clever one. Of course we would like this automatic transformation to apply to any type-checker, not only Esterel any more. Esterel will only be the leading example, because it includes the principal features of other real-size languages.

Abstract syntax of Esterel

We present in the following the abstract syntax of Esterel. Right now, we must point out that both the concrete and abstract syntax of Esterel may change, for this language is still in its development phase. We can already note the fairy large number of constructors. Features we want to insist on are:

- large number of declarations.
- block-structured instructions, like localvardecl, localsignaldecl, or tag
- exception handling instructions, like trapfailure.

Abstract Syntax of STRL

sorts

LFUNCTIONARG, COMP, EXPRESSION, SIGNALSUBST, LSIGNALSUBST, FAILUREHANDLER, LFAILURE-HANDLER, SELECTCASE, LSELECTCASE, LSIGNALDECL, VARIABLEINIT, LVARIABLEINIT, ONETYPE-VARDECLS, LVARIABLEDECL, LPROCEDUREARG, OPTIONALBINDING, OPTIONALNEXT, COMPTE, OC-CURENCE, OCCWITHOUTCOUNT, LIDENT, INSTRUCTION, LINCOMPATIBILITY, RELATIONDECL, SIG-NALDECL, LRELATIONDECL, LINPUTOUTPUTDECL, LOUTPUTDECL, LINPUTDECL, IDENT, TYPE, LTYPE, PROCEDUREDECL, FUNCTIONDECL, LCONSTANTIDENT, ONETYPECSTDECL, TYPEDECL, LPROCEDURE-DECL, LFUNCTIONDECL, LCONSTANTDECL, LTYPEDECL, DECLARATION, LDECLARATION, MODULE

subsorts

functions

'MODULE'

module : IDENT×LDECLARATION×INSTRUCTION → MODULE

'DECLARATIONS'

ldeclaration	:	DECLARATION*	→	LDECLARATION
typedecls	:	LTYPEDECL	→	DECLARATION
constantdecls	:	LCONSTANTDECL	→	DECLARATION
functiondecls	:	LFUNCTIONDECL	→	DECLARATION
proceduredecls	:	LPROCEDUREDECL	→	DECLARATION
inputdecls	:	LINPUTDECL	→	DECLARATION
outputdecls	:	LOUTPUTDECL	→	DECLARATION
inputoutputdecls	:	LINPUTOUTPUTDECL	→	DECLARATION
relationdecls	:	LRELATIONDECL	→	DECLARATION
ltypedecl	:	TYPEDECL*	→	LTYPEDECL
lconstantdecl	:	ONETYPECSTDECL*	→	LCONSTANTDECL
lfunctiondecl	:	FUNCTIONDECL*	→	LFUNCTIONDECL
lproceduredecl	:	PROCEDUREDECL*	→	LPROCEDUREDECL
typedecl	:		→	TYPEDECL
onetypecstdecl	:	LCONSTANTIDENT×TYPE	→	ONETYPECSTDECL
lconstantident	:	IDENT*	→	LCONSTANTIDENT
functiondecl	:	IDENT×LTYPE×TYPE	→	FUNCTIONDECL
proceduredecl	:	IDENT×LTYPE×LTYPE	→	PROCEDUREDECL
ltype	:	TYPE*	→	LTYPE
type	:		→	TYPE
ident	:		→	IDENT

'SIGNAUX'

linputdecl	:	SIGNALDECL*	→	LINPUTDECL
loutputdecl	:	SIGNALDECL*	→	LOUTPUTDECL
linputoutputdecl	:	SIGNALDECL*	→	LINPUTOUTPUTDECL
lrelationdecl	:	RELATIONDECL*	→	LRELATIONDECL
lsignaldecl	:	SIGNALDECL*	→	LSIGNALDECL
puresignaldecl	:	IDENT	→	SIGNALDECL
singlesignaldecl	:	IDENT×TYPE	→	SIGNALDECL
multiplesignaldecl	:	IDENT×TYPE×IDENT	→	SIGNALDECL
causalitydecl	:	IDENT×IDENT	→	RELATIONDECL
incompatibilitydecl	:	LINCOMPATIBILITY	→	RELATIONDECL
lincompatibility	:	IDENT*	→	LINCOMPATIBILITY

'INSTRUCTIONS'

sequence	:	INSTRUCTION*	→	INSTRUCTION
parallele	:	INSTRUCTION*	→	INSTRUCTION
nothing	:		→	INSTRUCTION
halt	:		→	INSTRUCTION
assignment	:	IDENT×EXPRESSION	→	INSTRUCTION
procedurecall	:	IDENT×LIDENT×LPROCEDUREARG	→	INSTRUCTION
emit	:	IDENT×EXPRESSION	→	INSTRUCTION
upto	:	INSTRUCTION×OCCURENCE	→	INSTRUCTION
inpresence	:	OCCWITHOUTCOUNT×INSTRUCTION	→	INSTRUCTION
loop	:	INSTRUCTION	→	INSTRUCTION
conditional	:	EXPRESSION×INSTRUCTION×INSTRUCTION	→	INSTRUCTION
tag	:	IDENT×INSTRUCTION	→	INSTRUCTION
exit	:	IDENT	→	INSTRUCTION
localvardecl	:	LVARIABLEDECL×INSTRUCTION	→	INSTRUCTION
localsignaldecl	:	LSIGNALDECL×INSTRUCTION	→	INSTRUCTION
repeat	:	EXPRESSION×INSTRUCTION	→	INSTRUCTION

await	:	OCCURENCE	→	INSTRUCTION
on	:	OCCURENCE×INSTRUCTION	→	INSTRUCTION
select	:	LSELECTCASE	→	INSTRUCTION
uptoeach	:	INSTRUCTION×OCCURENCE	→	INSTRUCTION
every	:	OCCURENCE×INSTRUCTION	→	INSTRUCTION
inabsence	:	OCCURENCE×INSTRUCTION	→	INSTRUCTION
watching	:	INSTRUCTION×OCCURENCE×INSTRUCTION	→	INSTRUCTION
trapfailure	:	LSIGNALDECL×INSTRUCTION×LFAILUREHANDLER	→	INSTRUCTION
failwith	:	IDENT×EXPRESSION	→	INSTRUCTION
copymodule	:	IDENT×LSIGNALSUBST	→	INSTRUCTION
null_tree	:		→	INSTRUCTION
lident	:	IDENT*	→	LIDENT
occwithoutcount	:	IDENT×OPTIONALBINDING	→	OCCWITHOUTCOUNT
occurence	:	COMPTE×IDENT×OPTIONALBINDING	→	OCCURENCE
compte	:	OPTIONALNEXT×EXPRESSION	→	COMPTE
next	:		→	OPTIONALNEXT
null_tree	:		→	OPTIONALNEXT
optionalbinding	:	IDENT	→	OPTIONALBINDING
null_tree	:		→	OPTIONALBINDING
lprocedurearg	:	EXPRESSION*	→	LPROCEDUREARG
lvariabledecl	:	ONETYPEVARDECLS*	→	LVARIABLEDECL
onetypevardecls	:	LVARIABLEINIT×TYPE	→	ONETYPEVARDECLS
lvariableinit	:	VARIABLEINIT*	→	LVARIABLEINIT
variableinit	:	IDENT×EXPRESSION	→	VARIABLEINIT
lselectcase	:	SELECTCASE*	→	LSELECTCASE
selectcase	:	OCCURENCE×INSTRUCTION	→	SELECTCASE
lfailurehandler	:	FAILUREHANDLER*	→	LFAILUREHANDLER
failurehandler	:	OCCWITHOUTCOUNT×INSTRUCTION	→	FAILUREHANDLER
lsignalsubst	:	SIGNALSUBST*	→	LSIGNALSUBST
null_tree	:		→	LSIGNALSUBST
signalsubst	:	IDENT×IDENT	→	SIGNALSUBST

'EXPRESSIONS'

minus	:	EXPRESSION	→	EXPRESSION
puiss	:	EXPRESSION×EXPRESSION	→	EXPRESSION
mult	:	EXPRESSION×EXPRESSION	→	EXPRESSION
div	:	EXPRESSION×EXPRESSION	→	EXPRESSION
plus	:	EXPRESSION×EXPRESSION	→	EXPRESSION
moins	:	EXPRESSION×EXPRESSION	→	EXPRESSION
compar	:	EXPRESSION×COMP×EXPRESSION	→	EXPRESSION
not	:	EXPRESSION	→	EXPRESSION
or	:	EXPRESSION×EXPRESSION	→	EXPRESSION
and	:	EXPRESSION×EXPRESSION	→	EXPRESSION
ident	:		→	EXPRESSION
string	:		→	EXPRESSION
natural	:		→	EXPRESSION
bool	:		→	EXPRESSION
functioncall	:	IDENT×LFUNCTIONARG	→	EXPRESSION
null_tree	:		→	EXPRESSION
comp	:		→	COMP
lfunctionarg	:	EXPRESSION*	→	LFUNCTIONARG

6.1. The simple Esterel type-checker

The simple type-checker is nothing but the specification in Typol of what is a well-typed Esterel program. The conditions an Esterel program must fullfill are the same as for usual Pascal-like languages. Every identifier used must be declared before it is used, and the scope rules are the usual ones. Esterel does not support overloading, nor polymorphic types. Signal declarations are more complex. First, signals may be of kind input or output or both. Then they may carry or not a value of a certain type. Finally, one may specify the composition function to apply between two values emitted at the same time by two emissions of the same signal.

The complete type-checker is listed below, but we will now give comments for a few rules.

$$\frac{\overset{\textbf{initenvir}}{\vdash} \; : \rho \qquad \rho \overset{\textbf{tc}}{\vdash} \text{ARBRE} :}{\vdash \text{ARBRE} :}$$

This is the toplevel rule of the typechecking process. To typecheck any tree ARBRE, one must take an initial environment ρ, then typecheck the tree in this environment.

$$\frac{\rho \vdash \text{LDECL} : \rho' \qquad \rho' \vdash \text{INSTR}}{\rho \vdash \text{module}\,\text{NOM} : \text{LDECL} \;\; \text{INSTR}.}$$

This is how to typecheck a complete Esterel program, which is called a module, in any environment ρ. First the declaration part must typecheck well, yielding a new environment ρ', then the instruction part must typecheck well in ρ'. Since instructions never change the environment, this gives no new environment.

$$\frac{\rho \quad \overset{\textbf{declaretype}}{\vdash} \quad \text{typedecl}\,\text{TYPE} : \rho'}{\rho \vdash \text{typedecl}\,\text{TYPE} : \rho'}$$

When the typechecker encounters a type declaration, it calls a special set which adds the declaration of this new type in the environment. Therefore this gives a new environment ρ'. All declaration routines, like declaretype, call the external "ajouter" routine, which checks for double declarations (warnings), or redeclarations (errors).

$$\frac{\rho \;\; \overset{\textbf{istype}}{\vdash} \;\; \text{type}\,\text{TYPE}}{\rho \vdash \text{type}\,\text{TYPE} : \text{TYPE}}$$

When a program uses a type, one must check this type was declared before. This is the use of the istype set, which gives back the name of the type, for later use.

$$\frac{\rho \vdash \text{TYPE} : \tau \qquad \rho \;\; \overset{\textbf{isfunc}}{\vdash} \;\; \text{FUNC} : \tau \times \tau \rightarrow \tau \qquad \rho \quad \overset{\textbf{declaresignal}}{\vdash} \quad \text{IDENT}, [\sigma; \tau] : \rho'}{\rho \vdash \text{multiple}\,\text{IDENT}(\text{TYPE}, \text{FUNC}), \sigma : \rho'}$$

Before declaring a new signal, one must verify that the TYPE used is declared, that the fuction given is declared of the correct type.

$$\frac{\rho \vdash \text{INSTR}_1 \qquad \rho \vdash \text{INSTR}_2}{\rho \vdash \text{INSTR}_1 \parallel \text{INSTR}_2}$$

Typechecking a parallel instruction is much simple. Just typecheck both sons in the current environment ρ.

$$\frac{\rho \overset{\textbf{issignal}}{\vdash} \text{ID} : [<\_, \rightarrow>; \tau] \qquad \rho \vdash \text{EXPR} : \tau}{\rho \vdash \text{emit ID}(\text{EXPR})}$$

If the program emits the signal ID along with a value EXPR, one must check first that the signal is at least an "output" one -this is the meaning of the right arrow- and that the signal may carry a value, which must be of the same type τ as the expression EXPR.

$$\frac{\rho \overset{\textbf{blocvar}}{\vdash} : \rho' \qquad \rho' \vdash \text{LDCLVAR} : \rho'' \qquad \rho'' \vdash \text{INSTR}}{\rho \vdash \text{var LDCLVAR in INSTR end}}$$

The instruction INSTR inside a local variable block must typecheck well in a new environment ρ'' which is obtained from the old one ρ in the following way. First open a new block for variable declarations, giving ρ'. This is to check correctly double declarations and redeclarations. Then add the declarations of the local variables in the usual way, yielding ρ''.

$$\frac{\pi \overset{\textbf{firstdecl}}{\vdash} \text{ID} \mapsto \tau' \qquad eq(\tau, \tau')}{\_; \pi; \_; \_; \_; \_ \vdash \text{ident ID} : \tau}$$

The environment is made of six parts, the second one being for variables. Looking for a variable in this environment π is looking for its latest declaration, if it exists, and only for it. This is done by the "firstdecl" routine, which is written in Prolog, for the time being.

$$\vdash : [\textit{integer} \cdot \textit{boolean} \cdot \textit{string}]; []; []; []; []; []$$

The initial environment is not completely empty, for integer, boolean and string types are predefined.

In the listing which follows, we have separated the rules into four groups. The first group is gathered in the program TC1, and these rules deal mostly with the declaration part of an Esterel program. The second group is in the program TC2, the rules of which deal with statements. Program TC3 contains rules about expressions, which are few, and last program TC4 contains utility routines to modify the environment, or to tell wether an identifier is declared or not.

We could have written TC4 in an alternate way, with one set for all declarations, one for ISVAR, ISTAG, etc. For this, we should include in the calls to "declare", for instance, a new field telling if we want to declare a type or a var or a signal,and so on. We should then get only one set "DECLARE", one set "IS", and one set "BLOC".

6.2. Type checking of Declarations

program TC1 is

use STRL

$$\frac{\overset{\textbf{initenvir}}{\vdash} : \rho \qquad \rho \overset{\textbf{tc}}{\vdash} \text{ARBRE} :}{\vdash \text{ARBRE} :}$$

set TC is

$$\frac{\rho \vdash \text{DECL} : \rho' \qquad \rho' \vdash \text{LDECL} : \rho''}{\rho \vdash \text{DECL LDECL} : \rho''}$$

$$\rho \vdash \textbf{ldeclaration}[\,] : \rho$$

$$\frac{\rho \vdash \text{LDCLTYPE} : \rho'}{\rho \vdash \textbf{typedecls}(\text{LDCLTYPE}) : \rho'}$$

$$\frac{\rho \vdash \text{TYPE} : \rho' \qquad \rho' \vdash \text{LTYPE} : \rho''}{\rho \vdash \text{TYPE}, \text{LTYPE} : \rho''}$$

$$\rho \vdash \textbf{ltypedecl}[\,] : \rho$$

$$\frac{\rho \overset{\textbf{declaretype}}{\vdash} \textbf{typedecl}\ \text{TYPE} : \rho'}{\rho \vdash \textbf{typedecl}\ \text{TYPE} : \rho'}$$

$$\frac{\rho \vdash \text{LDCLCNST} : \rho'}{\rho \vdash \textbf{constant}\ \text{LDCLCNST}; : \rho'}$$

$$\frac{\rho \vdash \text{DCLCNST} : \rho' \qquad \rho' \vdash \text{LDCLCNST} : \rho''}{\rho \vdash \text{DCLCNST}, \text{LDCLCNST} : \rho''}$$

$$\rho \vdash \textbf{lconstantdecl}[\,] : \rho$$

$$\frac{\rho \vdash \text{TYPE} : \tau \qquad \rho \vdash \text{LNOMS}, \tau : \rho'}{\rho \vdash \text{LNOMS} : \text{TYPE} : \rho'}$$

$$\frac{\rho \overset{\textbf{declarevar}}{\vdash} \text{ID}, \tau : \rho' \qquad \rho' \vdash \text{LNOMS}, \tau : \rho''}{\rho \vdash \textbf{lconstantident}[\text{ID} \cdot \text{LNOMS}], \tau : \rho''}$$

$$\rho \vdash \textbf{lconstantident}[\,], \tau : \rho$$

$$\frac{\rho \vdash \text{LDCLFUNC} : \rho'}{\rho \vdash \textbf{function}\ \text{LDCLFUNC}; : \rho'}$$

$$\frac{\rho \vdash \text{DCLFUNC} : \rho' \qquad \rho' \vdash \text{LDCLFUNC} : \rho''}{\rho \vdash \text{DCLFUNC}, \text{LDCLFUNC} : \rho''}$$

$$\rho \vdash \textbf{lfunctiondecl}[\,] : \rho$$

$$\frac{\rho \vdash \text{TYPE} : \tau_1 \qquad \rho \vdash \text{LTYPE} : \tau_2 \qquad \rho \overset{\textbf{declarefunc}}{\vdash} \text{FUNC}, \tau_2 \rightarrow \tau_1 : \rho'}{\rho \vdash \text{FUNC}(\text{LTYPE}) : \text{TYPE} : \rho'}$$

$$\frac{\rho \vdash \text{TYPE} : \tau_1 \qquad \rho \vdash \text{LTYPE} : \tau_2}{\rho \vdash \text{TYPE}, \text{LTYPE} : \tau_1 \mid \tau_2}$$

$$\rho \vdash \textbf{ltype}[\,] : [\,]$$

$$\frac{\rho \overset{\textbf{istype}}{\vdash} \textbf{type}\,\mathrm{TYPE}}{\rho \vdash \textbf{type}\,\mathrm{TYPE} : \mathrm{TYPE}}$$

$$\frac{\rho \vdash \mathrm{LDCLPROC} : \rho'}{\rho \vdash \textbf{procedure}\,\mathrm{LDCLPROC}; : \rho'}$$

$$\frac{\rho \vdash \mathrm{DCLPROC} : \rho' \qquad \rho' \vdash \mathrm{LDCLPROC} : \rho''}{\rho \vdash \mathrm{DCLPROC}, \mathrm{LDCLPROC} : \rho''}$$

$$\rho \vdash \textbf{lproceduredecl}[] : \rho$$

$$\frac{\rho \vdash \mathrm{LTYPE}_1 : \tau_1 \qquad \rho \vdash \mathrm{LTYPE}_2 : \tau_2 \qquad \rho \overset{\textbf{declareproc}}{\vdash} \mathrm{PROC}, (\tau_1)(\tau_2) : \rho'}{\rho \vdash \mathrm{PROC}(\mathrm{LTYPE}_1)(\mathrm{LTYPE}_2) : \rho'}$$

$$\frac{\rho \vdash \mathrm{LDCLINPUT} : \rho'}{\rho \vdash \textbf{input}\,\mathrm{LDCLINPUT}; : \rho'}$$

$$\frac{\rho \vdash \mathrm{DCLSIGNAL}, <\leftarrow, \times> : \rho' \qquad \rho' \vdash \mathrm{LDCLINPUT} : \rho''}{\rho \vdash \mathrm{DCLSIGNAL}, \mathrm{LDCLINPUT} : \rho''}$$

$$\rho \vdash \textbf{linputdecl}[] : \rho$$

$$\frac{\rho \vdash \mathrm{LDCLOUTPUT} : \rho'}{\rho \vdash \textbf{output}\,\mathrm{LDCLOUTPUT}; : \rho'}$$

$$\frac{\rho \vdash \mathrm{DCLSIGNAL}, <\times, \rightarrow> : \rho' \qquad \rho' \vdash \mathrm{LDCLOUTPUT} : \rho''}{\rho \vdash \mathrm{DCLSIGNAL}, \mathrm{LDCLOUTPUT} : \rho''}$$

$$\rho \vdash \textbf{loutputdecl}[] : \rho$$

$$\frac{\rho \vdash \mathrm{LDCLIO} : \rho'}{\rho \vdash \textbf{inputoutput}\,\mathrm{LDCLIO}; : \rho'}$$

$$\frac{\rho \vdash \mathrm{DCLSIGNAL}, <\leftarrow, \rightarrow> : \rho' \qquad \rho' \vdash \mathrm{LDCLIO} : \rho''}{\rho \vdash \mathrm{DCLSIGNAL}, \mathrm{LDCLIO} : \rho''}$$

$$\rho \vdash \textbf{linputoutputdecl}[] : \rho$$

$$\frac{\rho \overset{\textbf{declaresignal}}{\vdash} \mathrm{IDENT}, [\sigma; \textit{void}] : \rho'}{\rho \vdash \textbf{pure}\,\mathrm{IDENT}, \sigma : \rho'}$$

$$\frac{\rho \vdash \mathrm{TYPE} : \tau \qquad \rho \overset{\textbf{declaresignal}}{\vdash} \mathrm{IDENT}, [\sigma; \tau] : \rho'}{\rho \vdash \textbf{single}\,\mathrm{IDENT}(\mathrm{TYPE}), \sigma : \rho'}$$

$$\frac{\rho \vdash \text{TYPE} : \tau \qquad \rho \overset{\text{isfunc}}{\vdash} \text{FUNC} : \tau \times \tau \rightarrow \tau \qquad \rho \overset{\text{declaresignal}}{\vdash} \text{IDENT}, [\sigma; \tau] : \rho'}{\rho \vdash \text{multiple}\,\text{IDENT}(\text{TYPE}, \text{FUNC}), \sigma : \rho'}$$

$$\rho \vdash \text{relation}\ \text{X}$$

end TC;

end TC1

6.3. Type checking of Statements

program TC2 is

use STRL

set TC is

$$\frac{\rho \vdash \text{LDECL} : \rho' \qquad \rho' \vdash \text{INSTR}}{\rho \vdash \text{module}\,\text{NOM} : \text{LDECL}\ \ \text{INSTR}.}$$

$$\rho \vdash \text{nothing}$$

$$\rho \vdash \text{halt}$$

$$\frac{\rho \vdash \text{INSTR}_1 \qquad \rho \vdash \text{INSTR}_2}{\rho \vdash \text{INSTR}_1 \parallel \text{INSTR}_2}$$

$$\rho \vdash \text{parallele}[]$$

$$\frac{\rho \vdash \text{INSTR}_1 \qquad \rho \vdash \text{INSTR}_2}{\rho \vdash \text{INSTR}_1 ; \text{INSTR}_2}$$

$$\rho \vdash \text{sequence}[]$$

$$\frac{\rho \overset{\text{isproc}}{\vdash} \text{ID} : (\tau_1)(\tau_2) \qquad \rho \vdash \text{LARGS}_1 : \tau_1 \qquad \rho \vdash \text{LARGS}_2 : \tau_2}{\rho \vdash \text{ID}(\text{LARGS}_1)(\text{LARGS}_2)}$$

$$\frac{\rho \overset{\text{issignal}}{\vdash} \text{ID} : [<\_, \rightarrow>; \mathit{void}]}{\rho \vdash \text{emit}\,\text{ID}}$$

$$\frac{\rho \overset{\text{issignal}}{\vdash} \text{ID} : [<\_, \rightarrow>; \tau] \qquad \rho \vdash \text{EXPR} : \tau}{\rho \vdash \text{emit}\,\text{ID}(\text{EXPR})}$$

$$\frac{\rho \vdash \text{INSTR} \qquad \rho \vdash \text{OCCUR}}{\rho \vdash \textbf{do}\ \text{INSTR}\ \textbf{upto}\ \text{OCCUR}}$$

$$\frac{\rho \vdash \text{INSTR} \qquad \rho \vdash \text{OCCUR}}{\rho \vdash \textbf{do}\ \text{INSTR}\ \textbf{uptoeach}\ \text{OCCUR}}$$

$$\frac{\rho \vdash \text{OCCUR} \qquad \rho \vdash \text{INSTR}}{\rho \vdash \textbf{inpresence}\ \text{OCCUR}\ \textbf{do}\ \text{INSTR}\ \textbf{end}}$$

$$\frac{\rho \overset{\textbf{issignal}}{\vdash} \text{ID} : [<\leftarrow, \_>; \_] \qquad \rho \vdash \text{INSTR}}{\rho \vdash \textbf{inabsence}\ \text{ID}\ \textbf{do}\ \text{INSTR}\ \textbf{end}}$$

$$\frac{\rho \vdash \text{INSTR}_1 \qquad \rho \vdash \text{OCCUR} \qquad \rho \vdash \text{INSTR}_2}{\rho \vdash \textbf{do}\ \text{INSTR}_1\ \textbf{watching}\ \text{OCCUR}\ \textbf{abnormal}\ \text{INSTR}_2\ \textbf{end}}$$

$$\frac{\rho \vdash \text{OCCUR} \qquad \rho \vdash \text{INSTR}}{\rho \vdash \textbf{every}\ \text{OCCUR}\ \textbf{do}\ \text{INSTR}\ \textbf{end}}$$

$$\frac{\rho \vdash \text{OCCUR} \qquad \rho \vdash \text{INSTR}}{\rho \vdash \textbf{on}\ \text{OCCUR}\ \textbf{do}\ \text{INSTR}\ \textbf{end}}$$

$$\frac{\rho \vdash \text{OCCUR}}{\rho \vdash \textbf{await}\ \text{OCCUR}}$$

$$\frac{\rho \vdash \text{LSELECTCASE}}{\rho \vdash \textbf{select}\ \text{LSELECTCASE}\ \textbf{end}}$$

$$\frac{\rho \vdash \text{SELECTCASE} \qquad \rho \vdash \text{LSELECTCASE}}{\rho \vdash \text{SELECTCASE}\ \ \text{LSELECTCASE}}$$

$$\rho \vdash \textbf{lselectcase}[]$$

$$\frac{\rho \vdash \text{OCCUR} \qquad \rho \vdash \text{INSTR}}{\rho \vdash \textbf{case}\ \text{OCCUR}\ \textbf{do}\ \text{INSTR}}$$

$$\frac{\rho \overset{\textbf{iscompte}}{\vdash} \text{EXPR} \qquad \rho \overset{\textbf{issignal}}{\vdash} \text{ID} : [<\leftarrow, \_>; \_]}{\rho \vdash \textbf{occurence}(\textbf{compte}(\text{X}, \text{EXPR}), \text{ID}, \textbf{null\_tree})}$$

$$\frac{\rho \overset{\textbf{iscompte}}{\vdash} \text{EXPR} \qquad \rho \overset{\textbf{issignal}}{\vdash} \text{ID}_1 : [<\leftarrow, \_>; \tau] \qquad \rho \vdash \text{ID}_2 : \tau}{\rho \vdash \textbf{occurence}(\textbf{compte}(\text{X}, \text{EXPR}), \text{ID}_1, \textbf{optionalbinding}(\text{ID}_2))}$$

$$\frac{\rho \overset{\textbf{issignal}}{\vdash} \text{ID} : [<\leftarrow, \_>; \_]}{\rho \vdash \textbf{occwithoutcount}(\text{ID}, \textbf{null\_tree})}$$

$$\frac{\rho \overset{\textbf{issignal}}{\vdash} \text{ID}_1 : [<\leftarrow, \_>; \tau] \qquad \rho \vdash \text{ID}_2 : \tau}{\rho \vdash \textbf{occwithoutcount}(\text{ID}_1, \textbf{optionalbinding}(\text{ID}_2))}$$

$$\frac{\rho \vdash \mathrm{INSTR}}{\rho \vdash \mathbf{loop}\ \mathrm{INSTR}\ \mathbf{end}}$$

$$\frac{\rho \vdash \mathrm{EXPR} : \mathit{boolean} \qquad \rho \vdash \mathrm{INSTR}_1 \qquad \rho \vdash \mathrm{INSTR}_2}{\rho \vdash \mathbf{if}\ \mathrm{EXPR}\ \mathbf{then}\ \mathrm{INSTR}_1\ \mathbf{else}\ \mathrm{INSTR}_2}$$

$$\frac{\rho \overset{\mathbf{bloctag}}{\vdash} : \rho' \qquad \rho' \overset{\mathbf{declaretag}}{\vdash} \mathrm{TAG} : \rho'' \qquad \rho'' \vdash \mathrm{INSTR}}{\rho \vdash \mathbf{tag}\ \mathrm{TAG}\ \mathbf{in}\ \mathrm{INSTR}\ \mathbf{end}}$$

$$\frac{\rho \overset{\mathbf{istag}}{\vdash} \mathrm{TAG}}{\rho \vdash \mathbf{exit}\ \mathrm{TAG}}$$

$$\frac{\rho \overset{\mathbf{blocvar}}{\vdash} : \rho' \qquad \rho' \vdash \mathrm{LDCLVAR} : \rho'' \qquad \rho'' \vdash \mathrm{INSTR}}{\rho \vdash \mathbf{var}\ \mathrm{LDCLVAR}\ \mathbf{in}\ \mathrm{INSTR}\ \mathbf{end}}$$

$$\frac{\rho \vdash \mathrm{DCLVAR} : \rho' \qquad \rho' \vdash \mathrm{LDCLVAR} : \rho''}{\rho \vdash \mathrm{DCLVAR}, \mathrm{LDCLVAR} : \rho''}$$

$$\rho \vdash \mathrm{lvariabledecl}[\,] : \rho$$

$$\frac{\rho \vdash \mathrm{TYPE} : \tau \qquad \rho \vdash \mathrm{LVARINIT}, \tau : \rho'}{\rho \vdash \mathrm{onetypevardecls}(\mathrm{LVARINIT}, \mathrm{TYPE}) : \rho'}$$

$$\frac{\rho \vdash \mathrm{VARIABLEINIT}, \tau : \rho' \qquad \rho' \vdash \mathrm{LVARINIT}, \tau : \rho''}{\rho \vdash \mathrm{lvariableinit}[\mathrm{VARIABLEINIT} \cdot \mathrm{LVARINIT}], \tau : \rho''}$$

$$\rho \vdash \mathrm{lvariableinit}[\,], \tau : \rho$$

$$\frac{\rho \overset{\mathbf{declarevar}}{\vdash} \mathrm{ID}, \tau : \rho'}{\rho \vdash \mathrm{ID}, \tau : \rho'}$$

$$\frac{\rho \vdash \mathrm{EXPR} : \tau \qquad \rho \overset{\mathbf{declarevar}}{\vdash} \mathrm{ID}, \tau : \rho'}{\rho \vdash \mathrm{ID} := \mathrm{EXPR}, \mathbf{type}\ \tau : \rho'}$$

$$\frac{\rho \vdash \mathrm{IDENT} : \tau \qquad \rho \vdash \mathrm{EXPR} : \tau}{\rho \vdash \mathrm{IDENT} := \mathrm{EXPR}}$$

$$\frac{\rho \overset{\mathbf{blocsign}}{\vdash} : \rho' \qquad \rho' \vdash \mathrm{LDCLSIGN} : \rho'' \qquad \rho'' \vdash \mathrm{INSTR}}{\rho \vdash \mathbf{signal}\ \mathrm{LDCLSIGN}\ \mathbf{in}\ \mathrm{INSTR}\ \mathbf{end}}$$

$$\frac{\rho \vdash \mathrm{DCLSIGNAL}, <\leftarrow, \rightarrow> : \rho' \qquad \rho' \vdash \mathrm{LDCLIO} : \rho''}{\rho \vdash \mathrm{DCLSIGNAL}, \mathrm{LDCLIO} : \rho''}$$

$$\rho \vdash \mathrm{lsignaldecl}[\,] : \rho$$

$$\frac{\rho \vdash \text{EXPR} : \text{“integer”} \qquad \rho \vdash \text{INSTR}}{\rho \vdash \textbf{repeat}\ \text{EXPR}\ \textbf{times}\ \text{INSTR}\ \textbf{end}}$$

$$\frac{\rho \vdash \text{SIGNDECL} : \rho' \qquad \rho' \vdash \text{INSTR} \qquad \rho' \vdash \text{LFAIL}}{\rho \vdash \textbf{trapfailure}\ \text{SIGNDECL}\ \textbf{in}\ \text{INSTR}\ \textbf{failure}\ \text{LFAIL}}$$

$$\frac{\rho \overset{\textbf{issignal}}{\vdash} \text{ID} : [<\leftarrow, \rightarrow>; \textit{void}]}{\rho \vdash \textbf{failwith}\ \text{ID}}$$

$$\frac{\rho \overset{\textbf{issignal}}{\vdash} \text{ID} : [<\leftarrow, \rightarrow>; \tau] \qquad \rho \vdash \text{EXPR} : \tau}{\rho \vdash \textbf{failwith}\ \text{ID}(\text{EXPR})}$$

$$\frac{\rho \vdash \text{FAILHAND} \qquad \rho \vdash \text{LFAILHAND}}{\rho \vdash \text{FAILHAND}\ \ \text{LFAILHAND}}$$

$$\rho \vdash \textbf{lfailurehandler}[]$$

$$\frac{\rho \vdash \text{OCCUR} \qquad \rho \vdash \text{INSTR}}{\rho \vdash \textbf{failure}\ \text{OCCUR}\ \textbf{do}\ \text{INSTR}\ \textbf{end}}$$

$$\rho \vdash \textbf{copymodule}\ \text{ID}[\text{SUBST}]$$

end TC;

end TC2

6.4. Type checking of Expressions

program TC3 is
use STRL

set TC is

$$\frac{\rho \vdash \text{EXPR}_1 : \textit{integer} \qquad \rho \vdash \text{EXPR}_2 : \textit{integer}}{\rho \vdash \text{EXPR}_1 + \text{EXPR}_2 : \textit{integer}}$$

$$\frac{\rho \vdash \text{EXPR}_1 : \textit{integer} \qquad \rho \vdash \text{EXPR}_2 : \textit{integer}}{\rho \vdash \text{EXPR}_1 - \text{EXPR}_2 : \textit{integer}}$$

$$\frac{\rho \vdash \text{EXPR}_1 : \textit{integer} \qquad \rho \vdash \text{EXPR}_2 : \textit{integer}}{\rho \vdash \text{EXPR}_1 * \text{EXPR}_2 : \textit{integer}}$$

$$\frac{\rho \vdash \text{EXPR}_1 : \textit{integer} \qquad \rho \vdash \text{EXPR}_2 : \textit{integer}}{\rho \vdash \text{EXPR}_1 / \text{EXPR}_2 : \textit{integer}}$$

$$\frac{\rho \vdash \text{EXPR}_1 : \textit{integer} \qquad \rho \vdash \text{EXPR}_2 : \textit{integer}}{\rho \vdash \text{EXPR}_1 ** \text{EXPR}_2 : \textit{integer}}$$

$$\frac{\rho \vdash \text{EXPR} : \textit{integer}}{\rho \vdash\sim \text{EXPR} : \textit{integer}}$$

$$\frac{\rho \vdash \text{EXPR}_1 : \textit{boolean} \qquad \rho \vdash \text{EXPR}_2 : \textit{boolean}}{\rho \vdash \text{EXPR}_1 \text{ or } \text{EXPR}_2 : \textit{boolean}}$$

$$\frac{\rho \vdash \text{EXPR}_1 : \textit{boolean} \qquad \rho \vdash \text{EXPR}_2 : \textit{boolean}}{\rho \vdash \text{EXPR}_1 \text{ and } \text{EXPR}_2 : \textit{boolean}}$$

$$\frac{\rho \vdash \text{EXPR} : \textit{boolean}}{\rho \vdash \text{not } \text{EXPR} : \textit{boolean}}$$

$$\frac{\rho \vdash \text{EXPR}_1 : \textit{integer} \qquad \rho \vdash \text{EXPR}_2 : \textit{integer}}{\rho \vdash \text{compar}(\text{EXPR}_1, \text{COMP}, \text{EXPR}_2) : \textit{boolean}}$$

$$\rho \vdash \text{string } \text{STRING} : \textit{string}$$

$$\rho \vdash \text{natural } \text{NATURAL} : \textit{integer}$$

$$\rho \vdash \text{bool } \text{BOOL} : \textit{boolean}$$

$$\frac{\rho \overset{\text{isfunc}}{\vdash} \text{ID} : \tau_2 \to \tau_1 \qquad \rho \vdash \text{LARGS} : \tau_2}{\rho \vdash \text{ID}(\text{LARGS}) : \tau_1}$$

$$\frac{\rho \overset{\text{isvar}}{\vdash} \text{ident } \text{ID} : \tau}{\rho \vdash \text{ident } \text{ID} : \tau}$$

$$\frac{\rho \vdash \text{EXPR} : \tau_1 \qquad \rho \vdash \text{LEXPR} : \tau_2}{\rho \vdash \text{EXPR}, \text{LEXPR} : \tau_1 \mid \tau_2}$$

$$\rho \vdash \text{lfunctionarg}[] : []$$

$$\frac{\rho \vdash \text{EXPR} : \tau_1 \qquad \rho \vdash \text{LEXPR} : \tau_2}{\rho \vdash \text{EXPR}, \text{LEXPR} : \tau_1 \mid \tau_2}$$

$$\rho \vdash \text{lprocedurearg}[] : []$$

$$\frac{\rho \vdash \text{ID} : \tau_1 \qquad \rho \vdash \text{LID} : \tau_2}{\rho \vdash \text{ID}, \text{LID} : \tau_1 \mid \tau_2}$$

$$\rho \vdash \text{lident}[] : []$$

end TC;

end TC3

6.5. Auxiliary Predicates

program TC4 is

use STRL

set DECLARETYPE is

$$\frac{\pi \overset{\text{ajouter}}{\vdash} \mathrm{ID} \mapsto \mathit{void} : \pi'}{\pi; \mathrm{B}; \mathrm{C}; \mathrm{D}; \mathrm{E}; \mathrm{F} \vdash \text{typedecl}\ \mathrm{ID} : \pi'; \mathrm{B}; \mathrm{C}; \mathrm{D}; \mathrm{E}; \mathrm{F}}$$

end DECLARETYPE;

set DECLARETAG is

$$\frac{\pi \overset{\text{ajouter}}{\vdash} \mathrm{ID} \mapsto \mathit{void} : \pi'}{\mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}; \mathrm{E}; \pi \vdash \text{ident}\ \mathrm{ID} : \mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}; \mathrm{E}; \pi'}$$

end DECLARETAG;

set DECLAREVAR is

$$\frac{\pi \overset{\text{ajouter}}{\vdash} \mathrm{ID} \mapsto \tau : \pi'}{\mathrm{A}; \pi; \mathrm{C}; \mathrm{D}; \mathrm{E}; \mathrm{F} \vdash \text{ident}\ \mathrm{ID}, \tau : \mathrm{A}; \pi'; \mathrm{C}; \mathrm{D}; \mathrm{E}; \mathrm{F}}$$

end DECLAREVAR;

set DECLAREFUNC is

$$\frac{\pi \overset{\text{ajouter}}{\vdash} \mathrm{ID} \mapsto \tau : \pi'}{\mathrm{A}; \mathrm{B}; \pi; \mathrm{D}; \mathrm{E}; \mathrm{F} \vdash \text{ident}\ \mathrm{ID}, \tau : \mathrm{A}; \mathrm{B}; \pi'; \mathrm{D}; \mathrm{E}; \mathrm{F}}$$

end DECLAREFUNC;

set DECLAREPROC is

$$\frac{\pi \overset{\text{ajouter}}{\vdash} \mathrm{ID} \mapsto \tau : \pi'}{\mathrm{A}; \mathrm{B}; \mathrm{C}; \pi; \mathrm{E}; \mathrm{F} \vdash \text{ident}\ \mathrm{ID}, \tau : \mathrm{A}; \mathrm{B}; \mathrm{C}; \pi'; \mathrm{E}; \mathrm{F}}$$

end DECLAREPROC;

set DECLARESIGNAL is

$$\frac{\pi \overset{\textbf{ajouter}}{\vdash} \mathrm{ID} \mapsto \tau : \pi'}{\mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}; \pi; \mathrm{F} \vdash \textbf{ident}\ \mathrm{ID}, \tau : \mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}; \pi'; \mathrm{F}}$$

end DECLARESIGNAL;

set ISTYPE is

$$\frac{\pi \overset{\textbf{firstdecl}}{\vdash} \mathrm{TYPE} \mapsto \mathit{void}}{\pi; \_; \_; \_; \_; \_ \vdash \textbf{type}\ \mathrm{TYPE}}$$

end ISTYPE;

set ISVAR is

$$\frac{\pi \overset{\textbf{firstdecl}}{\vdash} \mathrm{ID} \mapsto \tau' \qquad eq(\tau, \tau')}{\_; \pi; \_; \_; \_; \_ \vdash \textbf{ident}\ \mathrm{ID} : \tau}$$

end ISVAR;

set ISFUNC is

$$\frac{\pi \overset{\textbf{firstdecl}}{\vdash} \mathrm{ID} \mapsto \tau' \qquad eq(\tau, \tau')}{\_; \_; \pi; \_; \_; \_ \vdash \textbf{ident}\ \mathrm{ID} : \tau}$$

end ISFUNC;

set ISPROC is

$$\frac{\pi \overset{\textbf{firstdecl}}{\vdash} \mathrm{ID} \mapsto \tau' \qquad eq(\tau, \tau')}{\_; \_; \_; \pi; \_; \_ \vdash \textbf{ident}\ \mathrm{ID} : \tau}$$

end ISPROC;

set ISSIGNAL is

$$\frac{\pi \overset{\textbf{firstdecl}}{\vdash} \mathrm{ID} \mapsto \tau' \qquad eq(\tau, \tau')}{\_; \_; \_; \_; \pi; \_ \vdash \textbf{ident}\ \mathrm{ID} : \tau}$$

end ISSIGNAL;

set ISTAG is

$$\frac{\pi \overset{\textbf{firstdecl}}{\vdash} \mathrm{TAG} \mapsto \mathit{void}}{-;-;-;-;-;\pi \vdash \textbf{ident}\ \mathrm{TAG}}$$

end ISTAG;

set ISCOMPTE is

$$\rho \vdash \textbf{null\_tree}$$

$$\frac{\rho \overset{\textbf{tc}}{\vdash} \mathrm{EXPR} : \mathit{integer}}{\rho \vdash \mathrm{EXPR}}$$

end ISCOMPTE;

set INITENVIR is

$$\vdash :[\mathit{integer} \cdot \mathit{boolean} \cdot \mathit{string}]; []; []; []; []; []$$

end INITENVIR;

set BLOCVAR is

$$\frac{\pi \overset{\textbf{bloc}}{\vdash} : \pi'}{\mathrm{A}; \pi; \mathrm{C}; \mathrm{D}; \mathrm{E}; \mathrm{F} \vdash : \mathrm{A}; \pi'; \mathrm{C}; \mathrm{D}; \mathrm{E}; \mathrm{F}}$$

end BLOCVAR;

set BLOCSIGN is

$$\frac{\pi \overset{\textbf{bloc}}{\vdash} : \pi'}{\mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}; \pi; \mathrm{F} \vdash : \mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}; \pi'; \mathrm{F}}$$

end BLOCSIGN;

set BLOCTAG is

$$\frac{\pi \overset{\textbf{bloc}}{\vdash} : \pi'}{\mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}; \mathrm{E}; \pi \vdash : \mathrm{A}; \mathrm{B}; \mathrm{C}; \mathrm{D}; \mathrm{E}; \pi'}$$

end BLOCTAG;

end TC4

REFERENCES

[Amber] L. CARDELLI, "The AMBER Programming Language", AT&T Bell Laboratories, 1985

[Berry] G. BERRY, L. COSSERAT, "The ESTEREL synchronous programming language and its mathematical semantics", INRIA Research Report RR 327, September 1984

[CAM] G. COUSINEAU, P. L. CURIEN, M. MAUNY, "The Categorical Abstract Machine", in Functional Languages and Computer Architecture, Lecture Notes in Computer Science, Vol. 201, September 1985

[Cardelli] L. CARDELLI, "Basic Polymorphic Type-checking", Polymorphism, January 1985

[CW] L. CARDELLI, P. WEGNER, "On Understanding Types, Data abstraction, and Polymorphism", Draft, May 1985.

[DKL] V. DONZEAU-GOUGE, G. KAHN, B. LANG, "A complete machine checked definition of a simple programming language using denotational semantics", INRIA Research Report 330, October 1978.

[DM] L. DAMAS, R. MILNER, "Principal type-schemes for functional programs", *Proceedings Principles of Programming Languages 1982,* pp.207–212.

[DMQ] D. B. MACQUEEN, "Modules for standard ML", Private Communication, 1984

[JD] J. DESPEYROUX, "Proof of Translation in Natural Semantics", *Submitted for publication*

[Mentor] V. DONZEAU-GOUGE, G. HUET, G. KAHN, B. LANG, "Programming environments based on structured editors: The Mentor experience" INRIA Research Report no. 26, July 1980

[ML] M. GORDON, R. MILNER, C. WADSWORTH, G. COUSINEAU, G.HUET, L. PAULSON, "The ML Handbook, Version 5.1", INRIA, October 1984

[NS] D. CLÉMENT, J. DESPEYROUX, T. DESPEYROUX, L. HASCOET, G.KAHN, "Natural Semantics on the Computer", INRIA Research Report RR 416, June 1985

[Plotkin] PLOTKIN, G.D., "A Structural Approach to Operational Semantics", DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[Reynolds] J.C. REYNOLDS, "Three Approaches to Type Structure", Lecture Notes in Computer Science, Vol. 185, March 1985

[STML] R. MILNER, "The Dynamic Operational Semantics of Standard ML", University of Edinburgh, April 1985

[TD] TH. DESPEYROUX, "Executable Specification of Static Semantics", in Semantics of Data Types, Lecture Notes in Computer Science, Vol. 173, June 1984

GIPE : CEC 348/A/T4/8
version 2 November 1985

Proposal for an Algebraic Semantics Definition Formalism

Annexe D4.A3 of Deliverable D4 — Second Review —

J.A. Bergstra (University of Amsterdam)
N.W.P. van Diepen (CWI)
J. Heering (CWI)
P.R.H. Hendriks (CWI)
P. Klint (CWI)
A. Verhoog (BSO)

An algebraic definition formalism to be used in the static constraints as well as dynamic semantics sections of formal language definitions is proposed. Test cases are proposed for assessing its suitability for describing various key language features. A large example illustrating the style of specification in this formalism is given.

This annexe is an expanded version of J.A. Bergstra, J.Heering & P. Klint, "Algebraic specification of a simple programming language", Report CS-R8504, CWI, 1985.

1. INTRODUCTION

1.1. General features of the formalism

An algebraic definition essentially consists of a *signature* (itself consisting of declarations of sorts, constants and functions) and a set of *positive conditional equations* over that signature. This basic formalism has been extended in several ways:

- Because initial algebra specification of arbitrary finitely generated semicomputable algebras (data types) requires introduction of *hidden* or *auxiliary* sorts and functions [BT79], an *export* mechanism has been added. Everything not exported is hidden, i.e. invisible from the outside. The export construct constitutes the basic information hiding/abstraction mechanism.
- The formalism allows parameterized (generic) modules, binding of formal parameters to modules (actual parameters), and import of modules into other modules. In general, the way a module is built is given by a *module expression* describing its components as well as the way they are bound to each other.

The modularization mechanisms used are similar or identical to the ones discussed in [KLA83], [W83], [GAU84], [LOE84], or [EM85].

1.2. Semantics

We will always use the initial algebra semantics of (complete) algebraic specifications ([MG85], [EM85]). There are essentially three different ways of defining the meaning of parameterized or otherwise incomplete modules and module composition: In terms of the texts of the modules involved, in terms of their theories or in terms of their (initial) models. The initial model viewpoint does not seem to lend itself to generalization to other types of logic (such as first-order predicate logic). This is a disadvantage because we want to study modularization independently from the type of logic used internally in the modules. The modularization scheme developed must also work for syntax rules and TYPOL (Annexe D4.A2), for instance. We will give an algebraic semantics of module expressions as part of Task T7.

1.3. Compilation of algebraic definitions to executable code

Algebraic definitions are declarative, i.e. non-operational. Automatically compiling them to executable code is beyond the state of the art. These problems will be circumvented by imposing certain regularity restrictions on the equations used [BK82]. If this is done, the resulting specifications can simply be transformed into confluent term rewriting systems. These in turn can readily be compiled to PROLOG programs (see [DE84] and also Deliverable D1).

Our experiences with compiling general, i.e. unrestricted, algebraic specifications to term rewriting systems by means of the Knuth-Bendix algorithm (see for instance [OH80]) have been largely negative. The algorithm behaves unpredictably, often requires human intervention, and is totally unsuitable for large scale applications. We believe this approach to compiling algebraic specifications to be a dead end and do not intend to pursue it any further.

1.4. Integration of the syntax definition formalism

The current formalism allows definition of infix operators in the signature section of a specification and use of infix expressions in the equation section. This is more or less an add-on feature, however, and the next version of the formalism will use a signature section written in the syntax definition formalism proposed in Annexe D4.A1. The latter formalism allows definition of very general expression syntax.

2. THE SPECIFICATION FORMALISM

In this section we give a brief and informal description of the specification formalism. The formalism is based on *signatures* consisting of a set of *sorts* and a set of *functions* over these sorts. A signature combined with a set of equations over that signature and a set of variables occurring in the equations forms a specification (see, for instance, [KLA83]). We will always use the initial algebra semantics of these specifications.

2.1. Syntax of the specification formalism

```
<specification> ::= <module>+ .
<module>        ::= 'module'  <ident>
                    'begin'
                        <parameters>
                        <exports>
                        <imports>
                        <sorts>
                        <functions>
                        <variables>
                        <equations>
                    'end'  <ident> .
<parameters>    ::= [ 'parameters' {<parameter-module>  ','}+ ].
<parameter-module> ::=
                <ident> ['begin'
                                <sorts>
                                <functions>
                            'end' <ident>  ] .
<exports>       ::= [ 'exports' 'begin'
                        <sorts> <functions> 'end'] .
<imports>       ::= [ 'imports' { <module-expression>  ','}+ ] .
<module-expression>
                ::= <ident>
                    ['{'
                        [ 'renamed' 'by' <renames>]
                        ( <ident> 'bound' 'by'  <renames>
                         'to'  <ident> )*
                     '}'] .
<renames>       ::= '[' { <rename>  ',' }* ']' .
<rename>        ::= <fun-ident> [ '->' <fun-ident> ].
<sorts>         ::= [ 'sorts' <ident-list> ] .
<ident-list>    ::= {  <ident>  ',' }+.
<fun-ident-list>::= { <fun-ident>  ',' }+.
<functions>     ::= [ 'functions' <function-list> ] .
<function-list> ::= ( <fun-ident-list>  ':' <fun-type> )+.
<fun-ident>     ::= <ident>  | '_' <operator>  '_' | <operator>  '_' .
<fun-type>      ::= [ <type> ] '->' <out-type> .
<type>          ::= { <type-ident>  '#' }+ .
<out-type>      ::= <type-ident>  | '('  <type> ')' .
<type-ident>    ::= <ident>  | ('*' )+ .
<variables>     ::= [ 'variables' <variable-list> ] .
<variable-list> ::= (  <ident-list> ':' '->' <out-type>  )+.
<equations>     ::= [ 'equations' <cond-equation>+ ] .
<cond-equation> ::= <tag> <equation> [ 'when' <equation-list>].
<tag>           ::= ['[' <ident> ']' ].
<equation-list> ::= { <equation>  ',' }+.
```

```
<equation>      ::= <term> '=' <term>.
<term>          ::= <operator>  <term>  |
                    <primary> [<operator>  <term> ] .
<primary>       ::=  <ident> ['('  <term-list> ')']  |
                    <tuple>  | <string>  | '(' <term>  ')' .
<term-list>     ::= {  <term>  ',' }+.
<tuple>         ::= '<'  <term-list> '>' .
```

2.2. Lexical conventions

The lexical conventions of the specification language are as follows:

1) Identifiers (i.e. `<ident>` in the grammar in the previous section) consist of a non-empty sequence of letters and/or digits with embedded hyphens. For example, `a`, `Z16`, `Very-Long-Identifier` and `6` are legal identifiers, but `-a`, `-` or `a-` are illegal.

2) Strings (i.e. `<string>`) begin and end with a single quote (') and may contain letters, digits and the punctuation marks: (space) " (double quote) `( ) * + , - .  / : ; | = .`

3) Operators (i.e. `<operator>`) are denoted by a sequences of one or more of the following characters: `! @, $, %, ^, &, + ', |, \, ;, ', . ? / .`

4) Comments begin with two hyphens and end with either the end of the line or another pair of hyphens.

2.3. Various aspects of the specification formalism

Our formalism extends the basic algebraic specification formalism based on signatures and sets of equations in several ways. These extensions are discussed in the following subsections.

2.3.1. Prefix and infix operators

(User definable prefix and infix notation will be replaced by fully general user definable syntax once the syntax definition formalism of Annexe D4.A1 and the algebraic definition formalism have been merged.)

Monadic or dyadic functions may be denoted by respectively prefix or infix operators. Operators are denoted by operator-symbols consisting of one or more of the characters specified in the previous paragraph. In the signature, the position of operands of operators is indicated by the underline character (_). For instance,

```
_ + _ : S1 # S2 -> S3
```

defines the infix operator `+` with argument sorts `S1` and `S2` and output sort `S3`. All infix and prefix operators have the same priority. They are just an abbreviation device and can always be replaced by ordinary functions.

2.3.2. Multiple output values

In the signature tuples are allowed as output sorts, i.e. the function

```
f : S1 # S2 -> (S3 # S4)
```

has `S3 # S4` as output sort, this is an ordered sequence with first component of sort `S3` and second component of sort `S4`. In equations, tuples are written as a sequence of terms enclosed by angle brackets, i.e. < and >. It is required that the sorts of the constituents of a tuple are equal to the corresponding components of a tupled output sort in the signature. Tuples can be removed from the specification by introducing new sorts and construction/projection functions for each tupled output sort in the signature. The above tupled output sort (S3 # S4) can, for instance, be removed by introducing the additional sort S5 and the functions make-S5, first-S5, second-S5, as follows:

```
f         : S1 # S2 -> S5
make-S5   : S3 # S4 -> S5
first-S5  : S5      -> S3
second-S5 : S5      -> S4
```

2.3.3. Polymorphism

Functions may be polymorphic, i.e. the same function name may be used to denote different functions with different types, e.g. after defining

```
f : S1 # S2 -> S3
f : S2      -> S2
```

each occurrence of the function symbol f in a term will have to be disambiguated by considering the number and sorts of its arguments.

Definitions of functions may also contain *wild card* sorts, denoted by one or more asterisk characters (*). At the position of a wild card sort, a term of any legal sort is allowed. Wild card sorts are identified by the number of asterisks by which they are denoted. In this way, one can specify the multiple occurrence of the same, but arbitrary, sort. For instance,

```
g : * # S3 # ** # * -> *
```

specifies a function g with first and fourth argument of equal, but arbitrary sort, second argument of sort S3 and third argument of another arbitrary sort which may differ from the sort of the first and fourth argument. The output sort of g is the same as the sort of the first and fourth argument.

We impose some restrictions on polymorphic types which allow us to eliminate all polymorphism from the specification by means of simple textual transformations. It is required that all wild card sorts appearing in the output sorts of a function also appear among its input sorts. This restriction excludes, for instance, polymorphic constants. We also impose the restriction that the sets of input sorts of polymorphic functions are pairwise disjoint. This excludes, for instance,

```
f : * # S2 -> S3
f : S1 # *  -> S3
```

since there is a unifying type S1 # S2 -> S3 in this case.

2.3.4. Module expressions

Module expressions serve the purpose to rename sorts and functions of an existing module or to bind parameters of a module to actual values. The module described by the module expression may then be imported by another module. These three aspects of module expressions are now described in more detail:

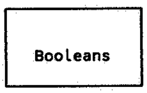
- **Exported names**: Each module may contain an exports clause giving a list of all names of sorts and functions which are exported from the module, i.e. which remain visible when the module is combined with other modules (see below). External names of a module can be renamed by means of the renamed by construct. Currently, all exported names are inherited, i.e. they are also exported by the modules that (directly or indirectly) use the module from which the names were originally exported. This simple scheme has the undesirable property that the number of exported names cannot be controlled. In future versions of the specification formalism, a better mechanism offering more refined control over exported names will be introduced.

- **Parameterization:** In order to make modules more generally usable in different contexts, a form of parameterization is available in the specification language. Parameterization is described by adding one or more `parameters` clauses to a module. Each (formal) parameter is a (possibly incomplete) submodule and contains one or more names of sorts and functions. All these names are formal names which -- in a later stage -- have to be bound to actual ones. This is achieved by the `bound by` construct. Not all parameters of a module have to be bound before it can be imported in another module. Such unbound parameters are *inherited* by the importing module and are indistinguishable from parameters that are specified in the importing module itself.
- **Import of modules:** Import of a module in another module is the fundamental composition operation for modules. It is described by the `imports` clause. The import of module `B` in module `A` is equivalent to constructing a new module `A'` that consists of the unions of the signatures and equations of `A` and `B`. Note that only the exported names of `B` are used for the construction of this union. In the specifications that follow we will -- for reasons of clarity -- frequently import more modules than is strictly necessary.
- **Name identification:** When modules are combined the problem arises how multiple declarations of names should be interpreted. For identification of names we therefore adopt the *origin principle*:
  1) names with identical spelling and type, originating from the same module are equal,
  2) names with identical spelling and type but different origin are forbidden.

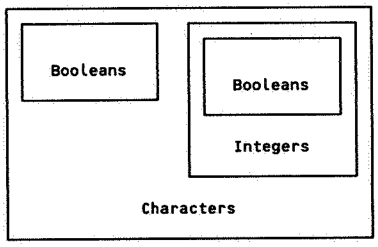
This scheme allows the multiple inclusion of the same module (via different routes), but forbids collisions of names with identical spelling and type, originating from different modules.

2.4. Structure diagrams

The overall modular structure of specifications will be illustrated by *structure diagrams*. Each module is represented by a rectangular box. The name of each module is shown at the bottom of its box. For example, module `Booleans` does not import any other modules and is represented by:

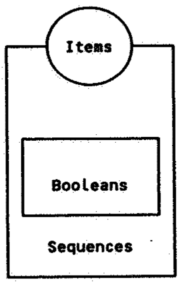


All modules imported by a module M are represented by structure diagrams inside the box representing M. For nested structure diagrams levels of detail may be suppressed to gain space. For example, `Characters` imports `Booleans` and `Integers` (which in its turn also imports `Booleans`) and is represented by:

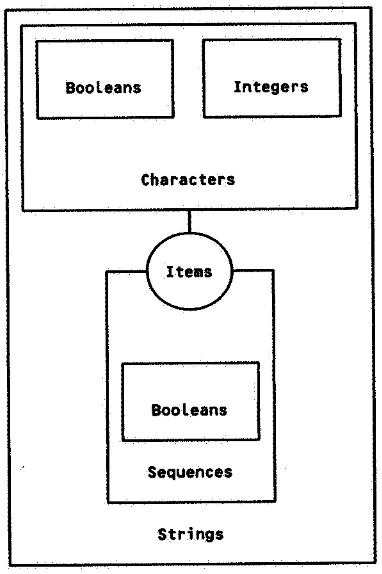


All parameters of a module are represented by ellipses carrying the name of the parameter.

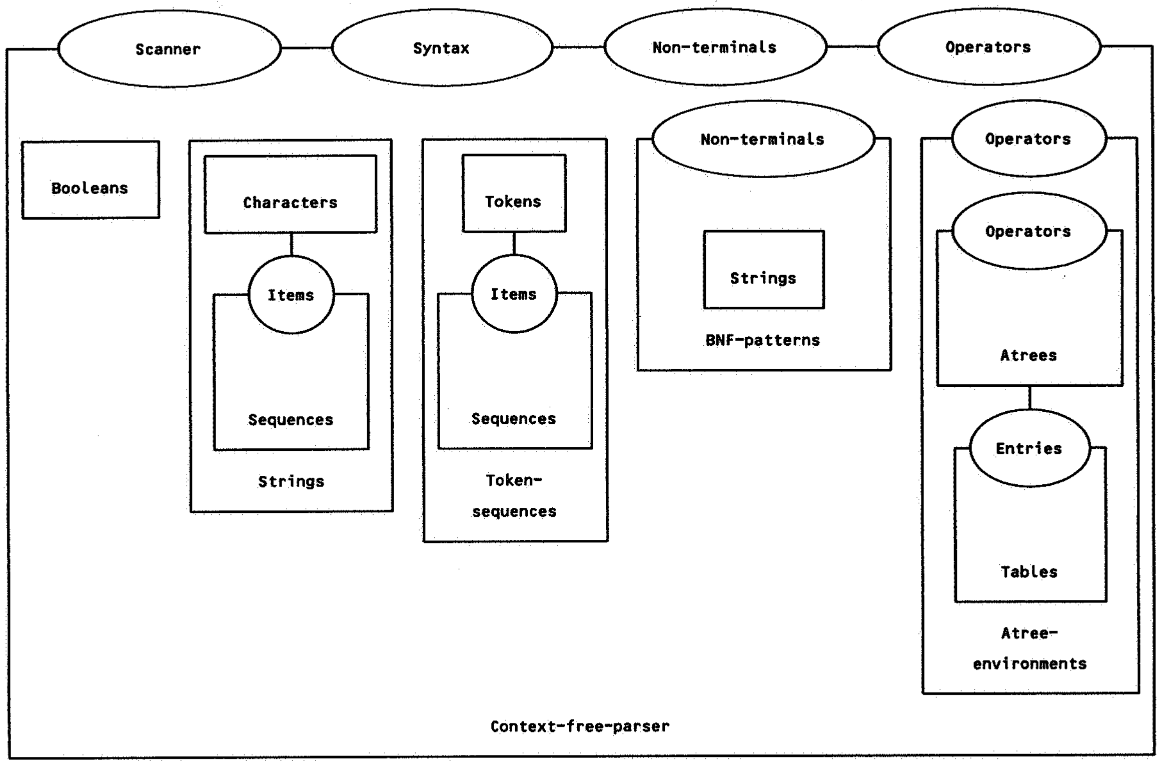
For example, Sequences, which has parameter Items and imports Booleans, is represented by:



The binding of a formal parameter is represented by a line joining the formal parameter and the module to which it is bound. For example, Strings are defined by binding the parameter Items of Sequences to Characters. The corresponding structure diagram is:



Unbound, inherited parameters are -- not yet very satisfactorily -- represented in structure diagrams by repeating the inherited parameter as a parameter of the module that inherits it. For example, Context-free-parser has formal parameters Scanner and Syntax and imports, among others, BNF-patterns with unbound parameter Non-terminals and Atree-environments with unbound parameter Operators. This is represented by the following diagram:



All structure diagrams appearing in this paper have been generated automatically; they were derived from the *text* of the specification.

3. SELECTION OF TEST CASES

Many of the following test cases not only test the algebraic formalism but also the syntax definition formalism proposed in Annexe D4.A1. We assume both formalisms will have been integrated at the time work on the test cases starts.

3.1. Basic data types

Most problems (like error handling, definition of proper syntax and use of parameterized modules) already manifest themselves in very simple settings, so the most frequently used data types - although relatively simple - are not to be underestimated as test cases.

Examples of basic data types we will use as test cases are:

- Booleans
- Integers
- (Parameterized) sequences, multisets, and sets
- (Parameterized) stacks, queues, lists, and tables
- Labeled trees.

None of these data types is uniquely defined. Most of them have many variants. In specifying them we will emphasize:

- Proper error handling.
- Introduction of proper notations for constants and functions by means of the syntax definition formalism of Annexe D4.A1.

A library of basic data types which can be used by all language definitions will be constructed.

3.2. Definition of a simple language and language system

We propose two test cases involving a very simple programming language called PICO:

(1) Algebraic definition of a PICO system involving
   - lexical analysis
   - syntactic analysis
   - construction of abstract syntax trees
   - checking of static semantics
   - definition of dynamic semantics.

This test case has already been specified (4).

(2) Definition of PICO as it would be accepted by the kind of environment generator we have in mind (Deliverable D5). This definition will consist of three sections and will use the syntax definition formalism of Annexe D4.A1 for defining the PICO concrete and abstract syntax and a restricted algebraic formalism (1.3) for defining the PICO static and dynamic semantics.

3.3. A language with jumps

Jump statements are among the most difficult language constructs to specify and hence description of a programming language containing them is a good test case for a specification formalism. A simple example of such a language is the language SMALL extended with goto-statements. The SMALL family of languages has been used by Gordon [GOR79] to illustrate the use of denotational semantics for describing various language features. Other members of the SMALL family will allow us to study the algebraic specification of various restricted versions of the jump construct.

Furthermore, since SMALL has already been specified using denotational semantics, it will allow us to compare algebraic and denotational specifications. Of special interest will be the effects of modules (not present in denotational semantics) and higher order functions (not present in algebraic semantics).

3.4. Type inference and higher order functions

The majority of the TYPOL test cases proposed in Annexe D4.A2 will also be tried algebraically, so that the suitability of the algebraic formalism for defining static constraints and dynamic semantics can be compared with that of TYPOL. In particular, we will specify the type checking and the dynamic semantics of (a suitably chosen) subset of ML, and compare the resulting specifications with the corresponding TYPOL versions (Annexe D4.A2). Because it has polymorphic types and type from context inference, ML is currently perhaps the most challenging language from the viewpoint of type checking. In defining ML type checking we will base ourselves on the theory of polymorphic type checking in Damas & Milner [DM82] and the specification of a typechecker for a part of ML written in Standard ML by Cardelli [CAR85]. ML is a fully high order functional language and it will be interesting to see whether a satisfactory algebraic semantics of it can be given.

3.5. Parallelism

The language POOL-T [AM85], a parallel object-oriented language allowing the dynamic creation of processes (objects), will serve as a further test case for the algebraic formalism. An inference rule semantics for (a subset of) POOL-T has recently been given by America, De Bakker, Kok & Rutten [ABKR85].

4. ALGEBRAIC DEFINITION OF A SIMPLE LANGUAGE SYSTEM

4.1. Introduction

4.1.1. General

The following specification describes in detail all necessary steps from entering a program in the simple programming language PICO in its textual form to computing its value. The specification has been made more general than strictly necessary. A major part of it does not depend on any specific properties of PICO but is equally usable for definitions of other programming languages.

In this case study, we have *not* considered the specification of errors and exceptions, for two reasons:

(1) We want to concentrate first on the basic functionality and the alternatives for modularization of the system to be designed; specifying error situations would obscure the design and would probably double its size.

(2) Specification of errors within the algebraic framework has not yet been solved satisfactorily and requires separate research.

The entire PICO system constitutes a semi-computable algebra in the sense of [BT79].

4.1.2. Relations with previous research

Many people have carried out similar exercises, for instance [GP81], the work of the CIP project in München has been partly devoted to the topic of algebraic specifications of programming languages. Further, several people have worked on the related topic of algebraic compiler specification, for instance Bothe [BO81], Ganzinger [GAN82] and Gaudel [GAU80].

4.1.3. Verification and validation

It is a major problem to get insight in the correctness of a given formal specification. The algebraic specification method provides a relatively simple formalism with unambiguous semantics, but constructing proofs of correctness remains as difficult as ever. We have the following opinion on this matter:

(1) We consider algebraic specifications as the highest level of specification available, i.e. there is no "super high level" specification against which the correctness of the algebraic specification can be proved.

(2) Specifications can only be *validated* against informal requirements.

(3) A proof will be required that some program correctly implements a given algebraic specification. This will involve verification of the translation steps between an algebraic specification and its implementation.

4.1.4. Conclusions

Our conclusions can be summarized as follows:

(1) The specifications as presented in the body of this paper are in our opinion satisfactory. The techniques developed for specifying various aspects of our toy programming language can also be used in the specifications of other -- more realistic -- languages.

(2) Polymorphism was found to be convenient -- though not indispensable -- for shortening the specifications and making them more readable. Conditional equations were essential for the modeling of partial functions. They also tended to shorten several parts of the specification. The primitive abbreviation scheme used for introducing infix operators was unsatisfactory. The way in which we have to treat integer and string constants is also clumsy. The syntax definition formalism of Annexe D4.A1 has been developed to solve this problem but became available only after this specification was finished.

(3) The algebraic specification techniques have been of considerable heuristic value in understanding how the specification should (could) be modularized. However, the various modularization techniques (such as import and parameterization) are not orthogonal. It will be important to develop sound heuristics about which technique is to be used where.

(4) Structure diagrams (a high-level graphical notation described in section 2.4) are a considerable aid in finding the proper modularization of a specification.

(5) In view of the size of the specification it was necessary to implement some simple tools for consistency checking. We have implemented a checker for the syntax and type correctness of specifications and generators for structure diagrams and cross reference tables. For the development of larger specifications it will be necessary to have more sophisticated editing facilities, such as a syntax-directed editor with incremental type checking. The question will have to be addressed how user definable syntax can be handled by such an editor.

4.1.5. Perspectives for further research

During this exercise we have identified the following areas that need further clarification (Task T7):

(1) Treatment of errors and exceptions.

(2) Multiple export signatures per module.

(3) More flexible export rules with which the number of exported names can be minimized.

(4) Parameterization of modules and formulation of constraints on parameters.

(5) More explicit specification of inherited parameters.

(6) Heuristic rules for proper modularization.

(7) Further development of structure diagrams.

(8) Techniques and tools for creating, modifying, maintaining and incremental checking of algebraic specifications.

(9) Techniques and tools for transforming algebraic specifications into executable prototypes.

4.2. Informal definition of the language PICO

The language PICO is extremely simple. It is essentially the language of while-programs. A program consists of declarations followed by statements. All variables occurring in the statements have to be declared to be either of type integer or of type string. Statements may be assignment statements, if-statements and while-statements. Expressions may be a single identifier, integer addition or string concatenation.

At the lexical level, PICO programs consist of a sequence of lexical items separated by layout. Lexical items are keywords, identifiers, integer and string constants and punctuation marks. The lexical grammar for PICO is:

```
<lexical-stream>      ::= <lexical-item> <lexical-stream> |
                          <lexical-item> .
<lexical-item>        ::= <optional-layout>
                          (<keyword-or-id> |
                           <integer-constant> |
                           <string-constant>  |
                           <literal>) .
<optional-layout>     ::= <layout> | <empty>.
<keyword-or-id>       ::= 'begin' | 'end' | 'declare' | 'integer' |
                          'string' | 'if' | 'then' | 'else' | 'fi' |
                          'while' | 'do' | 'od' |
                          <id> .
<id>                  ::= <letter> <id-chars> .
<id-chars>            ::= <id-char> <id-chars> | <empty> .
<id-char>             ::= <letter> | <digit> .

<integer-constant>    ::= <digit> <digits>
<digits>              ::= <digit> <digits> | <empty> .

<string-constant>     ::= <quote> <string-tail> .
<string-tail>         ::= <any-char-but-quote> <string-tail> | <quote> .

<quote>               ::= '"' .

<any-char-but-quote> ::= <letter> | <digit> | <literal> | <layout> .
<literal>             ::= '(' | ')' | '+' | '-' | ';' | ',' |
                          '||' | ':' | ':=' .
<letter>              ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
                          'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
                          'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
                          'v' | 'w' | 'x' | 'y' | 'z' |
                          'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
                          'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
                          'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' |
                          'V' | 'W' | 'X' | 'Y' | 'Z' .
<digit>               ::= '0' | '1' | '2' | '3' | '4' |
                          '5' | '6' | '7' | '8' | '9' .
<layout>              ::= ' '  | <newline> | <tab> .
```

Here, `<newline>` and `<tab>` are assumed to be primitive notions corresponding to, respectively, the newline character and the tabulation.

The concrete syntax of PICO is:

```
<pico-program>  ::= 'begin' <decls> <series> 'end'  .
<decls>         ::= 'declare' <id-type-list> ';' .
<id-type-list>  ::= <id> ':' <type> (<empty> | ',' <id-type-list>)
<type>          ::= 'integer' | 'string' .
<series>        ::= <empty> | <stat> (<empty> | ';' <series>)
<stat>          ::=  <assign> | <if> | <while> .
<assign>        ::= <id> ':=' <exp> .
<if>            ::= 'if' <exp> 'then' <series>
                             'else' <series> 'fi'
<while>         ::= 'while' <exp> 'do' <series> 'od' .
<exp>           ::= <id> | <integer-constant> | <string-constant> |
                    <plus> | <conc> | '(' <exp> ')' .
<plus>          ::= <exp> '+' <exp> .
<conc>          ::= <exp> '||' <exp> .
<empty>         ::= '' .
```

The non-terminals `<id>`, `<integer-constant>` and `<string-constant>` are defined in the lexical grammar given above and represent identifiers, integer constants and string constants respectively.

There are two overall static semantic constraints on programs:

1) All identifiers occurring in a program should have been declared and their use should be compatible with their declaration. More precisely, this means that all `<id>`s occurring in an `<assign>` or an `<exp>` should have been declared, i.e. should occur in some `<id-type>` in the `<id-type-list>` of the `<decls>`-part of the PICO-program, and that the type of `<id>`s should be compatible with the expressions in which they occur.

2) The `<exp>` occurring in an `<if>`- or `<while>`-statement should be of type integer.

A type can be given to `<exp>`s depending on their syntactic form:

- if an `<exp>` consists of an `<id>`, that `<id>` should have been declared and the type of the `<exp>` is the same as the type of the `<id>` in its declaration;
- an `<exp>` consisting of an `<integer-constant>` has type integer;
- an `<exp>` consisting of a `<string-constant>` has type string;
- an `<exp>` consisting of a `<plus>` has type integer;
- an `<exp>` consisting of a `<conc>` has type string.

Given this notion of types of `<exp>`s, the static semantic constraints can be formulated in more detail:

- The `<exp>`s occurring in a `<plus>` should be of type integer;
- The `<exp>`s occurring in a `<conc>` should be of type string;
- The `<id>` and `<exp>` that occur in an `<assign>` should have the same type.
- The `<exp>`s that occur in `<if>` and `<while>` should have type integer.

The dynamic semantics of PICO are straightforward except that

1) integer variables are initialized with value 0,
2) string variables are initialized with "" (empty string),
3) the `<exp>` in an `<if>` or `<while>` is assumed to be true if its value is unequal to 0.

4.3. Elementary data types

As a prerequisite for the PICO specification some elementary data types are defined in this chapter, specifications are given for:

- **Booleans** (4.3.1): truth values `true` and `false` with functions `and`, `or`, `not` and the polymorphic function `if`.
- **Integers** (4.3.2): natural numbers with constants `0`, `1` and `10` and functions `succ` (successor), `add` (addition), `mul` (multiplication), `eq` (equality of integers), `less` (less than), `lesseq` (less than or equal), `greater` (greater than) and `greatereq` (greater than or equal).
- **Characters** (4.3.3): the alphabet consists of constants for letters, digits, and punctuation marks. The functions `eq` (equality of characters), `ord` (ordinal number of character in the alphabet), `is-letter` (is character a letter?), `is-upper` (is character an upper case letter?), `is-lower` (is character a lower case letter?) and `is-digit` (is character a digit?) are defined on them.
- **Sequences** (4.3.4): linear lists of items. Sequences are parameterized with the data type of the items. The only constant is `null`, the empty sequence. The following functions are defined: `eq` (equality of sequences), `seq` (combine item with sequence), `conc` (concatenate two sequences) and `conv-to-seq` (convert an item to a sequence).
- **Strings** (4.3.5): sequences of characters, i.e. sequences with items bound by characters. The only constant is `null-string`, the empty string. The following functions are defined: `eq` (equality of strings), `seq` (combine character with a string), `conc` (concatenate two strings), `string` (convert a character to a string) and `str-to-int` (convert a string to an integer).
- **Tables** (4.3.6): mapping from strings to entries, where entries are a parameter. The only constant is `null-table`, the empty table. The following functions are defined: `table` (add new entry to table), `lookup` (searches for an entry in a table), `delete` (deletes an entry from a table) and `eq` (equality of tables).

4.3.1. Booleans

4.3.1.a. Gobal description

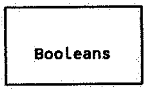
Booleans are truth values `true` and `false` with functions `and`, `or`, `not` and the polymorphic function `if` (see section 2 for a discussion of polymorphism).

Apart from the `if`-function, this is the simplest initial algebra specification of the Booleans. It contains only closed equations. Note that, for instance, the equation

```
not(not(x)) = x
```

is not derivable by equational logic from the axioms given, although it is valid in the initial model. Adding this equation to `Booleans`, does not affect the initial model, but only causes an increase in the power of the specification in the sense that more of the (open) equations valid in the initial model can be derived from the specification by equational logic. See Annexe D5.A1 for a discussion of this subject.

4.3.1.b. Structure diagram



4.3.1.c. Specification

```
module Booleans
begin
   exports
        begin
           sorts         BOOL
           functions
                true    :                     -> BOOL
                false   :                     -> BOOL
                or      : BOOL # BOOL         -> BOOL
                and     : BOOL # BOOL         -> BOOL
                not     : BOOL                -> BOOL
                if      : BOOL # * # *        -> *
        end

   variables
        x, y    : -> *

   equations

   [1]    or(true, true)          = true
   [2]    or(true, false)         = true
   [3]    or(false, true)         = true
   [4]    or(false, false)        = false

   [5]    and(true, true)         = true
   [6]    and(true, false)        = false
   [7]    and(false, true)        = false
   [8]   and(false, false)      = false

   [9]   not(true)              = false
   [10]  not(false)             = true

   [11]  if(true, x, y)         = x
   [12]  if(false, x, y)        = y

end Booleans
```

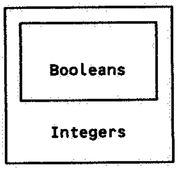
4.3.2. Integers

4.3.2.a. Global description

Integers as defined here are in fact natural numbers with constants `0`, `1` and `10` and functions `succ` (successor), `add` (addition), `mul` (multiplication), `eq` (equality), `less` (less than), `lesseq` (less than or equal), `greater` (greater than) and `greatereq` (greater than or equal).

The equations for the constants `1` and `10` are not very satisfactory. Clearly, a mechanism is needed for defining a shorthand notation for *all* integer constants. In section 4.3.5.a this subject is discussed in connection with string constants.

4.3.2.b. Structure diagram



4.3.2.c. Specification

```
module Integers
begin

   exports
        begin
           sorts         INTEGER

           functions
                0        :                          -> INTEGER
                1        :                          -> INTEGER
                10       :                          -> INTEGER
                succ     : INTEGER                  -> INTEGER
                add      : INTEGER # INTEGER        -> INTEGER
                mul      : INTEGER # INTEGER        -> INTEGER
                eq       : INTEGER # INTEGER        -> BOOL
                less     : INTEGER # INTEGER        -> BOOL
                lesseq   : INTEGER # INTEGER        -> BOOL
                greater  : INTEGER # INTEGER        -> BOOL
                greatereq: INTEGER # INTEGER        -> BOOL
        end

   imports Booleans

   variables
        x, y, z  : -> INTEGER

   equations

   [13]   1                       = succ(0)
   [14]   10                      = succ(succ(succ(succ(succ(succ(
                                        succ(succ(succ(succ(0))))))))))
```

```
    [15]   add(x, 0)              = x
    [16]   add(x, succ(y))        = succ(add(x, y))

    [17]   mul(x, 0)              = 0
    [18]   mul(x, succ(y))        = add(x, mul(x, y))

    [19]   eq(x, x)               = true
    [20]   eq(x, y)               = eq(y, x)
    [21]   eq(succ(x), succ(y))   = eq(x, y)
    [22]   eq(0, succ(x))         = false

    [23]   less(x, 0)             = false
    [24]   less(0, succ(x))       = true
    [25]   less(succ(x), succ(y))= less(x, y)

    [26]   lesseq(x, y)           = or(less(x, y), eq(x, y))

    [27]   greater(x, y)          = not(lesseq(x, y))

    [28]   greatereq(x, y)        = or(greater(x, y), eq(x, y))

end Integers
```

4.3.3. Characters

4.3.3.a. Global description

The alphabet of characters consists of constants for letters, digits, and punctuation marks. The functions `eq` (equality), `ord` (ordinal number of character in the alphabet), `is-letter` (is character a letter?), `is-upper` (is character an upper case letter?), `is-lower` (is character a lower case letter?) and `is-digit` (is character a digit?) are defined on them.

Two observations can be made about this specification. First, one may notice that the absence of integer constants forces us two write equations of the form

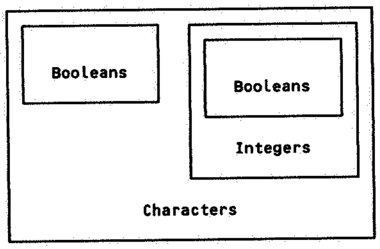
```
ord(char-3)      = succ(ord(char-2))
```

instead of the more natural form

```
ord(char-3)      = 3.
```

Secondly, it is clear that some abbreviation mechanism is needed for specifications that contain many constants as is the case here. At the expense of additional complexity of the specification, this could have been achieved by defining characters in two stages: first, a basic alphabet is defined which consists only of lower case letters and a hyphen; next, this basic alphabet is used to generate all constants for the full alphabet. Names of constants are then only allowed to contain symbols from the basic alphabet, i.e. `char-upper-case-a` instead of `char-A`.

4.3.3.b. Structure diagram



4.3.3.c. Specification

```
module Characters
begin

   exports
        begin
           sorts          CHAR
           functions
                eq               : CHAR # CHAR    -> BOOL
                is-upper         : CHAR           -> BOOL
                is-lower         : CHAR           -> BOOL
                is-letter        : CHAR           -> BOOL
                is-digit         : CHAR           -> BOOL
                ord              : CHAR           -> INTEGER

                char-0           :                -> CHAR
```

```
char-1          :          -> CHAR
char-2          :          -> CHAR
char-3          :          -> CHAR
char-4          :          -> CHAR
char-5          :          -> CHAR
char-6          :          -> CHAR
char-7          :          -> CHAR
char-8          :          -> CHAR
char-9          :          -> CHAR

char-ht         :          -> CHAR        -- tab --
char-nl         :          -> CHAR        -- new line --
char-space      :          -> CHAR        -- space --
char-quote      :          -> CHAR        -- " --
char-lpar       :          -> CHAR        -- ( --
char-rpar       :          -> CHAR        -- ) --
char-times      :          -> CHAR        -- * --
char-plus       :          -> CHAR        -- + --
char-comma      :          -> CHAR        -- , --
char-minus      :          -> CHAR        -- - --
char-point      :          -> CHAR        -- . --
char-slash      :          -> CHAR        -- / --
char-bar        :          -> CHAR        -- | --
char-equal      :          -> CHAR        -- = --
char-colon      :          -> CHAR        -- : --
char-semi       :          -> CHAR        -- ; --

char-A          :          -> CHAR
char-B          :          -> CHAR
char-C          :          -> CHAR
char-D          :          -> CHAR
char-E          :          -> CHAR
char-F          :          -> CHAR
char-G          :          -> CHAR
char-H          :          -> CHAR
char-I          :          -> CHAR
char-J          :          -> CHAR
char-K          :          -> CHAR
char-L          :          -> CHAR
char-M          :          -> CHAR
char-N          :          -> CHAR
char-O          :          -> CHAR
char-P          :          -> CHAR
char-Q          :          -> CHAR
char-R          :          -> CHAR
char-S          :          -> CHAR
char-T          :          -> CHAR
char-U          :          -> CHAR
char-V          :          -> CHAR
char-W          :          -> CHAR
char-X          :          -> CHAR
char-Y          :          -> CHAR
char-Z          :          -> CHAR

char-a          :          -> CHAR
```

```
                char-b          :                   -> CHAR
                char-c          :                   -> CHAR
                char-d          :                   -> CHAR
                char-e          :                   -> CHAR
                char-f          :                   -> CHAR
                char-g          :                   -> CHAR
                char-h          :                   -> CHAR
                char-i          :                   -> CHAR
                char-j          :                   -> CHAR
                char-k          :                   -> CHAR
                char-l          :                   -> CHAR
                char-m          :                   -> CHAR
                char-n          :                   -> CHAR
                char-o          :                   -> CHAR
                char-p          :                   -> CHAR
                char-q          :                   -> CHAR
                char-r          :                   -> CHAR
                char-s          :                   -> CHAR
                char-t          :                   -> CHAR
                char-u          :                   -> CHAR
                char-v          :                   -> CHAR
                char-w          :                   -> CHAR
                char-x          :                   -> CHAR
                char-y          :                   -> CHAR
                char-z          :                   -> CHAR

    end

imports Booleans, Integers

variables
    c, c1, c2       :                   -> CHAR

equations

[29]    ord(char-0)             = 0
[30]    ord(char-1)             = succ(ord(char-0))
[31]    ord(char-2)             = succ(ord(char-1))
[32]    ord(char-3)             = succ(ord(char-2))
[33]    ord(char-4)             = succ(ord(char-3))
[34]    ord(char-5)             = succ(ord(char-4))
[35]    ord(char-6)             = succ(ord(char-5))
[36]    ord(char-7)             = succ(ord(char-6))
[37]    ord(char-8)             = succ(ord(char-7))
[38]    ord(char-9)             = succ(ord(char-8))

[39]    ord(char-ht)            = succ(ord(char-9))
[40]    ord(char-nl)            = succ(ord(char-ht))
[41]    ord(char-space)         = succ(ord(char-nl))
[42]    ord(char-quote)         = succ(ord(char-space))
[43]    ord(char-lpar)          = succ(ord(char-quote))
[44]    ord(char-rpar)          = succ(ord(char-lpar))
[45]    ord(char-times)         = succ(ord(char-rpar))
[46]    ord(char-plus)          = succ(ord(char-times))
[47]    ord(char-comma)         = succ(ord(char-plus))
```

```
[48]   ord(char-minus)       = succ(ord(char-comma))
[49]   ord(char-point)       = succ(ord(char-minus))
[50]   ord(char-slash)       = succ(ord(char-point))
[51]   ord(char-bar)         = succ(ord(char-slash))
[52]   ord(char-equal)       = succ(ord(char-bar))
[53]   ord(char-colon)       = succ(ord(char-equal))
[54]   ord(char-semi)        = succ(ord(char-colon))

[55]   ord(char-A)           = succ(ord(char-semi))
[56]   ord(char-B)           = succ(ord(char-A))
[57]   ord(char-C)           = succ(ord(char-B))
[58]   ord(char-D)           = succ(ord(char-C))
[59]   ord(char-E)           = succ(ord(char-D))
[60]   ord(char-F)           = succ(ord(char-E))
[61]   ord(char-G)           = succ(ord(char-F))
[62]   ord(char-H)           = succ(ord(char-G))
[63]   ord(char-I)           = succ(ord(char-H))
[64]   ord(char-J)           = succ(ord(char-I))
[65]   ord(char-K)           = succ(ord(char-J))
[66]   ord(char-L)           = succ(ord(char-K))
[67]   ord(char-M)           = succ(ord(char-L))
[68]   ord(char-N)           = succ(ord(char-M))
[69]   ord(char-O)           = succ(ord(char-N))
[70]   ord(char-P)           = succ(ord(char-O))
[71]   ord(char-Q)           = succ(ord(char-P))
[72]   ord(char-R)           = succ(ord(char-Q))
[73]   ord(char-S)           = succ(ord(char-R))
[74]   ord(char-T)           = succ(ord(char-S))
[75]   ord(char-U)           = succ(ord(char-T))
[76]   ord(char-V)           = succ(ord(char-U))
[77]   ord(char-W)           = succ(ord(char-V))
[78]   ord(char-X)           = succ(ord(char-W))
[79]   ord(char-Y)           = succ(ord(char-X))
[80]   ord(char-Z)           = succ(ord(char-Y))

[81]   ord(char-a)           = succ(ord(char-Z))
[82]   ord(char-b)           = succ(ord(char-a))
[83]   ord(char-c)           = succ(ord(char-b))
[84]   ord(char-d)           = succ(ord(char-c))
[85]   ord(char-e)           = succ(ord(char-d))
[86]   ord(char-f)           = succ(ord(char-e))
[87]   ord(char-g)           = succ(ord(char-f))
[88]   ord(char-h)           = succ(ord(char-g))
[89]   ord(char-i)           = succ(ord(char-h))
[90]   ord(char-j)           = succ(ord(char-i))
[91]   ord(char-k)           = succ(ord(char-j))
[92]   ord(char-l)           = succ(ord(char-k))
[93]   ord(char-m)           = succ(ord(char-l))
[94]   ord(char-n)           = succ(ord(char-m))
[95]   ord(char-o)           = succ(ord(char-n))
[96]   ord(char-p)           = succ(ord(char-o))
[97]   ord(char-q)           = succ(ord(char-p))
[98]   ord(char-r)           = succ(ord(char-q))
[99]   ord(char-s)           = succ(ord(char-r))
[100]  ord(char-t)           = succ(ord(char-s))
```

```
  [101]  ord(char-u)              = succ(ord(char-t))
  [102]  ord(char-v)              = succ(ord(char-u))
  [103]  ord(char-w)              = succ(ord(char-v))
  [104]  ord(char-x)              = succ(ord(char-w))
  [105]  ord(char-y)              = succ(ord(char-x))
  [106]  ord(char-z)              = succ(ord(char-y))

  [107]  eq(c1, c2)               = eq(ord(c1), ord(c2))
  [108]  is-upper(c)              = and(greatereq(ord(c), ord(char-A)),
                                        lesseq(ord(c), ord(char-Z)))
  [109]  is-lower(c)              = and(greatereq(ord(c), ord(char-a)),
                                        lesseq(ord(c), ord(char-z)))
  [110]  is-digit(c)              = and(greatereq(ord(c), ord(char-0)),
                                        lesseq(ord(c), ord(char-9)))
  [111]  is-letter(c)             = or(is-upper(c), is-lower(c))

end Characters
```

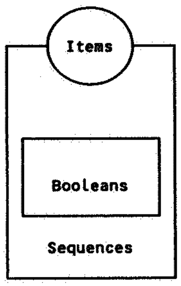
4.3.4. Sequences

4.3.4.a. Global description

Sequences are linear lists of items; they are parameterized with the data type of the items. The only constant is `null`, the empty sequence. The following functions are defined: `eq` (equality), `seq` (combine item with sequence), `conc` (concatenate two sequences) and `conv-to-seq` (convert an item to a sequence).

Note that the function `eq` in the above specification is polymorphic.

4.3.4.b. Structure diagram



4.3.4.c. Specification

```
module Sequences
begin

   parameters Items
        begin
           sorts         ITEM
           functions
                eq : ITEM # ITEM -> BOOL

        end Items

   exports
        begin
           sorts         SEQ
           functions
                null            :                -> SEQ
                seq             : ITEM # SEQ     -> SEQ
                conc            : SEQ # SEQ      -> SEQ
                eq              : SEQ # SEQ      -> BOOL
                conv-to-seq     : ITEM           -> SEQ
        end

   imports Booleans

   variables
        s, s1, s2       : -> SEQ
        it, it1, it2    : -> ITEM

   equations
```

```
    [112]  conc(s, null)                 = s
    [113]  conc(null, s)                 = s
    [114]  conc(seq(it, s1), s2)         = seq(it, conc(s1, s2))

    [115]  eq(s, s)                      = true
    [116]  eq(s1, s2)                    = eq(s2, s1)
    [117]  eq(null, seq(it, s))          = false
    [118]  eq(seq(it1,s1), seq(it2,s2))  = and(eq(it1,it2), eq(s1,s2))

    [119]  conv-to-seq(it)               = seq(it, null)

end Sequences
```

4.3.5. Strings

4.3.5.a. Global description

Strings are sequences of characters, i.e. `Sequences` with `Items` bound to `Characters`. The only constant is `null-string`, the empty string. The following functions are defined: `eq` (equality), `seq` (combine character with a string), `conc` (concatenate two strings), `string` (convert a character to a string) and `str-to-int` (convert a string to an integer).

In the case of the data type string there is an urgent need for a short hand notation for string constants. The PICO specification would become unreadable without it. We will therefore use an, *ad hoc*, convenient notation for string constants to denote the terms generated by the module `Strings`, e.g. the term

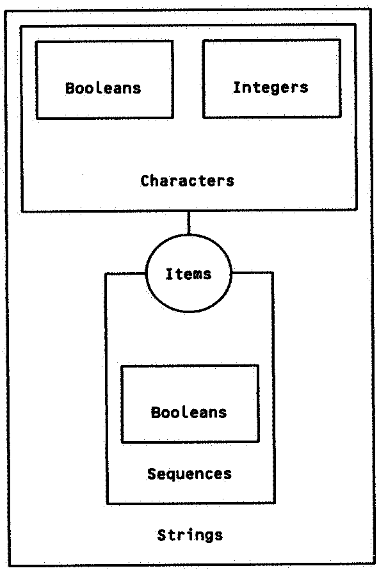
```
seq(char-a, seq(char-b, null-string))
```

will be written as

```
"ab".
```

The empty string, i.e. the constant `null-string`, will be written as "".

4.3.5.b. Structure diagram



4.3.5.c. Specification

```
module Strings
begin

   exports
      begin
        functions
           str-to-int : STRING -> INTEGER
      end

   imports Sequences
```

```
                { renamed by
                        [ SEQ -> STRING,
                          null -> null-string,
                          conv-to-seq -> string]
                  Items bound by
                        [ ITEM -> CHAR,
                          eq -> eq]
                        to Characters
                }

   variables
        c        :-> CHAR
        str      :-> STRING

   equations

   [120] str-to-int(seq(c, str)) = if(eq(str, null-string),
                                      ord(c),
                                      add(mul(ord(c), 10), str-to-int(str)))
   [121] str-to-int(null-string) = 0

end Strings
```

4.3.6. Tables

4.3.6.a. Global description

Tables are mappings from strings to entries, where entries are a parameter. The only constant is `null-table`, the empty table. The following functions are defined: `table` (add new entry to table), `lookup` (searches for an entry in a table), `delete` (deletes an entry from a table) and `eq` (equality of tables).

Note that adding a pair (`name`, `error-entry`) to a table has the somewhat strange, but harmless, effect that

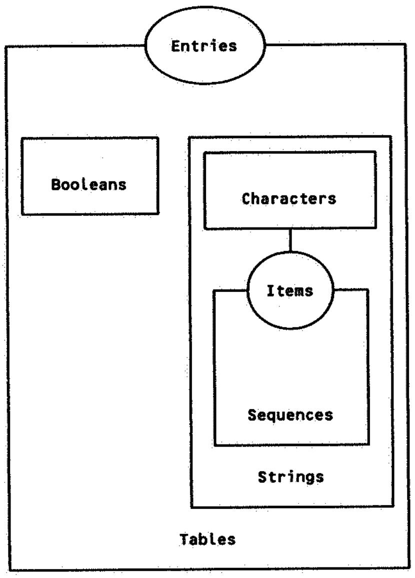
```
lookup(name, table(name, error-entry, tbl1)) = <true, error-entry>
```

and that

```
lookup(name, null-table) = <false, error-entry>.
```

Only in the first case `name` occurs in the table, but except for the `true/false` flag, the same value is delivered.

4.3.6.b. Structure diagram



4.3.6.c. Specification

```
module Tables
begin

   parameters Entries
        begin
           sorts        ENTRY
           functions
                error-entry :              -> ENTRY
                eq          : ENTRY # ENTRY -> BOOL
        end Entries
```

```
exports
     begin
        sorts TABLE
        functions

             null-table      :                        -> TABLE
             table           : STRING # ENTRY # TABLE -> TABLE
             lookup          : STRING # TABLE         -> (BOOL # ENTRY)
             delete          : STRING # TABLE         -> TABLE
             eq              : TABLE # TABLE          -> BOOL
     end

imports Booleans, Strings

variables
     name, name1, name2      : -> STRING
     e, e1, e2               : -> ENTRY
     tbl, tbl1, tbl2         : -> TABLE
     found                   : -> BOOL

equations

[122]  table(name1, e1, table(name2, e2, tbl))
                             = if(eq(name1,name2),
                                  table(name1, e1, tbl),
                                  table(name2, e2, table(name1, e1, tbl)))

[123]  lookup(name, null-table)
                             = <false, error-entry>

[124]  lookup(name1, table(name2, e, tbl))
                             = if(eq(name1, name2),
                                  <true, e>,
                                  lookup(name1, tbl))
[125]  delete(name, null-table)
                             = null-table
[126]  delete(name1, table(name2, e, tbl))
                             = if(eq(name1, name2),
                                  delete(name1, tbl),
                                  table(name2, e, delete(name1, tbl)))


[127]  eq(tbl1, tbl2)        = eq(tbl2, tbl1)

[128]  eq(null-table, null-table)
                             = true

[129]  eq(null-table, table(name, e, tbl))
                             = false

[130]  eq(table(name, e1, tbl1), tbl2)
                             = if(and(found, eq(e1,e2)),
                                  eq(delete(name, tbl1), delete(name, tbl2)),
                                  false)
                               when <found, e2> = lookup(name, tbl2)
```

```
end Tables
```

4.4. Context-free parsing

In this chapter the problem will be addressed how a context-free grammar can be specified within the algebraic framework and how the parsing process is to be described. A syntactic definition of a language can globally be subdivided in definitions for:

lexical syntax:
which defines the tokens of the language, i.e., keywords, identifiers, punctuation marks, etc.

context-free syntax:
which defines the concrete form of programs, i.e. the sequences of tokens that constitute a legal program.

abstract syntax:
which defines the abstract tree structure underlying the concrete (textual) form of programs. All further operations on programs may be defined as operations on the abstract syntax tree (see 4.5).

In this chapter, we will define a parser (`Context-free-parser`, see 4.4.4) which is parameterized with a lexical scanner and a grammar describing the concrete syntax and the construction rules for abstract syntax trees. The parsing problem is decomposed as follows:

1) Lexical analysis is delegated to a `Scanner` (a parameter of `Context-free-parser`), which transforms an input string into a sequence of lexical tokens (4.4.1). A token is a pair of strings: the first describes the lexical category of the token, the second gives the string value of the token, e.g. `token("identifier", "xyz")` or `token("integer-constant", "35")`.

2) Abstract syntax trees are represented by the data type `Atrees`. Rules for the construction of abstract syntax trees are part of the grammar for a given language. The essential function is `build`, which specifies for each non-terminal how certain (named) components of the syntax rule have to be combined into an abstract syntax tree (4.4.2).

3) BNF patterns (4.4.3) are introduced to allow the description of arbitrary context-free grammars. The main functions and operators introduced are `t` (indicates a terminal in the grammar), `n` (indicates a non-terminal), `+` (sequential composition of components of a grammar rule), and `|` (alternation). A grammar constructed by means of these operators can later be bound to the parameter `Syntax` of `Context-free-parser`.

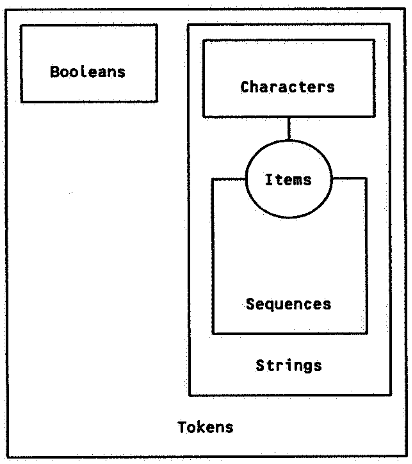
4) Actual parsing is described in `Context-free-parser` (4.4.4). This module has four parameters of which two are inherited from imported modules. The parameters `Scanner` and `Syntax` define the interface with the lexical scanner and with the concrete syntax and abstract syntax. `Context-free-parser` imports `BNF-patterns` (inheriting the unbound parameter `Non-terminals`) and `Atree-environments` (inheriting the unbound parameter `Operators`). `Context-free-parser` describes a parser which is driven by the BNF operators occurring in `Syntax`. Currently, we require that `Syntax` satisfies the LL(1) restrictions.

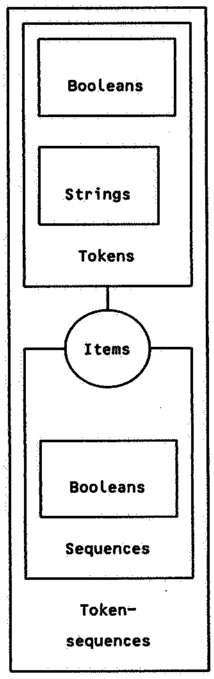
4.4.1. Interface with lexical scanner

4.4.1.a. Global description

Lexical analysis transforms an input string into a sequence of lexical tokens. A token is a pair of strings: the first describes the lexical category of the token, the second gives the string value of the token, e.g. `token("identifier", "xyz")` or `token("integer-constant", "35")`. In this section, the data types `Tokens` and `Token-sequences` are defined.

4.4.1.b. Structure diagrams





4.4.1.c. Specification

```
module Tokens
begin
   exports
        begin
           sorts TOKEN
           functions
                token    : STRING # STRING        -> TOKEN
                eq       : TOKEN # TOKEN          -> BOOL
        end

   imports Booleans, Strings

   variables
        s1, s2, s3, s4 : -> STRING

   equations

   [131]  eq(token(s1, s2), token(s3, s4))      = and(eq(s1, s3), eq(s2, s4))

end Tokens

module Token-sequences
begin
   imports Sequences
                { renamed by
                        [ SEQ -> TOKEN-SEQUENCE,
                          null -> null-token-sequence ]
                  Items bound by
                        [ ITEM -> TOKEN,
                          eq -> eq ]
                        to Tokens
                }

end Token-sequences
```

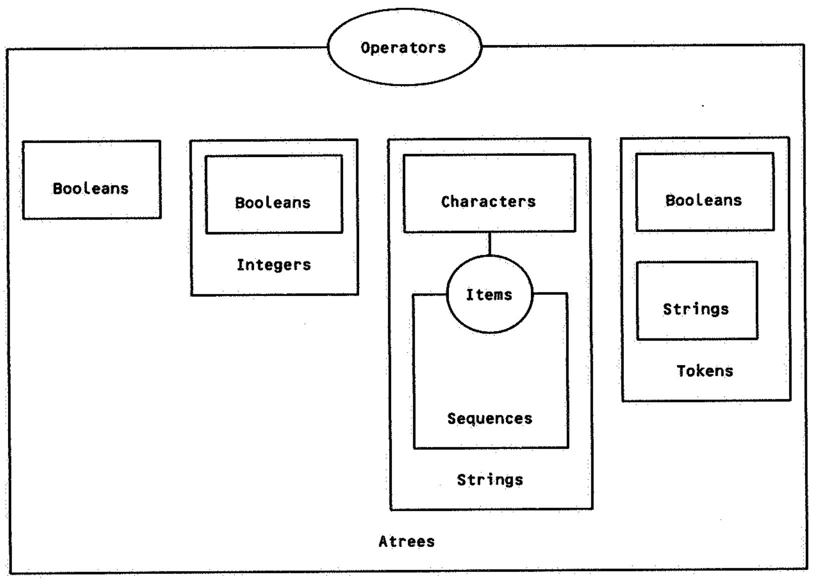
4.4.2. Interface with abstract syntax tree constructor

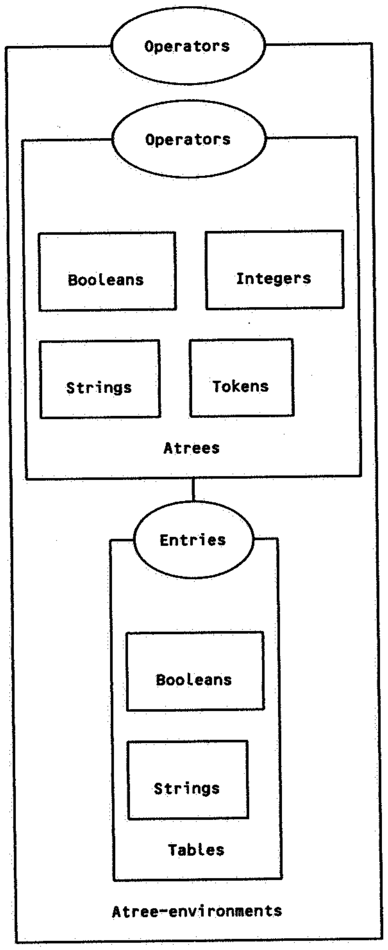
4.4.2.a. Global description

Abstract syntax trees are defined by the data type `Atrees`. Abstract syntax trees are essentially labelled trees whose nodes consist of an operator, indicating the construction operator of the node, and zero or more abstract syntax trees as sons. `Atrees` has one parameter `Operators`, which defines the interface to the set of operators for constructing abstract syntax trees. Conversion functions are defined for the common cases that the leaves of the abstract syntax tree consist of `Strings`, `Integers` or `Tokens`.

The construction process for abstract syntax trees as described in 4.4.4 uses the notion of environments of abstract syntax trees, i.e. tables which map strings onto abstract syntax trees. This notion is realized by the data type `Atree-environments`. Note that the parameter `Operators` of `Atrees` is inherited by `Atree-environments`.

4.4.2.b. Structure diagrams





4.4.2.c. Specification

```
module Atrees
begin

   parameters
      Operators
        begin
           sorts OPERATOR

           functions
                eq: OPERATOR # OPERATOR -> BOOL
        end Operators

   exports
        begin
           sorts ATREE

           functions
                error-atree    :                             -> ATREE
                null-atree     :                             -> ATREE
                atree          : OPERATOR                    -> ATREE
```

```
            atree         : OPERATOR # ATREE                   -> ATREE
            atree         : OPERATOR # ATREE # ATREE           -> ATREE
            atree         : OPERATOR # ATREE # ATREE # ATREE -> ATREE
            string-atree  : STRING                             -> ATREE
            integer-atree : INTEGER                            -> ATREE
            lexical-atree : TOKEN                              -> ATREE
            eq            : ATREE # ATREE                      -> BOOL
    end

imports Booleans, Integers, Strings, Tokens

variables
    c, c1, c2 :-> OPERATOR
    a, a1, a2, a3, a4 :-> ATREE
    b1, b2, b3, b4 :-> ATREE
    s, s1, s2 :-> STRING
    n, n1, n2 :-> INTEGER
    t, t1, t2 :-> TOKEN

equations

[132]  eq(a1, a2)                            = eq(a2, a1)

[133]  eq(null-atree, null-atree)            = true
[134]  eq(null-atree, error-atree)           = false
[135]  eq(null-atree, atree(c))              = false
[136]  eq(null-atree, atree(c, a))           = false
[137]  eq(null-atree, atree(c, a1, a2))      = false
[138]  eq(null-atree, atree(c, a1, a2, a3))  = false
[139]  eq(null-atree, string-atree(s))       = false
[140]  eq(null-atree, integer-atree(n))      = false
[141]  eq(null-atree, lexical-atree(t))      = false

[142]  eq(error-atree, error-atree)           = true
[143]  eq(error-atree, atree(c))              = false
[144]  eq(error-atree, atree(c, a))           = false
[145]  eq(error-atree, atree(c, a1, a2))      = false
[146]  eq(error-atree, atree(c, a1, a2, a3)) = false
[147]  eq(error-atree, string-atree(s))       = false
[148]  eq(error-atree, integer-atree(n))      = false
[149]  eq(error-atree, lexical-atree(t))      = false

[150]  eq(atree(c1), atree(c2))              = eq(c1, c2)
[151]  eq(atree(c1), atree(c2, a1))          = false
[152]  eq(atree(c1), atree(c2, a1, a2))      = false
[153]  eq(atree(c1), atree(c2, a1, a2, a3))  = false
[154]  eq(atree(c), string-atree(s))         = false
[155]  eq(atree(c), integer-atree(n))        = false
[156]  eq(atree(c), lexical-atree(t))        = false

[157]  eq(atree(c1, a1), atree(c2, b1))      = and(eq(c1, c2), eq(a1, b1))
[158]  eq(atree(c1, a1), atree(c2, b1, b2))  = false
[159]  eq(atree(c1, a1), atree(c2, b1, b2, b3))
                                             = false
[160]  eq(atree(c1, a1), string-atree(s))    = false
```

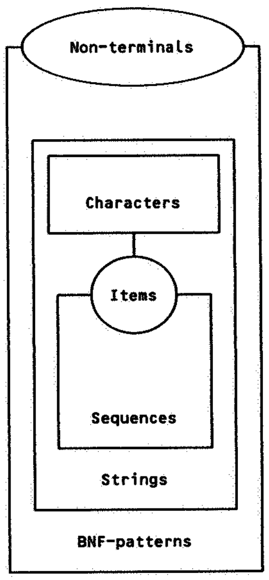
```
[161]  eq(atree(c1, a1), integer-atree(n))    = false
[162]  eq(atree(c1, a1), lexical-atree(t))    = false


[163]  eq(atree(c1, a1, a2), atree(c2, b1, b2))
                                              = and(eq(c1, c2),
                                                    and(eq(a1, b1),
                                                        eq(a2, b2)))
[164]  eq(atree(c1, a1, a2), atree(c2, b1, b2, b3))
                                              = false
[165]  eq(atree(c1, a1, a2), string-atree(s))= false
[166]  eq(atree(c1, a1, a2), integer-atree(n))
                                              = false
[167]  eq(atree(c1, a1, a2), lexical-atree(t))
                                              = false


[168]  eq(atree(c1, a1, a2, a3), atree(c2, b1, b2, b3))
                                              = and(eq(c1, c2),
                                                    and(eq(a1, b1),
                                                        and(eq(a2, b2),
                                                            eq(a3, b3))))
[169]  eq(atree(c1, a1, a2, a3), string-atree(s))
                                              = false
[170]  eq(atree(c1, a1, a2, a3), integer-atree(n))
                                              = false
[171]  eq(atree(c1, a1, a2, a3), lexical-atree(t))
                                              = false



[172]  eq(string-atree(s1), string-atree(s2))= eq(s1, s2)
[173]  eq(string-atree(s), integer-atree(n)) = false
[174]  eq(string-atree(s), lexical-atree(t)) = false

[175]  eq(integer-atree(n1), integer-atree(n2))
                                              = eq(n1, n2)
[176]  eq(integer-atree(n), lexical-atree(t))= false

[177]  eq(lexical-atree(t1), lexical-atree(t2))
                                              = eq(t1, t2)

end Atrees

module Atree-environments
begin
   exports
        begin
           functions
                _^_ : STRING # ATREE-ENV -> ATREE
        end

   imports Tables
        { renamed by
                [ TABLE -> ATREE-ENV,
                  null-table -> null-atree-env]
          Entries bound by
                [ ENTRY -> ATREE,
                  eq -> eq,
                  error-entry -> error-atree]
                to Atrees
        }
   variables
        s :-> STRING
        e :-> ATREE-ENV
        f :-> BOOL
        v :-> ATREE

   equations

   [178]  s ^ e          = v
                           when <f, v> = lookup(s, e)

end Atree-environments
```

4.4.3. BNF patterns

4.4.3.a. Global description

BNF patterns are introduced to allow the description of arbitrary context-free grammars. The main functions and operators introduced are `t` (indicates a terminal in the grammar), `n` (indicates a non-terminal), `lexical` (indicates a lexical item), `+` (sequential composition of components of a grammar rule), and `|` (alternation). The functions `t`, `n` and `lexical` have two variants: the variant with one argument indicates respectively a terminal, non-terminal or lexical item; the variant with two arguments also associates a name with the syntaxctic notion. These names can later be used to refer to the abstract syntax tree which is the result of parsing the given syntactic notion. An actual grammar constructed with these operators can be bound to the parameter `Syntax` of `Context-free-parser`. Examples of grammars using this notation are the lexical syntax (4.5.2.2) and concrete syntax (4.5.3.2) of PICO.

4.4.3.b. Structure diagram



4.4.3.c. Specification

```
module BNF-patterns
begin

   parameters
      Non-terminals
        begin
           sorts NON-TERMINAL
        end Non-terminals

   exports
        begin
           sorts PATTERN

           functions
```

```
                  _ + _    : PATTERN # PATTERN      -> PATTERN
                  _ | _    : PATTERN # PATTERN      -> PATTERN
                  t        : STRING                 -> PATTERN
                  t        : STRING # STRING        -> PATTERN
                  n        : NON-TERMINAL           -> PATTERN
                  n        : NON-TERMINAL # STRING  -> PATTERN
                  lexical  : STRING                 -> PATTERN
                  lexical  : STRING # STRING        -> PATTERN
                  null-pattern :                    -> PATTERN

          end

     imports Strings

end BNF-patterns
```

4.4.4. Context-free parser

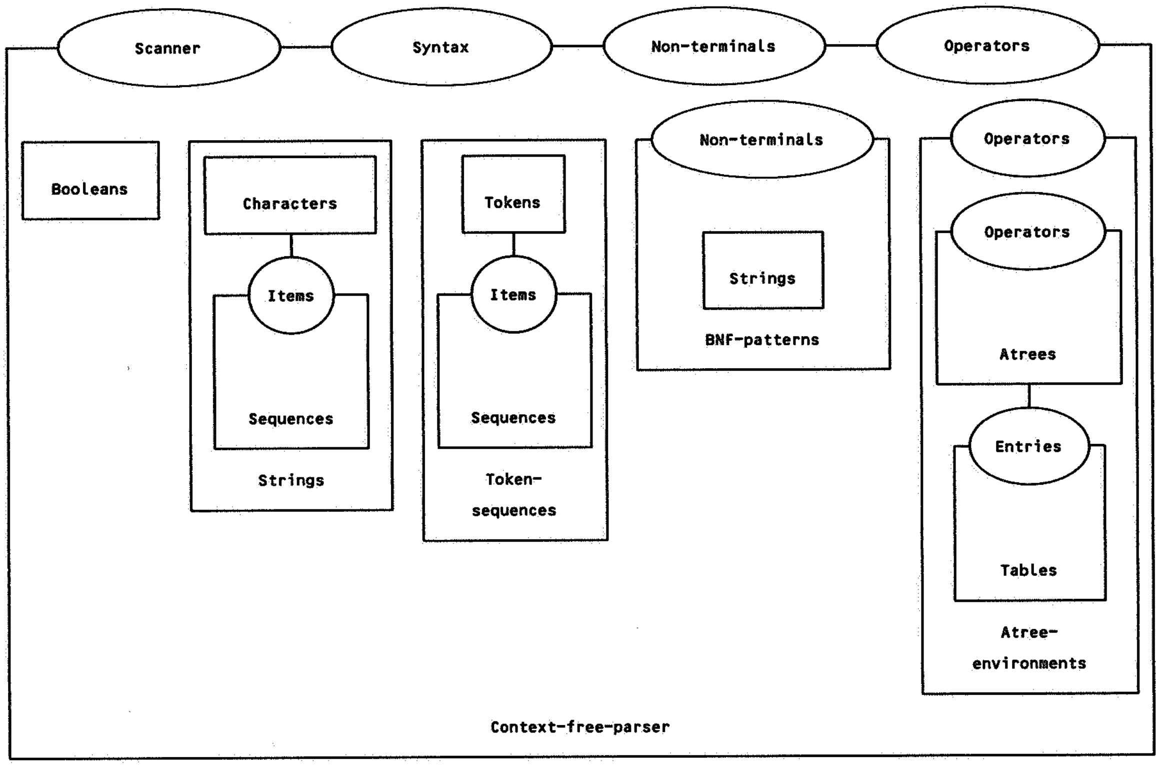
4.4.4.a. Global description

`Context-free-parser` describes the actual parsing process. It has four parameters of which two are inherited from imported modules. Parameter `Scanner` defines the interface with the lexical scanner, i.e. the function `scan` which converts input strings to `Token-sequences`. Parameter `Syntax` defines the interface with the rules of the syntax (function `rule`) and with the rules for constructing abstract syntax trees (function `build`). `Context-free-parser` imports `BNF-patterns` (inheriting the unbound parameter `Non-terminals`, which defines the interface with the set of non-terminals of the syntax) and `Atree-environments` (inheriting the unbound parameter `Operators`, which defines the interface with the set of construction operators for the abstract syntax).

`Context-free-parser` describes a parser for the language described by the syntax rules. The equations in `Context-free-parser` describe for each type of BNF operator the conditions under which (a part of) the input `Token-sequence` is acceptable. The BNF operator `n` (non-terminal) uses the function `rule` from parameter `Syntax` to associate a pattern with a non-terminal. Acceptance of a part of the input is expressed by constructing an `Atree-environment` consisting of named `Atrees`. Acceptance of a non-terminal is expressed by the function `build` from `Syntax` for that non-terminal .

Currently, we require that the syntax satisfies the LL(1) restrictions. This simplifies the definition of `Context-free-parser` considerably: in the definition given below only *one* abstract syntax tree has to be constructed instead of a *set* of abstract syntax trees as would be necessary in the case of an ambiguous input string if the grammar were not LL(1).

4.4.4.b. Structure diagram



4.4.4.c. Specification

```
module Context-free-parser
begin

   parameters
      Scanner
        begin
           functions
                scan : STRING -> TOKEN-SEQUENCE
        end Scanner,

      Syntax
        begin

           functions

                rule    : NON-TERMINAL                    -> PATTERN
                build   : NON-TERMINAL # ATREE-ENV        -> ATREE

        end Syntax

   exports
        begin
           functions
                parse   : NON-TERMINAL # STRING           -> ATREE
        end

   imports Booleans, Strings, Token-sequences, BNF-patterns, Atree-environments

   functions

        parse-rule: NON-TERMINAL # TOKEN-SEQUENCE
                                 -> (BOOL # ATREE # TOKEN-SEQUENCE)
        parse-pat : PATTERN # TOKEN-SEQUENCE # ATREE-ENV
                                 -> (BOOL # ATREE-ENV # TOKEN-SEQUENCE)

   variables
        x                        : -> NON-TERMINAL
        p, p1, p2                : -> PATTERN
        env, env1, env2          : -> ATREE-ENV
        atree, atree1, atree2    : -> ATREE
        s, tail, tail1, tail2    : -> TOKEN-SEQUENCE
        id, val, str, lextype    : -> STRING
        r, r1, r2                : -> BOOL

   equations

   [179]  parse(x, str)          = if(and(r, eq(tail, null-token-sequence)),
                                      atree,
                                      error-atree)
                                   when  <r, atree, tail> = parse-rule(x, scan(str))

   [180]  parse-rule(x, s)       = if(r, < true, build(x, env), tail >,
                                 < false, error-atree, tail >)
                           when  <r, env, tail> =
                                 parse-pat(rule(x), s, null-atree-env)

[181]  parse-pat(null-pattern, s, env)
                          = <true, env, s>

[182]  parse-pat(p1 + p2, s, env1)
                          = if(r, parse-pat(p2, tail, env2),
                                 < false, env2, tail >)
                            when <r, env2, tail> = parse-pat(p1, s, env1)

[183]  parse-pat(p1 | p2, s, env)
                          = if(not(r1),
                               < r2, env2, tail2 >,
                               if(not(r2),
                                  < r1, env1, tail1 >,
                                  < false, env, s >))
                            when  <r1, env1, tail1> = parse-pat(p1, s, env),
                                  <r2, env2, tail2> = parse-pat(p2, s, env)

[184]  parse-pat(n(x), s, env)
                          = <r, env, tail>
                            when <r, atree, tail> = parse-rule(x, s)

[185]  parse-pat(n(x,id), s, env)
                          = if(r, < true, table(id, atree, env), tail >,
                                  < false, env, tail >)
                            when <r, atree, tail> = parse-rule(x, s)

[186]  parse-pat(t(str), seq(token(lextype, val), s), env)
                          = if(and(eq(str, val),
                                   or(eq(lextype, "keyword"),
                                      eq(lextype, "literal"))),
                               < true, env, s>,
                               < false, env, s> )

[187]  parse-pat(t(str), null-token-sequence, env)
                          = if(eq(str, null-string),
                               <true, env, null-token-sequence>,
                               <false, env, null-token-sequence>)

[188]  parse-pat(t(str, id), seq(token(lextype, val), s), env)
                          = if(and(eq(str, val),
                                   or(eq(lextype, "keyword"),
                                      eq(lextype, "literal"))),
                               < true,
                                 table(id,
                                       lexical-atree(token(lextype,str)),
                                       env),
                                 s>,
                               < false, env, s> )

[189]  parse-pat(t(str, id), null-token-sequence, env)
                          = if(eq(str, null-string),
                                      < true,
                                        table(id,
                                              lexical-atree(token("literal","")),
                                              env),
                                        null-token-sequence>,
                                      <false, env, null-token-sequence>)

    [190]  parse-pat(lexical(str), seq(token(lextype, val), s), env)
                                 = if(eq(lextype, str),
                                      < true, env, s >,
                                      < false, env, s> )

    [191]  parse-pat(lexical(str), null-token-sequence, env)
                                 = <false, env, null-token-sequence>

    [192]  parse-pat(lexical(str,id), seq(token(lextype, val), s), env)
                                 = if(eq(lextype, str),
                                      < true,
                                        table(id,
                                              lexical-atree(token(lextype,val)),
                                              env),
                                        s >,
                                      < false, env, s> )

    [193]  parse-pat(lexical(str,id), null-token-sequence, env)
                                 = <false, env, null-token-sequence>

end Context-free-parser
```

4.5. Algebraic specification of PICO

After the preparations in the previous chapters, the following steps are still needed to obtain a complete specification of PICO:

1) The notions of *types* and *values* in PICO programs have to be formalized (4.5.1).
2) The lexical syntax of PICO has to be specified. This is done by constructing a lexical scanner on the basis of `Context-free-parser` as defined in the previous chapter (4.5.2).
3) The concrete syntax of PICO and the rules for the construction of abstract syntax trees have to be specified. This is accomplished by a *second* application of `Context-free-parser` (4.5.3).
4) The static semantics of PICO has to be specified, defining certain constraints on programs, i.e. constraints that do not depend on input data. For instance, in a "legal" program all variables should have been declared, all expressions should be type consistent, etc. This is described in 4.5.4.
5) Dynamic semantics of PICO has to be specified, defining the meaning of a program, i.e. the relation between its input and output data (4.5.5).
6) All the above components of the PICO specification have to be combined into one *PICO system* (4.5.6).

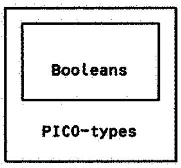
4.5.1. Types and values

4.5.1.1. Types

4.5.1.1.a. Global description

The data type `PICO-types` defines the allowed types in PICO programs, i.e. integers and strings. An additional `error-type` is introduced for describing typing errors.

4.5.1.1.b. Structure diagram



4.5.1.1.c. Specification

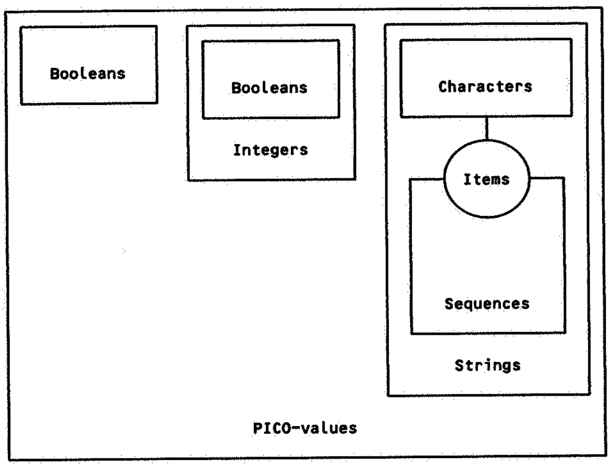
```
module PICO-types
begin
   exports
        begin

           sorts PICO-TYPE

           functions
                integer-type   :                         -> PICO-TYPE
                string-type    :                         -> PICO-TYPE
                error-type     :                         -> PICO-TYPE
                eq             : PICO-TYPE # PICO-TYPE -> BOOL
        end

   imports Booleans

   variables
        x, y : -> PICO-TYPE

   equations

   [194]  eq(x,  x)                          = true
   [195]  eq(x, y)                           = eq(y, x)
   [196]  eq(integer-type, string-type)      = false
   [197]  eq(integer-type,  error-type)      = false
   [198]  eq(string-type,  error-type)       = false

end PICO-types
```

4.5.1.2. Values

4.5.1.2.a. Global description

The data type `PICO-values` defines the allowed values as they may occur during the execution of PICO programs, i.e. integers and strings. An additional `error-value` is introduced for describing values that are the result of evaluating erroneous programs. Note that there is no integer or string corresponding to `error-value`. Two conversion functions are defined for converting `Integers` and `Strings` into `PICO-values`.

4.5.1.2.b. Structure diagram



4.5.1.2.c. Specification

```
module PICO-values
begin

   exports
        begin
           sorts PICO-VALUE

           functions
                error-value     :                              -> PICO-VALUE
                pico-value      : INTEGER                      -> PICO-VALUE
                pico-value      : STRING                       -> PICO-VALUE
                eq              : PICO-VALUE # PICO-VALUE      -> BOOL
        end

   imports Booleans, Integers, Strings

   variables
        x, y             : -> PICO-VALUE
        int, int1, int2  : -> INTEGER
        str, str1, str2  : -> STRING

   equations

   [199]  eq(x, x)                                    = true
```

```
  [200]  eq(x, y)                                    = eq(y, x)
  [201]  eq(pico-value(int1),  pico-value(int2))     = eq(int1, int2)
  [202]  eq(pico-value(int),   pico-value(str))      = false
  [203]  eq(pico-value(int),   error-value)          = false
  [204]  eq(pico-value(str1),  pico-value(str2))     = eq(str1, str2)
  [205]  eq(pico-value(str),   error-value)          = false

end PICO-values
```

4.5.2. Lexical syntax

The lexical syntax describes the lexical tokens that may occur in a PICO program. We construct a lexical scanner for PICO by means of `Context-free-parser`:

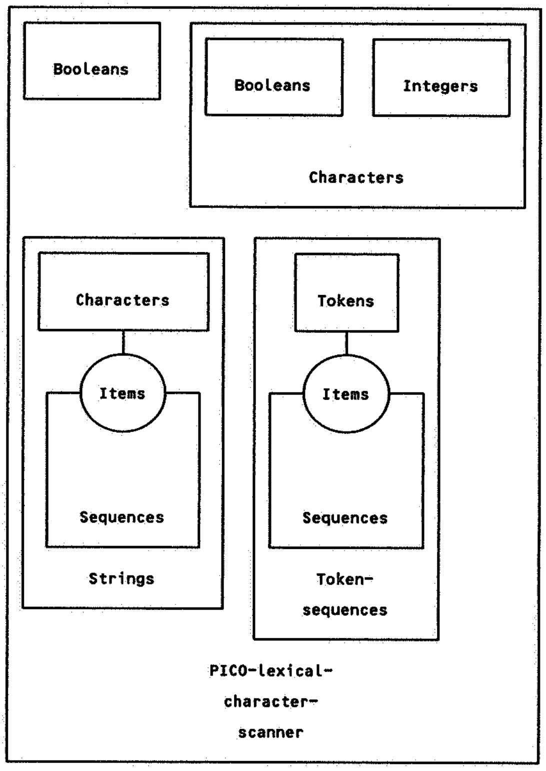
1) A character-level scanner is defined (4.5.2.1). This character-level scanner distinguishes characters according to their character types, i.e. letter, digit, layout, etc.
2) The lexical syntax for PICO and the construction rules for lexical tokens are defined (4.5.2.2). This amounts to defining the syntactic form of identifiers, strings, etc. and to defining the result for each case, e.g. parsing the non-terminal `integer-constant` of the lexical syntax will have as result `token("integer-constant", x)`, where `x` is the string representation of the integer constant.
3) A lexical scanner for PICO is obtained by combining the results of the previous two steps with `Context-free-parser`. (4.5.2.3).

4.5.2.1. Lexical character scanner

4.5.2.1.a. Global description

PICO-lexical-character-scanner defines the character-level scanner char-scan which distinguishes characters according to their character types, i.e. letter, digit, layout and literal, and converts the input string into a Token-sequnece.

4.5.2.1.b. Structure diagram



4.5.2.1.c. Specification

```
module PICO-lexical-character-scanner
begin
   exports
        begin
           functions
                char-scan : STRING -> TOKEN-SEQUENCE
        end

   imports Booleans, Characters, Strings, Token-sequences

   functions
        char-scan1 : CHAR -> TOKEN
        is-layout  : CHAR -> BOOL

   variables
        c        : -> CHAR
        str       : -> STRING

    equations

    [206]  char-scan(seq(c, str))        = seq(char-scan1(c), char-scan(str))

    [207]  char-scan("")          = null-token-sequence

    [208]  char-scan1(c)          = if(is-layout(c), token("layout", string(c)),
                                    if(is-letter(c),token("letter", string(c)),
                                    if(is-digit(c),token("digit", string(c)),
                                       token("literal", string(c)))))

    [209]  is-layout(c)           = or(eq(c, char-space),
                                       or(eq(c, char-ht),
                                          eq(c, char-nl)))

end PICO-lexical-character-scanner
```

4.5.2.2. Lexical syntax and rules for token construction

4.5.2.2.a. Global description

The lexical syntax for PICO and the construction rules for lexical tokens are defined in this section. This amounts to defining the syntactic form of identifiers, strings, etc. and to defining the result for each case, e.g. parsing the non-terminal `integer-constant` of the lexical syntax will have as result `token("integer-constant", x)`, where `x` is the string representation of the integer constant.

The following data types are defined here:

`PICO-non-terminals-of-lexical-syntax`: defines the sort `LEX-NON-TERMINAL` and all non-terminals of the lexical syntax.

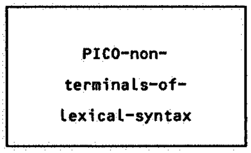
`PICO-lex-BNF-patterns`: defines a version of `BNF-patterns` with parameter `Non-terminals` bound to `PICO-non-terminals-of-lexical-syntax`.

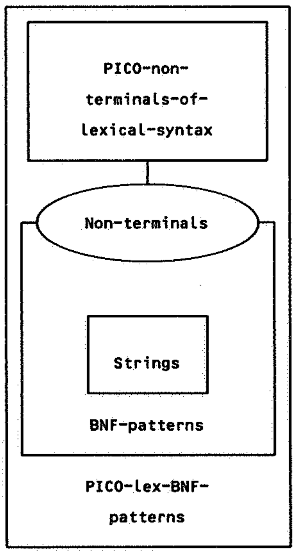
`PICO-atree-operators-of-lexical-syntax`: defines the sort `LEX-OPERATOR` and the operators for constructing abstract syntax trees for the lexical syntax.

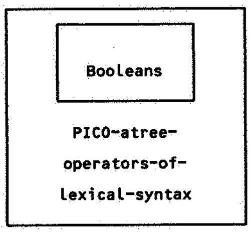
`PICO-lex-atree-environments`: defines a version of `Atree-environments` with parameter `Operators` bound to `PICO-atree-operators-of-lexical-syntax`.

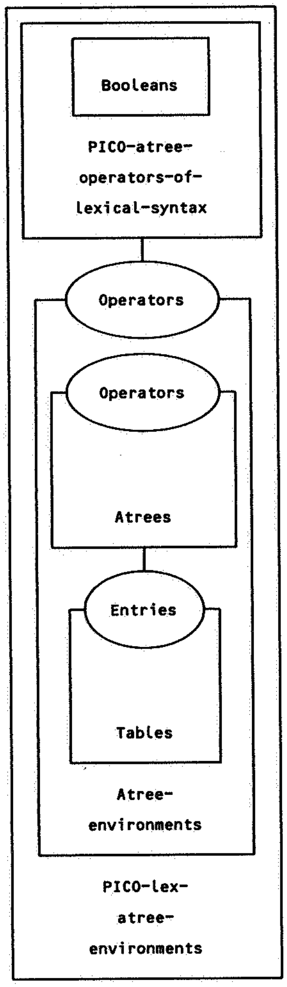
`PICO-lexical-syntax`: defines the lexical syntax for PICO and the rules for token construction. Essentially the grammar contains for each non-terminal pairs of equations for the functions `rule` (i.e. the actual syntax rule) and `build` (i.e. the construction procedure for abstract syntax trees). Note that all syntax rules with names starting with `non-empty` do not appear in the original grammar. These rules are artefacts made necessary by limitations in the descriptive power of `BNF-patterns`; most notably, it is impossible to associate different `build` functions with the alternatives in one `rule`.

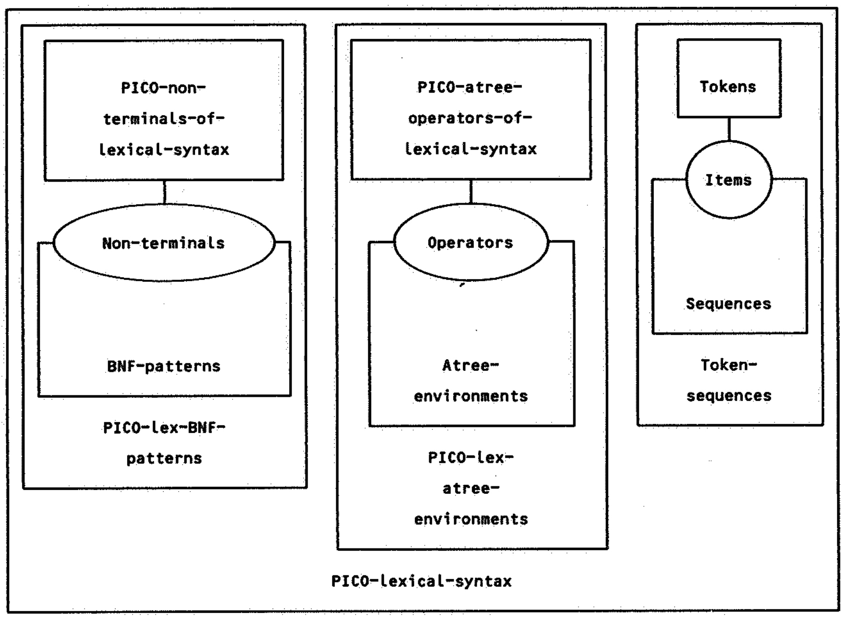
4.5.2.2.b. Structure diagrams






Booleans
PICO-atree-operators-of-lexical-syntax


Booleans
PICO-atree-operators-of-lexical-syntax
Operators
Operators
Atrees
Entries
Tables
Atree-environments
PICO-lex-atree-environments



4.5.2.2.c. Specification

```
module PICO-non-terminals-of-lexical-syntax
begin
   exports
        begin
           sorts LEX-NON-TERMINAL

           functions
                lexical-stream            :           -> LEX-NON-TERMINAL
                non-empty-lexical-stream:             -> LEX-NON-TERMINAL
                empty-lexical-stream      :           -> LEX-NON-TERMINAL
                lexical-item              :           -> LEX-NON-TERMINAL
                optional-layout           :           -> LEX-NON-TERMINAL
                keyword-or-ident          :           -> LEX-NON-TERMINAL
                ident                     :           -> LEX-NON-TERMINAL
                ident-chars               :           -> LEX-NON-TERMINAL
                non-empty-ident-chars     :           -> LEX-NON-TERMINAL
                ident-char                :           -> LEX-NON-TERMINAL
                integer-const             :           -> LEX-NON-TERMINAL
                digits                    :           -> LEX-NON-TERMINAL
                non-empty-digits          :           -> LEX-NON-TERMINAL
                digit                     :           -> LEX-NON-TERMINAL
                string-const              :           -> LEX-NON-TERMINAL
                string-tail               :           -> LEX-NON-TERMINAL
                non-empty-string-tail     :           -> LEX-NON-TERMINAL
                quote                     :           -> LEX-NON-TERMINAL
                any-char-but-quote        :           -> LEX-NON-TERMINAL
                letter                    :           -> LEX-NON-TERMINAL
                layout                    :           -> LEX-NON-TERMINAL
                literal                   :           -> LEX-NON-TERMINAL
                concat                    :           -> LEX-NON-TERMINAL
                assign-or-colon           :           -> LEX-NON-TERMINAL
                empty                   :                   -> LEX-NON-TERMINAL
        end

end PICO-non-terminals-of-lexical-syntax

module PICO-lex-BNF-patterns
begin
   imports BNF-patterns
        { renamed by
                [ PATTERN -> LEX-PATTERN,
                  t -> lt ]
          Non-terminals bound by
                [ NON-TERMINAL -> LEX-NON-TERMINAL ]
                to PICO-non-terminals-of-lexical-syntax
        }

end PICO-lex-BNF-patterns

module PICO-atree-operators-of-lexical-syntax
begin
   exports
      begin
         sorts LEX-OPERATOR

         functions
           op-lex-item   :                              -> LEX-OPERATOR
           op-lex-stream:                               -> LEX-OPERATOR
           eq            : LEX-OPERATOR # LEX-OPERATOR -> BOOL

      end

   imports Booleans

   variables
        o1, o2 :-> LEX-OPERATOR

   equations

   [210]  eq(o1, o2)                           = eq(o2, o1)
   [211]  eq(op-lex-item, op-lex-item)         = true
   [212]  eq(op-lex-item, op-lex-stream)       = false
   [213]  eq(op-lex-stream, op-lex-stream)     = true


end PICO-atree-operators-of-lexical-syntax

module PICO-lex-atree-environments
begin

   imports Atree-environments
        { renamed by
                [ ATREE -> LEX-ATREE,
                  atree -> lex-atree,
                  null-atree -> null-lex-atree,
                  error-atree -> error-lex-atree,
                lexical-atree -> lexical-lex-atree,
                ATREE-ENV -> LEX-ATREE-ENV,
                null-atree-env -> null-lex-atree-env,
                ATREE -> LEX-ATREE,
                error-atree -> error-lex-atree ]
            Operators bound by
               [ OPERATOR -> LEX-OPERATOR,
                eq -> eq ]
               to PICO-atree-operators-of-lexical-syntax
        }

end  PICO-lex-atree-environments

module PICO-lexical-syntax
begin

   exports
        begin
           functions
               rule       : LEX-NON-TERMINAL                  -> LEX-PATTERN
               build      : LEX-NON-TERMINAL # LEX-ATREE-ENV -> LEX-ATREE
               lex-stream : TOKEN-SEQUENCE                    -> LEX-ATREE
               lex-item   : TOKEN                             -> LEX-ATREE
        end

   imports PICO-lex-BNF-patterns, PICO-lex-atree-environments, Token-sequences


   variables
        env        :-> LEX-ATREE-ENV
        l, l1, l2 :-> TOKEN-SEQUENCE
        t, t1, t2 :-> TOKEN
        s, s1, s2 :-> STRING
        d, d1, d2 :-> STRING

   equations

   [214]  rule(lexical-stream)  = n(non-empty-lexical-stream,"ls") |
                                    n(empty-lexical-stream,"ls")
   [215]  build(lexical-stream, env)
                                 = "ls" ^ env
   [216]  rule(non-empty-lexical-stream)
                                 = n(lexical-item,"t") + n(lexical-stream,"l")
   [217]  build(non-empty-lexical-stream, env)
                                 = lex-atree(op-lex-stream, lex-stream(seq(t, l)))
                                   when  lex-atree(op-lex-item, lex-item(t))
                                                = "t" ^ env,
                                         lex-atree(op-lex-stream, lex-stream(l))
                                                = "l" ^ env
   [218]  rule(empty-lexical-stream)
                                 = n(empty)
   [219]  build(empty-lexical-stream, env)
                                 = lex-atree(op-lex-stream,
                                         lex-stream(null-token-sequence))
```

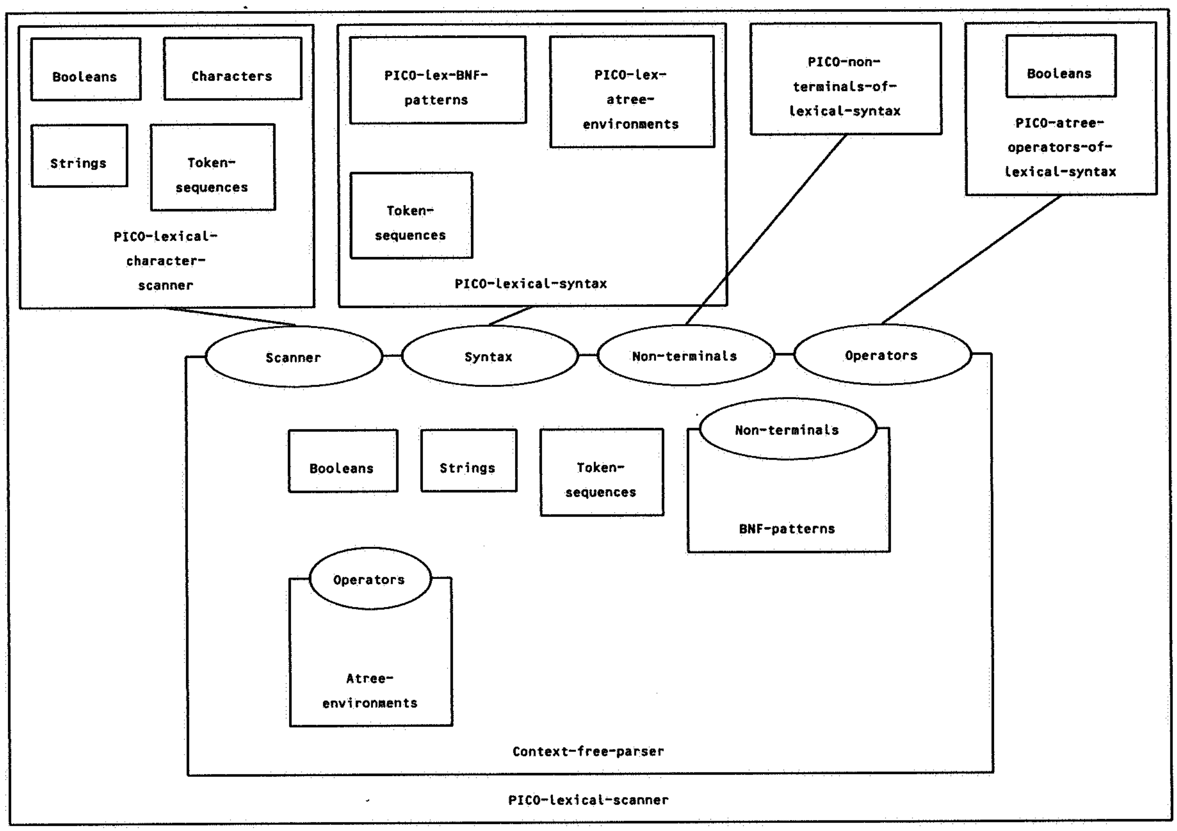
```
[220]  rule(lexical-item)     = n(optional-layout) +
                                ( n(keyword-or-ident,"i") |
                                  n(integer-const,"i") |
                                  n(string-const,"i") |
                                  n(literal,"i")
                                )
[221]  build(lexical-item, env)
                              = "i" ^ env

[222]  rule(optional-layout) = n(layout) | n(empty)
[223]  build(optional-layout, env)
                              = null-lex-atree

[224]  rule(keyword-or-ident)= n(ident,"i")
[225]  build(keyword-or-ident, env)
                              = if(or(eq(s, "begin"),
                                   or(eq(s, "end"),
                                   or(eq(s, "declare"),
                                   or(eq(s, "integer"),
                                   or(eq(s, "string"),
                                   or(eq(s, "if"),
                                   or(eq(s, "then"),
                                   or(eq(s, "else"),
                                   or(eq(s, "fi"),
                                   or(eq(s, "while"),
                                   or(eq(s, "do"),
                                      eq(s, "od")))))))))))),
                                      lex-atree(op-lex-item,
                                            lex-item(token("keyword", s))),
                                      lex-atree(op-lex-item,
                                            lex-item(token("id", s))))

                                when lexical-lex-atree(token("id",s)) = "i" ^ env


[226]  rule(ident)            = n(letter,"s1") + n(ident-chars,"s2")
[227]  build(ident, env)      = lexical-lex-atree(token("id", conc(s1, s2)))
                                when  string-atree(s1) = "s1" ^ env,
                                      string-atree(s2) = "s2" ^ env

[228]  rule(ident-chars)      = n(non-empty-ident-chars,"s") | n(empty,"s")
[229]  build(ident-chars, env)
                              = "s" ^ env
[230]  rule(non-empty-ident-chars)
                              = n(ident-char,"s1") + n(ident-chars,"s2")
[231]  build(non-empty-ident-chars, env)
                              = string-atree(conc(s1, s2))
                                when  string-atree(s1) = "s1" ^ env,
                                      string-atree(s2) = "s2" ^ env
[232]  rule(ident-char)       = n(letter,"x") | n(digit,"x")
[233]  build(ident-char, env)= "x" ^ env

[234]  rule(integer-const)   = n(digit,"d1") + n(digits,"d2")
[235]  build(integer-const, env)
                              = lex-atree(op-lex-item,
                                    lex-item(token("integer-constant",
                                                   conc(d1, d2))))
                                when  string-atree(d1) = "d1" ^ env,
                                      string-atree(d2) = "d2" ^ env

[236]  rule(digits)           = n(non-empty-digits,"d") | n(empty,"d")
[237]  build(digits, env)     = "d" ^ env

[238]  rule(non-empty-digits)= n(digit,"d1") + n(digits,"d2")
[239]  build(non-empty-digits, env)
                              = string-atree(conc(d1, d2))
                                when  string-atree(d1) = "d1" ^ env,
                                      string-atree(d2) = "d2" ^ env

[240]  rule(string-const)     = n(quote) + n(string-tail,"s")
[241]  build(string-const, env)
                              = lex-atree(op-lex-item,
                                      lex-item(token("string-constant", s)))
                                when  string-atree(s) = "s" ^ env

[242]  rule(string-tail)      = n(non-empty-string-tail,"s") | n(quote,"s")
[243]  build(string-tail, env)
                              = "s" ^ env
[244]  rule(non-empty-string-tail)
                              = n(any-char-but-quote,"s1") + n(string-tail,"s2")
[245]  build(non-empty-string-tail, env)
                              = string-atree(conc(s1, s2))
                                when  string-atree(s1) = "s1" ^ env,
                                      string-atree(s2) = "s2" ^ env

[246]  rule(quote)            = lt(string(char-quote))
[247]  build(quote, env)      = string-atree("")

[248]  rule(any-char-but-quote)
                              = n(letter,"c") |
                                n(digit,"c") |
                                n(literal,"c") |
                                n(layout,"c")
[249]  build(any-char-but-quote, env)
                              = "c" ^ env

[250]  rule(letter)           = lexical("letter","s")
[251]  build(letter, env)     = string-atree(s)
                                when lexical-lex-atree(token("letter",s)) = "s" ^ env

[252]  rule(digit)            = lexical("digit","d")
[253]  build(digit, env)      = string-atree(d)
                                when lexical-lex-atree(token("digit", d)) = "d" ^ env
[254]  rule(layout)           = lexical("layout","s")
[255]  build(layout, env)     = string-atree(s)
                                when lexical-lex-atree(token("layout", s)) = "s" ^ env
[256]  rule(literal)          = lt("(","s") |
                                lt(")","s") |
                                lt("+","s") |
                                lt("-","s") |
                                      lt(";","s") |
                                      lt(",","s") |
                                      n(concat,"s") |
                                      n(assign-or-colon,"s")
[257]  build(literal, env)         = "s" ^ env

[258]  rule(concat)                = lt("|") + lt("|")
[259]  build(concat, env)          = string-atree("||")

[260]  rule(assign-or-colon) = lt(":") + (lt("=","s") | n(empty,"s"))
[261]  build(assign-or-colon, env)
                                   = if(eq(s, "="),
                                        string-atree(":="),
                                        string-atree(":"))
                                      when string-atree(s) = "s" ^ env

[262]  rule(empty)                 = lt("")
[263]  build(empty, env)           = string-atree("")


end PICO-lexical-syntax
```

4.5.2.3. Lexical scanner

4.5.2.3.a. Global description

In this section a lexical scanner for PICO is obtained by combining `PICO-lexical-character-scanner`, `PICO-lexical-syntax`, `PICO-non-terminals-of-lexical-syntax` and `PICO-atree-operators-of-lexical-syntax` with `Context-free-parser`.

4.5.2.3.b. Structure diagram



4.5.2.3.c. Specification

```
module PICO-lexical-scanner
begin
   exports
        begin
           functions
                lex-scan : STRING -> TOKEN-SEQUENCE
        end

   imports Context-free-parser
        { Scanner bound by
                [ scan -> char-scan ]
                to PICO-lexical-character-scanner
          Syntax bound by
                [ rule -> rule,
                  build -> build ]
```

```
              to PICO-lexical-syntax
        Non-terminals bound by
              [ NON-TERMINAL -> LEX-NON-TERMINAL ]
              to PICO-non-terminals-of-lexical-syntax
        Operators bound by
              [ OPERATOR -> LEX-OPERATOR,
                eq -> eq ]
              to PICO-atree-operators-of-lexical-syntax
      }

  variables
      l :-> TOKEN-SEQUENCE
      s :-> STRING

  equations

  [264]  lex-scan(s)          = l
                                when  lex-atree(op-lex-stream, lex-stream(l)) =
                                      parse(lexical-stream, s)

end PICO-lexical-scanner
```

4.5.3. Abstract and concrete syntax

In this section we specify the abstract and concrete syntax for PICO; this will result in a specification for a parser that transforms PICO-programs from their textual form into abstract syntax trees. We proceed as follows:

1) The abstract syntax for PICO is defined (4.5.3.1).
2) The concrete syntax and the rules for constructing abstract syntax trees are defined (4.5.3.2).
3) The lexical scanner (as defined in the previous section), the concrete syntax and the rules for the construction of abstract syntax trees (both defined in this section) are combined with `Context-free-parser`. In this way we obtain a parser that transforms PICO programs into abstract syntax trees (4.5.3.3).

4.5.3.1. Abstract syntax

4.5.3.1.a. Global description

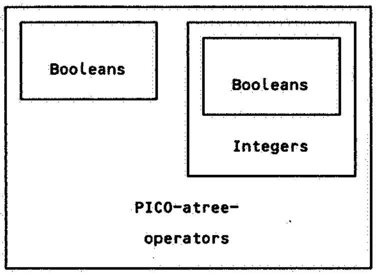
In this section the abstract syntax for PICO is defined. This involves the following data types:

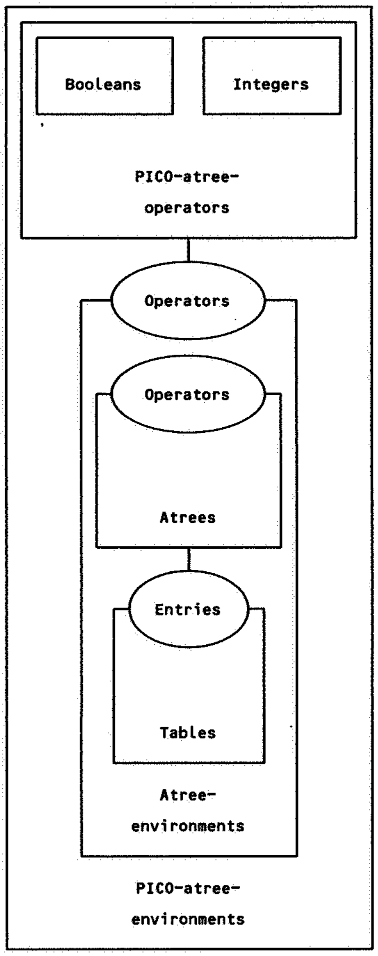
`PICO-atree-operators`: the operators for constructing abstract syntax trees.

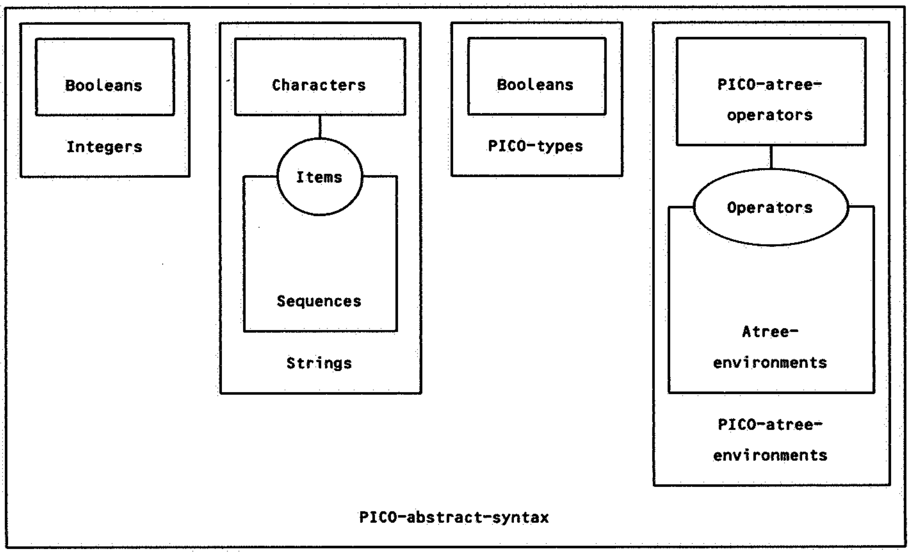
`PICO-atree-environments`: a version of `Atree-environments` with parameter `Operators` bound to `PICO-atree-operators`.

`PICO-abstract-syntax`: defines the actual abstract syntax. Essentially, this module defines higher-level constructor functions (e.g. `abs-if`, `abs-while`, etc.) which allow a natural expression of the PICO abstract syntax tree. These constructor functions are defined in terms of `Atrees`.

4.5.3.1.b. Structure diagrams




Booleans
Integers
PICO-atree-operators
Operators
Operators
Atrees
Entries
Tables
Atree-environments
PICO-atree-environments


Booleans
Integers
Characters
Items
Sequences
Strings
Booleans
PICO-types
PICO-atree-operators
Operators
Atree-environments
PICO-atree-environments
PICO-abstract-syntax

4.5.3.1.c. Specification

```
module PICO-atree-operators
begin
   exports
        begin
           sorts PICO-OPERATOR

           functions

                op-pico-program        :               -> PICO-OPERATOR
                op-decls               :               -> PICO-OPERATOR
                op-empty-decls         :               -> PICO-OPERATOR
                op-series              :               -> PICO-OPERATOR
                op-empty-series        :               -> PICO-OPERATOR
                op-assign              :               -> PICO-OPERATOR
                op-if                  :               -> PICO-OPERATOR
                op-while               :               -> PICO-OPERATOR
                op-plus                :               -> PICO-OPERATOR
                op-conc                :               -> PICO-OPERATOR
                op-var                 :               -> PICO-OPERATOR
                op-integer-constant    :               -> PICO-OPERATOR
                op-string-constant     :               -> PICO-OPERATOR
                op-id                  :               -> PICO-OPERATOR
                op-integer-type        :               -> PICO-OPERATOR
                op-string-type         :               -> PICO-OPERATOR

                ord            : PICO-OPERATOR                 -> INTEGER
                eq             : PICO-OPERATOR # PICO-OPERATOR -> BOOL
        end

   imports Booleans, Integers

   variables

        c1, c2 :-> PICO-OPERATOR

   equations

   [265]  ord(op-pico-program)        = 0
   [266]  ord(op-decls)               = succ(ord(op-pico-program))
   [267]  ord(op-empty-decls)         = succ(ord(op-decls))
   [268]  ord(op-series)              = succ(ord(op-empty-decls))
   [269]  ord(op-empty-series)        = succ(ord(op-series))
   [270]  ord(op-assign)              = succ(ord(op-empty-series))
   [271]  ord(op-if)                  = succ(ord(op-assign))
   [272]  ord(op-while)               = succ(ord(op-if))
   [273]  ord(op-plus)                = succ(ord(op-while))
   [274]  ord(op-conc)                = succ(ord(op-plus))
   [275]  ord(op-var)                 = succ(ord(op-conc))
   [276]  ord(op-integer-constant)    = succ(ord(op-var))
   [277]  ord(op-string-constant)     = succ(ord(op-integer-constant))
   [278]  ord(op-id)                  = succ(ord(op-string-constant))
```

```
  [279]  ord(op-integer-type)         = succ(ord(op-id))
  [280]  ord(op-string-type)          = succ(ord(op-integer-type))

  [281]  eq(c1, c2)                   = eq(ord(c1), ord(c2))

end PICO-atree-operators

module PICO-atree-environments
begin

  imports Atree-environments
       { renamed by
               [ ATREE -> PICO-ATREE,
                 atree -> pico-atree,
                 null-atree -> null-pico-atree,
                 error-atree -> error-pico-atree,
                 string-atree -> string-pico-atree,
                 integer-atree -> integer-pico-atree,
                 lexical-atree -> lexical-pico-atree,
                 ATREE-ENV -> PICO-ATREE-ENV,
                 null-atree-env -> null-pico-atree-env]
         Operators bound by
               [ OPERATOR -> PICO-OPERATOR,
                 eq -> eq]
               to PICO-atree-operators
       }

  variables
       s :-> STRING
       e :-> PICO-ATREE-ENV
       f :-> BOOL
       v :-> PICO-ATREE

  equations

  [282]  s ^ e          = v
                          when <f, v> = lookup(s, e)


end PICO-atree-environments

module PICO-abstract-syntax
begin
  exports
       begin
          sorts  PICO-PROGRAM, DECLS, EXP, ID, SERIES, STATEMENT

          functions

               abs-pico-program     : DECLS # SERIES              -> PICO-PROGRAM
               abs-decls            : ID # PICO-TYPE # DECLS      -> DECLS
               abs-empty-decls      :                             -> DECLS
               abs-series           : STATEMENT # SERIES          -> SERIES
               abs-empty-series     :                             -> SERIES
               abs-assign           : ID  # EXP                   -> STATEMENT
```

```
        abs-if                : EXP # SERIES # SERIES     -> STATEMENT
        abs-while             : EXP # SERIES              -> STATEMENT
        abs-plus              : EXP # EXP                 -> EXP
        abs-conc              : EXP # EXP                 -> EXP
        abs-var               : ID                        -> EXP
        abs-integer-constant  : INTEGER                   -> EXP
        abs-string-constant   : STRING                    -> EXP
        abs-id                : STRING                    -> ID

        pico-program          : PICO-ATREE                -> PICO-PROGRAM
        decls                 : PICO-ATREE                -> DECLS
        series                : PICO-ATREE                -> SERIES
        statement             : PICO-ATREE                -> STATEMENT
        exp                   : PICO-ATREE                -> EXP
        id                    : PICO-ATREE                -> ID

        pico-type-atree       : PICO-TYPE                 -> PICO-ATREE

        append-statement      : SERIES # STATEMENT        -> SERIES
    end

imports Integers, Strings, PICO-types, PICO-atree-environments

variables

    ds              :-> PICO-ATREE
    sr, sr1, sr2    :-> PICO-ATREE
    st              :-> PICO-ATREE
    i               :-> PICO-ATREE
    t               :-> PICO-TYPE
    x, x1, x2       :-> PICO-ATREE
    str             :-> STRING
    n               :-> INTEGER
    stat, stat1, stat2 :-> STATEMENT
    ser             :-> SERIES

equations

[283]  abs-pico-program(decls(ds), series(sr))
                            = pico-program(pico-atree(op-pico-program, ds, sr))
[284]  abs-decls(id(i), t, decls(ds))
                            = decls(pico-atree(op-decls, i, pico-type-atree(t), ds))
[285]  abs-empty-decls      = decls(pico-atree(op-empty-decls))
[286]  abs-series(statement(st), series(sr))
                            = series(pico-atree(op-series, st, sr))
[287]  abs-empty-series     = series(pico-atree(op-empty-series))
[288]  abs-assign(id(i), exp(x))
                            = statement(pico-atree(op-assign, i, x))
[289]  abs-if(exp(x), series(sr1), series(sr2))
                            = statement(pico-atree(op-if, x, sr1, sr2))
[290]  abs-while(exp(x), series(sr))
                            = statement(pico-atree(op-while, x, sr))
[291]  abs-plus(exp(x1), exp(x2))
                            = exp(pico-atree(op-plus, x1, x2))
[292]  abs-conc(exp(x1), exp(x2))
```

```
                                = exp(pico-atree(op-conc, x1, x2))
   [293]  abs-var(id(i))        = exp(pico-atree(op-var, i))
   [294]  abs-integer-constant(n)
                                = exp(pico-atree(op-integer-constant,
                                                 integer-pico-atree(n)))
   [295]  abs-string-constant(str)
                                = exp(pico-atree(op-string-constant,
                                                 string-pico-atree(str)))
   [296]  abs-id(str)           = id(pico-atree(op-id, string-pico-atree(str)))


   [297]  append-statement(abs-empty-series, stat)
                                = abs-series(stat, abs-empty-series)
   [298]  append-statement(abs-series(stat1, ser), stat2)
                                = abs-series(stat1, append-statement(ser, stat2))

end PICO-abstract-syntax
```

4.5.3.2. Concrete syntax and rules for abstract syntax tree construction

4.5.3.2.a. Global description

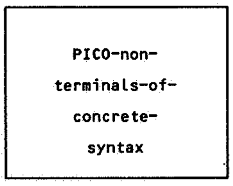
In this section the concrete syntax and the rules for abstract syntax tree construction for PICO are defined. This involves the following modules:

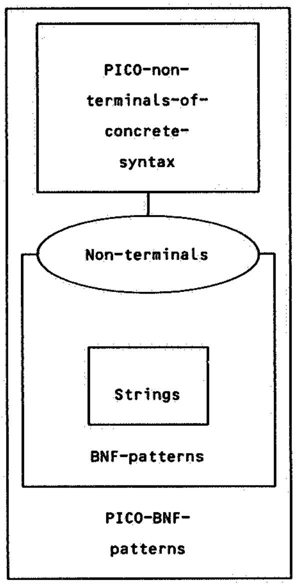
`PICO-non-terminals-of-concrete-synyax`: defines the sort `PICO-NON-TERMINAL` and all non-terminals of the concrete syntax.

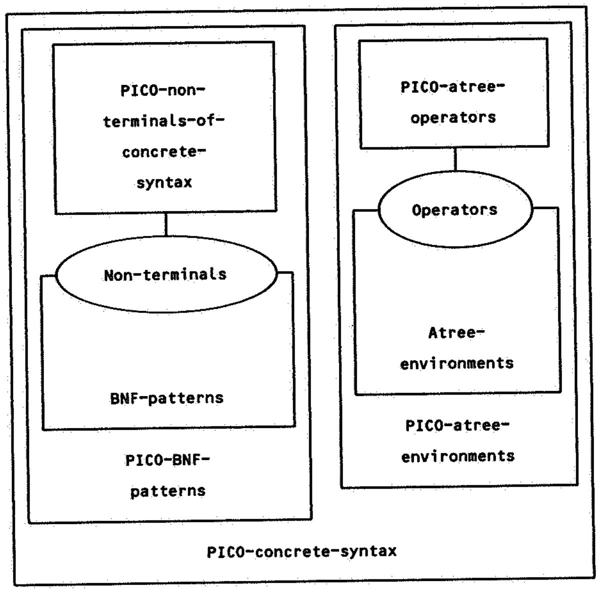
`PICO-BNF-patterns`: defines a version of `BNF-patterns` with parameter `Non-terminals` bound to `PICO-non-terminals-of-concrete-syntax`.

`PICO-concrete-syntax`: defines the concrete syntax for PICO and the rules for abstract syntax tree construction. Essentially the grammar contains for each non-terminal in the concrete syntax pairs of equations for the functions `rule` (i.e. the actual syntax rule) and `build` (i.e. the construction procedure for abstract syntax trees).

4.5.3.2.b. Structure diagrams







4.5.3.2.c. Specification

```
module PICO-non-terminals-of-concrete-syntax
begin
   exports
        begin
           sorts PICO-NON-TERMINAL

           functions
                pico-program     :              -> PICO-NON-TERMINAL
                decls            :              -> PICO-NON-TERMINAL
                empty-decls      :              -> PICO-NON-TERMINAL
                id-type-list     :              -> PICO-NON-TERMINAL
                type             :              -> PICO-NON-TERMINAL
                type-integer     :              -> PICO-NON-TERMINAL
                type-string      :              -> PICO-NON-TERMINAL
                series           :              -> PICO-NON-TERMINAL
                empty-series     :              -> PICO-NON-TERMINAL
                non-empty-series:               -> PICO-NON-TERMINAL
                stat             :              -> PICO-NON-TERMINAL
                assign           :              -> PICO-NON-TERMINAL
                if               :              -> PICO-NON-TERMINAL
                while            :              -> PICO-NON-TERMINAL
                exp              :              -> PICO-NON-TERMINAL
                plus             :              -> PICO-NON-TERMINAL
                conc             :              -> PICO-NON-TERMINAL
                var              :              -> PICO-NON-TERMINAL
                id               :              -> PICO-NON-TERMINAL
                integer-constant:               -> PICO-NON-TERMINAL
                string-constant :               -> PICO-NON-TERMINAL
        end

 end PICO-non-terminals-of-concrete-syntax
```

```
module PICO-BNF-patterns
begin
   imports BNF-patterns
        { renamed by
                [ PATTERN -> PICO-PATTERN,
                 t -> pt,
                 lexical -> plexical ]
          Non-terminals bound by
                [ NON-TERMINAL -> PICO-NON-TERMINAL ]
                to PICO-non-terminals-of-concrete-syntax
        }

end PICO-BNF-patterns

module PICO-concrete-syntax
begin
   exports
        begin
           functions
                rule :  PICO-NON-TERMINAL                     -> PICO-PATTERN
                build: PICO-NON-TERMINAL # PICO-ATREE-ENV -> PICO-ATREE
        end

   imports PICO-BNF-patterns, PICO-atree-environments

   variables
        env     :-> PICO-ATREE-ENV
        str     :-> STRING


   equations


   [299] rule(pico-program)      = pt("begin") + n(decls,"d")
                                              + n(series,"s") + pt("end")

   [300] build(pico-program, env)
                                 = pico-atree(op-pico-program, "d" ^ env, "s" ^ env)

   [301] rule(decls)             = pt("declare") + n(id-type-list,"l") + pt(";")
   [302] build(decls, env)       = "l" ^ env

   [303] rule(empty-decls)       = pt("")
   [304] build(empty-decls, env)= pico-atree(op-empty-decls)

   [305] rule(id-type-list)      = n(id,"i") + pt(":") + n(type,"t") +
                                     ( n(empty-decls,"l") |
                                       pt(",") + n(id-type-list,"l")
                                     )
   [306] build(id-type-list, env)
                                 = pico-atree(op-decls,
                                          "i" ^ env,
                                          "t" ^ env,
                                          "l" ^ env)


   [307] rule(type)              = n(type-integer,"t") | n(type-string,"t")
```

```
[308] build(type, env)          = "t" ^ env

[309] rule(type-integer)        = pt("integer")
[310] build(type-integer, env)
                                = pico-atree(op-integer-type)

[311] rule(type-string)         = pt("string")
[312] build(type-string, env)
                                = pico-atree(op-string-type)

[313] rule(series)              = n(empty-series,"s") | n(non-empty-series,"s")
[314] build(series, env)        = "s" ^ env

[315] rule(empty-series)        = pt("")
[316] build(empty-series, env)
                                = pico-atree(op-empty-series)

[317] rule(non-empty-series) = n(stat,"st") + ( n(empty-series,"s") |
                                                pt(";") + n(series,"s")
                                              )

[318] build(non-empty-series, env)
                                = pico-atree(op-series, "st" ^ env, "s" ^ env)
[319] rule(stat)                = n(assign,"st") | n(if,"st") | n(while,"st")
[320] build(stat, env)          = "st" ^ env

[321] rule(assign)              = n(id,"i") + pt(":=") + n(exp,"e")
[322] build(assign,env)         = pico-atree(op-assign, "i" ^ env, "e" ^ env)

[323] rule(if)                  = pt("if") + n(exp,"e")
                                    + pt("then") + n(series,"s1")
                                    + pt("else") + n(series,"s2") + pt("fi")

[324] build(if, env)            = pico-atree(op-if,
                                      "e" ^ env,
                                      "s1" ^ env,
                                      "s2" ^ env)

[325] rule(while)               = pt("while") + n(exp,"e")
                                      + pt("do") + n(series,"s") + pt("od")
[326] build(while, env)         = pico-atree(op-while, "e" ^ env, "s" ^ env)

[327] rule(exp)                 = n(var,"e") |
                                  n(integer-constant,"e") |
                                  n(string-constant,"e") |
                                  n(plus,"e") |
                                  n(conc,"e") |
                                  ( pt("(") + n(exp,"e") + pt(")") )
[328] build(exp, env)           = "e" ^ env

[329] rule(plus)                = n(exp,"e1") + pt("+") + n(exp,"e2")
[330] build(plus, env)          = pico-atree(op-plus, "e1" ^ env, "e2" ^ env)

[331] rule(conc)                = n(exp,"e1") + pt("||") + n(exp,"e2")
[332] build(conc, env)          = pico-atree(op-conc, "e1" ^ env, "e2" ^ env)
[333] rule(var)                 = n(id,"i")
[334] build(var, env)           = pico-atree(op-var, "i" ^ env)
```

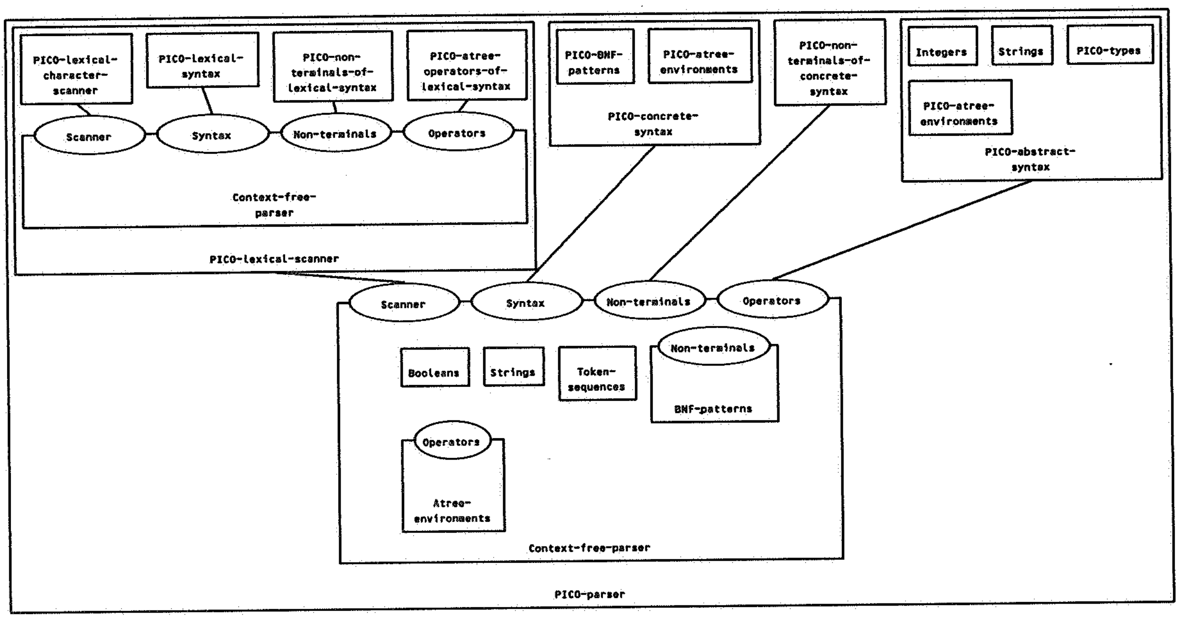
```
  [335] rule(id)              = plexical("id","i")
  [336] build(id, env)        = pico-atree(op-id, string-pico-atree(str))
                                  when lexical-pico-atree(token("id", str)) = "i" ^ env

  [337] rule(integer-constant) = plexical("integer-constant","i")

  [338] build(integer-constant, env)
                              = pico-atree(op-integer-constant,
                                           integer-pico-atree(str-to-int(str)))
                                  when lexical-pico-atree(token("integer-constant", str))
                                           = "i" ^ env

  [339] rule(string-constant)  = plexical("string-constant","s")
  [340] build(string-constant, env)
                              = pico-atree(op-string-constant, string-pico-atree(str))
                                  when lexical-pico-atree(token("string-constant", str))
                                           = "s" ^ env

end PICO-concrete-syntax
```

4.5.3.3. Parser

4.5.3.3.a. Global description

In this section a parser for PICO is obtained by combining `PICO-lexical-scanner`, `PICO-concrete-syntax`, `PICO-non-terminals-of-concrete-syntax` and `PICO-atree-operators-of-concrete-syntax` with `Context-free-parser`.

4.5.3.3.b. Structure diagram



4.5.3.3.c. Specification

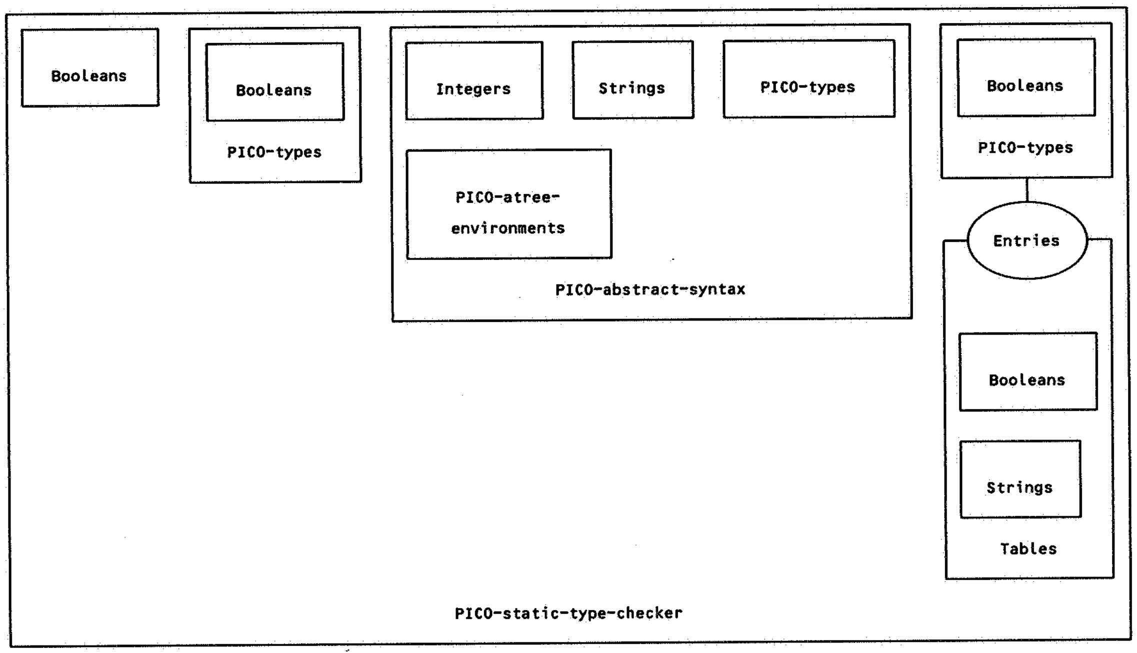
```
module PICO-parser
begin
   exports
        begin
           functions
                parse-and-construct : STRING -> PICO-ATREE
        end

   imports Context-free-parser
        { Scanner bound by
                [ scan -> lex-scan ]
                to PICO-lexical-scanner
          Syntax bound by
                [ rule -> rule,
                  build -> build ]
                to PICO-concrete-syntax
          Non-terminals bound by
                [ NON-TERMINAL -> PICO-NON-TERMINAL ]
                to PICO-non-terminals-of-concrete-syntax
          Operators bound by
                [ OPERATOR -> PICO-OPERATOR,
                  eq -> eq ]
                to PICO-abstract-syntax
      }

   variables
      str :-> STRING

   equations

   [341] parse-and-construct(str) = parse(pico-program, str)

end PICO-parser
```

4.5.4. Static semantics

4.5.4.a. Global description

In this section we specify the checking of static semantic constraints on PICO programs as defined informally in section 4.2. The principal function is `check` which operates on an abstract PICO program and checks whether this program is in accordance with the static semantic constraints. For each construct in the abstract syntax tree these constraints are expressed as transformations on a type-environment. Type-environments are defined as a combination of `Tables` and `PICO-types`. Checking the declaration section of a PICO program amounts to constructing a type-environment, and checking the statement section amounts to checking each statement for conformity with a given type-environment.

4.5.4.b. Structure diagram



4.5.4.c. Specification

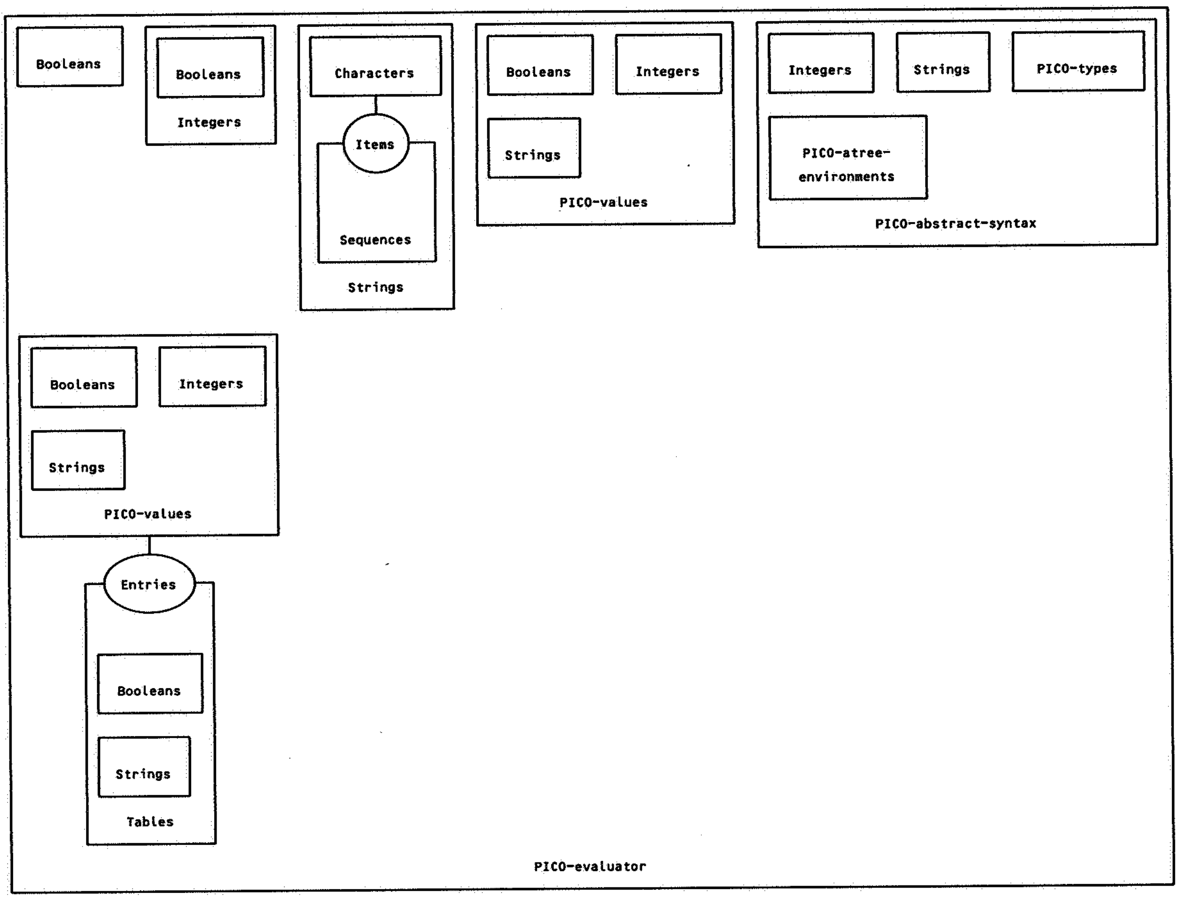
```
module PICO-static-type-checker
begin
   exports
        begin
           functions
                check: PICO-PROGRAM               -> BOOL
                check: DECLS # TYPE-ENV           -> (BOOL # TYPE-ENV)
                check: SERIES # TYPE-ENV          -> (BOOL # TYPE-ENV)
                check: STATEMENT # TYPE-ENV       -> (BOOL # TYPE-ENV)
        end

   imports Booleans, PICO-types, PICO-abstract-syntax,
           Tables
                { renamed by
                        [ TABLE -> TYPE-ENV,
                    null-table -> null-type-env ]
            Entries bound by
                  [ ENTRY -> PICO-TYPE,
                    eq -> eq,
                    error-entry -> error-type ]
                  to PICO-types
          }
functions
     type-of-exp      : EXP # TYPE-ENV        -> PICO-TYPE

variables
     dec : -> DECLS
     ser, ser1, ser2 : -> SERIES
     stat : -> STATEMENT
     name : -> STRING
     int : -> INTEGER
     typ : -> PICO-TYPE
     str : -> STRING
     x, x1, x2 : -> EXP
     env, env1, env2 : -> TYPE-ENV
     b, b1, b2, found : -> BOOL

equations

[342]  check(abs-pico-program(dec, ser))
                     = and(b1, b2)
                       when  <b1, env1> = check(dec, null-type-env),
                             <b2, env2> = check(ser, env1)

[343]  check(abs-decls(abs-id(name), typ, dec), env)
                     = check(dec, table(name, typ, env))
[344]  check(abs-empty-decls, env)
                     = < true, env >

[345]  check(abs-series(stat, ser), env)
                     = < and(b1, b2), env2 >
                       when  <b1, env1> = check(stat, env),
                             <b2, env2> = check(ser, env1)
[346]  check(abs-empty-series, env)
                     = < true, env >

[347]  check(abs-assign(abs-id(name), x), env)
                     = < and(found, eq(typ, type-of-exp(x, env))), env >
                       when <found, typ> = lookup(name, env)
[348]  check(abs-if(x, ser1, ser2), env)
                     = < and(eq(type-of-exp(x,env),integer-type), and(b1,b2)),
                         env2 >
                       when  <b1, env1> = check(ser1, env),
                             <b2, env2> = check(ser2, env1)
[349]  check(abs-while(x, ser), env)
                     = < and(eq(type-of-exp(x, env), integer-type), b),
                         env1 >
                       when <b, env1> = check(ser, env)
[350]  type-of-exp(abs-plus(x1, x2), env)
                     = if(and(eq(type-of-exp(x1, env), integer-type),
                                    eq(type-of-exp(x2, env), integer-type)),
                                 integer-type,
                                 error-type)

    [351]  type-of-exp(abs-conc(x1, x2), env)
                             = if(and(eq(type-of-exp(x1, env), string-type),
                                    eq(type-of-exp(x2, env), string-type)),
                                 string-type,
                                 error-type)

    [352]  type-of-exp(abs-integer-constant(int), env)
                             = integer-type
    [353]  type-of-exp(abs-string-constant(str), env)
                             = string-type
    [354]  type-of-exp(abs-var(abs-id(name)), env)
                             = if(found, typ, error-type)
                               when <found, typ> = lookup(name, env)

end PICO-static-type-checker
```

4.5.5. Dynamic semantics

4.5.5.a. Global description

In this section the evaluation of PICO programs is defined. To a first approximation, the evaluation of programs is defined by defining the evaluation of each kind of construct that may appear in the abstract syntax tree. Evaluation is expressed as transformation on value-environments which describe the values of the variables in the program. Value-environments are defined as combinations of `Tables` and `PICO-values`. However, since programs need not terminate this would make the evaluation function a partial function. Therefore, we introduce the notion of a `program-state` and define program evaluation as a function from program-states to program-states. This transformation of program-states can be described by a total function. The cases in which programs do not terminate are covered by conditional equations: conditions appearing in the `when`-parts of equations which describe the evaluation of a certain language construct enforce the evaluation of that construct to be only defined if the evaluation of all of its components terminates.

4.5.5.b. Structure diagram



4.5.5.c. Specification

```
module PICO-evaluator
begin

   exports
```

```
    begin
       sorts        PROGRAM-STATE

       functions
            program-state    : PICO-PROGRAM               -> PROGRAM-STATE
            program-state    : SERIES # VALUE-ENV         -> PROGRAM-STATE
            program-state    : STATEMENT # VALUE-ENV      -> PROGRAM-STATE
            program-state    : EXP # VALUE-ENV            -> PROGRAM-STATE
            program-state    : VALUE-ENV                  -> PROGRAM-STATE

            eval             : PROGRAM-STATE              -> PROGRAM-STATE
            eval-decls       : DECLS # VALUE-ENV          -> VALUE-ENV
            eval-exp         : EXP # VALUE-ENV            -> PICO-VALUE
    end

imports Booleans, Integers, Strings, PICO-values, PICO-abstract-syntax,
        Tables
            { renamed by
                     [ TABLE -> VALUE-ENV,
                       null-table -> null-value-env]
              Entries bound by
                     [ ENTRY -> PICO-VALUE,
                       eq -> eq,
                       error-entry -> error-value]
                     to PICO-values
            }
variables
     dec : -> DECLS
     ser, ser1, ser2 : -> SERIES
     stm : -> STATEMENT
     name : -> STRING
     int, int1, int2 : -> INTEGER
     val, val1, val2 : -> PICO-VALUE
     str, str1, str2 : -> STRING
     x, x1, x2 : -> EXP
     env, env1, env2 : -> VALUE-ENV
     found : -> BOOL

equations

[355]  eval(program-state(abs-pico-program(dec, ser)))
           = eval(program-state(ser, eval-decls(dec, null-value-env)))

[356]  eval-decls(abs-decls(abs-id(name), integer-type, dec), env)
           = eval-decls(dec, table(name, pico-value(0), env))
[357]  eval-decls(abs-decls(abs-id(name), string-type, dec), env)
           = eval-decls(dec, table(name, pico-value(null-string), env))
[358]  eval-decls(abs-empty-decls, env)
           = env

[359]  eval(program-state(abs-series(stm, ser), env))
           = eval(program-state(ser, env1))
             when  eval(program-state(stm, env)) = program-state(env1)

[360]  eval(program-state(abs-empty-series, env))
```

```
            = program-state(env)

[361]  eval(program-state(abs-assign(abs-id(name), x), env))
            = program-state(table(name, eval-exp(x, env), env))

[362]  eval(program-state(abs-if(x, ser1, ser2), env))
            = if(eq(eval-exp(x,env), pico-value(0)),
                 eval(program-state(ser2, env)),
                 eval(program-state(ser1, env)))

[363]  eval(program-state(abs-while(x, ser), env))
            = if(eq(eval-exp(x,env), pico-value(0)),
                 program-state(env),
                 eval(program-state(append-statement(ser, abs-while(x,ser)),
                                    env)))

[364]  eval-exp(abs-plus(x1, x2), env)
            = pico-value(add(int1,int2))
              when  pico-value(int1) = eval-exp(x1,env),
                    pico-value(int2) = eval-exp(x2,env)

[365]  eval-exp(abs-conc(x1, x2), env)
            = pico-value(conc(str1, str2))
              when  pico-value(str1) = eval-exp(x1, env),
                    pico-value(str2) = eval-exp(x2, env)

[366]  eval-exp(abs-integer-constant(int), env)
            = pico-value(int)

[367]  eval-exp(abs-string-constant(str), env)
            = pico-value(str)

[368]  eval-exp(abs-var(abs-id(name)), env)
            = val
              when <found, val> = lookup(name, env)

end PICO-évaluator
```

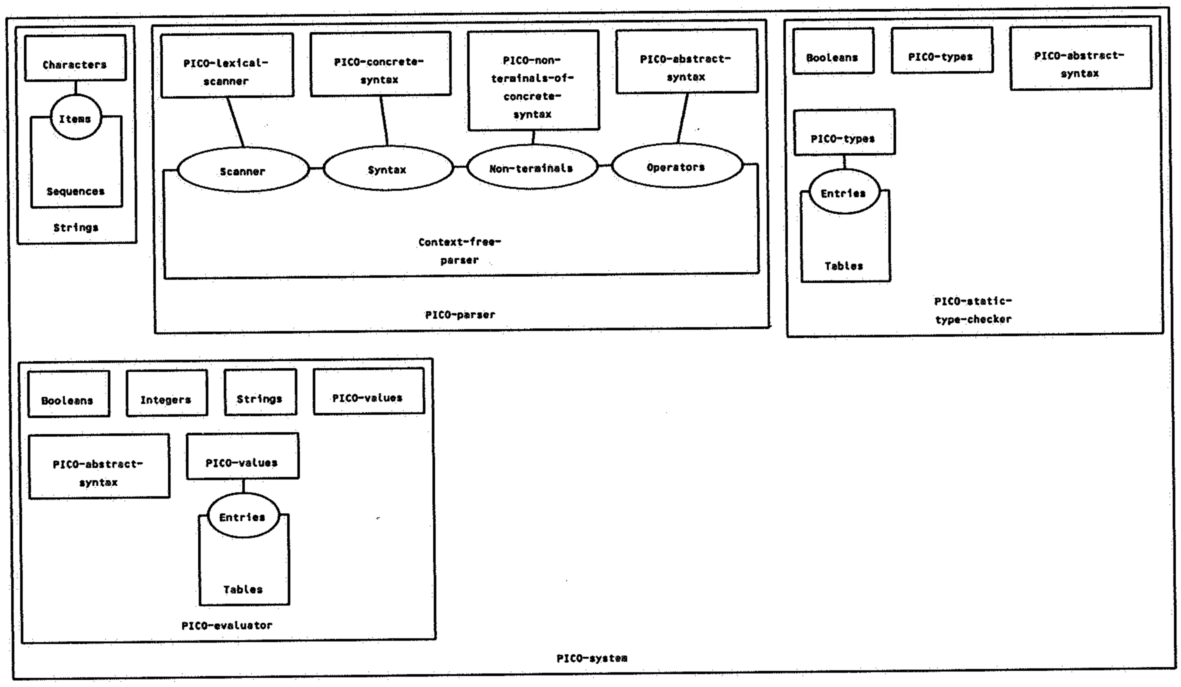
4.5.6. The PICO system

4.5.6.a. Global description

In this final section we combine all previously defined modules to form a PICO system. The top level function is `run` which converts, if this is possible, a string into a PICO-value. The following steps are necessary:

1) The input string is parsed and converted into an abstract syntax tree using `parse-and-construct` as defined in `PICO-parser`.
2) The types of the, syntactically correct, program are checked using `check` as defined in `PICO-static-type-checker`.
3) The, statically correct, program is evaluated using `eval` as defined in `PICO-evaluator`. If this evaluation terminates it produces a value-environment. The result of evaluating the original program is the final value of the variable `output` as extracted from this value-environment.

4.5.6.b. Structure diagram



4.5.6.c. Specification

```
module PICO-system
begin
   exports
        begin
           functions
                run:    STRING  -> PICO-VALUE
        end

   imports Strings, PICO-parser, PICO-static-type-checker, PICO-evaluator
```

```
functions
     run1: PICO-ATREE          -> PICO-VALUE
     run2: PICO-PROGRAM        -> PICO-VALUE

variables
     s        : -> STRING
     p        : -> PICO-ATREE
     abs-prog: -> PICO-PROGRAM
     has-output: -> BOOL
     v        : -> PICO-VALUE
     env      : -> VALUE-ENV

equations

[369]  run(s)                    = run1(parse-and-construct(s))

[370]  run1(error-pico-atree)    = error-value
[371]  run1(p)                   = if(check(pico-program(p)),
                                      run2(pico-program(p)),
                                      error-value)

[372]  run2(abs-prog)            = if(has-output, v, error-value)

                                   when  program-state(env) =
                                                 eval(program-state(abs-prog)),
                                         <has-output, v> =
                                                 lookup("output", env)

end PICO-system
```

5. LITERATURE

[AM85] America, P., "Definition of the programming language POOL-T", Report D 0091 85/09/09, Philips Research Laboratories, 9 September 1985.

[ABKR85] America, P., De Bakker, J.W., Kok, J.N. & Rutten, J., "Operational semantics of a parallel object-oriented language", Centre for Mathematics and Computer Science, Report CS-R8515, 1985.

[BHK84] Bergstra, J.A., Heering, J. & Klop, J.W., "Object-oriented algebraic specifications: proposal for a notation and 12 examples", Centre for Mathematics and Computer Science, Report CS-R8411, 1984.

[BIE84] Biebow, B., "Specification of a telephone subscriber connection unit using abstract algebraic data types in the language PLUSS", Laboratoire de Marcoussi, Centre de Recherche de la C.G.E., France, 1984.

[BK82] Bergstra, J.A. & Klop, J.W., "Conditional rewrite rules", Centre for Mathematics and Computer Science, Report IW198/82, 1982.

[BO81] Bothe, K., "Restructuring a compiler by abstract data types — an experiment in using abstractions for software modularization", Humboldt University Berlin, Seminar Bericht Nr. 40, 1981.

[BT79] Bergstra, J.A. & Tucker, J.V., "Algebraic specifications of computable and semicomputable data structures", Centre for Mathematics and Computer Science, Report IW 115/79, 1979.

[CAR85] Cardelli, L., "Basic polymorphic typechecking", Polymorphism, January 1985.

[DE84] Drosten, K. & Ehrich, H.-D., "Translating algebraic specifications to PROLOG programs", Informatik Bericht Nr. 84-08, Technische Universität Braunschweig, 1984.

[DM82] Damas, L. & Milner, R., "Principal type-schemes for functional programs", 9th Ann. ACM Symp. on Principles of Programming Languages, ACM, 1982, 207-212.

[EM85] Ehrig, H. & Mahr, B., *Fundamentals of Algebraic Specification*, Volume I, *Equations and Initial Semantics*, Springer-Verlag, 1985.

[FGJM85] Futatsugi, K., Goguen, J.A., Jouannaud, J.P. & Meseguer, J., "Principles of OBJ2", *Conf. Record 12th Ann. ACM Symp. Principles of Programming Languages*, ACM, 1985, pp. 52-66.

[GAU80] Gaudel, M.C., "Specification of compilers as abstract data type representations", in: Springer Lecture Notes in Computer Science, Volume 94, 1980.

[GAU84] Gaudel, M.C., "Introduction to PLUSS", draft document, Paris, 1984.

[GAN82] Ganzinger, H., "Denotational semantics for languages with modules", Proceedings of IFIP Working Conference *Formal Description of Programming Concepts*, North-Holland, 1982.

[GOR79] Gordon, M.J.C., *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[GP81] Goguen, J.A. & Parsaye-Ghomi, K., "Algebraic denotational semantics using parameterized abstract modules", in: Diaz, J. & Ramos, I. (eds.), *Formalizing Programming Concepts*, Springer Lecture Notes in Computer Science, Volume 107, 1981, 292-309.

[KLA83] Klaeren, H.A., *Algebraische Spezifikation: Eine Einführung*, Springer-Verlag, 1983.

[LOE84] Loeckx, J., "Algorithmic specifications: a constructive method for abstract data types", Report A84/03, Universität des Saarlandes, 1984.

[MG85] Meseguer, J. & Goguen, J.A., Initiality, induction, and computability, Preprint, Computer Science Laboratory, SRI International, n.d.; to appear in: Nivat, M., & Reynolds, J. (eds.), *Algebraic Methods in Semantics* (Cambridge University Press).

[OH80] Oppen, D.C. & Huet, G., "Equations and rewrite rules", in: R. Book (ed.), *Formal Languages: Perspectives and Open Problems*, Academic Press, 1980.

[W83] Wirsing, M., "A Specification Language", Dissertation, Münich University, 1983.

APPENDIX A.1. Dependency hierarchy of modules

Module	imports the modules
Atree-environments:	Tables
Atrees:	Booleans, Integers, Strings, Tokens
BNF-patterns:	Strings
Booleans:	-
Characters:	Booleans, Integers
Context-free-parser:	Atree-environments, BNF-patterns, Booleans, Strings, Token-sequences
Integers:	Booleans
PICO-BNF-patterns:	BNF-patterns
PICO-abstract-syntax:	Integers, PICO-atree-environments, PICO-types, Strings
PICO-atree-environments:	Atree-environments
PICO-atree-operators:	Booleans, Integers
PICO-atree-operators-of-lexical-syntax:	Booleans
PICO-concrete-syntax:	PICO-BNF-patterns, PICO-atree-environments
PICO-evaluator:	Booleans, Integers, PICO-abstract-syntax, PICO-values, Strings, Tables
PICO-lex-BNF-patterns:	BNF-patterns
PICO-lex-atree-environments:	Atree-environments
PICO-lexical-character-scanner:	Booleans, Characters, Strings, Token-sequences
PICO-lexical-scanner:	Context-free-parser
PICO-lexical-syntax:	PICO-lex-BNF-patterns, PICO-lex-atree-environments, Token-sequences
PICO-non-terminals-of-concrete-syntax:	-
PICO-non-terminals-of-lexical-syntax:	-
PICO-parser:	Context-free-parser
PICO-static-type-checker:	Booleans, PICO-abstract-syntax, PICO-types, Tables
PICO-system:	PICO-evaluator, PICO-parser, PICO-static-type-checker, Strings
PICO-types:	Booleans
PICO-values:	Booleans, Integers, Strings
Sequences:	Booleans
Strings:	Sequences
Tables:	Booleans, Strings
Token-sequences:	Sequences
Tokens:	Booleans, Strings

APPENDIX A.2. Declaration of sorts per module

Module	declares the sorts
Atree-environments:	-
Atrees:	ATREE, OPERATOR
BNF-patterns:	NON-TERMINAL, PATTERN
Booleans:	BOOL
Characters:	CHAR
Context-free-parser:	-
Integers:	INTEGER
PICO-BNF-patterns:	-
PICO-abstract-syntax:	DECLS, EXP, ID, PICO-PROGRAM, SERIES, STATEMENT
PICO-atree-environments:	-
PICO-atree-operators:	PICO-OPERATOR
PICO-atree-operators-of-lexical-syntax:	LEX-OPERATOR
PICO-concrete-syntax:	-
PICO-evaluator:	PROGRAM-STATE
PICO-lex-BNF-patterns:	-
PICO-lex-atree-environments:	-
PICO-lexical-character-scanner:	-
PICO-lexical-scanner:	-
PICO-lexical-syntax:	-
PICO-non-terminals-of-concrete-syntax:	PICO-NON-TERMINAL
PICO-non-terminals-of-lexical-syntax:	LEX-NON-TERMINAL
PICO-parser:	-
PICO-static-type-checker:	-
PICO-system:	-
PICO-types:	PICO-TYPE
PICO-values:	PICO-VALUE
Sequences:	ITEM, SEQ
Strings:	-
Tables:	ENTRY, TABLE
Token-sequences:	-
Tokens:	TOKEN

APPENDIX A.3. Declaration of functions per module

Module	declares the functions
Atree-environments:	_^_
Atrees:	atree, eq, error-atree, integer-atree, lexical-atree, null-atree, string-atree
BNF-patterns:	_+_, _|_, lexical, n, null-pattern, t
Booleans:	and, false, if, not, or, true
Characters:	char-0, char-1, char-2, char-3, char-4, char-5, char-6, char-7, char-8, char-9, char-A, char-B, char-C, char-D, char-E, char-F, char-G, char-H, char-I, char-J, char-K, char-L, char-M, char-N, char-O, char-P, char-Q, char-R, char-S, char-T, char-U, char-V, char-W, char-X, char-Y, char-Z, char-a, char-b, char-bar, char-c, char-colon, char-comma, char-d, char-e, char-equal, char-f, char-g, char-h, char-ht, char-i, char-j, char-k, char-l, char-lpar, char-m, char-minus, char-n, char-nl, char-o, char-p, char-plus, char-point, char-q, char-quote, char-r, char-rpar, char-s, char-semi, char-slash, char-space, char-t, char-times, char-u, char-v, char-w, char-x, char-y, char-z, eq, is-digit, is-letter, is-lower, is-upper, ord
Context-free-parser:	build, parse, parse-pat, parse-rule, rule, scan
Integers:	0, 1, 10, add, eq, greater, greatereq, less, lesseq, mul, succ
PICO-BNF-patterns:	-
PICO-abstract-syntax:	abs-assign, abs-conc, abs-decls, abs-empty-decls, abs-empty-series, abs-id, abs-if, abs-integer-constant, abs-pico-program, abs-plus, abs-series, abs-string-constant, abs-var, abs-while, append-statement, decls, exp, id, pico-program, pico-type-atree, series, statement
PICO-atree-environments:	-
PICO-atree-operators:	eq, op-assign, op-conc, op-decls, op-empty-decls, op-empty-series, op-id, op-if, op-integer-constant, op-integer-type, op-pico-program, op-plus, op-series, op-string-constant, op-string-type, op-var, op-while, ord
PICO-atree-operators-of-lexical-syntax:	eq, op-lex-item, op-lex-stream
PICO-concrete-syntax:	build, rule
PICO-evaluator:	eval, eval-decls, eval-exp, program-state
PICO-lex-BNF-patterns:	-
PICO-lex-atree-environments:	-
PICO-lexical-character-scanner:	char-scan, char-scan1, is-layout

PICO-lexical-scanner:	lex-scan
PICO-lexical-syntax:	build, lex-item, lex-stream, rule
PICO-non-terminals-of-concrete-syntax:	assign, conc, decls, empty-decls, empty-series, exp, id, id-type-list, if, integer-constant, non-empty-series, pico-program, plus, series, stat, string-constant, type, type-integer, type-string, var, while
PICO-non-terminals-of-lexical-syntax:	any-char-but-quote, assign-or-colon, concat, digit, digits, empty, empty-lexical-stream, ident, ident-char, ident-chars, integer-const, keyword-or-ident, layout, letter, lexical-item, lexical-stream, literal, non-empty-digits, non-empty-ident-chars, non-empty-lexical-stream, non-empty-string-tail, optional-layout, quote, string-const, string-tail
PICO-parser:	parse-and-construct
PICO-static-type-checker:	check, type-of-exp
PICO-system:	run, run1, run2
PICO-types:	eq, error-type, integer-type, string-type
PICO-values:	eq, error-value, pico-value
Sequences:	conc, conv-to-seq, eq, null, seq
Strings:	str-to-int
Tables:	delete, eq, error-entry, lookup, null-table, table
Token-sequences:	-
Tokens:	eq, token

APPENDIX A.4. Modules in which each function is declared

Function	is declared in module
0:	Integers
1:	Integers
10:	Integers
_+_:	BNF-patterns
_^_:	Atree-environments
_|_:	BNF-patterns
abs-assign:	PICO-abstract-syntax
abs-conc:	PICO-abstract-syntax
abs-decls:	PICO-abstract-syntax
abs-empty-decls:	PICO-abstract-syntax
abs-empty-series:	PICO-abstract-syntax
abs-id:	PICO-abstract-syntax
abs-if:	PICO-abstract-syntax
abs-integer-constant:	PICO-abstract-syntax
abs-pico-program:	PICO-abstract-syntax
abs-plus:	PICO-abstract-syntax
abs-series:	PICO-abstract-syntax
abs-string-constant:	PICO-abstract-syntax
abs-var:	PICO-abstract-syntax
abs-while:	PICO-abstract-syntax
add:	Integers
and:	Booleans
any-char-but-quote:	PICO-non-terminals-of-lexical-syntax
append-statement:	PICO-abstract-syntax
assign:	PICO-non-terminals-of-concrete-syntax
assign-or-colon:	PICO-non-terminals-of-lexical-syntax
atree:	Atrees
build:	Context-free-parser, PICO-concrete-syntax, PICO-lexical-syntax
char-0:	Characters
char-1:	Characters
char-2:	Characters
char-3:	Characters
char-4:	Characters
char-5:	Characters
char-6:	Characters
char-7:	Characters
char-8:	Characters

char-9:	Characters
char-A:	Characters
char-B:	Characters
char-C:	Characters
char-D:	Characters
char-E:	Characters
char-F:	Characters
char-G:	Characters
char-H:	Characters
char-I:	Characters
char-J:	Characters
char-K:	Characters
char-L:	Characters
char-M:	Characters
char-N:	Characters
char-O:	Characters
char-P:	Characters
char-Q:	Characters
char-R:	Characters
char-S:	Characters
char-T:	Characters
char-U:	Characters
char-V:	Characters
char-W:	Characters
char-X:	Characters
char-Y:	Characters
char-Z:	Characters
char-a:	Characters
char-b:	Characters
char-bar:	Characters
char-c:	Characters
char-colon:	Characters
char-comma:	Characters
char-d:	Characters
char-e:	Characters
char-equal:	Characters
char-f:	Characters
char-g:	Characters
char-h:	Characters
char-ht:	Characters
char-i:	Characters
char-j:	Characters

char-k:	Characters
char-l:	Characters
char-lpar:	Characters
char-m:	Characters
char-minus:	Characters
char-n:	Characters
char-nl:	Characters
char-o:	Characters
char-p:	Characters
char-plus:	Characters
char-point:	Characters
char-q:	Characters
char-quote:	Characters
char-r:	Characters
char-rpar:	Characters
char-s:	Characters
char-scan:	PICO-lexical-character-scanner
char-scan1:	PICO-lexical-character-scanner
char-semi:	Characters
char-slash:	Characters
char-space:	Characters
char-t:	Characters
char-times:	Characters
char-u:	Characters
char-v:	Characters
char-w:	Characters
char-x:	Characters
char-y:	Characters
char-z:	Characters
check:	PICO-static-type-checker
conc:	PICO-non-terminals-of-concrete-syntax, Sequences
concat:	PICO-non-terminals-of-lexical-syntax
conv-to-seq:	Sequences
decls:	PICO-abstract-syntax, PICO-non-terminals-of-concrete-syntax
delete:	Tables
digit:	PICO-non-terminals-of-lexical-syntax
digits:	PICO-non-terminals-of-lexical-syntax
empty:	PICO-non-terminals-of-lexical-syntax
empty-decls:	PICO-non-terminals-of-concrete-syntax
empty-lexical-stream:	PICO-non-terminals-of-lexical-syntax
empty-series:	PICO-non-terminals-of-concrete-syntax

eq:	Atrees, Characters, Integers, PICO-atree-operators, PICO-atree-operators-of-lexical-syntax, PICO-types, PICO-values, Sequences, Tables, Tokens
error-atree:	Atrees
error-entry:	Tables
error-type:	PICO-types
error-value:	PICO-values
eval:	PICO-evaluator
eval-decls:	PICO-evaluator
eval-exp:	PICO-evaluator
exp:	PICO-abstract-syntax, PICO-non-terminals-of-concrete-syntax
false:	Booleans
greater:	Integers
greatereq:	Integers
id:	PICO-abstract-syntax, PICO-non-terminals-of-concrete-syntax
id-type-list:	PICO-non-terminals-of-concrete-syntax
ident:	PICO-non-terminals-of-lexical-syntax
ident-char:	PICO-non-terminals-of-lexical-syntax
ident-chars:	PICO-non-terminals-of-lexical-syntax
if:	Booleans, PICO-non-terminals-of-concrete-syntax
integer-atree:	Atrees
integer-const:	PICO-non-terminals-of-lexical-syntax
integer-constant:	PICO-non-terminals-of-concrete-syntax
integer-type:	PICO-types
is-digit:	Characters
is-layout:	PICO-lexical-character-scanner
is-letter:	Characters
is-lower:	Characters
is-upper:	Characters
keyword-or-ident:	PICO-non-terminals-of-lexical-syntax
layout:	PICO-non-terminals-of-lexical-syntax
less:	Integers
lesseq:	Integers
letter:	PICO-non-terminals-of-lexical-syntax
lex-item:	PICO-lexical-syntax
lex-scan:	PICO-lexical-scanner
lex-stream:	PICO-lexical-syntax
lexical:	BNF-patterns
lexical-atree:	Atrees
lexical-item:	PICO-non-terminals-of-lexical-syntax

pico-type-atree:	PICO-abstract-syntax
pico-value:	PICO-values
plus:	PICO-non-terminals-of-concrete-syntax
program-state:	PICO-evaluator
quote:	PICO-non-terminals-of-lexical-syntax
rule:	Context-free-parser, PICO-concrete-syntax, PICO-lexical-syntax
run:	PICO-system
run1:	PICO-system
run2:	PICO-system
scan:	Context-free-parser
seq:	Sequences
series:	PICO-abstract-syntax, PICO-non-terminals-of-concrete-syntax
stat:	PICO-non-terminals-of-concrete-syntax
statement:	PICO-abstract-syntax
str-to-int:	Strings
string-atree:	Atrees
string-const:	PICO-non-terminals-of-lexical-syntax
string-constant:	PICO-non-terminals-of-concrete-syntax
string-tail:	PICO-non-terminals-of-lexical-syntax
string-type:	PICO-types
succ:	Integers
t:	BNF-patterns
table:	Tables
token:	Tokens
true:	Booleans
type:	PICO-non-terminals-of-concrete-syntax
type-integer:	PICO-non-terminals-of-concrete-syntax
type-of-exp:	PICO-static-type-checker
type-string:	PICO-non-terminals-of-concrete-syntax
var:	PICO-non-terminals-of-concrete-syntax
while:	PICO-non-terminals-of-concrete-syntax

D5 - GLOBAL OUTLINE OF AN ENVIRONMENT GENERATION SYSTEM

D5.A1 - PARTIAL EVALUATION AND ω-COMPLETENESS OF ALGEBRAIC SPECIFICATIONS

GIPE : CEC 348/A/T5/1
version 6 January 1986

Global Outline of an Environment Generation System

Deliverable D5 of Task T5 — Second Review —

J. Heering (CWI)
P. Klint (CWI)
B. Lang (INRIA)
A. Verhoog (BSO)

An outline is given of an environment generator in which the results of Esprit Project 348 can be incorporated. The functional aspects of the generated environments, and the functional aspects and organization of the environment generation system itself are discussed.

1. INTRODUCTION

This note gives a rough outline of an "Environment Generator" in which the results of ESPRIT project 348 could eventually be incorporated. The envisaged system is based on *language definitions*. These are sufficiently detailed to enable the system to derive programming environments from them permitting editing, analysis and execution of programs in the corresponding languages.

The word "language" should here be understood as meaning a language (for instance, for expressing programs, specifications or database queries), characterized by an *abstract syntax*, i.e. a standard syntactical representation of its programs by means of labelled trees. Each language may possess a semantics and a variety of properties or representations that have to be specified to the generator.

New languages can be defined by means of a *Language Definition Formalism* (LDF). See deliverable D4 for a description of this formalism. A language definition in LDF should at least define the abstract syntax of the language. Other aspects such as concrete syntax, rules for pretty-printing, static (type) constraints and dynamic semantics may be specified according to need or feasibility.

2. FUNCTIONAL ASPECTS OF THE ENVIRONMENTS GENERATED BY THE SYSTEM

The functions of the environment generated for a language L depend on the nature, i.e. semantic intent, of L. L may be an executable programming language or a language for defining abstract data types, but it may also be a specification language with a purpose other than programming (e.g. specification of a production chain in a factory, or a VLSI circuit) or with a purely descriptive purpose (e.g. a language for describing chemical formulae).

We should therefore not set a limit to the kind of facilities that should ultimately be made available in a generated environment. However, within the current aims of the project, we shall confine ourselves to the study of languages related to programming.

The backbone of the system is a collection of user-defined languages $L_1, \ldots, L_n$. For each L_i

there exists an environment generated by the system for manipulating a (dynamically changing) collection P_{L_i} of *programs* in L_i.

In the sequel, we will use the generic name *program* to mean a complete or incomplete L_i-program, i.e. whose text can potentially be derived from the start symbol of the L_i-grammar, but with some parts not yet filled in. We will reserve the word *subprogram*(1) to mean a part of a *program*, i.e. whose text can potentially be derived from *another* nonterminal than the start symbol of the L_i-grammar. The notion of "data", i.e. values of certain types created and used during program execution, is covered by this definition of subprogram. With both programs and subprograms certain system-related information will be associated such as, for instance

- The name under which this program will be stored.
- The language L_i in which it is written.
- The L_i-program itself (constructed via edit operations).
- Cursor(s) pointing to subprogram(s) of the L_i-program (used and set by all functions for searching and editing).
- Static semantic function *check*, a function derived from the definition of L_i that incrementally checks the static semantic constraints; *check* is invoked after each elementary editing operation.
- Evaluation function *eval*, a function derived from the definition of L_i that incrementally evaluates L_i-programs.
- Other attributes such as creation date, owner, version, etc. (to be specified).

Many issues still have to be addressed. We now only discuss the *editing model* to be used in the generated environment and the *interactive operation* of the various tools in the environment.

The two prevailing editing models are *pure text editing* and *pure syntax-directed editing*. The former gives the user flexible editing operations but does not assist in constructing syntactically correct programs. The latter supports the construction of syntactically correct programs, but this often goes at the expense of the power of the editing operations. Experience shows that pure syntax-directed editing is too restrictive. One should therefore strive for a compromise between the two models by extending the text editing model with syntax-directed operations (e.g. positioning the cursor on the enclosing construct, finding the next if-statement, etc.) while retaining the character-level positioning and editing operations.

The generated environments are *interactive*. This implies that all commands and tools available should give the user as much feedback as possible about their progress and about the errors encountered so far. Examples are interactive checking of syntax and of type constraints. The interactive operation of tools can be realized by using (or generating) incremental algorithms for them. This issue is further discussed in sections 4 and 5.

In the following subsections we briefly summarize the kind of tools that may (ultimately) be incorporated in the generated environments. The aim of this summary is to make an inventory of potentially useful tools and to anticipate their inclusion in the generated environments in a later stage of the project.

2.1. Editing tools

The environment generated for language L supports syntax-directed editing of L-programs. The editing primitives supplied are not only accessible through a standard user-interface but are also callable from programming languages. (2) This allows the mechanization of repeatedly occurring standard manipulations such as, e.g., adding standard comments to procedure headings. The editing tools include:

(1) Not to be confused with the more conventional meaning ("procedure") of this word.

(2) Note that this is a requirement on the available operations in the programming languages to be used for the implementation of the system.

- navigation primitives for traversing the syntactic structure of a program;
- addressing primitives for memorizing positions in this structure;
- modification and construction primitives;
- pattern-matching primitives for structural search or identification (this includes pattern-directed modification of programs).
- help facilities.

2.2. Checking tools

These tools verify the internal or global consistency of programs. This includes checking of syntactic correctness and of various static (type) constraints. Checks on adherence to a standard programming style (such as style of comments, naming conventions for variables, etc.) also fall into this category. Other tools may check the mutual consistency of several programs related to the same project (including documentation and specification).

2.3. Analysis tools

The structure of programs may be analyzed and summarized in (machine-readable) tables or diagrams. This includes structural information concerning:

- data flow;
- cross references;
- aliases;
- side-effects.

Other tools in this category may perform the quantitative measurement of the static complexity of programs ("software metrics").

2.4. Dynamic semantics tools

These tools are related to the actual execution of programs:

- interpreters;
- compilers;
- optimizers;
- symbolic debugging and tracing;
- measurement of the dynamic behaviour of programs.

2.5. Filing tools

The current version of all programs and data created in the environment is automatically filed. Optionally, older versions may be retained as well (version control).

2.6. Functions available in the generated environments

The realization of several of the functional aspects described in sections 2.1-2.5 requires a mechanism for annotating programs with a variety of information such as assertions, cross-references or dynamic measurements of program execution. This annotation mechanism is not yet reflected in the current outline.

We now give a -- very tentative -- list of functions that will be available in a generated system.

2.6.1. Management of programs

Create (name, language, nonterminal)
create a new (sub)program *name*, derivable from a given *nonterminal* in a given *language*.

Remove (name)
remove the (sub)program *name*.

Rename (*name* 1,*name* 2)
give name *name* 2 to the (sub)program which currently has name *name* 1.

Print (*name* 1,*name* 2)
Print (sub)program *name* 1 on external text file *name* 2.

Read (*name* 1,*name* 2)
Read from the text file *name* 2 a (sub)program and give it name *name* 1.

Get-attribute (*name*,*attribute*)
get value of *attribute* of (sub)program *name* (see 2.1.).

Set-attribute (*name*,*attribute*,*value*)
set value of *attribute* of (sub)program *name* to *value*.

2.6.2. Cursor movement in programs

The following functions move the cursor in terms of the syntactic structure of a *program*. The notion of "structure" is language dependent, i.e. *Cursor-up* may, for instance, either mean "go to the parent node" (while editing a Pascal-program) or "go to the character on the previous line in the current column" (while editing text). Issues to be resolved:

- The editing model.
- Does editing require more information than contained in the syntax section of the language definition. For instance, should cursor movements be expressible in LDF?

Cursor-up (*program*)

Cursor-left (*program*)

Cursor-right (*program*)

Cursor-down (*program*)

2.6.3. Searching in programs

Search (*program*,*nonterminal*)
Search for the next occurrence of a *nonterminal* in *program* and place the cursor there.

Search (*program*,*string*)
Search for the next occurrence of *string* in (the printed image of) *program* and place the cursor there. Note: the precise meaning of this function depends on the editing model.

Search (*program*,*subprogram*)
Search for the next occurrence of *subprogram* in *program* and place the cursor there.

2.6.4. Editing of programs

Insert (*program*,*subprogram*)
Insert *subprogram* in *program* at the current cursor position; *subprogram* should be a syntactically legal insertion. After the insertion *check* (*program*) is evaluated, where *check* is the incremental type check function associated with *program*.

Insert-as-text (*program*,*string*)
Parse *string* and insert the resulting program in *program* at the current cursor position. After the insertion *check* (*program*) is evaluated.

Copy (*program*,*name*)
Make a copy of the subprogram of *program* at the current cursor position and give this copy the name *name*.

Cut (*program*)
Remove the subprogram of *program* at the current cursor position. After this *check* (*program*) is evaluated.

2.6.5. Evaluation of programs

Evaluate(*program*)
: Evaluates *eval*(*program*), where *eval* is the incremental evaluation function associated with *program.*

3. DEFINITION OF A LANGUAGE IN LDF

The definition of a language must always specify its abstract syntax. It defines a tree-structured representation for programs which is their standard representation within the environment. All other aspects of the language are defined with respect to this standard representation.

The central component of the specification language LDF should therefore be the formalism for defining the abstract syntax of new languages. Other sections in a LDF-definition define other aspects of languages and environments as sketched in the previous section.

Some of these sections can already be identified:

(1) A *syntax section* containing a definition of the abstract and concrete syntax (including lexical syntax and pretty printing) of the language to be defined.
(From this part the programming environment generator has to derive a syntax-directed editor.)

(2) A *static constraints* section containing a definition of the *type constraints* or *static semantics* of the language.
(From this part the programming environment generator has to derive an incremental type-checker.)

(3) A definition of the *(dynamic) semantics* of the language.
(From this part the programming environment generator has to derive an incremental evaluator and, at a later stage, a compiler.)

The nature of other sections can not yet be made precise, since:

- The project aims at exploring the potentials of a variety of specification formalisms and should therefore provide a general framework in which these formalisms can be incorporated. Any bias in the direction of a particular specification formalism is premature.
- Different specification formalisms may be more appropriate for the definition of certain aspects of a language than others, e.g. BNF notation is appropriate for describing concrete syntax, while abstract algebra or denotational semantics are appropriate for describing semantics.
- There is no a priori limit to the number and kind of properties (and associated processors) one may want to specify for a given language.

Thus the specification language LDF must have an extensible structure and allow the simultaneous use of different specification methods within one overall framework. The specification of a language *L* then consists of the definition of its abstract syntax and specifications of particular aspects of *L* each using an appropriate specification technique. All these specifications are organized around the specification of the abstract syntax of *L*.

Some remaining issues to be resolved are:

- How are the semantic relationships expressed between the various parts of a language definition (each using its own specification method).
- Which different "views" of language definitions should be made available in order to display the semantic structure of the defined language. One can, for instance, imagine a "programming language manual"-view which collects all parts of the language definition related to the specification of a particular language construct or a "structural"-view that gives high level information on the modular structure of the language definition.

More detailed information on the language definition formalism is given in deliverable D4.

4. FUNCTIONAL ASPECTS OF THE ENVIRONMENT GENERATION SYSTEM

The system for the generation of environments supports:

(1) the management of language definitions (including creation, modification, filing and composition of language definitions), and

(2) the generation of programming environments from these language definitions.

4.1. Management of language definitions

The environment generation system supports the creation, syntax-directed editing, filing and composition of language definition modules written in the fixed language definition formalism LDF. Note that when L_i is modified the programs in P_{L_i} may become inconsistent, in other words, modification of a language definition has repercussions for the programs that were filed in an environment generated from an older version of that language definition.

4.2. Generation of environments

The environment generation system supports the compilation of language definitions into programming environments. This consists of:

- Generation of a syntax-directed editor on the basis of the information in a language definition. Issues: (1) generation of help and explain facilities; (2) error handling.
- Generation of an incremental static constraints checker. The specification of static constraints in the language definition is probably "batch-oriented", i.e. the constraints are specified for *complete* programs but it is not specified how these constraints are affected after *modification* of a program. If possible, an incremental checker for the static constraints has to be derived from this non-incremental specification and its cooperation/synchronization with the syntax-directed editor has to be realized.
- Transformation of the definition of dynamic semantics into an incremental evaluator. Issues: (1) impose restrictions on definitions to allow this transformation; (2) how is a non-incremental specification transformed into an incremental evaluator; (3) allow the implementation of functions defined in a language definition by means of functions written in some programming language.

5. ORGANISATION OF THE ENVIRONMENT GENERATION SYSTEM

5.1. Management of language definitions

As stated in the previous section the environment generation system should allow editing, composition and filing of language definitions. We do not make any assumptions about the way this editing and filing facility is realized. An obvious approach is to bootstrap the system, i.e. to generate the LDF-environment using the environment generation system itself.

5.2. Generation of environments

The environment generation system is organized around the notion of *compiling* the various parts of an LDF-definition of language L into operational components of the L-environment.

Two major ways of compiling specifications can be distinguished:

(1) compile (part of) the specification into an executable *program* which is included in the generated environment (see figure 1).

(2) compile (part of) the specification into *tables* and include these tables in the generated environment together with interpreters for them (see figure 2).

Alternative (2) is a refinement of the (more general) alternative (1).

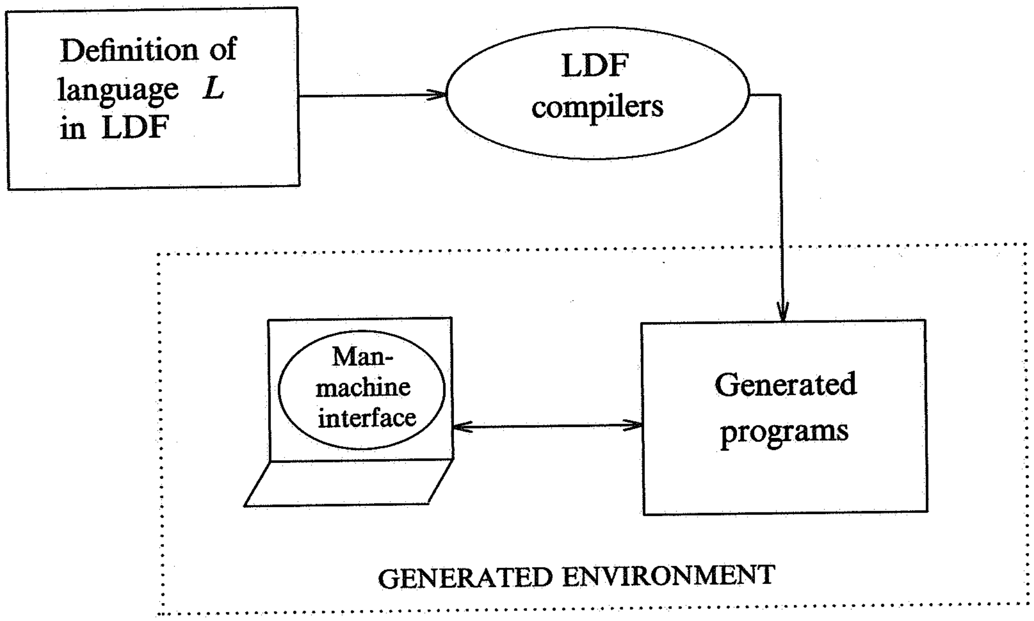


Figure 1. Alternative 1 -- generation of programs.

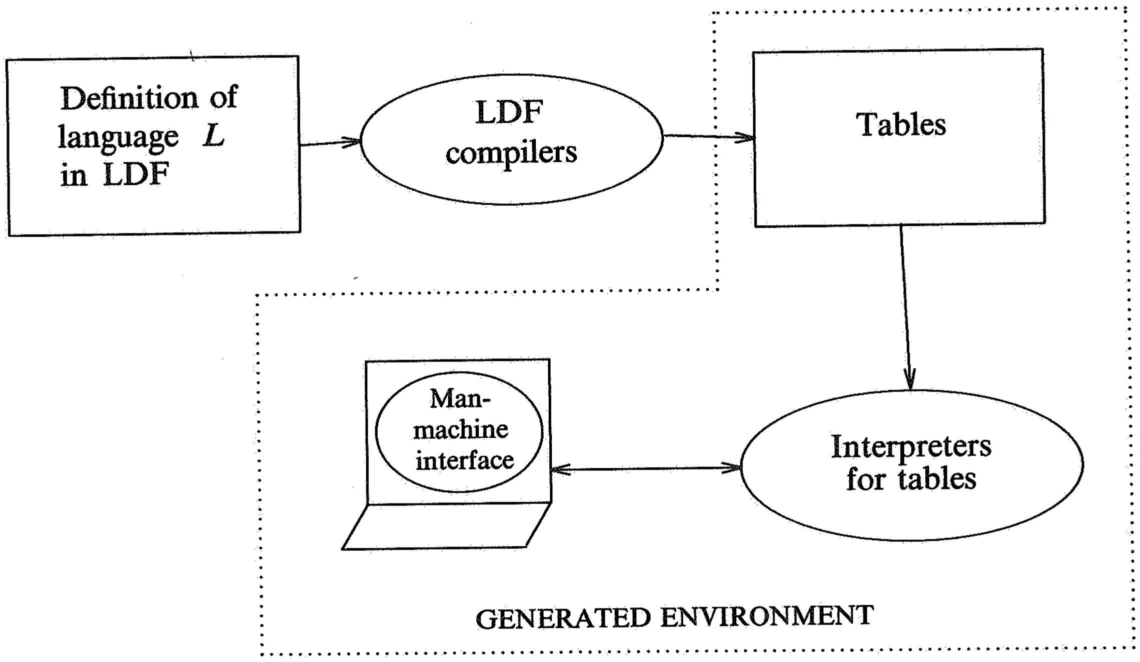


Figure 2. Alternative 2 -- generation of tables.

Both the language definition and the tables/programs have a modular structure. Distinct parts of the specification may be based on different specification techniques. Similarly the tables/programs correspond to different operational aspects of the language specified.

Alternative (2) has the advantage that one physical copy of an interpreter can be shared by all generated environments: the generated environment only has to contain the tables to be interpreted by the shared interpreter. This alternative has the disadvantage of introducing an additional level of interpretation. In alternative (1) specialized code is generated (eliminating interpretation overhead), but no code can be shared by different generated environments.

Another advantage of alternative (2) is that different compilers may produce tables for the same interpreter. Each table is produced by a compiler from one or several components of a specification. There need not be a one to one relationship between the components of the specification and the corresponding tables. One component may produce several tables through different compilers, either alone or together with some other component. Similar (or even identical) tables may be produced for the same interpreter, from different specification components with the same semantical intent. Similarly there is no one to one relationship between tables and interpreters, and several tables pertaining to distinct aspects of the formalism may be interpreted by the same interpreter.

In general, no choice can be made between these two alternatives and the system will have to provide facilities for both of them. In the sequel, we will refer to either of these alternatives, without selecting one, as "tables/programs".

A minimal environment generation system consists of:

(1) An LDF-compiler which compiles an LDF-definition of language *L* into tables/programs. Necessary components in the generated system are:

- editor tables/programs (includes the syntax derived from *L* and constructor functions for nodes in the abstract syntax tree);
- unparser (the prettyprinter derived from *L*);
- parser (the parser derived from the grammar of *L*);
- type checker (operational version of static semantics of *L*);
- evaluator (operational version of dynamic semantics of *L*).

(2) A file system for *L*-programs and *L*-data.

(3) A command interpreter/man-machine interface for accessing the generated environment.

To a first approximation, a generated environment for language *L* is composed of a set of tools, plus a fixed man-machine interface. Part of this interface is derivable from the syntax section of language definition (editing menus, pretty-printers, on-line help, etc.). However, additional aspects of the man-machine interface (such as more advanced help or tracing facilities) could be described in a separate section of the LDF-definition and the interface could then be derived from it in a similar way as the other tools.

It was already stated in section 2 that the generated environments are *interactive* and the generated tools should therefore operate *incrementally*. The tools therefore need information on incremental modifications due to, for instance, editing. An essential problem yet to be solved concerns the communication of this information between the various tools.

6. THE ROLE OF PARTIAL EVALUATION IN THE ENVIRONMENT GENERATION SYSTEM

Perhaps the biggest single problem confronting us in implementing an environment generation system is compiling language definitions to reasonably efficient code. On the part of CWI it is felt that *partial evaluation* or *mixed computation* [ER82, JSS85] is a potentially important optimization technique which may help us in achieving this goal. Although rather vague in scope, partial evaluation is basically a form of constant propagation. Suppose $P(x,y)$ is a program with two arguments, whose first argument has a known value c, but whose second argument is still unknown. Partial evaluation of $P(c,y)$ with unbound y results (or rather: should result) in a specialized residual program $P_c(y)$ in which "as much as possible" has been computed on the basis of c. For instance, if P is a general context-free parser having as arguments a grammar and a string, partial evaluation of P with known grammar G (which in the environment generation system would be part of a language definition) and unknown string should lead to a specialized parser P_G by propagating G in P.

Annexe D5.A1 is a first theoretical study of partial evaluation in the context of initial algebra specification and term rewriting systems. Automatic (partial) ω-enrichment of algebraic specifications (Annexe D5.A1) and the use of partial evaluation for generating compilers from algebraic specifications are currently being investigated.

7. LITERATURE

[ER82] A.P. Ershov, "Mixed computation: potential applications and problems for study", *Theoretical Computer Science*, **18** (1982), pp. 41-67.

[FGJM85] K. Futatsugi, J.A. Goguen, J.P. Jouannaud & J. Meseguer, "Principles of OBJ2", *Conf. Record 12th Ann. ACM Symp. Principles of Programming Languages*, ACM, 1985, pp. 52-66.

[GAN85] Special issue devoted to the GANDALF system, *The Journal of Systems and Software*, **5** (1985), 2.

[HEE83] J. Heering, "Taaldefinities als kern voor een programmeeromgeving" ("A programming environment based on language definitions") (in Dutch), in: J. Heering & P. Klint (Eds.), *Colloquium Programmeeromgevingen*, CWI Syllabus 30, 1983.

[JSS85] N.D. Jones, P. Sestoft & H. Søndergaard, An experiment in partial evaluation: the generation of a compiler generator, Report 85/1, Institute of Datalogy, University of Copenhagen, 1985.

[KL83] P. Klint, A survey of three language independent programming environments, Report IW 240/83, CWI, 1983.

[MMV85] B. Mélèse, V. Migot & D. Verove, The Mentor-V5 documentation, Report No. 43, INRIA Rocquencourt, 1985.

[RTD83] Th. Reps, T. Teitelbaum & A. Demers, "Incremental context-dependent analysis for language-based editors", *ACM Trans. Programming Languages and Systems*, **5** (1983), 3, pp. 449-477.

[SN85] G. Snelting, "Experiences with the PSG - Programming System Generator", in: *Formal Methods and Software Development*, TAPSOFT Proceedings, Vol. 2, LNCS 186, Springer-Verlag, 1985, pp. 148-162.

GIPE: **CEC 348/A/T5/2**
November 1985

PARTIAL EVALUATION AND ω-COMPLETENESS OF ALGEBRAIC SPECIFICATIONS

Annexe D5.A1 of Deliverable D5 — Second Review —

Jan Heering
Centre for Mathematics and Computer Science

Suppose $P(x,y)$ is a program with two arguments, whose first argument has a known value c, but whose second argument is not yet known. *Partial evaluation* of $P(c,y)$ results (or rather: should result) in a specialized residual program $P_c(y)$ in which "as much as possible" has been computed on the basis of c. In the literature on partial evaluation this is often more or less loosely expressed by saying that partial evaluation amounts to "making maximal use of incomplete information." In this paper a precise meaning is given to this notion in the context of equational logic, initial algebra specification and term rewriting systems. If maximal propagation of incomplete information is to be achieved within this context, as a first step it is necessary to add equations to the algebraic specification in question until it is ω-*complete* (if ever). The basic properties of ω-complete specifications are discussed, and some examples of ω-complete specifications as well as of specifications that do not have a finite ω-complete enrichment are given.

Note: This is a revised version of report CS-R8501, Centre for Mathematics and Computer Science, 1985. It will be published in *Theoretical Computer Science,* 43 (1986).

1. Introduction

1.1. Partial evaluation

The current investigation was inspired by the notion of *partial evaluation* or *mixed computation* as discussed for instance by Ershov [6] (who gives many references), Komorowski [12], and Jones *et al.* [11]. Although rather vague in scope, partial evaluation is basically a form of constant propagation. Suppose $P(x,y)$ is a program with two arguments, whose first argument has a known value c, but whose second argument is still unknown. Partial evaluation of $P(c,y)$ with unbound y results (or rather: should result) in a specialized residual program $P_c(y)$ in which "as much as possible" has been computed on the basis of c. For instance, if P is a general context-free parser having as arguments a grammar and a string, partial evaluation of P with known grammar G and unknown string should lead to a specialized parser P_G by propagating G in P.

Partial evaluation is first and foremost an important unifying concept, shedding light on the relationship between interpretation and compilation, on the possible meaning of an ill-defined term like *compile-time*, on program optimization and program generators in general, and on type checking. Secondly, it is a useful technique in strictly limited and well-defined contexts in which the axioms and rules required can be hand-tailored to the application at hand.

The notion of "computing as much as possible on the basis of incomplete information" is widespread in the partial evaluation literature. As Ershov puts it ([6], p. 49): "A well-defined mixed computation which in a sense makes *a maximal use* of the information contained in the bound argument yields a rather efficient residual program." And Komorowski says ([12], p. 59): "Partial evaluation is a case of program transformation. It attempts to improve efficiency of program execution by eliminating run-time checks and *performing as much computation in advance as possible*. However, it does not modify algorithms." (Emphasis added in both cases.)

When experimenting with partial evaluation in the context of term rewriting systems (Huet & Oppen [10]), one quickly discovers that making maximal use of incomplete information or computing as much in advance as possible is very difficult or even impossible. The rewrite rules used to evaluate *closed* (i.e. variable-free) terms are usually found to be inadequate when applied to *open* terms (i.e. terms containing variables) and numerous new and more general rules have to be added if anything like a canonical or in some sense simplest form is to be reached. Suppose, for example, that the following simple term rewriting system R for a function *max* on the natural numbers with constant 0 and successor function S is given (with $1=S(0)$):

$$\begin{aligned} &max(0,x) \rightarrow x \\ &max(x,0) \rightarrow x \\ &max(S(x),S(y)) \rightarrow S(max(x,y)). \end{aligned}$$

Partial evaluation of

$$max(max(1,1),x)$$

to

$$max(1,x)$$

requires no new rewrite rules, but for

$$max(max(1,x),1)$$

the same result can only be obtained by applying the commutative and associative properties of *max*, which are not needed for the evaluation of closed *max*-terms. Similarly, R is unable to reduce $max(x,x)$ to x or $max(S(x),x)$ to $S(x)$. In a larger context this implies that a term like

$$\textbf{if } max(x,x) = x \textbf{ then } E \textbf{ else } E' \textbf{ fi}$$

cannot be reduced to E. This may block yet another reduction, and so on.

In general, the additional rewrite rules required correspond to valid equations from the viewpoint of initial algebra semantics (Meseguer & Goguen [15]). In principle, new rules have to be added as long as the term rewriting system is incomplete with respect to the equational theory of the initial algebra in question. If, as a first step, one considers equations instead of rewrite rules, this means that new equations have to be added until the equational specification is complete with respect to the equational theory of the initial algebra (if ever), i.e. until the equational specification is *ω-complete*. As a second step one then has to consider the compilation of ω-complete specifications to term rewriting systems. The latter step falls outside the scope of this paper.

1.2. Algebraic specification, equational logic, and initial algebra semantics - some basic facts

In this section I give a brief summary of some basic facts of algebraic specification theory which are essential to an understanding of what follows. Good references are Burstall & Goguen's introductory paper [2] and Meseguer & Goguen's survey [15].

An algebraic specification S consists of two parts:

(i) a many-sorted *signature* Σ_S, defining a language of strongly typed *terms* (expressions), and

(ii) a set E_S of *equations* (identities) between Σ_S-terms, defining an *equational theory* consisting of all eqúations provable from E_S by means of many-sorted *equational logic.*

The rules of inference of equational logic are essentially the rules of *reflexivity*, *symmetry*, *transitivity*, and *substitution*. Two more rules are needed if Σ_S has void sorts - see §4.3 of Meseguer & Goguen [15].

Models of algebraic specifications are many-sorted algebras A such that (the interpretations of) all equations in E_S are valid in A. This is the well-known Tarski-semantics, but generalized to the many-sorted case.

If a Σ_S-equation is valid in *all* models of S, it is provable from E_S by means of equational logic.

This is the *completeness property* of many-sorted equational logic. In general, however, one is not interested in the full class of models of an algebraic specification, but only in a single model (or isomorphism class of models) which is isomorphic to the algebra (the data type) one wishes to specify. The model closest to ordinary programming practice is the *initial algebra* I_S which is characterized by the following two properties:

(i) Every element of I_S corresponds to at least one closed Σ_S-term ("no junk").

(ii) I_S is maximally free, which means that elements of I_S are never equal unless the corresponding closed terms can be proved equal from E_S ("no confusion").

Every algebraic specification (without void sorts) has an initial algebra which is uniquely determined up to isomorphism.

1.3. ω-completeness of algebraic specifications

Because of the "no junk" property, the initial algebra I_S of an algebraic specification S almost always has a much richer equational theory than can be derived from the equations E_S of S by means of equational logic alone, i.e. in general equational logic is not complete with respect to the initial algebra. Although the closed equations valid in I_S can always be proved from E_S using equational reasoning, open equations valid in I_S do not in general yield to such simple means of deduction, but require stronger rules of inference (such as structural induction) for their proofs. For instance, consider the following specification:

module *BOOL*
begin
  sort *bool*

functions	$F, T\colon \rightarrow bool$	(false, true)
	$\neg\colon bool \rightarrow bool$	(not)
	$+\colon bool \times bool \rightarrow bool$	(exclusive-or)
	$.\,, \vee\colon bool \times bool \rightarrow bool$	(and, or)

equations $\neg F = T$
$\neg T = F$

$T + F = F + T = T$
$F + F = T + T = F$

$T.T = T$
$T.F = F.T = F.F = F$

$T \vee T = T \vee F = F \vee T = T$
$F \vee F = F$

end *BOOL.*

The initial model I_{BOOL} is a Boolean algebra with two elements. Because every closed term over Σ_{BOOL} is equal to T or F, proving the validity in I_{BOOL} of the laws of Boolean algebra (such as De Morgan's laws and the commutativity and associativity of $+$, $.$ and $\vee$) amounts to checking a finite number of closed instances for each law to be proved. These laws are not provable from E_{BOOL} by means of equational reasoning, however, as can easily be seen by constructing a model of *BOOL* in which they are false.

Completeness with respect to the equational theory of the initial algebra can be obtained in full generality by adding the so-called *ω-rule* to equational logic. This infinitary rule of inference allows one to infer an open Σ_S-equation e from a (possibly infinite) set of premises consisting of the closed Σ_S-instances of e. Using this extended version of equational logic, the equations valid in the initial algebra I_S can always be proved from E_S (even if they are not recursively enumerable!). Adding the ω-rule to equational logic has the general effect of making the class of models of a specification smaller and of highlighting the role of the initial model.

The ω-rule is rather unwieldy and the question arises whether it is possible to achieve completeness of a specification with respect to the equational theory of its initial algebra without transcending the limits of purely equational reasoning. More specifically, given a specification S, is it possible to add equations to it in such a way that (i) the initial algebra is not affected, and (ii) all open equations valid in the initial algebra become provable by purely equational means?

I shall call a specification having property (ii) *ω-complete.* I shall discuss the basic properties of non-parameterized ω-complete specifications (§2), give some examples (§3), and, finally, sketch an approach towards automatic addition of significant new equations valid in the initial algebra, i.e. automatic (partial) ω-enrichment (§4).

1.4. Related work
While revising this paper for publication, it was brought to my attention that the notion of ω-completeness as discussed in this paper was investigated by Paul [17] in the context of "inductionless induction" under the name *inductive completeness.** Paul gives several examples of inductively complete algebraic specifications and their compilation to complete term rewriting systems (§§3.1-3.2 of this paper). He also shows that some specifications do not have a finite inductive closure, i.e. no finite ω-complete enrichment.

Taylor's survey [19] gives pointers to relevant work on (non-)finitely based algebras done in the context of universal algebra, while Davis *et al.* [5] and Henkin [7] discuss the equational theory of the natural numbers with addition, multiplication, and various other functions (§3.1 of this paper). Plotkin [18] has shown that the $\lambda K\beta\eta$-calculus is ω-incomplete (§3.4 of this paper).

Because the terminology in this field is rather confusing a brief comparative list of terms used by various authors may be helpful:

Inductive completeness (Paul [17])	= *ω-completeness* (this paper)
Inductive closure (Paul [17])	= *ω-complete enrichment* (this paper)
Inductive closure (Paul [17])	≠ *Inductive closure* (Nourani [16])
Inductive completion (Huet & Hullot [9])	= *Inductionless induction* (§6.7 of Meseguer & Goguen [15])
Inductive completion (Huet & Hullot [9])	≠ *Inductive closure* (Paul [17])
Inductive completion (Huet & Hullot [9])	≠ *Inductive closure* (Nourani [16])

2. THE ω-COMPLETENESS PROPERTY

Provable will always mean *provable by purely equational means* unless otherwise noted. Only finite specifications are considered. The semantics of a specification will always be the initial algebra semantics.

DEFINITION 2.1: A finite algebraic specification S with signature Σ_S and set of Σ_S-equations E_S is *ω-complete* if every open equation all of whose closed Σ_S-instances are provable from E_S is itself provable from E_S.

THEOREM 2.1: An algebraic specification S is ω-complete if and only if all equations valid in its initial algebra I_S are provable from E_S.

PROOF: For any S the closed equations valid in I_S are precisely the closed equations provable from E_S. Hence, the open equations valid in I_S are precisely the equations all of whose closed instances are provable from E_S. Hence, S is ω-complete if and only if not only every closed equation but also every open equation valid in I_S is provable from E_S. □

THEOREM 2.2: The equations valid in the initial algebra I_S of an ω-complete specification S are valid in all other models of S as well.

* I am indebted to P. Lescanne for pointing this out to me.

PROOF: According to theorem 2.1, the equations valid in I_S are provable by purely equational means. Hence, according to the soundness property of equational logic they are valid in all models of S. □

As explained in §1.3, open equations valid in the initial algebra of a specification generally require for their proofs rules of inference that are stronger than the simple rules of equational logic. Theorem 2.1 says that ω-complete specifications do not need these stronger rules of inference, i.e. they trade rules of inference for equational axioms. As far as their proofs are concerned, the open equations valid in the initial algebra of an ω-complete specification can be treated in the same way as their closed counterparts.

THEOREM 2.3: If an algebraic specification S is ω-complete, the set of equations valid in its initial algebra I_S is recursively enumerable.

PROOF: The set of equations valid in I_S is equal to the set of consequences of E_S according to theorem 2.1. The latter set is recursively enumerable. □

THEOREM 2.4: If an algebraic specification S is ω-complete and if validity of closed equations in the initial algebra I_S is decidable, validity of open equations in I_S is decidable as well.

PROOF: On the one hand, the set of equations valid in I_S is recursively enumerable according to theorem 2.3. On the other hand, each invalid open equation in I_S is finitely refutable because the set of all of its closed instances is recursively enumerable and the validity of closed equations in I_S is decidable according to the second assumption of the theorem. □

Neither theorem 2.3 nor theorem 2.4 uses any specific properties of equational logic. In fact, their truth depends solely on the existence of a complete - but not necessarily purely equational - theory of the equations valid in the initial algebra.

Given a specification S, is there always a specification T such that
(i) $\Sigma_T = \Sigma_S$, $E_T \supseteq E_S$;
(ii) $I_T = I_S$;
(iii) T is ω-complete?
Even if I_S is finite, the answer is *no*. Lyndon has given an example of a single-sorted algebra with seven elements and one binary function, whose equational theory is not finitely based (not finitely axiomatizable) [14]. With this result he settled the question "Does every finite algebra possess a finite set of identities from which all others are derivable?" raised by him in [13]. Because it has a (straightforward) initial algebra specification, this also means that Lyndon's algebra has no ω-complete initial algebra specification. Other examples are mentioned in §9 of Taylor [19].

From an abstract data type viewpoint (but not necessarily from a strictly logical viewpoint) it is quite natural to allow extension of the signature with hidden sorts and functions. In that case ω-completeness can be achieved for a wider class of specifications. For instance, Lyndon's abovementioned algebra has an ω-complete initial algebra specification with addition and multiplication mod 7 as hidden functions (see §3.2 for details).

Unlike the set of closed equations, the set of open equations valid in the initial algebra of a (finite) specification need not be recursively enumerable. For instance, the set of equations valid in the natural numbers with addition, multiplication and a $<$-predicate is not recursively enumerable (see §3.1). Such an algebra cannot have an ω-complete specification according to theorem 2.3. Extension of the signature does not help in such cases.

An obvious question is whether extension of the signature always helps if the equational theory of the initial algebra is recursively enumerable:

OPEN QUESTION 2.1: Suppose the set of equations valid in the initial algebra I_S of an algebraic specification S is recursively enumerable. Does this imply the existence of a specification T such that
(i) $\Sigma_T \supseteq \Sigma_S$, $E_T \supseteq E_S$;
(ii*a*) T is conservative with respect to the closed theory of S, i.e. for all closed Σ_S-equations e

$$E_T \vdash e \Rightarrow E_S \vdash e;$$

(ii*b*) For every closed Σ_T-term t of a sort belonging to Σ_S there is a closed Σ_S-term t' such that

$$E_T \vdash t = t';$$

(iii) All equations valid in I_S are provable from E_T?

Note that T itself is not required to be ω-complete. This would be an even stronger requirement.

Consider a finitely generated algebra whose equational theory is recursively enumerable. The subset of closed equations valid in such an algebra is *a fortiori* recursively enumerable, and hence, according to theorem 4.1 of Bergstra & Tucker [3], it has a (finite) initial algebra specification with hidden sorts and functions. Hence, if the answer to question 2.1 is affirmative, every finitely generated algebra with a recursively enumerable equational theory has an ω-complete initial algebra specification with hidden sorts and functions.

If the answer to question 2.1 is affirmative, a further question is whether the hidden sorts can be dispensed with, that is, whether every specification has an ω-complete enrichment with hidden functions only. If the answer to this question is also affirmative, one would like to conclude that every finitely generated algebra with a recursively enumerable equational theory has an ω-complete initial algebra specification with hidden functions only. But this depends on yet another open problem: It is unknown whether every finitely generated algebra whose closed equational theory is recursively enumerable has an initial algebra specification with hidden functions only (see Bergstra & Tucker [4]).

3. Examples

This section contains two examples of non-parameterized ω-complete specifications (§§3.1-2), a discussion of the conditional function from the viewpoint of ω-completeness (§3.3), and a brief discussion of the ω-incompleteness of strong combinatory logic and related questions (§3.4).

3.1. The natural numbers with addition and multiplication

A simple initial algebra specification of the natural numbers with addition and multiplication looks as follows:

module NAT
begin
  sort N
  functions $0: \to N$
    $S: N \to N$
    $+, . : N \times N \to N$
  variables $x,y: \to N$
  equations $x+0=x$ (1)
    $x+S(y)=S(x+y)$ (2)
    $x.0=0$ (3)
    $x.S(y)=x+(x.y)$ (4)
end NAT.

By adding the commutative, associative and distributive laws for addition and multiplication an ω-complete version of NAT is obtained:

module $\mathbb{N}$
begin
  include *NAT*
  variables $x,y,z\colon \rightarrow N$
  equations $x+y=y+x$ (5)
  $x+(y+z)=(x+y)+z$ (6)
  $x.y=y.x$ (7)
  $x.(y.z)=(x.y).z$ (8)
  $x.(y+z)=(x.y)+(x.z)$ (9)
end $\mathbb{N}$.

THEOREM 3.1.1 (Henkin [7]): $\mathbb{N}$ has the same initial algebra as *NAT* and is ω-complete.

SKETCH OF PROOF: (*a*) $I_{\mathbb{N}}=I_{NAT}$, because (1) $\Sigma_{\mathbb{N}}=\Sigma_{NAT}$, and (2) the commutative, associative and distributive laws for addition and multiplication are valid in I_{NAT} (proof by multiple structural induction).
(*b*) For every open or closed $\Sigma_{\mathbb{N}}$-term t there is a $\Sigma_{\mathbb{N}}$-term P in canonical polynomial form such that $E_{\mathbb{N}} \vdash t=P$. Canonical forms are generated by the grammar

P ::= 0 | **sum**
sum ::= **M** | (**sum** + **sum**)
M ::= $S(0)$ | **C** | **vars** | (**C.vars**)
vars ::= **var** | (**vars.vars**)
var ::= x | y | $\cdots$
C ::= $S(S(0))$ | S(**C**),

with the additional condition that the number of monomials (maximal subterms produced by **M**) is minimal. Canonical forms are unique modulo associativity and commutativity of addition and multiplication. Two terms t_1 and t_2 are equal in $I_{\mathbb{N}}$ if and only if the corresponding canonical forms P_1 and P_2 are syntactically identical modulo the associative and commutative laws. Otherwise there would be a non-trivial polynomial with *integer* coefficients which would be identically equal to zero. □

Paul [17] gives a proof of theorem 3.1.1 based on a complete term rewriting system for $\mathbb{N}$.
If *cut-off subtraction* $\dot{-}\colon N\times N \rightarrow N$ defined by the equations

$$x\dot{-}0=x$$
$$0\dot{-}x=0$$
$$S(x)\dot{-}S(y)=x\dot{-}y$$

is added to *NAT*, the equations valid in the initial algebra of the resulting specification *NAT′* are not recursively enumerable (§8 of Davis *et al.* [5]). Hence, according to proposition 2.3 no ω-complete specification of the natural numbers with addition, multiplication and cut-off subtraction is possible. The same result holds if a $<$-predicate is added to *NAT*. (See also Paul [17]. The same argument was used by Nourani [16] to show that equational reasoning + structural induction is not necessarily complete with respect to the equational theory of the initial algebra.)
This shows that even in (seemingly) very simple cases complete partial evaluation is impossible.

3.2. Boolean algebra
BOOL of §1.3 is an ω-incomplete specification of Boolean algebra. An (almost) ω-complete version of *BOOL* is obtained by adding the equation $S(S(x))=x$ to $\mathbb{N}$. This treatment of Boolean algebra is very economical and leads to an interesting canonical form for Boolean terms which is a direct descendant of the polynomial form for $\Sigma_{\mathbb{N}}$-terms defined in the previous paragraph. Consider

```
module 𝔹
begin
    include ℕ with renaming [N ↦ bool, 0 ↦ F, S ↦ ¬]
    functions   T: → bool
                ∨: bool×bool → bool
    variables   x,y: → bool
    equations   ¬¬x = x                                  (10)
                x.x = x                                  (11)
                T = ¬F                                   (12)
                x∨y = (x.y)+(x+y)                        (13)
end 𝔹.
```

The successor function of $\mathbb{N}$ becomes negation in $\mathbb{B}$, addition becomes exclusive-or, multiplication becomes conjunction, etc. Equation (10) corresponds to $S(S(x))=x$. Equation (11) has been added for the sake of ω-completeness.

THEOREM 3.2.1: $\mathbb{B}$ is an ω-complete specification of Boolean algebra.

PROOF: (*a*) $I_{\mathbb{B}}=I_{BOOL}$, because (1) $\Sigma_{\mathbb{B}}=\Sigma_{BOOL}$, (2) if $e\in E_{BOOL}$, then $E_{\mathbb{B}}\vdash e$ and hence $I_{\mathbb{B}}\models e$, and (3) if $e\in E_{\mathbb{B}}$, then all closed $\Sigma_{\mathbb{B}}$-instances of e are provable from E_{BOOL} and hence $I_{BOOL}\models e$.
(*b*) (See also part (*b*) of the proof of theorem 3.1.1.) For every open or closed $\Sigma_{\mathbb{B}}$-term t there is a $\Sigma_{\mathbb{B}}$-term P in canonical form such that $E_{\mathbb{B}}\vdash t=P$. Canonical forms are generated by the grammar

```
P ::= F | sum
sum ::= M | (sum + sum)
M ::= T | vars
vars ::= var | (vars.vars)
var ::= x | y | ··· ,
```

with the additional condition that the number of monomials is minimal and that all monomials are linear. Canonical forms are unique modulo the associative and commutative laws. Bringing a $\Sigma_{\mathbb{B}}$-term into canonical form involves the following steps (the equations of $\mathbb{N}$ **with renaming** [$N\mapsto bool$, $0\mapsto F$, $S\mapsto\neg$] are numbered (1)-(9) in the same order in which they occur in $\mathbb{N}$):

(S1) Eliminate all occurrences of $\vee$ and T by means of (13) and (12).

(S2) Bring the resulting term into $\mathbb{N}$-canonical form (§3.1) (taking the renaming into account) by means of (1)-(9).

(S3a) Reduce all coefficients to F or $\neg F$ by means of (10). Eliminate all coefficients of the form $\neg F$ by means of the equation $\neg F.x=x$ (which is provable from $E_{\mathbb{B}}$). Replace monomials consisting only of $\neg F$ by T by means of (12). Eliminate all monomials with coefficient F (except perhaps one) by means of (7), (3), (5) and (1).

(S3b) Linearize all monomials by means of (7), (8) and (11).

(S3c) Eliminate all monomials occurring more than once by means of (5)-(8), the equation $x+x=F$ (which is provable from $E_{\mathbb{B}}$), and (1).

Two terms t_1 and t_2 are equal in $I_{\mathbb{B}}$ if and only if the corresponding canonical forms P_1 and P_2 are syntactically identical modulo the associative and commutative laws. Otherwise there would be a non-trivial P in canonical form such that $I_{\mathbb{B}}\models P=F$. But if P is of the form $\neg Q$, it assumes the value

T because either Q is F or it assumes the value F if all variables have the value F. If P is not of the form $\neg Q$, consider a monomial q of P containing the least number of variables. Because monomials do not occur more than once, every other monomial contains at least one variable not occurring in q. If the variables occurring in q are given the value T and all other variables the value F, P assumes the value T. □

The canonical forms used in the above proof are Hsiang's "normal expressions" [8]. Besides being the most natural ones from the present viewpoint, these canonical forms have the further merit of being the normal forms of a complete term rewriting system which can be derived from $\mathbb{B}$ by a generalized Knuth-Bendix completion procedure. Other known canonical forms, such as the complete disjunctive normal form, do not have this property. Further details can be found in [8].

Paul [17] gives an ω-complete specification of the integers mod p with addition and multiplication (p prime) and proves theorem 3.2.1 by taking $p=2$. (If p is not prime ω-completeness is more difficult to achieve because the equation $x^p=x$ which corresponds to equation (11) of $\mathbb{B}$ no longer holds and the existence of zero-divisors gives rise to equations like $2x^2+2x=0$ (mod 4) and $x^3+5x=0$ (mod 6).) Paul's result can be applied as follows. Consider Lyndon's example of a seven element algebra having no ω-complete initial algebra specification without hidden sorts and functions (§2). It has a straightforward initial algebra specification:

```
module L
begin
    sort A
    functions  0, 1, 2, 3, 4, 5, 6: → A
               λ: A×A → A
    variable   x: → A
    equations  λ(4,1)=4
               λ(4,2)=λ(5,1)=λ(5,2)=λ(5,3)=5
               λ(4,3)=λ(6,1)=λ(6,2)=λ(6,3)=6
               λ(0,x)=λ(1,x)=λ(2,x)=λ(3,x)=0
               λ(x,0)=λ(x,4)=λ(x,5)=λ(x,6)=0
end L.
```

Every k-ary function on a set of p elements (p prime) corresponds to a polynomial in k variables over the integers mod p. Take $p=7$ and let $\mathbb{Z}_7$ be an ω-complete specification of the integers mod 7 with sort A, constants 0, . . . ,6, and functions + and . , then L has the following ω-complete hidden function enrichment:

```
module 𝕃
begin
    include ℤ7
    hidden functions  + , .
    function   λ: A×A → A
    variables  x,y: → A
    equation   λ(x,y)=4.P_{4,1}(x,y)+5.(P_{4,2}(x,y)+P_{5,1}(x,y)+P_{5,2}(x,y)+P_{5,3}(x,y))+
                      +6.(P_{4,3}(x,y)+P_{6,1}(x,y)+P_{6,2}(x,y)+P_{6,3}(x,y))
```

$$\text{where } P_{m,n}(x,y)=\prod_{\substack{i=0\\ i+m\neq 0}}^{6}(x+i).\prod_{\substack{j=0\\ j+n\neq 0}}^{6}(y+j)$$

```
end 𝕃.
```

$P_{m,n}(x,y)$ has the property

$P_{m,n}(m,n)=1$
$P_{m,n}(x,y)=0 \quad x \neq m,\ y \neq n.$

The above method of obtaining an ω-complete hidden function enrichment applies to all single-sorted algebras with p elements (p prime).

3.3. The conditional function
The following module contains a simple definition of a polymorphic conditional function *if*:

```
module IF
begin
    include 𝔹
    variable    σ: → sorts
    function    if: bool×σ×σ → σ
    variables   u,v: → σ
    equations   if (F,u,v)=v                                   (1)
                if (T,u,v)=u                                   (2)
end IF.
```

Sort variable σ ranges over all sorts occurring in the specification, i.e. if *IF* is combined with a specification *S*, *if*: $bool \times \sigma \times \sigma \rightarrow \sigma$ expands into a non-polymorphic if_s: $bool \times s \times s \rightarrow s$ for every sort $s \in \Sigma_{S+IF}$.
Let DIF be the union of *IF* and

```
module D
begin
    sort data
    functions   d₁, d₂, . . . , dₘ: → data    (m>1)
end D.
```

In *DIF* the *if*-function has two non-polymorphic instances, namely if_{bool}:$bool \times bool \times bool \rightarrow bool$ and if_{data}:$bool \times data \times data \rightarrow data$.
D is trivially ω-complete for $m>1$, but in the degenerate case $m=1$ the equation $u=d_1$ (with u a variable of sort *data*) is valid in I_D. From now on $m>1$ is assumed.
DIF is not ω-complete. The equation

$$if\,(X,u,u)=u \qquad (a)$$

is an example of an equation which is valid in I_{DIF}, but not provable from E_{DIF}. In conventional programming languages, for instance, equations (1) and (2) hold but (*a*) does not, because the evaluation of X may loop or have side-effects.
The following version of *IF* is better from the viewpoint of ω-completeness:

```
module IFa
begin
    include IF
    variables  σ: → sorts
               u,v,w: → σ
               X,Y,Z: → bool
    equations  if(X,u,v)=if(X,u,if(¬X,v,w))                    (3)
               if(X,u,if(Y,v,w))=if(¬X.Y,v,if(X,u,w))           (4)
               if(X,u,if(Y,u,v))=if(X∨Y,u,v)                     (5)
               if(X,if(Y,u,v),w)=if(X.Y,u,if(X.¬Y,v,w))         (6)
               if(X,Y,Z)=(X.Y)+(¬X.Z)                            (7)
end IFa.
```

THEOREM 3.3.1: $DIFa=D+IFa$ has the same initial algebra as DIF and is ω-complete.

PROOF: (*a*) $I_{DIFa}=I_{DIF}$, because $\Sigma_{DIFa}=\Sigma_{DIF}$ and all equations in E_{DIFa} are valid in I_{DIF}. (*b*) If t is a Σ_{DIFa}-term of sort *bool* it can be brought into $\mathbb{B}$-canonical form (§3.2) because all *if*s can be eliminated from t by means of (7). If t is a Σ_{DIFa}-term of sort *data* containing distinct Boolean variables $X_1, \ldots, X_k$ $(k \geq 0)$ and distinct variables of sort *data* $u_1, \ldots, u_l$ $(l \geq 0)$, it can be brought into the canonical form

$$\delta_1$$

or

$$if(\xi_n,\delta_n,if(\xi_{n-1},\delta_{n-1}, \ldots, if(\xi_1,\delta_1,v)\ldots)) \quad (n \geq 2)\,.$$

The δ_i's are constants or variables of sort *data* (i.e. elements of $\{d_1, \ldots, d_m, u_1, \ldots, u_l\}$), v is an arbitrarily chosen variable of sort *data*, and the ξ_i's are Boolean terms in $\mathbb{B}$-canonical form, such that

(i) $\delta_i \neq \delta_j \quad (i \neq j)$
(ii) ξ_i is not of the form F or T
(iii) $\xi_i.\xi_j =_{\mathbb{B}} F \quad (i \neq j)$
(iv) $\bigvee_{i=1}^{n} \xi_i =_{\mathbb{B}} T.$

Two canonical forms are equal in I_{DIFa} if and only if they are syntactically identical modulo commutativity and associativity of . and +, modulo the shuffling of (ξ_i,δ_i)-pairs, and modulo the choice of v. It takes the following steps to bring a Σ_{DIFa}-term of sort *data* into canonical form:

(S1) Eliminate all Boolean *if*s by means of (7).

(S2) Eliminate all *if*s from the second argument of other *if*s by means of (6).

(S3) Expand the innermost $if(\xi,\delta,\delta')$ (if it exists) into $if(\xi,\delta,if(\neg\xi,\delta',v))$ by means of (3). The resulting term satisfies (iv).

(S4) Merge all *if*s whose second argument contains the same constant or variable by means of (4) and (5). The resulting term satisfies (i).

(S5) If at this point the canonical form *in statu nascendi* is of the form

$$if(\eta_n,\delta_n,if(\eta_{n-1},\delta_{n-1}, \ldots, if(\eta_1,\delta_1,v)\ldots)) \qquad (n>1),$$

then turn it inside out, i.e. turn it by means of $\frac{n(n-1)}{2}$ applications of (4) into

$$if(\theta_1,\delta_1, \ldots, if(\theta_{n-1},\delta_{n-1},if(\theta_n,\delta_n,v))\ldots)$$

with $\theta_n=\eta_n$, $\theta_{n-1}=\neg\eta_n.\eta_{n-1}$, $\theta_{n-2}=\neg(\neg\eta_n.\eta_{n-1}).(\neg\eta_n.\eta_{n-2})$, etc.

The resulting term satisfies (iii).

(S6) Bring all θ_i's into $\mathbb{B}$-canonical form ξ_i.

(S7a) If ξ_i is of the form F for some i, eliminate the corresponding *if* and δ_i by means of (1).

(S7b) If ξ_i is of the form T for some i, the term is of the form $if(T,\delta,v)$ because of property (iii) and (S7a). Reduce it to δ by means of (2). The resulting term satisfies (ii) and is in canonical form. □

Although, according to theorem 3.4.1, *IFa* is ω-complete when combined with the simplest possible D, ω-completeness is lost if D is somewhat more complicated. For instance, the equations

$$S(if(X,x,y)) = if(X,S(x),S(y))$$
$$if(X,x,y).if(X,y,x) = x.y$$

are valid in $I_{\mathbb{N}+IFa}$ but not provable from $E_{\mathbb{N}+IFa}$. This can be remedied by adding the distributive property of *if* to *IFa*:

```
module IFb
begin
    include IFa
    variables   X: → bool
                σ,τ: → sorts
                u,v: → σ
                Φ: σ → τ
    equation    Φ(if(X,u,v)) = if(X, Φ(u),Φ(v))                (8)
end IFb.
```

Equation (8) is to be interpreted as follows. If *IFb* is combined with a specification S, (8) expands into n separate instances for every n-ary function $f \in \Sigma_{S+IFb}$ by substitution of $(\lambda x_k)f(x_1, \ldots, x_k, \ldots, x_n)$ for Φ $(1 \leq k \leq n)$. For example, one of the instances of (8) is $(f = if,\ k=2)$

$$if(Y, if(X,u,v),w) = if(X, if(Y,u,w), if(Y,v,w)),$$

which is provable from E_{IFa}.

THEOREM 3.3.2: $S+IFb$ is ω-complete for every ω-complete specification S that does not contain functions of one or more Boolean arguments or with a Boolean result.

PROOF: Use for every sort $s \in \Sigma_S$ a canonical form similar to the one used in the proof of theorem 3.3.1, but with δ_i a term of sort s in S-canonical form. To bring a term into canonical form, follow steps (S1)-(S7b) of theorem 3.3.1 with two additional steps between (S1) and (S2), and a slightly different step (S4):

(S1.1) Move all *if*s to outermost positions by means of (8).

(S1.2) Bring all maximal *if*-free subterms (all of which are necessarily of the same sort) into S-canonical form.

(S4′) Merge all *if*s whose second argument contains syntactically identical S-canonical forms by means of (4) and (5). The resulting term satisfies (i). □

If S contains functions of Boolean arguments or with a Boolean result (as indeed it will in all realistic cases), the selective action of the first argument of the *if*-function gives rise to new equations and theorem 3.3.2 fails. For instance, suppose an ω-complete specification S containing $\mathbb{B}$ is sufficiently-complete with respect to $\mathbb{B}$, i.e. all closed Σ_S-terms of sort *bool* can be proved equal to T or F. Suppose further that Σ_S contains a sort *data* and functions f,g: *bool* → *data* and h,k: *bool*×*bool* → *data*. In that case some typical equations valid in I_{S+IFb} but not provable from E_{S+IFb} are

$$if(X,f(X),g(X))=if(X,f(T),g(F)) \quad (9)$$
$$if(X+Y,h(X,Y),k(X,Y))=if(X+Y,h(X,\neg X),k(X,X)) \quad (10)$$
$$if(X.Y,h(X,Y),k(X,Y))=if(X.Y,h(T,T),if(X+Y,k(X,\neg X),k(F,F))). \quad (11)$$

Contrary to equations (1)-(8), which are valid in I_{S+IF} for all S satisfying the sufficient-completeness requirement just mentioned, equations like (9)-(11) are very much dependent on the particular S involved.

If interpreted as a left-to-right rewrite rule, equation (11) is typical of a whole class of rules whose right-hand sides contain more *if*s then their left-hand sides. Application of such rules easily leads to terms containing an enormous number of alternatives, because in general most of the new branches only lead to further branches.

3.4. Combinatory logic

Consider the following algebraic specification of strong combinatory logic:

```
module CLX
begin
    sort F
    functions  K, S: → F
               . : F×F → F   (application)
               Note. The infix dot is not written and application
               associates to the left, i.e. (K.x).y is written as Kxy, etc.
    variables  x,y,z: → F
    equations  Kxy = x
               Sxyz = xz(yz)
               S(S(KS)(S(KK)(S(KS)K)))(KK) = S(KK)
               S(KS)(S(KK)) = S(KK)(S(S(KS)(S(KK)(SKK)))(K(SKK)))
               S(K(S(KS)))(S(KS)(S(KS))) =
                  = S(S(KS)(S(KK)(S(KS)(S(K(S(KS)))S))))(KS)
               S(S(KS)K)(K(SKK)) = SKK
 end CLX.
```

CLX is identical to $CL+A_{\beta\eta}$ in Barendregt [1]. Hence, according to [1], theorem 7.3.14, *CLX* is equivalent to the $\lambda K\beta\eta$-calculus. The last four closed equations (the so-called *combinatory axioms*) give *CLX* the *extensional property*, i.e. if for two (possibly open) Σ_{CLX}-terms f and g not containing the variable x

$$E_{CLX} \vdash fx=gx,$$

then also

$$E_{CLX} \vdash f=g.$$

Is *CLX* ω*-extensional*? That is, does

$$E_{CLX} \vdash fa=ga \quad \text{for all closed } a$$

imply

$$E_{CLX} \vdash f=g?$$

Plotkin has shown that the $\lambda K\beta\eta$-calculus is not ω-extensional (Plotkin [18] and Barendregt [1], theorem 17.3.30). Hence, *CLX* is not ω-extensional either. Because

$$\omega\text{-completeness} + \text{extensionality} \Rightarrow \omega\text{-extensionality}, \qquad (1)$$

CLX is not ω-complete. In fact, as far as *CLX* is concerned the notions of ω-extensionality and ω-completeness are equivalent. This is not difficult to prove. In view of (1) plus the fact that *CLX* is combinatorially complete, it is enough to show that

$$\text{combinatorial completeness} + \omega\text{-extensionality} \Rightarrow \omega\text{-completeness}. \qquad (2)$$

Consider a Σ_{CLX}-equation $f=g$ all of whose closed instances are provable from E_{CLX}. Assume further that f and g contain the same variables $x_1, \ldots, x_k$ $(k \geqslant 1)$. (If f contains a variable x not in g, then replace some variable or constant v in g by Kvx, etc.) By combinatorial completeness of *CLX* there exist closed terms ϕ and ψ such that

$$E_{CLX} \vdash f=\phi x_1 \cdots x_k,\ \ g=\psi x_1 \cdots x_k$$

Applying ω-extensionality k times gives

$$E_{CLX} \vdash \phi=\psi.$$

Hence

$$E_{CLX} \vdash \phi x_1 \cdots x_k=\psi x_1 \cdots x_k$$

and

$$E_{CLX} \vdash f=g.$$

This proves (2).

Two questions I have not succeeded in answering are:

OPEN QUESTION 3.4.1: Are the open equations valid in the initial algebra of *CLX* recursively enumerable?

OPEN QUESTION 3.4.2: Does *CLX* have an ω-complete enrichment in the sense of open question 2.1?

If - as would be my guess - the answer to the first question is *no*, the answer to the second question must also be *no* according to theorem 2.3. If the answer to the first question is *yes*, the second question is a special case of open question 2.1.

4. TOWARDS AUTOMATIC (PARTIAL) ω-ENRICHMENT

Describing semantics by means of term rewriting systems has the advantage of yielding evaluators that work on both closed and open terms. Their performance on open terms (partial evaluation) is often disappointing, however, as many more or less trivial simplifications of open terms are beyond the power of the rewrite rules required for evaluating closed terms (§1.1). In such cases even rudimentary ω-enrichment may be rewarding, and the question arises whether this can be done automatically. (Even if the answer to open question 2.1 is affirmative, partial ω-enrichment is the best one can hope for in many cases. See §3.1.)

While "inductionless induction" or "inductive completion" algorithms (§1.4) can sometimes help in proving the validity of a given potential ω-enrichment, they do not help in suggesting significant new ω-enrichments (or, for that matter, in giving ω-completeness proofs).

An approach I am currently investigating is automatic partial ω-enrichment by means of sets of *enrichment rules*. This works roughly as follows. An enrichment rule

$$P(\sigma_1, \ldots, \sigma_m, \Phi_1, \ldots, \Phi_n) \rightarrow E(\sigma_1, \ldots, \sigma_m, \Phi_1, \ldots, \Phi_n)$$

is a *specification rewrite rule* consisting of a *specification pattern P* and an *enrichment scheme E*. The signatures of *P* and *E* contain *sort variables* σ_i and *function variables* Φ_j. If *P* matches the specification to be enriched *S*, i.e. if there is an instance of *P* which is a subspecification of *S*, the part of *S* matched by *P* is replaced by the corresponding instance of the enrichment scheme *E*, possibly after renaming the hidden sorts and functions introduced by E to avoid name clashes with the hidden items of S. Special care has to be taken to ensure that enrichment steps are correct.

This approach has the advantage of being rather natural. Its success depends on whether a large enough number of generally applicable enrichment rules can be found and on whether the validity of enrichment steps can be guaranteed.

ACKNOWLEDGEMENTS

While writing this paper I had helpful discussions with Jan Bergstra, Paul Klint, Jan Willem Klop, and Ed Kuijpers.

REFERENCES

[1] H.P. Barendregt, *The Lambda Calculus* (North-Holland, 1981).

[2] R.M. Burstall & J.A. Goguen, Algebras, theories and freeness: an introduction for computer scientists, in: M. Broy & G. Schmidt, eds., *Theoretical Foundations of Programming Methodology* (D. Reidel, 1982) 329-348.

[3] J.A. Bergstra & J.V. Tucker, Algebraic specifications of computable and semi-computable data structures, Report IW 115/79, Department of Computer Science, Centre for Mathematics and Computer Science, Amsterdam, 1979; to appear in *Theoretical Computer Science.*

[4] J.A. Bergstra & J.V. Tucker, Initial and final algebra semantics for data type specifications: two characterization theorems, *SIAM Journal on Computing* **12** (1983) 2 366-387.

[5] M. Davis, Y. Matijasevic & J. Robinson, Hilbert's tenth problem: positive aspects of a negative solution, in: F.E. Browder, ed., *Mathematical Developments Arising from Hilbert Problems* (American Mathematical Society, 1976) 323-378.

[6] A.P. Ershov, Mixed computation: potential applications and problems for study, *Theoretical Computer Science* **18** (1982) 41-67.

[7] L. Henkin, The logic of equality, *The American Mathematical Monthly*, **84** (1977) 597-612.

[8] J. Hsiang, Topics in automated theorem proving and program generation, Report UIUCDCS-R-82-1113, Department of Computer Science, University of Illinois at Urbana-Champaign, 1982.

[9] G. Huet, G. & J.M. Hullot, Proofs by induction in equational theories with constructors, *Journal of Computer and System Sciences*, **25** (1982) 239-266.

[10] G. Huet & D.C. Oppen, Equations and rewrite rules: a survey, in: R. Book, ed., *Formal Languages: Perspectives and Open Problems* (Academic Press, 1980).

[11] N.D. Jones, P. Sestoft & H. Søndergaard, An experiment in partial evaluation: the generation of a compiler generator, Report 85/1, Institute of Datalogy, University of Copenhagen, 1985.

[12] H.J. Komorowski, *A Specification of an Abstract* PROLOG *Machine and its Application to Partial Evaluation*, Dissertation No. 69, Linköping University, 1981.

[13] R.C. Lyndon, Identities in two-valued calculi, *Transactions of the American Mathematical Society*, **71** (1951) 457-465.

[14] R.C. Lyndon, Identities in finite algebras, *Proceedings of the American Mathematical Society*, **5** (1954) 8-9.

[15] J. Meseguer & J.A. Goguen, Initiality, induction, and computability, Preprint, Computer Science Laboratory, SRI International, n.d.; in: M. Nivat & J. Reynolds, eds., *Algebraic Methods in Semantics* (Cambridge University Press, 1986).

[16] F. Nourani, On induction for programming logic: syntax, semantics, and inductive closure, *Bulletin of the European Association for Theoretical Computer Science*, 13, February 1981, 51-64.

[17] E. Paul, Proof by induction in equational theories with relations between constructors, in: B. Courcelle, ed., *Ninth Colloquium on Trees in Algebra and Programming* (Cambridge University Press, 1984).

[18] G.D. Plotkin, The λ-calculus is ω-incomplete, *Journal of Symbolic Logic*, **39** (1974) 313-317.

[19] W. Taylor, Equational logic, *Houston Journal of Mathematics*, Survey 1979.