Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P.R.H. Hendriks

Type-checking mini-ML:
an algebraic specification with user-defined syntax

Computer Science/Department of Software Technology Report CS-R8737 August

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Type-Checking Mini-ML: an Algebraic Specification with User-Defined Syntax

P.R.H. Hendriks
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

An algebraic specification of a type-checker for Mini-ML, a sublanguage of ML, is given. As specification formalism a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF is used.

Key Words & Phrases: Software Engineering, Algebraic Specifications, Formal Definition of Programming Languages, Specification Languages, Executable Specifications, User-definable Syntax, Syntax Definition Formalism, Type-checker, Polymorphism, Type Inference.

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

1986 CR Categories: D.2.1 [**Software Engineering**]: Requirements, Specifications; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs - *Specification techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages - *Algebraic Approaches to Semantics*. F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and other Rewriting Systems - *Parsing*.

Note: Partial support received from the European Communities under ESPRIT project 348 (Generation of Interactive Programming Environments - GIPE).

Note: This paper will be submitted for publication elsewhere.

1. Introduction

The goal of ESPRIT-project 348 (GIPE - Generation of Interactive Programming Environments) is to make a system which can generate an interactive programming environment for a programming language from the specification of the language. In this context the formalisms ASF and SDF were developed as intermediate steps in the development of a formalism to specify languages. ASF [BHK87, BHK85] is an algebraic specification formalism and SDF [HK86, HHKR87] is a syntax definition formalism by means of which the concrete and abstract syntax of a language can be specified. The type-checker for Mini-ML is the first example of a specification written in a combination of both formalisms.

Several problems have been specified in the algebraic specification formalism ASF:

- The lexical analyzer, parser, type-checker and dynamic semantics of the toy language PICO [BHK85].
- The dynamic semantics of a language with goto-statements SMALL [Die86].
- A type-checker for the parallel, object-oriented language POOL [Wal86].

ASF has a simple, fixed syntax which permits the use of functions with fixed arity and of a limited form of unary and binary infix operators. It was obvious that a more liberal use of syntax would be convenient and would improve readability of the specifications. Facilities for specifying syntax are clearly necessary for a language definition formalism; without such facilities, major parts of each language definition will have to

deal with syntactic matters (as can be seen in [BHK85]). The syntax definition formalism SDF [HK86, HHKR87] is a formalism allowing the user to define arbitrary context-free syntax. The essence of an SDF-specification is to define simultaneously the concrete and abstract syntax of a language. SDF has been developed independently of any particular specification formalism. In principle, it can be combined with any specification formalism based on first-order signatures. Here we experiment with a combination of ASF and SDF.

The specification of the type-checker for the functional language ML is a challenge because ML allows polymorphism and requires type inference. For an extensive overview of typing schemes we refer to [CW85]. Type-checking ML (or parts of the language) has been the subject of several papers. [DM82] describes an inference system which yields type schemes for expressions and also gives an algorithm for computing the most general type of an expression. Our specification is based on this algorithm. It specifies not only which expressions are typeable but also gives *false* if an expression is not typeable. Each method which is solely based on the set of inference rules mentioned above can only show which expressions are typeable and a proof at the meta-level is needed to show that an expression cannot be typed. [Car84] describes a system of type equations, a type inference system to type-check ML expressions, and an implementation of a type-checker in ML. The Mini-ML specification in TYPOL [CDDK86, Kah87] resembles the type inference system presented in [DM82] and [Car84]. TYPOL [Kah87] is a specification formalism developed to describe the static and dynamic semantics of programming languages.

Section 2 contains a description of the formalism used in our specification. The type-checker is informally described in section 3 and the specification itself is presented in section 4. In section 5 we describe how an implementation for a type-checker can be derived from the algebraic specification. Finally, sections 6 and 7 contain conclusions and some remarks on questions related to our specification.

2. The Specification Formalism

In this paper we will use as specification formalism a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF. This section summarizes the main concepts of both formalisms. For more detailed information we refer to [BHK87] in which ASF is described and [HK86] for the informal description of SDF and [HHKR87] for its formal description.

2.1. ASF: Algebraic Specification Formalism

ASF [BHK87, BHK85] is an algebraic specification formalism similar to OBJ-2 [FGJM85], ACT-ONE [EM85], and RAP [Hus86]. The general idea is to give a signature and a set of (conditional) equations over that signature. The signature consists of a set of sorts and a set of functions over these sorts.

The meaning of an ASF-specification is its initial algebra [MG85]. An example of the initial algebra is the term-model. It is defined by constructing all closed terms over the signature and divide these into sets according to their sorts. Next, an congruence relation is defined on closed terms by defining two closed terms to be equal if and only if we can deduce their equality from the set of (conditional) equations using many-sorted equational logic. Finally, each set of closed terms is divided into classes in accordance with the congruence relation.

ASF has several features to subdivide a specification into modules:

- Exports:
  Each module may have an **exports**-section consisting of a (possibly incomplete) signature. These sorts and functions are visible outside the module.
- Hiding:
  Sorts and functions that are local to a module are declared in the **sorts**- and **functions**-section.
- Imports:
  The **imports**-section contains the names of modules that have to be incorporated in a module. While importing a module it is possible to bind its parameters, to rename its signature (see below) or to perform a combination of these.
- Parameters:
  In the **parameters**-section of a module we can declare its parameters. These are declarations of (possibly incomplete) signatures which are formal parameters of the module and they can be bound

to actual parts of a module when the parameterized module is imported.

- Renamings:

  Upon importing a module we can rename parts of the signature of the module if we want to keep the structure of it, but some changes in names of sorts and/or functions are desirable.

Modularization has been studied separately in [BHK86], which gives an algebraic specification of the main concepts of modularization, except parameterization. In [BHK87] an extensive description of ASF is given and it also describes the normalization strategy, which defines how compound modules have to be evaluated in the context of the total specification to which they belong. The result of such a normalization is a module without imports, but possibly with some remaining unbound parameters.

Throughout this paper pictures are given, which depict the modular structure of a specification. Each module is represented as a box with the name of the module in the lower part of the box. The parameters of the module are shown as ellipses on the upper side of the box. The names of the parameters are in the ellipses. The import of one module in another module is represented by a nested box. Binding of parameters is shown by drawing a line from the parameter to the actual module. Details like the signature, the equations and renaming of signatures are not shown in the pictures.

2.2. SDF: Syntax Definition Formalism

The syntax definition formalism SDF allows the user of a specification formalism to extend that formalism with new syntactic forms. An SDF-specification is a combination of the abstract syntax (in the form of a signature) and the concrete syntax (in the form of BNF-rules, read in reversed order) of a language. Hence, each SDF-specification implicitly defines a lexical analyzer and a parser for the language it defines. An experimental implementation of SDF [Rek87] has been used to parse the equations of the specification of the Mini-ML type-checker.

The following example shows how simple expressions can be defined in SDF:

```
module Example
begin
    lexical syntax
        sorts
            DIGIT, INTEGER, LETTER, ID
        layout
            SPACE
        functions
            [0-9]                   -> DIGIT
            DIGIT+                  -> INTEGER
            [a-zA-Z]                -> LETTER
            LETTER DIGIT*           -> ID
            [ \t\n\r]               -> SPACE

    context-free syntax
        sorts
            EXP
        priorities
            (EXP "-" EXP , "+") < "*" < "-" EXP
        functions
            ID                      -> EXP
            INTEGER                 -> EXP
            "-" EXP                 -> EXP {par}
            EXP "-" EXP             -> EXP {left-assoc, par}
            EXP "+" EXP             -> EXP {assoc, par}
            EXP "*" EXP             -> EXP {assoc, par}
end Example
```

An SDF-specification consists of two parts in which the lexical and context-free syntax of a language are defined. Both sections contain the definition of a signature extended with the definition of the concrete syntactic form of each of the functions. Extra features of SDF are:

- The `layout`-section in the lexical syntax part of the specification defines the name of a sort which is used to specify the layout of the language.

- A sort followed by a `*` stands for zero or more repetitions of the sort. A `+` stands for one ore more repetitions. The notations `{SORT t}*` and `{SORT t}+` are used to denote lists of elements of `SORT` separated by the symbol `t`.
- Character classes like `[0-9]` and `[a-zA-Z]` are used to abbreviate the lexical definition.
- The `priority`-section defines the priority of a function declaration relative to other function declarations. It is sufficient to give only the terminals occurring in a function declaration if they uniquely determine the declaration in question.
- Some attributes can be added to a function declaration in the context-free syntax part of an SDF specification:
  - The attribute `par` indicates that parentheses may surround instances of a function.
  - `assoc`, `left-assoc` or `right-assoc` indicate that a function is associative, left-associative or right-associative.

2.3. The combination of ASF and SDF

We use a combination of ASF and SDF as a specification formalism. Modules in this formalism are similar to ASF modules, except that the signature definition is now replaced by an SDF-definition. The concrete syntax in the signature-part of each module defines the syntactic form of the expressions which may be used in the equations-part. Conversely, specifications in the combined ASF/SDF formalism can be reduced to ASF specifications as follows:

- replace each SDF-definition by its underlying signature;
- parse all equations using the grammar defined by the SDF-definitions and replace each equation by the result of this parse (the result is an equation containing terms in prefix form instead of arbitrary strings).

The specification formalism we will use should be considered as an intermediate step in the integration of ASF and SDF. Therefore, we have not used all features of SDF and certain features are only used in a limited form:

- Module composition in ASF/SDF is not yet fully understood. What are, for instance, the constraints on parameter bindings, renamings or priority declarations? In the Mini-ML specification, we use the module composition operations only in a straightforward way to ensure that they have a well-defined meaning.
- Syntax definitions are not divided in a lexical part and a context-free part. This avoids the question how modules with different layout definitions should be combined.
- The terminal alphabet of SDF has been extended with various fonts in different point sizes; this improves the readability of the specification.

When adding user-defined syntax to ASF there is a tradeoff between improving readability and introducing syntactic ambiguities. In ASF, overloading of function symbols is allowed as long as no ambiguities are introduced: it is forbidden to define functions with the same name and the same input type. It is not possible to decide *in general* whether an SDF-specification defines an ambiguous language or not ([HU79],p. 200). In the Mini-ML specification ambiguities are avoided by introducing extra syntax when necessary. In this way, some expressions in the specification may be ambiguous when considered in isolation, but all expressions have a unique parse when their context is taken into account. If the subscripts in the definitions of $\in_{TYPE}$ and $\in_{GT}$ were omitted, for example, then equation **[98]** would be ambiguous and it would be impossible to write equation **[104]**.

3. Mini-ML

Mini-ML is a small sublanguage of the Standard ML Core Language [Mil85]. The version of Mini-ML used here is a slight modification of the language used in [CDDK86]. As far as type-checking is concerned, Mini-ML contains all essential elements of ML. Information on the static semantics of ML (and Mini-ML) is given in [CDDK86, Car84, DM82]. [DM82] describes a type-checking algorithm for a sublanguage of ML, which is even smaller than Mini-ML. It determines the most general type for every expression of the language. This algorithm is essentially the one used in the algebraic specification presented in section 4.

The syntax of Mini-ML and the Mini-ML type-checker are described in the following sections.

3.1. Syntax of Mini-ML

Mini-ML expressions have the following syntax:

<expr>	::=	'true' |	
		'false' |	boolean constants
		<natural-number> |	
		<identifier> |	
		'(' <expr> <expr> ')' |	application
		'λ' <identifier> '.' <expr> |	lambda-abstraction
		'let' <identifier> '=' <expr> 'in' <expr> |	declaration
		'letrec' <identifier> '=' <expr> 'in' <expr> |	recursive declaration
		'if' <expr> 'then' <expr> 'else' <expr> 'fi' |	conditional
		'(' <expr> ',' <expr> ')' .	Cartesian product

In this definition <identifier> and <natural-number> are predefined lexical notions for, respectively, identifiers and natural numbers.

3.2. Type-checking Mini-ML

First, we will now describe the syntax and semantics of types and of generalized types, and we will give some examples of type-checking. Next, we present an informal description of the type-check algorithm in section 3.2.2.

3.2.1. Syntax and semantics of types and generalized types

Each closed expression (i.e., expressions without free variables: all identifiers are bound by lambda-abstraction, a declaration or a recursive declaration) of Mini-ML denotes a basic notion (like booleans or natural numbers), a function, or a Cartesian product. Hence we can attach a type to each expression, as defined by the following syntax:

<type>	::=	<var> |	
		'bool' |	type of the booleans
		'nat' |	type of the natural numbers
		<type> '→' <type> |	function-type; right-associative
		<type> '×' <type> .	Cartesian product of types; left-associative

Here, <var> is a non-terminal which produces type variables. In the following, the symbols σ0, σ1, σ2, ... are used as type variables. $\sigma 0 \rightarrow \sigma 1$ is the type of an expression, which is a function from expressions of type σ0 to expressions of type σ1. $\sigma 0 \times \sigma 1$ is the Cartesian product of the types σ0 and σ1. The Cartesian product × binds stronger than →.

An expression can have several possible types. For instance, $bool \rightarrow bool$, $(\sigma 2 \times \sigma 2) \rightarrow (\sigma 2 \times \sigma 2)$ or $\sigma 1 \rightarrow \sigma 1$ are some of the possible types of the identity function $\lambda x . x$. The type-check algorithm always returns the most general type of an expression, i.e., the type from which all other possible typings can be derived using a substitution which replaces type variables by types. The most general type of an expression is unique modulo the renaming of the type variables occurring in it. The most general type of $\lambda x . x$ is $\sigma 1 \rightarrow \sigma 1$.

An identifier defined in a (recursive) declaration is assumed to be polymorphic. If the identifier occurs more than once in the expression-part (the in-part of the let- or letrec-construction) each occurrence may have another type. Consider the following examples:

$$\text{let } x = \lambda y . y \text{ in } (\cdots x \cdots x \cdots) \tag{1}$$

$$(\lambda x . (\cdots x \cdots x \cdots) \lambda y . y) \tag{2}$$

In the first expression x is declared to be the polymorphic identity function. Both occurrences of x in (1) should have a type of the form $\sigma 1 \rightarrow \sigma 1$, but if the first occurrence is forced to be of type $nat \rightarrow nat$ this should not influence the type of the second occurrence of x. In expression (2) both occurrences of the identifier x should always have the same type. As it should be possible to bind x to $\lambda y . y$, both occurrences of x should have a type of the form $\sigma 1 \rightarrow \sigma 1$. If in this case the first occurrence of x is forced

to be of type *nat* → *nat* the second occurrence is also forced to be of type *nat* → *nat*.

It is not possible to describe polymorphic declarations by attaching types to identifiers. The notion of types has to be generalized in order to distinguish generic type variables (i.e., different occurrences may have different type values) and "normal" type variables (i.e., all occurrences have the same type value). Generalized types have the following syntax:

<gen-type> ::= <type> |
<gen-var> |
<gen-type> '→' <gen-type> |
<gen-type> '×' <gen-type> .

Here, <gen-var> represents generic type variables which we will write as β0, β1, β2, ... Rules for priority and associativity of → and × are similar to the ones given earlier for types. The syntax of generalized types (or type schemes) is somewhat different from the one used in [DM82]. In [DM82] polymorphism is described by type schemes: types prefixed with universal quantifiers in order to bind some of the type variables in the type. This syntax could be handled in our specification, but for reasons of readability generic type variables are simply represented by β0, β1, β2, ...

The type-check algorithm associates a generalized type with each identifier in a Mini-ML expression. This information is kept in a type environment. If an instance of a generalized type is needed, all generic type variables are changed to "fresh" type variables (i.e., type variables which have not yet been used by the type-check algorithm). The type computed for an identifier defined by a (recursive) declaration is generalized, i.e., each type variable that does not occur in the type environment is potentially polymorphic and is changed into a generic type variable.

A syntactically correct Mini-ML expression is only typeable if it satisfies the following constraints:

- The expression is closed. Otherwise it would contain identifiers not bound by a lambda-abstraction, a declaration or a recursive declaration. An example is: λ x . y.
- The type structure of Mini-ML expressions is hierarchical, which means that an expression may not be applied to itself. Expressions like λ x . (x x) and λ x . λ y . ((x y) x) are forbidden.
- In a recursive declaration the identifier and all its occurrences in the declaration part must be typeable with the same type. An expression like letrec x = (x 4) in ··· is not typeable.
- In an if-then-else-fi expression, the first expression should have type boolean and the then- and else-part of the expression should have the same types. Examples of erroneous expressions are: λ x . if 4 then x else x fi, and λ x . if x then 2 else true fi.
- All subexpressions of an expression should be typeable using the information from the type environment in case subexpressions are not closed.

3.2.2. The type-checker

In this section the type-check algorithm is described informally. Type-checking is defined recursively on the (abstract) syntactic structure of an expression E. The type-checker computes the most general type of E, provided this type exists. During type-checking, a type environment T is constructed, which contains the generalized types of the identifiers occurring in E. In this informal description the following details will not be considered:

- changes in the type environment as a consequence of type-checking of subexpressions.
- changes in types as a result of unification.

These details can be found in the algebraic specification of the algorithm.

The type-check algorithm distinguishes the following cases:

- E is 'true' or E is 'false':
  'true' and 'false' are both of constant type *bool*.
- E is a natural number:
  All natural numbers are of constant type *nat*.
- E is an identifier:
  Each identifier in an expression must have been bound by lambda-abstraction, a declaration or a recursive declaration. We only need to examine the type environment T: if the identifier is present

in T, the type of E is an instance of the generalized type found in T. Otherwise, E cannot be typed.

- $E \equiv (E_1\ E_2)$:
  The main idea is to determine the types of both expressions E_1 and E_2, which gives τ_1 and τ_2 respectively. These types must be such that τ_1 can "eat" τ_2, i.e., we must unify τ_1 and $\tau_2 \rightarrow \sigma 1$, where $\sigma 1$ is a "fresh" variable. The type of the whole expression is $\sigma 1$.
- $E \equiv \lambda\, x\, .\, E_1$:
  First, the identifier x is added to the type environment T with as type the "fresh" type variable $\sigma 1$. Then the type τ_1 of E_1 is determined using the new type environment T_1. The type of E is $\sigma 1 \rightarrow \tau_1$. The identifier x and its corresponding generalized type are removed from the type environment T_1.
- $E \equiv$ let $x = E_1$ in E_2:
  First, E_1 is type-checked in the type environment T, and then the type of E_1 is generalized resulting in a generalized type α_1. Now the identifier x and its associated generalized type α_1 is added to the environment and the type of E_2 is determined using this type environment. The type of the entire expression is the type of E_2. Finally x and its generalized type are removed from the environment.
- $E \equiv$ letrec $x = E_1$ in E_2:
  The identifier x is added to the type environment T with a "fresh" type variable $\sigma 1$ and then the type of E_1 is determined. The type of E_1 and $\sigma 1$ have to be equal, i.e., these types have to be unified (note that $\sigma 1$ could have been changed due to unification while type-checking E_1). Now x and its associated generalized type are removed from the type environment and the type resulting from the unification is generalized, resulting in the generalized type α_1. Next, x with the generalized type α_1 is added to the type environment and E_2 is type-checked. The type of E is the type of E_2 and, once again, x and its generalized type are removed from the type environment.
- $E \equiv$ if E_1 then E_2 else E_3 fi:
  Let τ_1, τ_2 and τ_3 be the types of E_1, E_2 and E_3 respectively. τ_1 has to be unified with *bool* and τ_2 with τ_3 resulting in the type τ_4. The type of the expression E is τ_4.
- $E \equiv (E_1\ ,\ E_2)$:
  The type of E is the Cartesian product $\tau_1 \times \tau_2$ of the types τ_1 and τ_2 of E_1 and E_2 respectively.

In order to illustrate this type-check algorithm we will now type-check the expression $E \equiv \lambda\, x\, .\, \lambda\, y\, .\, (x\ (x\ y\ ))$. We start with an empty type environment T_0, and perform the following steps:

- Add (x, $\sigma 1$), the identifier x and a "fresh" type variable $\sigma 1$ to the environment resulting in $T_1 \equiv [(x, \sigma 1)]$.
- Type-check the subexpression $\lambda\, y\, .\, (x\ (x\ y\ ))$ in the environment T_1:
  - Extend the environment T_1 to $T_2 \equiv [(y, \sigma 2), (x, \sigma 1)]$ and type-check $(x\ (x\ y\ ))$:
    - First, the type of the first part of the application has to be determined: Examining the environment T_2 shows that the type of x is $\sigma 1$.
    - Next, the second part of the application i.e., $(x\ y)$ is type-checked.
      - x has type $\sigma 1$.
      - y has type $\sigma 2$.
      - In order to unify $\sigma 1$ and $\sigma 2 \rightarrow \sigma 3$, (where $\sigma 3$ is a "fresh" type variable) $\sigma 1$ is simply changed into $\sigma 2 \rightarrow \sigma 3$. As a consequence, the environment T_2 changes into $T_3 \equiv [(y, \sigma 2), (x, \sigma 2 \rightarrow \sigma 3)]$ and the type of $(x\ y)$ is $\sigma 3$.
    - The expression $(x\ (x\ y\ ))$ can now be type-checked. The type of the first part is $\sigma 2 \rightarrow \sigma 3$ and the type of the second part is $\sigma 3$. Hence we have to unify $\sigma 2 \rightarrow \sigma 3$ and $\sigma 3 \rightarrow \sigma 4$. The solution is to change $\sigma 2$ and $\sigma 3$ into $\sigma 4$. The type of $(x\ (x\ y\ ))$ therefore is $\sigma 4$ and the environment is changed into: $[(y, \sigma 4), (x, \sigma 4 \rightarrow \sigma 4)]$.
  - The type of $\lambda\, y\, .\, (x\ (x\ y\ ))$ is $\sigma 4 \rightarrow \sigma 4$ and we can delete the information about y from the environment.
- The type of the expression is $(\sigma 4 \rightarrow \sigma 4) \rightarrow (\sigma 4 \rightarrow \sigma 4)$ and the environment is rendered empty after the deletion of the identifier x and its generalized type.

4. Algebraic specification

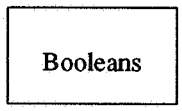
4.1. Basic tools

This section contains the algebraic specification of three basic notions (Booleans, Natural-Numbers and Keyed-Lists), necessary to specify the type-checker.

4.1.1. Booleans

These are specified in module Booleans, in which sort BOOL with constants (functions without arguments) *true* and *false*, and boolean operators ∨ (or), ∧ (and) and ¬ (negation) are defined.

Note that sort BOOL is not used to define the Mini-ML-expressions 'true' and 'false'. If sort BOOL would have been used for that purpose we would automatically define extra expressions in Mini-ML such as, e.g., ¬ true.



module Booleans
begin

exports
begin
sorts BOOL
priorities
"∨" < "∧" < "¬"
functions

true	→ BOOL	
false	→ BOOL	
BOOL "∨" BOOL	→ BOOL	**{par, assoc}**
BOOL "∧" BOOL	→ BOOL	**{par, assoc}**
"¬" BOOL	→ BOOL	**{par}**

end

variables
bool → BOOL

equations
[1] *true* ∨ bool = *true*
[2] *false* ∨ bool = bool

[3] *true* ∧ bool = bool
[4] *false* ∧ bool = *false*

[5] ¬ *true* = *false*
[6] ¬ *false* = *true*

end Booleans

4.1.2. Natural-Numbers

Natural numbers will be used in several ways throughout the specification:

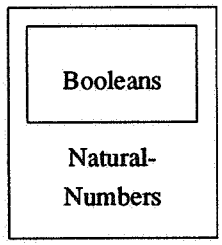
- Natural numbers are expressions in Mini-ML and as such we import module Natural-Numbers in the module Mini-ML-Expressions. For that purpose sort LEX-NAT is defined, which contains the lexical definition of the natural numbers as a sequence of one or more digits beginning with a non-zero digit.

- The variables and the generic variables in types and generalized types are renamings of the natural numbers.

Natural numbers are written in ordinary decimal notation and all operations on numbers are defined on this decimal representation. Compare this with the ''classical'' algebraic definition which uses a unary representation: the constant 0 and the successor function generate all numbers.

In the module Natural-Numbers sorts NZD (the sort of non-zero-digits), DIGIT, LEX-NAT and NAT are defined together with some functions. Additional remarks on some of the functions:

- [1-9] → NZD abbreviates the function-definitions of nine separate constants.
- 0 → DIGIT and NZD → DIGIT define all the elements of DIGIT.
- DIGIT → LEX-NAT and NZD DIGIT+ → LEX-NAT give the lexical definition of the natural numbers.
- NAT "$\equiv_{NAT}$" NAT → BOOL defines equality on natural numbers.
- *mult10* "(" NAT ")" → NAT is a hidden function. It is not a part of the **exports**-section, meaning that it can only be used in the module Natural-Numbers itself. The function defines multiplication of natural numbers by 10 and is used to define the successor of a natural number ending in 9 (remark that it is not possible to write equation **[26]** as: *succ*(n x 9) = *succ*(n x) 0. The right-hand-side of this equation cannot be parsed using the syntax defined in the **exports**-section).



module Natural-Numbers
begin

exports
begin
sorts NZD, DIGIT, LEX-NAT, NAT
functions

[1-9]	→ NZD	
0	→ DIGIT	
NZD	→ DIGIT	
DIGIT	→ LEX-NAT	
NZD DIGIT+	→ LEX-NAT	
LEX-NAT	→ NAT	
succ "(" NAT ")"	→ NAT	
NAT "$\equiv_{NAT}$" NAT	→ BOOL	**{par}**

end

imports
Booleans

functions
mult10 "(" NAT ")" → NAT

variables

n	→ NZD
x	→ DIGIT*
nat, nat_1, nat_2	→ NAT

equations

[7] $succ(0) = 1$
[8] $succ(1) = 2$
[9] $succ(2) = 3$
[10] $succ(3) = 4$
[11] $succ(4) = 5$
[12] $succ(5) = 6$
[13] $succ(6) = 7$
[14] $succ(7) = 8$
[15] $succ(8) = 9$
[16] $succ(9) = 10$
[17] $succ(n\,x\,0) = n\,x\,1$
[18] $succ(n\,x\,1) = n\,x\,2$
[19] $succ(n\,x\,2) = n\,x\,3$
[20] $succ(n\,x\,3) = n\,x\,4$
[21] $succ(n\,x\,4) = n\,x\,5$
[22] $succ(n\,x\,5) = n\,x\,6$
[23] $succ(n\,x\,6) = n\,x\,7$
[24] $succ(n\,x\,7) = n\,x\,8$
[25] $succ(n\,x\,8) = n\,x\,9$
[26] $succ(n\,x\,9) = mult10(succ(n\,x))$

[27] $0 \equiv_{NAT} 0 = true$
[28] $0 \equiv_{NAT} succ(nat) = false$
[29] $succ(nat) \equiv_{NAT} 0 = false$
[30] $succ(nat_1) \equiv_{NAT} succ(nat_2) = nat_1 \equiv_{NAT} nat_2$

[31] $mult10(0) = 0$
[32] $mult10(n\,x) = n\,x\,0$

end Natural-Numbers

4.1.3. Keyed-Lists

Keyed-Lists is a parameterized module defining lists of key-entry-pairs. This module has Keys and Entries as parameters and it is used in various ways: to define type environments and to define substitutions on types (see section 4.3. subsections 2, 3 and 4).

The **parameters**-section of the module Keyed-Lists describes the requirements on the export signatures of the modules which will be bound to the parameters Keys and Entries. Parameter Keys defines a sort KEY and an equality function $\equiv_{KEY}$ on sort KEY. Parameter Entries defines a sort ENTRY and a constant *error-entry*, which is used to render the lookup-function a total function. In our specification all defined functions are total functions. A partial function will give extra equivalence-classes in the initial algebra of the specification. Further research is needed to investigate whether partial functions could be used for some kind of error-handling in algebraic specifications.

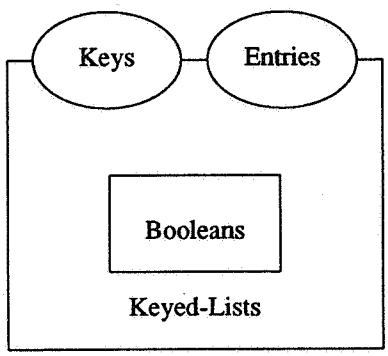
Some remarks:

- This module exports sorts PAIR and LIST.
- "(" KEY "," ENTRY ")" → PAIR and "[" {PAIR ","}* "]" → LIST are the constructor-functions of the elements of sorts PAIR and LIST respectively.
- PAIR "⊕" LIST → LIST is a function, which adds a pair to a list.
- *lookup* KEY *in* LIST → ENTRY is the lookup-function. It gives the first entry associated with a given key in a list. The function returns *error-entry* if the key is missing.
- *del* KEY *from* LIST → LIST deletes the first occurrence of a pair with the given key as key.

The definitions are such that multiple occurrences of a key are not removed. This is essential when Keyed-Lists are used as type environments and nested declarations of identifiers have to be handled. The

treatment of multiple occurrences is irrelevant in the other instances of Keyed-Lists.

Equations [35] and [37] are the first two equations of our specification in which the analogue of the built-in conditional function `if` of ASF is used. It is a polymorphic function which cannot be defined in ASF and likewise not in the combination of ASF and SDF. Its first argument has to be an element of sort BOOL. The second argument, the third argument and the result of the function have to be elements of an arbitrary sort of the specification.



module Keyed-Lists
begin

parameters
Keys
begin
sorts KEY
functions
KEY "$\equiv_{KEY}$" KEY → BOOL {**par**}
end Keys,
Entries
begin
sorts ENTRY
functions
error-entry → ENTRY
end Entries

exports
begin
sorts PAIR, LIST
functions
"(" KEY "," ENTRY ")" → PAIR
"[" {PAIR ","}* "]" → LIST
PAIR "⊕" LIST → LIST {**par**}
lookup KEY *in* LIST → ENTRY {**par**}
del KEY *from* LIST → LIST {**par**}
end

imports
Booleans

variables
key, key_1, key_2 → KEY
entry → ENTRY
pair → PAIR
pairs → {PAIR ","}*

equations

[33] pair ⊕ [pairs] = [pair, pairs]

[34] *lookup* key *in* [ ] = *error-entry*

[35] *lookup* key_1 *in* [(key_2, entry), pairs]
= *if* key_1 $\equiv_{KEY}$ key_2 *then* entry *else lookup* key_1 *in* [pairs] *fi*

[36] *del* key *from* [ ] = [ ]

[37] *del* key_1 *from* [(key_2, entry), pairs]
= *if* key_1 $\equiv_{KEY}$ key_2 *then* [pairs] *else* (key_2, entry) ⊕ *del* key_1 *from* [pairs] *fi*

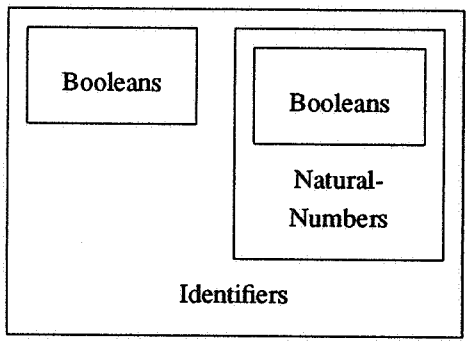
end Keyed-Lists

4.2. Specification of the syntax of Mini-ML

The modules Identifiers and Mini-ML-Expressions define the syntax of expressions in Mini-ML.

4.2.1. Identifiers

Identifiers are non-empty sequences of digits or letters preceded by a letter. Their definition is based on the definition of sorts LETTER, DIGIT-OR-LETTER and ID. The function "#" DIGIT-OR-LETTER → NAT is an auxiliary functions used to define the equality function $\equiv_{ID}$ on identifiers. It defines ordinality on sort DIGIT-OR-LETTER. Equality on identifiers is used when defining type environments (i.e., the module Identifiers is then bound to the Keys parameter of the module Keyed-Lists).



module Identifiers
begin

exports
begin
sorts LETTER, DIGIT-OR-LETTER, ID
functions

[a-z]	→ LETTER	
DIGIT	→ DIGIT-OR-LETTER	
LETTER	→ DIGIT-OR-LETTER	
LETTER DIGIT-OR-LETTER*	→ ID	
ID "$\equiv_{ID}$" ID	→ BOOL	**{par}**

end

imports
Booleans, Natural-Numbers

functions
"#" DIGIT-OR-LETTER → NAT

variables

lt_1, lt_2 → LETTER
dl_1, dl_2 → DIGIT-OR-LETTER
dls → DIGIT-OR-LETTER+
dls_1, dls_2 → DIGIT-OR-LETTER*

equations

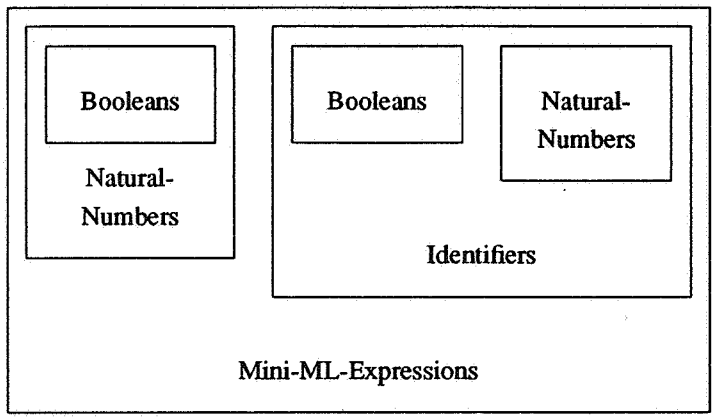
[38] $\#\,0 = 0$
[39] $\#\,1 = 1$
[40] $\#\,2 = 2$
[41] $\#\,3 = 3$
[42] $\#\,4 = 4$
[43] $\#\,5 = 5$
[44] $\#\,6 = 6$
[45] $\#\,7 = 7$
[46] $\#\,8 = 8$
[47] $\#\,9 = 9$
[48] $\#\,a = 10$
[49] $\#\,b = 11$
[50] $\#\,c = 12$
[51] $\#\,d = 13$
[52] $\#\,e = 14$
[53] $\#\,f = 15$
[54] $\#\,g = 16$
[55] $\#\,h = 17$
[56] $\#\,i = 18$
[57] $\#\,j = 19$
[58] $\#\,k = 20$
[59] $\#\,l = 21$
[60] $\#\,m = 22$
[61] $\#\,n = 23$
[62] $\#\,o = 24$
[63] $\#\,p = 25$
[64] $\#\,q = 26$
[65] $\#\,r = 27$
[66] $\#\,s = 28$
[67] $\#\,t = 29$
[68] $\#\,u = 30$
[69] $\#\,v = 31$
[70] $\#\,w = 32$
[71] $\#\,x = 33$
[72] $\#\,y = 34$
[73] $\#\,z = 35$

[74] $lt_1 \equiv_{ID} lt_2 = \#\,lt_1 \equiv_{NAT} \#\,lt_2$
[75] $lt_1\ dls \equiv_{ID} lt_2 = \mathit{false}$
[76] $lt_1 \equiv_{ID} lt_2\ dls = \mathit{false}$
[77] $lt_1\ dls_1\ dl_1 \equiv_{ID} lt_2\ dls_2\ dl_2 = lt_1\ dls_1 \equiv_{ID} lt_2\ dls_2 \wedge \#\,dl_1 \equiv_{NAT} \#\,dl_2$

end Identifiers

4.2.2. Mini-ML-Expressions

Module Mini-ML-Expressions defines the syntax of expressions in Mini-ML.



module Mini-ML-Expressions
begin

exports
begin
sorts EXP
functions

true	→ EXP
false	→ EXP
LEX-NAT	→ EXP
ID	→ EXP
"(" EXP EXP ")"	→ EXP
"λ" ID "." EXP	→ EXP
let ID "=" EXP in EXP	→ EXP
letrec ID "=" EXP in EXP	→ EXP
if EXP then EXP else EXP fi	→ EXP
"(" EXP "," EXP ")"	→ EXP

end

imports
Natural-Numbers, Identifiers

end Mini-ML-Expressions

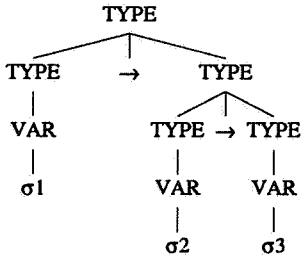
4.3. Types and tools to handle types

This section contains the algebraic specification of the syntax of types and generalized types (4.3.1.). In addition to this, several operations on types are defined. Section 4.3.2. describes type-substitutions. These have two applications: The result of unification is a type-substitution and during type-checking changes in the type environment are represented by a type-substitution. Type environments are defined in section 4.3.3. and the functions to generalize a type and to instantiate a generalized type can be found in the module Type-Instant-Generalize (in 4.3.4.). Finally, the unification algorithm is specified in section 4.3.5.

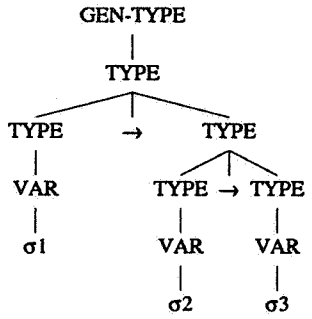
4.3.1. Types

Module Types defines the syntax of types (sort TYPE) and generalized types (sort GEN-TYPE).

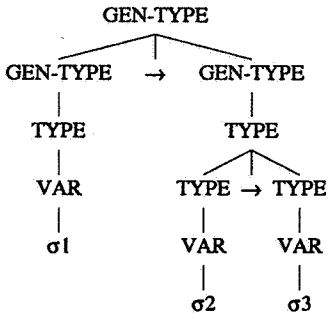
The **priorities**-section gives the priorities as they were informally specified in section 3.2.1. The first line disambiguates expressions of sort TYPE and the second line expressions of sort GEN-TYPE. The first inequality of the second line: TYPE < GEN-TYPE "→" GEN-TYPE states that the injection of TYPE in GEN-TYPE has a lower priority than the other functions of GEN-TYPE. As an example the next figure shows the four possible parse trees of the expression: $\sigma 1 \rightarrow \sigma 2 \rightarrow \sigma 3$ if no priorities would have been defined.



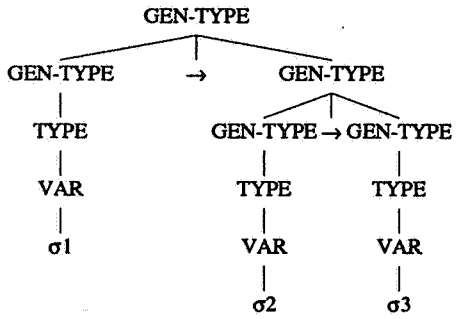
(a)



(b)



(c)



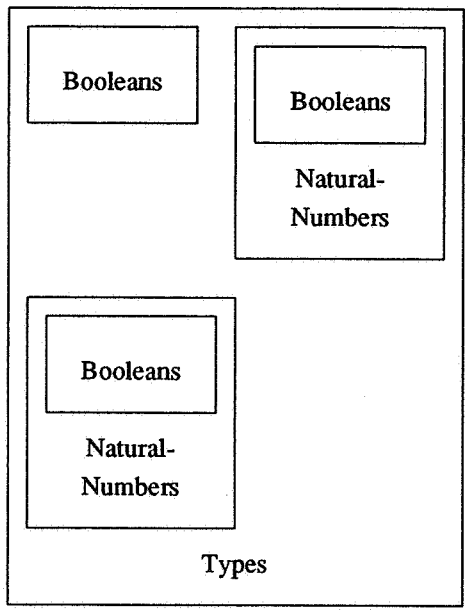
(d)

Parse tree (a) is selected if the expression is of sort TYPE and the priorities force (b) to be chosen if the expression is an element of sort GEN-TYPE.

The element *error* in TYPE is added to give a total specification of some of the functions. Especially the type-check-function will return *error* for expressions which cannot be typed. Remark that by adding this extra constant we also add elements like *error* → σ3 and *bool* × *error* to TYPE and GEN-TYPE. These are not used in the specification, but we have to be aware of the fact that these extra elements do not change the meaning of the specification.

Furthermore, the function $\equiv_{TYPE}$ tests equality on types and the functions $\in_{TYPE}$ and $\in_{GT}$ detect whether a type variable occurs in a type or generalized type.

In the **imports**-section we see the use of the natural numbers to define type variables (sort VAR) and generic type variables (sort GEN-VAR). The left-hand-side of ⇒ gives the name of the sort (in the **sorts**-section) or the syntax of the function (in the **functions**-section) which is renamed to the right-hand-side.



module Types
begin

exports
begin
sorts TYPE, GEN-TYPE
priorities
TYPE "→" TYPE < TYPE "×" TYPE
TYPE < GEN-TYPE "→" GEN-TYPE < GEN-TYPE "×" GEN-TYPE
functions

error	→ TYPE		
VAR	→ TYPE		
bool	→ TYPE		
nat	→ TYPE		
TYPE "→" TYPE	→ TYPE		{par, right-assoc}
TYPE "×" TYPE	→ TYPE		{par, left-assoc}
TYPE "$\equiv_{TYPE}$" TYPE	→ BOOL		{par}
VAR "$\in_{TYPE}$" TYPE	→ BOOL		{par}
TYPE	→ GEN-TYPE		
GEN-VAR	→ GEN-TYPE		
GEN-TYPE "→" GEN-TYPE	→ GEN-TYPE		{par, right-assoc}
GEN-TYPE "×" GEN-TYPE	→ GEN-TYPE		{par, left-assoc}
VAR "$\in_{GT}$" GEN-TYPE	→ BOOL		{par}

end

imports
Booleans,
Natural-Numbers
renamed by
sorts
NAT ⇒ VAR
functions
LEX-NAT → NAT ⇒ "σ" LEX-NAT → VAR

succ "(" NAT ")" → NAT ⇒ *next* "(" VAR ")" → VAR
NAT "$\equiv_{NAT}$" NAT → BOOL {par} ⇒ VAR "$\equiv_{VAR}$" VAR → BOOL {par}

end renaming ,

Natural-Numbers

renamed by

sorts

NAT ⇒ GEN-VAR

functions

LEX-NAT → NAT ⇒ "β" LEX-NAT → GEN-VAR
succ "(" NAT ")" → NAT ⇒ *next* "(" GEN-VAR ")" → GEN-VAR
NAT "$\equiv_{NAT}$" NAT → BOOL {par} ⇒ GEN-VAR "$\equiv_{GV}$" GEN-VAR → BOOL {par}

end renaming

variables

var, var_1, var_2 → VAR
gv → GEN-VAR
type, $type_1$, $type_2$, $type_3$, $type_4$ → TYPE
gt, gt_1, gt_2 → GEN-TYPE

equations

[78] type $\equiv_{TYPE}$ type = *true*
[79] $type_1$ $\equiv_{TYPE}$ $type_2$ = $type_2$ $\equiv_{TYPE}$ $type_1$
[80] *error* $\equiv_{TYPE}$ var = *false*
[81] *error* $\equiv_{TYPE}$ *bool* = *false*
[82] *error* $\equiv_{TYPE}$ *nat* = *false*
[83] *error* $\equiv_{TYPE}$ $type_1 \to type_2$ = *false*
[84] *error* $\equiv_{TYPE}$ $type_1 \times type_2$ = *false*
[85] var_1 $\equiv_{TYPE}$ var_2 = var_1 $\equiv_{VAR}$ var_2
[86] var $\equiv_{TYPE}$ *bool* = *false*
[87] var $\equiv_{TYPE}$ *nat* = *false*
[88] var $\equiv_{TYPE}$ $type_1 \to type_2$ = *false*
[89] var $\equiv_{TYPE}$ $type_1 \times type_2$ = *false*
[90] *bool* $\equiv_{TYPE}$ *nat* = *false*
[91] *bool* $\equiv_{TYPE}$ $type_1 \to type_2$ = *false*
[92] *bool* $\equiv_{TYPE}$ $type_1 \times type_2$ = *false*
[93] *nat* $\equiv_{TYPE}$ $type_1 \to type_2$ = *false*
[94] *nat* $\equiv_{TYPE}$ $type_1 \times type_2$ = *false*
[95] $type_1 \to type_2$ $\equiv_{TYPE}$ $type_3 \to type_4$ = $type_1$ $\equiv_{TYPE}$ $type_3$ ∧ $type_2$ $\equiv_{TYPE}$ $type_4$
[96] $type_1 \to type_2$ $\equiv_{TYPE}$ $type_3 \times type_4$ = *false*
[97] $type_1 \times type_2$ $\equiv_{TYPE}$ $type_3 \times type_4$ = $type_1$ $\equiv_{TYPE}$ $type_3$ ∧ $type_2$ $\equiv_{TYPE}$ $type_4$

[98] var $\in_{TYPE}$ *error* = *false*
[99] var_1 $\in_{TYPE}$ var_2 = var_1 $\equiv_{VAR}$ var_2
[100] var $\in_{TYPE}$ *bool* = *false*
[101] var $\in_{TYPE}$ *nat* = *false*
[102] var $\in_{TYPE}$ $type_1 \to type_2$ = var $\in_{TYPE}$ $type_1$ ∨ var $\in_{TYPE}$ $type_2$
[103] var $\in_{TYPE}$ $type_1 \times type_2$ = var $\in_{TYPE}$ $type_1$ ∨ var $\in_{TYPE}$ $type_2$

[104] var $\in_{GT}$ type = var $\in_{TYPE}$ type
[105] var $\in_{GT}$ gv = *false*
[106] var $\in_{GT}$ $gt_1 \to gt_2$ = var $\in_{GT}$ gt_1 ∨ var $\in_{GT}$ gt_2
[107] var $\in_{GT}$ $gt_1 \times gt_2$ = var $\in_{GT}$ gt_1 ∨ var $\in_{GT}$ gt_2

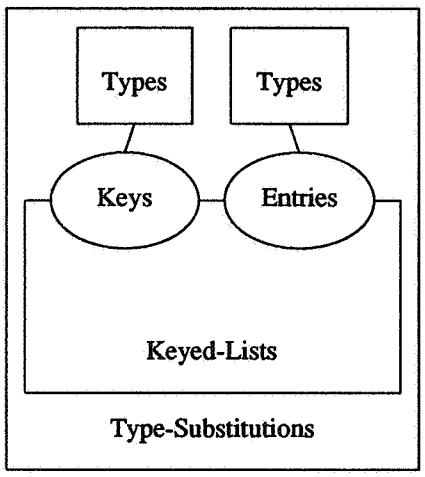
end Types

4.3.2. Type-Substitutions

Type-substitutions are used to express changes in types and type environments during type-checking. The result of the unification of a set of type-equations is a type-substitution, provided that the unification succeeds. Type-substitutions are lists of variable-type-pairs. We define type-substitutions as an instance of Keyed-Lists in which parameter Keys is bound to Types (or more precisely: sort KEY is bound to sort VAR) and parameter Entries is also bound to Types (but sort ENTRY is bound to TYPE). After binding the parameters we rename the sorts PAIR and LIST as can be seen in the **imports**-section of the specification of this module.

The functions *apply-type* and *apply-gt* define how a type-substitution should be applied to an element of TYPE or GEN-TYPE, respectively. If a type variable does not occur in the type-substitution, then the result of applying the type-substitution to a type in which this type variable occurs does not change that type variable. If a type variable occurs more than once as key in a type-substitution only the first occurrence is important. All other occurrences are ignored. Application of a type-substitution is defined such that the substitution is performed simultaneously on the whole type or generalized type. For instance, the result of *apply-type*([(σ1, σ2 → σ3), (σ2, *bool*), (σ1, *nat*)], σ1 → σ2) is (σ2 → σ3) → *bool*.

The composition of two type-substitutions (the ∘-operator) is defined such that all the above mentioned properties of type-substitutions also hold for the composition. The result of applying the composition of two type-substitutions to a type or generalized type gives the same result as first applying the right one and afterwards apply the left substitution to the result. For example: the expression [(σ2, σ1 → *bool*)] ∘ [(σ1, σ2 → σ3), (σ2, *bool*), (σ1, *nat*)] is equal to [(σ1, (σ1 → *bool*) → σ3), (σ2, *bool*), (σ1, *nat*), (σ2, σ1 → *bool*)] and these substitutions give the same result when applied to, for instance σ1 and σ2.



module Type-Substitutions
begin

exports
begin
functions
apply-type "(" TYPE-SUBS "," TYPE ")" → TYPE
apply-gt "(" TYPE-SUBS "," GEN-TYPE ")" → GEN-TYPE
TYPE-SUBS "∘" TYPE-SUBS → TYPE-SUBS **{par, assoc}**
end

imports
Keyed-Lists

Keys **bound by**
sorts
KEY ⇒ VAR
functions
KEY "$\equiv_{KEY}$" KEY → BOOL {**par**} ⇒ VAR "$\equiv_{VAR}$" VAR → BOOL {**par**}
to Types
Entries **bound by**
sorts
ENTRY ⇒ TYPE
functions
error-entry → ENTRY ⇒ *error* → TYPE
to Types
renamed by
sorts
PAIR ⇒ SUBS-PAIR
LIST ⇒ TYPE-SUBS
end renaming

variables
var → VAR
gv → GEN-VAR
type, $type_1$, $type_2$ → TYPE
gt, gt_1, gt_2 → GEN-TYPE
pairs → {SUBS-PAIR ","}*
subs → TYPE-SUBS

equations
[108] *apply-type*(subs, *error*) = *error*
[109] *apply-type*(subs, var) = *if* type $\equiv_{TYPE}$ *error then* var *else* type *fi*
when type = *lookup* var *in* subs
[110] *apply-type*(subs, *bool*) = *bool*
[111] *apply-type*(subs, *nat*) = *nat*
[112] *apply-type*(subs, $type_1 \rightarrow type_2$) = *apply-type*(subs, $type_1$) → *apply-type*(subs, $type_2$)
[113] *apply-type*(subs, $type_1 \times type_2$) = *apply-type*(subs, $type_1$) × *apply-type*(subs, $type_2$)

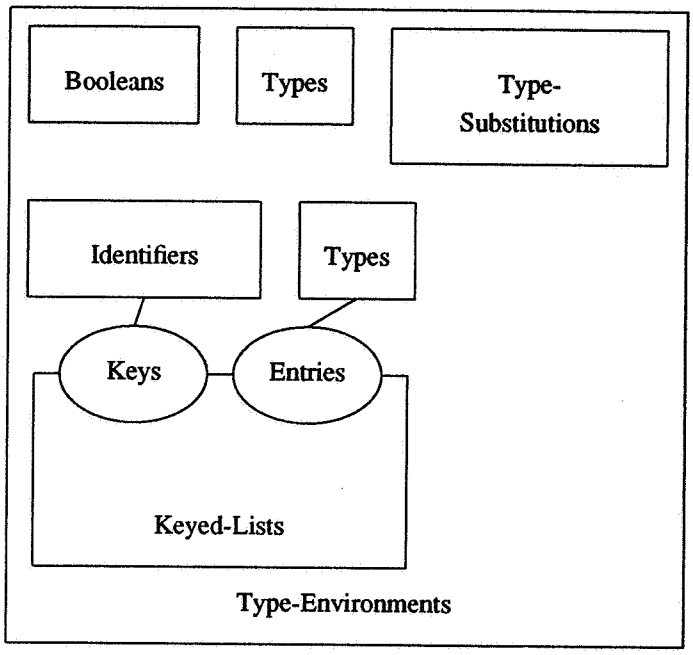
[114] *apply-gt*(subs, type) = *apply-type*(subs, type)
[115] *apply-gt*(subs, gv) = gv
[116] *apply-gt*(subs, $gt_1 \rightarrow gt_2$) = *apply-gt*(subs, gt_1) → *apply-gt*(subs, gt_2)
[117] *apply-gt*(subs, $gt_1 \times gt_2$) = *apply-gt*(subs, gt_1) × *apply-gt*(subs, gt_2)

[118] subs ∘ [ ] = subs
[119] subs ∘ [(var, type), pairs] = (var, *apply-type*(subs, type)) ⊕ (subs ∘ [pairs])

end Type-Substitutions

4.3.3. Type-Environments

Type-Environments is an instance of the module Keyed-Lists in which parameter Keys is bound to Identifiers and parameter Entries is bound to Types. It is important to allow multiple occurrences of an identifier in the list of identifier-type-pairs. This is needed to type-check expressions in which an identifier occurs more than once in a nested fashion. Type-checking the expression λ x . (x (λ x . x *true*)), for instance, should give the same result as type-checking λ x . (x (λ y . y *true*)).



module Type-Environments
begin

exports
begin
functions
apply-env "(" TYPE-SUBS "," TYPE-ENV ")" → TYPE-ENV
VAR "$\in_{ENV}$" TYPE-ENV → BOOL **{par}**
end

imports
Booleans, Types, Type-Substitutions,
Keyed-Lists
Keys **bound by**
sorts
KEY ⇒ ID
functions
KEY "$\equiv_{KEY}$" KEY → BOOL **{par}** ⇒ ID "$\equiv_{ID}$" ID → BOOL **{par}**
to Identifiers
Entries **bound by**
sorts
ENTRY ⇒ GEN-TYPE
functions
error-entry → ENTRY ⇒ *error* → GEN-TYPE
to Types
renamed by
sorts
PAIR ⇒ ENV-PAIR
LIST ⇒ TYPE-ENV
end renaming

variables

id	→	ID
var	→	VAR
gt	→	GEN-TYPE
subs	→	TYPE-SUBS
pairs	→	{ENV-PAIR ","}*

equations

[120] *apply-env*(subs, [ ]) = [ ]

[121] *apply-env*(subs, [(id, gt), pairs]) = (id, *apply-gt*(subs, gt)) ⊕ *apply-env*(subs, [pairs])

[122] var $\in_{ENV}$ [ ] = *false*

[123] var $\in_{ENV}$ [(id, gt), pairs] = var $\in_{GT}$ gt ∨ var $\in_{ENV}$ [pairs]

end Type-Environments

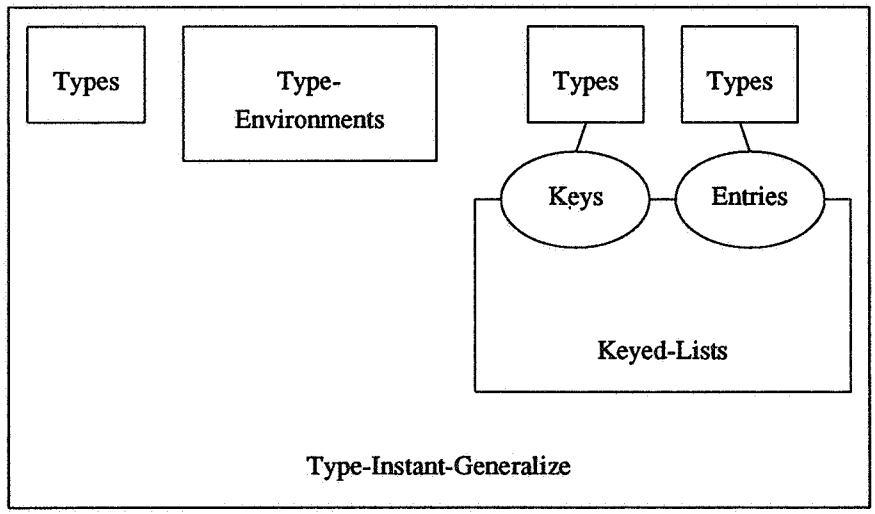
4.3.4. Type-Instant-Generalize

The module Type-Instant-Generalize gives the algebraic specification of the functions to instantiate a generalized type and to generalize a type.

To instantiate a generalized type we simply change all generic type variables occurring in the generalized type into "fresh" type variables ("fresh" type variables are created by means of the *next* function). The arguments of the function *instant* are a generalized type and a type variable. The latter should be the last type variable which has been used during type-checking. The output of the function is a tuple consisting of the result-type and of the last type variable used by the instantiation-process.

A type variable substitution is constructed during instantiation. When a generic type variable is encountered we look up the generic type variable in the substitution. If it is not there we change the generic type variable into a "fresh" type variable and add the pair of generic and "fresh" type variable to the substitution. If the generic type variable is already in the substitution, we change it into the corresponding entry. In this manner we assure that all occurrences of the same generic type variable will be instantiated to the same "fresh" type variable. We can use σ0 as an *error-entry* in the binding of the parameter Entries to the module Types because we assured in equation [132] that σ0 is never used as an ordinary entry.

To generalize a type, we just have to change all type variables in the type that do not occur in the type environment into the corresponding generic type variable.



module Type-Instant-Generalize
begin

exports
begin
sorts INSTANT-OUT
functions
instant "(" GEN-TYPE "," VAR ")" → INSTANT-OUT
"<" TYPE "," VAR ">" → INSTANT-OUT
generalize "(" TYPE "," TYPE-ENV ")" → GEN-TYPE
end

imports
Types, Type-Environments,
Keyed-Lists
Keys **bound by**
sorts
KEY ⇒ GEN-VAR
functions
KEY "$\equiv_{KEY}$" KEY → BOOL **{par}** ⇒ GEN-VAR "$\equiv_{GV}$" GEN-VAR → BOOL **{par}**
to Types
Entries **bound by**
sorts
ENTRY ⇒ VAR
functions
error-entry → ENTRY ⇒ σ0 → VAR
to Types
renamed by
sorts
PAIR ⇒ VAR-PAIR
LIST ⇒ VAR-SUBS
end renaming

sorts INS-OUT

functions
ins-subs "(" GEN-TYPE "," VAR "," VAR-SUBS ")" → INS-OUT
"<" TYPE "," VAR "," VAR-SUBS ">" → INS-OUT

variables
n → LEX-NAT
var, var_1, var_2 → VAR
gv → GEN-VAR
type, $type_1$, $type_2$ → TYPE
gt, gt_1, gt_2 → GEN-TYPE
var-subs, $var\text{-}subs_1$, $var\text{-}subs_2$ → VAR-SUBS
type-env → TYPE-ENV

equations
[124] *instant*(gt, var) = <type, var_1>
when <type, var_1, var-subs> = *ins-subs*(gt, var, [ ])

[125] *generalize*(*error*, type-env) = *error*
[126] *generalize*(σ n, type-env) = *if* σ n $\in_{ENV}$ type-env *then* σ n *else* β n *fi*
[127] *generalize*(*bool*, type-env) = *bool*
[128] *generalize*(*nat*, type-env) = *nat*

[129] *generalize*($type_1 \rightarrow type_2$, type-env)
$= generalize(type_1, \text{type-env}) \rightarrow generalize(type_2, \text{type-env})$

[130] *generalize*($type_1 \times type_2$, type-env)
$= generalize(type_1, \text{type-env}) \times generalize(type_2, \text{type-env})$

[131] *ins-subs*(type, var, var-subs) = <type, var, var-subs>

[132] *ins-subs*(gv, var, var-subs) = *if lookup* gv *in* var-subs $\equiv_{VAR}$ σ0
then <*next*(var), *next*(var), (gv, *next*(var)) ⊕ var-subs>
else <*lookup* gv *in* var-subs, var, var-subs>
fi

[133] *ins-subs*($gt_1 \rightarrow gt_2$, var, var-subs) = <$type_1 \rightarrow type_2$, var_2, var-subs_2>
when <$type_1$, var_1, var-subs_1> = *ins-subs*(gt_1, var, var-subs)
and <$type_2$, var_2, var-subs_2> = *ins-subs*(gt_2, var_1, var-subs_1)

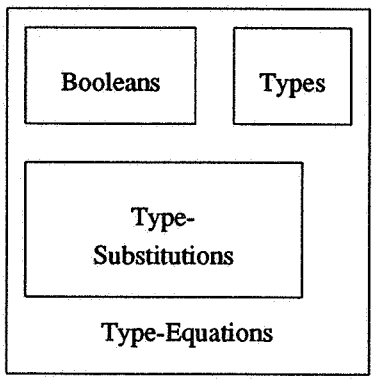
[134] *ins-subs*($gt_1 \times gt_2$, var, var-subs) = <$type_1 \times type_2$, var_2, var-subs_2>
when <$type_1$, var_1, var-subs_1> = *ins-subs*(gt_1, var, var-subs)
and <$type_2$, var_2, var-subs_2> = *ins-subs*(gt_2, var_1, var-subs_1)

end Type-Instant-Generalize

4.3.5. Type-Equations

This module defines type-equations (sort TYPE-EQ) and lists of type-equations (sort TYPE-EQS). One or more type-equations can be unified. The result of the unification of a type-equation or a list of type-equations is a boolean and a type-substitution. The boolean is *true* if the unification succeeds and *false* otherwise. The type-substitution is the minimal substitution which has to be made in the equation(s) in order to solve them.

In this module three hidden functions are needed. One function to add a type-equation to a list of type-equations and two functions that apply a type-substitution to a type-equation and a list of type-equations, respectively.



module Type-Equations
begin

exports
begin
sorts TYPE-EQ, TYPE-EQS, UNIFY-OUT
functions

TYPE *is* TYPE	→ TYPE-EQ	{par}
"{" {TYPE-EQ ","}* "}"	→ TYPE-EQS	
unify "(" TYPE-EQ ")"	→ UNIFY-OUT	
unify "(" TYPE-EQS ")"	→ UNIFY-OUT	

"<" BOOL "," TYPE-SUBS ">" → UNIFY-OUT

end

imports
Booleans, Types, Type-Substitutions

functions
TYPE-EQ "⊕" TYPE-EQS → TYPE-EQS {par}
apply-eq "(" TYPE-SUBS "," TYPE-EQ ")" → TYPE-EQ
apply-eqs "(" TYPE-SUBS "," TYPE-EQS ")" → TYPE-EQS

variables
$bool_1$, $bool_2$ → BOOL
var, var_1, var_2 → VAR
type, $type_1$, $type_2$, $type_3$, $type_4$ → TYPE
subs, $subs_1$, $subs_2$ → TYPE-SUBS
teq → TYPE-EQ
pairs → {TYPE-EQ ","}*

equations
[135] *unify*($type_1$ *is* $type_2$) = *unify*($type_2$ *is* $type_1$)
[136] *unify*(*error is* type) = <*false*, [ ]>
[137] *unify*(var_1 *is* var_2) = *if* $var_1 \equiv_{VAR} var_2$ *then* <*true*, [ ]> *else* <*true*, [(var_1, var_2)]> *fi*
[138] *unify*(var *is bool*) = <*true*, [(var, *bool*)]>
[139] *unify*(var *is nat*) = <*true*, [(var, *nat*)]>
[140] *unify*(var *is* $type_1 \rightarrow type_2$) = *if* var $\in_{TYPE}$ $type_1 \rightarrow type_2$ *then* <*false*, [ ]> *else* <*true*, [(var, $type_1 \rightarrow type_2$)]> *fi*
[141] *unify*(var *is* $type_1 \times type_2$) = *if* var $\in_{TYPE}$ $type_1 \times type_2$ *then* <*false*, [ ]> *else* <*true*, [(var, $type_1 \times type_2$)]> *fi*
[142] *unify*(*bool is bool*) = <*true*, [ ]>
[143] *unify*(*bool is nat*) = <*false*, [ ]>
[144] *unify*(*bool is* $type_1 \rightarrow type_2$) = <*false*, [ ]>
[145] *unify*(*bool is* $type_1 \times type_2$) = <*false*, [ ]>
[146] *unify*(*nat is nat*) = <*true*, [ ]>
[147] *unify*(*nat is* $type_1 \rightarrow type_2$) = <*false*, [ ]>
[148] *unify*(*nat is* $type_1 \times type_2$) = <*false*, [ ]>
[149] *unify*($type_1 \rightarrow type_2$ *is* $type_3 \rightarrow type_4$) = *unify*({ $type_1$ *is* $type_3$, $type_2$ *is* $type_4$ })
[150] *unify*($type_1 \rightarrow type_2$ *is* $type_3 \times type_4$) = <*false*, [ ]>
[151] *unify*($type_1 \times type_2$ *is* $type_3 \times type_4$) = *unify*({ $type_1$ *is* $type_3$, $type_2$ *is* $type_4$ })

[152] *unify*({ }) = <*true*, [ ]>
[153] *unify*({ teq , pairs }) = *if* $bool_1 \wedge bool_2$ *then* <*true*, $subs_2 \circ subs_1$> *else* <*false*, [ ]> *fi*
when <$bool_1$, $subs_1$> = *unify*(teq)
and <$bool_2$, $subs_2$> = *unify*(*apply-eqs*($subs_1$, { pairs }))

[154] teq ⊕ { pairs } = { teq , pairs }

[155] *apply-eq*(subs, $type_1$ *is* $type_2$) = *apply-type*(subs, $type_1$) *is apply-type*(subs, $type_2$)

[156] *apply-eqs*(subs, { }) = { }
[157] *apply-eqs*(subs, { teq , pairs }) = *apply-eq*(subs, teq) ⊕ *apply-eqs*(subs, { pairs })

end Type-Equations

4.4. Mini-ML-Type-Check

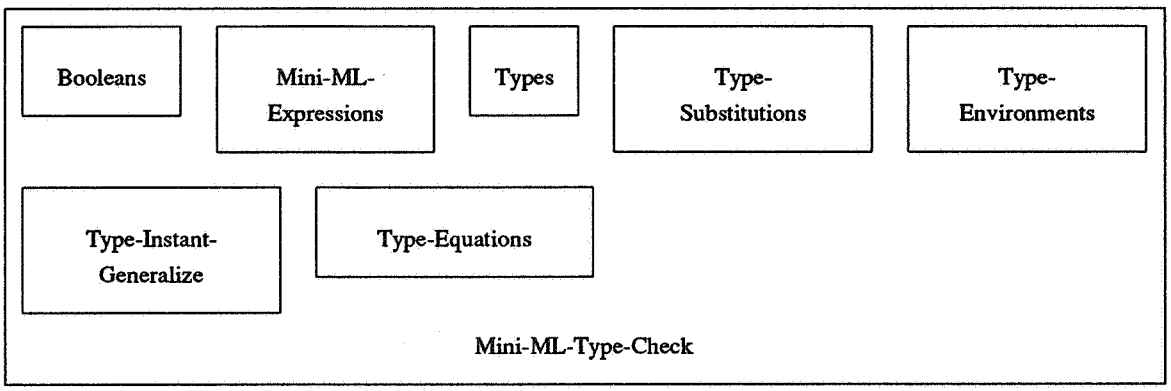
The most important function of module Mini-ML-Type-Check is the hidden function *check-exp*. The arguments of this function are:

- A syntactically correct Mini-ML expression.
- A type environment in which the expression has to be checked. This type environment should at least contain the type of the identifiers in the expression which are not bound by a lambda-, a let- or a letrec-construction.
- A type variable, which is the last type variable used in type-checking.

The output of the function *check-exp* is:

- The most general type of the expression.
- A type-substitution containing the changes in the type environment due to unification.
- The last type variable used in the type-checking-process.

This module exports the function *check*, which returns *true* if a given expression is typeable and *false* if it is not. The function *check-exp* is defined in such a way that it returns *error* when the expression cannot be typed.



module Mini-ML-Type-Check
begin

exports
begin
functions
check "(" EXP ")" → BOOL
end

imports
Booleans, Mini-ML-Expressions,
Types, Type-Substitutions, Type-Environments,
Type-Instant-Generalize, Type-Equations

sorts CHECK-OUT

functions
check-exp "(" EXP "," TYPE-ENV "," VAR ")" → CHECK-OUT
"<" TYPE "," TYPE-SUBS "," VAR ">" → CHECK-OUT

variables

unifiable	→ BOOL
nat	→ LEX-NAT
id	→ ID
exp, exp_1, exp_2, exp_3	→ EXP
var, var_1, var_2, var_3	→ VAR
type, $type_1$, $type_2$, $type_3$	→ TYPE
gt_1	→ GEN-TYPE
subs, $subs_1$, $subs_2$, $subs_3$, u-subs	→ TYPE-SUBS
env	→ TYPE-ENV

equations

[158] *check*(exp) = ¬ (type ≡$_{TYPE}$ *error*)
when <type, subs, var> = *check-exp*(exp, [ ], σ0)

[159] *check-exp*(true, env, var) = <*bool*, [ ], var>
[160] *check-exp*(false, env, var) = <*bool*, [ ], var>
[161] *check-exp*(nat, env, var) = <*nat*, [ ], var>
[162] *check-exp*(id, env, var) = <type, [ ], var_1>
when <type, var_1> = *instant*(*lookup* id *in* env, var)
[163] *check-exp*((exp_1 exp_2), env, var)
= *if* $type_1$ ≡$_{TYPE}$ *error* ∨ $type_2$ ≡$_{TYPE}$ *error* ∨ ¬ unifiable
then <*error*, [ ], var>
else <*apply-type*(u-subs, *next*(var_2)), u-subs ∘ $subs_2$ ∘ $subs_1$, *next*(var_2)>
fi
when <$type_1$, $subs_1$, var_1> = *check-exp*(exp_1, env, var)
and <$type_2$, $subs_2$, var_2> = *check-exp*(exp_2, *apply-env*($subs_1$,env), var_1)
and <unifiable, u-subs> = *unify*(*apply-type*($subs_2$, $type_1$) *is* $type_2$ → *next*(var_2))
[164] *check-exp*(λ id . exp, env, var)
= *if* type ≡$_{TYPE}$ *error*
then <*error*, [ ], var>
else <*apply-type*(subs, *next*(var)) → type, subs, var_1>
fi
when <type, subs, var_1> = *check-exp*(exp, (id, *next*(var)) ⊕ env, *next*(var))
[165] *check-exp*(let id = exp_1 in exp_2, env, var)
= *if* $type_1$ ≡$_{TYPE}$ *error* ∨ $type_2$ ≡$_{TYPE}$ *error*
then <*error*, [ ], var>
else <$type_2$, $subs_2$ ∘ $subs_1$, var_2>
fi
when <$type_1$, $subs_1$, var_1> = *check-exp*(exp_1, env, var)
and gt_1 = *generalize*($type_1$, *apply-env*($subs_1$,env))
and <$type_2$, $subs_2$, var_2> = *check-exp*(exp_2, (id, gt_1) ⊕ *apply-env*($subs_1$,env), var_1)
[166] *check-exp*(letrec id = exp_1 in exp_2, env, var)
= *if* $type_1$ ≡$_{TYPE}$ *error* ∨ $type_2$ ≡$_{TYPE}$ *error* ∨ ¬ unifiable
then <*error*, [ ], var>
else <$type_2$, $subs_2$ ∘ u-subs ∘ $subs_1$, var_2>
fi
when <$type_1$, $subs_1$, var_1> = *check-exp*(exp_1, (id, *next*(var)) ⊕ env, *next*(var))
and <unifiable, u-subs> = *unify*($type_1$ *is apply-type*($subs_1$, *next*(var)))
and gt_1 = *generalize*(*apply-type*(u-subs, $type_1$), *apply-env*(u-subs ∘ $subs_1$,env))
and <$type_2$, $subs_2$, var_2>
= *check-exp*(exp_2, (id, gt_1) ⊕ *apply-env*(u-subs ∘ $subs_1$,env), var_1)

[167] *check-exp*(if exp_1 then exp_2 else exp_3 fi, env, var)
= *if* $type_1 \equiv_{TYPE}$ *error* $\vee$ $type_2 \equiv_{TYPE}$ *error* $\vee$ $type_3 \equiv_{TYPE}$ *error* $\vee \neg$ unifiable
then <*error*, [ ], var>
else <*apply-type*(u-subs, $type_3$), u-subs $\circ$ $subs_3$ $\circ$ $subs_2$ $\circ$ $subs_1$, var_3>
fi
when <$type_1$, $subs_1$, var_1> = *check-exp*(exp_1, env, var)
and <$type_2$, $subs_2$, var_2> = *check-exp*(exp_2, *apply-env*($subs_1$,env), var_1)
and <$type_3$, $subs_3$, var_3> = *check-exp*(exp_3, *apply-env*($subs_2 \circ subs_1$,env), var_2)
and <unifiable, u-subs> = *unify*({*apply-type*($subs_3 \circ subs_2$, $type_1$) *is bool*,
apply-type($subs_3$, $type_2$) *is* $type_3$})

[168] *check-exp*((exp_1 , exp_2), env, var)
= *if* $type_1 \equiv_{TYPE}$ *error* $\vee$ $type_2 \equiv_{TYPE}$ *error*
then <*error*, [ ], var>
else <*apply-type*($subs_2$, $type_1$) $\times$ $type_2$, $subs_2 \circ subs_1$, var_2>
fi
when <$type_1$, $subs_1$, var_1> = *check-exp*(exp_1, env, var)
and <$type_2$, $subs_2$, var_2> = *check-exp*(exp_2, *apply-env*($subs_1$,env), var_1)

end Mini-ML-Type-Check

To illustrate the specification we will present the same example as was given in section 3.2.2.:

(1?) *check*(λ x . λ y . (x (x y))) = ???
(1?) *check-exp*(λ x . λ y . (x (x y)), [ ], σ0) = ???
(1?) *check-exp*(λ y . (x (x y)), (x, *next*(σ0)) $\oplus$ [ ], *next*(σ0))
= *check-exp*(λ y . (x (x y)), [(x, σ1)], σ1) = ???
(1?) *check-exp*((x (x y)), (y, *next*(σ1)) $\oplus$ [(x, σ1)], *next*(σ1))
= *check-exp*((x (x y)), [(y, σ2), (x, σ1)], σ2) = ???
(1) *check-exp*(x, [(y, σ2), (x, σ1)], σ2)
= <σ1, [ ], σ2>
(2?) *check-exp*((x y), *apply-env*([ ],[(y, σ2), (x, σ1)]), σ2)
= *check-exp*((x y), [(y, σ2), (x, σ1)], σ2) = ???
(1) *check-exp*(x, [(y, σ2), (x, σ1)], σ2)
= <σ1, [ ], σ2>
(2) *check-exp*(y, *apply-env*([ ],[(y, σ2), (x, σ1)]), σ2)
= *check-exp*(y, [(y, σ2), (x, σ1)], σ2)
= <σ2, [ ], σ2>
(3) *unify*(*apply-type*([ ], σ1) *is* σ2 $\rightarrow$ *next*(σ2))
= *unify*(σ1 *is* σ2 $\rightarrow$ σ3)
= <*true*, [(σ1, σ2 $\rightarrow$ σ3)]>
(2=) *check-exp*((x y), [(y, σ2), (x, σ1)], σ2)
= <*apply-type*([(σ1, σ2 $\rightarrow$ σ3)], *next*(σ2)),
[(σ1, σ2 $\rightarrow$ σ3)] $\circ$ [ ] $\circ$ [ ],
next(σ2)>
= <σ3, [(σ1, σ2 $\rightarrow$ σ3)], σ3>
(3) *unify*(*apply-type*([(σ1, σ2 $\rightarrow$ σ3)], σ1) *is* σ3 $\rightarrow$ *next*(σ3))
= *unify*(σ2 $\rightarrow$ σ3 *is* σ3 $\rightarrow$ σ4)
= <*true*, [(σ2, σ4), (σ3, σ4)]>
(1=) *check-exp*((x (x y)), [(y, σ2), (x, σ1)], σ2)
= <*apply-type*([(σ2, σ4), (σ3, σ4)], *next*(σ3)),
[(σ2, σ4), (σ3, σ4)] $\circ$ [(σ1, σ2 $\rightarrow$ σ3)] $\circ$ [ ],
next(σ3)>
= <σ4, [(σ1, σ4 $\rightarrow$ σ4), (σ2, σ4), (σ3, σ4)], σ4>
(1=) *check-exp*(λ y . (x (x y)), [(x, σ1)], σ1)
= <*apply-type*([(σ1, σ4 $\rightarrow$ σ4), (σ2, σ4), (σ3, σ4)], *next*(σ1)) $\rightarrow$ σ4,
[(σ1, σ4 $\rightarrow$ σ4), (σ2, σ4), (σ3, σ4)],

$\sigma 4>$
$= <\sigma 4 \rightarrow \sigma 4, [(\sigma 1, \sigma 4 \rightarrow \sigma 4), (\sigma 2, \sigma 4), (\sigma 3, \sigma 4)], \sigma 4>$

(1=) *check-exp*$(\lambda\, x\, .\, \lambda\, y\, .\, (\, x\, (\, x\, y\,)\,),\, [\,],\, \sigma 0)$
$= <$*apply-type*$([(\sigma 1, \sigma 4 \rightarrow \sigma 4), (\sigma 2, \sigma 4), (\sigma 3, \sigma 4)],$ *next*$(\sigma 0)) \rightarrow (\, \sigma 4 \rightarrow \sigma 4\,),$
$[(\sigma 1, \sigma 4 \rightarrow \sigma 4), (\sigma 2, \sigma 4), (\sigma 3, \sigma 4)],$
$\sigma 4>$
$= <(\, \sigma 4 \rightarrow \sigma 4) \rightarrow (\, \sigma 4 \rightarrow \sigma 4\,), [(\sigma 1, \sigma 4 \rightarrow \sigma 4), (\sigma 2, \sigma 4), (\sigma 3, \sigma 4)], \sigma 4>$

(1=) *check*$(\lambda\, x\, .\, \lambda\, y\, .\, (\, x\, (\, x\, y\,)\,))$
$= \neg\, (\, (\, \sigma 4 \rightarrow \sigma 4) \rightarrow (\, \sigma 4 \rightarrow \sigma 4\,) \equiv_{TYPE} error\,)$
$=$ *true*

5. Implementations of this specification

How can we derive an implementation from an algebraic specification, assuming that the specification satisfies certain constraints and is augmented with extra information?

The first question we have to answer is: What is an implementation of an algebraic specification? We have transformed the algebraic specification into a confluent and terminating term rewriting system and this has been implemented using C-Prolog [PWBBP85] and, alternatively, the Equation Interpreter [ODo85]. These implementations can transform a closed expression (an expression without variables) into its normal form, where the normal forms are implicitly defined by the term rewriting system. These implementations are no general equation solvers and some of them cannot reduce open expressions in an appropriate way.

The steps needed to implement the specification as a term rewriting system are:

- Remove syntax

  First we have to remove the user-defined syntax from the specification in order to create a specification written in "pure" ASF. This amounts to changing all SDF descriptions into corresponding signatures and parsing the equations such that only equations with terms over these signatures remain. In our specification we have added enough syntax to ensure that each equation can be parsed unambiguously. In general the syntax defined by the user could be ambiguous and in that case even some of the equations could be multi-interpretable. Further study has to be done to investigate whether it is possible to work with ambiguous syntax definitions and/or equations. The result of this step is a modular ASF specification.

- Normalize

  Next, we normalize this ASF-specification, i.e., we remove all modular structure from the module until a module without imports remains. This normalization strategy is described in [BHK87]. It is necessary to investigate the possibilities for a modular implementation of an algebraic specification, but at this moment we only know how to implement normalized modules.

- Transform to a term rewriting system

  The normalized ASF-specification of the module has to be transformed into a (conditional) term rewriting system. For most equations it is obvious which side is more complex than the other one and, at least in our case, it is easy to give a direction to most (conditional) equations. Depending on the strategy chosen to implement the term rewriting system, the rewrite-rules have to satisfy some constraints. Note that the algebraic specification as presented does not give any information about normal forms of expressions.

- Implement the term rewriting system

  Finally, we created several implementations of the term rewriting system transforming it into source code for the Equation Interpreter and C-Prolog. Both languages are untyped and hence all type-information has to be encoded in extra parameters of the functions or in the names of the functions.

  - The specification language for the Equation Interpreter contains a *where*-construct, but this could not be used to implement our conditional rewrite rules. It turns out that all occurrences of conditional equations in our specification serve the purpose of abbreviating equations by introducing auxiliary variables. The first implementation using the Equation Interpreter was created by substituting the appropriate term for each auxiliary variable introduced in the conditional part. This leads to a very inefficient implementation because the Interpreter cannot recognize

identical subterms in the right-hand side of equations and hence the values of identical subexpressions are often recomputed. Robert Strandh [Str86] suggested to circumvent this problem by introducing auxiliary functions. Each equation of the form $L(x_1, ..., x_n) = R(x_1, ..., x_n, y)$ **when** $y = f(x_1, ..., x_n)$ is changed into two equations: $L(x_1, ..., x_n) = g(x_1, ..., x_n, f(x_1, ..., x_n))$ and $g(x_1, ..., x_n, y) = R(x_1, ..., x_n, y)$.

- Several implementations have been made using C-Prolog. The first one used the translation scheme described by Drosten and Ehrich in [DE84]. This scheme performs innermost reduction of closed terms. Innermost reduction is a very inefficient reduction strategy: it recomputes the normal form of identical terms and it loops if the term rewriting system is not strongly terminating (the latter is not the case in our specification). We experimented with a "cache"-mechanism in which each term and its computed normal form are stored in a database: this eliminates repeated computations of normal forms for the same term. Next, a scheme performing parallel outermost reduction was implemented (which is described in [Die86]). Both implementation strategies can be generated automatically from an algebraic specification without adding extra information to it. The fastest implementation used the translation method described in [EY86]. For this translation it is necessary to distinguish defined functions from constructor functions. The graph of each defined function is a relation and each equation can be transformed into Horn-clauses describing these relations. Unfortunately, it is not evident how such an implementation can be generated automatically from an algebraic specification in which the constructors are not indicated as such.

6. Further research

The work described here raises several questions that require further research:

- While creating the specification for the type-checker we gained experience with a combination of the formalisms ASF and SDF. Research has to be done to improve both formalisms and to carefully design the combination of them in order to achieve a convenient specification formalism with a solid theoretical base.
- The implementation of the algebraic specification gives rise to questions like: How should an algebraic specification be implemented (in an efficient and/or modular way)?
- Finally, we might profit from studying other (algebraic) specifications of type-checkers for ML (or Mini-ML).

We will now discuss these questions into somewhat more detail.

6.1. Improvement of the formalism

- The hiding-mechanism should be improved:
  - In ASF, the exported signature of each imported module is automatically included in the export signature of the importing module. Hence Natural-Numbers not only exports the signature given in its own **exports**-section but also the **exports**-section of module Booleans. As a consequence, we cannot use sort BOOL to define the Mini-ML-expressions 'true' and 'false' (see section 4.1.1.) and similarly sort LEX-NAT is necessary to define the natural numbers in Mini-ML instead of sort NAT (see section 4.1.2.).
  - Sorts NZD and DIGIT in Natural-Numbers are in fact irrelevant outside the module. These sorts are needed to define the grammar of the specification. We do not want to change their internal structure outside the module and we do not want to write equations over these sorts. It is not possible to hide these sorts because in that case all functions in the **exports**-section of the module which use these sorts must be hidden too. The hiding-mechanism of the formalism should be augmented or extended to give the user of the specification formalism the possibility to express these kinds of properties of sorts.
- It is not possible to give a generic specification of a unification algorithm in ASF that has an arbitrary signature as parameter.
- We have not yet experimented with modularization in SDF. The combination of modularization with the separation of syntax-definitions in a lexical and a context-free part is not evident. We envisage, for example, problems when two modules with different layout conventions have to be combined.

- In many cases we can subdivide the set of functions of an algebraic specification in a set of constructors and a set of functions defined over these constructors. It would be convenient if we could state that all closed terms that are not equal to members of a predefined class of terms are "error"-elements. In our specification: It would be nice if we could define all terms which are not equal to *true* to be equal to *false* or at least that an error-message would be generated in these cases (This would be similar to what is done in TYPOL [Kah87]). Such a construction would shorten the specification considerably but in general it may be difficult to give a sound semantics to it.
- In the combination of ASF and SDF we have to answer the question what has to be done with ambiguous syntax definitions (see also section 2.3.). One of the major problems of ambiguities is that it is in general undecidable whether a syntax definition is ambiguous ([HU79], p. 200). Ambiguities of terms in several sorts should be supported by the formalism because it would be unpleasant if the user of the formalism has to come up with extra syntax for zero as a natural number, integer, rational number and real. Ambiguity of terms within a sort are more hazardous because it is not clear which interpretation of the term is intended by the user. Restricting the formalism to unambiguous interpretations of each term forces one to add extra syntax to the specification.

6.2. Implementing algebraic specifications

- As mentioned before one of the greatest problems is to decide what an implementation should do. If the specification is a term rewriting system or if it can automatically be transformed into a term rewriting system, an implementation can reduce closed terms to their normal form. Confluence and termination of the term rewriting system is needed to assure uniqueness of normal forms and termination of the program. It would be very nice if an implementation could also reduce open expressions or solve equations (this is the case in, for instance, RAP [Hus86]).
- Which subset of algebraic specifications can be implemented? Is it necessary to restrict ourselves to specifications which are in fact confluent and terminating term rewriting systems?
- Which information should be added to an algebraic specification to generate an efficient implementation for it?
- Research has to be done to study the possibilities to infer a modular implementation from a modular specification.

6.3. The specification of a type-checker for Mini-ML

- Is it possible to generate an *incremental* type-checker from our specification? It might be possible to change the specification in order to make it easier to generate an incremental type-checker. The general idea is to assign a set of type-equations to each sub-expression of the Mini-ML program to be type-checked. A sub-expression is potentially correct if its set of equations can be solved. A program is correct if all of its sub-expressions are correct and if no unbound identifiers occur in it.
- Our specification includes not only all information on *what* a type-checker for Mini-ML should do but it also states *how* it should be done. It would be nice if the specification would only define what type-checking is and leave the algorithmic details to the implementor of the specification.

7. Conclusions

The conclusions of the research which led to this paper are twofold:

- We wanted to investigate whether it is possible to give an acceptable algebraic specification of one of the toughest examples of type-checking, i.e., a type-checker which involves polymorphism and type-inference. Our specification shows that such a specification can indeed be given, but that it contains a lot of low-level details.
- As the specification was developed we felt the urge to use more liberal syntax than the syntax which was allowed in ASF. This led to a first attempt to combine ASF with SDF. This has not only identified some problem areas in the formal specification of SDF but it has also convinced us of the potential of algebraic specifications with user-definable syntax.

Acknowledgements

Linda van der Gaag, Paul Klint and Hans Mulder commented on earlier drafts of this paper. Discussions with Niek van Diepen, Jan Rekers, Ard Verhoog and especially Jan Heering and Paul Klint helped to improve the specification.

References

[BHK85] J.A. Bergstra, J. Heering, and P. Klint, "Algebraic definition of a simple programming language," Report CS-R8504, Centre for Mathematics and Computer Science, Amsterdam, February 1985.

[BHK86] J.A. Bergstra, J. Heering, and P. Klint, "Module algebra," Report CS-R8617, Centre for Mathematics and Computer Science, Amsterdam, May 1986.

[BHK87] J.A. Bergstra, J. Heering, and P. Klint, "ASF — an algebraic specification formalism," Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam, January 1987.

[CDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn, "A simple applicative language: mini-ML," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 13-27, ACM, 1986.

[CW85] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *Computing Surveys*, vol. 17, no. 4, pp. 471-522, 1985.

[Car84] L. Cardelli, "Basic polymorphic typechecking," Computing Science Technical Report No. 112, AT&T Bell Laboratories, Murray Hill, NJ, September 1984.

[DE84] K. Drosten and H.-D. Ehrich, "Translating algebraic specifications to Prolog programs," Informatik-Bericht Nr. 84-08, Technische Universität Braunschweig, Braunschweig.

[DM82] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 207-212, ACM, Albuquerque, New Mexico, 1982.

[Die86] N.W.P. van Diepen, "A study in algebraic specification: a language with goto-statements," Report CS-R8627, Centre for Mathematics and Computer Science, Amsterdam, August 1986.

[EM85] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications*, vol. I, Equations and Initial Semantics, Springer-Verlag, 1985.

[EY86] M.H. van Emden and K. Yukawa, "Equational logic programming," Technical Report CS-86-05, University of Waterloo, Department of Computer Science, Waterloo, Ontario, March 1986.

[FGJM85] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, and J. Meseguer, "Principles of OBJ2," in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp. 52-66, ACM, 1985.

[HHKR87] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers, "SDF Reference Manual," Centre for Mathematics and Computer Science, Amsterdam, to appear 1987.

[HK86] J. Heering and P. Klint, "User definable syntax for specification languages," in *Proceedings NGI-SION Symposium 4*, pp. 63-73, NGI, 1986. Also in: *ESPRIT '86: Results and Achievements*, North-Holland, 1987, pp. 619-630.

[HU79] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Series in Computer Science, Addison-Wesley, New York and New Jersey, March 1979.

[Hus86] H. Hussmann, "Rapid prototyping for algebraic specifications — RAP system user's manual," Draft Report, Fakultät für Mathematik und Informatik, Universität Passau, Passau, September 1986.

[Kah87] G. Kahn, "Natural semantics," in *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, ed. F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Lecture Notes on Computer Science, vol. 247, Springer-Verlag, 1987.

[MG85] J. Meseguer and J.A. Goguen, "Initiality, induction, and computability," in *Algebraic Methods in Semantics*, ed. M. Nivat and J. Reynolds, pp. 459-541, Cambridge University

Press, 1985.

[Mil85] R. Milner, "The Standard ML core language," *Polymorphism, the ML/LCF/Hope Newsletter*, vol. II, no. 2, 1985.

[ODo85] M.J. O'Donnell, *Equational Logic as a Programming Language*, MIT Press, 1985.

[PWBBP85] F. Pereira, D. Warren, D. Bowen, L. Byrd, and L. Pereira, *C-Prolog User's Manual*, version 1.5., SRI International, Menlo Park, California, June 18, 1985.

[Rek87] J. Rekers, "A parser generator for finitely ambiguous context-free grammars," Proceedings of the conference on Computing Science in The Netherlands 1987. Also in: Report CS-R8712, Centre for Mathematics and Computer Science, Amsterdam, March 1987.

[Str86] R. Strandh, Personal communication, 16 July 1986.

[Wal86] H.R. Walters, "An annotated algebraic specification of the static semantics of Pool," Report FVI 86-20, University of Amsterdam, Computer Science Department, September 1986.