



Centrum voor Wiskunde en Informatica

REPORTRAPPORT

Animators and error reporters for generated programming environments

T.B. Dinesh and F. Tip

Computer Science/Department of Software Technology

CS-R9253 1992

Report CS-R9253
ISSN 0169-118X
CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.

SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Animators and Error Reporters for Generated Programming Environments

T.B. Dinesh and F. Tip

CWI

*P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
dinesh@cwi.nl, tip@cwi.nl*

Abstract

We study animators and error reporters for generated programming environments. An *error reporter* is a tool for indicating the exact position of a type-error in the source text. An *animator* visualizes program execution; typically, it highlights the statement that is currently executing. Applications of both tools are mainly to be found in the areas of debugging and tutoring. Instead of explicitly extending language specifications with these facilities, we claim that error reporters and animators can be generated from existing specifications for type-checkers and interpreters with little effort; to this end, a simple pattern-matching mechanism is used in conjunction with origin tracking, a generic tracing technique. In this paper, we discuss our claim, and at the same time investigate the limitations and deficiencies of origin tracking. Our techniques are illustrated using an example language named CLaX, a Pascal relative. The full specifications of the CLaX syntax, type-checker and interpreter are included in appendices.

1991 Mathematics Subject Classification: 68N20 [**Software**]: Compilers and generators, 68Q55 [**Theory of computing**]: Semantics, 68Q65 [**Theory of computing**]: Abstract data types; algebraic specification.

1991 CR Categories: D.2.5 [**Software engineering**]: Testing and debugging, D.2.6 [**Software engineering**]: Programming environments, D.3.1 [**Programming languages**]: Formal definitions and theory, F.3.2 [**Logics and meanings of programs**]: Semantics of programming languages.

Key Words & Phrases: Animators, debugging, error messages, error reporters, interpreters, programming environments, origin tracking, type-checking.

Note: Partial support received from the European Communities under ESPRIT project 5399: Compiler Generation for Parallel Machines – COMPARE.

Contents

1	Introduction	3
2	CLaX	4
3	Origin tracking	7
4	Error Reporters	8
4.1	Method	8
4.2	Limitations	10
5	Animators	11
5.1	Definition of events	11
5.2	Determining subjects	13
5.3	Example	13
5.4	Limitations	14
6	Related Work	16
7	Concluding Remarks	17
A	The syntax of CLaX	21
A.1	Comments	21
A.2	Tokens	22
A.3	Constants	22
A.4	Data Types	22
A.5	Declarations	23
A.6	Expressions	23
A.7	Statements	24
A.8	Blocks and Programs	25
A.9	The generated CLaX environment	25
B	CLaX syntax modules	27
B.1	module SyntaxLayout	27
B.2	module SyntaxTokens	28
B.3	module SyntaxConsts	29
B.4	module SyntaxTypes	30
B.5	module SyntaxHeaders	31
B.6	module SyntaxExpr	32
B.7	module SyntaxStats	34
B.8	module SyntaxProgram	35
C	A type-checker for CLaX	36
C.1	Overview	36
C.2	Extending the language	36
C.3	Type-environments	37
C.4	Evaluation of expressions over abstract values	37

C.5	Type-checking procedures	38
C.6	Type-checking labels	39
C.7	Type-checking programs	39
C.8	Generating error messages	40
C.9	Example	41
C.10	Type-checking incomplete programs	43
D	CLaX type-checking modules	45
D.1	module TcSyntaxExt	45
D.2	module TcTenv	46
D.3	module TcExpr	48
D.4	module TcBooleans	50
D.5	module TcProc	51
D.6	module TcLabel	53
D.7	module TcNint	55
D.8	module Tc	58
D.9	module TcErrors	61
E	An interpreter for CLaX	64
E.1	Basic Datatypes	64
E.2	Evaluation of Expressions	68
E.3	Execution of Programs and Statements	71
E.4	Input/Output	73
F	CLaX interpreter modules	76
F.1	module CodeStacks	76
F.2	module DataStacks	78
F.3	module EvalExpr	82
F.4	module EvalProgram	87
F.5	module Initializations	91
F.6	module Input	93
F.7	module Output	95
F.8	module Arithmetic	96
F.9	module Lisp	98

1 Introduction

We study animators and error reporters for generated programming environments. In our setting, a programming environment consists of a syntax-directed editor, a type-checker, and an interpreter for a programming language. We use the ASF+SDF Meta Environment [Hen91, Kli93, Wal91] to generate programming environments from algebraic specifications.

An *error reporter* is a tool for pinpointing the position of a type-error in the source text. An *animator* visualizes program execution; typically, it highlights the statement that is currently executing. Applications of both tools are mainly to be found in the areas of debugging and tutoring. Instead of explicitly extending language specifications with these facilities, we claim that error reporters and animators can be generated from

```

PROGRAM fibonacci;
DECLARE
  lab : LABEL;
  count : INTEGER;
  fib : ARRAY[1..20] OF INTEGER;
BEGIN
  count := 3;
  fib[1] := 1;
  fib[2] := 1;
  lab: fib[count] := fib[count-1] + fib[count-2];
  count := count + 1;
  WRITE("count = "); WRITE(count); WRITE("\n");
  IF count <= 20 THEN
    GOTO lab
  END
END.

```

Figure 1: Example of a CLaX program.

existing specifications for type-checkers and interpreters with little effort; to this end, a simple pattern-matching mechanism is used in conjunction with origin tracking, a generic tracing technique [DKT92]. In this paper, we discuss our claim, and at the same time investigate the limitations and deficiencies of origin tracking.

To illustrate our techniques, we use an example language named CLaX (short for Compare Language eXample). CLaX is a Pascal relative, and is indirectly derived from the example language LAX used in [GW84]. CLaX features the following language concepts: types, type coercion, overloaded operators, arrays, procedures, reference and value parameters, nested scopes, assignment statements, loop statements, conditional statements, and goto statements.

The remainder of this paper is organized as follows. First, we give a brief introduction to CLaX. Then, a short description of origin tracking is presented. After that, we describe in detail how error reporters and animators can be derived from the CLaX type-checker and interpreter specifications, respectively. Finally, related work in the areas of animation and error reporting is discussed, and some conclusions are presented. In particular, we evaluate the use of origin tracking, and we indicate possible directions for future research.

2 CLaX

CLaX is a Pascal-like imperative programming language, developed to serve as the demonstration language of the COMPARE project (ESPRIT). The most interesting features of CLaX are: nested scopes, overloaded operators, arrays, goto statements, and procedures with reference and value parameters. In Figure 1, an example of a CLaX program is shown.

We use the combined formalism ASF+SDF to define the syntax, the static semantics, and the dynamic semantics of CLaX. ASF+SDF is a combination of the formalisms ASF and SDF. The Algebraic Specification Formalism, ASF, [BHK89] features first-order

signatures, conditional equations, modules, and facilities for import, export, and hiding. The Syntax Definition Formalism, SDF [HHKR89], is used for the simultaneous definition of the lexical syntax, the context-free syntax, and the abstract syntax of a language. ASF+SDF [Hen91, Wal91] is an integrated formalism for the definition of the syntax and semantics of languages. In this paper, we elaborate on the features of ASF+SDF when needed, in an informal way only. For more details, the reader is referred to the cited papers.

The ASF+SDF Meta-environment [Kli93] is an implementation of ASF+SDF. By interpreting equations as rewrite-rules, specifications can be executed as term rewriting systems [Klo91]. In particular, the specifications of the static and dynamic semantics of CLaX described in this paper are executable. The former defines a type-checker for CLaX programs, and the latter an interpreter for CLaX programs.

In Appendix A, the specification of the CLaX syntax is described in detail; Appendix B contains the full specification text.

The CLaX type-checker is specified in a non-standard way. Its basic strategy consists of the following steps:

1. Distribution of the context (i.e., type information for all identifiers in the current scope) over every program construct.
2. Replacement of identifiers and values by a common abstract representation. We use the *type* of a value as its abstract representation.
3. Evaluation of expressions using the abstract values obtained in the previous step.

In the course of performing step 3, all type-correct program constructs are removed from the result. What remains is a list containing only the abstract values of *incorrect* program constructs. In an additional phase, we generate human-readable error-messages from these abstract values. For example, we consider the type-checking of the following program:

```
PROGRAM error;
DECLARE
  i : INTEGER;
  j : INTEGER;
BEGIN
  i := 0;
  j := 1.5
END.
```

Replacing the identifiers and constants by their abstract values results in:

```
PROGRAM error;
DECLARE
BEGIN
  INTEGER := INTEGER;
  INTEGER := REAL
END.
```

After removing the type-correct construct `INTEGER := INTEGER`, the type-incorrect construct `INTEGER := REAL` remains. From this, an error-message “assignment-incompatible `INTEGER := REAL`” is generated.

Appendix C contains a detailed description of the CLaX type-checker; the corresponding specification can be found in Appendix D. In Section 4 we discuss how, from this specification, an error reporter is derived. This CLaX error reporter is a tool for automatically determining the positions of errors in the source text.

The CLaX interpreter uses the well-known concept of a stack of activation records [ASU86]. This stack contains one record for every procedure that is being executed. Each record contains the code of that procedure, a ‘pointer’ to the current statement, and a set of values defined in the procedure. Executing a program is specified by way of a recursive evaluation function which modifies the stack of activation records. The result of executing a program is a list of variable-value pairs for each global variable in the program. For instance, the result of executing the program of Figure 1 is shown below. Observe that an array value consists of a triple containing the upper bound, the lower bound, and the list of values of the components.

```
count :    21
fib :      [ 1, 20, 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 ]
```

The CLaX interpreter specification is not a purely algebraic one, because hybrid functions [Wal91] are used to perform basic arithmetic operations and I/O in Lisp.

The specification is discussed in Appendix E; the text of the specification appears in Appendix F.

In Section 5 we discuss how, from the interpreter specification, an animator is derived, permitting us to visualize program execution.

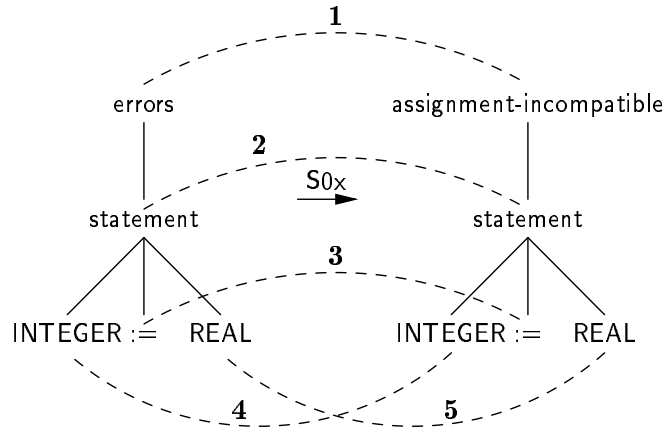


Figure 2: Example of single-step origin relations. Dashed lines indicate related subterms. The relation labeled **1** is the relation between the entire redex and the entire contractum. Relations **2** and **3** are caused by the common subterms $_SimpleType := _SimpleType'$ and $:=$, respectively. Moreover, relations **4** and **5** are the result of the presence of the common variables $_SimpleType$ and $_SimpleType'$, respectively.

3 Origin tracking

In the ASF+SDF Meta-environment, a specification can be executed as a (conditional) term rewriting system (TRS) [Klo91]. In a TRS, an *initial* term is transformed by repeatedly applying *rewrite-rules* to subterms. The application of a rewrite rule corresponds to replacing a subterm (called a *redex*, for *reducible expression*) by another subterm (called *contractum*). A term rewriting process terminates when no more rules are applicable; the result is referred to as a *normal form*.

In our setting, the initial term consists of an abstract syntax tree representing a CLaX program, to which a function `tc` or `eval` is applied. The term rewriting system is obtained from the specification by orienting the equations of the specification from left to right. In the case of the type-checker, the normal form is a term which represents a list of error-messages. In the case of the interpreter, the normal form is a list containing a variable-value pair for each global variable in the program.

Origin tracking is a generic technique for relating parts of intermediate terms, which occur during term rewriting, to parts of the initial term. The origin relation defines, given a subterm of an intermediate term, a set of related subterms in the initial term. This relation is defined as the transitive and reflexive closure of *single-step* origin relations. These single-step origin relations relate subterms for one-step reductions. Four kinds of single-step origin relations can be distinguished:

- The redex subterm is related to the contractum subterm.
- Subterms in the context of the redex are related to their counterparts in the context of the contractum.
- A common variable of the left-hand side and the right-hand side of the rewrite rule gives rise to relations between the subterms of its instantiations in the redex and the contractum.
- A common subterm of the left-hand side and the right-hand side of the rewrite rule gives rise to a relation between its instantiations in the redex and the contractum.

Figure 2 above shows the single-step origin relations established as a result of applying the rule

$$[S0x] \text{ errors}(_SimpleType := _SimpleType') = \\ \text{assignment-incompatible } _SimpleType := _SimpleType'$$

to a term `errors(INTEGER := REAL)`, resulting in a term `assignment-incompatible INTEGER := REAL`. For convenience, terms are depicted as trees in the figure.

For a formal definition of the origin function, we refer the reader to [DKT92]. We conclude this brief overview with a list of general principles and properties of origin tracking:

- Origin tracking does not depend on the confluence, and on the termination behavior of the TRS.
- No changes to the TRS are required; single-step origin relations are derived from the TRS automatically.

- Origin relations satisfy the following property: if a subterm t of the initial term is included in the origin of a term t' , then t rewrites to t' in zero or more rewrite-steps.
- [DKT92] describes sufficient conditions a TRS should satisfy in order to guarantee that all terms of a given sort have non-empty origins. These conditions can be verified by a static check of the TRS. For example, in case of the CLaX interpreter specification, subterms of sort STAT will always have non-empty origins.

4 Error Reporters

In Appendix C, an effective type-checker for CLaX is described. The result of type-checking a CLaX program consists of a list of error-messages in a human-readable form. These messages, albeit useful, provide no information regarding the specific language constructs that caused the errors, or the positions from which the errors originated. In the case that a large program contains a type-error, such information clearly is necessary.

An *error reporter* is a tool for determining the source position of type-errors. This task is commonly achieved by keeping track of line numbers at the specification (or implementation) level. An obvious disadvantage of this method is that the specification becomes less clear and concise. Moreover, it would involve a lot of extra work on specification/implementation of the type-checker. By contrast, our approach does not require any changes to the specification. The key issue is that positions of errors are automatically maintained by the origin tracking mechanism.

4.1 Method

Type-checking a program consists of rewriting a term consisting of the function tc applied to its abstract syntax tree. The result of this rewriting process consists of a term representing a list of error-messages. The origin function relates subterms of the latter term to subterms of the former term: parts of the generated error-messages are related to parts of the program's abstract syntax tree. This situation is illustrated in Figure 3.

In order to implement an error reporter for CLaX, the error-list which results from the type-checking of a program is put in an instance the ASF+SDF system's generic syntax-directed editor, GSE [Koo92]. GSE is a hybrid editor: it allows one to do text-editing as well as structure-editing. When in structure-editing 'mode', a *focus* indicates the current structural selection. In order to implement an error reporter, the user-interface of this editor is extended with a button labeled **Show Origin**. Pressing this button results in:

- Retrieval of the origins in the subtree corresponding to the focus.
- Highlighting the corresponding subterms of the abstract syntax tree of the program.

As an example, we consider the type-checking of the following program:

```
PROGRAM errors;
DECLARE
  i : INTEGER;

  PROCEDURE incr(VAR r : REAL);
  BEGIN (* incr *)
```

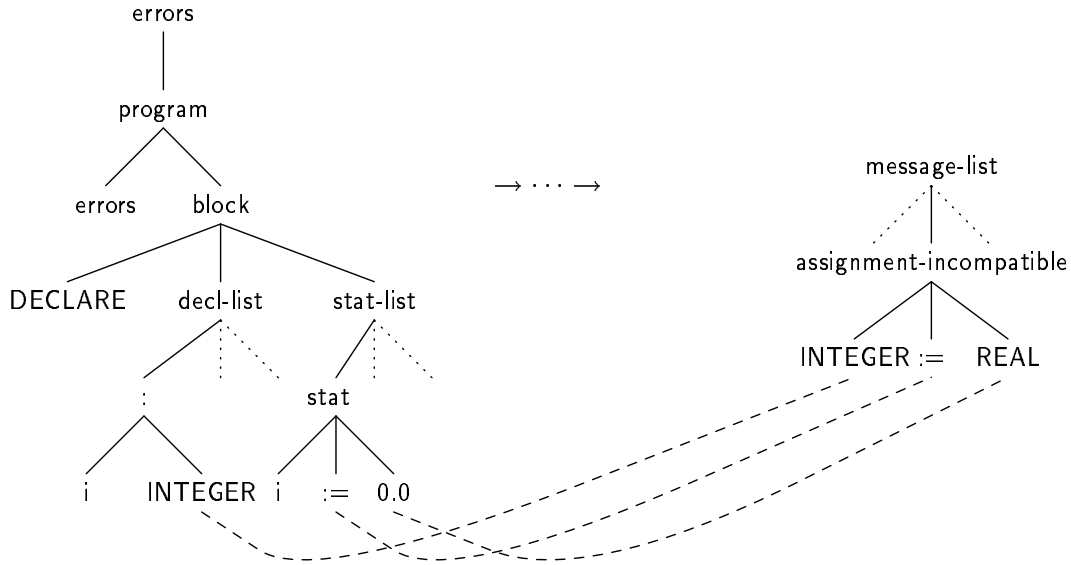


Figure 3: Origin relations between subterms of the abstract syntax tree, and subterms of the list of error-messages computed by the type-checker. The abstract syntax tree is shown partially, and only the origin relations involved in the error-message assignment-incompatible INTEGER := REAL are shown.

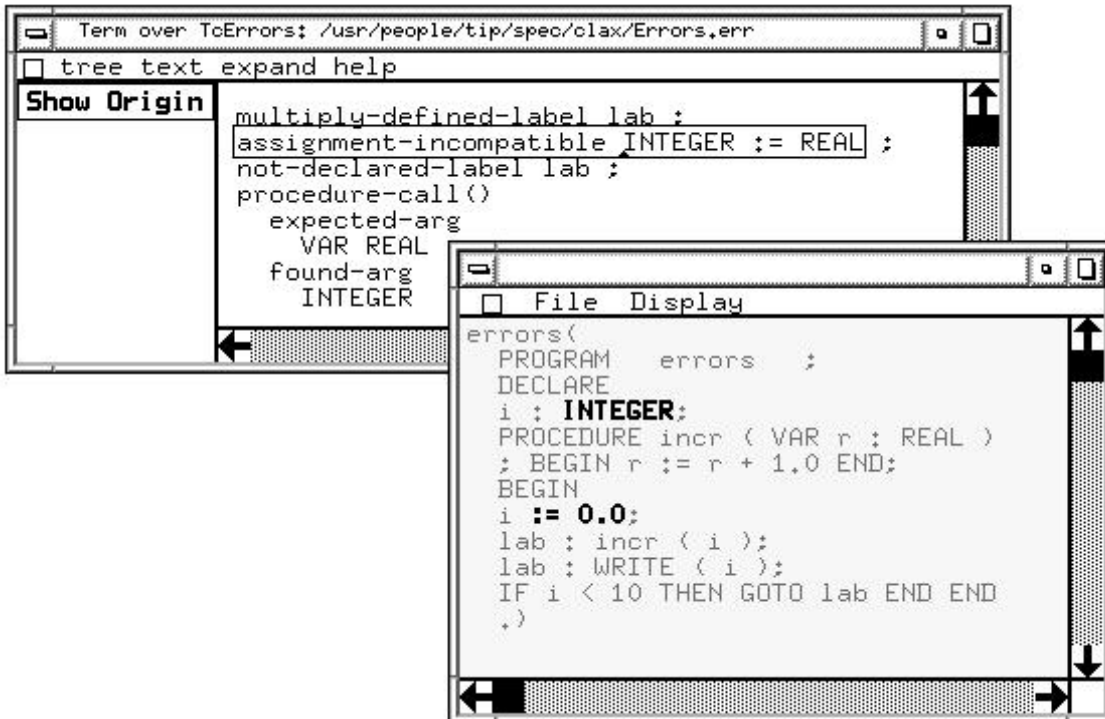


Figure 4: Snapshot of the generated error reporter for CLaX.

```

    r := r + 1.0
  END; (* incr *)

BEGIN (* errors *)
  i := 0.0;
  lab: incr(i);
  lab: WRITE(i);
  IF i < 10 THEN
    GOTO lab
  END
END. (* errors *)

```

Figure 4 above shows a snapshot of the generated error reporter for CLaX. One window contains the text of the program (with the function errors applied to it); the other window shows the list of error-messages produced by the CLaX type-checker. The error-message assignment-incompatible INTEGER := REAL has been selected, and the corresponding pieces of program text are highlighted. In this case, the indicated pieces of program text are:

- The := symbol of the statement containing the error (which is related to the := symbol in the error-message).
- The real constant 0.0 from which the abstract value REAL in the error-message originates.
- The type, INTEGER, of the variable used in the erroneous statement.

In Appendix C.8, it is argued how—in order to keep the amount of information manageable—multiple occurrences of the same error are ‘merged’. This merging of errors is performed by applying equation [2] of module TcErrors (shown below). According to the definition of the origin function, the binding of variable *_Msg* in the contractum is related to *both* messages in the redex that were involved in the merge. Hence, in the course of merging error-messages, no positional information is lost.

$$[2] \text{_MsgList ;_Msg ;_MsgList}' ;_Msg ;_MsgList'' = \text{_MsgList ;_Msg ;_MsgList}' ;_MsgList''$$

4.2 Limitations

The limitations of this method for generating error reporters are results of the restrictions imposed by the underlying origin function. Essentially, these restrictions consist of the fact that only ‘equal’ terms are related by the origin function. This type of tracing relationship appears to be particularly suitable for our method of reducing program constructs to their abstract values. Put more precisely, the existence of many direct relations between subterms common to the generated error-message and of the program’s abstract syntax tree enables the generation of effective error-reporters in this manner.

Conversely, this means that the success of our method is highly dependent on the degree in which such direct links are established. In more conventional methods of type-checking, where error-messages are collected during type-checking rather than being derived from erroneous type-constructs (see for example [Deu91]), the effectiveness of the origin relation

decreases. A solution to this problem could consist of a ‘stronger’ origin relation which would also take indirect dependencies between subterms into account. In Section 7, we outline several possibilities for such extensions of the origin relation.

It should be mentioned here that the syntax definition of CLaX is written such that a maximal number of relationships between parts of error-messages and program constructs could be achieved. In particular, we use separate tokens for all operators and language constructs of CLaX (as defined in module `SyntaxTokens`, see Appendix B.2). Unfortunately, this is somewhat in conflict with the demands of structure-editors, where a larger number of tokens leads to greater number of editing actions. We conclude that this situation is undesirable, and calls for an extension of the origin relation. In Section 7 we outline some generalizations of the origin function which would obviate changes to the specification.

5 Animators

An *animator* is a tool for the visualization of program execution. Typical applications of animators can mainly be found in the areas of tutoring and debugging. Visualization may be performed in several ways. Conventional tracing/debugging tools often provide the user with the line number of the statement that is currently being evaluated. More advanced tools highlight the current line or statement. In this paper, we assume that animation is done by highlighting pieces of program text, such as statements, declarations, etc.

Below, we describe a very flexible mechanism for defining animators, consisting of two phases:

- Definition of the *events* we are interested in. A typical example of such an event is the execution of a statement.
- Determination of the *subjects*, i.e., the parts of the program involved, for each of these events.

We define events by way of a simple but powerful pattern-matching mechanism. Origin tracking is used for determination of the subjects of events.

To illustrate our approach, we define several animation features for CLaX, based on the CLaX interpreter specification of Appendix E. We start by defining some events which occur during the execution of a CLaX program. Then, we describe how the subjects involved in these events can be found.

5.1 Definition of events

In Appendix E.3, we describe a function `eval` which performs a single evaluation step, by transforming the status of the interpreter. Evaluation steps consist of: (i) execution of a statement, (ii) return from a procedure call, or (iii) termination of the program. Assuming for the moment that each application of `eval` corresponds to the execution of a statement, we may state: a statement is being executed when the current redex *matches* the pattern (i.e., open term)

$$\text{eval}(_Status)$$

Here, `_Status` is a variable of sort `STATUS`. We will distinguish between the execution of a statement and other applications of `eval` shortly.

In a similar way, it can be argued that a predicate of an IF statement or a WHILE statement is being executed when the current redex matches the pattern

$$\text{eval-predicate}(_Expr, _DStack)$$

with $_Expr$ and $_DStack$ variables of sorts EXPR and D-STACK, respectively¹ (see Appendix F.4).

In fact, function `eval-exp` could be used for the evaluation of predicates. We introduced an extra function `eval-predicate` to be able to distinguish between the evaluation of predicates and the evaluation of other CLaX expressions.

The processing of declarations and parameters corresponds to an application of the functions `init-decls` and `init-params`, respectively (Appendix F.5). Both these functions operate on *lists* (of declarations and statements). The first element of a non-empty list corresponds to the declaration or parameter being processed. Therefore, the following patterns may be used to intercept events:

$$\begin{aligned} & \text{init-decls}(_Decl+, _Path) \\ & \text{init-params}(_Actual+, _Formal+, _DStack) \end{aligned}$$

Returning to the `eval` function, we observe that the execution of statements can be distinguished from other applications of `eval` by specializing the pattern (i.e., making the pattern more specific). In particular, we propose:

- An application of `eval` corresponds to a return from a procedure if the current stack frame (C-FRAME) contains no more code to be executed, and there is more than one C-FRAME on the stack. This event exactly corresponds to a match with the following pattern:

$$\text{eval}(\langle _Id, _Stat*,] _CRec+, _DStack \rangle)$$

- An application of `eval` corresponds to the termination of the program if the current C-FRAME contains no more code to be executed, and there is exactly one C-FRAME on the stack. The pattern which describes this event is:

$$\text{eval}(\langle _Id, _Stat*,], _DStack \rangle)$$

- An application of `eval` corresponds to the execution of a statement if the current C-FRAME contains at least one more statement which is to be executed. This event is described by the pattern:

$$\text{eval}(\langle _Id, _Stat*, _Stat+] _CRec*, _DStack \rangle)$$

Naturally, the patterns described above can be specialized even further, thus allowing different animation behavior for different statements. Table 1 below summarizes some events together with the corresponding patterns.

¹All variables used in subsequent patterns occur exactly as they are defined in the CLaX interpreter specification. Therefore we refrain from mentioning their sorts here.

Event	Pattern
execution of a statement	eval(<[_Id, _Stat*, _Stat+] _CRec*, _DStack>)
return from a procedure	eval(<[_Id, _Stat*,] _CRec+, _DStack>)
evaluation of a predicate	eval-predicate(_Expr, _DStack)
processing of a declaration	init-decls(_Decl+, _Path)
processing of a parameter	init-params(_Actual+, _Formal+, _DStack)

Table 1: Events during the execution of CLaX programs, and corresponding patterns.

Path(s)	Pattern and subterm(s) indicated by paths	Subject(s)
(1 1 1 3 1)	eval(<[_Id, _Stat*, _Stat];_Stat*'] _CRec*, _DStack>)	the statement
(1 1 2 3 1)	eval(<[_Id, _Stat*,] [_Id', _Stat*', _Stat];_Stat*''] _CRec*, _DStack>)	the procedure call
(1)	eval-predicate(_Expr , _DStack)	the predicate
(1 1)	init-decls(_Decl];_Decl*, _Path)	the declaration
(1 1), (2 1)	init-params(_Actual];_Actual*, _Formal];_Formal*, _DStack)	the actual and the formal parameter

Table 2: Subjects of events. The first column contains paths to parts of the pattern that correspond to subjects. The second column contains the patterns for event interception; the subterms indicated by paths are shown in a box. The third column contains the subjects which are obtained by retrieving the origins of the subterms of the redex matched against the ‘boxed’ subterms of the pattern.

5.2 Determining subjects

Having defined patterns for the interception of events, we now describe how the subject of an event can be determined. To this end, we use a concept of *paths* (a similar notion is used in Appendix E.1). Paths are used to indicate subterms of patterns, by interpreting the numbers in a path as argument positions of function symbols. For example, path (1) indicates the subterm `_Expr` of pattern `eval-predicate(_Expr, _DStack)`. Intuitively, this subterm ‘corresponds’ to the predicate that is being evaluated (and we want to be highlighted). This correspondence is formalized by the origin relation described earlier. The origin of the subterm at path (1) in the redex (i.e., the *origin* of the subterm matching `_Expr`) indicates the subject we are interested in: the predicate which is being evaluated.

Table 2 summarizes the paths to the subjects for each of the patterns shown in Table 1. We have ‘expanded’ the patterns here by replacing variables by their structural expansions. This permits us to indicate the subterms corresponding to the specified paths. Note that these expansions are of a non-essential nature—the patterns in Table 2 have exactly the same matching behavior as the corresponding patterns in Table 1.

5.3 Example

As an example, we use the patterns and paths described above to animate the execution of the following program:

```

PROGRAM example;
DECLARE
  i: INTEGER;
BEGIN
  i := 0;
  WHILE i < 5 DO
    i := i + 1
  END
END.

```

Executing a program corresponds to reducing a term consisting of the function `eval-program` applied to its abstract syntax tree. After starting the rewriting process, we first find a match with pattern `init-decls(_Decl;_Decl*, _Path)`; at this moment the redex consists of the term: `init-decls(i:INTEGER;, 1)`. We retrieve the origins in the subterm `i:INTEGER` at path (1 1) in the redex. Figure 5 (a) shows a window of the ASF+SDF system where the corresponding subterms of the initial term (here: the declaration of `i`) are highlighted.

Continuing the execution, a match with pattern `eval(<[_Id, _Stat*, _Stat;_Stat*'] _CRec*, _DStack>)` is encountered next; at this point the redex is: `eval(<[example, i:=0; WHILE i < 5 DO i:=i+1 END, i:=0; WHILE i < 5 DO i:=i+1 END], [example, , i:0] >)`. Figure 5 (b) shows the subject, which is found by retrieving the origins of the subterm `i:=0` at path (1 1 1 3 1) in the redex.

The next event corresponds to the execution of another statement (the `WHILE` statement), resulting in the entire statement being highlighted as shown in Figure 5 (c)

Subsequently, a match with pattern `eval-predicate(_Expr, _DStack)` is found, corresponding to the evaluation of the predicate. At this point, the redex consists of the term `eval-predicate(i < 5, [example, , i : 0])`. The origins of the subterm `i < 5` at path (1) in the redex are shown highlighted in Figure 5 (d).

The following animation steps consist of consecutively highlighting (i) the statement in the body of the `WHILE` construct (Figure 5 (e)), (ii) the entire `WHILE` construct (Figure 5 (f)), and (iii) the predicate of the `WHILE` (Figure 5 (d)) construct until termination of the program.

5.4 Limitations

Unfortunately, the usage of origin tracking for retrieving subjects imposes a restriction on the class of language constructs for which we can do animation. In essence, this restriction reflects the fact that we can only do visualization for language constructs which are guaranteed to have non-empty origins. For example, if one wishes to highlight statements, it is required that every subterm of sort `STAT`, which occurs in the course of the rewriting process, has a non-empty origin. Some remarks are in order here:

- It can be checked if a specification satisfies the property that all subterms of a given sort always have non-empty origins. This (simple) check is described in [DKT92].
- In the case of CLaX, it is guaranteed that empty origins will not occur for any of the patterns and paths we described.
- Extensions of the origin relation are envisaged which would overcome this problem. We will discuss this in Section 7.


```
eval-program(  
PROGRAM example ;  
DECLARE i : INTEGER ;  
BEGIN  
i := 0;  
WHILE i < 5 DO  
i := i + 1 END END  
,.)
```

(a)

```
eval-program(  
PROGRAM example ;  
DECLARE i : INTEGER ;  
BEGIN  
i := 0;  
WHILE i < 5 DO  
i := i + 1 END END  
,.)
```

(b)

```
eval-program(  
PROGRAM example ;  
DECLARE i : INTEGER ;  
BEGIN  
i := 0;  
WHILE i < 5 DO  
i := i + 1 END END  
,.)
```

(c)

```
eval-program(  
PROGRAM example ;  
DECLARE i : INTEGER ;  
BEGIN  
i := 0;  
WHILE i < 5 DO  
i := i + 1  
END END  
,.)
```

(d)

```
eval-program(  
PROGRAM example ;  
DECLARE i : INTEGER ;  
BEGIN  
i := 0;  
WHILE i < 5 DO  
i := i + 1  
END END  
,.)
```

(e)

```
eval-program(  
PROGRAM example ;  
DECLARE i : INTEGER ;  
BEGIN  
i := 0;  
WHILE i < 5 DO  
i := i + 1  
END END  
,.)
```

(f)

Figure 5: Animation of Execution in the ASF+SDF system.

When stated in an informal way, the check mentioned above verifies that the execution of a language construct is either defined in terms of the execution of its sub-constructs, or (recursively) as the execution of the same construct in a modified environment.

As an example of a violation of this rule, consider the equation below which expresses the execution of a REPEAT construct in terms of the execution of a WHILE construct.

$$\begin{aligned} \text{eval}(\langle [_Id, _Seq, \text{REPEAT } _Stat* \text{ UNTIL } _Exp \text{ END}; _Stat*'] _CRec*, _DStack \rangle) = \\ \text{eval}(\langle [_Id, _Seq, _Stat*; \text{WHILE NOT}(_Exp) \text{ DO } _Stat* \text{ END}; _Stat*'] _CRec*, _DStack \rangle) \end{aligned}$$

When using this rule, the animation of execution of REPEAT statements will fail because of the fact that the ‘generated’ WHILE statements will have empty origins. In this example, the problem can be solved by defining the execution of REPEAT in terms of itself.

6 Related Work

The CLaX type-checker specification described in Appendix C is written in a non-standard abstract interpretation style. A similar style of type-checking is discussed in [Hee92], using higher-order algebraic specifications. The question whether adequate human-readable error-messages can be derived automatically was posed in [Deu91]. An attempt to derive meaningful error-messages in an automatic fashion is described in [Bra92].

Few generic approaches for defining animators and error reporters exist, to our knowledge. Many interpreters use ad-hoc methods for dealing with animation and error reporting, mostly consisting of keeping track of line numbers. Moreover, these systems only give support for one specific programming language.

In the context of the PSG system [BS86], a generator for language-specific debuggers was described in [BMS87]. These debuggers are generated from a specification of the denotational semantics of a language, and some additional debugging functions. A set of built-in debugging concepts is provided to this end.

PSG generates language-specific compilers by compiling denotational semantics definitions to a functional language. A standard, language-independent interpreter is used to execute the generated functional language fragments. During the compilation of program fragments, correspondences between the abstract syntax tree and terms of the functional language are maintained. Built-in trace functions are used to visualize execution. Besides trace functions, other built-in concepts allow the inspection of the state of the interpreter, the definition of breakpoints, and so on.

The debugging/animation facilities present in the PSG system are powerful, but the specifications used to define these facilities are difficult to read. Moreover, there seems to be little flexibility for defining animation features.

In the CENTAUR system [BCD⁺89], the specification language Typol [Des88] is implemented, an implementation of natural semantics [Kah87]. Typol specifications can be used to define type-checkers, interpreters, compilers, and so on. A Typol specification consists of a set of axioms and inference rules which are compiled to Prolog. A key property of Typol specifications is that the meaning of a language construct is expressed in terms of the meanings of its sub-constructs.

Bertot [Ber91c, Ber91b] presents a formal framework for residuals and origin functions in left-linear unconditional term rewriting systems and the λ -calculus. He contributes a technique called *subject tracking* to Typol, for relating execution to locations in a program.

To this end, a special variable, *Subject*, indicates the language construct currently being processed. This variable may be used to define animation and debugging features, based on a specification of an interpreter. The main contrast with our approach is that origin tracking establishes mappings for all language constructs remaining in normal forms or intermediate terms, whereas in Typol only the construct currently being processed is tracked. To some extent, this problem can be circumvented by explicitly manipulating the established origin relations. Apart from the fact that this solution essentially means leaving the domain of automatically established tracing relations, it is unclear if this solution is sufficiently flexible to define useful error reporters.

Berry [Ber91a] presents a theory of animators. Animators are generated from formal specifications; these specifications are written in structured operational semantics. Views of the state of a program during execution are defined by extending existing specifications with semantic display rules. He distinguishes between *static* and *dynamic* views of a program. A static view consists of parts of the abstract syntax tree of a program, and a dynamic view is constructed from the program state during execution. As an example of a dynamic view, the evaluation of a predicate can be visualized as the actual truth value it obtains during execution. Another criterion he uses to classify animators is that of *source* views versus *environment* or *memory* views. Source views show parts of the source text of the program, and memory views display the current contents of memory locations. All animation features we describe in this paper can be classified as static views and source views.

7 Concluding Remarks

We have presented a uniform framework for incorporating animators and error reporters in generated programming environments. Existing specifications for type-checkers and interpreters are re-used. We show that origin tracking and a simple pattern-matching mechanism are sufficiently powerful to obtain animators and error reporters from these specifications.

Our experience with CLaX suggests that animators and error reporters can be defined for a wide range of imperative programming languages in a similar manner. We expect no fundamental problems with the definition of these tools for realistic languages such as Pascal and C. Moreover, we conjecture that our technique is also suitable for programming languages with parallel, or object-oriented features. It is our intention to verify this conjecture in the near future.

In Sections 4.2 and 5.4, we mentioned limitations of origin tracking. These limitations are caused by the fact that only ‘equal’ subterms are related. In principle, weaker concepts than ‘equality’ could be used as a basis for an origin relation. In particular, we envisage the following generalizations:

1. Subterms with the same function symbol which occur in rewrite-rules could be used as a criterion for establishing origin relations.
2. Subterms of the same sort which occur in rewrite-rules could be used as a criterion for establishing origin relations.
3. A notion of dependence could be used instead of a tracing relation such as origin tracking. Associated with every subterm occurring during the rewriting process

would be a minimal subset of the initial term which is responsible for the occurrence of that subterm.

A problem with the first two options is that there may still remain subterms which do not have origins. We are currently investigating the third option, which does not suffer from this problem. A potential pitfall of all three options is that they may give rise to less precise information. A multi-level approach may be adopted to circumvent the problem of having too vague information. We could use the current origin tracking information in the cases where it is available; in the remaining cases, we would then use the information computed by the generalized notion.

Apart from investigating extended notions of the origin function, we intend to study the following topics:

- Animators and error reporters for languages with parallel and object-oriented features.
- More advanced animation features. In particular, we are interested in extending our framework so as to be able to implement Berry's notion of environment and memory views [Ber91a].
- Extensions of the pattern-matching approach for the definition of generic, source-level debuggers. The expressive power of the patterns can be increased by allowing patterns to be *conditional*. In principle, this enables us to use arbitrary semantic constraints on the program status as breakpoints.

Acknowledgements

The authors would like to thank Arie van Deursen, Paul Klint, and Pum Walters for their comments on drafts of this paper.

References

- [ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [BCD⁺89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 14(2).
- [Ber91a] D. Berry. *Generating Program Animators from Programming Language Semantics*. PhD thesis, University of Edinburgh, 1991.
- [Ber91b] Y. Bertot. Occurrences in debugger specifications. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 327–337, 1991. Appeared as *SIGPLAN Notices* 26(6).
- [Ber91c] Y. Bertot. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle*. PhD thesis, INRIA, Sophia-Antipolis, 1991. In French.

- [BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.
- [BMS87] R. Bahlke, B. Moritz, and G. Snelting. A generator for language-specific debugging systems. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 92–101, 1987. Appeared as SIGPLAN Notices 22(7).
- [Bra92] M.G.J. van den Brand. *Pregmatic, A generator for incremental programming environments*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.
- [BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.
- [Con91] The COMPARE Consortium. Description of the COSY-prototype. Technical report, GMD, 1991. unpublished.
- [Des88] T. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, 1988.
- [Deu91] A. van Deursen. An algebraic specification for the static semantics of Pascal. Report CS-R9129, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991. Extended abstract in: *Conference Proceedings of Computing Science in the Netherlands CSN'91*, pages 150-164.
- [DKT92] A. van Deursen, P. Klint, and F. Tip. Origin tracking. Report CS-R9230, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.
- [GW84] Gerhard Goos and William M. Waite. *Compiler Construction*. Springer Verlag, Berlin, Heidelberg, New York, Tokyo, 1984.
- [Hee92] J. Heering. Second-order algebraic specification of static semantics. Report CS-R9254, Centrum voor Wiskunde en Informatica (CWI), 1992.
- [Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.
- [HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
- [Kah87] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
- [Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 1993. Also CWI report CS-R9064.

- [Klo91] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol II*. Oxford University Press, 1991. Also published as CWI report CS-R9073, Amsterdam, 1990.
- [Koo92] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 168–177. SION, 1992. Also appeared as Report P9202, University of Amsterdam.
- [LeL90] INRIA, Rocquencourt, France. *Le-Lisp Reference Manual*, version 15.23 edition, 1990.
- [Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

A The syntax of CLaX

In this section, we give an overview of the most significant features of CLaX. The modules involved can be found in Appendix B.

A CLaX program essentially consists of declarations and statements. Each variable occurring in a statement must be preceded by a declaration which associates a type with that variable. This type defines the set of values that may be assigned to that variable.

There are three basic types: *boolean*, *integer* and *real*. Array types are defined by describing the types of their components and an integer range (module `SyntaxTypes` in Appendix B.4).

A value is obtained by evaluating an expression. Expressions consist of variables, constants and operations on denoted quantities. CLaX defines a set of operators which can be subdivided into arithmetic, boolean, and relational operators. The corresponding sorts for each of these operators are `AOPx`, `BOPx` and `COP`, respectively (module `SyntaxExpr`, Appendix B.6).

Statements are described in module `SyntaxStats` (Appendix B.7). An assignment statement specifies a value to be assigned to a variable, a procedure statement causes the execution of the designated procedure, and a GOTO statement causes the execution to continue with the statement specified by a designated label. Sequential execution is specified by statement sequences, selective execution by IF statements and repeated execution by WHILE statements.

Module `SyntaxHeaders` describes variable, label and procedure declarations. Procedure declarations effectively identify a statement sequence, and optionally declarations, with a name. A procedure has a fixed number of parameters, each of which is denoted within the procedure by an identifier called *formal* parameter. The formal parameter can be of two kinds: *value* or *variable* parameters. An actual parameter, during a procedure activation, will be evaluated once for value parameters while in the case of a variable parameter, the actual parameter will be a variable that stands for the formal parameter. Possible array indices are evaluated before execution of a procedure.

A.1 Comments

ASF+SDF provides a special sort called LAYOUT, using which, text-to-be-discarded can be easily defined. This facility is used here to define *comment* in module `SyntaxLayout`.

module SyntaxLayout

exports

lexical syntax

```
[ \t\n]          → LAYOUT
“{” ~ [ ] * “}”  → LAYOUT
“(*” ~ [*] * “)” → LAYOUT
```

This module defines that:

- all white spaces, tabs and line-feeds are to be ignored,
- a { followed by any sequence of characters not containing } followed by } should be ignored, and

- (* followed by any sequence of characters not containing * followed by *) should be ignored².

A.2 Tokens

Module `SyntaxTokens` defines sorts and tokens which represent reserved words and special characters. This is not an exhaustive list, but should be thought of as a list of tokens for which origin tracking is desired. The role of this module is elaborated on in Section 4.2.

A.3 Constants

The module `SyntaxConsts` defines the basic constants supported by CLaX. These constants can be broadly identified as identifiers (ID), Booleans (BOOL-CONST), integers (INT-CONST), reals (REAL-CONST) and strings (STRING). For example

$$[a-zA-Z][A-Za-z0-9]^* \rightarrow \text{ID}$$

defines a letter followed by a letter or digit to be an identifier. Similar definitions occur in module `SyntaxConsts` for numeric constants. E.g., the following are all well-defined constants: 1, 100, .1, and 87.35E-8. Likewise,

$$["] \sim ["\backslash n]^* ["] \rightarrow \text{STRING}$$

defines a character string as a sequence of characters enclosed in " which do not contain either a " or a line feed character in between³. These strings are used only to output messages (in `WRITE` statements).

A.4 Data Types

Module `SyntaxTypes` specifies the set of types in CLaX. The following *simple* types are defined:

```
"INTEGER"  → SIMPLE-TYPE
"REAL"     → SIMPLE-TYPE
"BOOLEAN"  → SIMPLE-TYPE
```

The values of type `INTEGER` are of sort `INT-CONST`, those of type `REAL` are of sort `REAL-CONST`, and those of type `BOOLEAN` are of sort `BOOL-CONST`.

An array type is a structure consisting of a fixed number of components which are all of the same type, called the *component type*. The elements of the array are designated by integer *indices*. The array type specifies the component type as well as a subrange of the integers to be used as indices. The following are valid arrays:

```
ARRAY [1..100] OF INTEGER
ARRAY [4..7] OF ARRAY [2..2] OF BOOLEAN
```

²[Con91] defines (* followed by any sequence of characters not containing * followed by *) should be ignored.

³[Con91] defines that strings could contain " characters as long as they are escaped by \.

A.5 Declarations

A declaration introduces an identifier for a label, variable or procedure⁴. Module `SyntaxHeaders` defines these declarations. Label declarations consist of an identifier denoting the new label, followed by the keyword `LABEL`. Variable declarations consist of an identifier denoting the new variable, followed by its type. Examples:

```
Error : LABEL
i: INTEGER
r: REAL
b: BOOLEAN
a: ARRAY [4..7] OF ARRAY [2..2] OF INTEGER
```

Procedure declarations serve to define parts of programs and to associate identifiers with them so that they can be activated by procedure call statements (module `SyntaxHeaders`):

```
PROC-HEAD “;” BLOCK           → PROC-DECL
VAR-DECL                       → FORMAL
VAR-LEX VAR-DECL              → FORMAL
PROCEDURE ID                   → PROC-HEAD
PROCEDURE ID LPAR {FORMAL “;”}+ RPAR → PROC-HEAD
```

The procedure heading (PROC-HEAD) specifies the identifier naming the procedure and the formal parameters (if any). The parameters are either value or variable parameters.

If a formal starts with the delimiter `VAR` it specifies a variable parameter, otherwise a value parameter. The use of the procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure. An example of a CLaX procedure is shown below.

```
PROCEDURE ReadPosInteger (VAR i: INTEGER);
DECLARE
  j: REAL;
BEGIN
  j := 0;
  WHILE NOT (0 < j) DO READ (j) END;
  i := j;
END
```

A.6 Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators.

The rules of composition specify operator *precedences* according to four classes of operators. The operator `NOT` and unary minus have the highest precedence, followed by

⁴We also allow empty declarations. Declaration lists consist of zero or more declarations separated by semi-colons. By contrast, [Con91] considers `;` to be a terminator in declaration lists.

the operators `*`, `/`, `%`, and `&`; then the operators `+`, `-`, and `|`; and finally, with the lowest precedence, the relational operators `<`, `<=`, `=`, `#`, `>=`, and `>`. Moreover, sequences of non-relational binary operators of the same precedence are interpreted as being left-associative.

Module `SyntaxExpr` defines these operators in different sorts and then defines priority rules over these operations, as summarized in Table 3. For example:

```

“*”           → AOP1
“+”           → AOP2
EXPR AOP1 EXPR → EXPR {left}
EXPR AOP2 EXPR → EXPR {left}

```

followed by the priority definition:

priorities

```

{left: EXPR AOP2 EXPR → EXPR}
<
{left: EXPR AOP1 EXPR → EXPR}

```

will parse `j + 2 * i + r` as `(j + (2 * i)) + r`.

Priority	Operator	left Operand	right Operand	Result	Operation
4	NOT		BOOLEAN	BOOLEAN	negation
	-		INTEGER	INTEGER	unary minus
			REAL	REAL	unary minus
3	*	INTEGER	INTEGER	INTEGER	integer multiplication
		REAL	REAL	REAL	real multiplication
	/	INTEGER	INTEGER	INTEGER	integer division
		REAL	REAL	REAL	real division
	%	INTEGER	INTEGER	INTEGER	integer remainder
&	BOOLEAN	BOOLEAN	BOOLEAN	boolean and	
2	+	INTEGER	INTEGER	INTEGER	integer addition
		REAL	REAL	REAL	real addition
	-	INTEGER	INTEGER	INTEGER	integer subtraction
		REAL	REAL	REAL	real subtraction
	BOOLEAN	BOOLEAN	BOOLEAN	boolean or	
1	< <=	INTEGER	INTEGER	BOOLEAN	integer comparison
	= #	REAL	REAL	BOOLEAN	real comparison
	> >=	BOOLEAN	BOOLEAN	BOOLEAN	boolean comparison

Table 3: Table of Operators

A.7 Statements

Statements denote algorithmic actions, and are said to be executable. Module `SyntaxStats` defines the statements of `CLaX`.

The empty statement executes no action. A statement may be labeled. The optional label is defined by using a auxiliary sort STAT-AUX and an implicit function from STAT-AUX to STAT:

```
STAT-AUX          → STAT
LABEL “:” STAT-AUX → STAT
```

The assignment statement serves to replace the current value of a variable by a new value defined by an expression. A procedure statement serves to execute the procedure denoted by the procedure identifier. The conditional statement specifies that a statement sequence is to be executed only if a certain condition (Boolean expression) is TRUE. If it is FALSE, the sequence following the (optional) delimiter ELSE is to be executed. The loop statement specifies that a certain statement is to be executed repeatedly, as long as the condition holds. If it evaluates to FALSE at the beginning, the statement is not executed at all. The GOTO statement passes the control flow to the statement, which is labeled with that identifier. It is forbidden to jump to a statement outside of the same block (i.e. procedure) or to jump into another block (i.e. procedure). Input and output of values of simple types is achieved by the standard procedures READ and WRITE.

A statement is executed when the control flow reaches it. The control flow starts with the first statement (textually), of a statement sequence. All statements except GOTO's and procedure calls pass the control flow after its execution to the next statement (textually) in the statement sequence. An example statement sequence (STAT-SEQ) is:

```
WRITE ( " read integers and write until a non-positive \n");
WRITE ( " number is read \n");
READ (i);
WHILE 0 < i DO
  WRITE (i); READ (i)
END ;
IF i < 0 THEN i := 1 ELSE i := 2 END ;
next ;
Transpose(a,m,n) ;
; (* empty statement *)
```

A.8 Blocks and Programs

Module `SyntaxProgram` defines BLOCKS and PROGRAMS. A BLOCK contains an optional declaration part. If present, this starts with the reserved keyword DECLARE, followed by a list of zero or more declarations. Furthermore, each block consists of a list of statements between delimiters BEGIN and END.

A CLaX program has the form of a procedure declaration except for its heading. The identifier following the symbol PROGRAM is the program name; it has no further significance inside the program.

A.9 The generated CLaX environment

A snapshot of an editor containing a CLaX program is shown in Figure 6. The syntax definition in Appendix B is used to derive the structured editor tuned for CLaX. The

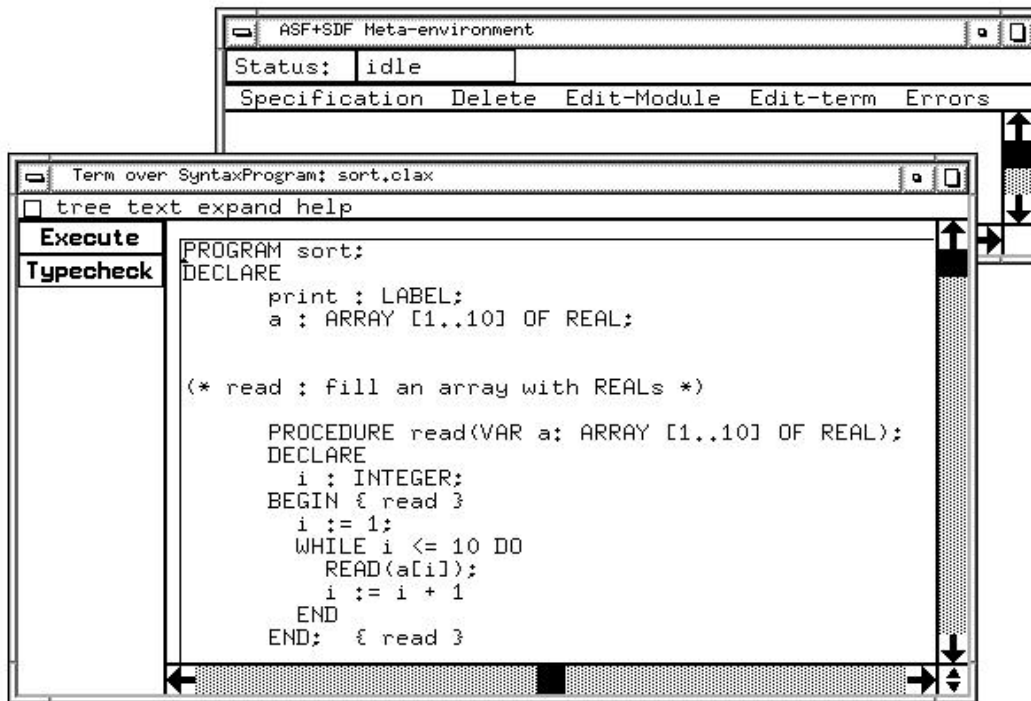
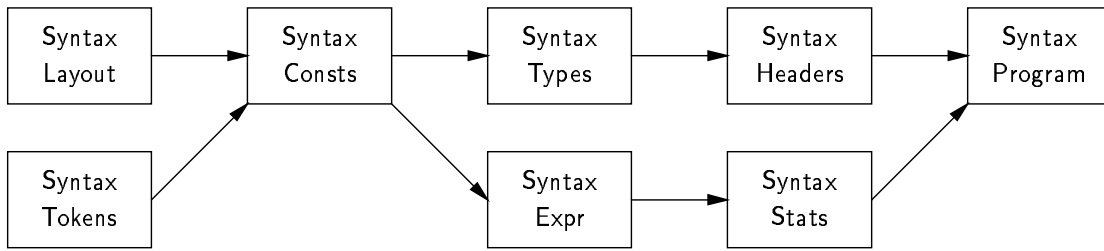


Figure 6: A snapshot of the generated CLaX environment.

buttons **Typecheck** and **Execute** respectively invoke the type-checker (described in Appendix C) and the interpreter (described in Appendix E) for the program in the editor.

B CLaX syntax modules

The specification in this appendix defines the syntax of CLaX. The import diagram for the syntax modules is:



B.1 module SyntaxLayout

module SyntaxLayout

exports

lexical syntax

$[\backslash t \backslash n]$ → LAYOUT
“{” ~ [] * “}” → LAYOUT
“(” ~ [*] * “)” → LAYOUT

B.2 module SyntaxTokens

module SyntaxTokens

exports

sorts LABEL-LEX PROCEDURE VAR-LEX DECLARE BEGIN ASGN IF THEN ELSE
END WHILE OF DO GOTO READ WRITE LPAR RPAR LSQ RSQ

context-free functions

"LABEL"	→ LABEL-LEX
"PROCEDURE"	→ PROCEDURE
"VAR"	→ VAR-LEX
"OF"	→ OF
"("	→ LPAR
")"	→ RPAR
"["	→ LSQ
"]"	→ RSQ
"DECLARE"	→ DECLARE
"BEGIN"	→ BEGIN
":="	→ ASGN
"IF"	→ IF
"THEN"	→ THEN
"ELSE"	→ ELSE
"END"	→ END
"WHILE"	→ WHILE
"DO"	→ DO
"GOTO"	→ GOTO
"READ"	→ READ
"WRITE"	→ WRITE

B.3 module SyntaxConsts

module SyntaxConsts

imports SyntaxLayout SyntaxTokens

exports

sorts ID STRING BOOL-CONST INT-CONST REAL-CONST SIGN SCALE-FACTOR
UNS-INT-CONST UNS-REAL-CONST

lexical syntax

[a-zA-Z] [A-Za-z0-9]*	→ ID
"_"	→ SIGN
"+"	→ SIGN
UNS-INT-CONST	→ INT-CONST
SIGN UNS-INT-CONST	→ INT-CONST
"E" [+ -] UNS-INT-CONST	→ SCALE-FACTOR
UNS-REAL-CONST	→ REAL-CONST
SIGN UNS-REAL-CONST	→ REAL-CONST
"~" ~["\n"]* [""]	→ STRING
"TRUE"	→ BOOL-CONST
"FALSE"	→ BOOL-CONST
[0-9]+	→ UNS-INT-CONST
"." UNS-INT-CONST	→ UNS-REAL-CONST
"." UNS-INT-CONST SCALE-FACTOR	→ UNS-REAL-CONST
UNS-INT-CONST "." UNS-INT-CONST	→ UNS-REAL-CONST
UNS-INT-CONST "." UNS-INT-CONST SCALE-FACTOR	→ UNS-REAL-CONST

variables

[_] <i>Id</i> ['']*	→ ID
[_] <i>IntConst</i> ['']*	→ INT-CONST
[_] <i>BoolConst</i> ['']*	→ BOOL-CONST
[_] <i>RealConst</i> ['']*	→ REAL-CONST
[_] <i>String</i> ['']*	→ STRING

B.4 module SyntaxTypes

module SyntaxTypes

imports SyntaxConsts

exports

sorts TYPE SIMPLE-TYPE ARRAY-TYPE

context-free functions

"INTEGER" → SIMPLE-TYPE

"REAL" → SIMPLE-TYPE

"BOOLEAN" → SIMPLE-TYPE

"ARRAY" LSQ INT-CONST ".." INT-CONST RSQ OF TYPE → ARRAY-TYPE

SIMPLE-TYPE → TYPE

ARRAY-TYPE → TYPE

variables

$[-]$ *Type* $[']^*$ → TYPE

$[-]$ *SimpleType* $[']^*$ → SIMPLE-TYPE

$[-]$ *ArrayType* $[']^*$ → ARRAY-TYPE

B.5 module SyntaxHeaders

module SyntaxHeaders

imports SyntaxTypes

exports

sorts PROC-HEAD LABEL-DECL PROC-DECL VAR-DECL DECL DECL-LIST
FORMAL BLOCK

context-free functions

ID ":" LABEL-LEX → LABEL-DECL
ID ":" TYPE → VAR-DECL
PROC-HEAD ";" BLOCK → PROC-DECL
→ EMPTY-DECL

VAR-DECL → DECL
PROC-DECL → DECL
LABEL-DECL → DECL
EMPTY-DECL → DECL

{DECL ";" }* → DECL-LIST
VAR-DECL → FORMAL
VAR-LEX VAR-DECL → FORMAL
PROCEDURE ID → PROC-HEAD
PROCEDURE ID LPAR {FORMAL ";" }+ RPAR → PROC-HEAD

variables

[_] *Decl* "+" [0-9']* → {DECL ";" }+
[_] *Decl* "*" [0-9']* → {DECL ";" }*
[_] *LabelDecl* [0-9']* → LABEL-DECL
[_] *VarDecl* [0-9']* → VAR-DECL
[_] *ProcDecl* [0-9']* → PROC-DECL
[_] *ProcHead* [0-9']* → PROC-HEAD
[_] *Decl* [0-9']* → DECL
[_] *Block* [0-9']* → BLOCK
[_] *Formal* [0-9']* → FORMAL
[_] *Formal* "+" [0-9']* → {FORMAL ";" }+
[_] *DeclList* [0-9']* → DECL-LIST
[_] *EmptyDecl* [0-9']* → EMPTY-DECL

hiddens

sorts EMPTY-DECL

B.6 module SyntaxExpr

module SyntaxExpr

imports SyntaxConsts

exports

sorts UAOP AOP1 AOP2 UBOP BOP1 BOP2 COP VARIABLE EXPR

context-free functions

"_"	→ UAOP
"NOT"	→ UBOP
"*"	→ AOP1
"/"	→ AOP1
"%"	→ AOP1
"&"	→ BOP1
"+"	→ AOP2
"-"	→ AOP2
" "	→ BOP2
"<"	→ COP
"<="	→ COP
"="	→ COP
">="	→ COP
">"	→ COP
"#"	→ COP
"(" EXPR ")"	→ EXPR { bracket }
ID	→ VARIABLE
VARIABLE LSQ EXPR RSQ	→ VARIABLE
VARIABLE	→ EXPR
BOOL-CONST	→ EXPR
INT-CONST	→ EXPR
REAL-CONST	→ EXPR
EXPR AOP1 EXPR	→ EXPR { left }
EXPR AOP2 EXPR	→ EXPR { left }
EXPR BOP1 EXPR	→ EXPR { left }
EXPR BOP2 EXPR	→ EXPR { left }
EXPR COP EXPR	→ EXPR { left }
UAOP EXPR	→ EXPR
UBOP EXPR	→ EXPR

variables

$[-] \text{Aop1 } [']^* \rightarrow \text{AOP1}$
 $[-] \text{Aop2 } [']^* \rightarrow \text{AOP2}$
 $[-] \text{Bop1 } [']^* \rightarrow \text{BOP1}$
 $[-] \text{Bop2 } [']^* \rightarrow \text{BOP2}$
 $[-] \text{Cop } [']^* \rightarrow \text{COP}$
 $[-] \text{Uaop } [']^* \rightarrow \text{UAOP}$
 $[-] \text{Ubop } [']^* \rightarrow \text{UBOP}$
 $[-] \text{Var } [0-9']^* \rightarrow \text{VARIABLE}$
 $[-] \text{Expr } [0-9']^* \rightarrow \text{EXPR}$

priorities

$\{ \text{EXPR COP EXPR} \rightarrow \text{EXPR} \} <$
 $\{\text{left: EXPR AOP2 EXPR} \rightarrow \text{EXPR}, \text{ EXPR BOP2 EXPR} \rightarrow \text{EXPR}\}$
<
 $\{\text{left: EXPR AOP1 EXPR} \rightarrow \text{EXPR}, \text{ EXPR BOP1 EXPR} \rightarrow \text{EXPR}\}$
< $\{ \text{UAOP EXPR} \rightarrow \text{EXPR}, \text{ UBOP EXPR} \rightarrow \text{EXPR} \}$

B.7 module SyntaxStats

module SyntaxStats

imports SyntaxExpr

exports

sorts LABEL STAT STAT-SEQ ASSIGN-STAT PROC-STAT
COND-STAT LOOP-STAT IN-OUT-STAT GOTO-STAT EMPTY-STAT

context-free functions

ID	→ LABEL
{STAT ";" }+	→ STAT-SEQ
	→ EMPTY-STAT
ID	→ PROC-STAT
VARIABLE ASGN EXPR	→ ASSIGN-STAT
IF EXPR THEN STAT-SEQ ELSE STAT-SEQ END	→ COND-STAT
IF EXPR THEN STAT-SEQ END	→ COND-STAT
WHILE EXPR DO STAT-SEQ END	→ LOOP-STAT
READ "(" VARIABLE ")"	→ IN-OUT-STAT
WRITE "(" EXPR ")"	→ IN-OUT-STAT
WRITE "(" STRING ")"	→ IN-OUT-STAT
GOTO LABEL	→ GOTO-STAT
ID LPAR {EXPR ";" }+ RPAR	→ PROC-STAT

ASSIGN-STAT	→ STAT-AUX
COND-STAT	→ STAT-AUX
LOOP-STAT	→ STAT-AUX
PROC-STAT	→ STAT-AUX
GOTO-STAT	→ STAT-AUX
IN-OUT-STAT	→ STAT-AUX
EMPTY-STAT	→ STAT-AUX

STAT-AUX	→ STAT
LABEL ":" STAT-AUX	→ STAT

variables

$[-]$ <i>StatSeq</i> $[0-9]^*$	→ {STAT ";" }+
$[-]$ <i>StatSeq</i> $[0-9]^*$ "*"	→ {STAT ";" }*
$[-]$ <i>Stat</i> $[0-9]^*$	→ STAT
$[-]$ <i>ExprList</i> $[0-9]^*$	→ {EXPR ";" }*
$[-]$ <i>StatAux</i> $[']^*$	→ STAT-AUX
$[-]$ <i>Label</i> $[']^*$	→ LABEL
$[-]$ <i>AssignStat</i>	→ ASSIGN-STAT
$[-]$ <i>CondStat</i>	→ COND-STAT
$[-]$ <i>LoopStat</i>	→ LOOP-STAT
$[-]$ <i>InOutStat</i>	→ IN-OUT-STAT
$[-]$ <i>ProcStat</i>	→ PROC-STAT
$[-]$ <i>EmptyStat</i>	→ EMPTY-STAT

hiddens

sorts STAT-AUX

B.8 module SyntaxProgram

module SyntaxProgram

imports SyntaxHeaders SyntaxStats

exports

sorts PROGRAM

context-free functions

DECLARE DECL-LIST BEGIN STAT-SEQ END → BLOCK

BEGIN STAT-SEQ END → BLOCK

"PROGRAM" ID ";" BLOCK "." → PROGRAM

variables

[*-*] *Program* [0-9']* → PROGRAM

C A type-checker for CLaX

Once a program is accepted as valid according to the CLaX syntax, the context-dependent aspects of the program need to be checked. E.g., multiple declarations of the same variable are not allowed within one scope. The formal static semantics presented here were defined using the (informal) description of CLaX presented in [Con91]. The modules which constitute the complete static semantics of CLaX can be found in Appendix D. From these modules, a type-checker for CLaX is generated by the ASF+SDF Meta-Environment. In this section, we present the highlights of the specification of the CLaX type-checker. The specification is written in a non-standard abstract interpretation style.

C.1 Overview

Type-checking is performed in a compositional manner: the meaning of a compound CLaX construct is defined in terms of the meanings of its sub-constructs. The basic strategy of the CLaX type-checker consists of the following steps:

1. Distribution of the context (i.e., type information for all identifiers in the current scope) over every program construct.
2. Replacement of identifiers and values by a common abstract representation. We use *types* for abstract representations.
3. Evaluation of expressions using the abstract values obtained in the previous step.

In the course of performing step 3, all type-correct program constructs are removed from the result. What remains is a list containing only the abstract values of the *incorrect* program constructs. In an additional phase, we generate acceptable error-messages from these abstract values. Section 4 discusses how the system automatically locates the source positions of these errors.

C.2 Extending the language

The syntax of the language is generalized in the module `TcSyntaxExt` shown below, in order to be able to replace program constructs by their abstract values. Note that this does not affect the CLaX language itself, since this extension will only be used in the type-checking modules.

```
module TcSyntaxExt
imports SyntaxProgram
exports
  context-free functions
    EXPR ASGN EXPR    → ASSIGN-STAT
    EXPR LSQ EXPR RSQ → EXPR
    READ "(" EXPR ")" → IN-OUT-STAT
```

Module `TcSyntaxExt` (above) generalizes (i) the assignment statement, (ii) the array indexing and (iii) the input statement. The reason for these generalizations is to make the specification of the type-checker uniform over all constructs of the language. These extensions will be elaborated on as needed later.

C.3 Type-environments

The module `TcTenv` specifies the type-environment (or the context) in which the statements of a particular `BLOCK` will be type-checked. The declarations as seen from a particular point in the program are represented simply as a list (`TENV`) of variable-declaration (`VAR-DECL`) mappings. The sort `TYPE` is also extended with `LABEL` and (later) sort `PROC-TYPE`, so that all identifiers can be uniformly mapped to their types using the `ID : TYPE` notation.

module TcTenv

```
imports TcSyntaxExt
```

```
exports
```

```
  sorts TENV
```

```
  context-free functions
```

```
    TYPE                → EXPR
    LABEL-LEX           → TYPE
    “[” {VAR-DECL “;”}* “[” → TENV
    TENV “.” EXPR       → TYPE
    tenv(TENV*)         → TENV
```

Because we want to use types as the abstract values, sort `TYPE` is now injected into sort `EXPR`. Equations [1]—[3] below (over sort `EXPR`) rewrite all constants *found in expressions* to their abstract values.

[1] `_IntConst` = INTEGER [2] `_RealConst` = REAL [3] `_BoolConst` = BOOLEAN

The operation `TENV.EXPR` extracts the abstract value of an expression from a type-environment. The inclusion of this operation in `TYPE` indicates the intention that it reduces to an abstract value and also helps us to define this operation by distributing it over the operations of the expressions (See equations [V0] and [V1] of module `TcExpr` in Appendix D.3).

Function `tenv` takes a list of `TENV`s and returns the effective `TENV` by (i) only considering the `LABEL` declarations of the most recent scope (`TENV`) as seen in equation [T1] and (ii) by considering the most recent mapping of an identifier when there are multiple mappings. The idea is that, during type-checking, a *list* of type-environments (`ID : TYPE` mappings) is seen from a given point in the program. This list, however, can be reduced to *one* effective type-environment for that point.

C.4 Evaluation of expressions over abstract values

Module `TcExpr` describes the evaluation of expressions over abstract values. The expressions are transformed to a more general form (sort `OP`) so that (i) the equations that distribute an environment over an expression, (ii) the equations that evaluate an expression (over abstract values), and (iii) those that generate readable error messages can be written in a generalized fashion.

The main idea is that all *type-correct* expressions are converted to their abstract value, whereas *type-incorrect* expressions will only be evaluated partially. As an example, we show

equations [t0] and [7] of module TcExpr. The former transforms type-correct comparison expressions to a generalized form; the latter replaces generalized comparison expressions by their abstract value, i.e., BOOLEAN.

$$[t0] \frac{_Op = _Cop}{_Expr _Cop _Expr' = _Expr _Op _Expr'}$$

$$[7] \frac{_Op = _Cop}{_SimpleType _Op _SimpleType = \text{BOOLEAN}}$$

C.5 Type-checking procedures

The most significant parts of module TcProc (Appendix D.5), which deals with type-checking of procedure calls, are shown below. In order to ease the specification of type-checking procedures, we introduce the sort PROC-TYPE (denoting the signature of a procedure) and inject this into sort TYPE. This allows procedure signatures to be in the range of ID : TYPE mappings in a type-environment (TENV). For convenience, we introduce a sort VTYPE denoting a type that is (optionally) preceded by the keyword VAR.

A procedure header (sort PROC-HEAD) is reduced to the ID : PROC-TYPE form by the function signature. The formal variable declarations in a procedure heading along with their type declarations are reduced to a type-environment by the function formals.

Procedure calls are type-checked by matching the abstract form of the procedure header against the abstract form of the procedure call. In the case of a variable parameter, we have the additional constraint that the actual parameter must be a variable; this is checked by the function vararg. All type-correct calls are eliminated, resulting in abstract forms of type-incorrect calls only.

module TcProc

imports TcExpr TcBooleans

exports

sorts PROC-TYPE VTYPE TYPE-LIST

context-free functions

"PROC" "(" {VTYPE ";" }* ")"	→ PROC-TYPE
PROC-TYPE	→ TYPE
{TYPE ";" }*	→ TYPE-LIST
TENV ".((" {EXPR ";" }* ")")	→ TYPE-LIST
formals PROC-HEAD	→ TENV
signature PROC-HEAD	→ VAR-DECL
TYPE	→ VTYPE
VAR-LEX TYPE	→ VTYPE
vtype(FORMAL)	→ VTYPE
isproc "(" EXPR LPAR TYPE-LIST RPAR ")")	→ BOOL
vararg "(" EXPR LPAR {EXPR ";" }* RPAR ")")	→ BOOL

C.6 Type-checking labels

Module `TcLabel` (Appendix D.6) handles the various cases of label consistency that must be checked so as to type-check a GOTO statement. The various cases are: (i) a label must be declared before being used, (ii) a label must be defined so that a GOTO can succeed, and (iii) a label must be uniquely defined.

The most significant functions of module `TcLabel` are shown below. `ID*` defines a LABEL-LIST, an auxiliary sort used in the definition of other label consistency check functions. The list of labels that are actually used in the GOTO statements of a given statement sequence is generated by `gotos`. The list of labels that are defined (and possibly multiply defined) in a given statement sequence is generated by `defines`. The uniqueness of label definitions is checked using the function `unique`, which returns true if the list of labels does not contain elements more than once. For checking whether labels used in GOTO statements are defined, the function `ID* def ID*` is used, which returns true if the list on the left is a subset of the list on the right. The function is used in checking if the list of labels generated by `gotos` is a subset of the list of labels generated by `defines`. Checking if an (abstract) value is a label is done by function `islabel`, which returns true if the expression is LABEL.

Note that in the above, definitions of `gotos` and `defines` assume a statement sequence without any (non-trivial) complex statements (like WHILE and IF) since these statements will be flattened by the function `flat` defined in module `Tc`.

module TcLabel

imports TcProc

exports

sorts LABEL-LIST

context-free functions

<code>ID*</code>	<code>→ LABEL-LIST</code>
<code>"gotos" STAT-SEQ</code>	<code>→ LABEL-LIST</code>
<code>"defines" STAT-SEQ</code>	<code>→ LABEL-LIST</code>
<code>"unique" ID*</code>	<code>→ BOOL</code>
<code>ID* "def" ID*</code>	<code>→ BOOL</code>
<code>islabel(EXPR)</code>	<code>→ BOOL</code>

C.7 Type-checking programs

Module `Tc` (Appendix D.8) defines the function `tc` for type-checking entire programs; the syntax part of module `Tc` is shown below. The function `tc` is invoked by the **Typecheck** button on the editor window. The type-checking process is started by placing an initial pointer (denoted by a `^` symbol) at the beginning of the block with an initially empty local TENV and an initially empty global TENV. `TENV* ^ BLOCK` defines the semantics of type-checking the block in a given context (i.e., a list of type-environments from which an effective type-environment is derived).

Informally, the type-checking proceeds as follows:

- The declarations of the block are processed, yielding a local type-environment.
- Some simple checks on the local environment are performed. The function `unique-decls` checks that the association of identifiers is unique within the scope, and `nonemptyarray`

checks if the index ranges of arrays contain at least one element.

- All IF and WHILE statements are *flattened*: the statement series inside these statements are moved outside the IF/WHILE.
- All statements and expressions are type-checked using the previously described functions for type-checking statements and expressions.
- The list of statements in the block is transformed into a *conjunction* of statements, which can, in principle, be processed in parallel. To this end, sort STAT is injected in sort BOOL (module TcBooleans, Appendix D.4). Each type-correct statement that evaluates to true disappears, because true is an identity in conjunctions.

module Tc

imports TcLabel TcNint

exports

context-free functions

tc(PROGRAM)	→	BOOL
TENV* “” BLOCK	→	BOOL
STAT	→	BOOL
isbool(EXPR)	→	BOOL
unique-decls(TENV)	→	BOOL
nonemptyarray(TENV)	→	BOOL

hiddens

context-free functions

flat STAT-SEQ	→	STAT-SEQ
---------------	---	----------

C.8 Generating error messages

As we already mentioned, the result of type-checking a CLaX program is a list of abstract values representing incorrect constructs. These constructs can be transformed into human-readable error messages in a modular manner, by applying the function errors of module TcErrors (Appendix D.9). The function errors is executed by distributing a function errors over all transformed statements which remained after type-checking. Thus, each equation for the function errors handles one particular type-error. It should be noted that module TcErrors does not constitute an exhaustive set of messages that can be derived from the output of the tc function, but—in order to keep the amount of information manageable—a rather conservative list of messages:

- If a statement contains more than one error, only one error is reported.
- Multiple occurrences of the same error are merged⁵.

As a result, when a user fixes all reported errors, other errors could still be reported from the corrected program text. As an example, we show the processing of LABEL := EXPR; here an error-message cannot-assign-to LABEL is generated.

[S0z] errors(LABEL := *Expr*) = cannot-assign-to LABEL in :=

⁵Nevertheless, no essential information is lost, because origin tracking permits us to determine the source positions of all errors (see Section 4).

C.9 Example

Consider the type-checking of the following CLaX program:

```
PROGRAM test;
DECLARE
  n : REAL;
  i : INTEGER;
PROCEDURE square (n : INTEGER);
DECLARE
  x : REAL;
  step : LABEL;
BEGIN
  x := 0; step := n; step := step * 0.01;
  WHILE x < 1.0 DO
    WRITE (x); WRITE (" ** 2 = "); WRITE (x * x); WRITE ("\n");
    step: x := x + step
  END ;
  GOTO step ;
  step:
END ;
BEGIN (* main program *)
  i := 0;
  WHILE i < 0 DO
    WRITE("Enter number greater than 0");
    READ(i);
  END;
  square(n)
END.
```

Our type-checker basically involves the following steps:

1. Change constants to their abstract values. For instance, the main program in the example will look like:

```
BEGIN
  i := INTEGER;
  WHILE i < INTEGER DO
    WRITE("Enter number greater than 0");
    READ(i);
  END;
  square ( n )
END.
```

Note that integer constants are represented by their abstract values, however, since the strings are not of sort TYPE in CLaX (and there are no operations defined over strings) they don't have an abstract value.

2. Build the context—which is the effective TENV for a given statement and thus for a given expression. For instance, before entering the type-checking of the statements in the procedure square, a snap-shot might look like:

```

[ i : INTEGER;
  square : PROC (INTEGER);
  n : INTEGER;
  x : REAL;
  step : LABEL
]^
DECLARE
  BEGIN
    x := INTEGER;
    step := n;
    step := step * REAL; ...
  END
&
[ n : REAL;
  i : INTEGER;
  square : PROC (INTEGER)
]^
DECLARE
  BEGIN
    i := INTEGER; ...
  END

```

3. Check the consistency of GOTO statements before checking a block. For instance, before spawning the checking of the statements in procedure square, the following label error is produced (in this case):

```

unique step step
&
[ i : INTEGER;
  square : PROC (INTEGER);
  n : INTEGER;
  x : REAL;
  step : LABEL ]^
  BEGIN x := INTEGER;
    step := n;
    step := step * REAL; ...
  END ...

```

4. Convert the list of statements to a conjunction of statements. For instance, the next step might look like:

```

unique step step &
REAL := INTEGER &

```

Normal form	Cause of the error
unique step step &	Label step is defined twice
LABEL := INTEGER &	Cannot assign to label
LABEL := LABEL * REAL &	Cannot operate on label
REAL := REAL + LABEL &	Cannot operate on label
isproc (PROC (INTEGER) (REAL))	Procedure called with incompatible arguments

Table 4: The result of type-checking the program.

```
LABEL := INTEGER &
LABEL := LABEL * REAL &
...
```

5. Perform abstract evaluation of expressions.
6. Eliminate the cases of correct statements.

Table 4 shows the normal form of the type-checking process. For each of the conjuncts in this error message, the cause of the error is listed.

7. Generate error messages. The normal form of Table 4 is translated to:

```
multiply-defined-label step ;
cannot-assign-to LABEL in := ;
used-as-operand LABEL ;
procedure-call () expected-arg INTEGER found-arg REAL
```

The translator has converted LABEL := LABEL * REAL into the error-message cannot-assign-to LABEL in :=. Moreover, it has merged two occurrences of the same error-message.

Note that the generated error messages do not contain information regarding the *positions* where the errors occurred. Section 4 discusses how the type-checker specification described above can be used to automatically derive an error reporter, i.e., a tool which indicates the positions of errors.

C.10 Type-checking incomplete programs

Next we introduce some placeholders to the example program to illustrate how we can type-check incomplete programs. For simplicity, the type-errors above are removed from the program.

```
PROGRAM test;
DECLARE
  n : INTEGER;
  <DECL>;
  i : INTEGER;
  PROCEDURE square (n : INTEGER);
DECLARE
```

```

x : REAL;
step : REAL;
BEGIN
x := 0; <STAT>; step := step * 0.01;
WHILE x < 1.0 DO
WRITE (x); WRITE (" ** 2 = "); WRITE (x * x); WRITE ("\n");
lab0: x := x + <EXPR>
END;
GOTO lab0;
END;
BEGIN (* main program *)
i := 0;
WHILE i < 0 DO
WRITE("Enter number greater than 0");
READ(i);
END;
square(n)
END.

```

The result of type-checking this incomplete program is:

```

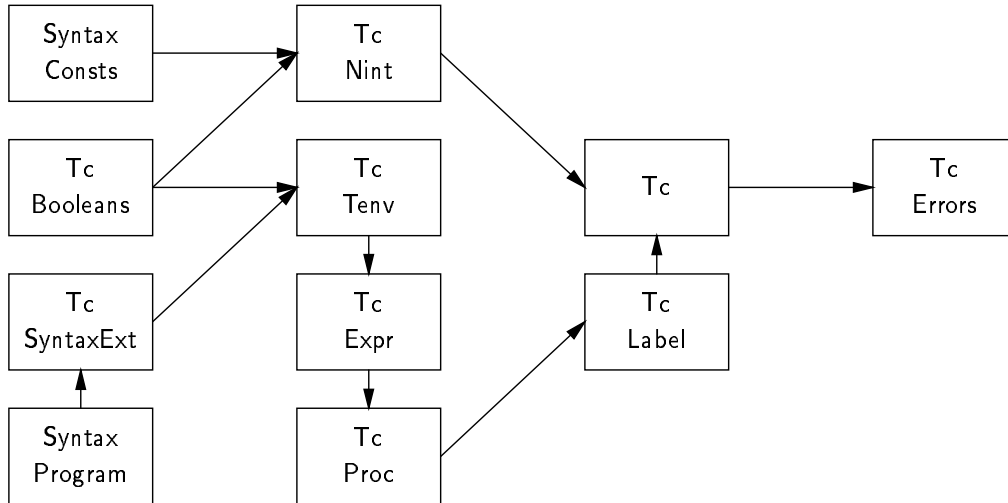
isproc ( <STAT> ( ) ) &
islabel([ <DECL> ] . lab0) &
REAL := REAL + <EXPR> &
islabel([ <DECL> ] . lab0)

```

Except for the first conjunct which thinks a <STAT> placeholder is a procedure call (due to an idiosyncrasy in the system), the other messages seem proper and obvious to decipher.

D CLaX type-checking modules

The specification below is the full specification of the CLaX type-checker. It imports the CLaX syntax defined in module `SyntaxProgram` (see Appendix B). The import diagram for the type-checking modules is:



D.1 module `TcSyntaxExt`

module `TcSyntaxExt`

imports `SyntaxProgram`

exports

context-free functions

`EXPR ASGN EXPR` \rightarrow `ASSIGN-STAT`

`EXPR LSQ EXPR RSQ` \rightarrow `EXPR`

`READ "(" EXPR ")"` \rightarrow `IN-OUT-STAT`

priorities

{ `VARIABLE ASGN EXPR` \rightarrow `ASSIGN-STAT`,
`VARIABLE LSQ EXPR RSQ` \rightarrow `VARIABLE`,
`READ "(" VARIABLE ")"` \rightarrow `IN-OUT-STAT` } >

{ `EXPR ASGN EXPR` \rightarrow `ASSIGN-STAT`, `EXPR LSQ EXPR RSQ` \rightarrow `EXPR`,
`READ "(" EXPR ")"` \rightarrow `IN-OUT-STAT` }

equations

$$[t1] \frac{_Expr' = _Var}{_Var := _Expr = _Expr' := _Expr}$$

$$[t2] \frac{_Expr = _Var}{READ(_Var) = READ(_Expr)}$$

D.2 module TcTenv

module TcTenv

imports TcSyntaxExt TcBooleans

exports

sorts TENV

context-free functions

TYPE \rightarrow EXPR
LABEL-LEX \rightarrow TYPE
“[” {VAR-DECL “;”}* “]” \rightarrow TENV
TENV “.” EXPR \rightarrow TYPE
tenv(TENV*) \rightarrow TENV

variables

[_] C “*” \rightarrow TENV*
[_] D “*” [']* \rightarrow {VAR-DECL “;”}*
[_] D [']* \rightarrow VAR-DECL
[_] D “+” [']* \rightarrow {VAR-DECL “;”}+
[_] Tenv [']* \rightarrow TENV

hiddens

context-free functions

merge(TENV, TENV) \rightarrow TENV
occurs(ID, TENV) \rightarrow BOOL

equations

[1] *_IntConst* = INTEGER [2] *_RealConst* = REAL [3] *_BoolConst* = BOOLEAN

[O1] $\text{occurs}(_Id, [_Id : _Type; _D*]) = \text{true}$

[O2] $\text{occurs}(_Id, []) = \text{false}$

[O3]
$$\frac{_Id \neq _Id'}{\text{occurs}(_Id, [_Id' : _Type; _D*]) = \text{occurs}(_Id, [_D*])}$$

[M1] $\text{merge}([], _Tenv) = _Tenv$

[M2]
$$\frac{\begin{array}{l} \text{occurs}(_Id, [_D*']) = \text{true}, \\ _Type \neq \text{LABEL} \end{array}}{\text{merge}([_Id : _Type; _D*], [_D*']) = \text{merge}([_D*], [_D*'])}$$

[M3]
$$\frac{\begin{array}{l} \text{occurs}(_Id, [_D*']) = \text{false}, \\ \text{merge}([_D*], [_D*']) = [_D*'], \\ _Type \neq \text{LABEL} \end{array}}{\text{merge}([_Id : _Type; _D*], [_D*']) = [_Id : _Type; _D*']}$$

[M4] $\text{merge}([_Id : \text{LABEL}; _D*], _Tenv) = \text{merge}([_D*], _Tenv)$

[T1] $\text{tenv}(_Tenv _Tenv' _C*) = \text{tenv}(\text{merge}(_Tenv, _Tenv') _C*)$

[T2] $\text{tenv}(_Tenv) = _Tenv$

[E0] $_Tenv . _Type = _Type$

[E1] $[_D*; _Id : _Type; _D*'] . _Id = _Type$

[E2]
$$\frac{_Id \neq _Id'}{[_D*; _Id' : _Type; _D*'] . _Id = [_D*; _D*'] . _Id}$$

[E3] $_Tenv . _Var [_Expr'] = _Tenv . _Var [_Tenv . _Expr']$

[E4] $\text{ARRAY} [_IntConst .. _IntConst'] \text{ OF } _Type [\text{INTEGER}] = _Type$

D.3 module TcExpr

module TcExpr

imports TcTenv

exports

sorts AOP BOP UOP OP

context-free functions

AOP1 → AOP
BOP1 → BOP
AOP2 → AOP
BOP2 → BOP
AOP → OP
BOP → OP
COP → OP
UAOP → UOP
UBOP → UOP

EXPR OP EXPR → EXPR

UOP EXPR → EXPR

variables

[_] Aop [']* → AOP

[_] Bop [']* → BOP

[_] Op [']* → OP

[_] Uop [']* → UOP

priorities

{ UAOP EXPR → EXPR, UBOP EXPR → EXPR, EXPR COP EXPR → EXPR,
EXPR AOP2 EXPR → EXPR, EXPR BOP2 EXPR → EXPR,
EXPR AOP1 EXPR → EXPR, EXPR BOP1 EXPR → EXPR } >
{ UOP EXPR → EXPR, EXPR OP EXPR → EXPR }

equations

$$[t0] \frac{Op = Cop}{Expr Cop Expr' = Expr Op Expr'}$$

$$[t1] \frac{Op = Aop_1}{Expr Aop_1 Expr' = Expr Op Expr'}$$

$$[t2] \frac{Op = Aop_2}{Expr Aop_2 Expr' = Expr Op Expr'}$$

$$[t3] \frac{Op = Bop_1}{Expr Bop_1 Expr' = Expr Op Expr'}$$

$$[t4] \frac{Op = Bop_2}{Expr Bop_2 Expr' = Expr Op Expr'}$$

$$[t5] \frac{_{Uop} = _{Ubop}}{_{Ubop} _Expr = _{Uop} _Expr}$$

$$[t6] \frac{_{Uop} = _{Uaop}}{_{Uaop} _Expr = _{Uop} _Expr}$$

$$[1] \frac{_{Uop} = \text{NOT}}{_{Uop} \text{ BOOLEAN} = \text{BOOLEAN}}$$

$$[2] \frac{_{Uop} \neq \text{NOT}}{_{Uop} \text{ INTEGER} = \text{INTEGER}}$$

$$[3] \frac{_{Uop} \neq \text{NOT}}{_{Uop} \text{ REAL} = \text{REAL}}$$

$$[4] \frac{_{Op} = _{Aop}}{\text{INTEGER } _{Op} \text{ INTEGER} = \text{INTEGER}}$$

$$[5] \frac{_{Op} = _{Aop}, _{Op} \neq \%}{\text{REAL } _{Op} \text{ REAL} = \text{REAL}}$$

$$[6] \frac{_{Op} = _{Bop}}{\text{BOOLEAN } _{Op} \text{ BOOLEAN} = \text{BOOLEAN}}$$

$$[7] \frac{_{Op} = _{Cop}}{_{SimpleType} _Op _SimpleType = \text{BOOLEAN}}$$

$$[v0] _Tenv _Uop _Expr = _{Uop} _Tenv _Expr$$

$$[v1] _Tenv _Expr _Op _Expr' = _{Tenv} _Expr _Op _Tenv _Expr'$$

D.4 module TcBooleans

module TcBooleans

exports

sorts BOOL BOOL-CON

context-free functions

true → BOOL-CON

false → BOOL-CON

BOOL-CON → BOOL

BOOL "&" BOOL → BOOL {**assoc**}

(" BOOL ") → BOOL {**bracket**}

variables

Bool [1-9']* → BOOL

Bool-con [1-9']* → BOOL-CON

equations

[1] true & *Bool* = *Bool*

[2] *Bool* & true = *Bool*

D.5 module TcProc

module TcProc

imports TcExpr

exports

sorts PROC-TYPE VTYPE TYPE-LIST

context-free functions

"PROC" "(" {VTYPE ";" }* ")"	→ PROC-TYPE
PROC-TYPE	→ TYPE
{TYPE ";" }*	→ TYPE-LIST
TENV ".(((" {EXPR ";" }* "))"	→ TYPE-LIST
formals PROC-HEAD	→ TENV
signature PROC-HEAD	→ VAR-DECL
TYPE	→ VTYPE
VAR-LEX TYPE	→ VTYPE
vtype(FORMAL)	→ VTYPE
isproc "(" EXPR LPAR TYPE-LIST RPAR ")"	→ BOOL
vararg "(" EXPR LPAR {EXPR ";" }* RPAR ")"	→ BOOL

variables

[_] <i>TypeList</i> [<i>l</i>]*	→ {TYPE ";" }*
[_] <i>ProcType</i> [<i>l</i>]*	→ PROC-TYPE
[_] <i>Vtype</i> [<i>l</i>]*	→ VTYPE
[_] <i>VtypeList</i> [<i>l</i>]*	→ {VTYPE ";" }*

equations

[AE0] $_Tenv .(()) =$

[AE1]
$$\frac{_Type = _Tenv . _Expr, _Tenv .((_ExprList)) = _TypeList}{_Tenv .((_Expr, _ExprList)) = _Type; _TypeList}$$

[F0] formals PROCEDURE $_Id = []$

[F1] formals PROCEDURE $_Id (_D) = [_D]$

[F2] formals PROCEDURE $_Id (VAR _D) = [_D]$

[F3]
$$\frac{\begin{array}{l} \text{formals PROCEDURE } _Id (_Formal) = [_D], \\ \text{formals PROCEDURE } _Id (_Formal+) = [_D+] \end{array}}{\text{formals PROCEDURE } _Id (_Formal; _Formal+) = [_D; _D+]}$$

[S0] signature PROCEDURE $_Id = _Id : PROC()$

[S1] signature PROCEDURE $_Id (_Formal) = _Id : PROC(vtype(_Formal))$

[S2]
$$\frac{\text{signature PROCEDURE } _Id (_Formal+) = _Id : PROC(_VtypeList)}{\text{signature PROCEDURE } _Id (_Formal; _Formal+) = _Id : PROC(vtype(_Formal); _VtypeList)}$$

[P0] isproc(PROC() ()) = true

[P1] isproc(PROC(_Type; _VtypeList) (_Type; _TypeList)) =
isproc(PROC(_VtypeList) (_TypeList))

[P2] isproc(PROC(VAR _Type; _VtypeList) (_Type; _TypeList)) =
isproc(PROC(_VtypeList) (_TypeList))

[VA0] `vararg(PROC() ()) = true`
[VA1] `vararg(PROC(_Type;_VtypeList) (_Expr,_ExprList)) = vararg(PROC(_VtypeList) (_ExprList))`
[VA2] `vararg(PROC(VAR _Type;_VtypeList) (_Var,_ExprList)) = vararg(PROC(_VtypeList) (_ExprList))`

[VT0] `vtype(VAR _Id : _Type) = VAR _Type`
[VT1] `vtype(_Id : _Type) = _Type`

D.6 module TcLabel

module TcLabel

imports TcProc

exports

sorts LABEL-LIST

context-free functions

ID* → LABEL-LIST
"gotos" STAT-SEQ → LABEL-LIST
"defines" STAT-SEQ → LABEL-LIST
"unique" ID* → BOOL
ID* "def" ID* → BOOL
islabel(EXPR) → BOOL

variables

[_] Labels [']* → ID*
[_] Labels "+" → ID+

equations

[L0] defines $_StatAux =$

[L1] defines $_Id : _StatAux = _Id$

$_Labels' = \text{defines } _Stat,$

$_Labels'' = \text{defines } _StatSeq,$

[L2]
$$\frac{_Labels = _Labels' _Labels''}{\text{defines } _Stat; _StatSeq = _Labels}$$

[U0] unique = true

[U1] unique $_Id = \text{true}$

[U2]
$$\frac{_Id \neq _Id'}{\text{unique } _Id _Id' = \text{true}}$$

[U3] unique $_Id _Id' _Labels+ =$
unique $_Id _Id' \ \& \ \text{unique } _Id _Labels+ \ \& \ \text{unique } _Id' _Labels+$

[G0] gotos $_Id : _StatAux = \text{gotos } _StatAux$

[G1] gotos GOTO $_Id = _Id$

[G2a] gotos $_AssignStat =$

[G2b] gotos $_CondStat =$

[G2c] gotos $_LoopStat =$

[G2d] gotos $_InOutStat =$

[G2e] gotos $_ProcStat =$

[G2f] gotos $_EmptyStat =$

$$\text{[G3]} \frac{\begin{array}{l} _Labels = \text{gotos } _Stat, _Labels' = \text{gotos } _StatSeq, \\ _Labels'' = _Labels _Labels' \end{array}}{\text{gotos } _Stat; _StatSeq = _Labels''}$$

[S0] def *_Labels* = true
 [S1] *_Id* def *_Id* *_Labels* = true
 [S2]
$$\frac{_Id \neq _Id'}{_Id \text{ def } _Id' _Labels = _Id \text{ def } _Labels}$$

 [S3] *_Id* *_Labels*+ def *_Labels* = *_Id* def *_Labels* & *_Labels*+ def *_Labels*
 [IsL0] islabel(LABEL) = true

D.7 module TcNint

module TcNint

imports TcBooleans SyntaxConsts

exports

context-free functions

INT-CONST islessthan INT-CONST → BOOL

hiddens

sorts INT INT-CON POS NEG NAT AUX

lexical syntax

[1-9] [0-9]* → POS

[+\-0] → AUX

context-free functions

toint INT-CONST → INT

“-” POS → NEG

“0” → NAT

POS → NAT

NAT → INT-CON

NEG → INT-CON

INT-CON → INT

“P” INT → INT

“S” INT → INT

“(” INT “)” → INT {**bracket**}

INT “<” INT → BOOL

“-” INT → INT

variables

Int [0-9']* → INT

hiddens

context-free functions

INT “;” INT → INT {**left**}

hd “(” INT “)” → INT

tl “(” INT “)” → INT

“bigpos?” “(” INT “)” → BOOL

variables

Int [0-9']* → INT

[*xy*] [0-9']* → INT

[*z*] [0-9']* → NEG

[*n*] [0-9']* → POS

c [0-9']* → CHAR

c [0-9']* “+” → CHAR+

c [0-9']* “*” → CHAR*

priorities

“;” < { “S” INT → INT, “P” INT → INT, “-” INT → INT } <
{ “-” POS → NEG }

equations

[S0] $S - 1 = 0$	[S1] $S 0 = 1$	[S2] $S 1 = 2$
[S3] $S 2 = 3$	[S4] $S 3 = 4$	
[S5] $S 4 = 5$	[S6] $S 5 = 6$	[S7] $S 6 = 7$
[S8] $S 7 = 8$	[S9] $S 8 = 9$	
[P0] $P 0 = - 1$	[P1] $P 1 = 0$	[P2] $P 2 = 1$
[P3] $P 3 = 2$	[P4] $P 4 = 3$	
[P5] $P 5 = 4$	[P6] $P 6 = 5$	[P7] $P 7 = 6$
[P8] $P 8 = 7$	[P9] $P 9 = 8$	

$$[11] \frac{tl(n) \neq 9, \text{bigpos?}(n) = \text{true}}{S\ n = \text{hd}(n); S\ tl(n)}$$

$$[12] \frac{tl(n) = 9}{S\ n = S\ \text{hd}(n); 0}$$

$$[13] \frac{n' = P\ n}{S\ -\ n = -\ n'}$$

$$[14] \frac{tl(n) \neq 0, \text{bigpos?}(n) = \text{true}}{P\ n = \text{hd}(n); P\ tl(n)}$$

$$[15] \frac{tl(n) = 0, \text{bigpos?}(n) = \text{true}}{P\ n = P\ \text{hd}(n); 9}$$

$$[16] \frac{n' = S\ n}{P\ -\ n = -\ n'}$$

- [l1] $0 < 0 = \text{false}$
- [l2] $0 < n = \text{true}$
- [l3] $0 < z = \text{false}$
- [l4] $n < x = P\ n < P\ x$
- [l5] $z < x = S\ z < S\ x$

[h1] $\text{hd}(\text{pos}(c+ c)) = \text{pos}(c+)$
[h2] $\text{hd}(\text{pos}(c)) = 0$

[t1] $tl(\text{pos}(c+ "0")) = 0$

$$[t2] \frac{\text{pos}(c* c) \neq \text{pos}(c* "0")}{tl(\text{pos}(c* c)) = \text{pos}(c)}$$

[o1] $\text{bigpos?}(\text{pos}(c+ c)) = \text{true}$

[o2] $\text{bigpos?}(\text{pos}(c)) = \text{false}$

[o2] $\text{pos}(c+); \text{pos}(c'+) = \text{pos}(c+ c'+)$

[o3] $\text{pos}(c+); 0 = \text{pos}(c+ "0")$

[o4] $0; x = x$

[e1] $- n = - n$

[e2] $- - n = n$

[e3] $- 0 = 0$

[T0] $\text{toint int-const}("+ c+) = \text{toint int-const}(c+)$

[T1] $\text{toint int-const}("- c+) = - \text{toint int-const}(c+)$

[T2] $\text{toint int-const}("0" c+) = \text{toint int-const}(c+)$

[T3] $\text{toint int-const}(c) = \text{pos}(c)$

$\text{aux}(c) \neq \text{aux}("0"),$

$\text{aux}(c) \neq \text{aux}("+"),$

$\text{aux}(c) \neq \text{aux}("-")$

[T4]
$$\frac{\text{aux}(c) \neq \text{aux}("0"),$$

 $\text{aux}(c) \neq \text{aux}("+"),$
 $\text{aux}(c) \neq \text{aux}("-")}{\text{toint int-const}(c c+) = \text{pos}(c c+)}$

[L0]
$$\frac{\text{toint } _IntConst < \text{toint } _IntConst' = \text{true}}{_IntConst \text{ islessthan } _IntConst' = \text{true}}$$

D.8 module Tc

module Tc

imports TcLabel TcNint

exports

context-free functions

tc(PROGRAM) → BOOL
TENV* "" BLOCK → BOOL
STAT → BOOL
isbool(EXPR) → BOOL
unique-decls(TENV) → BOOL
nonemptyarray(TENV) → BOOL

hiddens

context-free functions

flat STAT-SEQ → STAT-SEQ

equations

[P0] tc(PROGRAM *Id*; *Block* .) = [] [] ^ *Block*

[T0] $_C^* \[_D^*]^\wedge \text{DECLARE } _EmptyDecl; _Decl^* \text{ BEGIN } _StatSeq \text{ END} =$
 $_C^* \[_D^*]^\wedge \text{DECLARE } _Decl^* \text{ BEGIN } _StatSeq \text{ END}$

[T1]
$$\frac{_LabelDecl = _Id : \text{LABEL}}{_C^* \[_D^*]^\wedge \text{DECLARE } _LabelDecl; _Decl^* \text{ BEGIN } _StatSeq \text{ END} =$$

 $_C^* \[_D^*; _Id : \text{LABEL}]^\wedge \text{DECLARE } _Decl^* \text{ BEGIN } _StatSeq \text{ END}$

$$\begin{aligned} _ProcDecl &= _ProcHead; _Block, \\ _D &= \text{signature } _ProcHead, \\ _Tenv &= \text{formals } _ProcHead, \\ Bool_1 &= _C^* \[_D^*; _D] _Tenv^\wedge _Block, \\ Bool_2 &= _C^* \[_D^*; _D]^\wedge \text{DECLARE } _Decl^* \text{ BEGIN } _StatSeq \text{ END} \end{aligned}$$

[T2]
$$\frac{Bool_2 = _C^* \[_D^*; _D]^\wedge \text{DECLARE } _Decl^* \text{ BEGIN } _StatSeq \text{ END}}{_C^* \[_D^*]^\wedge \text{DECLARE } _ProcDecl; _Decl^* \text{ BEGIN } _StatSeq \text{ END} =$$

 $\text{nonemptyarray}(_Tenv) \& Bool_1 \& Bool_2$

[T3] $_C^* \[_D^*]^\wedge \text{DECLARE } _VarDecl; _Decl^* \text{ BEGIN } _StatSeq \text{ END} =$
 $\text{nonemptyarray}(\[_VarDecl]) \& _C^* \[_D^*; _VarDecl]^\wedge \text{DECLARE } _Decl^* \text{ BEGIN } _StatSeq \text{ END}$

[T4] $_C^* _Tenv _Tenv'^\wedge \text{BEGIN } _StatSeq \text{ END} = _C^* _Tenv _Tenv'^\wedge \text{DECLARE } \text{BEGIN } _StatSeq$
 END

$$\begin{aligned} _StatSeq' &= \text{flat } _StatSeq, \\ _Labels &= \text{gotos } _StatSeq', \\ _Labels' &= \text{defines } _StatSeq' \end{aligned}$$

[T5]
$$\frac{_StatSeq' = \text{flat } _StatSeq, _Labels = \text{gotos } _StatSeq', _Labels' = \text{defines } _StatSeq'}{_C^* _Tenv^\wedge \text{DECLARE } \text{BEGIN } _StatSeq \text{ END} =$$

 $_Labels \text{ def } _Labels' \& \text{unique } _Labels' \& \text{unique-decls}(_Tenv) \&$
 $\text{tenv}(_C^* _Tenv)^\wedge \text{BEGIN } _StatSeq' \text{ END}$

[T6] $_Tenv \wedge \text{BEGIN } _Stat; _StatSeq \text{ END} =$
 $_Tenv \wedge \text{BEGIN } _Stat \text{ END} \ \& \ _Tenv \wedge \text{BEGIN } _StatSeq \text{ END}$

[TS0] $_Tenv \wedge \text{BEGIN } \text{END} = \text{true}$

[TS1] $_Tenv \wedge \text{BEGIN } _Id : _StatAux \text{ END} = \text{islabel}(_Tenv . _Id) \ \& \ _Tenv \wedge \text{BEGIN } _StatAux \text{ END}$

[TS2] $_Tenv \wedge \text{BEGIN } _Expr := _Expr' \text{ END} = _Tenv . _Expr := _Tenv . _Expr'$

[TS2x] $_SimpleType := _SimpleType = \text{true}$

[TS2y] $\text{REAL} := \text{INTEGER} = \text{true}$

[TS3] $_Tenv \wedge \text{BEGIN } _Id \text{ END} = \text{isproc}(_Tenv . _Id (\))$

[TS4] $_Tenv \wedge \text{BEGIN } _Id (_ExprList) \text{ END} =$
 $\text{isproc}(_Tenv . _Id (_Tenv . ((_ExprList)))) \ \& \ \text{vararg}(_Tenv . _Id (_ExprList))$

[TS5] $_Tenv \wedge \text{BEGIN IF } _Expr \text{ THEN ELSE END END} =$
 $\text{IF } _Tenv . _Expr \text{ THEN END}$

[TS5x] $\text{IF BOOLEAN THEN END} = \text{true}$

[TS6] $_Tenv \wedge \text{BEGIN IF } _Expr \text{ THEN END END} =$
 $\text{IF } _Tenv . _Expr \text{ THEN END}$

[TS7] $_Tenv \wedge \text{BEGIN WHILE } _Expr \text{ DO END END} =$
 $\text{WHILE } _Tenv . _Expr \text{ DO END}$

[TS7x] $\text{WHILE BOOLEAN DO END} = \text{true}$

[TS8] $_Tenv \wedge \text{BEGIN READ}(_Expr) \text{ END} = \text{READ}(_Tenv . _Expr)$

[TS8x] $\text{READ}(_SimpleType) = \text{true}$

[TS8] $_Tenv \wedge \text{BEGIN WRITE}(_Expr) \text{ END} = \text{WRITE}(_Tenv . _Expr)$

[TS8x] $\text{WRITE}(_SimpleType) = \text{true}$

[TS8y] $_Tenv \wedge \text{BEGIN WRITE}(_String) \text{ END} = \text{true}$

[S9] $_Tenv \wedge \text{BEGIN GOTO } _Id \text{ END} = \text{islabel}(_Tenv . _Id)$

[FL0]
$$\frac{\text{flat } _StatAux = _StatAux'; _StatSeq' *}{\text{flat } _Id : _StatAux = _Id : _StatAux'; _StatSeq' *}$$

[FL1] $\text{flat } _Expr := _Expr' = _Expr := _Expr'$

[FL2] $\text{flat } _Id = _Id$

[FL3] $\text{flat } _Id (_ExprList) = _Id (_ExprList)$

[FL4] $\text{flat READ}(_Expr) = \text{READ}(_Expr)$

[FL5] $\text{flat WRITE}(_Expr) = \text{WRITE}(_Expr)$

[FL6] flat WRITE(*_String*) = WRITE(*_String*)

[FL7] flat GOTO *_Id* = GOTO *_Id*

[FL8] flat =

[FL9]
$$\frac{_StatSeq' = \text{flat } _StatSeq}{\text{flat IF } _Expr \text{ THEN } _StatSeq \text{ END} = \text{IF } _Expr \text{ THEN } \text{END}; _StatSeq'}$$

[FL10]
$$\frac{_StatSeq'' = \text{flat } _StatSeq; _StatSeq'}{\text{flat IF } _Expr \text{ THEN } _StatSeq \text{ ELSE } _StatSeq' \text{ END} = \text{IF } _Expr \text{ THEN } \text{END}; _StatSeq''}$$

[FL11]
$$\frac{_StatSeq' = \text{flat } _StatSeq}{\text{flat WHILE } _Expr \text{ DO } _StatSeq \text{ END} = \text{WHILE } _Expr \text{ DO } \text{END}; _StatSeq'}$$

[FL12]
$$\frac{_StatSeq' = \text{flat } _Stat, _StatSeq'' = \text{flat } _StatSeq}{\text{flat } _Stat; _StatSeq = _StatSeq'; _StatSeq''}$$

[IB0] isbool(BOOLEAN) = true

[U0] unique-decls([]) = true

[U1] unique-decls([*_Id* : *_Type*]) = true

[U2]
$$\frac{_Id \neq _Id'}{\text{unique-decls}([_Id : _Type; _Id' : _Type'; _D*]) = \text{unique-decls}([_Id : _Type; _D*]) \ \& \ \text{unique-decls}([_Id' : _Type'; _D*])}$$

[NA0] nonemptyarray([*_Id* : *_SimpleType*]) = true

[NA1] nonemptyarray([*_Id* : LABEL]) = true

[NA2] nonemptyarray([*_Id* : ARRAY [*_IntConst* .. *_IntConst'*] OF *_Type*]) = *_IntConst* islessthan *_IntConst'* & nonemptyarray([*_Id* : *_Type*])

[NA3] nonemptyarray([]) = true

[NA4] nonemptyarray([*_D*; *_D+*]) = nonemptyarray([*_D*]) & nonemptyarray([*_D+*])

D.9 module TcErrors

module TcErrors

imports Tc

exports

sorts MESSAGE MSG-LIST

context-free functions

errors(PROGRAM) → MSG-LIST
errors(BOOL) → MSG-LIST
{MESSAGE “;”}+ → MSG-LIST
err(TYPE-LIST) → MSG-LIST

no-errors → MESSAGE
err(EXPR) → MESSAGE
incomp(MESSAGE) → MESSAGE

undeclared-identifier ID → MESSAGE
incompatible-operands(EXPR) → MESSAGE
incompatible-array-access(EXPR) → MESSAGE
used-as-operand EXPR → MESSAGE
assignment-incompatible BOOL → MESSAGE
cannot-assign-to EXPR “in” ASGN → MESSAGE
“Boolean-expected” IF MESSAGE THEN → MESSAGE
“Boolean-expected” WHILE MESSAGE DO → MESSAGE
undeclared-procedure-called ID → MESSAGE
procedure-call LPAR RPAR expected-arg VTYPE found-arg MESSAGE → MESSAGE
procedure-call LPAR RPAR expected-no-more-args-but-found TYPE-LIST → MESSAGE
procedure-call LPAR RPAR expected-variable-arg VTYPE found-arg EXPR → MESSAGE
only-simple-type-variable-allowed-in READ (“ MESSAGE “) → MESSAGE
only-simple-type-variable-allowed-in WRITE (“ MESSAGE “) → MESSAGE
array-decl-must-have-positive-size (“ INT-CONST “..” INT-CONST “) → MESSAGE
expected-label-found MESSAGE → MESSAGE
multiply-defined-label ID → MESSAGE
not-defined-label ID → MESSAGE
not-declared-label ID → MESSAGE
multiple-declaration-in-same-scope ID → MESSAGE
unary-operator UOP not-allowed-on-operand-of-type TYPE → MESSAGE
TYPE → MESSAGE

variables

[_] *Msg* [']* → MESSAGE
[_] *MsgList* [']* → {MESSAGE “;”}*

equations

[M-1] errors(*_Program*) = errors(tc(*_Program*))

[0] errors(true) = no-errors

- [1]
$$\frac{\begin{array}{l} _MsgList = errors(Bool_1), \\ _MsgList' = errors(Bool_2) \end{array}}{errors(Bool_1 \& Bool_2) = _MsgList ;_MsgList'}$$
- [2] $_MsgList ;_Msg ;_MsgList' ;_Msg ;_MsgList'' = _MsgList ;_Msg ;_MsgList' ;_MsgList''$
- [S0x] errors($_SimpleType := _SimpleType'$) =
assignment-incompatible $_SimpleType := _SimpleType'$
- [S0y] errors($_Expr [_Expr'] := _Expr''$) = err($_Expr [_Expr']$)
- [S0z] errors(LABEL := $_Expr$) = cannot-assign-to LABEL in :=
- [S0z1] errors($_SimpleType := _Expr [_Expr']$) = incomp(err($_Expr [_Expr']$))
- [S0z2] errors($_SimpleType := _Expr_Op _Expr'$) = incomp(err($_Expr_Op _Expr'$))
- [S0z3] errors($_SimpleType := _Uop _Expr$) = incomp(err($_Uop _Expr$))
- [S1a] errors(isproc($[] . _Id (_TypeList)$)) = undeclared-procedure-called $_Id$
- [S1b] errors(isproc(PROC($_Vtype; _VtypeList$) ($_Type; _TypeList$))) =
procedure-call () expected-arg $_Vtype$ found-arg incomp(err($_Type$))
- [S1c] errors(isproc(PROC() ($_Type; _TypeList$))) =
procedure-call () expected-no-more-args-but-found $_Type; _TypeList$
- [S1d] errors(vararg($[] . _Id (_ExprList)$)) = undeclared-procedure-called $_Id$
- [S1e] errors(vararg(PROC($_Vtype; _VtypeList$) ($_Expr, _ExprList$))) =
procedure-call () expected-variable-arg $_Vtype$ found-arg $_Expr$
- [S2] errors(IF $_Expr$ THEN END) = Boolean-expected IF incomp(err($_Expr$)) THEN
- [S2x] errors(WHILE $_Expr$ DO END) = Boolean-expected WHILE incomp(err($_Expr$)) DO
- [S3] errors(READ($_Expr$)) = only-simple-type-variable-allowed-in READ(incomp(err($_Expr$)))
- [S4] errors(WRITE($_Expr$)) = only-simple-type-variable-allowed-in WRITE(incomp(err($_Expr$)))
- [S5] errors(islabel($_Expr$)) = expected-label-found incomp(err($_Expr$))
- [E0x] err($_Uop [] . _Id$) = err($[] . _Id$)
- [E0y] err($_Expr_Op [] . _Id$) = err($[] . _Id$)
- [E0z] err($[] . _Id_Op _Expr$) = err($[] . _Id$)
- [E1a] err($_Uop _Type _Op _Expr$) = err($_Uop _Type$)
- [E1b] err($_Type _Op _Type' _Op' _Expr$) = err($_Type _Op _Type'$)
- [E1c] err($_Expr _Op _Type _Op' _Type'$) = err($_Type _Op' _Type'$)
- [E1d] err(ARRAY [$_IntConst .. _IntConst'$] OF $_Type [_Expr]$) =
incompatible-array-access($_Expr$)

- [E1e]
$$\frac{\text{err}(_Expr') = \text{incompatible-array-access}(_Expr)}{\text{err}(_Expr' _Op _Expr'') = \text{incompatible-array-access}(_Expr)}$$
- [E1f]
$$\frac{\text{err}(_Expr'') = \text{incompatible-array-access}(_Expr)}{\text{err}(_Expr' _Op _Expr'') = \text{incompatible-array-access}(_Expr)}$$
- [E2a] $\text{err}(_Uop _Type) = \text{unary-operator } _Uop \text{ not-allowed-on-operand-of-type } _Type$
- [E2b] $\text{err}([\] . _Id) = \text{undeclared-identifier } _Id$
- [O1] $\text{incomp}(\text{undeclared-identifier } _Id) = \text{undeclared-identifier } _Id$
- [O2] $\text{incomp}(\text{err}(_SimpleType)) = _SimpleType$
- [O3] $\text{incomp}(\text{err}(_SimpleType _Op _SimpleType')) = \text{incompatible-operands}(_SimpleType _Op _SimpleType')$
- [O4] $\text{incomp}(\text{err}(\text{LABEL } _Op _Expr)) = \text{used-as-operand LABEL}$
- [O5] $\text{incomp}(\text{err}(_Expr _Op \text{LABEL})) = \text{used-as-operand LABEL}$
- [O6]
$$\frac{\text{err}(_Expr') = \text{err}(_Expr)}{\text{incomp}(\text{incompatible-array-access}(_Expr)) = \text{incompatible-array-access}(_Expr')}$$
- [O7] $\text{incomp}(\text{unary-operator } _Uop \text{ not-allowed-on-operand-of-type } _Type) = \text{unary-operator } _Uop \text{ not-allowed-on-operand-of-type } _Type$
- [O8] $\text{incomp}(\text{err}(_Uop _Expr _Op _Expr')) = \text{incomp}(\text{err}(_Uop _Expr))$
- [O9] $\text{incomp}(\text{err}(_Expr _Op _Uop _Expr')) = \text{incomp}(\text{err}(_Uop _Expr'))$
- [O10] $\text{incomp}(\text{err}(_Expr _Op _Expr' _Op' _Expr'')) = \text{incomp}(\text{err}(_Expr _Op _Expr'))$
- [O11] $\text{incomp}(\text{err}(_Expr _Op _Expr' _Op' _Expr'')) = \text{incomp}(\text{err}(_Expr' _Op _Expr''))$
- [L0] $\text{errors}(\text{unique } _Id _Id _Labels) = \text{multiply-defined-label } _Id$
- [L1] $\text{errors}(_Id \text{ def }) = \text{not-defined-label } _Id$
- [L2] $\text{errors}(\text{islabel}([\] . _Id)) = \text{not-declared-label } _Id$
- [UE0] $\text{errors}(\text{unique-decls}([_Id : _Type; _Id : _Type'; _D*])) = \text{multiple-declaration-in-same-scope } _Id$
- [AL0] $\text{errors}(_IntConst \text{ islessthan } _IntConst') = \text{array-decl-must-have-positive-size}(_IntConst .. _IntConst')$

E An interpreter for CLaX

This appendix contains the specification of an interpreter for CLaX. The specification is not a purely algebraic one, because hybrid functions [Wal91] are used to perform basic arithmetic operations and I/O in Lisp. This specification imports the CLaX syntax, as defined in Appendix B. Programs which are to be executed are assumed to be type-correct according to the CLaX type-checker as defined in Appendix D. Currently, no provisions are made to detect or report run-time errors, such as divisions by zero.

E.1 Basic Datatypes

The status of the interpreter is defined by way of two stacks: a *code stack* and a *data stack*. Together, these stacks represent the well-known concept of a stack of activation records [ASU86]. We use two separate stacks in this specification because it allows us to separate the control flow issues from the operations on the data. We believe that this separation of issues results in a more clear specification. Code stacks (sort C-STACK) are defined in module CodeStacks (see Appendix F.1) and contain the current point of execution for each procedure on the call stack (comparable to the return address). Data stacks (sort D-STACK) are defined in module DataStacks (see Appendix F.2) and contain the current values of the variables for each of the procedures on the call stack. In module EvalProgram, an interpreter status is defined as:

"<" C-STACK "," D-STACK ">" → STATUS

Code Stacks

In module C-Stacks, a code stack is defined as a list of zero or more code records (sort C-RECORD). During execution, one code record exists for each procedure on the call stack. A code record consists of the following parts: the name of the procedure, the list of statements that forms the procedure body, and a 'pointer' to an element of that list which indicates the statement which is currently being executed. Below, the definitions of code records and code stacks are shown. In this specification, the pointer to the current statement is modeled by means of another list of statements—the statements which are yet to be executed; the current statement is the first element of this list. Thus, the first list in a code record contains all code of a block, whereas the second code record contains the code which is yet to be executed.

"[" ID "," {STAT ";"}* "," {STAT ";"}* "]" → C-RECORD
C-RECORD* → C-STACK

Data Stacks

The other component of the status of the CLaX interpreter, the data stack, is defined in module D-Stacks. Like code stacks, data stacks consist of a list of zero or more data records, one for each procedure on the code stack. A data record for procedure P contains variable-value pairs for all local variables of P and parameters of P. In addition, it contains procedure definitions for all procedures which are declared in the declaration section of P.

Data record elements are represented by sort D-ELEM. The definitions of data records and data record elements will follow shortly.

Simple Values and Array Values

We inject the sort EXPR into the sort VALUE⁶. During execution, however, expressions occurring in variable-value pairs in data records will always be integer, real, or boolean constants. Array values are represented by triples which denote the lower bound, the upper bound, and a list of values bound to the successive elements of the array. Below, the relevant parts of module D-Stacks are shown.

EXPR	→ VALUE
“[” INT-CONST “,” INT-CONST “,” VALUE+ “]”	→ VALUE
ID “:” VALUE	→ D-ELEM

Static Scoping

CLaX, like Pascal, is a language with nested procedures and static scope rules (also called lexical scope rules). In [ASU86], static scoping in the presence of nested procedures is discussed. The difference between dynamic and static scoping consists of the way references to non-local variables are handled. Dynamic scoping corresponds to successively searching in the next record on the stack. Static scoping corresponds to successively searching in the most recent record for the immediately surrounding scope.

The following strategy is adopted to specify static scoping. Associated with every procedure in the program is a *path*: a (possibly empty) sequence of natural numbers. This path reflects the relative position of the procedure in the abstract syntax tree of the program, with respect to other procedures and the main program only. For example: if the declaration section of program P contains declarations for Q and R (in that order), and the declaration section of procedure R contains a declaration for procedure S, the paths (), (1), (2), and (2, 1) are associated with P, Q, R, and S, respectively. Each data record contains the path to the associated procedure, and lexical scoping is specified in the following way: if a non-local cannot be found, and the path of the current data record is $(n_1 \cdots n_k)$, the search is to be continued in the first record with path $(n_1 \cdots n_{k-1})$ that is encountered.

The specification of procedure definition entries, data records (sort D-RECORD) and data stacks is shown below.

ID “:” PATH “:” PROC-DECL	→ D-ELEM
“[” ID “,” PATH “,” D-ELEM* “]”	→ D-RECORD
D-RECORD*	→ D-STACK

⁶Alternatively, the sorts INT-CONST, REAL-CONST, and BOOL-CONST could be injected in sort EXPR. The advantage of the adopted approach is that it results in a more compact specification: EXPR can serve as the output sort of function eval-exp. Hence, eval-exp can be specified in a simple recursive manner.

Parameter Passing Mechanisms

In CLaX, both call-by-value and call-by-reference parameter passing are supported. Call-by-value parameter passing is specified by treating formal value parameters as local variables of the same name, which obtain their initial value from the corresponding actual parameter. Call-by-reference parameter passing is handled as follows. Instead of an actual value, a *reference* to an entry in a previous data record is entered as the value part of the variable-value pair. Any uses or updates of a reference value are simply regarded as uses or updates of the variable that is referred to. The specification of reference values is given below.

$$\text{ref}(\text{ID}, \text{PATH}) \rightarrow \text{VALUE}$$

Referencing and Updating of Values

The functions `lookup` and `update` define the lookup of the current value of a variable, and the update of the value of a variable, in a data stack. Simple variables and array variables are handled in a uniform way. The first two arguments of both `lookup` and `update` are an identifier and a path. In the case of a simple variable, the path is empty, and in the case of an array variable the path contains the actual indices of the array element involved. For example, when a lookup is done for array element `i[1][2]`, the identifier is `i` and the path is `(1 2)`. Several auxiliary functions are used in the definition of `lookup` and `update`; the most significant of these are the functions `lookup-list` and `update-list` which do a lookup and update of an array element, respectively. Furthermore, a function `next-scope` determines, given a data stack, the immediately surrounding scope, `get-id` and `get-value` are access functions for data stack elements, and `is-ref` is a boolean predicate which tests whether or not a value is a reference value. The signature of all functions mentioned above is presented below.

context-free functions

$$\begin{aligned} \text{lookup}(\text{ID}, \text{PATH}, \text{D-STACK}) &\rightarrow \text{VALUE} \\ \text{update}(\text{ID}, \text{PATH}, \text{D-STACK}, \text{VALUE}) &\rightarrow \text{D-STACK} \end{aligned}$$

hiddens

context-free functions

$$\begin{aligned} \text{lookup-list}(\text{PATH}, \text{VALUE}) &\rightarrow \text{VALUE} \\ \text{update-list}(\text{PATH}, \text{VALUE}, \text{VALUE}) &\rightarrow \text{VALUE} \\ \text{next-scope}(\text{D-STACK}, \text{PATH}) &\rightarrow \text{D-STACK} \\ \text{get-id}(\text{D-ELEM}) &\rightarrow \text{ID} \\ \text{get-value}(\text{D-ELEM}) &\rightarrow \text{VALUE} \\ \text{is-ref}(\text{VALUE}) &\rightarrow \text{BOOL-CONST} \end{aligned}$$

Below, the equations which define the semantics of `lookup` and `lookup-list` are listed; the definitions of `update` and `update-list` are similar (see Appendix F.2).

Equation [lo1] describes the case where a variable cannot be found in the current data record—the search is continued in the next scope. In [lo2], the identifier is not equal to the identifier of the first element of the data record—the search continues with the next element of the same record. In [lo3], an entry for a non-array variable is found in the

current record; it is not a reference value, so the lookup terminates. [lo4] is very similar to the previous case: an entry for the identifier is found in the current record, and it is not a reference value. However, as we search for an array element the search is continued by looking for an array element. This is expressed by means of lookup-list. [lo5] handles the case where a reference value is found for a simple (i.e., non-array) variable. The search continues in the data record for the calling procedure with the variable that is referred to. Finally, in [lo6], a reference value is found for an array element. The search is continued in the next record.

Recall that array structures consist of triples with the lower bound, the upper bound, and a list of values. In equation [ll1], the index of the element that is searched is equal to the lower bound of the array; the first element of the list of values is the value to be found. [ll2] describes a case similar to the previous one, however, the search has to continue because the array is multi-dimensional. This is expressed by a recursive application of lookup-list. Finally, [ll3] handles the remaining case: the element to be found is not the first element of the list of values. This is defined by way of an auxiliary function *incr* which increments an integer constant by one (module Arithmetic, Appendix F.8), and by continuing the search in the tail of the list.

$$[lo1] \frac{_DRec*' = \text{next-scope}(_DRec*, _Int*)}{\text{lookup}(_Id, _Path, [_Id', _Int* _Int,] _DRec*) = \text{lookup}(_Id, _Path, _DRec*')}$$

$$[lo2] \frac{_Id \neq \text{get-id}(_DElem)}{\text{lookup}(_Id, _Path, [_Id', _Path', _DElem _DElem*] _DRec*) = \text{lookup}(_Id, _Path, [_Id', _Path', _DElem*] _DRec*)}$$

$$[lo3] \frac{\text{is-ref}(_Value) = \text{FALSE}}{\text{lookup}(_Id, , [_Id', _Path, _Id : _Value _DElem*] _DRec*) = _Value}$$

$$[lo4] \frac{\text{is-ref}(_Value) = \text{FALSE}}{\text{lookup}(_Id, _Int+, [_Id', _Path, _Id : _Value _DElem*] _DRec*) = \text{lookup-list}(_Int+, _Value)}$$

$$[lo5] \text{lookup}(_Id, , [_Id', _Path, _Id : \text{ref}(_Id'', _Path') _DElem*] _DRec*) = \text{lookup}(_Id'', _Path', _DRec*)$$

$$[lo6] \text{lookup}(_Id, _Int+, [_Id', _Path, _Id : \text{ref}(_Id'',) _DElem*] _DRec*) = \text{lookup}(_Id'', _Int+, _DRec*)$$

$$[ll1] \text{lookup-list}(_Int, [_Int, _High, _Value _Value*]) = _Value$$

$$[112] \text{lookup-list}(_Int _Int+, [_Low, _High, _Value*]) = \text{lookup-list}(_Int+, \text{lookup-list}(_Int, [_Low, _High, _Value*]))$$

$$[113] \frac{_Int \neq _Low}{\text{lookup-list}(_Int, [_Low, _High, _Value _Value*]) = \text{lookup-list}(_Int, [\text{incr}(_Low), _High, _Value*])}$$

E.2 Evaluation of Expressions

In module EvalExpr, the evaluation of an expression in a given data stack is specified. Below, the **exports** section of this module is shown.

```
eval-exp(EXPR, D-STACK)      → EXPR
get-index-list(VARIABLE, D-STACK) → PATH
get-id(VARIABLE)             → ID
```

Eval-exp defines the evaluation of an expression, given a data stack. Before describing this function in detail, we comment on the other two exported functions which are auxiliary functions for dealing with variables. Get-index-list computes a path denoting the actual indices, given a variable (represented by a term of sort VAR) and a data stack. For example, for a non-array variable get-index-list computes the empty path, and in a situation where variable i has value 1, get-index-list computes the path (1 2) for array variable $a[i][i+i]$. In equations [gil1] and [gil2] below, get-index-list is recursively defined in terms of eval-exp. Get-id is a function which retrieves the identifier of a variable.

$$[gil1] \text{get-index-list}(_Id, _DStack) =$$

$$[gil2] \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Int, \\ \text{get-index-list}(_Var, _DStack) = _Int* \end{array}}{\text{get-index-list}(_Var [_Exp], _DStack) = _Int* _Int}$$

At this point, we describe the most significant issues which arise in the specification of eval-exp. The full definition can be found in Appendix F.3. The equations which define eval-exp can be classified roughly as follows:

1. equations for evaluating constants
2. equations for evaluating variables
3. equations for simplifying complex expressions
4. equations for basic computations on simplified expressions
5. equations for auxiliary notions used in 1–4

Below, some examples of each of these classes are studied.

Evaluating Constants

Constants can be evaluated without any access to the data stack. As an example, equation [ee3] for evaluating real constants (represented by sort REAL-CONST) is shown. The variables *_Real* and *_DStack* are of sorts REAL-CONST and D-STACK, respectively.

$$[\text{ee3}] \text{ eval-exp}(_Real, _DStack) = _Real$$

Evaluating Variables

Evaluating an expression which consists of a single variable corresponds to doing a lookup for that variable in the current D-STACK; this is expressed in equation [ee4] below. The previously described functions *get-index-list* and *get-id* are used in order to retrieve the index list and the identifier of the variable.

$$[\text{ee4}] \frac{\begin{array}{l} _Int* = \text{get-index-list}(_Var, _DStack), \\ _Id = \text{get-id}(_Var) \end{array}}{\text{eval-exp}(_Var, _DStack) = \text{lookup}(_Id, _Int*, _DStack)}$$

Simplifying Complex Expressions

Equations [ee5]—[ee20] deal with simplifying expressions by transforming (combinations of) operators into other operators, or by removing redundant operators. Two examples are shown: in equation [ee13] an addition in combination with a unary minus is transformed into a subtraction, and in [ee20] the % operator is defined in terms of other operators.

$$[\text{ee13}] \text{ eval-exp}(_Exp + - _Exp', _DStack) = \text{eval-exp}(_Exp - _Exp', _DStack)$$

$$[\text{ee20}] \text{ eval-exp}(_Exp \% _Exp', _DStack) = \text{eval-exp}(_Exp - \text{eval-exp}(\text{eval-exp}(_Exp / _Exp', _DStack) * _Exp', _DStack), _DStack)$$

Evaluating Simplified Expressions

Equations [ee21]—[ee47] describe the evaluation of expressions which are “simplified”: operands are values of sort INT-CONST, REAL-CONST or BOOL-CONST.

In equation [ee28] shown below, the evaluation of the multiplication of two real constants is expressed. The two conditions of this equation force the simplification of the operands, by recursive calls to *eval-exp*. The auxiliary function *hybrid-real-mul*, which will be discussed shortly, performs the actual multiplication of two real constants. As another example, equation [ee37] is shown. Here, the relational operator *<* is defined by way of an auxiliary function *>0* which decides whether or not an expression represents a value greater than zero.

$$[ee28] \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Real, \\ \text{eval-exp}(_Exp', _DStack) = _Real' \end{array}}{\text{eval-exp}(_Exp * _Exp', _DStack) = \text{hybrid-real-mul}(_Real, _Real')}$$

$$[ee37] \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Real, \\ \text{eval-exp}(_Exp', _DStack) = _Real', \\ _Real'' = \text{hybrid-real-sub}(_Real', _Real) \end{array}}{\text{eval-exp}(_Exp < _Exp', _DStack) = >0 (_Real'')}$$

Auxiliary Notions

The following kinds of auxiliary functions occur in module EvalExpr:

- operations on boolean constants
- auxiliary notions >0 and ≥ 0 used for defining relational operators
- arithmetic operations on integer and real constants

The arithmetic operations are not defined in module EvalExp, but in module Arithmetic (see Appendix F.8). For efficiency reasons, hybrid functions are used for integer and real arithmetic: the actual computations are done in LeLisp [LeL90]. Details of this feature are discussed in [Wal91]. Module Arithmetic provides functions for integer and real addition, subtraction, multiplication, division, and for computing the absolute value of integer and real constants.

The operations on booleans are straightforward; the reader is referred to Appendix F.3 for details.

The definition of the function > 0 makes use of the automatically generated lexical constructor functions `int-const` and `real-const`, allowing access to the individual characters of terms belonging to lexical sorts (see [HHKR89]). Equations [gt1]—[gt4] handle the cases in which the result is the boolean constant `FALSE`. This is true when the result of evaluating the expression is any constant starting with a minus sign, or the (integer or real) constant zero. Equations [gt5] and [gt6] describe the cases where the argument is a positive integer or real value, respectively. Hybrid functions are used here to check whether a number is greater than, or equal to zero (by checking if it is equal to its absolute value).

$$\begin{array}{l} [gt1] >0 (\text{int-const}(\text{"-"} _Char+)) = \text{FALSE} \\ [gt2] >0 (\text{real-const}(\text{"-"} _Char+)) = \text{FALSE} \\ [gt3] >0 (\text{int-const}(\text{"0"})) = \text{FALSE} \\ [gt4] >0 (\text{real-const}(\text{"0"} \text{"."} \text{"0"})) = \text{FALSE} \end{array}$$

$$[gt5] \frac{\begin{array}{l} _Int = \text{hybrid-int-abs}(_Int), \\ _Int \neq \text{int-const}(\text{"0"}) \end{array}}{>0 (_Int) = \text{TRUE}}$$

$$[\text{gt6}] \frac{\begin{array}{l} _Real = \text{hybrid-real-abs}(_Real), \\ _Real \neq \text{int-const}("0" \text{ "}" "0") \end{array}}{>0 (_Real) = \text{TRUE}}$$

E.3 Execution of Programs and Statements

Execution of Programs

The execution of a CLaX program is defined by equation [ep1] of module EvalProgram below (see Appendix F.4). The function `eval-program` applied to a CLaX program computes a list of data stack elements (sort D-ELEMS) containing the values for the global variables of the program. The output generated by WRITE statements is handled as a side-effect, and is not reflected by this list of variable-value pairs. Several auxiliary notions are used in the definition of `eval-program`. First, `get-seq` is a simple access function which retrieves the sequence of statements in a block. Second, `init` is a function which computes the initial values in a data record. As arguments it takes a block, a list of actual parameters (here: empty), a list of formal parameters (here: empty), the previous data stack (here: the empty stack), and the path corresponding to the current scope (see E.1). The definition of `init` can be found in Appendix F.5. Third, a function `eval` is used which transforms a status into another status (by executing a statement). Below, the variables `_CRec` and `_DRec` correspond to the initial code record and data record on the respective stacks.

$$[\text{ep1}] \frac{\begin{array}{l} _Seq = \text{get-seq}(_Block), \\ _CRec = [_Id, _Seq, _Seq], \\ _DElem* = \text{init}(_Block, , , ,), \\ _DRec = [_Id, , _DElem*] \end{array}}{\text{eval-program}(\text{PROGRAM } _Id; _Block .) = \text{eval}(\langle _CRec, _DRec \rangle)}$$

Execution of Statements

As mentioned, `eval` computes the effect on the status of executing a single statement; `eval` is defined by equations [ev1]—[ev18] of module EvalProgram (see Appendix F.4). Below, we will describe a few of these equations in some detail.

[ev1] handles the case where no more statements have to be executed—the values in the final data record are the result of executing the program. The function `values` is an auxiliary function which throws away all procedure definition elements from a data record. Equation [ev2] is very similar; it handles the case in which no more code is to be executed in the current procedure. The procedure call statement in the code record for the calling procedure is removed, and execution proceeds with the next statement in that record.

$$[\text{ev1}] \text{eval}(\langle [_Id, _Seq,], [_Id, , _DElem*] \rangle) = \text{values}(_DElem*)$$

$$[\text{ev2}] \text{eval}(\langle [_Id, _Seq,] [_Id', _Seq', _Stat; _Stat*] _CRec*, _DRec _DRec+ \rangle) = \\ \text{eval}(\langle [_Id', _Seq', _Stat*] _CRec*, _DRec+ \rangle)$$

The execution of an assignment statement $_Var := _Exp$ is specified by equation [ev4] below. First the index list and the identifier of $_Var$ are determined (see E.2). Then, the expression is evaluated and the resulting value is bound to the variable $_Value$. Subsequently, the value is converted from sort INT-CONST to sort REAL-CONST if necessary; this is expressed by way of two auxiliary functions `type` and `convert` (not shown). Finally, the updated data stack is computed, using the (converted) value $_Value'$, and a new status is constructed by discarding the assignment statement, and replacing the old data stack by the new data stack.

$$\begin{array}{l}
 _Int* = \text{get-index-list}(_Var, _DStack), \\
 _Id' = \text{get-id}(_Var), \\
 _Value = \text{eval-exp}(_Exp, _DStack), \\
 _Type = \text{type}(_Id', _Int*, _DStack), \\
 _Value' = \text{convert}(_Type, _Value), \\
 _DStack' = \text{update}(_Id', _Int*, _DStack, _Value') \\
 \hline
 \text{[ev4]} \quad \frac{\text{eval}(\langle [_Id, _Seq, _Var := _Exp; _Stat*] _CRec*, _DStack \rangle)}{\text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DStack' \rangle)} =
 \end{array}$$

The execution of an IF-THEN-ELSE statement is defined by equations [ev7] and [ev8]. Depending on the result of evaluating the predicate $_Exp$, resulting in TRUE or FALSE, the statement is replaced by either the THEN branch $_Stat*'$, or the ELSE branch $_Stat*''$.

$$\begin{array}{l}
 \text{eval-predicate}(_Exp, _DStack) = \text{TRUE} \\
 \hline
 \text{[ev7]} \quad \frac{\text{eval}(\langle [_Id, _Seq, \text{IF } _Exp \text{ THEN } _Stat*' \text{ ELSE } _Stat*'' \text{ END; } _Stat*] _CRec*, _DStack \rangle)}{\text{eval}(\langle [_Id, _Seq, _Stat*'; _Stat*] _CRec*, _DStack \rangle)} =
 \end{array}$$

$$\begin{array}{l}
 \text{eval-predicate}(_Exp, _DStack) = \text{FALSE} \\
 \hline
 \text{[ev8]} \quad \frac{\text{eval}(\langle [_Id, _Seq, \text{IF } _Exp \text{ THEN } _Stat*' \text{ ELSE } _Stat*'' \text{ END; } _Stat*] _CRec*, _DStack \rangle)}{\text{eval}(\langle [_Id, _Seq, _Stat*''; _Stat*] _CRec*, _DStack \rangle)} =
 \end{array}$$

The execution of a procedure call is expressed by equations [ev15] and [ev16]. The former handles procedure calls without parameters, whereas the latter handles calls with parameters. In the first condition, the definition of the procedure is retrieved, by means of an auxiliary function `lookup-proc` (see Appendix F.2). Next, a new data record is created, using `init` (see the description of equation [ep1], above). Finally, a new code record is created, and the execution continues with the extended code and data stacks. Observe, that the call statement itself is not removed at this point, but upon return from the procedure—this was described earlier (equation [ev2]).

$$\begin{array}{c}
\text{lookup-proc}(_Id', _DRec*) = _Id''' : _Path : \text{PROCEDURE } _Id''; _Block, \\
\quad _DElem* = \text{init}(_Block, , , _Path), \\
\quad _Seq' = \text{get-seq}(_Block), \\
\quad _DRec' = [_Id'', _Path, _DElem*], \\
\quad _CRec' = [_Id'', _Seq', _Seq'] \\
\hline
[\text{ev15}] \quad \text{eval}(\langle [_Id, _Seq, _Id'; _Stat*] _CRec*, _DRec* \rangle) = \\
\quad \text{eval}(\langle _CRec' [_Id, _Seq, _Id'; _Stat*] _CRec*, _DRec' _DRec* \rangle) \\
\\
\text{lookup-proc}(_Id', _DRec*) = _Id''' : _Path : \text{PROCEDURE } _Id'' (_Formal+); _Block, \\
\quad _DElem* = \text{init}(_Block, _Actual+, _Formal+, _DRec*, _Path), \\
\quad _Seq' = \text{get-seq}(_Block), \\
\quad _DRec' = [_Id'', _Path, _DElem*], \\
\quad _CRec' = [_Id'', _Seq', _Seq'] \\
\hline
[\text{ev16}] \quad \text{eval}(\langle [_Id, _Seq, _Id' (_Actual+); _Stat*] _CRec*, _DRec* \rangle) = \\
\quad \text{eval}(\langle _CRec' [_Id, _Seq, _Id' (_Actual+); _Stat*] _CRec*, _DRec' _DRec* \rangle)
\end{array}$$

Finally, we show equation [ev11] which specifies the executing of a statement GOTO *_Label*. The condition of [ev11] uses function find-label (see Appendix F.1) to extract a sequence of statements from *_Seq* which starts with the statement labeled with *_Label*. Executing the GOTO corresponds to substituting this list for the list of statements to be executed.

$$\begin{array}{c}
\quad _Seq' = \text{find-label}(_Label, _Seq) \\
\hline
[\text{ev11}] \quad \text{eval}(\langle [_Id, _Seq, \text{GOTO } _Label; _Stat*] _CRec*, _DStack \rangle) = \\
\quad \text{eval}(\langle [_Id, _Seq, _Seq'] _CRec*, _DStack \rangle)
\end{array}$$

E.4 Input/Output

In CLaX, input and output is performed by way of READ and WRITE statements, respectively. In our specification, hybrid functions are used to implement these statements. Output is written to the ASF+SDF System's output window. Input is handled in a more elegant manner, by re-using the system's generic text and structure editor, GSE [Koo92]. Whenever a READ statement is executed, an instance of this editor pops up asking the user to 'fill in' a placeholder of the appropriate type. After entering this value, the input will be parsed. In case of a failing parse, the user is notified of this fact; execution will be suspended until a value of the correct sort is entered. In Figure 7 below, a snapshot is shown in which the user is asked to enter a value of type REAL.

Input

The CLaX READ statement is defined by equation [ev17]. Notice the similarity between this equation, and equation [ev4] for an assignment. The actual reading of the value is done by function read-value. This function is implemented by a hybrid equation which calls Lisp to read a value of the appropriate type *_Type* (see Appendix F.6).

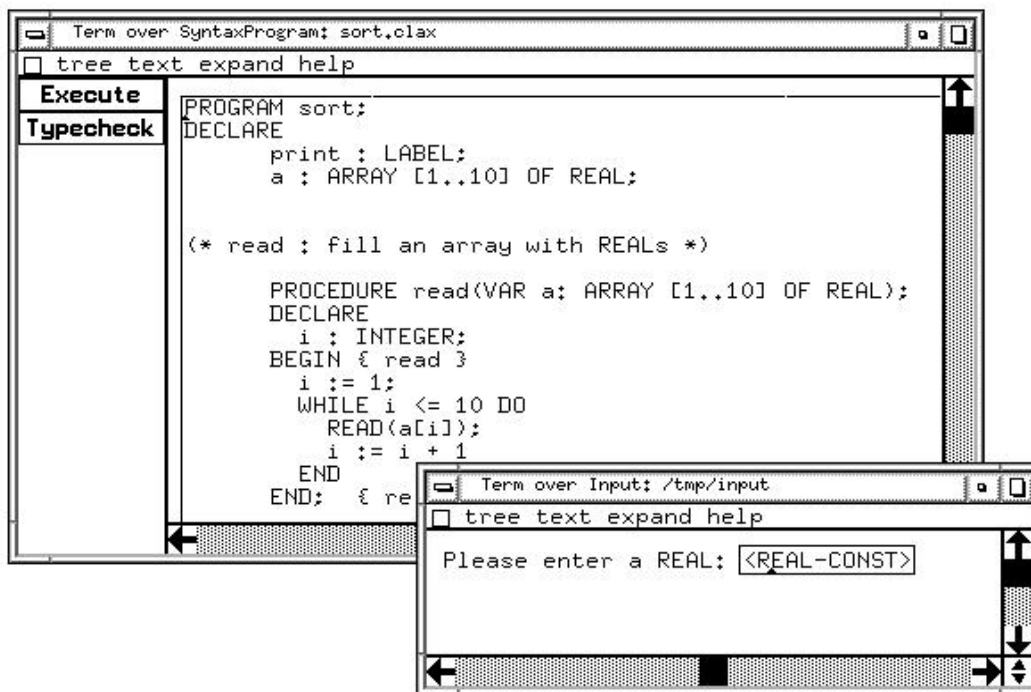


Figure 7: Interactive input in the CLaX environment.

$$\begin{array}{l}
 _Int* = \text{get-index-list}(_Var, _DRec*), \\
 _Id' = \text{get-id}(_Var), \\
 _Type = \text{type}(_Id', _Int*, _DRec*), \\
 _Value' = \text{read-value}(_Type), \\
 _DRec*' = \text{update}(_Id', _Int*, _DRec*, _Value') \\
 \hline
 \text{[ev17] } \text{eval}(\langle [_Id, _Seq, \text{READ}(_Var); _Stat*] _CRec*, _DRec* \rangle) = \\
 \text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DRec*' \rangle)
 \end{array}$$

Output

In the specification, the effect of a WRITE statement on the computed values is nil. As a side-effect, however, a value or string is printed in the LeLisp window. This is expressed in equations [ev12]—[ev14] below. The conditions $_Dummy = \text{emit}(_Value)$ and $_Dummy = \text{emits-string}(_String)$ generate the desired side-effect. The functions `emit` and `emit-string` are specified as hybrid equations containing calls to Lisp for printing the values. Observe that in the case of writing booleans, a conversion to the external display format is performed. For details, the reader is referred to Appendix F.7.

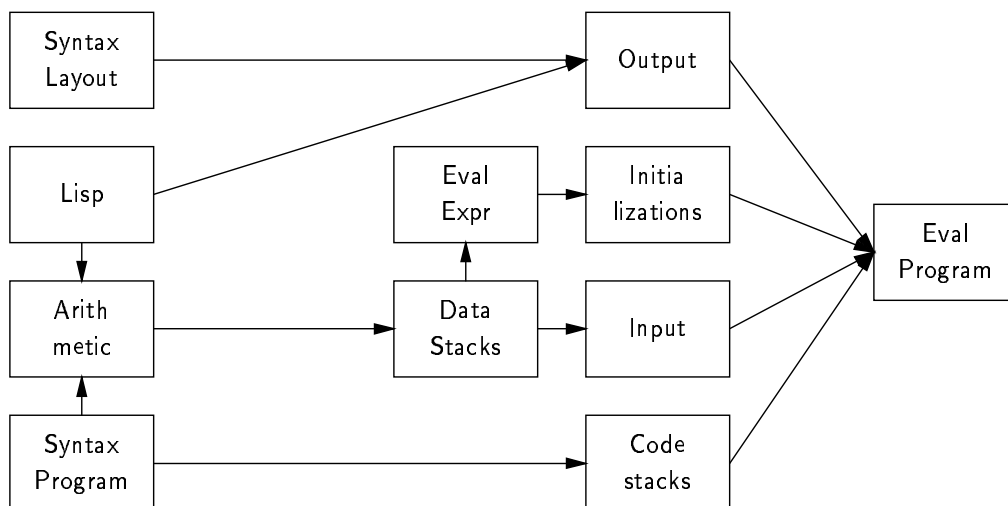
$$\begin{array}{c}
_Bool = \text{eval-exp}(_Exp, _DStack), \\
_Dummy = \text{emit-string}(\text{external-representation}(_Bool)) \\
\hline
[\text{ev12}] \quad \text{eval}(\langle [_Id, _Seq, \text{WRITE}(_Exp); _Stat*] _CRec*, _DStack \rangle) = \\
\text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DStack \rangle)
\end{array}$$

$$\begin{array}{c}
_Value = \text{eval-exp}(_Exp, _DStack), \\
_Dummy = \text{emit}(_Value) \\
\hline
[\text{ev13}] \quad \text{eval}(\langle [_Id, _Seq, \text{WRITE}(_Exp); _Stat*] _CRec*, _DStack \rangle) = \\
\text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DStack \rangle)
\end{array}$$

$$\begin{array}{c}
_Dummy = \text{emit-string}(_String) \\
\hline
[\text{ev14}] \quad \text{eval}(\langle [_Id, _Seq, \text{WRITE}(_String); _Stat*] _CRec*, _DStack \rangle) = \\
\text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DStack \rangle)
\end{array}$$

F CLaX interpreter modules

The specification below is the full text of the CLaX interpreter. It imports the CLaX syntax defined in module `SyntaxProgram` (see Appendix B). The import diagram for the interpreter modules is:



F.1 module CodeStacks

module CodeStacks

imports `SyntaxProgram`

exports

sorts C-RECORD C-STACK

context-free functions

[" ID ";" {STAT ";"}* ";" {STAT ";"}* "]" → C-RECORD
 C-RECORD* → C-STACK
 find-label(LABEL, STAT-SEQ) → STAT-SEQ

variables

[_] CRec [']* → C-RECORD
 [_] CRec [*] [']* → C-RECORD*
 [_] CRec [+] [']* → C-RECORD+
 [_] CStack [']* → C-STACK
 [_] Stat [*] [']* → {STAT ";"}*
 [_] Stat [+] [']* → {STAT ";"}+

hiddens

sorts FL

context-free functions

f1(LABEL, STAT-SEQ) → FL
 [" BOOL-CONST ";" STAT-SEQ "]" → FL

equations

$$\begin{array}{l}
\text{[fl1]} \quad \frac{\text{fl}(_Label, _Stat*) = [\text{TRUE}, _Stat*']}{\text{find-label}(_Label, _Stat*) = _Stat*'} \\
\text{[fl2]} \quad \frac{_Label = _Label'}{\text{fl}(_Label, _Label' : _StatAux; _Stat*) = [\text{TRUE}, _Label' : _StatAux; _Stat*]} \\
\text{[fl3]} \quad \frac{_Label \neq _Label'}{\text{fl}(_Label, _Label' : _StatAux; _Stat*) = \text{fl}(_Label, _StatAux; _Stat*)} \\
\text{[fl4]} \quad \text{fl}(_Label, ; _Stat*) = \text{fl}(_Label, _Stat*) \\
\text{[fl5]} \quad \text{fl}(_Label, \text{GOTO } _Label'; _Stat*) = \text{fl}(_Label, _Stat*) \\
\text{[fl6]} \quad \text{fl}(_Label, _Var := _Expr; _Stat*) = \text{fl}(_Label, _Stat*) \\
\text{[fl7]} \quad \frac{\text{fl}(_Label, _Stat*') = [\text{FALSE}, _Stat*']}{\text{fl}(_Label, \text{WHILE } _Expr \text{ DO } _Stat*' \text{ END}; _Stat*) = \text{fl}(_Label, _Stat*)} \\
\text{[fl8]} \quad \frac{\text{fl}(_Label, _Stat*') = [\text{TRUE}, _Stat+]}{\text{fl}(_Label, \text{WHILE } _Expr \text{ DO } _Stat*' \text{ END}; _Stat*) = [\text{TRUE}, _Stat+; \text{WHILE } _Expr \text{ DO } _Stat*' \text{ END}; _Stat*]} \\
\text{[fl9]} \quad \frac{\text{fl}(_Label, _Stat*') = [\text{FALSE}, _Stat*']}{\text{fl}(_Label, \text{IF } _Expr \text{ THEN } _Stat*' \text{ END}; _Stat*) = \text{fl}(_Label, _Stat*)} \\
\text{[fl10]} \quad \frac{\text{fl}(_Label, _Stat*') = [\text{TRUE}, _Stat+]}{\text{fl}(_Label, \text{IF } _Expr \text{ THEN } _Stat*' \text{ END}; _Stat*) = [\text{TRUE}, _Stat+; _Stat*]} \\
\text{[fl11]} \quad \frac{\begin{array}{l} \text{fl}(_Label, _Stat*') = [\text{FALSE},], \\ \text{fl}(_Label, _Stat*') = [\text{FALSE},] \end{array}}{\text{fl}(_Label, \text{IF } _Expr \text{ THEN } _Stat*' \text{ ELSE } _Stat*'' \text{ END}; _Stat*) = \text{fl}(_Label, _Stat*)} \\
\text{[fl12]} \quad \frac{\text{fl}(_Label, _Stat*') = [\text{TRUE}, _Stat+]}{\text{fl}(_Label, \text{IF } _Expr \text{ THEN } _Stat*' \text{ ELSE } _Stat*'' \text{ END}; _Stat*) = [\text{TRUE}, _Stat+; _Stat*]} \\
\text{[fl13]} \quad \frac{\text{fl}(_Label, _Stat*') = [\text{TRUE}, _Stat+]}{\text{fl}(_Label, \text{IF } _Expr \text{ THEN } _Stat*' \text{ ELSE } _Stat*'' \text{ END}; _Stat*) = [\text{TRUE}, _Stat+; _Stat*]} \\
\text{[fl14]} \quad \frac{_Stat* = _Stat*; _Stat*}{\text{fl}(_Label, _Stat*) = [\text{FALSE},]}
\end{array}$$

F.2 module DataStacks

module DataStacks

imports Arithmetic

exports

sorts VALUE PATH D-ELEM D-ELEMS D-RECORD D-STACK

context-free functions

EXPR	→ VALUE
“[” INT-CONST “,” INT-CONST “,” VALUE+ “]”	→ VALUE
ref(ID, PATH)	→ VALUE
INT-CONST*	→ PATH
ID “:” VALUE	→ D-ELEM
ID “:” PATH “:” PROC-DECL	→ D-ELEM
D-ELEM*	→ D-ELEMS
“[” ID “,” PATH “,” D-ELEM* “]”	→ D-RECORD
D-RECORD*	→ D-STACK
type(ID, PATH, D-STACK)	→ TYPE
lookup(ID, PATH, D-STACK)	→ VALUE
update(ID, PATH, D-STACK, VALUE)	→ D-STACK
lookup-proc(ID, D-STACK)	→ D-ELEM
get-name(PROC-DECL)	→ ID

variables

[_] <i>Id</i> [']*	→ ID
[_] <i>Int</i> [']*	→ INT-CONST
[_] <i>Real</i> [']*	→ REAL-CONST
[_] <i>Bool</i> [']*	→ BOOL-CONST
“_Low” [']*	→ INT-CONST
“_High” [']*	→ INT-CONST
[_] <i>Int</i> [*] [']*	→ INT-CONST*
[_] <i>Int</i> [+] [']*	→ INT-CONST+
[_] <i>Exp</i> [']*	→ EXPR
[_] <i>Exp</i> [*] [']*	→ {EXPR “,”}*
[_] <i>Seq</i> [']*	→ {STAT “,”}*
[_] <i>ProcDecl</i>	→ PROC-DECL
[_] <i>Formal</i> [+] [']*	→ {FORMAL “;”}+
[_] <i>Block</i>	→ BLOCK
[_] <i>Value</i> [']*	→ VALUE
[_] <i>Value</i> [*] [']*	→ VALUE*
[_] <i>Value</i> [+] [']*	→ VALUE+
[_] <i>Path</i> [']*	→ PATH
[_] <i>DElem</i> [']*	→ D-ELEM
[_] <i>DElem</i> [*] [']*	→ D-ELEM*
[_] <i>DRec</i> [']*	→ D-RECORD
[_] <i>DRec</i> [*] [']*	→ D-RECORD*
[_] <i>DRec</i> [+] [']*	→ D-RECORD+
[_] <i>DStack</i> [']*	→ D-STACK

hiddens

context-free functions

lookup-list(PATH, VALUE)	→ VALUE
update-list(PATH, VALUE, VALUE)	→ VALUE
next-scope(D-STACK, PATH)	→ D-STACK
get-id(D-ELEM)	→ ID
get-value(D-ELEM)	→ VALUE
is-ref(VALUE)	→ BOOL-CONST

equations

$$[\text{sn1}] \quad \text{next-scope}([_Id, _Path, _DElem*] _DRec*, _Path) = [_Id, _Path, _DElem*] _DRec*$$

$$[\text{sn2}] \quad \frac{_Path \neq _Path'}{\text{next-scope}([_Id, _Path', _DElem*] _DRec*, _Path) = \text{next-scope}(_DRec*, _Path)}$$

$$[\text{lo1}] \quad \frac{_DRec*' = \text{next-scope}(_DRec*, _Int*)}{\text{lookup}(_Id, _Path, [_Id', _Int* _Int,] _DRec*) = \text{lookup}(_Id, _Path, _DRec*')}$$

$$[\text{lo2}] \quad \frac{_Id \neq \text{get-id}(_DElem)}{\text{lookup}(_Id, _Path, [_Id', _Path', _DElem _DElem*] _DRec*) = \text{lookup}(_Id, _Path, [_Id', _Path', _DElem*] _DRec*)}$$

$$[\text{lo3}] \quad \frac{\text{is-ref}(_Value) = \text{FALSE}}{\text{lookup}(_Id, , [_Id', _Path, _Id : _Value _DElem*] _DRec*) = _Value}$$

$$[\text{lo4}] \quad \frac{\text{is-ref}(_Value) = \text{FALSE}}{\text{lookup}(_Id, _Int+, [_Id', _Path, _Id : _Value _DElem*] _DRec*) = \text{lookup-list}(_Int+, _Value)}$$

$$[\text{lo5}] \quad \text{lookup}(_Id, , [_Id', _Path, _Id : \text{ref}(_Id'', _Path') _DElem*] _DRec*) = \text{lookup}(_Id'', _Path', _DRec*)$$

$$[\text{lo6}] \quad \text{lookup}(_Id, _Int+, [_Id', _Path, _Id : \text{ref}(_Id'',) _DElem*] _DRec*) = \text{lookup}(_Id'', _Int+, _DRec*)$$

$$[\text{ll1}] \quad \text{lookup-list}(_Int, [_Int, _High, _Value _Value*]) = _Value$$

$$[\text{ll2}] \quad \text{lookup-list}(_Int _Int+, [_Low, _High, _Value*]) = \text{lookup-list}(_Int+, \text{lookup-list}(_Int, [_Low, _High, _Value*]))$$

$$\begin{array}{l}
\text{[ll3]} \frac{_Int \neq _Low}{\text{lookup-list}(_Int, [_Low, _High, _Value _Value*]) = \text{lookup-list}(_Int, [\text{incr}(_Low), _High, _Value*])} \\
\\
\text{[up1]} \frac{\begin{array}{l} _DRec*' = \text{next-scope}(_DRec*, _Int*), \\ _DRec* = _DRec*' _DRec*'', \\ _DRec*''' = \text{update}(_Id, _Path, _DRec*'', _Value) \end{array}}{\text{update}(_Id, _Path, [_Id', _Int* _Int,] _DRec*, _Value) = [_Id', _Int* _Int,] _DRec*' _DRec*'''} \\
\\
\text{[up2]} \frac{\begin{array}{l} _Id \neq \text{get-id}(_DElem), \\ \text{update}(_Id, _Path, [_Id', _Path', _DElem*] _DRec*, _Value) = \\ [_Id', _Path', _DElem*'] _DRec*' \end{array}}{\text{update}(_Id, _Path, [_Id', _Path', _DElem _DElem*] _DRec*, _Value) = [_Id', _Path', _DElem _DElem*'] _DRec*' } \\
\\
\text{[up3]} \frac{\text{is-ref}(_Value) = \text{FALSE}}{\text{update}(_Id, , [_Id', _Path, _Id : _Value _DElem*] _DRec*, _Value') = [_Id', _Path, _Id : _Value' _DElem*] _DRec*} \\
\\
\text{[up4]} \frac{\begin{array}{l} \text{is-ref}(_Value) = \text{FALSE}, \\ _Value'' = \text{update-list}(_Int+, _Value, _Value') \end{array}}{\text{update}(_Id, _Int+, [_Id', _Path, _Id : _Value _DElem*] _DRec*, _Value') = [_Id', _Path, _Id : _Value'' _DElem*] _DRec*} \\
\\
\text{[up5]} \frac{\text{update}(_Id', _Path', _DRec*, _Value) = _DRec*'}{\text{update}(_Id, , [_Id', _Path, _Id : \text{ref}(_Id', _Path') _DElem*] _DRec*, _Value) = [_Id', _Path, _Id : \text{ref}(_Id', _Path') _DElem*] _DRec*' } \\
\\
\text{[up6]} \frac{\text{update}(_Id', _Int+, _DRec*, _Value) = _DRec*'}{\text{update}(_Id, _Int+, [_Id', _Path, _Id : \text{ref}(_Id',) _DElem*] _DRec*, _Value) = [_Id', _Path, _Id : \text{ref}(_Id',) _DElem*] _DRec*' } \\
\\
\text{[ul1]} \text{update-list}(_Low, [_Low, _High, _Value _Value*], _Value') = [_Low, _High, _Value' _Value*]
\end{array}$$

$$\begin{array}{c}
_Int \neq _Low, \\
_Low' = \text{incr}(_Low), \\
\text{update-list}(_Int, [_Low', _High, _Value*], _Value') = [_Low', _High, _Value*'] \\
\hline
\text{update-list}(_Int, [_Low, _High, _Value _Value*], _Value') = \\
[_Low, _High, _Value _Value*']
\end{array}$$

$$\begin{array}{c}
\text{update-list}(_Low _Int+, [_Low, _High, _Value _Value*], _Value') = \\
[_Low, _High, \text{update-list}(_Int+, _Value, _Value') _Value*]
\end{array}$$

$$\begin{array}{c}
_Int \neq _Low, \\
_Low' = \text{incr}(_Low), \\
\text{update-list}(_Int _Int+, [_Low', _High, _Value*], _Value') = [_Low', _High, _Value*'] \\
\hline
\text{update-list}(_Int _Int+, [_Low, _High, _Value _Value*], _Value') = \\
[_Low, _High, _Value _Value*']
\end{array}$$

$$\begin{array}{c}
_DRec*' = \text{next-scope}(_DRec*, _Int*) \\
\hline
\text{lookup-proc}(_Id, [_Id', _Int* _Int,] _DRec*) = \text{lookup-proc}(_Id, _DRec*')
\end{array}$$

$$\begin{array}{c}
_Id \neq \text{get-id}(_DElem) \\
\hline
\text{lookup-proc}(_Id, [_Id', _Path, _DElem _DElem*] _DRec*) = \\
\text{lookup-proc}(_Id, [_Id', _Path, _DElem*] _DRec*)
\end{array}$$

$$\begin{array}{c}
\text{lookup-proc}(_Id, [_Id', _Path, _Id : _Path' : _ProcDecl _DElem*] _DRec*) = \\
_Id : _Path' : _ProcDecl
\end{array}$$

$$\begin{array}{c}
\text{lookup}(_Id, _Path, _DStack) = _Int \\
\hline
\text{type}(_Id, _Path, _DStack) = \text{INTEGER}
\end{array}$$

$$\begin{array}{c}
\text{lookup}(_Id, _Path, _DStack) = _Real \\
\hline
\text{type}(_Id, _Path, _DStack) = \text{REAL}
\end{array}$$

$$\begin{array}{c}
\text{lookup}(_Id, _Path, _DStack) = _Bool \\
\hline
\text{type}(_Id, _Path, _DStack) = \text{BOOLEAN}
\end{array}$$

$$[\text{ir1}] \text{is-ref}(\text{ref}(_Id, _Path)) = \text{TRUE}$$

$$[\text{ir2}] \text{is-ref}(_Expr) = \text{FALSE}$$

$$[\text{ir3}] \text{is-ref}([_Low, _High, _Value+]) = \text{FALSE}$$

$$[\text{gn1}] \text{get-name}(\text{PROCEDURE } _Id; _Block) = _Id$$

$$[\text{gn2}] \text{get-name}(\text{PROCEDURE } _Id (_Formal+); _Block) = _Id$$

$$[\text{gi1}] \text{get-id}(_Id : _Value) = _Id$$

$$[\text{gi2}] \text{get-id}(_Id : _Path : _ProcDecl) = _Id$$

F.3 module EvalExpr

module EvalExpr

imports DataStacks

exports

context-free functions

eval-exp(EXPR, D-STACK) → EXPR
get-index-list(VARIABLE, D-STACK) → PATH
get-id(VARIABLE) → ID

variables

[_] Bool [*'*]* → BOOL-CONST
[,] Int [*'*]* → INT-CONST
[,] Real [*'*]* → REAL-CONST
[,] Var [*'*]* → VARIABLE
[,] Exp [*'*]* → EXPR
[,] Lisp [*'*]* → LISP
[,] Char [+]*'** → CHAR+
[,] Char [*]*'** → CHAR*
[,] Char [*'*]* → CHAR

hiddens

context-free functions

">0" "(" EXPR ")" → BOOL-CONST
">=0" "(" EXPR ")" → BOOL-CONST

bool-lt(BOOL-CONST, BOOL-CONST) → BOOL-CONST
bool-le(BOOL-CONST, BOOL-CONST) → BOOL-CONST
bool-and(BOOL-CONST, BOOL-CONST) → BOOL-CONST
bool-or(BOOL-CONST, BOOL-CONST) → BOOL-CONST
bool-not(BOOL-CONST) → BOOL-CONST

equations

[ee1] eval-exp(*_Bool*, *_DStack*) = *_Bool*
[ee2] eval-exp(*_Int*, *_DStack*) = *_Int*
[ee3] eval-exp(*_Real*, *_DStack*) = *_Real*

$$\text{[ee4] } \frac{\begin{array}{l} \textit{_Int*} = \text{get-index-list}(\textit{_Var}, \textit{_DStack}), \\ \textit{_Id} = \text{get-id}(\textit{_Var}) \end{array}}{\text{eval-exp}(\textit{_Var}, \textit{_DStack}) = \text{lookup}(\textit{_Id}, \textit{_Int*}, \textit{_DStack})}$$

[ee5] - - *_Exp* = *_Exp*
[ee6] - int-const("-" *_Char+*) = int-const(*_Char+*)
[ee7] int-const("+" *_Char+*) = int-const(*_Char+*)

$$\text{[ee8] } \frac{\text{int-const}(\textit{_Char} \textit{"1"}) \neq \text{int-const}(\textit{"-"} \textit{"1"})}{\text{- int-const}(\textit{_Char} \textit{_Char*}) = \text{int-const}(\textit{"-"} \textit{_Char} \textit{_Char*})}$$

[ee9] - real-const("-" *_Char+*) = real-const(*_Char+*)

$$[ee10] \text{ real-const}(\text{"+" } _Char+) = \text{real-const}(_Char+)$$

$$[ee11] \frac{\text{real-const}(_Char \text{"." "1"}) \neq \text{real-const}(\text{"-" "1" "1"})}{- \text{real-const}(_Char _Char*) = \text{real-const}(\text{"-" } _Char _Char*)}$$

$$[ee12] \text{eval-exp}(- _Exp + _Exp', _DStack) = \text{eval-exp}(_Exp' - _Exp, _DStack)$$

$$[ee13] \text{eval-exp}(_Exp + - _Exp', _DStack) = \text{eval-exp}(_Exp - _Exp', _DStack)$$

$$[ee14] \text{eval-exp}(- _Exp - _Exp', _DStack) = - \text{eval-exp}(_Exp + _Exp', _DStack)$$

$$[ee15] \text{eval-exp}(_Exp - - _Exp', _DStack) = - \text{eval-exp}(_Exp + _Exp', _DStack)$$

$$[ee16] \text{eval-exp}(- _Exp * _Exp', _DStack) = - \text{eval-exp}(_Exp * _Exp', _DStack)$$

$$[ee17] \text{eval-exp}(_Exp * - _Exp', _DStack) = - \text{eval-exp}(_Exp * _Exp', _DStack)$$

$$[ee18] \text{eval-exp}(- _Exp / _Exp', _DStack) = - \text{eval-exp}(_Exp / _Exp', _DStack)$$

$$[ee19] \text{eval-exp}(_Exp / - _Exp', _DStack) = - \text{eval-exp}(_Exp / _Exp', _DStack)$$

$$[ee20] \text{eval-exp}(_Exp \% _Exp', _DStack) = \text{eval-exp}(_Exp - \text{eval-exp}(\text{eval-exp}(_Exp / _Exp', _DStack) * _Exp', _DStack), _DStack)$$

$$[ee21] \frac{\text{eval-exp}(_Exp, _DStack) = _Bool}{\text{eval-exp}(\text{NOT } _Exp, _DStack) = \text{bool-not}(_Bool)}$$

$$[ee22] \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Bool, \\ \text{eval-exp}(_Exp', _DStack) = _Bool' \end{array}}{\text{eval-exp}(_Exp \& _Exp', _DStack) = \text{bool-and}(_Bool, _Bool')}$$

$$[ee23] \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Bool, \\ \text{eval-exp}(_Exp', _DStack) = _Bool' \end{array}}{\text{eval-exp}(_Exp | _Exp', _DStack) = \text{bool-or}(_Bool, _Bool')}$$

$$[ee24] \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Int, \\ \text{eval-exp}(_Exp', _DStack) = _Int' \end{array}}{\text{eval-exp}(_Exp * _Exp', _DStack) = \text{hybrid-int-mul}(_Int, _Int')}$$

$$[ee25] \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Int, \\ \text{eval-exp}(_Exp', _DStack) = _Int' \end{array}}{\text{eval-exp}(_Exp / _Exp', _DStack) = \text{hybrid-int-div}(_Int, _Int')}$$

$$[ee26] \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Int, \\ \text{eval-exp}(_Exp', _DStack) = _Int' \end{array}}{\text{eval-exp}(_Exp + _Exp', _DStack) = \text{hybrid-int-add}(_Int, _Int')}$$

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = _Int, \\
\text{eval-exp}(_Exp', _DStack) = _Int' \\
\hline
\text{eval-exp}(_Exp - _Exp', _DStack) = \text{hybrid-int-sub}(_Int, _Int')
\end{array}$$

[ee27]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = _Real, \\
\text{eval-exp}(_Exp', _DStack) = _Real' \\
\hline
\text{eval-exp}(_Exp * _Exp', _DStack) = \text{hybrid-real-mul}(_Real, _Real')
\end{array}$$

[ee28]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = _Real, \\
\text{eval-exp}(_Exp', _DStack) = _Real' \\
\hline
\text{eval-exp}(_Exp / _Exp', _DStack) = \text{hybrid-real-div}(_Real, _Real')
\end{array}$$

[ee29]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = _Real, \\
\text{eval-exp}(_Exp', _DStack) = _Real' \\
\hline
\text{eval-exp}(_Exp + _Exp', _DStack) = \text{hybrid-real-add}(_Real, _Real')
\end{array}$$

[ee30]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = _Real, \\
\text{eval-exp}(_Exp', _DStack) = _Real' \\
\hline
\text{eval-exp}(_Exp - _Exp', _DStack) = \text{hybrid-real-sub}(_Real, _Real')
\end{array}$$

[ee31]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = \text{eval-exp}(_Exp', _DStack) \\
\hline
\text{eval-exp}(_Exp = _Exp', _DStack) = \text{TRUE}
\end{array}$$

[ee32]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) \neq \text{eval-exp}(_Exp', _DStack) \\
\hline
\text{eval-exp}(_Exp = _Exp', _DStack) = \text{FALSE}
\end{array}$$

[ee33]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) \neq \text{eval-exp}(_Exp', _DStack) \\
\hline
\text{eval-exp}(_Exp \# _Exp', _DStack) = \text{TRUE}
\end{array}$$

[ee34]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = \text{eval-exp}(_Exp', _DStack) \\
\hline
\text{eval-exp}(_Exp \# _Exp', _DStack) = \text{FALSE}
\end{array}$$

[ee35]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = _Int, \\
\text{eval-exp}(_Exp', _DStack) = _Int', \\
_Int'' = \text{hybrid-int-sub}(_Int', _Int) \\
\hline
\text{eval-exp}(_Exp < _Exp', _DStack) = >0 (_Int'')
\end{array}$$

[ee36]

$$\begin{array}{c}
\text{eval-exp}(_Exp, _DStack) = _Real, \\
\text{eval-exp}(_Exp', _DStack) = _Real', \\
_Real'' = \text{hybrid-real-sub}(_Real', _Real) \\
\hline
\text{eval-exp}(_Exp < _Exp', _DStack) = >0 (_Real'')
\end{array}$$

[ee37]

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Int, \\
\text{eval-exp}(_Exp', _DStack) = _Int', \\
_Int'' = \text{hybrid-int-sub}(_Int', _Int) \\
\hline
\text{[ee38]} \quad \text{eval-exp}(_Exp \leq _Exp', _DStack) = \geq 0 (_Int'')
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Real, \\
\text{eval-exp}(_Exp', _DStack) = _Real', \\
_Real'' = \text{hybrid-real-sub}(_Real', _Real) \\
\hline
\text{[ee39]} \quad \text{eval-exp}(_Exp \leq _Exp', _DStack) = \geq 0 (_Real'')
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Int, \\
\text{eval-exp}(_Exp', _DStack) = _Int', \\
_Int'' = \text{hybrid-int-sub}(_Int, _Int') \\
\hline
\text{[ee40]} \quad \text{eval-exp}(_Exp > _Exp', _DStack) = > 0 (_Int'')
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Real, \\
\text{eval-exp}(_Exp', _DStack) = _Real', \\
_Real'' = \text{hybrid-real-sub}(_Real, _Real') \\
\hline
\text{[ee41]} \quad \text{eval-exp}(_Exp > _Exp', _DStack) = > 0 (_Real'')
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Int, \\
\text{eval-exp}(_Exp', _DStack) = _Int', \\
_Int'' = \text{hybrid-int-sub}(_Int, _Int') \\
\hline
\text{[ee42]} \quad \text{eval-exp}(_Exp \geq _Exp', _DStack) = \geq 0 (_Int'')
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Real, \\
\text{eval-exp}(_Exp', _DStack) = _Real', \\
_Real'' = \text{hybrid-real-sub}(_Real, _Real') \\
\hline
\text{[ee43]} \quad \text{eval-exp}(_Exp \geq _Exp', _DStack) = \geq 0 (_Real'')
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Bool, \\
\text{eval-exp}(_Exp', _DStack) = _Bool' \\
\hline
\text{[ee44]} \quad \text{eval-exp}(_Exp < _Exp', _DStack) = \text{bool-lt}(_Bool, _Bool')
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Bool, \\
\text{eval-exp}(_Exp', _DStack) = _Bool' \\
\hline
\text{[ee45]} \quad \text{eval-exp}(_Exp \leq _Exp', _DStack) = \text{bool-le}(_Bool, _Bool')
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Bool, \\
\text{eval-exp}(_Exp', _DStack) = _Bool' \\
\hline
\text{[ee46]} \quad \text{eval-exp}(_Exp > _Exp', _DStack) = \text{bool-lt}(_Bool', _Bool)
\end{array}$$

$$\begin{array}{l}
\text{eval-exp}(_Exp, _DStack) = _Bool, \\
\text{eval-exp}(_Exp', _DStack) = _Bool' \\
\hline
\text{[ee47]} \quad \text{eval-exp}(_Exp \geq _Exp', _DStack) = \text{bool-le}(_Bool', _Bool)
\end{array}$$

[blt1] $\text{bool-lt}(\text{FALSE}, \text{TRUE}) = \text{TRUE}$
 [blt2] $\text{bool-lt}(\text{TRUE}, _Bool) = \text{FALSE}$
 [blt3] $\text{bool-lt}(\text{FALSE}, \text{FALSE}) = \text{FALSE}$
 [ble1] $\text{bool-le}(\text{TRUE}, \text{FALSE}) = \text{FALSE}$
 [ble2] $\text{bool-le}(\text{TRUE}, \text{TRUE}) = \text{TRUE}$
 [ble3] $\text{bool-le}(\text{FALSE}, _Bool) = \text{TRUE}$
 [ba1] $\text{bool-and}(\text{TRUE}, _Bool) = _Bool$
 [ba2] $\text{bool-and}(\text{FALSE}, _Bool) = \text{FALSE}$
 [bo1] $\text{bool-or}(\text{TRUE}, _Bool) = \text{TRUE}$
 [bo2] $\text{bool-or}(\text{FALSE}, _Bool) = _Bool$
 [bn1] $\text{bool-not}(\text{FALSE}) = \text{TRUE}$
 [bn2] $\text{bool-not}(\text{TRUE}) = \text{FALSE}$

[gt1] $>0 (\text{int-const}(\text{"-"} _Char+)) = \text{FALSE}$
 [gt2] $>0 (\text{real-const}(\text{"-"} _Char+)) = \text{FALSE}$
 [gt3] $>0 (\text{int-const}(\text{"0"})) = \text{FALSE}$
 [gt4] $>0 (\text{real-const}(\text{"0"} \text{"."} \text{"0"})) = \text{FALSE}$

$$\text{[gt5]} \frac{\begin{array}{l} _Int = \text{hybrid-int-abs}(_Int), \\ _Int \neq \text{int-const}(\text{"0"}) \end{array}}{>0 (_Int) = \text{TRUE}}$$

$$\text{[gt6]} \frac{\begin{array}{l} _Real = \text{hybrid-real-abs}(_Real), \\ _Real \neq \text{int-const}(\text{"0"} \text{"."} \text{"0"}) \end{array}}{>0 (_Real) = \text{TRUE}}$$

[ge1] $\geq 0 (\text{int-const}(\text{"-"} _Char+)) = \text{FALSE}$
 [ge2] $\geq 0 (\text{real-const}(\text{"-"} _Char+)) = \text{FALSE}$

$$\text{[ge3]} \frac{_Int = \text{hybrid-int-abs}(_Int)}{\geq 0 (_Int) = \text{TRUE}}$$

$$\text{[ge4]} \frac{_Real = \text{hybrid-real-abs}(_Real)}{\geq 0 (_Real) = \text{TRUE}}$$

[gil1] $\text{get-index-list}(_Id, _DStack) =$

$$\text{[gil2]} \frac{\begin{array}{l} \text{eval-exp}(_Exp, _DStack) = _Int, \\ \text{get-index-list}(_Var, _DStack) = _Int* \end{array}}{\text{get-index-list}(_Var [_Exp], _DStack) = _Int* _Int}$$

[gi1] $\text{get-id}(_Var [_Exp]) = \text{get-id}(_Var)$
 [gi2] $\text{get-id}(_Id) = _Id$

F.4 module EvalProgram

module EvalProgram

imports CodeStacks Input Output Initializations

exports

context-free functions

eval-program(PROGRAM) → D-ELEMS
eval-predicate(EXPR, D-STACK) → BOOL-CONST

hiddens

sorts STATUS

context-free functions

"<" C-STACK "," D-STACK ">" → STATUS
eval(STATUS) → D-ELEMS
get-seq(BLOCK) → STAT-SEQ
VALUE → OUTPUT
STRING → OUTPUT
convert(TYPE, VALUE) → VALUE
external-representation(BOOL-CONST) → STRING
values(D-ELEMS) → D-ELEMS

variables

[_] *Var* → VARIABLE
[,] *Stat* → STAT
[,] *StatAux* → STAT-AUX
[,] *Stat* [*] [']* → {STAT ";"}*
[,] *Label* [']* → LABEL
[,] *Block* → BLOCK
[,] *String* → STRING
[,] *Char* [+][']* → CHAR+

equations

$$\begin{aligned} & _Seq = \text{get-seq}(_Block), \\ & _CRec = [_Id, _Seq, _Seq], \\ & _DElem* = \text{init}(_Block, , ,), \\ & _DRec = [_Id, , _DElem*] \\ \hline [\text{ep1}] \quad & \text{eval-program}(\text{PROGRAM } _Id; _Block .) = \text{eval}(\langle _CRec, _DRec \rangle) \end{aligned}$$

$$[\text{ev1}] \quad \text{eval}(\langle [_Id, _Seq,], [_Id, , _DElem*] \rangle) = \text{values}(_DElem*)$$

$$[\text{ev2}] \quad \text{eval}(\langle [_Id, _Seq,] [_Id', _Seq', _Stat; _Stat*] _CRec*, _DRec _DRec+ \rangle) = \text{eval}(\langle [_Id', _Seq', _Stat*] _CRec*, _DRec+ \rangle)$$

$$[\text{ev3}] \quad \text{eval}(\langle [_Id, _Seq, _Label : _StatAux; _Stat*] _CRec*, _DStack \rangle) = \text{eval}(\langle [_Id, _Seq, _StatAux; _Stat*] _CRec*, _DStack \rangle)$$

$$\begin{array}{l}
\text{_Int*} = \text{get-index-list}(\text{_Var}, \text{_DStack}), \\
\text{_Id'} = \text{get-id}(\text{_Var}), \\
\text{_Value} = \text{eval-exp}(\text{_Exp}, \text{_DStack}), \\
\text{_Type} = \text{type}(\text{_Id'}, \text{_Int*}, \text{_DStack}), \\
\text{_Value'} = \text{convert}(\text{_Type}, \text{_Value}), \\
\text{_DStack'} = \text{update}(\text{_Id'}, \text{_Int*}, \text{_DStack}, \text{_Value'}) \\
\hline
[\text{ev4}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Var} := \text{_Exp}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Stat*}] \text{_CRec*}, \text{_DStack'} \rangle) \\
\\
\text{eval-predicate}(\text{_Exp}, \text{_DStack}) = \text{TRUE} \\
\hline
[\text{ev5}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{IF } \text{_Exp} \text{ THEN } \text{_Stat*}' \text{ END}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Stat*}'; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) \\
\\
\text{eval-predicate}(\text{_Exp}, \text{_DStack}) = \text{FALSE} \\
\hline
[\text{ev6}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{IF } \text{_Exp} \text{ THEN } \text{_Stat*}' \text{ END}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) \\
\\
\text{eval-predicate}(\text{_Exp}, \text{_DStack}) = \text{TRUE} \\
\hline
[\text{ev7}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{IF } \text{_Exp} \text{ THEN } \text{_Stat*}' \text{ ELSE } \text{_Stat*}'' \text{ END}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Stat*}'; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) \\
\\
\text{eval-predicate}(\text{_Exp}, \text{_DStack}) = \text{FALSE} \\
\hline
[\text{ev8}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{IF } \text{_Exp} \text{ THEN } \text{_Stat*}' \text{ ELSE } \text{_Stat*}'' \text{ END}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Stat*}''; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) \\
\\
\text{eval-predicate}(\text{_Exp}, \text{_DStack}) = \text{TRUE} \\
\hline
[\text{ev9}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{WHILE } \text{_Exp} \text{ DO } \text{_Stat*}' \text{ END}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Stat*}'; \text{WHILE } \text{_Exp} \text{ DO } \text{_Stat*}' \text{ END}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) \\
\\
\text{eval-predicate}(\text{_Exp}, \text{_DStack}) = \text{FALSE} \\
\hline
[\text{ev10}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{WHILE } \text{_Exp} \text{ DO } \text{_Stat*}' \text{ END}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) \\
\\
\text{_Seq'} = \text{find-label}(\text{_Label}, \text{_Seq}) \\
\hline
[\text{ev11}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{GOTO } \text{_Label}; \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Seq'}] \text{_CRec*}, \text{_DStack} \rangle) \\
\\
\text{_Bool} = \text{eval-exp}(\text{_Exp}, \text{_DStack}), \\
\text{_Dummy} = \text{emit-string}(\text{external-representation}(\text{_Bool})) \\
\hline
[\text{ev12}] \quad \text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{WRITE}(\text{_Exp}); \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle) = \\
\text{eval}(\langle [\text{_Id}, \text{_Seq}, \text{_Stat*}] \text{_CRec*}, \text{_DStack} \rangle)
\end{array}$$

$$\begin{array}{c}
_Value = \text{eval-exp}(_Exp, _DStack), \\
_Dummy = \text{emit}(_Value) \\
\hline
[\text{ev13}] \quad \text{eval}(\langle [_Id, _Seq, \text{WRITE}(_Exp); _Stat*] _CRec*, _DStack \rangle) = \\
\text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DStack \rangle)
\end{array}$$

$$\begin{array}{c}
_Dummy = \text{emit-string}(_String) \\
\hline
[\text{ev14}] \quad \text{eval}(\langle [_Id, _Seq, \text{WRITE}(_String); _Stat*] _CRec*, _DStack \rangle) = \\
\text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DStack \rangle)
\end{array}$$

$$\begin{array}{c}
\text{lookup-proc}(_Id', _DRec*) = _Id''' : _Path : \text{PROCEDURE } _Id'''; _Block, \\
_DElem* = \text{init}(_Block, , , _Path), \\
_Seq' = \text{get-seq}(_Block), \\
_DRec' = [_Id'', _Path, _DElem*], \\
_CRec' = [_Id''', _Seq', _Seq'] \\
\hline
[\text{ev15}] \quad \text{eval}(\langle [_Id, _Seq, _Id'; _Stat*] _CRec*, _DRec* \rangle) = \\
\text{eval}(\langle _CRec' [_Id, _Seq, _Id'; _Stat*] _CRec*, _DRec' _DRec* \rangle)
\end{array}$$

$$\begin{array}{c}
\text{lookup-proc}(_Id', _DRec*) = _Id''' : _Path : \text{PROCEDURE } _Id'' (_Formal+); _Block, \\
_DElem* = \text{init}(_Block, _Actual+, _Formal+, _DRec*, _Path), \\
_Seq' = \text{get-seq}(_Block), \\
_DRec' = [_Id'', _Path, _DElem*], \\
_CRec' = [_Id''', _Seq', _Seq'] \\
\hline
[\text{ev16}] \quad \text{eval}(\langle [_Id, _Seq, _Id' (_Actual+); _Stat*] _CRec*, _DRec* \rangle) = \\
\text{eval}(\langle _CRec' [_Id, _Seq, _Id' (_Actual+); _Stat*] _CRec*, _DRec' _DRec* \rangle)
\end{array}$$

$$\begin{array}{c}
_Int* = \text{get-index-list}(_Var, _DRec*), \\
_Id' = \text{get-id}(_Var), \\
_Type = \text{type}(_Id', _Int*, _DRec*), \\
_Value' = \text{read-value}(_Type), \\
_DRec*' = \text{update}(_Id', _Int*, _DRec*, _Value') \\
\hline
[\text{ev17}] \quad \text{eval}(\langle [_Id, _Seq, \text{READ}(_Var); _Stat*] _CRec*, _DRec* \rangle) = \\
\text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DRec*' \rangle)
\end{array}$$

$$[\text{ev18}] \quad \text{eval}(\langle [_Id, _Seq, ; _Stat*] _CRec*, _DRec* \rangle) = \\
\text{eval}(\langle [_Id, _Seq, _Stat*] _CRec*, _DRec* \rangle)$$

[evp] $\text{eval-predicate}(_Expr, _DStack) = \text{eval-exp}(_Expr, _DStack)$

[val1] $\text{values}() =$

$$[\text{val2}] \quad \frac{\text{values}(_DElem*) = _DElem*'}{\text{values}(_Id : _Value _DElem*) = _Id : _Value _DElem*'}$$

[val3] $\text{values}(_Id : _Path : _ProcDecl _DElem^*) = \text{values}(_DElem^*)$

[con1] $\text{convert}(\text{REAL}, \text{int-const}(_Char+)) = \text{real-const}(_Char+ \text{ "." } "0")$

[con2] $\text{convert}(\text{REAL}, _RealConst) = _RealConst$

[con3]
$$\frac{_Type \neq \text{REAL}}{\text{convert}(_Type, _Value) = _Value}$$

[gs1] $\text{get-seq}(\text{BEGIN } _Seq \text{ END}) = _Seq$

[gs2] $\text{get-seq}(\text{DECLARE } _DeclList \text{ BEGIN } _Seq \text{ END}) = _Seq$

[ex1] $\text{external-representation}(\text{TRUE}) = \text{"T"}$

[ex2] $\text{external-representation}(\text{FALSE}) = \text{"F"}$

F.5 module Initializations

module Initializations

imports EvalExpr

exports

context-free functions

init(BLOCK, {EXPR ";"}, {FORMAL ";"}, D-STACK, PATH) → D-ELEMS
init-decls(DECL-LIST, PATH) → D-ELEMS
init-params({EXPR ";"}, {FORMAL ";"}, D-STACK) → D-ELEMS

variables

[_] *Actual* [*] [']* → {EXPR ";"}
[_] *Formal* [*] [']* → {FORMAL ";"}
[_] *Actual* [+] [']* → {EXPR ";"}
[_] *Formal* [+] [']* → {FORMAL ";"}
[_] *LabDecl* → LABEL-DECL
[_] *Type* → TYPE
[_] *IntConst* [*] [']* → INT-CONST*

hiddens

context-free functions

init-locals(BLOCK, PATH) → D-ELEMS
initial-value(TYPE) → VALUE

equations

$$\begin{array}{l} \text{[in1]} \quad \frac{\begin{array}{l} _DElem*{}' = \text{init-params}(_Actual+, _Formal+, _DStack), \\ _DElem* = \text{init-locals}(_Block, _Path) \end{array}}{\text{init}(_Block, _Actual+, _Formal+, _DStack, _Path) = _DElem* _DElem*{}'} \end{array}$$

$$\text{[in2]} \quad \text{init}(_Block, , , _DStack, _Path) = \text{init-locals}(_Block, _Path)$$

$$\text{[il1]} \quad \text{init-locals}(\text{BEGIN } _Seq \text{ END}, _Path) =$$

$$\text{[il2]} \quad \text{init-locals}(\text{DECLARE } _DeclList \text{ BEGIN } _Seq \text{ END}, _IntConst*) = \text{init-decls}(_DeclList, _IntConst* 1)$$

$$\text{[id1]} \quad \text{init-decls}(_Path) =$$

$$\text{[id2]} \quad \text{init-decls}(_LabDecl; _Decl*, _Path) = \text{init-decls}(_Decl*, _Path)$$

$$\text{[id3]} \quad \frac{\begin{array}{l} _DElem = \text{get-name}(_ProcDecl) : _Int* _Int : _ProcDecl, \\ _DElem* = \text{init-decls}(_Decl*, _Int* \text{incr}(_Int)) \end{array}}{\text{init-decls}(_ProcDecl; _Decl*, _Int* _Int) = _DElem _DElem*}$$

$$\text{[id4]} \quad \frac{\begin{array}{l} _DElem = _Id : \text{initial-value}(_Type), \\ _DElem* = \text{init-decls}(_Decl*, _Path) \end{array}}{\text{init-decls}(_Id : _Type; _Decl*, _Path) = _DElem _DElem*}$$

$$\text{[id5]} \quad \text{init-decls}(_Decl*, _Path) = \text{init-decls}(_Decl*, _Path)$$

$$\text{[iv1]} \quad \text{initial-value}(\text{INTEGER}) = 0$$

[iv2] initial-value(REAL) = 0.0

[iv3] initial-value(BOOLEAN) = FALSE

[iv4] initial-value(ARRAY [*_Low* .. *_Low*] OF *_Type*) =
[*_Low*, *_Low*, initial-value(*_Type*)]

$$\begin{array}{c} \textit{_Low} \neq \textit{_High}, \\ \textit{_Low}' = \textit{incr}(\textit{_Low}), \\ \textit{_Value} = \textit{initial-value}(\textit{_Type}), \\ \hline \text{[iv5]} \frac{\textit{initial-value}(\textit{ARRAY} [\textit{_Low}' .. \textit{_High}] \textit{OF} \textit{_Type}) = [\textit{_Low}', \textit{_High}, \textit{_Value}*]}{\textit{initial-value}(\textit{ARRAY} [\textit{_Low} .. \textit{_High}] \textit{OF} \textit{_Type}) = [\textit{_Low}, \textit{_High}, \textit{_Value} \textit{_Value}*]} \end{array}$$

[ip1] init-params(, , *_DStack*) =

$$\begin{array}{c} \textit{init-params}(\textit{_Actual}* , \textit{_Formal}* , \textit{_DStack}) = \textit{_DElem}* , \\ \textit{_Value} = \textit{eval-exp}(\textit{_Expr}, \textit{_DStack}) \\ \hline \text{[ip2]} \frac{}{\textit{init-params}(\textit{_Expr}, \textit{_Actual}* , \textit{_Id} : \textit{_Type}; \textit{_Formal}* , \textit{_DStack}) = \\ \textit{_Id} : \textit{_Value} \textit{_DElem}*} \end{array}$$
$$\begin{array}{c} \textit{init-params}(\textit{_Actual}* , \textit{_Formal}* , \textit{_DStack}) = \textit{_DElem}* , \\ \textit{_IntConst}* = \textit{get-index-list}(\textit{_Var}, \textit{_DStack}), \\ \textit{_Id}' = \textit{get-id}(\textit{_Var}), \\ \textit{_Value} = \textit{ref}(\textit{_Id}', \textit{_IntConst}*) \\ \hline \text{[ip3]} \frac{}{\textit{init-params}(\textit{_Var}, \textit{_Actual}* , \textit{VAR} \textit{_Id} : \textit{_Type}; \textit{_Formal}* , \textit{_DStack}) = \\ \textit{_Id} : \textit{_Value} \textit{_DElem}*} \end{array}$$

F.6 module Input

module Input

imports DataStacks

exports

context-free functions

read-value(TYPE) → VALUE

hiddens

sorts INPUT TF

context-free functions

"T" → TF

"F" → TF

tf2bool(TF) → BOOL-CONST

"Please enter an INTEGER:" INT-CONST → INPUT

"Please enter a REAL:" REAL-CONST → INPUT

"Please enter T or F:" TF → INPUT

"LISP" LISP → TF

"TF" LISP → TF

"<<" TF ">>" → LISP

equations

```
[rv1] read-value(INTEGER) =
  INT-CONST (let
    (vtp
      (meta (send ' meta (send ' module #:EQM:sel))))
    (#:SEAL:create-input
      "/tmp/input"
      "input"
      (list "Please enter an INTEGER:<INT-CONST>")
      "INPUT"
      (send ' config-table meta))
    (setq vtp (#:SEAL:select
      "/tmp/input"
      "INT-CONST"
      (send ' config-table meta)))
    (#:SEAL:kill-inputs (send ' config-table meta))
    (bitmap-flush)
    (#:EQM:tree:vtp2 vtp))
```

```
[rv2] read-value(REAL) =
REAL-CONST (let
  (vtp
    (meta (send ' meta (send ' module #:EQM:sel))))
  (#:SEAL:create-input
    "/tmp/input"
    "input"
    (list "Please enter a REAL:<REAL-CONST>")
    "INPUT"
    (send ' config-table meta))
  (setq vtp (#:SEAL:select
    "/tmp/input"
    "REAL-CONST"
    (send ' config-table meta)))
  (#:SEAL:kill-inputs (send ' config-table meta))
  (bitmap-flush)
  (#:EQM:tree:vtp2 vtp))
```

```
[rv3] read-value(BOOLEAN) =
tf2bool(TF (let
  (vtp
    (meta (send ' meta (send ' module #:EQM:sel))))
  (#:SEAL:create-input
    "/tmp/input"
    "input"
    (list "Please enter T or F:<TF>")
    "INPUT"
    (send ' config-table meta))
  (setq vtp (#:SEAL:select
    "/tmp/input"
    "TF"
    (send ' config-table meta)))
  (#:SEAL:kill-inputs (send ' config-table meta))
  (bitmap-flush)
  (#:EQM:tree:vtp2 vtp)))
```

[tf1] tf2bool(T) = TRUE

[tf2] tf2bool(F) = FALSE

F.7 module Output

module Output

imports SyntaxLayout Lisp

exports

sorts OUTPUT

context-free functions

"emit" "(" OUTPUT ")" → OUTPUT

"emit-string" "(" OUTPUT ")" → OUTPUT

"LISP" LISP → OUTPUT

"OUTPUT" LISP → OUTPUT

"<<" OUTPUT ">>" → LISP

variables

[_] *Dummy* → OUTPUT

equations

```
[out1] emit(_Dummy) =
  OUTPUT (let*
    ((theEQM (#:EQM:EQMsel:eqm #:EQM:sel))
      (text (#:EQM:tree:pretty theEQM
              (#:EQM:tree:leximplode theEQM << _Dummy >>))))
    (prinflush text)
    << _Dummy >>)
```

```
[out2] emit-string(_Dummy) =
  OUTPUT (let*
    ((theEQM (#:EQM:EQMsel:eqm #:EQM:sel))
      (text (string (#:EQM:tree:pretty theEQM
                    (#:EQM:tree:leximplode theEQM << _Dummy >>))))
      start end
      (len (plength text)))
    (setq current 1)
    (setq end (decr len 2))
    (while (le current end)
      (if (neq (sref text current) 92)
        (progn
          (prin (substring text current 1))
          (incr current 1))
        (progn
          (if (eq (sref text (add 1 current)) 110)
            (print))
          (incr current 2))))
    (prinflush)
    << _Dummy >>)
```

F.8 module Arithmetic

module Arithmetic

imports SyntaxProgram Lisp

exports

context-free functions

hybrid-int-add(INT-CONST, INT-CONST)	→ INT-CONST
hybrid-int-sub(INT-CONST, INT-CONST)	→ INT-CONST
hybrid-int-mul(INT-CONST, INT-CONST)	→ INT-CONST
hybrid-int-div(INT-CONST, INT-CONST)	→ INT-CONST
hybrid-int-abs(INT-CONST)	→ INT-CONST
incr(INT-CONST)	→ INT-CONST
hybrid-real-add-REAL-REAL-CONST, REAL-CONST)	→ REAL-CONST
hybrid-real-sub-REAL-REAL-CONST, REAL-CONST)	→ REAL-CONST
hybrid-real-mul-REAL-REAL-CONST, REAL-CONST)	→ REAL-CONST
hybrid-real-div-REAL-REAL-CONST, REAL-CONST)	→ REAL-CONST
hybrid-real-abs-REAL-REAL-CONST)	→ REAL-CONST
"LISP" LISP	→ INT-CONST
"INT-CONST" LISP	→ INT-CONST
"<<" INT-CONST ">>"	→ LISP
"LISP" LISP	→ REAL-CONST
"REAL-CONST" LISP	→ REAL-CONST
"<<" REAL-CONST ">>"	→ LISP

hiddens

variables

[_] Bool [']*	→ BOOL-CONST
[_] Int [']*	→ INT-CONST
[_] Real [']*	→ REAL-CONST

equations

[hi1] hybrid-int-add(*_Int*, *_Int'*) =
INT-CONST (convert-to-lexical
<< 1 >>
(add (convert-lexical-to-lisp << *_Int* >>
(convert-lexical-to-lisp << *_Int'* >>))))

[hi2] hybrid-int-sub(*_Int*, *_Int'*) =
INT-CONST (convert-to-lexical
<< 1 >>
(sub (convert-lexical-to-lisp << *_Int* >>
(convert-lexical-to-lisp << *_Int'* >>))))

[hi3] hybrid-int-mul(*_Int*, *_Int'*) =
INT-CONST (convert-to-lexical
<< 1 >>
(mul (convert-lexical-to-lisp << *_Int* >>
(convert-lexical-to-lisp << *_Int'* >>))))

[hi4] $\text{hybrid-int-div}(_Int, _Int')$ =
 INT-CONST (convert-to-lexical
 << 1 >>
 (div (convert-lexical-to-lisp << $_Int$ >>)
 (convert-lexical-to-lisp << $_Int'$ >>)))

[hi5] $\text{hybrid-int-abs}(_Int)$ =
 INT-CONST (convert-to-lexical
 << 1 >>
 (abs (convert-lexical-to-lisp << $_Int$ >>)))

[Incr] $\text{incr}(_Int)$ =
 INT-CONST (convert-to-lexical << 1 >>
 (add 1 (convert-lexical-to-lisp << $_Int$ >>)))

[hr1] $\text{hybrid-real-add}(_Real, _Real')$ =
 REAL-CONST (convert-to-lexical
 << 1.0 >>
 (fadd (convert-lexical-to-lisp << $_Real$ >>)
 (convert-lexical-to-lisp << $_Real'$ >>)))

[hr2] $\text{hybrid-real-sub}(_Real, _Real')$ =
 REAL-CONST (convert-to-lexical
 << 1.0 >>
 (fsub (convert-lexical-to-lisp << $_Real$ >>)
 (convert-lexical-to-lisp << $_Real'$ >>)))

[hr3] $\text{hybrid-real-mul}(_Real, _Real')$ =
 REAL-CONST (convert-to-lexical
 << 1.0 >>
 (fmul (convert-lexical-to-lisp << $_Real$ >>)
 (convert-lexical-to-lisp << $_Real'$ >>)))

[hr4] $\text{hybrid-real-div}(_Real, _Real')$ =
 REAL-CONST (convert-to-lexical
 << 1.0 >>
 (fdiv (convert-lexical-to-lisp << $_Real$ >>)
 (convert-lexical-to-lisp << $_Real'$ >>)))

[hr5] $\text{hybrid-real-abs}(_Real)$ =
 REAL-CONST (convert-to-lexical
 << 1.0 >>
 (abs (convert-lexical-to-lisp << $_Real$ >>)))

F.9 module Lisp

module Lisp

exports

sorts LISP

sorts LispAtom StringPart PackagePart QuotedLispAtom
Lispld LispString

lexical syntax

`[$%&*\/\ -0-9=?@A-Z\\a-z~]+` → LispAtom

`“:”` LispAtom → PackagePart

`“#”` PackagePart+ → Lispld

`“#%”` [01]+ → Lispld

`“#$”` [0-9a-fA-F]+ → Lispld

`“#^”` [a-zA-Z@] → Lispld

`“|” ~[|]* “|”` → QuotedLispAtom

QuotedLispAtom+ → LispAtom

LispAtom → Lispld

`“\” ~[~]* “\”` → StringPart

StringPart+ → LispString

context-free functions

Lispld → LISP

LispString → LISP

`“(” LISP* “)”` → LISP

`“<<” Condition “>>”` → LISP

`“/” LISP` → LISP