ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

M P A Sellink (Ed)

2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam 1997

Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam, 25-26 September 1997

A Case Study of a Slicing-Based Approach for Locating Type Errors

T.B. Dinesh and F. Tip

Published in collaboration with the British Computer Society

Springer

ISBN: 3-540-76228-0

A Case Study of a Slicing-Based Approach for Locating Type Errors

T. B. Dinesh
CWI, Amsterdam,
Amsterdam, The Netherlands
email: `dinesh@cwi.nl`

Frank Tip
IBM T.J. Watson Research Center,
Yorktown Heights, NY, USA
email: `tip@watson.ibm.com`

Abstract

Type checkers have been specified in ASF+SDF for many languages, using a variety of specification styles. Several mechanisms for tracking positional information have been proposed, each with shortcomings and/or restrictions on specifications. We propose the use of dynamic dependence tracking for tracking positional information. In this approach, a slice of the program being type checked is associated with each type error. This slice contains precisely those program fragments that caused the type error under consideration. Our approach is completely language-independent, imposes no restrictions on ASF+SDF specifications, and has been applied successfully to a significant subset of Pascal. We report on several experiments that have shown an interesting correlation between the amount of nondeterminism in the specification, and the accuracy of the slices associated with type errors. Generally, more accurate error locations are obtained as the specification gets less deterministic.

1 Introduction

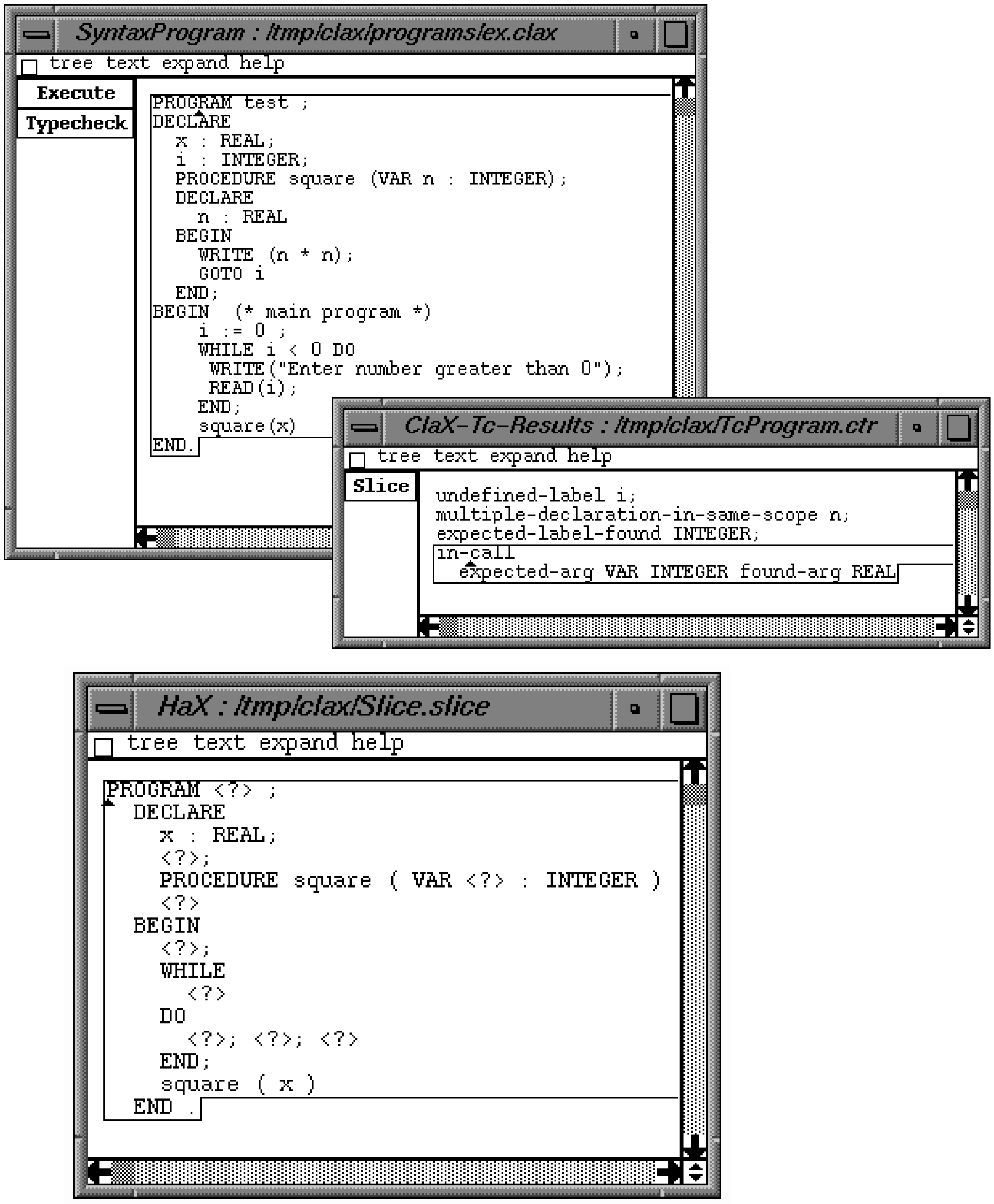
The effectiveness of a type checking tool strongly depends on the quality of the positional information associated with type errors reported by the tool. In a previous paper [11], we presented a framework in which type checkers are specified by way of a set of conditional equations, and where the location of a type error e is provided by way of a slice P_e of the program P being type checked. Here, P_e is a program that contains precisely those constructs in P "responsible" for type-error e, in the sense that type checking P_e is guaranteed to produce the same error e.

We applied this approach to a specification of CLaX, a realistic Pascal-like language, using the ASF+SDF Meta-Environment [15]. Figure 1 shows a snapshot of a type checking tool generated from the CLaX specification; the top window shows a program editor with two buttons attached to it for invoking the type checker and the interpreter, respectively. The middle window shows a list of four error messages reported by the type checker. After selecting an error message in this window, the user may press the "Slice" button to obtain the associated slice. In the figure, the error message "`in-call expected-arg VAR INTEGER found-arg REAL`" is selected, indicating that there is a mismatch between the types of formal and actual parameters in a procedure call. The bottom window shows the slice computed by the system for this error message. This slice contains all program components that contributed to the selected type error. The '`<?>`' symbols in the slice denote placeholders for program constructs not contributing to the error message.

The CLaX language was originally developed as the demonstration language of the ESPRIT-II Compare (Compiler Generation for Parallel Machines) project [1]. Since then, CLaX has been used as a basis for various software tools, including type checkers, interpreters, and debuggers [10, 8, 9, 23, 21], as well as a test-bed for origin-tracking techniques [7, 5, 14].

Several experiments with the ClaX specification revealed that the accuracy of the computed slices depends on a number of specification factors. In particular, we found that decreasing the amount of determinism in the specification improved slice accuracy. The present paper describes the involved engineering aspects for this case study.

The remainder of this paper is organized as follows. In Section 2, we give an overview of earlier work on CLaX, and techniques used for obtaining positional information. We assume the reader to be somewhat familiar with equa-


SyntaxProgram : /tmp/clax/programs/ex.clax
tree text expand help
Execute
Typecheck
ClaX-Tc-Results : /tmp/clax/TcProgram.ctr
tree text expand help
Slice
HaX : /tmp/clax/Slice.slice
tree text expand help

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

```
undefined-label i;
multiple-declaration-in-same-scope n;
expected-label-found INTEGER;
in-call
  expected-arg VAR INTEGER found-arg REAL
```

```
PROGRAM <?> ;
  DECLARE
    x : REAL;
    <?>;
    PROCEDURE square ( VAR <?> : INTEGER )
    <?>
  BEGIN
    <?>;
    WHILE
      <?>
    DO
      <?>; <?>; <?>
    END;
    square ( x )
  END .
```

Figure 1: The CLaX environment. The top window is a program editor with two buttons attached to it for invoking a type checker and an interpreter, respectively. The middle window shows a list of four type errors reported by the type checker, in which the error message “in-call expected-arg VAR INTEGER found-arg REAL” is selected, indicating a mismatch between formal and actual parameter types in a procedure call. The bottom window shows the slice computed for this error message, containing all program components that contributed to the selected type error. The ‘<?>’ symbols in the slice denote placeholders for program constructs not contributing to the error message.

tional specifications and their execution through term rewriting, but some of the idiosyncrasies of ASF+SDF that are relevant for the present paper will be discussed briefly in Section 3. For a more detailed overview, the reader is referred to [23]. Section 4 presents a high-level overview of the specification of the ClaX type checker specification. Section 5 reviews dynamic dependence tracking, the technique used in this paper for computing slices. In Section 6, we report on some of the more interesting changes we made to the ClaX specification, and how these changes affected the computed slices. Finally, in Section 7, we present conclusions and plans for future work.

2 Historical Perspective

The present paper is closely related to earlier work by the same authors. In this section, we give a brief overview of the history of this work.

The CLaX language [10] was originally developed as the demonstration language of the COMPARE (compiler generation for parallel machines) project [1], which was part of the European Union's ESPRIT-II program. Part of CWI's contribution to COMPARE consisted of an algebraic specification of the CLaX language, and the development of a number of programming tools, including a type checker and an interpreter, for CLaX [10].

In both the CLaX type checker and the CLaX interpreter, a mechanism for correlating results to source-text positions was required. To this end, we initially used *origin tracking*, a method specifically developed for this purpose by van Deursen, Klint, and Tip [7]. As defined originally in [7], origin tracking is a relation on a sequence of terms in a rewriting process, and has the property that only "equal" terms are related: if a (sub)term s occurs in the origin of a (sub)term t, then s rewrites to t in zero or more rewriting steps. This property implies that a term only has meaningful origins if it literally occurred in the source term, and was "moved around" but not "erased" in the course of the rewriting process. In the case of the CLaX interpreter, the current term contains a list of statements that remains to be executed. Since these statements originate directly from the program's abstract syntax tree in original term, only a few trivial changes to the specification were required to obtain meaningful origins in all cases. In general, origin tracking is capable of providing useful positional information for "syntax-directed" algebraic specifications that effectively perform a traversal of a program's abstract syntax tree.

The situation was different, however, in the case of the CLaX type checker, which is written in an abstract interpretation style, as will be discussed in Section 4. In many cases, the error messages produced by the type checker did not contain fragments of the program's source, and consequently meaningful origins were only available in a limited number of cases. At the time, our solution to this problem was to modify the specification in a number of ways to improve the performance of origin tracking. The most significant of these changes consisted of:

Tokenization. Some of the origin information that is lost can be recovered by adapting the specification. A straightforward adaptation consists of transforming the syntax to an applicative form (for details, see Appendix C). This allows access to parts of the concrete syntax as these are now effectively arguments to an abstract application function. We refer to this modification as *tokenization*. This slight redefinition of the syntax to its applicative variant, makes it possible to write the specification such that some origin information is obtainable [8].

Other changes. Since origin tracking only maintains relationships between equal subterms, we manipulated the specification in such a way that the resulting error messages always contain source tokens. In general, this is may be a hard thing to do as object language and the source language have not many tokens in common. In the case of the CLaX type checker this turned out to be feasible, although some contrived manipulations were need. For example, we needed to turn an error message "`cannot-assign-to-label`" into "`cannot-assign-to LABEL in :=`", such that the "`LABEL`" and "`:=`"-token in the error message have non-empty origins (e.g., "`:=`"-token would have as origin the assignment where the error occurred).

Making these changes to the specification resulted in accurate and useful positional information. However, we consider this solution to be unsatisfactory because the style in which a specification is written should not be dictated by an implicit mechanism for tracking positional information. In addition, it was not always easy to understand why a particular origin relation was or was not established, and what needed to be changed to create the desired origin relations.

Later work by van Deursen [6, 5] sought to alleviate some of the problems with origin tracking by relaxing the requirement that origin tracking only establishes relationships between equal terms. In this work, van Deursen distinguishes between two kinds of function symbols: language symbols (i.e., symbols occurring in abstract syntax trees), and symbols of functions that "process" language constructs. This additional structure allows one to create meaningful relationships in cases where one language symbol is transformed into another language symbol, or where auxiliary functions are used to process language constructs. A drawback of this method is that it requires additional information in a specification. For certain specification styles, such as the abstract interpretation style used in the present paper, van Deursen's syntax-directed origins are unlikely to create very precise origins.

The dynamic dependence relation used in this paper was developed by Field and Tip [14, 20] with the intention to use it in tools for *program slicing*. A *program slice* [24, 25, 22] is usually defined as the set of statements in a program P that may affect the values computed at the *slicing criterion*, a designated point of interest in P. Two kinds of program slices are usually distinguished. *Static* program slices are computed using compile-time dependence information, i.e., without making assumptions about a program's inputs. In contrast, *dynamic* program slices are computed for a specific execution of a program. An overview of program slicing techniques can be found in [22]. By applying dependence tracking to different rewriting systems, various kinds of program slices can be obtained. In [13] programs are translated to an intermediate graph representation named PIM [12, 2]. An equational logic defines the optimization/simplification and (symbolic) execution of PIM-graphs. Both the translation to PIM and the equational logic for simplification of PIM-graphs are implemented as rewriting systems, and dependence tracking is used to obtain program slices for selected program values. By selecting different PIM-subsystems, different kinds of slices can be computed, allowing for various cost/accuracy tradeoffs to be made. In [21], dynamic program slices are obtained by applying dependence tracking to a previously written specification for a CLaX-interpreter.

The slice notion presented in the current paper differs from the traditional program slice concept in the following way. In program slicing, the objective is to determine a projection of a program that preserves part of its *execution* behavior. By contrast, the slice notion we have used here is a projection of the program for which part of another program property—*type checker* behavior—is preserved.

3 ASF+SDF Idiosyncrasies

In the remainder of this paper, we will assume the reader to be familiar with algebraic specifications, and the execution of algebraic specifications by term rewriting. We refer the uninitiated reader to [3] for details on algebraic specifications, and to [16] for a detailed overview of term rewriting techniques.

The specification discussed in this paper relies on some of ASF+SDF's more interesting features, such as the use of concrete syntax in equations, list functions and list matching, and default equations. The current section briefly reviews these features to the extent that they are used in the CLaX specification. For an overview of the ASF+SDF specification formalism and the ASF+SDF Meta-environment, we refer the reader to [15, 23].

3.1 Lexical Syntax

Lexical tokens can be described using regular expressions. Regular expressions may contain strings, sort names, character classes, or repetition operators. For example, the following specification fragment:

```
sorts ID

lexical syntax
   [a-zA-Z] [a-zA-Z0-9]*        -> ID
   "%%" ~ [\n] * "\n"           -> LAYOUT
```

defines identifier lexicals that begin with an upper case or lower case letter, followed by zero or more alphanumerical characters. `LAYOUT` is a special sort in ASF+SDF, using which one can specify white space. Here, we define a comment convention that everything between '`%%`' and an end-of-line character '`\n`' is a comment (white space).

3.2 Lists and Injections

Description of the repetition of a syntactic notion or of list structures (with or without separators) containing a syntactic notion is done using ASF+SDF lists. Lists can be with or without separators: `S*` and `S+` define zero or more repetitions of sort `S`, and one or more repetitions of sort `S`, without separators, respectively Similarly, `{S SEP}*` and `{S SEP}+` define lists separated by the literal `SEP`.

In CLaX, `DECL-LIST`, a list of declarations, is defined as follows:

```
sorts ID DECL TYPE
lexical syntax
   [a-zA-Z] [a-zA-Z0-9]*      -> ID
context-free syntax
   ID ":" "LABEL"             -> LABEL-DECL
   LABEL-DECL                 -> DECL
   VAR-DECL                   -> DECL
   { DECL ";" }*              -> DECL-LIST
```

The function `LABEL-DECL  -> DECL` is an injection of one sort into another. Here, a declaration `DECL` can either be a label declaration `LABEL-DECL` or a variable declaration `VAR-DECL`.

3.3 Default Rules

When the prefix '`default-`' is used in the label of an equation, the equation is considered only after considering all other equations. For example, in Appendix B.5 we see:

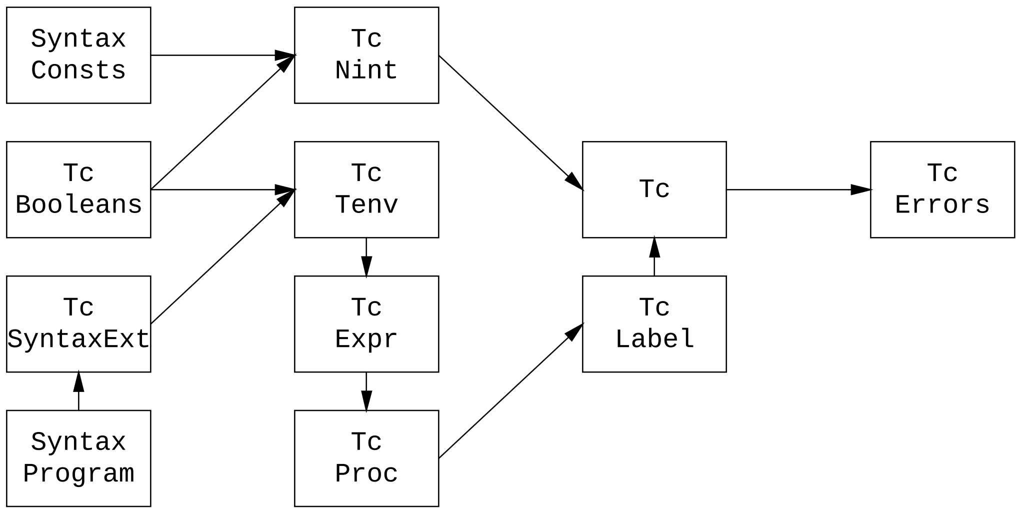
```
[C0]  check(_LabelList, _Stat;_StatSeq) =
        check(_LabelList, _Stat) & check(_LabelList, _StatSeq)

[C1]  check(_LabelList, _AssignStat) = true
[C2]  check(_LabelList, _TestStat) = true
[C3]  check(_LabelList, _InOutStat) =  true
[C4]  check(_LabelList, _ProcStat) = true
[C5]  check(_LabelList, _EmptyStat) = true
[C6]  check(_LabelList, _Id: _StatAux) = check(_LabelList, _StatAux)
[C7]  check(_Id* _Id _Id*', GOTO _Id) = true

[default-C8]
                        _Id != _Id'
                     ================
         check(_Id* _Id' _Id*', GOTO _Id) = check(_Id* _Id*', GOTO _Id)
```

equation `C8` is considered only after `C0` to `C7` are considered. However, `C0` to `C7` themselves are considered in a nondeterministic way.

Default equations can be used for different purposes. Default equations can be used to avoid the use of negative conditions in conditional equations. Such use of default equations is problematic from a dependence tracking point of view, because the applicability of a default equation no longer depends on that equation alone, but also on the fact that other equations are *not* applicable. In the CLaX specification, default equations are used in only a few places, and only to improve efficiency. In the case of the above example, having `C8` as "default" does not change the semantics but results in some performance gain.



4 Overview of the Specification

Once a program is accepted as valid according to the CLaX syntax (see Appendix A), the context-dependent aspects of the program need to be checked. For example, 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 [19]. The modules that constitute the complete static semantics of CLaX can be found in Appendix B. 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, which is written in an abstract interpretation style.

The specification of the CLaX type checker imports the CLaX syntax defined in module `SyntaxProgram` (see Appendix A). The import diagram for the type checking modules is shown in Figure 4. 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 human-readable error messages from these abstract values. Section 5 discusses how the system automatically locates the source positions of these errors.

4.1 Extending the Language

The syntax of the language is generalized slightly in module `TcSyntaxExt` (see Appendix B.3, relevant fragments 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 the syntax extensions are only used in the type checking modules.

module TcSyntaxExt

```
imports SyntaxProgram

exports
  context-free syntax
   EXPR ":=" EXPR                 -> ASSIGN-STAT
   EXPR "[" EXPR "]"              -> EXPR
   "READ"  "(" EXPR ")"           -> IN-OUT-STAT
```

Module `TcSyntaxExt` generalizes (i) the assignment statement, (ii) 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.

4.2 Type-environments

Module `TcTenv` (see Appendix B.8, relevant details shown below) 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`) pairs. Later, the sort `TYPE` is also extended with `LABEL-TYPE` and sort `PROC-TYPE`, so that all identifiers can be uniformly represented using the `ID : TYPE` notation.

module TcTenv

```
imports TcSyntaxExt TcBooleans

exports
  sorts TENV
  context-free syntax
    TYPE                          -> EXPR
    "[" {DECL ";"}* "]"           -> TENV
    TENV*                         -> TENV-LIST
    type-of(TENV-LIST, EXPR)      -> TYPE
```

Because we want to use types as abstract values, sort `TYPE` is 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 `type-of(TENV-LIST, 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 B.4).

4.3 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 `[t17]` 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] _Expr _Cop _Expr'  = _Expr _Op _Expr'  when _Op = _Cop
[t17] _SimpleType _Op _SimpleType = BOOLEAN when _Op = _Cop
```

4.4 Type Checking Procedures

The most significant parts of module `TcProc` (Appendix B.7), 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's `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

exports
  sorts PROC-TYPE VTYPE TYPE-LIST
  context-free syntax
    "LABEL-TYPE"                             -> TYPE
    "PROC""(" {VTYPE ";"}* ")"               -> PROC-TYPE
    PROC-TYPE                                -> TYPE
    {TYPE ";"}*                              -> TYPE-LIST
    types-of(TENV-LIST, {EXPR ","}*)         -> TYPE-LIST
    formals(PROC-DECL)                       -> TENV
    signature(PROC-HEAD)                     -> PROC-TYPE
    TYPE                                     -> VTYPE
    "VAR" TYPE                               -> VTYPE
    vtype(FORMAL)                            -> VTYPE
    isproc"(" EXPR "(" TYPE-LIST ")" ")"     -> BOOL
    vararg"(" EXPR "(" {EXPR ","}* ")" ")"   -> BOOL
    body-of( PROC-DECL )                     -> BLOCK
```

4.5 Type Checking labels

Module `TcLabel` (Appendix B.5) handles the various cases of label consistency that must be checked so as to type check a `GOTO` statement. The 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.

module TcLabel

```
imports TcProc

exports
  sorts LABEL-LIST
  context-free syntax
    ID*                                -> LABEL-LIST
    defines( STAT-SEQ )                -> LABEL-LIST
    unique( LABEL-LIST )               -> BOOL
    no-dups( LABEL-LIST )              -> BOOL
    check( LABEL-LIST, STAT-SEQ )      -> BOOL
    islabel(EXPR)                      -> BOOL
```

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 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 `check(LABEL-LIST,STAT-SEQ)` is used, which returns `true` if the labels used in the `GOTO` statements in the `STAT-SEQ` are in the `LABEL-LIST`. Checking if an (abstract) value is a label is done by function `islabel`, which returns `true` if the expression is `LABEL`.

4.6 Type Checking Programs

Module `Tc` (Appendix B.1) 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 $\boxed{\textbf{Typecheck}}$ button on the editor window.

Informally, the type checking proceeds as follows:

- The declarations of the block are processed, yielding a local type-environment.
- Some 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`, and the condition of the `IF` or `WHILE` is transformed into an “abstract” `TEST` statement. The allows us to specify the checking of the validity of the conditional expression once for all conditional statements.
- 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 B.2). Type-correct statements evaluate to `true`.

module Tc

```
imports TcLabel TcNint

exports
  sorts TENV-LIST
  context-free syntax
```

```
    tc(PROGRAM)                         -> BOOL
    collect( TENV-LIST, BLOCK )         -> BOOL
    distribute( TENV-LIST, BLOCK )      -> BOOL
    STAT                                -> BOOL
    isbool(EXPR)                        -> BOOL
    unique-decls(TENV*)                 -> BOOL
    nonemptyarray(TENV)                 -> BOOL
    no-dups(TENV)                       -> BOOL
    get-id(DECL)                        -> ID
    get-type(DECL)                      -> TYPE
```

4.7 Generating Error Messages

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 B.10). This function is distributed over all transformed statements that remain after type checking. Each equation for the function `errors` handles one particular type-error.

As an example, we show the processing of `LABEL := EXPR`; here an error-message `cannot-assign-to-label` is generated by the following equation:

```
[S03] errors(LABEL-TYPE := _Expr) = cannot-assign-to-label
```

4.8 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. After this step, the main program will look as follows:

   ```
   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 strings are not first class `TYPE`s in CLaX (there are no operations defined over strings), they do not have an abstract value.

2. Collect 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 procedure `square`, a snap-shot might look as follows:

   ```
   collect([ i :  INTEGER;
     square :  PROC (INTEGER);
     n :  INTEGER;
     x :  REAL;
     step :  LABEL
   ],
     DECLARE
       BEGIN
           x := INTEGER;
           step := n;
           step := step * REAL; ···
       END)
     &
   collect([ 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 (for the fact that label `step` is defined twice):

   ```
   unique(step step)
     &
   distribute([ i :  INTEGER;
     square :  PROC (INTEGER);
   ```

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

Table 1: The result of type checking the program.

```
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) & true &
REAL := INTEGER &
LABEL-TYPE := INTEGER &
LABEL-TYPE := LABEL-TYPE * REAL &
...
```

5. Perform abstract evaluation of expressions.

6. Generate error messages. The normal form of Table 1 is translated to:

```
multiply-defined-label step ;
cannot-assign-to-label ;
cannot-assign-to-label ;
label-used-as-operand ;
in-call expected-arg INTEGER found-arg REAL
```

The translator has converted `LABEL-TYPE := LABEL-TYPE * REAL` into the error-message `cannot-assign-to-label`. There are 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 5 discusses how such information can be obtained automatically using dynamic dependence tracking.

5 Dynamic Dependence Tracking

In order to obtain positional information for type errors, we use a technique called *dependence tracking* that was developed by Field and Tip [14, 20]. For a given sequence of rewriting steps $T_0 \to \cdots \to T_n$, dependence tracking computes a slice of the original term, T_0, for each function symbol or subcontext (a notion that will be presented below) of the result term, T_n.

We will use the following simple specification of integer arithmetic (taken from [21]) as an example to illustrate dependence tracking:

```
A1  intmul(0,X)              =  0
A2  intmul(intmul(X, Y),Z)  =  intmul(X,intmul(Y, Z))
```

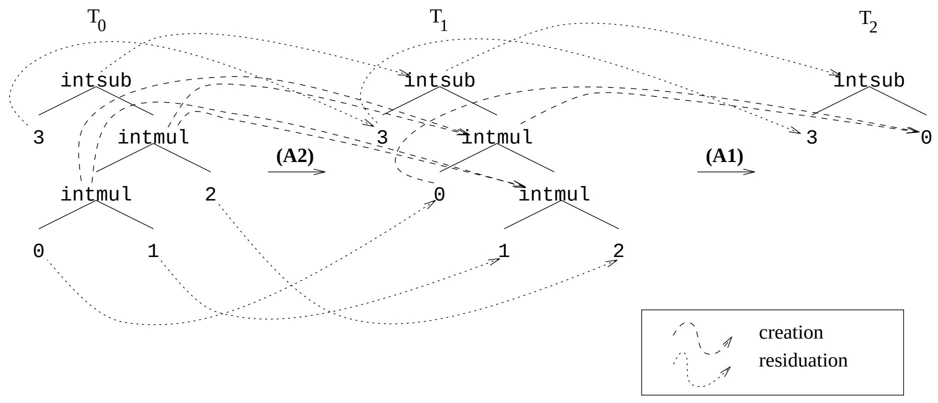


Figure 2: Example of creation and residuation relations.

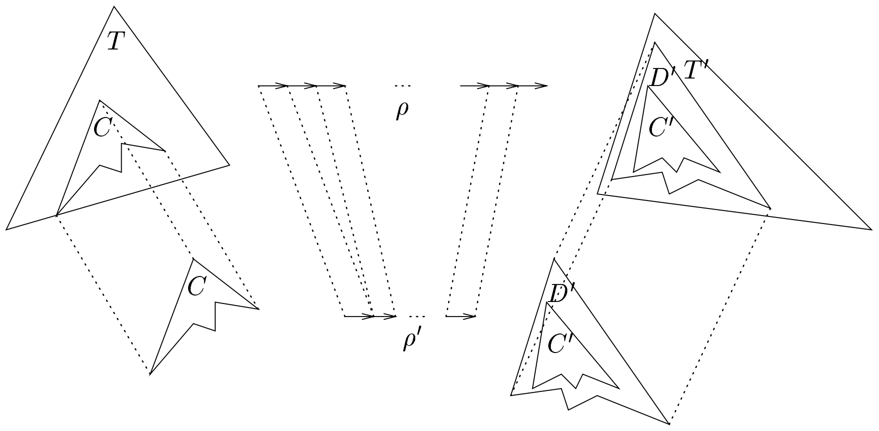


Figure 3: Depiction of the definition of a term slice.

By applying these equations, the term `intsub(3, intmul(intmul(0, 1), 2))` may be rewritten as follows (subterms affected by rule applications are underlined):

$$\begin{aligned}
T_0 =\ & \texttt{intsub}(3, \underline{\texttt{intmul(intmul(0, 1), 2)}}) \\
& \xrightarrow{\texttt{A2}} \\
T_1 =\ & \texttt{intsub}(3, \underline{\texttt{intmul(0, intmul(1, 2))}}) \\
& \xrightarrow{\texttt{A1}} \\
T_2 =\ & \texttt{intsub}(3, 0)
\end{aligned}$$

By carefully studying this example, one can observe the following:

- The outer context `intsub(3, <?>)` of T_0 (‘`<?>`’ denotes a missing subterm) is not affected at all, and therefore reappears in T_1 and T_2.

- The occurrence of variables *X*, *Y*, and *Z* in both the left-hand side and the right-hand side of `A2` causes the respective subterms `0`, `1`, and `2` of the underlined subterm of T_0 to reappear in T_1.

- Variable *X* only occurs in the left-hand side of `A1`. Consequently, the subterm `intmul(1, 2)` (of T_1) that is matched against *X* does not reappear in T_2. In fact, we can make the stronger observation that the subterm matched against *X* is *irrelevant* for producing the constant `0` in T_2: the “creation” of this subterm `0` only requires the presence of the context `intmul(0, <?>)` in T_1.

The above observations are the cornerstones of the dynamic dependence relation of [14, 20]. Notions of *creation* and *residuation* are defined for single rewrite-steps. The former involves function symbols produced by rewrite rules whereas the latter corresponds to situations where symbols are copied, erased, or not affected by rewrite rules[1]. Figure 2 shows all residuation and creation relations for the example reduction discussed above.

Roughly speaking, the dynamic dependence relation for a sequence of rewriting steps ρ consists of the transitive closure of creation and residuation relations for the individual steps in ρ. In [14, 20], the dynamic dependence relation is defined as a relation on *contexts*, i.e., connected sets of function symbols in a term. The fact that C is a *subcontext* of a term T is denoted $C \sqsubseteq T$. For any sequence of rewrite steps $\rho : T \to \cdots \to T'$, a *term slice* with respect to some $C' \sqsubseteq T'$ is defined as the subcontext $C \sqsubseteq T$ that is found by tracing back the dynamic dependence relations from C'. The term slice C satisfies the property that C can be rewritten to a term $D' \sqsupseteq C'$ via a sequence of rewrite steps ρ', where ρ' contains a subset of the rule applications in ρ. This property is illustrated in Figure 3.

Returning to the example, we can determine the term slice with respect to the entire term T_2 by tracing back all creation and residuation relations to T_0. The reader may verify that the term slice with respect to `intsub(3, 0)` consists of the context `intsub(3, intmul(intmul(0, <?>), <?>))`.

The middle window of the CLaX environment of Figure 1 is a textual representation of a term that represents a list of errors. The slice shown in the bottom window of Figure 1 was computed by tracing back the dependence relations from the selected “error” subterm.

6 Experiments

In this section, we summarize a number of changes we made to the specification in order to improve the accuracy of the computed slices. As it turns out, almost all of these changes have the flavor of “removing redundant determinism” or “over-specification”. In addition to the changes we outline below, we made the following other changes:

- We removed the “tokenization syntax” that was introduced to facilitate origin tracking (see Appendix C for details on this issue).

[1] The notions of creation and residuation become more complicated in the presence of so-called *left-nonlinear* rules and *collapse rules*. This is discussed at greater length in [14, 20].

- We made a number of syntactic changes to the specification to improve readability. In particular, we now use standard prefix functions instead of mix-fix function symbols. For example, the act of collecting the type environment and distributing it to the statements was earlier done using a function `TENV* "^" BLOCK -> BOOL`. For readability, we have split this into two functions that collect and distribute type-environments:

```
collect( TENV-LIST, BLOCK )         -> BOOL
distribute( TENV-LIST, BLOCK )      -> BOOL
```

- We restored the syntax of error messages to their "natural" form. The use of origin tracking required that program fragments were retained in error messages (otherwise these error messages would have non-empty origins). Since dependence tracking does not impose such a restriction, we adopted a more "natural" style for error messages. For example, the message associated with the error `LABEL := EXPR` needed the token `:=` for origin tracking to relate enough origins. Thus we had an equation

```
[Er] errors(LABEL := _Expr) = cannot-assign-to LABEL in :=
```

  Now, we can simply have an expression `cannot-assign-to-label` on the right hand side of this equation, and still get all the desired dependence relations.

6.1 Over-Specification: Unnecessarily Specific Matching

In a number of places, the type checker specification of [10] was matching unnecessarily specific subterms, which gave rise to spurious dependences. For example, the original specification contained an equation:

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

which expressed the fact that any declaration of the form `_Id : LABEL` is not a declaration of an array with 0 elements. Since the '`LABEL`' subterm of the declaration is explicitly matched in the equation, '`<?> : LABEL`' subterms inadvertently showed up in the slices reported by the tool. It turned out that using the following, slightly more general equation instead:

```
[NA1] nonemptyarray([_LabelDecl]) = true
```

had the desired effect of omitting the entire label declaration from the slice.

6.2 Flattening of Control-Flow Structures

Control-flow structures have little to do with the type checking of program constructs. Ignoring issues related to the scopes of variables, the type checking of a statement does not depend on the position of that statement in the program. This observation can be used to simplify the description of the type checker, by "flattening" the control flow constructs: All statements that occur inside an `IF` or `WHILE` construct can be hoisted outside that construct without affecting the type checking process. This has the pleasant property that the rules for type checking statements need only be concerned with straight-line code.

In the specification of [10], a function `flat` is explicitly applied to a statement list before any statements in that list are type checked. The equations for `flat` are shown below:

```
[FL0] flat _StatAux = _StatAux' ; _StatSeq'*
      ======================================
      flat _Id : _StatAux = _Id : _StatAux' ; _StatSeq'*

[FL1] flat _Expr := _Expr' = _Expr := _Expr'
[FL2] flat _Id = _Id
```

```
[FL3] flat _Id _( _ExprList _) = _Id _( _ExprList _)
[FL4] flat READ ( _Expr ) = READ ( _Expr )
[FL5] flat WRITE ( _Expr ) = WRITE ( _Expr )
[FL6] flat WRITE ( _String ) = WRITE ( _String )
[FL7] flat GOTO _Id = GOTO _Id
[FL8] flat =

[FL9] _StatSeq' = flat _StatSeq
       =========================
       flat IF _Expr THEN _StatSeq END = IF _Expr THEN END ; _StatSeq'

[FL10] _StatSeq'' = flat _StatSeq ; _StatSeq'
       ======================================
       flat IF _Expr THEN _StatSeq ELSE _StatSeq' END =
       IF _Expr THEN END ; _StatSeq''

[FL11] _StatSeq' = flat _StatSeq
       =========================
       flat WHILE _Expr DO _StatSeq END = WHILE _Expr DO END ; _StatSeq'

[FL12] _StatSeq' = flat _Stat, _StatSeq'' = flat _StatSeq
       ==============================================
       flat _Stat ; _StatSeq = _StatSeq' ; _StatSeq''
```

`flat` explicitly traverses a list of statements by recursively applying `flat` to sublists of the list of statements (equation `FL12`). Statements other than `IF` and `WHILE`, are left unchanged by `flat` (equations `FL0`–`FL8`). For `IF` and `WHILE` constructs, `flat` hoists the nested statement lists inside these constructs (equations `FL9`–`FL11`).

This approach to specifying the flattening process has a drawback. The dynamic dependence relations create a dependency of each statement in a “flattened list” on the surrounding `DECLARE--BEGIN--END` or `BEGIN--END` symbol(s).

We eliminated this spurious dependency by restating the flattening operation non-deterministically, as is shown below:

```
[flat1]  _StatSeq1*; WHILE _Expr DO_StatSeq2 END; _StatSeq3* =
            _StatSeq1*; TEST _Expr END; _StatSeq2; _StatSeq3*

[flat2]  _StatSeq1*; IF _Expr THEN_StatSeq2 END; _StatSeq3* =
            _StatSeq1*; TEST _Expr END; _StatSeq2; _StatSeq3*

[flat3]  _StatSeq1*; IF _Expr THEN_StatSeq2 ELSE _StatSeq3 END; _StatSeq4* =
            _StatSeq1*; TEST _Expr END; _StatSeq2; _StatSeq3; _StatSeq4*
```

Each of these equations apply implicitly to *any* statement list, i.e., there is no *explicit* call to a flattening function. Equation `flat1` transforms a statement list containing a `WHILE` statement by hoisting its body and transforming the `WHILE` into a `TEST` statement. Equations `flat2` and `flat3` perform similar transformations on `IF--THEN` and `IF--THEN--ELSE` constructs. The generated `TEST` statement is a “generic” conditional statement whose control predicate must be of a boolean type (the original specification contained distinct, similar checks for control predicates in `IF` and `WHILE` constructs). Using this approach, the specification can now assume all statement lists to be free of `IF` and `WHILE` constructs.

6.3 Elimination of Correct Program Constructs

If we look at module `TcBooleans` (Appendix B.2), we notice that there are no equations that indicate `true` is the identity value of the `BOOL`s. Originally, these simplification rules were present. This implied, however, that the "error values" depended on the `true` or non-error values. For example, in

```
  [Bool1] _Bool & true = _Bool
```

the resulting expression on the right-hand side depends on the presence of `true` on the left hand side.

In the current situation without boolean simplification rules, `true` subterms remain until the error processing is done in the next phase. Then, `errors(true)` reduces to "no-errors" by the following equation:

```
  [E0] errors(true)  = no-errors
```

Then, the list-match equation below eliminates `no-errors` subterms, when the rest of the list not empty. This causes the list symbol to depend on correct statements, but this is no problem since we're only interested in slices w.r.t. individual statements.

```
  [M0] _MsgList ; no-errors ; _MsgList'   = _MsgList ; _MsgList'
        when _MsgList ; _MsgList' = _MsgList'' ; _Msg
```

6.4 Elimination of Determinism: Duplicate Elements in Lists

Overspecification is undesirable because it may result in overly large slices. Unfortunately, it can be hard to control over-specification, and it is sometimes not apparent from the specification where over-specification occurs. We take the example of the function `unique` in module `TcLabel` (see Appendix B.5) to illustrate this point. If we have `unique` defined as follows (here, & denotes boolean conjunction):

```
 [xU1] unique(_LabelList) = no-dups(_LabelList)

 [xN0] no-dups() = true
 [xN1] no-dups(_Id) = true
 [xN2] no-dups(_Id _Id') = true when_Id != _Id'
 [xN3] no-dups(_Id _Id' _Label+) =
       no-dups(_Id _Id') & no-dups(_Id _Label+) & no-dups(_Id' _Label+)
```

then, the specification would state that a list is unique if it is true that there are no duplicates[2]. Thus, when a list is not unique, the non-uniqueness of the duplicate elements depends on other elements in the list.

Instead, we define `unique` as follows to avoid over-specification.

```
 [U1] unique(_LabelList) = true when no-dups(_LabelList) != false

 [N1] no-dups(_Id* _Id _Id*' _Id _Id*'') = false
```

The function `unique` defines a list to be unique only if it not the case that it has duplicate elements. Thus, when a list is indeed unique, the function `no-dups` does not match.

[2]Note that equation `no-dups(_Id _Id) = false` is not defined because that would be over-specification. We are only interested in the case where `unique` is `true`. For the case that it is not `true`, the resulting irreducible term can be post-processed into a human-readable error message.

7 Conclusions

We presented a slicing-based approach for determining locations of type errors in [11]. Our work assumes a framework in which type checkers are specified algebraically, and executed by way of term rewriting [16]. In this model, a type check function rewrites a program's abstract syntax tree to a list of type errors. Dynamic dependence tracking [14, 20] is used to associate a *slice* [24, 22] of the program with each error message.

In this paper, we have presented a case study of this approach for the non-toy, Pascal-like language CLaX. We have implemented this work in the context of the ASF+SDF Meta-environment [15, 23]. We have compared the method of this paper with earlier work for generating error reporters based on origin tracking. The earlier, origin tracking based work required non-trivial modifications to the specification to ensure that each term has a non-empty origin. By contrast, the approach of the present paper guarantees the availability of positional information, regardless of the specification style that is used. Experimentation with the CLaX specification revealed that the computed slices provide highly insightful information regarding the nature of type violations. Nonetheless, we have observed that the accuracy of the computed slices depends strongly on the amount of determinism in the specification, and our experiments led to a number of changes in the specification that made it less deterministic. In addition to producing improved slices, we believe these changes have improved the specification in the sense that it became more of a "specification" of the type checking process, and less focused on the specific mechanisms and algorithms for computing type errors. In this paper, we have presented a case study of this approach for the non-toy, Pascal-like language CLaX. We have implemented this work in the context of the ASF+SDF Meta-environment [15, 23].

As a direction for future work, we intend to study the applicability of slicing-based error location in the related area of type *inference* [4], in particular for object-oriented languages [18] and for ML [17]. Although a slice can be computed for each reported type inference error, it is unclear how accurate such slices will be in practice.

References

[1] ALT, M., ASSMANN, U., AND VAN SOMEREN, H. Cosy compiler phase embedding with the cosy compiler model. In *Compiler Construction '94* (1994), P. A. Fritzson, Ed., vol. 786 of *LNCS*, Springer-Verlag, pp. 278–293.

[2] BERGSTRA, J., DINESH, T., FIELD, J., AND HEERING, J. A complete transformational toolkit for compilers. In *Proc. European Symposium on Programming* (Linköping, Sweden, April 1996), vol. 1058 of *Lecture Notes in Computer Science*, Springer-Verlag. Full version: Technical Report CS-R9646, Centrum voor Wiskunde en Informatica (CWI), Amsterdam; To appear in TOPLAS, 1997.

[3] BERGSTRA, J., HEERING, J., AND KLINT, P., Eds. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[4] CLÉMENT, D., DESPEYROUX, J., DESPEYROUX, T., AND KAHN, G. A simple applicative language: Mini-ml. In *Proc. 1986 ACM Symposium on Lisp and Functional Programming* (1986), pp. 13–27.

[5] DEURSEN, A. V. *Executable Language Definitions—Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994.

[6] DEURSEN, A. V. Origin tracking in primitive recursive schemes. Report CS-R9401, Centrum voor Wiskunde en Informatica (CWI), 1994.

[7] DEURSEN, A. V., KLINT, P., AND TIP, F. Origin tracking. *Journal of Symbolic Computation 15* (1993), 523–545.

[8] DINESH, T. B. Type checking revisited: Modular error handling. In *Semantics of Specification Languages* (1994), D. J. Andrews, J. F. Groote, and C. A. Middelburg, Eds., Workshops in Computing, Springer-Verlag, pp. 216–231. Utrecht 1993.

[9] DINESH, T. B. Typechecking with modular error handling. In *Language Prototyping: An Algebraic Specification Approach*, A. v. Deursen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co., 1996, pp. 85–104.

[10] DINESH, T. B., AND TIP, F. Animators and error reporters for generated programming environments. Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), 1992.

[11] DINESH, T. B., AND TIP, F. A slicing-based approach for locating type errors. In *Proceedings of the USENIX Conference on Domain-Specific Languages (DSL'97)* (Santa Barbara, CA, October 1997), pp. 77–88.

[12] FIELD, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1992), pp. 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.

[13] FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[14] FIELD, J., AND TIP, F. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming* (1994), M. Hermenegildo and J. Penjam, Eds., vol. 844, Springer-Verlag, pp. 415–431.

[15] KLINT, P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology 2*, 2 (1993), 176–201.

[16] KLOP, J. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1992, pp. 1–116.

[17] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[18] PALSBERG, J., AND SCHWARTZBACH, M. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.

[19] THE COMPARE CONSORTIUM. Description of the COSY-prototype. Tech. rep., GMD, 1991. unpublished.

[20] TIP, F. *Generation of Program Analysis Tools*. PhD thesis, University of Amsterdam, 1995.

[21] TIP, F. Generic techniques for source-level debugging and dynamic program slicing. In *Proceedings of the Sixth International Joint Conference on Theory and Practice of Software Development* (Aarhus, Denmark, May 1995), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., vol. 915 of *LNCS*, Springer-Verlag, pp. 516–530.

[22] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[23] VAN DEURSEN, A., HEERING, J., AND KLINT, P., Eds. *Language Prototyping—An Algebraic Specification Approach*, vol. 5 of *AMAST Series in Computing*. World Scientific, 1996.

[24] WEISER, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[25] WEISER, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

A The Syntax of CLaX

A.1 Module SyntaxLayout

```
%% -------------------
%% Module SyntaxLayout
%% -------------------
%%
%% 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.
%%
%% Define lay out symbols of Clax programs. Note that in this specification
%% no * between comment delimitters (* and *) is allowed.

exports
  lexical syntax
    [ \t\n]         -> LAYOUT
    "{" ~[}]* "}" -> LAYOUT
    "(*" ~[*]* "*)" -> LAYOUT
```

A.2 Module SyntaxConsts

```
%% -------------------
%% Module SyntaxConsts
%% -------------------

imports SyntaxLayout

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-REAL-CONST
    UNS-INT-CONST "." UNS-INT-CONST               -> UNS-REAL-CONST
    UNS-INT-CONST "." UNS-INT-CONST SCALE-FACTOR -> UNS-REAL-CONST

variables
     [_]Id['']*                                   -> ID
     [_]IntConst['']*                             -> INT-CONST
     [_]BoolConst['']*                            -> BOOL-CONST
     [_]RealConst['']*                            -> REAL-CONST
     [_]String['']*                               -> STRING
```

A.3 Module SyntaxExpr

```
%% -----------------
%% Module SyntaxExpr
%% ----------------


imports SyntaxConsts

exports
  sorts UAOP AOP1 AOP2 UBOP BOP1 BOP2 COP VARIABLE EXPR

  context-free syntax
      "-"    -> UAOP
      "NOT"  -> UBOP

      "*"    -> AOP1
      "/"    -> AOP1
      "%"    -> AOP1
      "&"    -> BOP1

      "+"    -> AOP2
      "-"    -> AOP2
      "|"    -> BOP2

      "<"    -> COP
      "<="   -> COP
      "="    -> COP
      ">="   -> COP
      ">"    -> COP
      "#"    -> COP

    "(" EXPR ")"                       -> EXPR {bracket}

    ID                                 -> VARIABLE
    VARIABLE "[" EXPR "]"              -> 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
    [_]Aop1[']*    -> AOP1
    [_]Aop2[']*    -> AOP2
    [_]Bop1[']*    -> BOP1
    [_]Bop2[']*    -> BOP2
    [_]Cop[']*     -> COP
    [_]Uaop[']*    -> UAOP
    [_]Ubop[']*    -> UBOP
    [_]Var[0-9']*  -> VARIABLE
    [_]Expr[0-9']* -> EXPR
priorities
      {EXPR COP EXPR -> EXPR}
    < {left: EXPR AOP2 EXPR -> EXPR, EXPR BOP2 EXPR -> EXPR}
    < {left: EXPR AOP1 EXPR -> EXPR, EXPR BOP1 EXPR -> EXPR}
    < {UAOP EXPR -> EXPR, UBOP EXPR -> EXPR }
```

A.4 Module SyntaxHeaders

```
%% -------------------
%% Module SyntaxHeaders
%% -------------------

imports SyntaxTypes

exports
  %% BLOCK in SyntaxProgram
  sorts PROC-HEAD LABEL-DECL PROC-DECL VAR-DECL DECL DECL-LIST
        FORMAL BLOCK

  context-free syntax
    ID ":" "LABEL"                          -> 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" VAR-DECL                          -> FORMAL
    "PROCEDURE" ID                          -> PROC-HEAD
    "PROCEDURE" ID "(" {FORMAL ";"}+ ")"    -> 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
```

A.5 Module SyntaxProgram

```
%% -------------------
%% Module SyntaxProgram
%% -------------------

imports SyntaxHeaders SyntaxStats

exports
  sorts PROGRAM   %% BLOCK is defined in Module SyntaxHeaders
  context-free syntax

    "DECLARE" DECL-LIST "BEGIN" STAT-SEQ "END"  -> BLOCK
    "BEGIN" STAT-SEQ "END"                      -> BLOCK

    "PROGRAM" ID ";" BLOCK "."                  -> PROGRAM

  variables
    [_]Program[0-9’]*   -> PROGRAM
```

A.6 Module SyntaxStats

```
%% ------------------
%% Module SyntaxStats
%% ------------------

imports SyntaxExpr

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

  context-free syntax
    ID                                    -> LABEL
    { STAT ";" }+                         -> STAT-SEQ
                                          -> EMPTY-STAT
    ID                                    -> PROC-STAT
    VARIABLE ":=" EXPR                    -> ASSIGN-STAT
   "TEST" EXPR "END"                             -> TEST-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 "(" {EXPR ","}+ ")"                 -> 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
   TEST-STAT                          -> STAT-AUX
   STAT-AUX                               -> STAT
   LABEL ":" STAT-AUX                     -> STAT

variables
    [_]StatSeq[0-9’]*           -> {STAT ";"}+
    [_]StatSeq[0-9’]"*"         -> {STAT ";"}*
    [_]Stat[0-9’]*              -> STAT
    [_]ExprList[0-9]*           -> {EXPR ","}*
    [_]StatAux[’]*              -> STAT-AUX
    [_]Label[’]*                -> LABEL
    [_]AssignStat               -> ASSIGN-STAT
    [_]CondStat                 -> COND-STAT
    [_]LoopStat                 -> LOOP-STAT
    [_]InOutStat                -> IN-OUT-STAT
    [_]ProcStat                 -> PROC-STAT
    [_]EmptyStat                -> EMPTY-STAT
    [_]GotoStat                 -> GOTO-STAT
    [_]TestStat                 -> TEST-STAT
hiddens
   sorts STAT-AUX
```

A.7 Module SyntaxTypes

```
%% ------------------
%% Module SyntaxTypes
%% ------------------

imports SyntaxConsts

exports
  sorts TYPE SIMPLE-TYPE ARRAY-TYPE
  context-free syntax
```

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

    "ARRAY" "[" INT-CONST ".." INT-CONST "]" "OF" TYPE   -> ARRAY-TYPE

     SIMPLE-TYPE                                        -> TYPE
     ARRAY-TYPE                                         -> TYPE

variables
    [_]Type[']*            -> TYPE
    [_]SimpleType[']*      -> SIMPLE-TYPE
    [_]ArrayType[']*       -> ARRAY-TYPE
```

B Static Semantics of CLaX

B.1 Module Tc

```
%% ---------
%% Module Tc
%% ---------

imports TcLabel TcNint

exports
  sorts TENV-LIST
  context-free syntax
    tc(PROGRAM)                        -> BOOL
    collect( TENV-LIST, BLOCK )        -> BOOL
    distribute( TENV-LIST, BLOCK )     -> BOOL
    STAT                               -> BOOL
    isbool(EXPR)                       -> BOOL
    unique-decls(TENV*)                -> BOOL
    nonemptyarray(TENV)                -> BOOL
    no-dups(TENV)                      -> BOOL
    get-id(DECL)                       -> ID
    get-type(DECL)                     -> TYPE

equations

(* start: create initial empty type environment *)
(* -------------------------------------------- *)

 [P0] tc(PROGRAM _Id ; _Block .) =  collect([], _Block)

(* build type-environment by processing declarations *)
(* ------------------------------------------------- *)

 [C1] collect(_Tenv* [_Decl*],  DECLARE _EmptyDecl; _Decl*' BEGIN _StatSeq END) =
        collect(_Tenv* [_Decl*], DECLARE _Decl*' BEGIN _StatSeq END)

 [C2] collect(_Tenv* [_Decl*], DECLARE _VarDecl; _Decl*' BEGIN _StatSeq END) =
        collect(_Tenv* [_Decl*;_VarDecl], DECLARE _Decl*' BEGIN _StatSeq END)

 [C3] collect( _Tenv* [_Decl*], DECLARE _LabelDecl; _Decl*' BEGIN _StatSeq END) =
        collect(_Tenv* [_Decl*;_LabelDecl], DECLARE _Decl*' BEGIN _StatSeq END)

 [C4] collect(_Tenv* [_Decl*],DECLARE _ProcDecl; _Decl*' BEGIN _StatSeq END)
      = nonemptyarray(formals(_ProcDecl))
        & collect(_Tenv* [_Decl*;_ProcDecl]
                  formals(_ProcDecl), body-of(_ProcDecl))
        & collect(_Tenv* [_Decl*;_ProcDecl],
                  DECLARE _Decl*' BEGIN _StatSeq END)

 [C5] collect(_Tenv* [_Decl*],  BEGIN _StatSeq END) =
        collect(_Tenv* [_Decl*], DECLARE BEGIN _StatSeq END)
 [C6] collect(_Tenv* _Tenv , DECLARE BEGIN _StatSeq END) =
         check(defines(_StatSeq), _StatSeq)  &
         unique(defines(_StatSeq)) &
         unique-decls(_Tenv) &       (* only need to check current scope! *)
         distribute(_Tenv* _Tenv, BEGIN _StatSeq END)

(* flattening: move statements outside IF and WHILE statements *)
(* ----------------------------------------------------------- *)

  [flat1]  _StatSeq1*; WHILE _Expr DO_StatSeq2 END; _StatSeq3* =
              _StatSeq1*; TEST _Expr END; _StatSeq2; _StatSeq3*

  [flat2]  _StatSeq1*; IF _Expr THEN_StatSeq2 END; _StatSeq3* =
              _StatSeq1*; TEST _Expr END; _StatSeq2; _StatSeq3*

  [flat3]  _StatSeq1*; IF _Expr THEN_StatSeq2 ELSE _StatSeq3 END; _StatSeq4* =
              _StatSeq1*; TEST _Expr END; _StatSeq2; _StatSeq3; _StatSeq4*

(* distribute type environments *)
(* ---------------------------- *)

 [D1] distribute(_Tenv*, BEGIN _Stat ; _StatSeq END) =
       distribute(_Tenv*, BEGIN _Stat END) &
       distribute(_Tenv*, BEGIN _StatSeq END)

 [D2] distribute(_Tenv*, BEGIN END) = true

 [D3] distribute(_Tenv*, BEGIN  _Id : _StatAux  END) =
       islabel( type-of(_Tenv*, _Id)) &
       distribute(_Tenv*, BEGIN  _StatAux  END)

 [D4] distribute(_Tenv*, BEGIN _Expr := _Expr' END) =
       type-of(_Tenv*, _Expr) := type-of(_Tenv*, _Expr')

 [D5] distribute(_Tenv*, BEGIN _Id END) = isproc((type-of(_Tenv*, _Id)) ())

 [D6] distribute(_Tenv*, BEGIN _Id ( _ExprList ) END) =
        isproc(type-of(_Tenv*, _Id) (types-of(_Tenv*, _ExprList)))  &
        vararg(type-of(_Tenv*, _Id) (_ExprList) )

 [D7] distribute(_Tenv*, BEGIN READ ( _Expr ) END) =
       READ ( type-of(_Tenv*, _Expr) )

 [D8] distribute(_Tenv*, BEGIN WRITE ( _Expr ) END) =
       WRITE ( type-of(_Tenv*, _Expr) )

 [D9] distribute(_Tenv*, BEGIN WRITE ( _String ) END) = true

 [D10] distribute(_Tenv*, BEGIN TEST _Expr END END) =
        TEST type-of(_Tenv*, _Expr) END

 [D11] distribute(_Tenv*, BEGIN GOTO _Id END) =
        islabel( type-of(_Tenv*, _Id))

(* lookup of type in type environment *)
(* ---------------------------------- *)

  [T1] type-of(_Tenv*, _Type) = _Type


  [T2] get-id(_Decl) = _Id
      =======================
      type-of(_Tenv* [_Decl*; _Decl ; _Decl*'], _Id) = get-type(_Decl)

  [default-T3] (* default rule *)
```

```
      _Id != get-id(_Decl)
      =============================================
      type-of(_Tenv* [_Decl*; _Decl; _Decl*'], _Id)
      = type-of(_Tenv* [_Decl*; _Decl*'], _Id)


  [T4] type-of(_Tenv*[],  _Id) = type-of(_Tenv*, _Id)

  [T5] type-of(_Tenv* _Tenv, (_Var [ _Expr' ])) =
        (type-of(_Tenv* _Tenv, _Var))[type-of(_Tenv* _Tenv, _Expr')]

(* retrieve type of DECL *)
(* -------------------- *)

  [G1] get-type( _Id:_Type ) = _Type
  [G2] get-type( _Id:LABEL ) = LABEL-TYPE
  [G3] get-type( _ProcHead; _Block ) = signature(_ProcHead)


(* get-id: get the identifier of a DECL *)
(* ---------------------------------- *)

  [Id1] get-id( _Id:_Type) = _Id
  [Id2] get-id( _Id:LABEL) = _Id
  [Id3] get-id( PROCEDURE _Id; _Block ) = _Id
  [Id4] get-id( PROCEDURE _Id(_Formal+); _Block ) = _Id

(* reduce correct cases to "true" *)
(* ------------------------------ *)

  [R1]  _SimpleType := _SimpleType  = true
  [R2]  REAL := INTEGER = true
  [R3]  READ ( _SimpleType ) = true
  [R4]  WRITE ( _SimpleType ) = true
  [R5]  TEST BOOLEAN END = true

(* other stuff *)
(* ----------- *)

 [IB0] isbool(BOOLEAN) = true

(* check for uniqueness of declarations *)
(* ---------------------------------- *)

  [U1] unique-decls(_Tenv) = true when no-dups(_Tenv) != false

  [N1] get-id(_Decl) = get-id(_Decl')
       =======================================================
       no-dups([_Decl*;_Decl;_Decl*';_Decl';_Decl*'']) = false

(* check array range *)
(* ----------------- *)

 [NA0] nonemptyarray([_Id : _SimpleType]) = true
 [NA1] nonemptyarray([_LabelDecl]) = 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+])
```

B.2 Module TcBooleans

```
%% -----------------
%% Module TcBooleans
%% -----------------

exports
```

```
   sorts BOOL BOOL-CON
   context-free syntax
      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

%%There are no equations in TcBooleans module.
```

B.3 Module TcSyntaxExt

```
%% ------------------
%% Module TcSyntaxExt   (Syntax Extentions)
%% ------------------

imports SyntaxProgram

exports
  context-free syntax
   EXPR ":=" EXPR               -> ASSIGN-STAT
   EXPR "[" EXPR "]"            -> EXPR
   "READ"  "(" EXPR ")"         -> IN-OUT-STAT

priorities
  {VARIABLE ":=" EXPR           -> ASSIGN-STAT,
   VARIABLE "[" EXPR "]"        -> VARIABLE,
   "READ"  "(" VARIABLE ")"     -> IN-OUT-STAT}
  >
  {EXPR ":=" EXPR               -> ASSIGN-STAT,
   EXPR "[" EXPR "]"            -> EXPR,
   "READ"  "(" EXPR ")"         -> IN-OUT-STAT}

equations

[t1] _Var := _Expr  = _Expr’ := _Expr   when _Expr’ = _Var
[t2] READ ( _Var )  = READ ( _Expr )    when _Expr = _Var
```

B.4 Module TcExpr

```
%% -------------
%% Module TcExpr
%% -------------

imports TcTenv

exports
    sorts  AOP BOP UOP OP

context-free syntax
    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] _Expr _Cop _Expr'  = _Expr _Op _Expr'  when _Op = _Cop
 [t1] _Expr _Aop1 _Expr' = _Expr _Op _Expr'  when _Op = _Aop1
 [t2] _Expr _Aop2 _Expr' = _Expr _Op _Expr'  when _Op = _Aop2
 [t3] _Expr _Bop1 _Expr' = _Expr _Op _Expr'  when _Op = _Bop1
 [t4] _Expr _Bop2 _Expr' = _Expr _Op _Expr'  when _Op = _Bop2
 [t5] _Ubop _Expr = _Uop _Expr when _Uop = _Ubop
 [t6] _Uaop _Expr = _Uop _Expr when _Uop = _Uaop

 [t11] _Uop BOOLEAN = BOOLEAN   when _Uop = NOT
 [t12] _Uop INTEGER = INTEGER   when _Uop != NOT
 [t13] _Uop REAL    = REAL      when _Uop != NOT
 [t14] INTEGER _Op INTEGER = INTEGER   when _Op = _Aop
 [t15] REAL    _Op REAL    = REAL      when _Op = _Aop, _Op != %
 [t16] BOOLEAN _Op BOOLEAN  = BOOLEAN  when _Op = _Bop

 [t17] _SimpleType _Op _SimpleType = BOOLEAN when _Op = _Cop

 [V0] type-of(_Tenv*, _Uop _Expr) =
        _Uop (type-of(_Tenv*, _Expr))

 [V1] type-of(_Tenv*, (_Expr _Op _Expr')) =
       (type-of(_Tenv*,_Expr)) _Op (type-of(_Tenv*,_Expr'))
```

B.5 Module TcLabel

```
%% --------------
%% Module TcLabel
%% --------------

imports TcProc

exports
  sorts LABEL-LIST
  context-free syntax
    ID*                                 -> LABEL-LIST
    defines( STAT-SEQ )                 -> LABEL-LIST
    unique( LABEL-LIST )                -> BOOL
    no-dups( LABEL-LIST )               -> BOOL
    check( LABEL-LIST, STAT-SEQ )       -> BOOL
    islabel(EXPR)                       -> BOOL

  variables
   [_]Id"*"['']*         -> ID*
   [_]Id"+"['']          -> ID+
   [_]Labels['']*        -> ID*
   [_]Labels"+"          -> ID+
   [_]LabelList['']*     -> LABEL-LIST

equations

 [L0] defines(_StatAux ) =
 [L1] defines(_Id : _StatAux ) = _Id
 [L2] _Labels’ = defines(_Stat ),
      _Labels’’ = defines(_StatSeq),
      _Labels = _Labels’ _Labels’’
      ============================
      defines(_Stat ; _StatSeq) = _Labels

 [IsL0] islabel(LABEL-TYPE) = true

(* check if there are any multiply defined labels *)

 [U1] unique(_LabelList) = true when no-dups(_LabelList) != false

 [N1] no-dups(_Id* _Id _Id*’ _Id _Id*’’) = false


(* check if labels that are being jumped to are defined *)

 [C0]  check(_LabelList, _Stat;_StatSeq) =
         check(_LabelList, _Stat) & check(_LabelList, _StatSeq)

 [C1]  check(_LabelList, _AssignStat) = true
 [C2]  check(_LabelList, _TestStat) = true
 [C3]  check(_LabelList, _InOutStat) =  true
 [C4]  check(_LabelList, _ProcStat) = true
 [C5]  check(_LabelList, _EmptyStat) = true
 [C6]  check(_LabelList, _Id: _StatAux) = check(_LabelList, _StatAux)

 [C7]  check(_Id* _Id _Id*’, GOTO _Id) = true
 [default-C8]
          _Id != _Id’
        ================
        check(_Id* _Id’ _Id*’, GOTO _Id) = check(_Id* _Id*’, GOTO _Id)
```

B.6 Module TcNint

```
%% -------------
%% Module TcNint
%% -------------

imports TcBooleans SyntaxConsts

exports
  context-free syntax
    INT-CONST islessthan INT-CONST  -> BOOL
hiddens
  sorts INT INT-CON POS NEG NAT AUX

  lexical syntax
    [1-9][0-9]*                 -> POS  %% constr
    [+\-0]                      -> AUX

  context-free syntax
    toint INT-CONST             -> INT
    "-" POS                     -> NEG  %% constr
    "0"                         -> NAT  %% constr
    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 syntax
    INT ";" INT                  -> INT  {left}  %% concatenation
    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] S0=1   [S2] S1=2   [S3] S2=3   [S4] S3=4
  [S5]  S4=5   [S6] S5=6   [S7] S6=7   [S8] S7=8   [S9] S8=9

  [P0]  P0=-1  [P1] P1=0   [P2] P2=1   [P3] P3=2   [P4] P4=3
  [P5]  P5=4   [P6] P6=5   [P7] P7=6   [P8] P8=7   [P9] P9 = 8

  [11]  Sn  =  hd(n)   ; S tl(n)        when tl(n) != 9, bigpos?(n) = true
  [12]  Sn  =  S hd(n) ; 0              when tl(n)  = 9
  [13]  S(-n)  = - n’                   when n’ = Pn
  [14]  Pn  =  hd(n)   ; P tl(n)        when tl(n) != 0, bigpos?(n) = true
  [15]  Pn  =  P hd(n) ; 9              when tl(n)  = 0, bigpos?(n) = true
  [16]  P(-n)  = - n’                   when n’ = Sn

  [l1]  0 < 0     =  false
  [l2]  0 < n     =  true
  [l3]  0 < z     =  false
  [l4]  n < x     =  Pn < Px
  [l5]  z < x     =  Sz < Sx

  [h1]  hd( pos(c+ c) ) = pos(c+)
  [h2]  hd( pos( c )  ) = 0

  [t1]   tl(pos(c+  "0")) = 0
  [t2]  tl(pos(c* c)) = pos(c)  when pos(c* c) != pos(c* "0")

  [o1]  bigpos?(pos(c+ c)) = true

  [o2]  bigpos?(pos(c)) = false

  [o2]  pos(c+) ; pos(c’+) = pos(c+ c’+)
  [o3]  pos(c+) ; 0 = pos(c+ "0")
  [o4]  0 ; x = x

  [e1] - (n) = - n
  [e2] - - n = n
  [e3] - 0 = 0

  [T0] toint int-const("+" c+) = toint int-const(c+)
  [T1] toint int-const("-" c+) = - toint int-const(c+)
  [T2] toint int-const("0" c+) = toint int-const(c+)
  [T3] toint int-const(c) = pos(c)
  [T4] toint int-const(c c+) = pos(c c+)
       when  aux(c) != aux("0"),
             aux(c) != aux("+"),
             aux(c) != aux("-")

  [L0] _IntConst islessthan _IntConst’ = true
       when toint _IntConst < toint _IntConst’ = true
```

B.7 Module TcProc

```
%% -------------
%% Module TcProc
%% -------------

imports TcExpr

exports
  sorts PROC-TYPE VTYPE TYPE-LIST
  context-free syntax
    "LABEL-TYPE"                           -> TYPE
    "PROC""(" {VTYPE ";"}* ")"             -> PROC-TYPE
    PROC-TYPE                              -> TYPE
    {TYPE ";"}*                            -> TYPE-LIST
    types-of(TENV-LIST, {EXPR ","}*)       -> TYPE-LIST
    formals(PROC-DECL)                     -> TENV
    signature(PROC-HEAD)                   -> PROC-TYPE
    TYPE                                   -> VTYPE
    "VAR" TYPE                             -> VTYPE
    vtype(FORMAL)                          -> VTYPE
    isproc"(" EXPR "(" TYPE-LIST ")" ")"   -> BOOL
    vararg"(" EXPR "(" {EXPR ","}* ")" ")" -> BOOL
    body-of( PROC-DECL )                   -> BLOCK
  variables
    [_]Type"*"[’]*               -> {TYPE ";"}*
    [_]TypeList[’]*              -> {TYPE ";"}*
    [_]ProcType[’]*              -> PROC-TYPE
    [_]Vtype[’]*                 -> VTYPE
    [_]VtypeList[’]*             -> {VTYPE ";"}*

equations

(* compute a list of types for a list of expressions *)

  [T1] types-of(_TenvList, _Expr) = type-of(_TenvList, _Expr)

  [T2] type-of(_TenvList, _Expr) = _Type,
       types-of(_TenvList, _ExprList) = _Type*
       ===========================================
       types-of(_TenvList, _Expr,_ExprList) = _Type; _Type*

(* create TENV with the formal parameters of a procedure *)

 [F0] formals(PROCEDURE _Id; _Block) = []
 [F1] formals(PROCEDURE _Id (_VarDecl); _Block) = [_VarDecl]
 [F2] formals(PROCEDURE _Id (VAR _VarDecl); _Block) = [_VarDecl]
 [F3] formals(PROCEDURE _Id (_Formal); _Block) = [_Decl],
      formals(PROCEDURE _Id (_Formal+); _Block) = [_Decl+]
      ========================================
      formals(PROCEDURE _Id (_Formal;_Formal+); _Block) = [_Decl;_Decl+]

 [S0] signature(PROCEDURE _Id) = PROC ()
 [S1] signature(PROCEDURE _Id (_Formal)) = PROC (vtype(_Formal))
 [S2] signature(PROCEDURE Id (_Formal+)) = PROC(_VtypeList)  (* Id is a const :-*)
      ===============================================================
      signature(PROCEDURE _Id (_Formal; _Formal+)) = 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

 [B1] body-of( _ProcHead; _Block ) = _Block
```

B.8 Module TcTenv

```
%% -------------
%% Module TcTenv
%% -------------

imports TcSyntaxExt TcBooleans

exports
  sorts TENV
  context-free syntax
    TYPE                         -> EXPR
    "[" {DECL ";"}* "]"          -> TENV
    TENV*                        -> TENV-LIST
    type-of(TENV-LIST, EXPR)     -> TYPE

 variables
    [_]C"*"                       -> TENV*
    [_]D"*"[’]*                   -> {DECL ";"}*
    [_]D[’]*                      -> DECL
    [_]D"+"[’]*                   -> {DECL ";"}+
    [_]Tenv[’]*                   -> TENV
    [_]Tenv"*"[’]*                -> TENV*
    [_]Tenv"+"[’]*                -> TENV+
    [_]TenvList[’]*               -> TENV-LIST

equations

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

[4] (ARRAY[_IntConst .. _IntConst’] OF _Type) [ INTEGER ] = _Type
```

B.9 Module TcErrorsSyntax

```
%% ---------------------
%% Module TcErrorsSyntax
%% ---------------------

imports SyntaxProgram

exports
  sorts MESSAGE MSG-LIST
  context-free syntax
    errors(PROGRAM)                 -> MSG-LIST
    { MESSAGE ";" }+                -> MSG-LIST

    no-errors                       -> MESSAGE
    err(EXPR)                       -> MESSAGE
    incomp(MESSAGE)                 -> MESSAGE
    TYPE                                -> MESSAGE


    undeclared-identifier ID                                           -> MESSAGE
    incompatible-operands(EXPR)                                        -> MESSAGE
    incompatible-array-access(EXPR)                                    -> MESSAGE
    used-as-operand EXPR                                               -> MESSAGE
    label-used-as-operand                                              -> MESSAGE
    assignment-incompatible BOOL                                       -> MESSAGE
    cannot-assign-to EXPR                                              -> MESSAGE
    cannot-assign-to-label                                             -> MESSAGE
    "Boolean-expected-in-cond" MESSAGE                                 -> MESSAGE
    undeclared-procedure-called ID                                     -> MESSAGE
    in-call expected-arg VTYPE found-arg MESSAGE                       -> MESSAGE
    in-call expected-no-more-args-but-found TYPE-LIST                  -> MESSAGE
    in-call 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
    undefined-label  ID                                                -> MESSAGE
    undeclared-label  ID                                               -> MESSAGE
    multiple-declaration-in-same-scope ID                              -> MESSAGE
    unary-operator UOP not-allowed-on-operand-of-type TYPE             -> MESSAGE
```

B.10 Module TcErrors

```
%% ---------------
%% Module TcErrors
%% ---------------

imports Tc TcErrorsSyntax

exports
  context-free syntax
    errors(BOOL)                -> MSG-LIST
    err(TYPE-LIST)              -> MSG-LIST
    MSG-LIST "::" MSG-LIST      -> MSG-LIST

  variables
    [_]Msg[’]*      -> MESSAGE
    [_]MsgList[’]*  -> {MESSAGE ";"}*

equations

  [Top] errors(_Program) = errors(tc(_Program))

  [M0] _MsgList ; no-errors ; _MsgList’   = _MsgList ; _MsgList’
      when _MsgList ; _MsgList’ = _MsgList’’ ; _Msg

  [E0] errors(true)  = no-errors

  [E1] errors(Bool1 & Bool2) = errors(Bool1) :: errors(Bool2)

  [M11] _MsgList :: _MsgList’ = _MsgList ; _MsgList’

  [S01] errors(_SimpleType := _SimpleType’) =
          assignment-incompatible (_SimpleType := _SimpleType’)

  [S02] errors(_Expr[_Expr’] := _Expr’’) = err( _Expr[_Expr’] )

  [S03] errors(LABEL-TYPE := _Expr) = cannot-assign-to-label

  [S04] errors(_SimpleType := _Expr[_Expr’]) = incomp(err(_Expr[_Expr’]))
```

```
[S05] errors(_SimpleType := _Expr _Op _Expr') = incomp(err(_Expr _Op _Expr'))

[S06] errors(_SimpleType := _Uop _Expr) = incomp(err(_Uop _Expr))

[S07] errors(type-of(,_Id) :=  _Expr) = undeclared-identifier _Id

[S08] errors(_SimpleType := type-of(,_Id) ) = undeclared-identifier _Id

[S11] errors(isproc(type-of(,_Id)(_TypeList))) = undeclared-procedure-called _Id
[S12] errors(isproc(PROC(_Vtype; _VtypeList) (_Type;_TypeList) ))
       = in-call expected-arg _Vtype found-arg incomp(err(_Type))
[S13] errors(isproc(PROC() (_Type;_TypeList) ))
       = in-call expected-no-more-args-but-found _Type;_TypeList
[S14] errors(vararg(type-of(,_Id)(_ExprList))) = undeclared-procedure-called _Id
[S15] errors(vararg(PROC(_Vtype; _VtypeList) (_Expr, _ExprList) ))
       = in-call expected-variable-arg _Vtype found-arg _Expr

[S2] errors(TEST _Expr END) = Boolean-expected-in-cond incomp(err(_Expr))

[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 type-of(,_Id)) = err(type-of(,_Id))
[E0y] err(_Expr _Op type-of(,_Id)) = err(type-of(,_Id))
[E0z] err((type-of(,_Id)) _Op _Expr) = err(type-of(,_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] err(_Expr') = incompatible-array-access(_Expr)
      ============================================
      err(_Expr' _Op _Expr'') = incompatible-array-access(_Expr)

[E1f] err(_Expr'') = incompatible-array-access(_Expr)
      ============================================
      err(_Expr' _Op _Expr'') = incompatible-array-access(_Expr)

[E2a] err(_Uop _Type) = unary-operator _Uop not-allowed-on-operand-of-type _Type

[E2b] err(type-of(,_Id)) = undeclared-identifier _Id

[O1] incomp(undeclared-identifier _Id) = undeclared-identifier _Id
[O2] incomp(err(_SimpleType)) = _SimpleType
[O3] incomp(err(_SimpleType _Op _SimpleType'))
     = incompatible-operands(_SimpleType _Op _SimpleType')

[O4] incomp(err(LABEL-TYPE _Op _Expr)) = label-used-as-operand
[O5] incomp(err(_Expr _Op LABEL-TYPE)) = label-used-as-operand

[O6] err(_Expr') = err(_Expr)
     ======================
     incomp(incompatible-array-access(_Expr)) = incompatible-array-access(_Expr')

[O7] incomp(unary-operator _Uop not-allowed-on-operand-of-type _Type) =
       unary-operator _Uop not-allowed-on-operand-of-type _Type

[O8] incomp(err((_Uop _Expr) _Op _Expr')) = incomp(err(_Uop _Expr))

[O9] incomp(err(_Expr _Op (_Uop _Expr'))) = incomp(err(_Uop _Expr'))

[O10] incomp(err( (_Expr _Op _Expr') _Op' _Expr'' )) =
      incomp(err( (_Expr _Op _Expr’) ))

[O11] incomp(err( _Expr _Op (_Expr’ _Op’ _Expr’’))) =
      incomp(err( (_Expr’ _Op _Expr’’) ))

[O12] incomp(err( type-of(,_Id) [_Expr ])) =
      incomp(err( type-of(,_Id) ))

[L0] errors(unique(_Id*_Id _Id*’ _Id _Id*’’)) = multiply-defined-label _Id

[L1] errors(check(, GOTO _Id)) = undefined-label _Id

[L2] errors(islabel(type-of(,_Id))) = undeclared-label _Id

[UE0] get-id(_Decl) = get-id(_Decl’)
      =======================================
      errors(unique-decls([_D*;_Decl;_D*’; _Decl’; _D*’’])) =
        multiple-declaration-in-same-scope get-id(_Decl)

[AL0] errors(_IntConst islessthan _IntConst’)
      = array-decl-must-have-positive-size(_IntConst .. _IntConst’)
```

C Earlier Tokenization

Earlier, when we used origin tracking to generate error reporters, Module `SyntaxTokens` defined sorts and tokens which represent reserved words and special characters for which origin tracking is desired (see Section 6).The module `SyntaxTokens`, in general, would not be used in a straightforward syntax specification; it was used for the purpose increasing the origin tracking hits.

```
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 syntax

      "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


module SyntaxProgram

%% This module uses module SyntaxTokens. Note that
%% the keywords DECLARE, BEGIN and END are sorts with values
%% "DECLARE", "BEGIN" and "END" respectively
```

```
imports SyntaxHeaders SyntaxStats

exports
  sorts PROGRAM   %% BLOCK defined in module SyntaxHeaders
  context-free syntax

    DECLARE DECL-LIST BEGIN STAT-SEQ END  -> BLOCK
    BEGIN STAT-SEQ END                    -> BLOCK

    "PROGRAM" ID ";" BLOCK "."            -> PROGRAM

  variables
    [_]Program[0-9’]*   -> PROGRAM
```