A Slicing-Based Approach for Locating Type Errors

T.B. Dinesh, Frank Tip

Software Engineering (SEN)

SEN-R9824 October 1998

Report SEN-R9824
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

A Slicing-Based Approach for Locating Type Errors

T. B. Dinesh
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
T.B.Dinesh@cwi.nl

Frank Tip
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
tip@watson.ibm.com

Abstract

The effectiveness of a type checking tool strongly depends on the accuracy of the positional information that is associated with type errors. We present an approach where the location associated with an error message e is defined as a *slice* P_e of the program P being type checked. We show that this approach yields highly accurate positional information: P_e is a program that contains precisely those program constructs in P that caused error e. Semantically, we have the interesting property that type checking P_e is guaranteed to produce the same error e. Our approach is completely language-independent, and has been implemented for a significant subset of Pascal.

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

1991 Computing Reviews Classification System: **D.3.4.** [**Programming languages**]: Processors—*Translator writing systems and compiler generators*; **D.2.1.**[**Software engineering**]: Requirements/Specifications—*Languages, Tools*; **F.3.1.** [**Logics and meanings of programs**]: Specifying and verifying and reasoning about Programs—*Specification techniques.*

Key Words & Phrases: Semantics-based tool generation, Program slicing, Type checking, Static semantics specification, Pascal.

Note: This research was supported in part by the Netherlands Organization for Scientific Research (NWO) under the *Generic Tools for Program Analysis and Optimization* project.
This is a revised and expanded version of a paper that was presented at the USENIX Conference on Domain Specific Languages (DSL'97) [14]. The paper also borrows some material from a case study that was presented at the 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97) [13].

1 Introduction

Type checkers are tools for determining the constructs in a program that do not conform to a language's type system. Type checkers are usually incorporated in interactive programming environments where they provide programmers with rapid feedback on the nature and locations of type errors. The effectiveness of a type checker crucially depends on two factors:

- The "informativeness" of the type errors reported by the tool.
- The quality of the positional information provided for type errors.

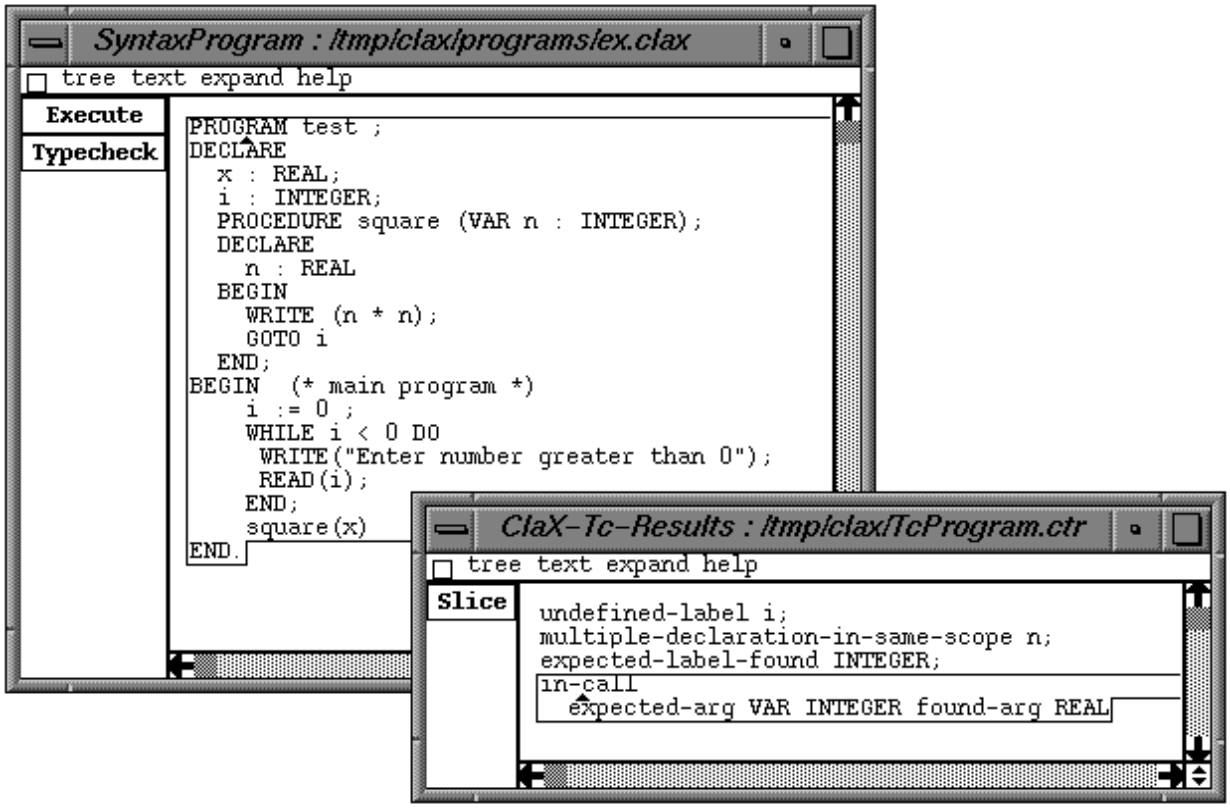

Figure 1: The CLaX environment. The top window is a program editor with two buttons attached to it for invoking a type checker and an interpreter, respectively. The bottom window shows a list of four type errors reported by the type checker. After selecting an error message in the bottom window, the **Slice** button can be pressed to obtain the associated slice.

We believe that the second factor is especially important. For example, consider an assignment statement $x = y$ where x and y are of two incompatible types. What is the source of the error? Specifically, one might ask whether the assignment construct itself is "causing" the error, or if the declarations of x and y, where the incompatible types are introduced, constitute the error's real "source". As another example, consider a situation where a label is defined twice inside some procedure. Ideally, the location of this error would comprise *both* occurrences of the label.

We pursue a semantically well-founded approach to answer the question of what the location of a type error should be. In this approach, the behavior of a type checker is algebraically specified by way of a set of conditional equations [2], which are interpreted as a conditional term rewriting system (CTRS) [26]. These rewriting rules express the type checking process by transforming a program's abstract syntax tree (AST) into a list of error messages. We use dynamic dependence tracking [17, 18] to determine a *slice* of the original program as the positional information associated with an error message. This approach has the following advantages:

- The tracking of positional information is completely language-independent and automated; no information needs to be maintained at the specification level.
- Unlike previous approaches [10, 34], no constraints are imposed on the style in which the type checker specification is written. Error locations are always available, regardless of the specification style being used.

- The approach is semantically well-founded. If type checking a program P yields an error message e, then the location P_e associated with e is a projection of P that, when type checked, will produce the same error message e. For details about semantic properties of slices, the reader is referred to [17, 18].

Although positional information is always available for any error message, the *accuracy*[1] of these locations depends inversely on the degree to which the specified type checker's behavior is deterministic. This issue will be explored in Section 4.3.

We have implemented a prototype type checking system using the ASF+SDF Meta-environment [25, 34], a programming environment generator that implements algebraic specifications by way of term rewriting. Dependence tracking was previously implemented in the ASF+SDF system's term rewriting engine for the purpose of supporting dynamic slicing in generated debugging environments [30]. Fig. 1 shows a snapshot of a type checking environment for the language CLaX, a Pascal-like language. The most interesting features of CLaX are: nested scopes, overloaded operators, arrays, goto statements, and procedures with reference and value parameters. The top window of Fig. 1 is a program editor, which has two buttons labeled '**TypeCheck**' and '**Execute**' attached to it, for invoking the type checker and the interpreter, respectively. The bottom window shows a list of four error messages reported by the type checker for this program.

1. The first error, `undefined-label i`, indicates that the program contains a reference to a label `i`, but there is no statement with label `i` in the same scope.

2. The second error message, `multiple-declaration-in-same-scope n`, points out that an identifier `n` is declared more than once in the same scope.

3. The third error, `expected-label-found INTEGER`, indicates that the program contains an identifier that has been declared as an integer, but which is used as a label.

4. The fourth error, `in-call expected-arg VAR INTEGER found-arg REAL`, points out a type error in a procedure call. In particular, that a procedure is called with a argument type REAL when it was expecting an argument of type INTEGER.

Note that these error messages do not provide any information as to *where* the type violations occurred in the program text.

However, positional information may be obtained by selecting an error message and clicking on the '**Slice**' button. In Fig. 2(a)–(d), the slices obtained for each of the four error messages of Fig. 1 are shown[2]. Each slice is a view of the program's source indicating the program parts that contribute to the selected error. Placeholders, indicated by '<?>' in the figure, indicate program components that do not contribute to the error under consideration. The semantics of "not contributing towards a certain error message" may be characterized informally as follows: If a placeholder in the slice with respect to an error e is replaced with a program component of the same kind[3], type checking the resulting program is guaranteed to produce the same error e.

1. Fig. 2(a) shows the slice for the `undefined- label` error. Clearly, the `GOTO i` statement is the source of the error, because there is no statement with label `i`.

2. Fig. 2(b) shows the slice obtained for the `multiple-declaration-in-same -scope` error. The problem here is that `n` is a parameter as well as a local variable of procedure `square`. Note that both declarations of `n` occur in the slice.

3. Fig. 2(c) shows the slice obtained for the `expected-label-found INTEGER` error. Note that, in addition to the `GOTO i` statement and the declaration of `i` as an `INTEGER`, all declarations in the inner scope appear in the slice. Informally, this is the case because replacing any of these declarations by

[1] Accuracy indicates the quality of the slice obtained. Generally, "small" slices, which contain few program constructs, are desirable because they convey the most insightful information.

[2] An alternative way for displaying slices would be to highlight the corresponding text areas in the program editor of Fig. 1.

[3] Although all placeholders are displayed as '<?>', placeholders are typed. In order to preserve syntactic validity of the program, an expression placeholder may only be replaced by another expression, an unlabeled-statement placeholder may only be replaced by another unlabeled-statement, etc.

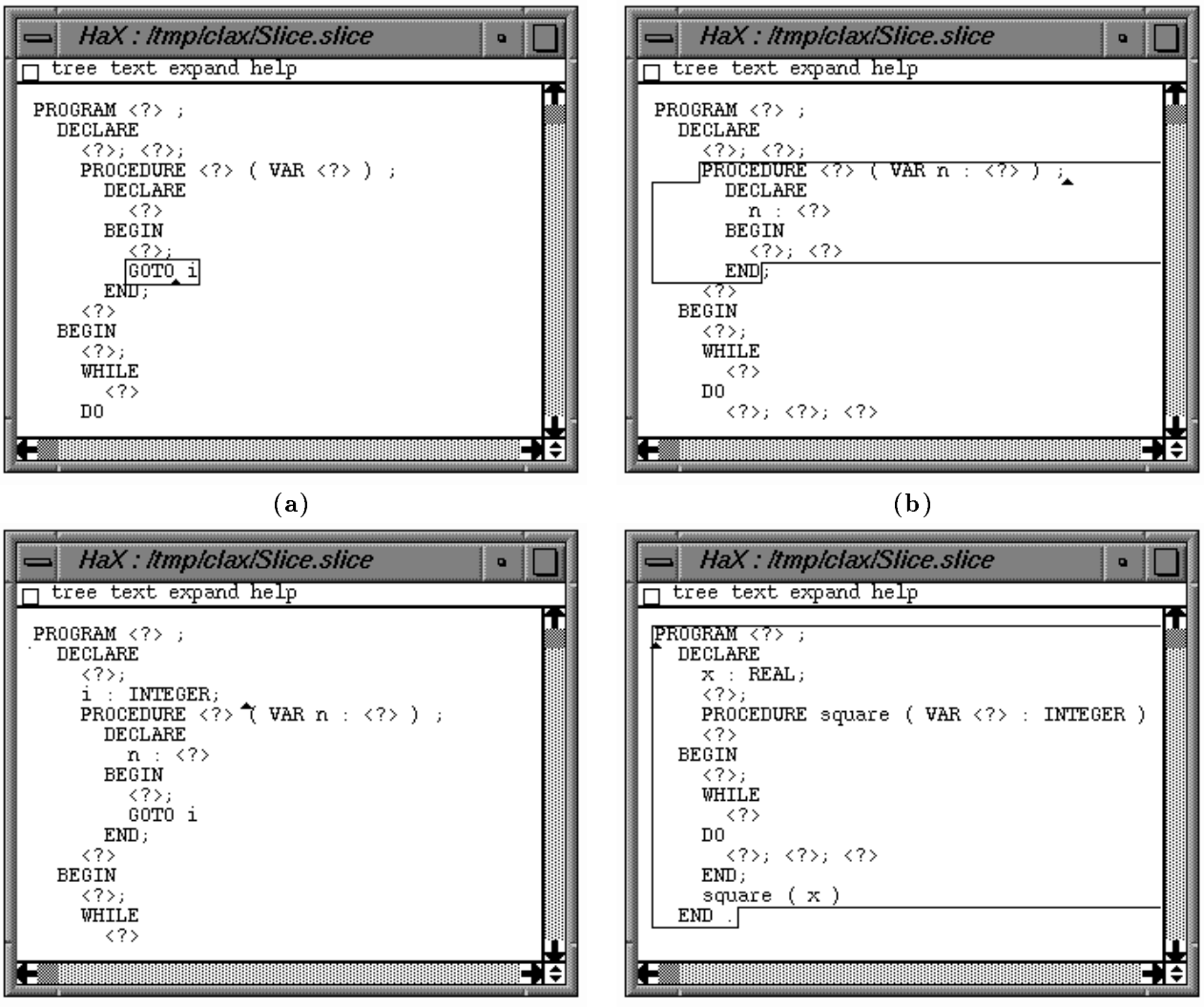

Figure 2: Slices reported by the CLaX environment for each of the type errors of Fig. 1.

declarations for variable `i` may affect the outcome of the type checking process, in the sense that the `expected-label-found INTEGER` error would no longer occur.

4. Fig. 2(d) shows the slice obtained for the `in-call expected-arg VAR INTEGER found-arg REAL` error. Observe that the slice precisely indicates the program components responsible for this problem: (i) the call site `square(x)` that gave rise to the problem, (ii) the type, `INTEGER`, of `square`'s formal parameter (note that the name of this parameter is irrelevant), and (iii) the declaration of variable `x` as a `REAL`.

The reader may observe at this point that, in addition to the program constructs responsible for a type error, a slice generally also contains certain structural information such as `BEGIN` and `END` keywords and declaration and statement list separators that are not directly related to an error. The occurrence of this structural information is due to the way slices are computed. If desired, displaying this information could easily be suppressed to a large extent. For example, removal of all `BEGIN`, `END`, and `DECLARE` keywords and list separators from the computed slices would reduce the amount of "noise" considerably. In certain cases, slices may contain `IF` or `WHILE` statements whose condition and body are omitted from the slice (see, e.g., Fig. 2(d)). Such constructs can also be removed from the slice without affecting the semantic content. We consider slice postprocessing to be primarily a user-interface issue, which is outside the scope of this paper.

The remainder of the paper is organized as follows. In Section 2, related work is discussed. In particular, the relation to our previous work on origin tracking is discussed, and the slice notion introduced in the present paper is compared with the traditional notion of a program slice. Section 3 presents our approach for specifying type checkers. In Section 4, the use of term rewriting for executing specifications, as well as *dependence tracking*, the mechanism for computing slices are presented. Section 5 presents a case study in which our techniques are applied to CLaX, a Pascal-like imperative language. We describe some experiments we conducted using the CLaX prototype, in particular, the effect of certain specification changes on the accuracy of the computed slices is discussed. Conclusions and possible directions for future work are stated in Section 6.

2 Related Work

The work presented in this paper is closely related to earlier work by the same authors. The CLaX type checker [12] was developed in the context of the COMPARE (compiler generation for parallel machines) project, which was part of the European Union's ESPRIT-II program. We originally used *origin tracking* [35] to associate source locations with type errors. Origin tracking is similar in spirit to dependence tracking in the sense that it establishes relationships between subterms of terms that occur in a rewriting process. The key difference between the two techniques is that origin tracking establishes relationships between *equal* subterms (either syntactically equal, or equal in the algebraic sense), whereas dependence tracking determines for each subterm the context needed to create it. The use of origin tracking for obtaining positional information was further investigated in [10, 11]. Although the results were encouraging (in terms of accuracy of positional information), origin tracking was found to impose restrictions on the style in which the type checker specification was written. Since origin tracking only establishes relationships between equal terms, the error messages generated by the type checker must contain fragments that literally occur in the program source; otherwise, positional information is unavailable. In [10, 11], this problem was circumvented by tokenization, i.e., using an applicative syntax structure and rewriting the specification in such a way that error messages always contain literal fragments of program source, which guarantees the non-emptyness of origins. Modification of the type checker specification resulted in adequate positional information for type errors. By contrast, our approach does not require any modifications to specifications at all. In the previous section, we have described techniques for improving the quality of positional information by avoiding determinism in specifications, but it should be emphasized that such improvements are completely optional.

The dependence tracking relation we use for obtaining positional information was developed by Field and Tip [17, 18] for the purpose of computing program slices. A *program slice* [37, 38, 31] 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 [31].

By applying dependence tracking to different rewriting systems, various types of slices can be obtained. In [16] programs are translated to an intermediate graph representation named PIM [15, 4]. 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 [30], 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 find 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. It would be interesting to investigate whether there are other abstract program properties for which a sensible slice notion exists.

Another approach to providing positional information for type errors is pursued by van Deursen [33, 32]. Van Deursen investigates a restricted class of algebraic specifications called Primitive Recursive Schemes (PRSs). In a PRS, there is an explicit distinction between constructor functions that represent language constructs, and other functions that process these constructs. Van Deursen extends the origin tracking notion of [35] by taking this additional structure into account, which enables the computation of more precise origins.

Heering [21] has experimented with higher-order algebraic specifications to specify static semantics. We believe that the approach of this paper would work very well with higher-order specifications, since these allow one to avoid deterministic behavior, which adversely affects slice accuracy. However, this would require extension of the dependence tracking notion of [17, 18] to higher-order rewriting systems.

Fraer [20] uses a variation on origin tracking [7, 6, 8] to trace the origins of assertions in a program verification system. In cases where an assertion cannot be proved, origin tracking enables one to determine the assertions and program components that contributed to the failure of the verification condition.

Flanagan et al. [19] have developed MrSpidey, an interactive debugger for Scheme, which performs a static analysis of the program to determine operations that may lead to run-time errors. In this analysis, a set of abstract values is determined for each program construct, which represents the set of run-times values that may be generated at that point. These abstract values are obtained by deriving a set of constraints from the program in a syntax-directed fashion, which approximate the data flow in the program. In addition, a value flow graph is constructed, which models the flow of values between program points. MrSpidey has an interactive user-interface that allows one to visually inspect values as well as flow-relationships.

3 Specification of Static Semantics and Type Checking

A *static semantics* specification only determines the validity of a program and is not concerned with pragmatic issues such as the source location where a violation of the static semantics occurred, or even what program construct caused the violation. A *type checker* specification typically uses the static semantics specification as a guideline, and specifies the presentation and source location of type errors in invalid programs. Adding such reporting information to a static semantics specification is a cumbersome and error-prone task, because keeping track of positional information can be nontrivial, especially if multiple program fragments together constitute a type error.

In [12], we introduced an abstract interpretation style for writing static semantics specifications. In a nutshell, this style advocates the following:

- reducing program constructs to their *type*,
- evaluating type expressions at an abstract level, and
- only specifying the type-correct cases.

[Eq1]	tc(begin *Decls Stats* end)	= dist(*Stats*, tenv(*Decls*))
[Eq2]	dist($Stat_1$; $Stat_2$, *Tenv*)	= dist($Stat_1$, *Tenv*); dist($Stat_2$, *Tenv*)
[Eq3]	dist(*Id* := *Exp*, *Tenv*)	= dist(*Id*, *Tenv*) := dist(*Exp*, *Tenv*)
[Eq4]	dist(Exp_1 + Exp_2, *Tenv*)	= dist(Exp_1, *Tenv*) + dist(Exp_2, *Tenv*)
[Eq5]	dist(*Id*, *Tenv*)	= type-of(*Id*, *Tenv*)
[Eq6]	type-of(*Id*, tenv(T_1^*; *Id*: *Type*; T_2^*))	= *Type*
[Eq7]	natural + natural	= natural
[Eq8]	natural := natural	= "correct"

Figure 3: Static semantics specification for determining the validity of assignments.

[Er1]	msgs($Stat_1$; $Stat_2$)	= msgs($Stat_1$);msgs($Stat_2$)
[Er2]	msgs("correct")	= "No errors"
[Er3]	*Msg**; "No errors"; *Msg*'*	= *Msg**; *Msg*'*
[Er4]	msgs(T_1 := T_2) when simpletype(T_2) $\neq$ true	= msgs(T_2)
[Er5]	msgs(T_1 := T_2) when simpletype(T_2) = true	= "Incompatible types in assignment."
[Er6]	msgs(T_1 + T_2)	= "Operands of + should have the same type."
[Er7]	simpletype(natural)	= true
[Er8]	simpletype(string)	= true

Figure 4: Postprocessing to obtain human-readable messages.

Operationally, the static semantics specification describes a transformation of a program to a set of type-expressions for program constructs that are type-incorrect.

Fig. 3 shows a tiny static semantics specification for determining the validity of assignment statements in straight-line flow programs. The reader should be aware that this specification only serves to illustrate the general style of specifying a static semantics and is incomplete; for example, it does not verify if variables are declared more than once. Equation **[Eq1]** defines a top-level function `tc` for checking a program. Informally, **[Eq1]** states that checking a program involves (i) creating an initial type-environment that contains variable-type pairs, and (ii) distributing the type-environment over the program's statements, using an auxiliary function `dist`. For the simple example we study here, the type-environment consists of the declaration section of the program, to which the constructor function `tenv` is applied. Equation **[Eq2]** expresses the distribution of type-environments over lists of statements, and **[Eq3]** and **[Eq4]** the distribution over assignment operators and '+' operators, respectively. **[Eq5]** states how an identifier is reduced to its type, using an auxiliary function `type-of`, which is defined in **[Eq6]**. Note that the variables T_1^* and T_2^* in **[Eq6]** match any sublist of (zero or more) declarations in a declaration section. Equation **[Eq7]** expresses the abstract evaluation of additions, and **[Eq8]** states that the assignment of a natural expression to a natural variable is valid.

As an example, consider checking the following program block:

```
tc(begin x :  natural; y :  string;x := x + x; x := y + x end)
```

Application of **[Eq1]** results in:

```
dist(x := x + x; x := y + x, tenv(x :  natural; y :  string))
```

Application of **[Eq2]** yields:

```
dist(x := x + x, tenv(x :  natural; y :  string));
dist(x := y + x, tenv(x :  natural; y :  string))
```

At this point, [**Eq3**] can be applied to both components, producing:

```
dist(x, tenv(x :  natural; y :  string))
    := dist(x + x, tenv(x :  natural; y :  string));
dist(x, tenv(x :  natural; y :  string))
    := dist(y + x, tenv(x :  natural; y :  string))
```

The left-hand sides of both assignments can be reduced to their types using [**Eq5**] and [**Eq6**], resulting in:

```
natural := dist(x + x, tenv(x :  natural; y :  string));
natural := dist(y + x, tenv(x :  natural; y :  string))
```

Using [**Eq4**] and [**Eq5**], the right-hand sides of the assignments can be simplified:

```
natural := natural + natural;
natural := string + natural
```

Using equation [**Eq7**], the first assignment can be simplified:

```
natural := natural;
natural := string + natural
```

Finally, application of [**Eq8**] yields the final result:

```
“correct”;
natural := string + natural
```

The fact that this term contains a subterm that cannot be reduced to “`correct`” indicates that the program is not type-correct. Note that the non-“correct” subterm already gives a rough indication of the nature of the type violation.

Fig. 4 shows a set of equations that define a function `msgs` that transforms the cryptic messages produced by the specification of Fig. 3 into human-readable form. The equations of Fig. 4 assume that the term to which they are applied is fully normalized w.r.t. type checking equations of Fig. 3.

Equation [**Er1**] distributes function `msgs` over all statements in a block. [**Er2**] transforms the constant `correct`, which was derived from a type-correct program construct, into a message “`No errors`”. Since we are not interested in generating messages for correct statements, equation [**Er3**] eliminates “`No errors`” from lists of messages. Equations [**Er4**] and [**Er5**] perform the post-processing of expressions that are derived from incorrect assignment statements. Note that these equations are *conditional*: they are only applicable if a certain condition holds. (Here, the condition verifies if the right-hand side of the expression is a simple type, using auxiliary equations [**Er7**] and [**Er8**].) [**Er4**] postprocesses assignment statements whose right-hand side consists of an irreducible expression; whereas [**Er5**] postprocesses assignments whose left-hand side and right-hand side are incompatible. Equation [**Er6**] postprocesses ‘`+`’ expressions with incompatible arguments. The reader should observe that the specification of Fig. 4 only serves to illustrate the general technique and that it is incomplete; For example, it does not handle nested expressions.

As an example, we will postprocess the term “`correct`”; `natural := string + natural` by applying the equations of Fig. 4 to the term:

```
msgs(“correct”; natural := string + natural)
```

Applying [**Er1**] produces:

```
msgs(“correct”); msgs(natural := string + natural)
```

Using equation [**Er2**], we obtain:

```
“No errors”; msgs(natural := string + natural)
```

By applying [**Er3**], the “`No errors`” message is eliminated:

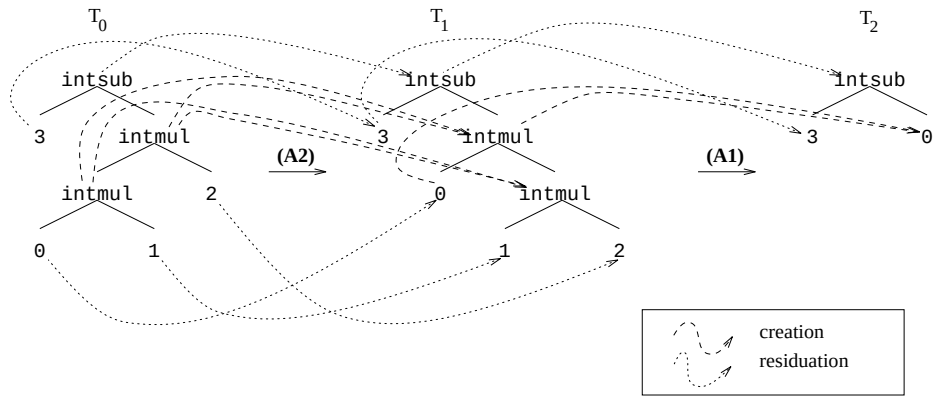

Figure 5: Example of creation and residuation relations.

```
msgs(natural := string + natural)
```

Since the right-hand side of the assignment is not of a simple type (we cannot derive the constant `true` from the term `simpletype(string + natural)`, conditional equation **[Er4]** can be applied, producing:

```
msgs(string + natural)
```

Application of **[Er6]** yields the human readable error message:

```
"Operands of + should have the same type."
```

The CLaX type checker specification that has been used to generate the snapshots of Fig. 1 and 2 follows the same basic principles that have been presented in this section. Language features such as gotos, nested scopes, and arrays introduce some additional complexity, but pose no fundamental problems. An annotated listing of the CLaX specification appears in [13].

4 Term Rewriting and Dependence Tracking

4.1 Term rewriting

In the previous section, specifications were "executed" by repeatedly applying equations to terms—a mechanism that is usually referred to as *term rewriting*. Both theoretical properties of term rewriting systems [26] such as termination behavior, and efficient implementations of rewriting systems [23, 24] have been studied extensively.

Term rewriting [26] can be viewed as a cyclic process where each cycle begins by determining a subterm t and a rule $l = r$ such that t and l *match*. This is the case if a substitution σ can be found that maps every variable X in l to a term $\sigma(X)$ such that $t \equiv \sigma(l)$ (σ distributes over function symbols). For rewrite rules without conditions, the cycle is completed by replacing t by the instantiated right-hand side $\sigma(r)$. A term for which no rule is applicable to any of its subterms is called a *normal form*; the process of rewriting a term to its normal form (if it exists) is referred to as *normalizing*. A conditional rewrite rule [3] (such as **[Er4]** and **[Er5]** in Figure 4) is only applicable if all its conditions succeed; this is determined by instantiating and normalizing the left-hand side and the right-hand side of each condition. Positive (equality) conditions (of the form $t_1 = t_2$) succeed iff the resulting normal forms are syntactically equal, negative (inequality) conditions ($t_1 \neq t_2$) succeed if they are syntactically different.

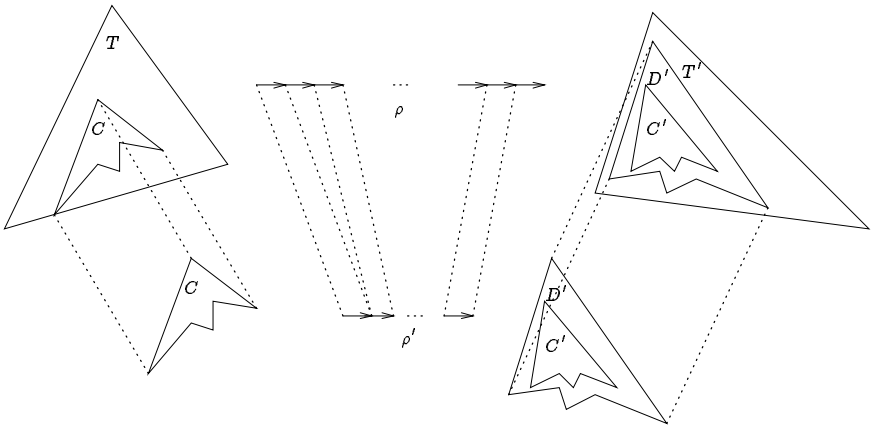

Figure 6: Depiction of the definition of a term slice.

4.2 Dependence tracking

Thus far, we have described the process of specifying a type checker, and the execution of such specifications by way of term rewriting. In order to obtain positional information, we use a technique called *dependence tracking* that was developed by Field and Tip [17, 18]. For a given sequence of rewriting steps $T_0 \rightarrow \cdots \rightarrow T_n$, 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 [30]) as an example to illustrate dependence tracking:

$$\begin{array}{lll} \textbf{[A1]} & \texttt{intmul}(\texttt{0}, X) & = \texttt{0} \\ \textbf{[A2]} & \texttt{intmul}(\texttt{intmul}(X,\ Y), Z) & = \texttt{intmul}(X, \texttt{intmul}(Y,\ Z)) \end{array}$$

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):

$$\begin{array}{rl} T_0 = & \texttt{intsub}(\texttt{3}, \underline{\texttt{intmul(intmul(0, 1), 2)}}) \\ & \qquad \longrightarrow \textbf{[A2]} \\ T_1 = & \texttt{intsub}(\texttt{3}, \underline{\texttt{intmul(0, intmul(1, 2))}}) \\ & \qquad \longrightarrow \textbf{[A1]} \\ T_2 = & \texttt{intsub}(\texttt{3}, \texttt{0}) \end{array}$$

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 [17, 18]. 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[4]. Figure 5 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 [17, 18], 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 $\rho : T \rightarrow \cdots \rightarrow T'$, 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 6.

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 bottom window of the CLaX environment of Figure 1 is a textual representation of a term that represents a list of errors. The slices shown in Figure 2(a)–(d) are computed by tracing back the dependence relations from each of the four "error" subterms.

4.3 The effect of determinism on slice accuracy

We have argued that our approach for obtaining positional information does not rely on a specific specification style. Nevertheless, experimentation with the CLaX type checker has revealed that the *accuracy* of the computed slices inversely depends on the degree to which the specification is *deterministic*. As a general principle, *more* determinism in a specification leads to *less* accurate slices. To understand why this is the case, consider the nature of dynamic dependence relations. Suppose that type checking a program P involves a sequence of rewrite steps r that ultimately lead to an error e. The slice P_e associated with e has the property that it can be rewritten to a term containing e, using a subset r' of the rewrite-steps in r. If the rewrite steps in r encode a deterministic process such as the explicit traversal of a list of statements, this deterministic behavior will also be exhibited by r', to the extent that it contributed to the creation of e.

As an example, consider rewriting the term:

```
type-of(tenv(x :  integer; y:  string; z :  integer), y)
```

according to the specification of Figure 3. By applying equation **[Eq6]**, this term rewrites to the constant `string`. By tracing back the dynamic dependence relations, we find that the context

```
type-of(tenv(●; y:  string; ●), y)
```

was needed to create this result. Now suppose that instead of equation **[Eq6]**, we use the following two equations for reducing the same term:

[Eq6a] `type-of(`Id`, tenv(`Id`:`$Type$`;` D^*`)) =` $Type$

[Eq6b] `type-of(`Id`, tenv(`Id'`:`$Type$`;` D^*`)) = type-of(`Id`, tenv(` D^*`))`
`when` Id' `!=` Id

The resulting term would be the same as before: the constant `string`, which is obtained by first applying equation **[Eq6b]** followed by applying equation **[Eq6a]**. However, the subcontext needed for creating this result would now consist of:

```
type-of(tenv(x :  ●; y:  string; ●), y)
```

The variable `x` in the first element of the type environment is now included in the slice because the *order* in which the type environment is traversed is made explicit in the specification. Informally stated, the resulting term `string` is now dependent on the fact that the first element of the type environment is not an entry for variable `y`.

[4] 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 [17, 18].

The use of list functions and list matching in specifications (i.e., allowing function symbols with a variable number of arguments and variables that match sublists) has the effect of reducing determinism, and therefore improving slice accuracy. We believe that more powerful mechanisms for expressing nondeterminism such as higher-order functions [21] can in principle improve slice accuracy even further.

Experimentation with the CLaX type checker specification of [12] revealed a small number of cases where slices were unnecessarily inaccurate due to overly deterministic behavior. Virtually all of these cases consisted of explicit traversals of lists, with the purpose of finding a specific list element, or verifying whether or not a list contained a certain element more than once. In each of these cases, the use of list functions allowed us to specify the same function nondeterministically with little effort.

5 A case study: type-checking the CLaX language

We now turn our attention to a case study, in which we apply our techniques to a Pascal-like imperative programming language named CLaX. The most interesting features of CLaX are: nested scopes, overloaded operators, arrays, goto statements, and procedures with reference and value parameters.

The CLaX language was originally developed as the demonstration language of the ESPRIT-II Compare (Compiler Generation for Parallel Machines) project [1], and the original (informal) description of the semantics of CLaX can be found in [29]. Since then, CLaX has been used as a basis for various software tools, including type checkers, interpreters, and debuggers [12, 10, 11, 34, 30], as well as a test-bed for origin-tracking techniques [35, 32, 17]. In the present paper, we will only present some of the highlights of the CLaX specification. For more details on the CLaX language, the reader is referred to [12].

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 the Algebraic Specification Formalism ASF [2] and the Syntax Definition Formalism, SDF [22]. ASF features first-order signatures, conditional equations, modules, and facilities for import, export, and hiding. SDF allows for the simultaneous definition of a language's lexical syntax, context-free syntax, and abstract syntax. The combined formalism, ASF+SDF [34], is unusually flexible in the sense that it allows one to specify the syntax of a language, and then define equations in terms of that user-defined syntax. The ASF+SDF Meta-environment [25] is an implementation of ASF+SDF. By interpreting equations as rewrite-rules, specifications can be executed as term rewriting systems.

5.1 Specification of the CLaX syntax in ASF+SDF

In order to give the reader an impression of what an ASF+SDF specification looks like, we will briefly address some of the highlights of the ASF+SDF-specification of CLaX, starting with the definition of the CLaX syntax. For a full overview of ASF+SDF, the reader is referred to [34, 25].

Fig. 7 shows two of the modules that together define the CLaX syntax. Module `SyntaxProgram` is the top-level module that defines the syntax of CLaX programs. Since module `SyntaxProgram` relies on several sorts (i.e., specification-level types) that are not defined locally, it needs to import the modules in which these sorts are defined. The `imports` section of `SyntaxProgram` consists of:

```
imports SyntaxHeaders SyntaxStats
```

stating that two auxiliary modules, `SyntaxHeaders` and `SyntaxStats`, are imported. Module `SyntaxProgram` defines a sort `PROGRAM`, and contains grammar rules for constructing programs. For instance, the rule

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

states that a `BLOCK` may consist of a keyword '`DECLARE`' followed by a declaration list (sort `DECL-LIST`), a keyword '`BEGIN`', a sequence of statements (sort `STAT-SEQ`), and a keyword '`END`'. Note that there is another grammar rule for the case where a `BLOCK` does not contain any declarations. Grammar rule

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

subsequently defines a `PROGRAM` to consist of the keyword '`PROGRAM`' followed by an identifier (sort `ID`), a `BLOCK`, and a period. Finally, the variables section of module `SyntaxProgram`

```
[_]Program[0-9']*   -> PROGRAM
```

defines variables of sort PROGRAM that can be used in the equations of any modules that imports SyntaxProgram. This rule defines the lexical syntax of a variable of sort Program to consist of an underscore character, followed by character sequence 'Program', followed by zero or more occurrences of a digit or a quote character.

Module SyntaxHeaders, which defines the syntax of declarations and procedure headers, is also shown in Fig. 7. Various types of declarations are defined. Label declarations (sort LABEL-DECL) consist of an identifier, followed by a colon, and the keyword 'LABEL'. Variable declarations consist of an identifier, a colon, and a TYPE (defined in module SyntaxTypes not shown here). Procedure declarations consist of a procedure header (sort PROC-HEAD), followed by a BLOCK. Finally, empty declarations (sort EMPTY-DECL) have no concrete syntax at all. Sort DECL is introduced to represent all of these kinds of declarations, so that they can be uniformly represented in declaration lists (sort DECL-LIST). Sort DECL-LIST illustrates the use of *lists* in ASF+SDF:

```
{ DECL ";" }*                          -> DECL-LIST
```

defines declaration list to be a sequence of zero or more declarations *separated* by semicolons. Formal parameters (sort FORMAL) are defined to consist of variable declarations, optionally preceded by the keyword 'VAR' (for reference parameters). Procedure headers are defined as follows:

```
"PROCEDURE" ID                         -> PROC-HEAD
"PROCEDURE" ID "(" {FORMAL ";"}+ ")"   -> PROC-HEAD
```

indicating that a procedure header consists of the keyword 'PROCEDURE', followed by an identifier, and optionally followed by an open bracket, a list of one or more formal parameters separated by semicolons, and a close bracket.

Fig. 8 shows an example of a CLaX program.

5.2 High-level overview of the CLaX type checker specification

Before delving into some of the more interesting details of the CLaX type checker specification, we will briefly overview the global design of the specification. As can be seen from the import diagram of the type checker modules (see Fig. 9), the type checker specification imports the CLaX syntax of module SyntaxProgram that was discussed previously. The CLaX type checker performs (roughly) the following steps in order to type check a BLOCK of statements:

- The declarations of a block are processed, yielding a local type environment. A type-environment essentially represents the context in which a particular statement, block, or expression is type checked.
- Some checks are performed on the local type environment. For example, we check if each identifier is unique within its scope, and if the index ranges of arrays contain at least one element.
- The local type environment is combined with the type environments corresponding to the BLOCK's surrounding scopes, and this combined type environment is *distributed* over every program construct.
- 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. This allows us to localize the checking of the validity of all conditional expressions in one place.
- Identifiers and values are rewritten to a common abstract interpretation. We use *types* for abstract representations. For example, any constant '17' is rewritten to 'INTEGER', and any identifier declared as a real is rewritten to 'REAL'.
- Expressions are interpreted abstractly using the abstract values obtained in the previous step. Any *type-correct* expression is rewritten to its abstract value. For example, an expression 'INTEGER + INTEGER' is rewritten to 'INTEGER'.

```
%% 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

%% Module SyntaxHeaders

imports SyntaxTypes

exports
  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
```

Figure 7: Some modules of the ASF+SDF specification of the CLaX syntax.

```
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 8: Example of a CLaX program.

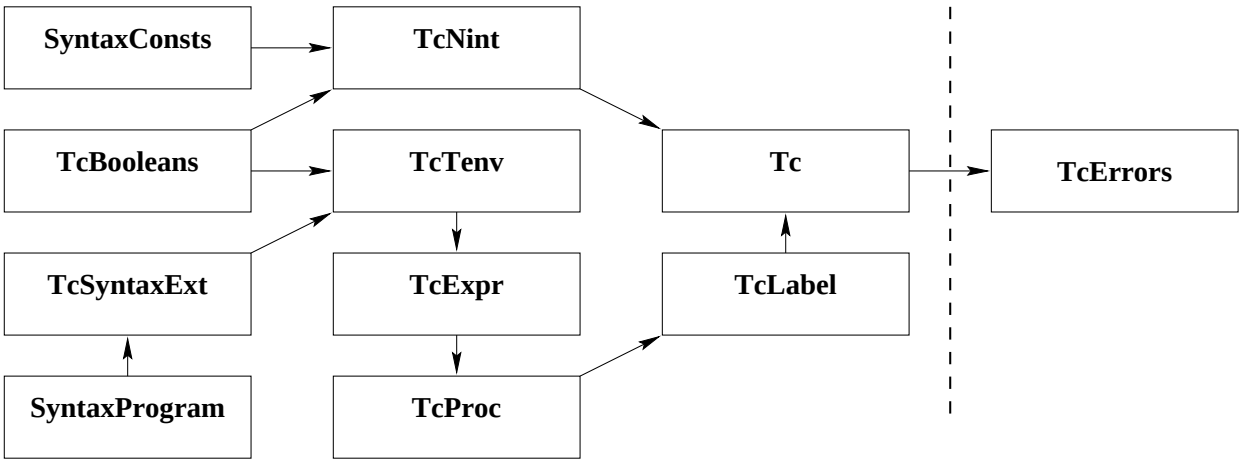

Figure 9: Import diagram for the type checking modules. The dashed line indicates the separation between the type checking phase, and the postprocessing phase in which human-readable error messages are produced.

```
%% 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
```

Figure 10: Module `TcTenv` of the ASF+SDF specification of the CLaX type checker.

- Type correct statements (e.g., assignments whose left-hand side and right-hand side are both rewritten to '`INTEGER`') are are reduced to the constant '`true`'.
- Human-readable error messages are generated from the list of remaining abstract expression in a way that is similar to that of Fig. 4. Any statement that was reduced to '`true`' in the previous step is simply removed at this point, since it did not contribute to the list of type errors.
- Dependence tracking (see Section 4) is used to trace these human-readable error messages back to the source.

In the next few sections, we will explore some of the more interesting aspects of these steps in more detail.

5.3 Type-environments

The ASF+SDF syntax definitions we have seen so far were used to describe the syntax of the CLaX language. It is important to understand that *exactly the same* kind of syntax definitions are used to express the auxiliary data structures used by the type checker. To illustrate this point, Fig. 10 shows module `TcTenv` of the CLaX type checker specification, which specifies the syntax of type-environments. The rule

```
"[" {DECL ";"}* "]"          -> TENV
```

defines a type-environment (sort `TENV`) to consist of a list of zero or more semicolon-separated declarations between square brackets. Combined type environments (sort `TENV-LIST`), which capture the declarations of multiple nested scopes, are simply defined as a list of `TENV`s.

Module `TcTenv` also defines an auxiliary function `type-of` that computes the type of an expression in the context of a given combined type environment. The inclusion of this operation in `TYPE` indicates our intention that it reduces expressions to an abstract value.

In order to be able to rewrite expressions to their abstract value (i.e., their type), sort `TYPE` is injected into sort `EXPR` by the following grammar rule:

```
TYPE                          -> EXPR
```

This enables us to write equations that rewrite constants that occur in expressions to their abstract value since the evaluation of constants does not rely on the type environment. Equations [1]—[3] of Fig. 10 over sort EXPR) rewrite all constants *found in expressions* to their abstract values. The variables *_IntConst*, *_RealConst*, and *_BoolConst* in these equations will only match terms that represent integer constants, real constants, and boolean constants, respectively.

5.4 Processing expressions and statements

To give the reader an impression of the equations that evaluate expressions in the abstract domain, two representative equations of module TcExpr are shown below:

```
[t14] INTEGER _Op INTEGER = INTEGER   when _Op = _Aop
[t17] _SimpleType _Op _SimpleType = BOOLEAN when _Op = _Cop
```

In these equations, variable _Op is of sort OP (operator), variable _Aop is of sort AOP (arithmetic operator), variable _Cop is of sort COP (comparison operator), and variable _SimpleType is of sort SIMPLE-TYPE. Equation [t14] states that an expression consisting of an arithmetic operator applied to two subexpressions of sort INTEGER evaluates to sort INTEGER. Equation [t14] states that an expression consisting of an comparison operator applied to two subexpressions of the same simple type evaluates to type BOOLEAN. Note that we use a variable of sort SIMPLE-TYPE here instead of a variable of sort TYPE because comparison operations on non-simple types such as arrays are not allowed in CLaX.

Below we show two of the equations (taken from module Tc) that abstractly evaluate statements.

```
[R1]  _SimpleType := _SimpleType  = true
[R4]  WRITE ( _SimpleType ) = true
```

Equation [R1] rewrites an assignment to true if its left-hand side and right-hand side are of the same simple (i.e., non-array) type. Equation [R4] rewrites type-correct WRITE statements to true.

5.5 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 to the output of the type checking function. 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
```

5.6 An extended example

As an example, we will study the type checking of the following CLaX program in some detail:

```
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.
```

After changing constants to their abstract values, 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 TYPEs in CLaX (there are no operations defined over strings), they do not have an abstract value, and hence are not affected in this step.

Next, the type environment for checking the statements is constructed. This is done by a recursive function `collect` in module `Tc`, which collects the declarations in a set of nested scopes into a combined type environment (sort `TENV-LIST`), as was discussed in Section 5.3. Function `collect` has two arguments: a `TENV-LIST` of type environments constructed so far, and a `BLOCK` that needs to be processed. 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
)
```

Next, some checks are performed on the local type environment and the consistency of `GOTO` statements is checked before checking the individual statements in a `BLOCK`. For instance, before distributing the type environment over the statements in procedure `square`, a label error

```
unique(step step)
```

is produced (for the fact that label `step` is defined twice). This subterm will later be transformed into a human-readable error message indicating that more than one statement has label `step` associated with it.

```
unique(step step)
  &
distribute([ i :  INTEGER;
  square :  PROC (INTEGER);
  n :  INTEGER;
  x :  REAL;
  step :  LABEL],
    BEGIN x := INTEGER;
      step := n;
      step := step * REAL; ···
    END) ···
```

After distribution of the type environment, evaluation of the expressions over the abstract domain of types, and rewriting type-correct statements to `true` the situation looks as follows:

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

Note that the assignment `REAL := INTEGER` was rewritten to true because CLaX allows assignments of integer-typed expressions to real-typed variables.

Finally, human-readable error messages are generated by distributing function `errors` of module `TcErrors` over the previous term. The resulting normal form is:

```
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 4 discusses how such information can be obtained automatically using dynamic dependence tracking.

5.7 Lessons learned

We will now summarize a number of changes we made to the specification in order to improve the accuracy of the computed slices. In addition to the changes discussed below, we effectively "undid" the changes that were made to the specification in order get reasonable error locations using origin tracking, as was discussed earlier in Section 2. As it turns out, almost all of the issues discussed below have the flavor of eliminating "redundant determinism" or "over-specification".

Over-specification: unnecessarily specific matching

In a number of places, the type checker specification of [12] was matching unnecessarily specific subterms, which gave rise to spurious symbols in the slice. 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.

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.

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.

Elimination of correct program constructs

In the original version of module `TcBooleans`, the following equation was used for the simplification of conjunctions:

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

This equation served to eliminate the `true` constants that originated from type-correct program constructs. Although this equation had the desired effect of removing the redundant `true` constants, it over-specified our intention in a subtle way. Instead of expressing the fact that a program is correct if it contains no incorrect statements, it specifies that the correctness of a list of statements depends on the correctness of all the elements contained in the list. The locations produced by dependence tracking reflected this: Since the boolean simplification took place *before* the distribution of the `errors` function of module `TcErrors`, the locations of an error message e contained adjacent type correct constructs. The solution to this problem was to do the elimination of type-correct constructs *after* distribution of the `errors` function. In the current situation, `true` subterms remain until distribution of the `errors` function. Then, `errors(true)` is reduced to "no-errors" by the following equation:

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

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

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

Elimination of determinism: duplicate elements in lists

Overspecification is undesirable because it may result in overly large slices. Unfortunately, over-specification can occur in subtle ways and very hard to control. To illustrate this point, the original version of the function `unique` (module `TcLabel`) is shown below (in this specification, `&` denotes boolean conjunction). Function `unique` takes a `LABEL-LIST`, and returns `true` if the list contains no duplicate elements. Originally, `unique` was defined in the following manner, using an auxiliary function `no-dups` for determining if a list contains duplicate elements.

```
[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+)
```

Hence, the specification states that a list is unique if it is true that there are no duplicates[5]. Consider the result of this approach: When a list is not unique, the locations of the duplicate elements in the resulting term become dependent on those of the other elements in the list. This will lead to undesirably large error locations.

Instead, we use the following definition of `unique`.

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

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

In this variation of `unique`, a list is defined to be unique only if it is not the case that it has duplicate elements. Thus, when a list is not unique, the function `no-dups` does not match. Consequently, the locations obtained with dependence tracking for duplicate elements will not be "polluted" with other elements.

[5] Note that equation `no-dups(_Id _Id) = false` is deliberately not defined because we were already trying to avoid some over-specification in the original version. We are only interested in the case where `unique` is *not* `true` because we want to be able to post-process the resulting irreducible term into a human-readable error message.

6 Conclusions

We have presented a slicing-based approach for determining locations of type errors. Our work assumes a framework in which type checkers are specified algebraically, and executed by way of term rewriting [26]. In this model, a type check function rewrites a program's abstract syntax tree to a list of type errors. Dynamic dependence tracking [17, 18] is used to associate a *slice* [37, 31] of the program with each error message. Unlike previous approaches for automatic determination of error locations [12, 10, 11, 33, 32, 7, 6, 8], ours does not rely on a specific specification style, nor does it require additional specification-level information for tracking locations. The computed slices have an interesting semantic property: The slice P_e associated with error message e is a projection of the original program P that, when type checked, is guaranteed to produce the same type error e.

We have implemented this work in the context of the ASF+SDF Meta-environment [25, 34] for a substantial subset of Pascal. Experimentation with CLaX revealed that the computed slices provide highly insightful information regarding the nature of type violations. We have observed that the amount of determinism in a specification is an important factor that determines the accuracy of the computed slices, and we consider this to be a topic that requires further study. As another direction for future work, one would study the applicability of slicing-based error location in the related area of type *inference* [9], in particular for object-oriented languages [28] and for ML [27]. Providing accurate positional information for type inference errors in ML is a difficult problem. Several proposals that rely on adapting or extending the underlying type system or inference algorithm have been presented (see, e.g., [5, 36]). In contrast, applying dependence tracking to a rewriting-based implementation of an ML type inferencer might require no changes to the type inference algorithm. 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., Heering, J., and Klint, P., Eds. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[3] Bergstra, J., and Klop, J. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences 32*, 3 (1986), 323–362.

[4] Bergstra, J. A., Dinesh, T. B., Field, J., and Heering, J. Toward a complete transformational toolkit for compilers. *ACM Transactions on Programming Languages and Systems 19*, 5 (September 1997), 639–684.

[5] Bernstein, K. L., and Stark, E. W. Debugging type errors (full version). Tech. rep., State University of New York at Stony Brook, Computer Science Department, 1995.

[6] Bertot, Y. Occurrences in debugger specifications. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 327–337. *SIGPLAN Notices* 26(6).

[7] Bertot, Y. *Une Automatisation du Calcul des Résidus en Sémantique Naturelle.* PhD thesis, INRIA, Sophia-Antipolis, 1991. In French.

[8] Bertot, Y. Origin functions in lambda-calculus and term rewriting systems. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming (CAAP '92)* (1992), J.-C. Raoult, Ed., vol. 581 of *LNCS*, Springer-Verlag.

[9] 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.

[10] 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.

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

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

[13] DINESH, T. B., AND TIP, F. A case-study of a slicing-based approach for locating type errors. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)* (Amsterdam, The Netherlands, September 1997), eWIC, electronic Workshops in Computing, p. 36 pages.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] FIELD, J., AND TIP, F. Dynamic dependence in term rewriting systems and its application to program slicing. Tech. Rep. RC 21117, IBM T.J. Watson Research Center, February 1998. To appear in *Information and Software Technology.*

[19] FLANAGAN, C., FLATT, M., KRISHNAMUTHI, S., WEIRICH, S., AND FELLEISEN, M. Catching bugs in the web of program invariants. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, 1996), pp. 23–32.

[20] FRAER, R. Tracing the origins of verification conditions. In *Proceedings of AMAST'96* (Munich, Germany, July 1996), vol. 1101, Springer-Verlag LNCS.

[21] HEERING, J. Second-order term rewriting specification of static semantics. In *Language Prototyping: An Algebraic Specification Approach*, A. van Deursen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co., 1996, pp. 295–306.

[22] HEERING, J., HENDRIKS, P. R. H., KLINT, P., AND REKERS, J. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices 24*, 11 (1989), 43–75.

[23] KAMPERMAN, J. *Compilation of Term Rewriting Systems.* PhD thesis, University of Amsterdam, 1996.

[24] KAMPERMAN, J., AND WALTERS, H. Minimal term rewriting systems. In *Recent trends in data type specification : 11th workshop on specification of abstract data types joint with the 8th COMPASS workshop: Oslo, Norway, 19-23.09.1995 : selected papers* (1996), vol. 1130 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 274–290.

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

[26] 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.

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

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

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

[30] 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.

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

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

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

[34] 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.

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

[36] WAND, M. Finding the source of type errors. In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, 1986), pp. 38–43.

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

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