The following paper was originally published in the
Proceedings of the Conference on Domain-Specific Languages
Santa Barbara, California, October 1997

A Slicing-Based Approach for Locating Type Errors

T. B. Dinesh
CWI
Frank Tip
IBM T.J. Watson Research Center

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

A Slicing-Based Approach for Locating Type Errors

T. B. Dinesh

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

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.

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 real "source" of the error. 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) [25]. 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 [18, 28] 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 [12, 31], 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 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 [18, 28].

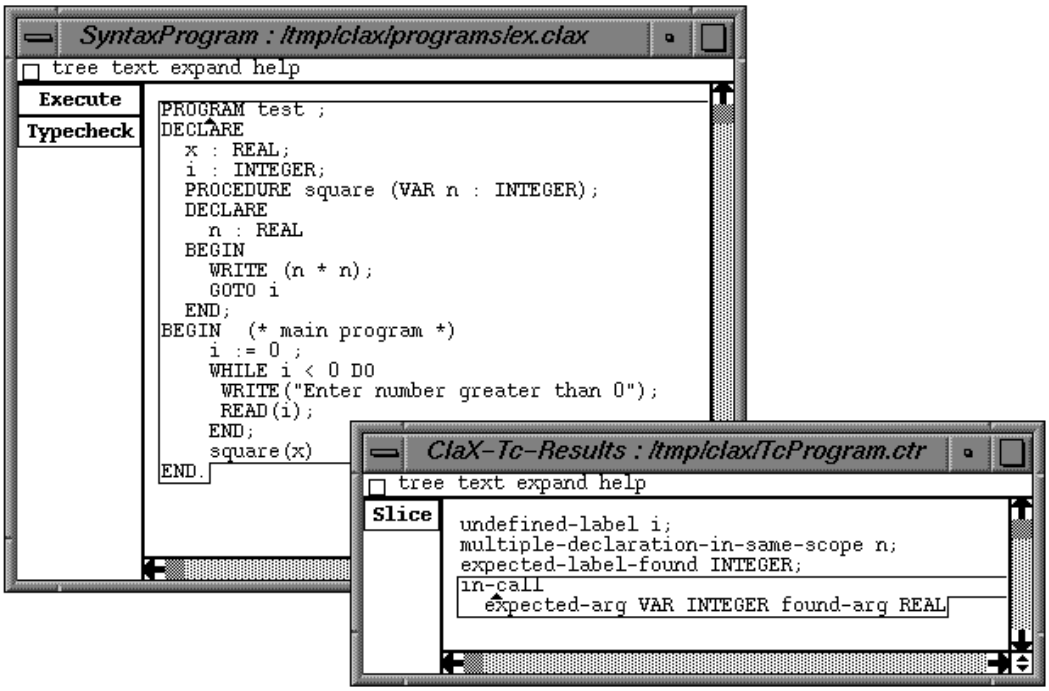

Figure 1: The CLaX environment. The top window is a program editor with two buttons attached to it for invoking a type checker and an interpreter, respectively. The bottom window shows a list of four type errors reported by the type checker. After selecting an error message in the bottom window, the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

Although positional information is always available for any error message, the *accuracy*[1] of these locations depends on the degree to which the specified type checker's behavior is deterministic. This issue will be explored in Section 4.

We have implemented a prototype type checking system using the ASF+SDF Meta-environment [24, 31], 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 [29]. Figure 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 Figure 1 is a program editor, which has two buttons labeled '**Type-Check**' and '**Execute**' attached to it, for invoking the type checker and the interpreter, respectively.

[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.

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.

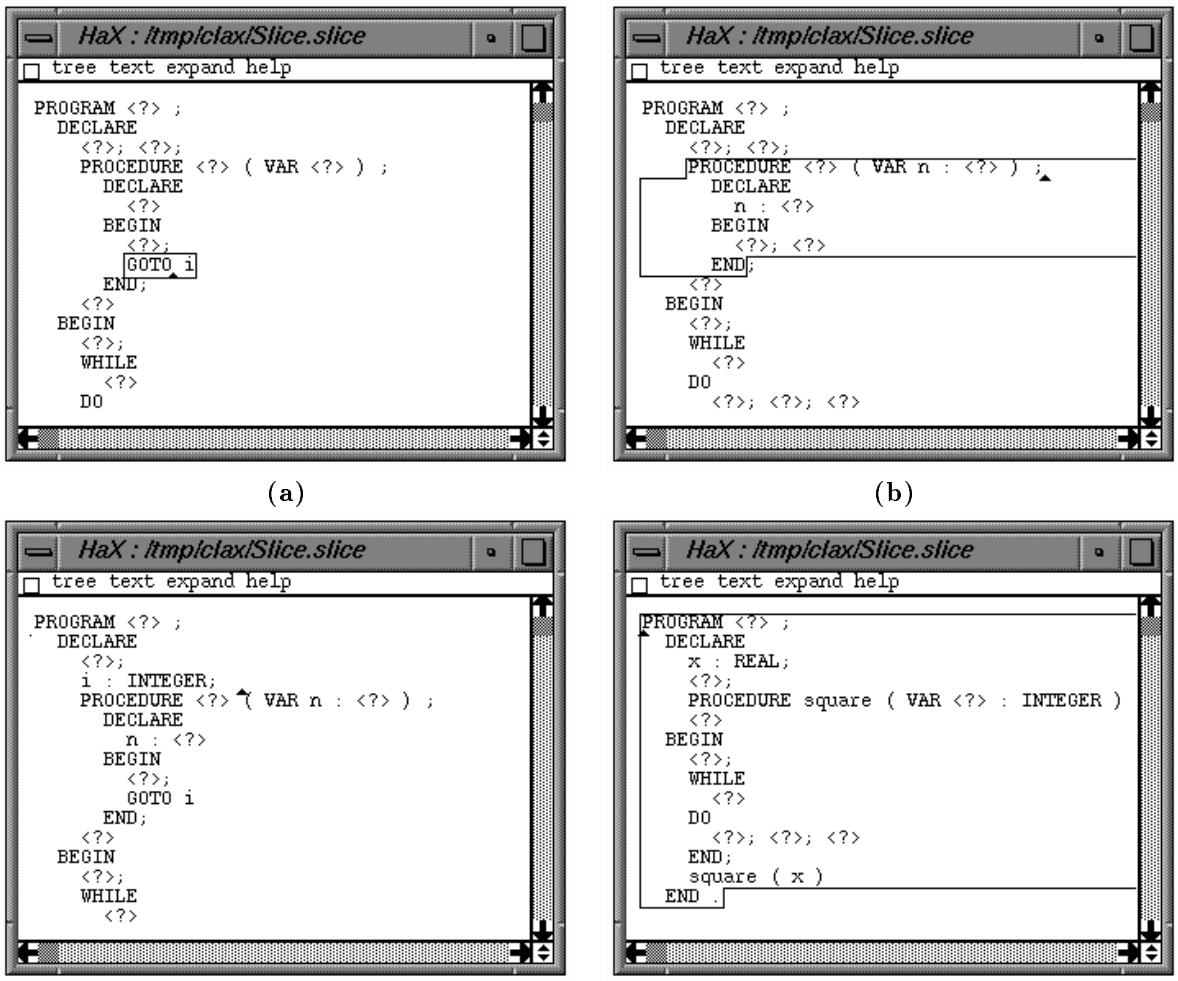

Figure 2: Slices reported by the CLaX environment for each of the type errors of Figure 1.

However, positional information may be obtained by selecting an error message and clicking on the '**Slice**' button. In Figure 2(a)–(d), the slices obtained for each of the four error messages of Figure 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. Figure 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. Figure 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. Figure 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 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. Figure 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., Figure 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. Section 2 presents our approach for specifying type checkers. In Section 3, the use of term rewriting for executing specifications is discussed. In addition, dependence tracking, the mechanism for computing slices is presented. Section 4 is concerned with the effect of determinism in the specification on slice accuracy. In Section 5, related work is discussed. In particular, the slice notion introduced in the present paper is compared with the traditional notion of a program slice. Conclusions and possible directions for future work are stated in Section 6.

2 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.

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

[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.

In [14], 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.

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

Figure 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
```

[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.

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.

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

```
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 Figures 1 and 2 follows the same basic principles that have been presented in this section. Language features such as

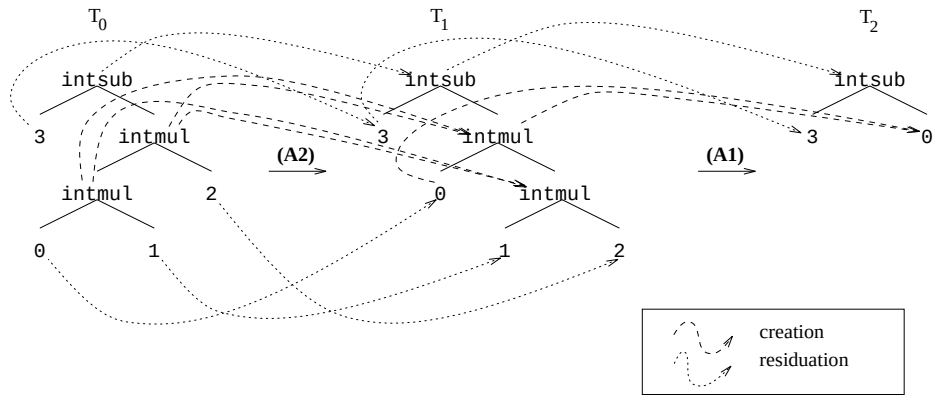

Figure 5: Example of creation and residuation relations.

gotos, nested scopes, and arrays introduce some additional complexity, but we experienced no fundamental problems. An annotated listing of the CLaX specification will appear in a technical report in the near future [15]. A previous version of the CLaX specification may be found in [14].

3 Term Rewriting and Dependence Tracking

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 [25] such as termination behavior, and efficient implementations of rewriting systems [22, 23] have been studied extensively.

Term rewriting [25] 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 3) 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.

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 [18, 28]. 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 [29]) as an example to illustrate dependence tracking:

[A1] `intmul`$(0, X)$ = 0
[A2] `intmul`(`intmul`$(X,\ Y), Z)$ = `intmul`$(X,$ `intmul`$(Y,\ Z))$

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):

$T_0 =$ `intsub(3,` $\underline{\texttt{intmul(intmul(0, 1), 2)}}$
$\longrightarrow$ **[A2]**
$T_1 =$ `intsub(3,` $\underline{\texttt{intmul(0, intmul(1, 2))}}$
$\longrightarrow$ **[A1]**
$T_2 =$ `intsub(3, 0)`

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.

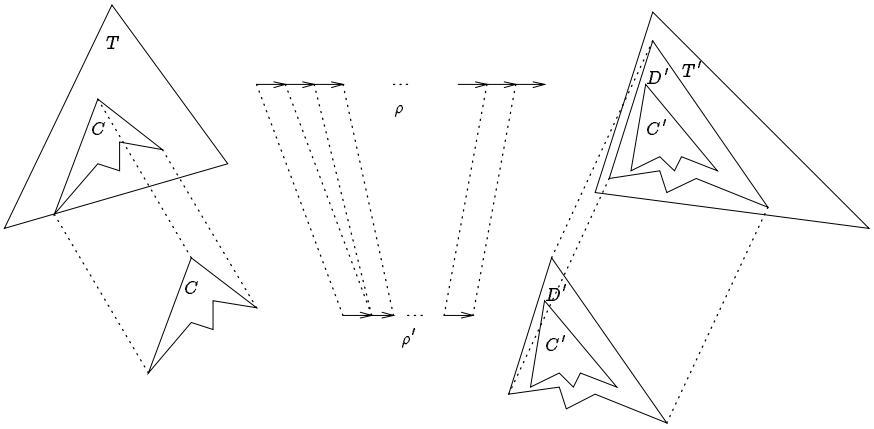

Figure 6: Depiction of the definition of a term slice.

- 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 [18, 28]. 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 [18, 28], 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 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 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

[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 [18, 28].

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`.
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 [14] 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. In a forthcoming technical report [15], we will present a brief overview of a few of the more interesting changes we made to the CLaX specification in order to make it less deterministic.

5 Related Work

The work presented in this paper is closely related to earlier work by the same authors. The CLaX type checker [14] 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* [11] 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 [12, 13]. 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 [12, 13], 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 [18, 28] for the purpose of computing program slices. A *program slice* [33, 34, 30] 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 [30].

By applying dependence tracking to different rewriting systems, various types of slices can be obtained. In [17] programs are translated to an intermediate graph representation named PIM [16, 1]. 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 [29], 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 [10, 9]. 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 [11] 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 [18, 28] to higher-order rewriting systems.

Fraer [20] uses a variation on origin tracking [6, 5, 7] 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.

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 [25]. In this model, a type check function rewrites a program's abstract syntax tree to a list of type errors. Dynamic dependence tracking [18, 28] is used to associate a *slice* [33, 30] of the program with each error message. Unlike previous approaches for automatic determination of error locations [14, 12, 13, 10, 9, 6, 5, 7], 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 [24, 31] for a substan-

tial 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, we intend to study the applicability of slicing-based error location in the related area of type *inference* [8], in particular for object-oriented languages [27] and for ML [26]. 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., [4, 32]). In contrast, we are interested in an approach that requires no changes to type inference algorithm or the type system. The basic idea is to apply dependence tracking to a rewriting-based implementation of an ML type inferencer. 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] 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.

[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] 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.

[5] 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).

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

[7] 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.

[8] 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.

[9] Deursen, A. v. *Executable Language Definitions—Case Studies and Origin Tracking Techniques.* PhD thesis, University of Amsterdam, 1994.

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

[11] Deursen, A. v., Klint, P., and Tip, F. Origin tracking. *Journal of Symbolic Computation 15* (1993), 523–545.

[12] 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.

[13] 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.

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

[15] Dinesh, T. B., and Tip, F. A slicing-based approach for locating type errors. Tech. rep., CWI/IBM, 1997. Forthcoming.

[16] 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.

[17] 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.

[18] 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.

[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. v. Deursen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co., 1996, pp. 295–306.

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

[23] 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.

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

[25] 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.

[26] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* The MIT Press, Cambridge, MA, 1990.

[27] Palsberg, J., and Schwartzbach, M. *Object-Oriented Type Systems.* John Wiley & Sons, 1993.

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

[29] 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.

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

[31] 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.

[32] 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.

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

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