A Slicing-Based Approach for Locating Type Errors

F. TIP
IBM T.J. Watson Research Center
and
T. B. DINESH
Pagelets, Inc.

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. We also report on experiments with object-oriented type systems, and with a subset of ML.

Categories and Subject Descriptors: 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*

Additional Key Words and Phrases: Semantics-based tool generation, program slicing, type-checking, static semantics, abstract interpretation

This research was supported in part by the Netherlands Organization for Scientific Research (NWO) under the *Generic Tools for Program Analysis and Optimization* project and was partly performed when the second author was affiliated with CWI.
This is a revised and expanded version of a paper that was presented at the USENIX Conference on Domain Specific Languages (DSL'97) [Dinesh and Tip 1997b]. 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) [Dinesh and Tip 1997a].
Authors' addresses: F. Tip, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; email: tip@watson.ibm.com; T. B. Dinesh, Pagelets, Inc., 2321 Hanover St., Palo Alto, CA 94306; email: dinesh@acm.org.

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 and compilers, 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 error messages reported by the tool.

—The quality of the positional information associated with these messages.

In our opinion, high-quality positional information is especially important because it assists the programmer with the tedious task of having to determine the program locations that need to be changed in order to fix type errors.

Many type errors are actually inconsistencies between the types of two expressions, and it is often the case that several noncontiguous program locations are involved in a type error. For example, consider an assignment statement `x := y` where `x` and `y` are of two incompatible types. What is the source of the error? In this case, three program fragments are potentially involved: the assignment itself, and the two locations where the *x* and *y* are declared. Ideally, all three of these locations are reported along with the error message, since any of these fragments may need to be changed to fix the problem.

Few type-checking tools provide positional information for type errors that is accurate and complete. The type-checkers embedded in present-day compilers typically provide only a limited amount of positional information. For example, for the above `x := y` error, the type-checkers embedded in the *gcc*[1] and in the *javac* [Gosling et al. 1996] compilers only report the number of the line containing the assignment. For languages with more complex type systems such as ML, the generation of precise positional information for type errors is known to be a difficult problem, and several approaches have been presented that rely on adapting or extending the underlying type system, or inference algorithm (e.g., see Bernstein and Stark [1995], Wand [1986], Johnson and Walz [1986], and Duggan and Bent [1996]).

In specification-based approaches to type-checking, a type-checker is derived from a high-level specification of the language's typing rules, which usually takes the form of sets of equations (e.g., see Klint [1993]) or inference rules (e.g., see Kahn [1987]). Keeping track of positional information at the specification level has the significant drawback that it clutters the specification and makes it less readable. Several approaches for automatically keeping track of positional information have been proposed previously, but all of these have the drawback providing incomplete information

[1]See `www.gnu.org/software/gcc/gcc.html`.

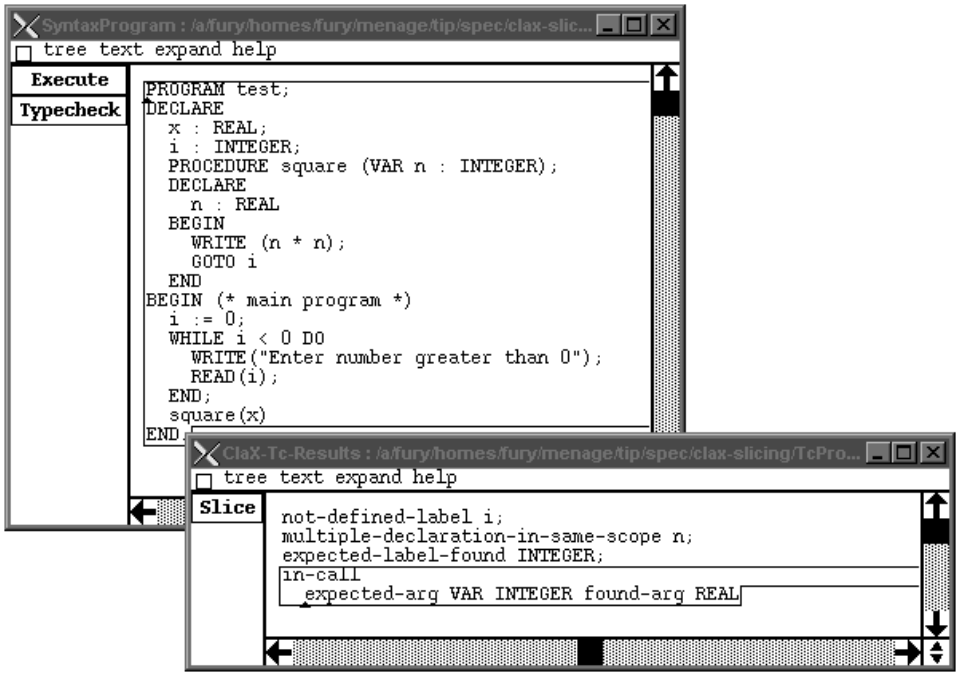

Fig. 1. The CLaX environment. The top window is a program editor with two buttons attached to it for invoking a type-checker and an interpreter, respectively. The bottom window shows a list of four type errors reported by the type-checker. After selecting an error message in the bottom window, the **Slice**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

(e.g., see van Deursen et al. [1993]), and/or considering only restricted classes of specifications (e.g., see van Deursen [1994a; 1994b]).

1.1 Outline of Approach

This paper presents a semantically well-founded approach where high-quality positional information is *automatically* computed for each type error. In our approach, the behavior of a type-checker is algebraically specified by way of a set of conditional equations [Bergstra et al. 1989], which are interpreted as a conditional term rewriting system (CTRS) [Klop 1992]. These rewriting rules express type-checking as the transformation of a program's abstract syntax tree (AST) into a list of error messages.

We use dependence tracking [Field and Tip 1994; 1998] to compute a *slice* [Tip 1995b; Weiser 1979] of the original program as the positional information associated with an error message. Dependence tracking is a fully automatic technique for establishing dependence relations between terms that occur in a term rewriting process. For a given sequence s of rewrite steps $T_0 \rightarrow \cdots \rightarrow T_n$, and a given *context* C_n (i.e., a contiguous set of function symbols) in term T_n, dependence tracking will compute a context C_0 in term T_0 such that C_0 can be rewritten to C_n using a subset of the rewrite steps in the original sequence s. This context C_0 in term T_0 is a

"slice" of T_0 that omits any function symbol that is not necessary for creating context C_n. 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 [Dinesh 1994; van Deursen et al. 1996], 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.

Although positional information is always available for any error message, the *accuracy*[2] of these locations depends inversely on the degree to which the specified type-checker explicitly traverses syntactic structures such as lists. This issue will be explored in Section 3.3.

1.2 A Prototype Implementation

We implemented a prototype type-checking system using the ASF+SDF Meta-environment [Klint 1993; van Deursen et al. 1996], 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 [Tip 1995a], and for computing constrained program slices [Field et al. 1995]. The main difference between the present work and these previous applications of dependence tracking is the fact that, to a type-checker, a program is a piece of data, and that dependence tracking is *not* used to compute a dynamic slice of the program that executes during type-checking (i.e., the type-checker itself), but to produce a slice of that program's data (i.e., the program being type-checked).

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 that 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.

[2]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.

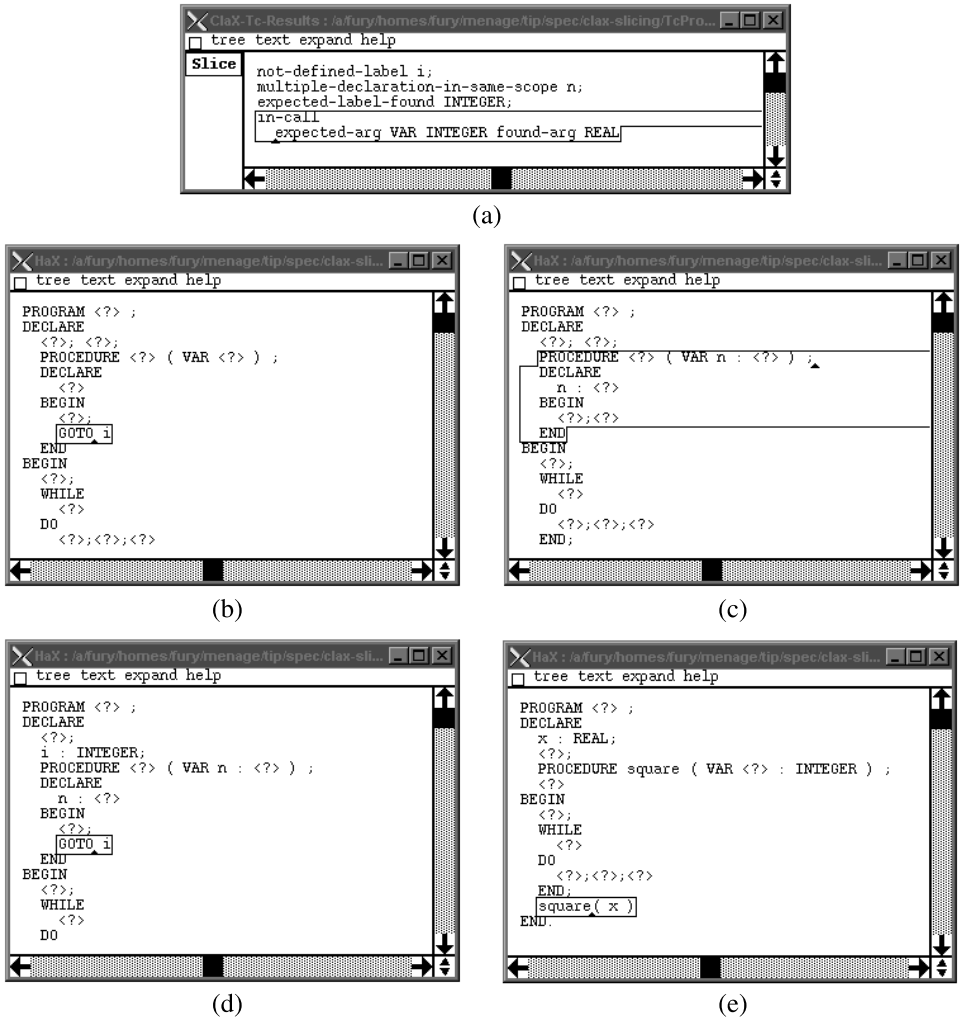

Fig. 2. (a) The error messages reported by the CLaX environment. (b)–(e) Slices reported by the CLaX environment for each of the type errors shown above.

(1) The first error, `not-defined-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 is declared as an integer, but 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 Figures 2(b)–(e), the slices obtained for each of the four error messages of Figure 1 are shown.[3] 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 toward an error message *e*" may be characterized informally as follows: There exists a mapping from placeholders to subterms such that type-checking the program obtained by replacing the placeholders in the slice with these subterms is guaranteed to produce the same error, *e*.

(1) Figure 2(b) shows the slice for the `not-defined-label` error. Clearly, the `GOTO i` statement is the source of the error, because no statement is labeled `i`.

(2) Figure 2(c) shows the slice 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(d) 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 names declared in the inner scope occur 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(e) 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 structural information such as `BEGIN` and `END` keywords and declaration and statement list separators that are not directly related to an error. In

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

addition, slices may contain "partial" statements such as `IF` and `WHILE` constructs whose condition and body are omitted from the slice (e.g., see Figure 2(e)). Section 4.8 discusses how such "syntactic noise" can be removed from slices.

1.3 Organization 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, as well as dependence tracking, the mechanism for computing slices are presented. Section 4 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. In Section 5, we study how a number of other features of type systems such as subtyping can be modeled, and we report on experiments with a type-checker for an ML subset. Section 6 discusses related work. Conclusions and future work are presented in Section 7.

2. SPECIFICATION OF STATIC SEMANTICS AND TYPE-CHECKING

A *static semantics* specification determines only 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. For example, in order to determine the type-correctness of an assignment statement `x := y`, information from (at least) three sources is involved: In addition to the assignment itself, the declaration of `x` and the declaration of `y` are required, which may occur in different scopes. In cases where there is a mismatch between the types of `x` and `y`, an informative error message ideally involves positional information from all three sources. In the absence of automated mechanisms for tracking positional information, specifications tend to become cluttered with extra function arguments and additional machinery for combining positional information, making the specification harder to understand.

In a previous paper [Dinesh and Tip 1992], we introduced an abstract interpretation style for writing static semantics specifications. In a nutshell, this style advocates the following:

—Rewriting program constructs to their *type*. This is accomplished by distributing type information over each program construct, and replacing each construct with its type. For example, an expression `x + y` where

Prog	::=	**declare** *Decls* **begin** *Stats* **end**
Decls	::=	{ *Decl* ";" }*
Stats	::=	{ *Stat* ";" }*
Decl	::=	*Id* : *Type*;
Stat	::=	*Exp* := *Exp*;
Exp	::=	*Id* | *Exp* + *Exp*
Type	::=	*BasicType*
BasicType	::=	**natural** | **string**

Fig. 3. BNF grammar for language $\mathcal{L}$.

both x and y are declared as NATURAL numbers is transformed into NATURAL + NATURAL.

—Evaluating type expressions at an abstract level. For example, the expression NATURAL + NATURAL is reduced to NATURAL.

—Only specifying the type-correct cases. Consequently, only type-incompatible expressions remain after the rewriting process terminates.

—In a separate step, the remaining irreducible expressions are rewritten into human-readable error messages. For example, an expression NATURAL + STRING is rewritten to a message "Operands of + should have the same type."

Operationally, the static semantics specification describes a transformation of a program to a set of type-expressions for program constructs that are type-incompatible. By only specifying type-correct cases, specifications are very compact and easy to understand.

We will illustrate these ideas using a small example language $\mathcal{L}$ of straight-line flow programs. An $\mathcal{L}$ program consists of the keyword declare, followed by a list of zero or more declarations, a keyword begin, a list of zero or more assignment statements, and a keyword end. $\mathcal{L}$ contains an overloaded "+" operator that can be used for adding natural numbers, and for concatenating strings, but that cannot be applied to arguments of different types. Figure 3 shows a BNF grammar for $\mathcal{L}$. In grammars such as the one of Figure 3, we will use notation of the form { X ";" }* to denote a list of zero or more elements of type X, separated by semicolons. Similarly, we will use { X ";" }+ to denote a list of one or more elements of type X, separated by semicolons.

Figure 4 shows a static semantics specification for determining the validity of $\mathcal{L}$-programs.[4] In the rules of Figure 4, we will follow the convention that typewriter font is used for function symbols, and *italics* font is used for variables. Each variable has a *sort* (i.e., a specification-level type), roughly corresponding to the nonterminals of the grammar of Figure 3, and only matches subterms of that sort. For example, the sort of variable

[4]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.

```
[Tc1] tc(declare Decls begin Stats end)                    -> dist(Stats, tenv(Decls))
[Tc2] dist(Stat;Stat+, Tenv)                               -> dist(Stat, Tenv); dist(Stat+, Tenv)
[Tc3] dist(Exp1 := Exp2, Tenv)                             ->
        check(dist(Exp1, Tenv) := dist(Exp2, Tenv), Tenv)
[Tc4] dist(Exp1 + Exp2, Tenv)                              -> dist(Exp1, Tenv) + dist(Exp2, Tenv)
[Tc5] dist(Id, Tenv)                                       -> type-of(Id, Tenv)
[Tc6] type-of(Exp, tenv(Decl1*; Exp:Type;Decl2*))          -> Type
[Tc7] BasicType + BasicType                                -> BasicType
[Tc8] check(BasicType := BasicType, Tenv)                  -> correct
```

Fig. 4. Static semantics specification for determining the validity of assignments.

Decls in rule [Tc1] corresponds to the nonterminal *Decls*, and only matches a list of declarations. We will typically use the nonterminals of Figure 3 with numbered subscripts so that the sort of a variable can be inferred from its name.

We will now address the rules of Figure 3 in some detail. Rule [Tc1] defines the top-level function tc for checking a program, which 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 symbol dist. For the simple language we study here, the type-environment consists of the declaration section of the program, to which a constructor function tenv is applied. Rule [Tc2] expresses the distribution of type-environments over lists of statements. The sort of variable $Stat+$ in rule [Tc2] is { *Stat* ";" }+, which implies that it matches a list of statements of length one or more. Rule [Tc3] distributes the type environment over the left-hand side and right-hand side of an assignment statement. Function symbol check is introduced in the right-hand side of [Tc3] in order to perform some additional checks on the validity of assignment statements. While it is possible to rewrite the specification without the check function, the current form of the specification will make it easier to accommodate language extensions such as pointers, records, and subtyping in Section 5. Rule [Tc4] distributes type environments over the operands of the '+' operator. The next rule, [Tc5], specifies how an identifier is reduced to its type, using an auxiliary function type-of, which is defined in [Tc6]. Note that the variables $Decl_1^*$ and $Decl_2^*$ in [Tc6] match any sublist of (zero or more) declarations in the type environment. Rule [Tc7] expresses the abstract evaluation of +-expressions. It is important to understand that the left-hand side of the rule contains two occurrences of the same variable, *BasicType*, and that the rule is therefore only applicable to terms that contain two *identical* subterms at the corresponding locations. Finally, rule [Tc8] states that the assignment of an assignment is correct if the left-hand side and the right-hand side of the assignment correspond to the same basic type.

As an example, consider checking the following program block:

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

Application of [Tc1] results in:

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

Application of [Tc2] yields:

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

At this point, [Tc3] can be applied to both components, producing:

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

The left-hand sides of both assignments can be reduced to their types using [Tc5] and [Tc6], resulting in:

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

Using [Tc4], [Tc5], and [Tc6], the right-hand sides of the assignments can be simplified:

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

Using rule [Tc7] to abstractly evaluate the first +-expression, we obtain:

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

Finally, application of [Tc8] yields the final result:

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

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

Figure 5 shows a set of rewriting rules that define a function `msgs` that transforms the cryptic terms produced by the specification of Figure 3 into human-readable messages. The rules of Figure 5 assume that the term to which they are applied is fully normalized with respect to the rules of Figure 4.

Rule [Er1] distributes the `msgs` function over a list of type-checking results (i.e., transformed statements) that was produced by the specification of Figure 4. The `concat` function introduced in the right-hand side of [Er1] is defined in [Er4] and serves to concatenate error messages. Rule [Er2] transforms the `correct` constant that was produced from a type-correct statement into a message `no-errors`. Since the programmer is

```
[Er1]  msgs(Result1 ; Result2)              -> concat(msgs(Result1), msgs(Result2))
[Er2]  msgs(correct)                        -> no-errors
[Er3]  Msg*1 ; no-errors; Msg*2             -> Msg*1 ; Msg*2
[Er4]  concat(Msg*1 , Msg*2)                -> Msg*1 ; Msg*2
[Er5]  msgs(check(Exp1 := Exp2 , Tenv)) -> msgs(Exp2)
       when is-type(Exp1) = true, is-error-exp(Exp2) = true
[Er6]  msgs(check(Exp1 := Exp2 , Tenv)) -> msgs(Exp1)
       when is-error-exp(Exp1) = true, is-type(Exp2) = true
[Er7]  msgs(check(Exp1 := Exp2 , Tenv)) -> concat(msgs(Exp1), msgs(Exp2))
       when is-error-exp(Exp1) = true, is-error-exp(Exp2) = true
[Er8]  msgs(check(Exp1 := Exp2 , Tenv)) -> assignment-incompatible Exp1 := Exp2
       when is-type(Exp1) = true, is-type(Exp2) = true
[Er9]  msgs(Exp1 + Exp2)                    -> operands-of-+-should-have-same-type
       when is-builtin-type(Exp1) = true, is-builtin-type(Exp2) = true
[Er10] msgs(Exp1 + Exp2)                    -> msgs(Exp1)
       when not-builtin-type(Exp1) = true, is-builtin-type(Exp2) = true
[Er11] msgs(Exp1 + Exp2)                    -> msgs(Exp2)
       when is-builtin-type(Exp1) = true, not-builtin-type(Exp2) = true
[Er12] msgs(Exp1 + Exp2)                    -> concat(msgs(Exp1), msgs(Exp2))
       when not-builtin-type(Exp1) = true, not-builtin-type(Exp2) = true
[Er13] msgs(type-of(Id, Tenv))              -> undeclared-variable: Id

[Er14] is-builtin-type(BasicType)           -> true
[Er15] is-type(Exp)                         -> true when is-builtin-type(Exp) = true
[Er16] is-error-exp(dist(Exp,Tenv))         -> true
[Er17] is-error-exp(type-of(Exp, Tenv))     -> true
[Er18] is-error-exp(Exp + Exp')             -> true
[Er19] not-builtin-type(Exp)                -> true when is-error-exp(Exp) = true
```

Fig. 5. Specification for postprocessing irreducible subterms in order to obtain human-readable error messages.

generally not interested in such messages, rule [Er3] removes any no-errors message from the list of error messages.

The next four rules, [Er5]–[Er8] are concerned with irreducible constructs of the form check(Exp_1 := Exp_2, $Tenv$). Such *conditional* rewriting rules are only applicable if all of their conditions hold. In order to evaluate conditions, each side of the condition is reduced to normal form (i.e., a term that cannot be rewritten further), and the resulting normal forms are checked for syntactic equality. In the case of [Er5]–[Er8], the conditions use auxiliary functions is-type and is-error-exp (which will be discussed shortly) to determine if the subterms matched against Exp_1 and Exp_2 represent a valid type, or an irreducible subterm derived from a type-incorrect expression. Rule [Er8] addresses the case where both subterms are type-correct, and the assignment is irreducible because two incompatible types are used (e.g., natural := string), and generates a message about this problem. The other three rules deal with situations where the right-hand side ([Er5]), the left-hand side ([Er6]) or both ([Er7]) contain a type error. In each case, a message is generated for the incorrect side(s) of the assignment by applying the msgs function to the appropriate expression(s).

Rules [Er9]–[Er12] are concerned with expressions of the form Exp_1 + Exp_2, and rely on auxiliary functions is-builtin-type and not-builtin-type that will be discussed shortly. Such expressions are valid if both operands are of the same built-in type. Rule [Er9] deals with the situation where + is applied to two different built-in types, and generates an appropriate error message, operands-of-+-should-have-same-type. The other three rules, [Er10]–[Er12], address situations where one or both operands are irreducible terms that were derived from a type-incorrect expression, and distribute the msgs function over these expressions. Rule [Er13] is concerned with postprocessing irreducible terms of the form type-of (Id, $Tenv$). Such expressions are irreducible because there is no entry for Id in the type environment, and an error message undeclared-variable: Id is generated.

Rules [Er14]–[Er19] define auxiliary functions is-builtin-type, is-type, is-error-exp, and not-builtin-type that inspect the syntactic structure of their argument to determine if the expression corresponds to a valid built-in type, a valid type, an irreducible term that was derived from a type-incorrect expression, and a nonbuilt-in type, respectively. The reader may have observed that the auxiliary functions defined in rules [Er14]–[Er19] could be specified a bit more succinctly, for example by combining the is-type and is-builtin-type functions. However, the adopted approach will allow us to extend $\mathcal{L}$ with pointers, records, and subtyping in Section 5 without making any changes to the existing rules.

As an example, we will postprocess the term we obtained earlier by applying the rules of Figure 5:

```
msgs(correct;
  check(natural := string + natural, tenv(x:natural;y:string)))
```

Applying [Er1] produces:

```
concat(msgs(correct),
  msgs(check(natural := string + natural,
  tenv(x:natural;y:string))))
```

At this point, [Er5] can be applied because only the left-hand side of the assignment is a valid type. Note that the conditions of this rule are satisfied because is-type(natural) can be reduced to true, and is-error-exp(string + natural) can be reduced to true. Hence, we obtain:

```
concat(msgs(correct), msgs(string + natural))
```

The first msgs-subterm can be rewritten to no-errors using [Er2], and the second msgs-subterm can be rewritten using [Er9]. The term now looks as follows:

```
concat(no-errors, operands-of-+-should-have-same-type)
```

The no-errors subterm is eliminated by applications of rules [Er4] and [Er3]. Hence, the final result is:

```
operands-of-+-should-have-same-type
```

3. TERM REWRITING AND DEPENDENCE TRACKING

3.1 Term Rewriting

In the previous section, specifications were "executed" by repeatedly applying rewriting rules to terms—a mechanism referred to as *term rewriting*. Both theoretical properties of term rewriting systems [Klop 1992] such as termination behavior, and efficient implementations of rewriting systems [Kamperman 1996; Kamperman and Walters 1996] have been studied extensively.

Term rewriting [Klop 1992] 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 rewriting 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 rewriting rule [Bergstra and Klop 1986] (such as `[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.

3.2 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 [1994; 1998]. Dependence tracking establishes relationships between function symbols that occur in the terms that arise during a term rewriting process. Informally, these relationships reflect the (parts of) an initial term that are responsible for the occurrence of (parts of) a result term.

We will now present a brief overview of dependence tracking; for a complete formal treatment, the reader is referred to Field and Tip [1998]. Dependence tracking assumes the existence of a sequence r of rewriting steps that reduce a term T_0 to a term T_n. A *slicing criterion* in this setting consists of a subcontext C_n of term T_n, where a *subcontext* is defined as a set of function symbols that occur contiguously in a term. A *term slice* is defined as any subcontext C_0 of the initial term T_0 such that: (i) C_0 can be rewritten to a subcontext D_n of T_n using a *subreduction* r' of r (roughly speaking, a subset of the rewriting steps in r), and (ii) D_n is a supercontext of C_n. This definition of a term slice is shown pictorially in Figure 6.

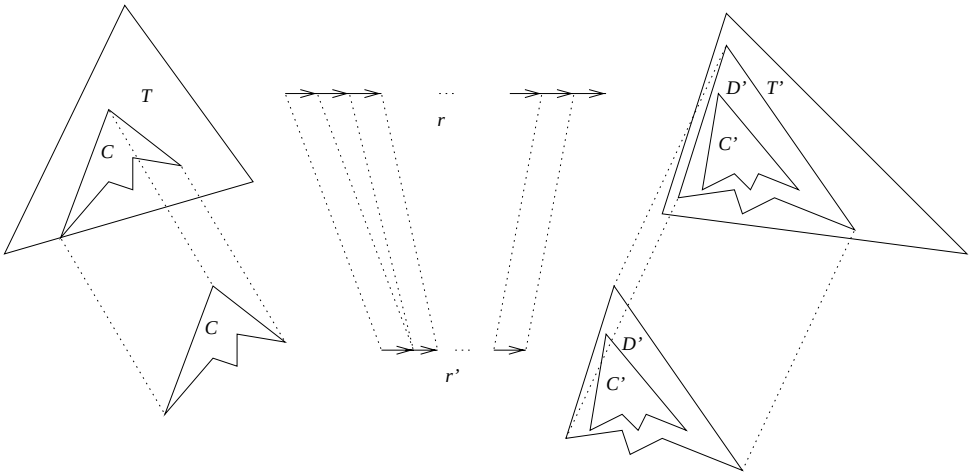

Fig. 6. Depiction of the definition of a term slice. For a given sequence r of rewrite steps that reduce a term T to a term T', a *slicing criterion* is defined as a subcontext (i.e., a contiguous set of function symbols) C' that occurs in T'. Then, a *term slice* C is a subcontext of the original term T such that C can be rewritten to a subcontext D' of T' using a subset r' of the reduction steps in r, and such that D' contains C'.

Dependence tracking is a method for computing term slices that relies on an analysis of rewriting rules to determine how the application of rewriting rules causes *creation* of new function symbols, and the *residuation* (i.e., copying, moving around, or erasing) of previously existing subterms. We will use the following simple specification of integer arithmetic (taken from Tip [1995a]) to illustrate these principles:

$$[\texttt{A1}]\ \texttt{intmul}(\texttt{0}, X) = 0$$

$$[\texttt{A2}]\ \texttt{intmul}(\texttt{intmul}(X, Y), Z) = \texttt{intmul}(X, \texttt{intmul}(Y, Z))$$

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

$$T_0 = \texttt{intsub}(3, \underline{\texttt{intmul}(\texttt{intmul}(0, 1), 2)}) \rightarrow [\texttt{A2}]$$

$$T_0 = \texttt{intsub}(3, \underline{\texttt{intmul}(\texttt{0,}\ \ \texttt{intmul}(1, 2))}) \rightarrow [\texttt{A1}]$$

$$T_2 = \texttt{intsub}(3, 0)$$

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

—The outer context $\texttt{intsub}(3, \bullet)$ of T_0 ('•' denotes a missing subterm) is not affected at all, and therefore reappears in T_1 and T_2.

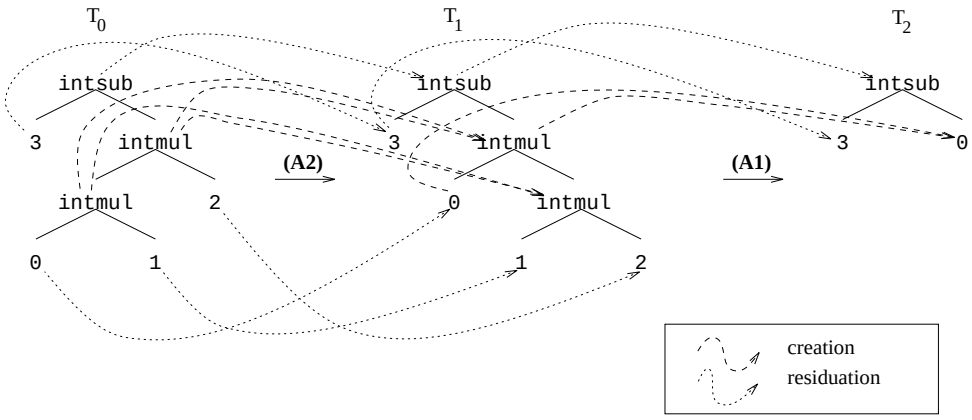

Fig. 7. Example of creation and residuation relations.

—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 Field and Tip [1994; 1998]. Notions of creation and residuation are defined for single rewrite steps, based on the syntactic structure[5] of the applied rules. Roughly speaking, the dynamic dependence relation for a sequence of rewriting steps r consists of the transitive closure of creation and residuation relations for the individual steps in r. The implementation computes the transitive creation and residuation relationships incrementally, with respect to all function symbols.

Figure 7 shows all residuation and creation relations for the example reduction discussed above. The term slice with respect to the entire term T_2 can be determined 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, •), •)). According to the semantics of term slices, there exists a sequence of rewriting steps that reduce the context intsub(3, intmul(intmul(0, •), •)) to the slicing criterion intsub(3,0). In fact, *any* term obtained by replacing

[5]In the presence of so-called *left-nonlinear* rules and *collapse rules*, the notions of creation and residuation become more complicated and also depend on the "history" of previously applied rewriting rules. This is discussed at greater length in Field and Tip [1994; 1998].

the holes in `intsub(3, intmul(intmul(0, ●), ●))` with arbitrary subterms will yield a term that can be rewritten to `intsub(3,0)`.[6]

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

3.3 The Effect of Explicit List Traversal 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 revealed that the *accuracy* of the computed slices inversely depends on the degree to which the specification explicitly traverses lists of syntactic constructs (e.g, statements or declarations). As a general principle, *more* explicit traversal of syntactic structures in a specification lead 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 the explicit traversal of a list of statements, this 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(y, tenv(x: integer; y: string; z: integer))
```

according to the specification of Figure 4. By applying rule `[Tc6]`, this term rewrites to the constant `string`. By tracing back the dynamic dependence relations, we find that the context

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

was needed to create this result. Now suppose that instead of rule `[Tc6]`, we use the following two rules for reducing the same term:

```
[Tc6a] type-of(Exp, tenv ((Exp: Type; Decl*)) −> Type
[Tc6b] type-of(Exp1, tenv (Exp2: Type; Decl*)) −>
type-of(Exp1, tenv(Decl*))
   when Exp1 != Exp2
```

The resulting term would be the same as before: the constant `string`, which is obtained by first applying rule `[Tc6b]` followed by applying rule `[Tc6a]`. However, the subcontext needed for creating this result would now consist of:

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

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

[6]This stronger property does not necessarily hold in cases where N rewriting processes are staged such that the input to rewriting process $I + 1$ is a normal form of rewriting process I.

traversed is made explicit in the specification. Informally, the result `string` now depends 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) reduces order-dependence, and therefore improves slice accuracy. It should be mentioned, however, that the use of list-matching is potentially more expensive than explicit traversal of lists constructed with binary `cons` operators. Performing a *single* list match is not inherently more expensive than is the case with explicit traversal, because in each case the list is traversed sequentially, either during the matching of the rule's left-hand side, or as a series of explicit rewrite steps. However, in cases where operations are repeatedly applied to different elements of a list (e.g., distributing an operation over all elements of a list), a subterm may be matched repeatedly. If we make the simplifying assumption[7] that the cost of matching a list of length n involves n elementary match operations on the subterms that represent list elements, the explicit traversal approach would require $O(n)$ of such operations, whereas the list match approach may require $O(n^2)$ operations.[8] In practice, we have not observed performance problems due to the use of list matching, and it is interesting to note that, in our experience, a specification based on list matching is nearly always more concise and readable, and preferable from a pure specification point of view.

4. A CASE STUDY: TYPE-CHECKING THE CLAX LANGUAGE

In order to validate our method, we applied our techniques to an existing static semantics specification for a Pascal-like imperative programming language named CLaX. CLaX features nested scopes, overloaded operators, arrays, goto statements, and procedures with reference and value parameters, and was originally developed as the demonstration language of the ESPRIT-II Compare (Compiler Generation for Parallel Machines) project [Alt et al. 1994]. The original (informal) description of the semantics of CLaX can be found in The COMPARE Consortium [1991]. Since then, CLaX has been used as a basis for various software tools, including type-checkers, interpreters, and debuggers [Dinesh 1994; 1996; Dinesh and Tip 1992; van Deursen et al. 1996; Tip 1995a], as well as a test-bed for origin-tracking techniques [van Deursen 1994a; van Deursen et al. 1993; Field and Tip 1994].

The type-checker specification for CLaX has essentially the same structure as the $\mathcal{L}$-specification we presented earlier, but is significantly larger (about 16 pages of specification text for the syntax and static semantics).

[7]These assumptions no longer hold if the same (list) variable occurs more than once in the rewriting rule's left-hand side.

[8]Although optimizations to avoid redundant repeated list matching are conceivable, we are not aware of any work on this topic.

Language features such as gotos, nested scopes, procedures, and arrays introduce some additional complexity, but pose no fundamental problems. The remainder of this section will address some of the highlights of the CLaX specification; a complete annotated listing of this specification appears in Dinesh and Tip [1997a]. The snapshots of Figures 1 and 2 were obtained by applying dependence tracking to the CLaX type-checker.

We use the combined formalism ASF+SDF to define the syntax, the static semantics, and the dynamic semantics of CLaX. ASF+SDF is a combination of the Algebraic Specification Formalism ASF [Bergstra et al. 1989] and the Syntax Definition Formalism, SDF [Heering et al. 1989]. 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 [van Deursen et al. 1996], 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 [Klint 1993] is an implementation of ASF+SDF. By interpreting equations as rewriting rules, specifications can be executed as term rewriting systems.

4.1 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 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 van Deursen et al. [1996] and Klint [1993].

Figure 8 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
```

```
%% 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
```

Fig. 8. Some modules of the ASF+SDF specification of the CLaX syntax.

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 module 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. During rewriting, variables will only match subterms of the corresponding sort (or subsorts thereof). In the above case, the variable will only match sort PROGRAM.

Module SyntaxHeaders, which defines the syntax of declarations and procedure headers, is also shown in Figure 8. Various kinds 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.

Figure 9 shows an example of a CLaX program.

4.2 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 Figure 10), the type-checker specification imports the CLaX syntax of module mod SyntaxProgram that was discussed previously. The CLaX type-checker performs (roughly) the following steps in order to type-check a BLOCK of statements:

```
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.
```

Fig. 9. Example of a CLaX program.

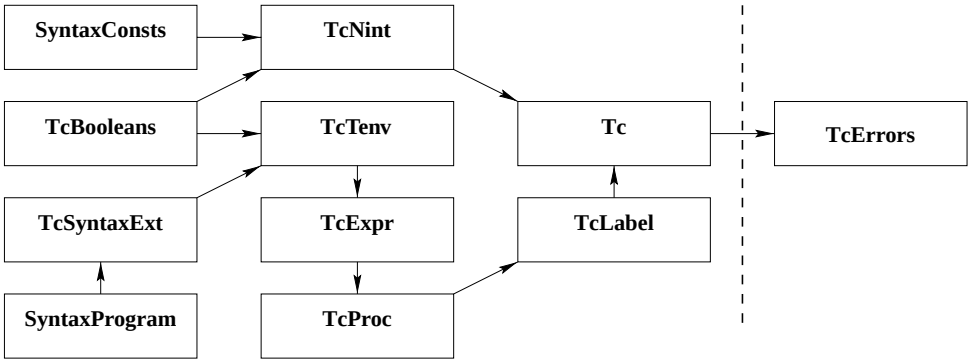

Fig. 10. Import diagram for the type-checking modules. The dashed line indicates the separation between the type-checking phase, and the postprocessing phase in which human-readable error messages are produced.

—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 statements inside these constructs are hoisted, 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 representation. 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`'.

—Type-correct statements (e.g., assignments whose left-hand side and right-hand side are both rewritten to '`INTEGER`') are reduced to the constant '`true`'.

—Human-readable error messages are generated from the list of remaining abstract expressions. 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.

4.3 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 define the auxiliary data structures used by the type-checker. To illustrate this point, Figure 11 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 zero or more `TENV`s.

Module `TcTenv` also defines an auxiliary function `type-of` for computing 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 reduce an expression 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 Figure 11 (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.

4.4 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:

```
%% 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
```

Fig. 11. Module TcTenv of the ASF+SDF specification of the CLaX type-checker.

```
[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 [t17] 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 nonsimple 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., nonarray) type. Equation [R4] rewrites type-correct WRITE statements to true.

4.5 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

```
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.
```

Fig. 12. Example CLaX program that contains several type errors.

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
```

In order to guarantee that all irreducible abstract values are transformed into human-readable error-messages, it suffices to determine that exactly one of the following properties holds for each abstract expression t:

—A rewriting rule of the static semantics specification can be applied to t (assuming that the appropriate arguments such as type environments are supplied as well). In this case, t is a term that can be simplified further, and no error message needs to be generated.

—A rewriting rule of the error-message generator can be applied to t.

If neither of the above properties holds, additional rules need to be added to the error message generator. In practice, we found that the set of type-incorrect abstract expressions is fairly small, and that determining the appropriate set of error-message generation rules is not very difficult. In the presence of expressions with multiple errors the question arises

whether to attempt the generation of a separate error message for each problem. Our approach is to generate a message for the top-level problem only, since it is likely that multiple errors are interrelated.

4.6 An Extended Example

As an example, we will study the type-checking of the CLaX program of Figure 12 in some detail. 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 now represented by their type, INTEGER. However, since strings are not first class TYPEs in CLaX (there are no operations on 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 by a recursive function collect that collects the declarations in a set of nested scopes into a combined type environment (sort TENV-LIST, see Section 4.3). Function collect has two arguments: a TENV-LIST of type environments constructed so far, and a BLOCK that needs to be processed. Whenever a procedure declaration is encountered, an entry for the procedure is added to the type environment for the current scope, and a separate collect "process" is spawned for the type-checking of the statements in the procedure's body, to which the declarations in the procedure body will be added. For instance, before entering the type-checking of the statements in procedure square, a snapshot would look as follows:

```
collect( [n: REAL; i: INTEGER; square: PROC (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, an irreducible term `unique(step step)` is produced, expressing 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. The parts of the term corresponding to procedure `square` now look as follows:

```
unique(step step) &
distribute([n: REAL; i: INTEGER; square: PROC (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 &
true &
LABEL-TYPE := INTEGER &
LABEL-TYPE := LABEL-TYPE * REAL &
...
```

Note that the assignment `x := 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`.[9] There are two occurrences of this error-message—the other message is generated for the assignment `step := n`.

We will now briefly discuss the main steps involved in type-checking the procedure call `square(n)` in the body of the main program. After the constructing the type environment from the declarations of the main program, and distributing this environment over each statement in the main program's body, the part of the rewritten term that corresponds to the call site looks as follows:

[9]Applying the multiplication operator to an operand of type `LABEL` is also a type error, but no message is generated because the current implementation does not attempt to generate separate error messages for erroneous subexpressions.

```
distribute([n: REAL; i: INTEGER; square: PROC (INTEGER)],
  BEGIN square(n) END)
```

In order to type-check a procedure, two conditions need to be verified. First, we must verify that the symbol `square` is indeed declared as a procedure whose formal parameter types match the types of the supplied arguments. Second, we have to check that any formal `VAR` parameter i.e., call-by-reference parameter) corresponds to an actual parameter that is a variable because it is a type error to pass a constant value or nontrivial expression as an actual by-reference parameter. The `distribute` function rewrites the above term as follows:

```
isproc(type-of([n:REAL; i:INTEGER; square:PROC(INTEGER)], square)
  (types-of([n:REAL; i:INTEGER; square:PROC(INTEGER)], n))) &
vararg(type-of([n:REAL;i:INTEGER;square:PROC(INTEGER)],square)(n))
```

Two auxiliary functions, `isproc` and `vararg`, perform the two checks mentioned above, and rely on functions `type-of` and `types-of` to look up the type of a name, and a list of names, respectively. Reducing all `type-of` and `types-of` subterms yields:

```
isproc( PROC( INTEGER) (REAL)) & vararg(PROC( INTEGER) (n))
```

Function `isproc` reduces to `true` if the types of the formal and actual parameters correspond. In this case, the formal parameter type `INTEGER` does not correspond to the actual parameter type `REAL`, and the resulting expression will therefore remain. A `vararg`-subterm reduces to `true` if each formal reference parameter corresponds to an actual parameter that is a variable. Since this is the case, the resulting term is:

```
isproc( PROC( INTEGER) (REAL)) & true
```

In the error-processing phase, the `isproc`-subterm is rewritten to:

```
in-call expected-arg INTEGER found-arg REAL
```

4.7 Lessons Learned

We will now discuss 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 "undid" the changes that were made to the specification in order get reasonable error locations using origin tracking (this will be discussed in more detail in Section 6). As it turns out, almost all of the issues discussed below have the flavor of eliminating explicit list traversals or "overspecification."

4.7.1 *Overspecification: Unnecessarily Specific Matching*. In a number of places, the type-checker specification of Dinesh and Tip [1992] 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
```

to express 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.

4.7.2 *Flattening of Control-Flow Structures*. Control-flow structures such as IF and WHILE statements have little to do with type-checking. For both types of statements, we need to verify that the control predicate is of type BOOLEAN, but the type-(in)correctness of any statement that occurs inside the "branches" of an IF and WHILE construct does not depend in any way on the control predicate. We exploit this observation by hoisting all statements that occur inside IF and WHILE constructs, in a process that we will refer to as "flattening" of control-flow structures. Flattening has the important benefit that it allows any operation to assume that statement lists are flat. This makes the specification more concise and declarative, because it obviates the need for distributing various operations over IF and WHILE constructs, and it isolates the structural traversal of these constructs in one place.

Figure 13(a) shows how we use list matching to specify flattening. 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. It is important to realize that these equations may be applied to *any* statement list at *any* time; there is no *explicit* call to a flattening function. The TEST statement generated by each of these rules is a "generic" conditional statement for which we check if its control predicate is of type BOOLEAN using equations [D10] and [R5]. The distribute function of [D10] distributes a type environment over a TEST statement, and the resulting expression is eventually rewritten to its type. Following our strategy to only specify type-correct cases, rule [R5] rewrites type-correct TEST statements to true so that in the end only TEST statements derived from type-incorrect control predicates remain.

Figure 13(b) shows how flattening was specified in a previous version of the CLaX specification [Dinesh and Tip 1992]. Here, a function flat was explicitly applied to a statement list before type-checking any statements in that list. This function flat explicitly traversed a list of statements by recursively applying itself to sublists (rule [FL12]). Statements other than IF and WHILE are left unchanged by flat (equations [FL0]–[FL8]). For IF and WHILE constructs, flat hoisted the nested statement lists outside these constructs (equations [FL9]–[FL11]). The drawback of this approach is that a dependency of each statement in a "flattened list" on the surrounding DECLARE–BEGIN–END or BEGIN–END symbol(s) is established, and all such symbols were consequently showing up as "noise" in the computed slices. The new approach we discussed above does not suffer from this problem.

```
[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*

[D10]   distribute(_Tenv*, BEGIN TEST _Expr END END) = TEST type-of(_Tenv*, _Expr) END
[R5]    TEST BOOLEAN END = true
```

(a)

```
[FL0] flat _StatAux = _StatAux´ ; _StatSeq´*
      ============================================
      flat _Id : _StatAux = _Id : _StatAux´ ; _StatSeq´*
[FL1] flat _Expr := _Expr´ = _Expr := _Expr´
[FL2] flat _Id = _Id
[FL3] flat _Id ( _ExprList ) = _Id ( _ExprList )
[FL4] flat READ ( _Expr ) = READ ( _Expr )
[FL5] flat WRITE ( _Expr ) = WRITE ( _Expr )
[FL6] flat WRITE ( _String ) = WRITE ( _String )
[FL7] flat GOTO _Id = GOTO _Id
[FL8] flat =
[FL9] _StatSeq´ = flat _StatSeq
       ========================
      flat IF _Expr THEN _StatSeq END = IF _Expr THEN END ; _StatSeq´
[FL10] _StatSeq´´ = flat _StatSeq ; _StatSeq´
       ================================
      flat IF _Expr THEN _StatSeq ELSE _StatSeq´ END =
      IF _Expr THEN END ; _StatSeq´´
[FL11] _StatSeq´ = flat _StatSeq
       ========================
      flat WHILE _Expr DO _StatSeq END = WHILE _Expr DO END ; _StatSeq´
[FL12] _StatSeq´ = flat _Stat, _StatSeq´´ = flat _StatSeq
       ============================================
      flat _Stat ; _StatSeq = _StatSeq´ ; _StatSeq´´
```

(b)

Fig. 13. Flattening of control-flow constructs. (a) Current solution based on list-matching. (b) Previous solution based on explicit list traversal.

4.7.3 *Elimination of Correct Program Constructs.* In the original version of module `TcBooleans`, the following equation was used for the simplification of conjunctions (here, '`&`' denotes boolean conjunction):

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

This equation served to eliminate the `true` constants that originated from type-correct program constructs. Although this had the desired effect of removing the redundant `true` constants, it overspecified 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 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 associated with an error message *e* contained

adjacent type-correct constructs. The solution to this problem was to eliminate the 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 with respect to individual statements.

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

4.7.4 *Elimination of Explicit List Traversals: Duplicate Elements in Lists*. Overspecification is undesirable because it may result in overly large slices. Unfortunately, overspecification can occur in subtle ways and can be very hard to control. To illustrate this point, the original version of the function `unique` (module `TcLabel`) is shown below. 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.[10] 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 definition 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.

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

4.8 Removing Syntactic Noise

We have already observed that the computed slices may contain a certain amount of syntactic "noise" such as list separators, `BEGIN` and `END` keywords. Such constructs never convey any useful positional information, and their highlighting can easily be suppressed.

More interesting are situations where the left-hand side of a declaration shows up in the slice, but where the right-hand side is a subterm that was not involved in the generation of the error message. Here, the construct in question is only involved in the generation of the error message to the extent that *replacing it with another construct might yield a program that does not produce the same error, when type-checked*. For example, consider the slice that was shown earlier in Figure 2(d):

```
PROGRAM;
DECLARE
  <?>;
  i: INTEGER:
  PROCEDURE <?> (VAR n: <?>);
  DECLARE
    n: <?>
  BEGIN
    <?>;
    GOTO i
  END
    <?>;
BEGIN
  ...
END.
```

This slice was computed for the error message '`expected-label-found INTEGER`' shown in Figure 1. The source of the error consists of the statement '`GOTO i`' and the declaration '`i: INTEGER`'. Note the occurrence of the declaration '`n:<?>`' in the inner scope, and the parameter '`VAR n: <?>`' in the slice. These subterms do not show up as complete holes ('`<?>` ') because the error message under consideration can only be derived in the absence of redeclarations of `i` in the inner scope that would hide the `i` in the outer scope. In this case, changing either of the declarations of `n` into a declaration '`i: LABEL`' would yield a program that does not produce the same error message, and the name of variable `n` therefore shows up in the slice. While it is nearly always desirable to suppress the display of partial assignments in a slice, there are a few situations where such declarations are part of an error's location in a legitimate way. For example, the slice of Figure 2(c) contains two partial declarations for variable `n` that constitute the source of the type error '`multiple-declaration-in-same-scope n`'. Therefore, we believe that the user should be able to select whether or not to suppress the display of partial declarations.

4.9 Implementation

We have implemented our techniques in the context of the ASF+SDF Meta-environment [Klint 1993; van Deursen et al. 1996]. Experimentation

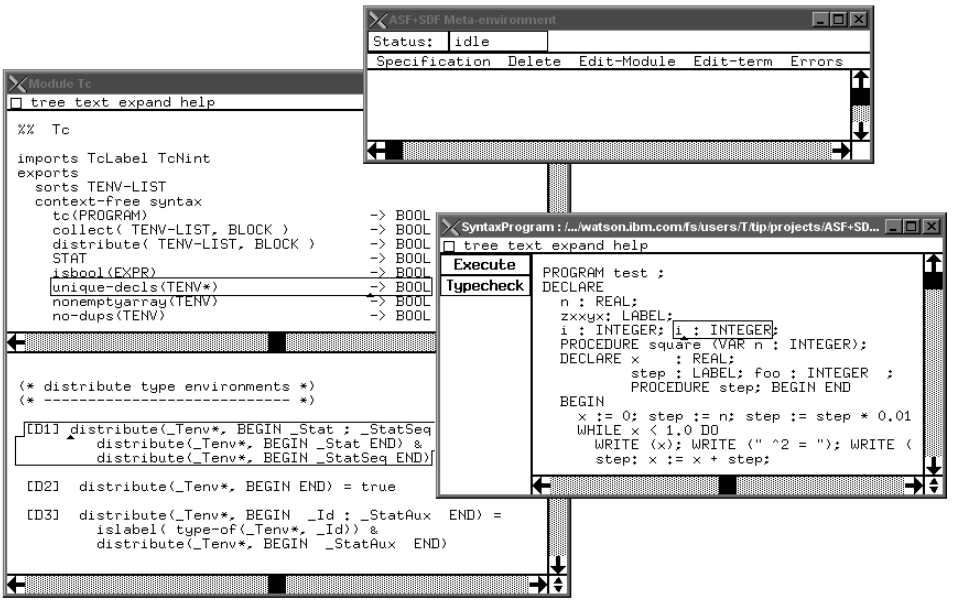

Fig. 14. A view of the ASF+SDF Meta-Environment. The top window contains a number of menus that allow the user to edit specification modules and terms in the generated environments. The window on the left shows how module Tc of the CLaX specification is edited in a syntax-directed module editor: The top part of the window contains grammar rules that define the term structure, and the bottom part contains equations (cf. rewriting rules) over the syntax defined in the top part. On the right, a term editor is shown in which CLaX programs can be edited. The **Execute** and **Typecheck** buttons attached to this window invoke an interpreter and the type-checker presented in this paper. Invoking the type-checker will result in the appearance of a window containing a list of type errors, as was shown in Figure 1.

with the CLaX specification revealed that the computed slices provide highly accurate positional information about type errors. Figures 1 and 2 show screendumps of the generated CLaX environment.

We consider the current implementation to be a proof-of-concept prototype. Our implementation is based on the ASF+SDF Meta-Environment [Klint 1993], an interpretive, Lisp-based implementation of ASF+SDF that is primarily intended for interactive language and tool design. The ASF+SDF Meta-Environment supports incremental updates to specifications in such a way that the programming environments generated from these specifications are automatically updated as well. Figure 14 shows a snapshot of the ASF+SDF Meta-Environment. While we found this environment to be extremely convenient for experimentation (changes to a specification are immediately reflected in the generated type-checking environment), the current implementation does not handle programs of more than a few hundred lines.

We conjecture that the limitations of the current implementation are mostly the repercussions of using an interpretive and interactive development environment, and *not* inherent shortcomings of rewriting-based technology.

In the past few years, the focus of the work on ASF+SDF has shifted from language and tool design toward the generation of efficient standalone environments. The *compilation* of term rewriting systems into abstract machine instructions [Walters and Kamperman 1996; Kamperman 1996; Fokkink et al. 1998] or directly into efficient C code [van den Brand et al. 1999] has been a fundamental part of this strategy, and a new version of the ASF+SDF system based on the latter approach is nearing completion. As a result of these recent advances in term rewriting technology, the new ASF+SDF system is about two orders of magnitude faster than the Lisp-based version of the system we used, and comparable in speed to modern functional programming language implementations. Recently, van den Brand et al. [1999] conducted a study in which the performance of compiled ASF+SDF specifications is compared to similar benchmark programs in Standard ML [Milner et al. 1997], Concurrent Clean [Plasmeijer and van Eekelen 1994], and Haskell [Hudak et al. 1992], and report performance results in line with the best functional language implementations.

We implemented dependence tracking by extending the ASF+SDF system's rewriting engine. We used a preexisting term-annotation mechanism to annotate each function symbol in a term with two bit-vectors representing its dependence information, and this information is updated whenever a rewriting rule is applied. The overhead associated with dependence tracking is linear in the size of the term being rewritten [Field and Tip 1998]. In practice, we measured a reduction in rewriting speed by at most a factor of two. An alternative approach for implementing dependence tracking consists of *transforming* the rewriting rules and the term being rewritten in a way that encodes dependence tracking. This approach was pursued by Fraer [1997], and will be discussed in Section 6.

5. ACCOMMODATING OTHER TYPE FEATURES

We have demonstrated how our techniques apply to imperative languages with procedures. However, we believe that our techniques can in principle accommodate any language feature. In order to illustrate how object-oriented languages can be handled, we will extend the example language $\mathcal{L}$ of Section 2 with pointers, records, and subtyping, and show how adding a relatively small set of rules to the specification of Figure 4 suffices to accommodate these features. The CLaX language can be extended similarly, although this would involve more work, due to the interaction of the new features with existing features such as arrays and static scoping.

5.1 Pointers

We begin by adding a simple notion of pointers to $\mathcal{L}$. Figure 15(a) shows the BNF rules that need to be added to the grammar of Figure 3 to model pointers. We allow types of the form '^ T', (i.e., pointer to T), for any type T. Note that this allows multiple levels of indirection. In addition, operators

```
"^" Type                                ::=    Type
"&" Exp                                 ::=    Exp
"*" Exp                                 ::=    Exp
```

(a)

```
[Tc9]   dist(& Exp, Tenv)                       -> ^ dist(Exp,Tenv)
[Tc10]  check(^ Exp1 := ^ Exp2, Tenv)           -> correct
        when check(Exp1 := Exp2, Tenv) = correct
[Tc11]  dist(* Exp, Tenv)                       -> * dist(Exp, Tenv)
[Tc12]  dist(^ Exp, Tenv)                       -> ^ dist(Exp, Tenv)
[Tc13]  * ^ Exp                                 -> Exp
```

(b)

```
[Er20]  is-ptr-type(^ Exp)                      -> true when is-type(Exp) = true
[Er21]  is-type(Exp)                            -> true when is-ptr-type(Exp) = true
[Er22]  is-error-exp(* Exp)                     -> true when is-builtin-type(Exp) = true
[Er23]  non-builtin-type(Exp)                   -> true when is-ptr-type(Exp) = true
[Er24]  is-error-exp(^ Exp)                     -> true when is-error-exp(Exp) = true
[Er25]  is-error-exp(* Exp)                     -> true when is-error-exp(Exp) = true
[Er26]  msgs(* Exp)                             -> cannot-dereference-type:  Exp
        when is-type(Exp) = true
[Er27]  msgs(^ Exp)                             -> incorrect-operand-type:  ^ Exp
        when is-type(Exp) = true
[Er28]  msgs(* Exp)                             -> msgs(Exp)
        when is-error-exp(Exp) = true
[Er29]  msgs(^ Exp)                             -> msgs(Exp)
        when is-error-exp(Exp) = true
```

(c)

Fig. 15. Extending $\mathcal{L}$ with pointers. (a) Additional grammar rules. (b) Additional type-checking rules. (c) Additional postprocessing rules.

'&' and '*' are added to the expressions syntax for taking the address of a variable, and for dereferencing an expression, respectively.

Figure 15(b) shows additional rules that need to be added to the specification of Figure 4 in order to check pointer usage.[11] Rule [Tc9] expresses the fact that taking the address of an expression whose type is T will yield an expression whose type is $\hat{}\ T$. Rule [Tc10] states that type-checking an assignment of the form $\hat{}\ T := \hat{}\ U$ succeeds if and only if type-checking an assignment $T := U$ succeeds. Rules [Tc11] and [Tc12] are concerned with distributing type environments over ^-expressions and *-expressions, respectively. Finally, rule [Tc13] states that type $*\ \hat{}\ T$ is equivalent with type T.

Figure 15(c) shows additional rules that need to be added to the specification of Figure 5 in order to produce human-readable error messages. Rule [Er20] defines an auxiliary function is-pointer-type that checks if its operand expression is of a pointer type. Next, [Er21]–[Er25] are fairly

[11]The reader should be aware that, in order to keep the amount of specification manageable, certain aspects of pointers are not modeled (e.g., we do not check if the operand of the & operator is an expression whose address can be taken). Adding such checks is straightforward.

straightforward rules that extend the previously introduced auxiliary functions `is-type`, `is-error-exp`, and `not-builtin-type` to deal with pointer expressions. Rule `[Er26]` generates an error message `cannot-dereference-type:` *Exp* from an irreducible expression *Exp* that could not be reduced due to incorrect usage of the * operator. Rule `[Er27]` generates an error message `incorrect-operand-type:` ^ *Exp* that will be produced in situations where an expression of a pointer type is used as an operand of the + operator. Finally, rules `[Er28]` and `[Er29]` process an erroneous expression to which the * or the ^ operator is applied, respectively.

As an example, we will study the checking of the following program:

```
tc(declare x:natural; y: ^ natural
begin y := &x; x := &y; *y := x; *x := *y end)
```

Applying `Tc1` results in:

```
dist(y := &x; x := &y; *y := x; *x := *y,
tenv(x:natural; y: ^ natural))
```

After several applications of `[Tc2]` and `[Tc3]`, the term looks as follows (we will use S as a shorthand for the subterm `tenv(x:natural; y: ^ natural)`):

```
check(dist(y, S) := dist(&x, S), S);
check(dist(x, S) := dist(&y, S), S);
check(dist(*y, S) := dist(x, S), S);
check(dist(*x, S) := dist(*y, S), S)
```

Using `[Tc9]` and `[Tc11]` to distribute the type environment over all &-expressions and *-expressions, we obtain:

```
check(dist(y, S) := ^ dist(x, S), S);
check(dist(x, S) := ^+ dist(y, S), S);
check(* dist(y, S) := dist(x, S), S);
check(* dist(x, S) := * dist(y, S), S)
```

At this point, rule `[Tc6]` can be applied to each `type-of`-subterm, resulting in:

```
check(^+ natural := ^ natural, S);
check(natural := ^ ^natural, S);
check(* ^ natural := natural, S);
check(* natural := * ^ natural, S)
```

We can now apply `[Tc13]` to the third and fourth assignment. The term now looks as follows:

```
check(^ natural := ^+ natural, S);
check(natural := ^ ^ natural, S);
check(natural := natural, S);
check(* natural := natural, S)
```

Rules [Tc10] and [Tc8] can be applied to the first statement, and [Tc8] to the third statement. The second and fourth statements are irreducible because they contain type errors. Hence, the final result is:

```
correct; check(natural := ^ ^ natural,
tenv(x:natural; y: ^ natural));
correct; check(* natural := natural, tenv(x:natural; y: ^ natural))
```

We will now consider how the above term is postprocessed into a human-readable error-message, by applying the msgs function of Figures 5 and 15(c): After distributing the msgs function over the constructs using rule [Er1], we have:

```
concat(msgs(correct),
concat(msgs(check(natural := ^ ^ natural,
  tenv(x:natural; y: ^ natural))),
concat(msgs(correct),
  msgs(check(* natural := natural,
  tenv(x:natural; y: ^ natural))))))
```

We can reduce both msgs(correct) subterms using rule [Er2]. Furthermore, rule [Er8] can be applied to the second msgs-subterm, using rules [Er20] and [Er21] to evaluate [Er8]'s conditions. The fourth msgs-subterm is reduced via an application of [Er6], using rule [Er22] to evaluate [Er6]'s conditions. The resulting term looks as follows:

```
concat(no-errors,
concat(assignment-incompatible: natural := ^ ^ natural,
concat(no-errors, msgs(* natural))))
```

After applying [Er26] to the msgs(* natural) subterm, and using rules [Er4] and [Er3] to eliminate the no-errors messages, we obtain the final result:

```
assignment-incompatible: natural := ^ ^ natural;
cannot-dereference-type: natural
```

5.2 Records

We will now add a simple notion of records to our example language ℒ. Our approach will be to allow variables to be declared as having a record type, and to have separate declarations for fields of records. For example, a declaration of a variable z of record type r, where r has a a field n of type natural and a field s of type string looks as follows:

```
z: record r;
field r.n: natural;
field r.s: string;
```

In addition, we extend the expression syntax with a field-selection operator '.' to select the field of a structured variable. This enables one to write statements such as x := x + z.n and x.n := x. Figure 16(a) shows how the grammar of Figures 3 and 15(a) is extended to accommodate records. Note that while this grammar enables the construction of nested records, we do not allow sequences of field-selection expressions such as a.b.c in order to keep the specification simple.

```
record Id                                   ::=    Type
field Id "." Id ":" Type                    ::=    Decl
Id "." Id                                   ::=    Exp
```

(a)

```
[Tc14] dist(Id1.Id2, Tenv)                          -> dist(dist(Id1, Tenv).Id2, Tenv)
[Tc15] dist(record Id1.Id2, Tenv)                   -> type-of(field Id1.Id2, Tenv)
[Tc16] check(record Id := record Id, Tenv)          -> correct
```

(b)

```
[Er30] is-record-type(record Id)                    -> true
[Er31] is-type(Exp)                                 -> true
       when is-record-type(Exp) = true
[Er32] not-builtin-type(Exp)                        -> true
       when is-record-type(Exp) = true
[Er33] is-error-exp(* Exp)                          -> true
       when is-record-type(Exp) = true
[Er34] msgs(dist(type-of(Id1,Tenv1).Id2, Tenv2)) -> undeclared-variable: Id1
[Er35] msgs(dist(Type.Id,Tenv))                     -> not-a-record-type: Type
       when is-builtin-type(Type) = true
[Er36] msgs(type-of(field Id1.Id2, Tenv))           -> no-field Id2 in-record Id1
[Er37] msgs(Type)                                   -> incorrect-operand-type: Type
       when is-record-type(Type) = true
```

(c)

Fig. 16. Extending ℒ with records. (a) Additional grammar rules. (b) Additional type-checking rles. (c) Additional postprocessing rules.

Records introduce a number of new type constraints that have to be verified. Previously introduced conditions such as the compatibility of the types used in assignments and +-expressions must be generalized to field-access expressions. Moreover, type violations such as the access to nonexistent fields in a record, and the access of fields from nonrecord types must be reported. Again, we will refrain from checking certain properties in order to keep the example small and accessible. For example, we do not check if records contain multiple fields with the same name but different types.

Figure 16(b) shows how the static semantics specification of Figures 4 and 15(b) is extended to check these constraints. Rule [Tc14] distributes type-environments over .-expressions by transforming an expression dist(Id_1. Id_2, $Tenv$), where T, the type of variable Id_1, is determined using a similar dist operation. The resulting term, dist(record T. Id_2, $Tenv$) is rewritten to a lookup for the type of the field, type-of(field T. Id_2, $Tenv$) by rule [Tc15]. Rule [Tc16] extends the previously introduced check function to allow assignments between variables that have the same record type, under the assumption that such assignments perform a shallow copy of the object.

Figure 16(c) shows the additional postprocessing rules needed to handle record types. Rule [Er30] introduces an auxiliary function is-record-type. Next, [Er31]–[Er33] are simple conditional rules that define the

relationship between `is-record-type` and `is-type`, `not-builtin-type`, and `is-error-exp`. The remaining rules, `[Er34]`–`[Er37]` are concerned with generating appropriate error messages from irreducible expressions. For example, rule `[Er36]` generates an error message `not-a-record-type:` *Type* from irreducible expressions that arise when accessing a field from a nonrecord variable.

As an example, we will study type-checking the following program:

```
declare
  n: natural;
  x: record r;
  field r.p: ^ natural
begin
  n := x.s;
  n := *x.p;
  n.n := n
end
```

After a number of applications of `[Tc2]` and `[Tc3]`, the term looks as follows (we will use T as an abbreviation for the subterm `tenv(n: natural; x:record r; field r.p: ^ natural)`):

```
check(dist(n, T) := dist(x.s, T), T);
check(dist(n, T) := dist(*x.p, T), T);
check(dist(n.n, T) := dist(n, T), T)
```

After reducing the `dist(n,` T`)`-subterms using rules `[Tc5]` and `[Tc6]`, applying rule `[Tc11]` to the right-hand side of the second assignment, and applying rule `[Tc14]` to the field-access expressions in all three assignments, the term looks as follows:

```
check(natural := dist(dist(x, T).s, T), T);
check(natural := * dist(dist(x, T).p, T), T);
check(dist(dist(n, T).n, T) := natural, T)
```

At this point, the three innermost `dist`-expressions are reduced using rules `[Tc5]` and `[Tc6]`, followed by an application of rule `[Tc15]` to the first and second constructs. The term now looks as follows:

```
check(natural := type-of(field r.s, T), T);
check(natural := * type-of(field r.p, T), T);
check(dist(natural.n, T) := natural, T)
```

The second subterm can be further reduced by applying rules `[Tc6]` and `[Tc13]`, producing the final result:

```
check(natural := type-of(field r.s, T), T);
correct;
check(dist(natural.n, T) := natural, T)
```

In order to illustrate the generation of human-readable error messages, we will apply the `msgs` function to the above term, and rewrite it according to the specification of Figure 16(c). After distribution of the `msgs` function

```
Id "extends" Id                                    ::=     Decl
```

(a)

```
[Tc17] exists(field Id1.Id2,tenv())                        -> false
[Tc18] exists(field Id1.Id2, tenv(Id3:Type; Decl*))
       -> exists(field Id1.Id2, tenv(Decl*))
[Tc19] exists(field Id1.Id2, tenv(field Id1.Id3 :  Type; Decl*))
       -> exists(field Id1.Id2, tenv(Decl*))
       when Id2 ≠ Id3
[Tc20] exists(field Id1.Id2, tenv(field Id3.Id4 :  Type; Decl*))
       -> exists(field Id1.Id2, tenv(Decl*))
       when Id1 ≠ Id3
[Tc21] exists(field Id1.Id2, tenv(Id3 extends Id4; Decl*))
       -> exists(field Id1.Id2, tenv(Decl*))
[Tc22] subtype(record Id, record Id, Tenv)                 -> true
[Tc23] subtype(^ Exp1, ^ Exp2, Tenv)                       -> subtype(Exp1, Exp2, Tenv)
[Tc24] subtype(record Id1, record Id2, Tenv)
       -> subtype(record Id3, record Id2, Tenv)
       when Id1 ≠ Id2, Tenv = tenv(Decl*1; Id1 extends Id3; Decl*2)
[Tc25] type-of(field Id1.Id2, Tenv)                        -> type-of(field Id3.Id2, Tenv)
       when exists(field Id1.Id2, Tenv) = false,
       Tenv = tenv(Decl*1; Id1 extends Id3; Decl*2)
[Tc26] check(Exp1 := Exp2, Tenv)                           -> correct
       when subtype(Exp2, Exp1, Tenv) = true
```

(b)

Fig. 17. Extending ℒ with subtyping. (a) Additional grammar rules. (b) Additional checking rules.

over the three constructs using [Er1], and reducing the correct-subterm using [Er2], we have:

```
concat(msgs(check(natural := type-of(field r.s, T), T)),
concat(no-errors,
msgs(check(dist(natural.n, T) := natural, T))))
```

We can apply [Er5] to the first assignment, and [Er6] to the third assignment, resulting in:

```
concat(msgs(type-of(field r.s, T)),
concat(no-errors, msgs(dist(natural.n, T))))
```

At this point, we can apply [Er36] to the first msgs-subterm, and [ER35] to the second msgs-subterm. Applications of [Er3] and [Er4] yield the final result:

```
no-field s in-record r;
not-a-record-type: natural
```

5.3 Subtyping

We now extend ℒ with a simple notion of subtyping. Figure 17(a) shows how the syntax of declarations is extended to allow declarations of the form X extends Y to express that X is a subtype of Y. Subtype-relationships will be modeled as follows:

—We will allow assignments of the form $x := y$, where the type X of x is a supertype of the type Y of y. Such an assignment will be assumed to copy the X -subobject of y into x in the style of C++ [Stroustrup 1998].

—We will also allow assignments of the form $x := y$, where the type of x is $\hat{} X$, and the type of y is $\hat{} Y$, and where X is a supertype of Y.

—A subtype is assumed to contain all fields of its supertype(s), and if a field is not found in the subtype, the search process recursively continues in its supertype.

—A field m in a subtype is assumed to *hide* a field with the same name, m, in its superclass, even if the two occurrences of m have different types.

In order to keep the amount of additional specification manageable, we will make a number of simplifying assumptions: (i) each type has at most one supertype, and (ii) there are no cyclical subtype-relationships. The specification we present here does not verify these assumptions. Verifying these assumptions would involve a fairly small number of additional rewriting rules in order to generate the appropriate error messages. Relaxing these assumptions (e.g., by allowing multiple inheritance) would involve significantly more work, but not pose any fundamental problems.

Figure 17(b) shows a set of additional rewriting rules for modeling subtyping. Rules [Tc17]–[Tc21] define a function `exists` that determines if a declaration `field` Id_1`.` Id_2 occurs in a given type environment. Since we are only interested in cases where the field does not exist, only the *false* case is specified. Rule [Tc17] states that no field exists in an empty type environment. The following four rules [Tc18]–[Tc21] are concerned with situations where the first declaration in the type environment is not a declaration `field` Id_1`.` Id_2, and rewrite the term to an `exists`-term with a type environment without the first declaration. The cases are: a declaration for a variable ([Tc18]), a declaration for a different field in the same record ([Tc19]), a declaration for a field in a different record ([Tc20]), and a declaration of a subtype-relationship ([Tc21]).

Rules [Tc22]–[Tc24] define a function `subtype`, which determines if a type is a subtype of another type. Rule [Tc22] states that any record type is a subtype of itself. Rule [Tc23] states that if X is a subtype of Y, then $\hat{} X$ is a subtype of $\hat{}Y$. Rule [Tc24] is concerned with the more interesting case where the first two arguments, Id_1 and Id_2, of `subtype` are different record types. The second condition of [Tc24] introduces a number of variables in its right-hand side that are not bound to subterms during the matching of the rule's left-hand side: ($Decl*_1$, $Decl*_2$, and Id_3). Such conditions are evaluated by normalizing the nonvariable-introducing side of the condition, and matching the resulting normal form against the condition's variable-introducing side. If the match succeeds, the condition succeeds, and the newly introduced variables are bound to the subterms they are matched against. In the specific case of rule [Tc24], the match is

used to determine the supertype Id_3 of Id_1. If this supertype exists, the rule is applicable, and the subtype-term can be rewritten to a new subtype-term that involves Id_3 and Id_2.

Rule [Tc25] is concerned with determining the type of a field that does not occur in a record Id_1 (verified by the first condition). In this case, the search for the field should continue in Id_1's supertype Id_3, which is determined in the second condition. Finally, [Tc26] rewrites assignments of the form check(record X := record Y, *Tenv)* to correct if Y is a subtype of X.

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

```
declare
  x: ^ natural;
  a: record A;
  b: record B;
  B extends A;
  field A.y: ^ natural;
  field A.z: ^ record A
begin
  b.y := x;
  a.z := &b;
  b := *a.z;
end
```

After distributing the type-environment over the statements, the term looks as follows (we will use U as an abbreviation for the subterm tenv(x: ^ natural; a:record A; b:record B; B extends A; field A.y: ^ natural; field A.z: ^ record A)):

```
dist(b.y := x, U); dist(a.z := &b, U); dist(b := *a.z, U)
```

After several applications of previously discussed rules, the term looks as follows:

```
check(type-of(field B.y, U) := ^ natural, U);
check(^ record A := ^ record B, U);
check(record B := record A, U)
```

Consider the first type-of-subterm in the above term. Since there is no field B.y, the first condition of rule [Tc25] succeeds. The second condition of [Tc25] succeeds as well, and results in binding variable Id_3 to A. After applying [Tc25], the term looks as follows:

```
check(type-of(field A.y, U) := ^ natural, U);
check(^ record A := ^ record B, U);
check(record B := record A, U)
```

After applying [Tc6] to the type-of-subterm, we have:

```
check(^ natural := ^ natural, U);
check(^ record A := ^ record B, U);
check(record B := record A, U)
```

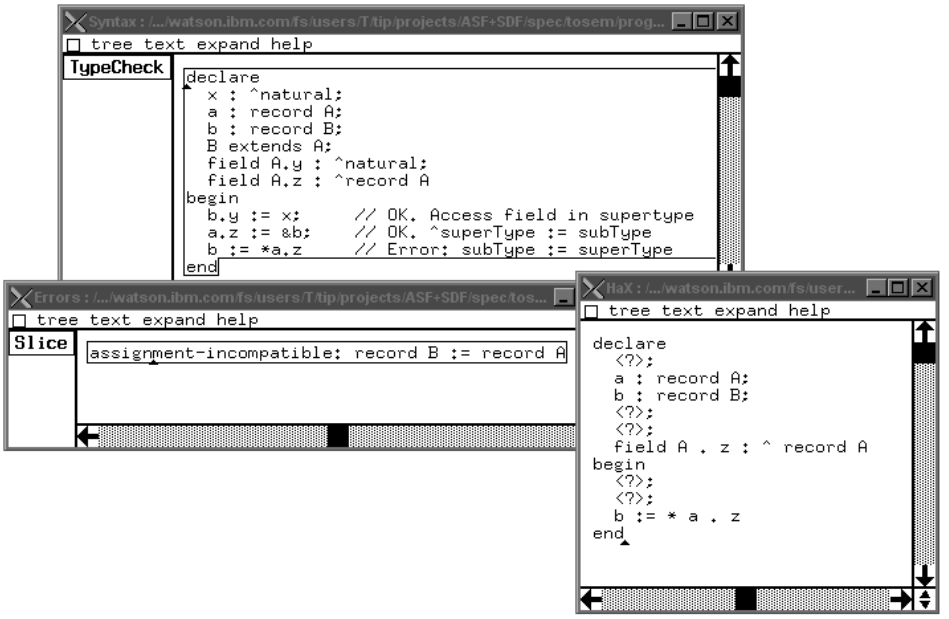

Fig. 18. Slice computed for an example program that uses subtyping.

The first `check`-subterm can be reduced to `correct` using rules `[Tc8]` and `[Tc10]`, and the second `check`-subterm using `[Tc26]` and `[Tc10]`, after determining that `subtype(record B, record A,` U`)` rewrites to `true`. The third subterm is irreducible, because `record B` is not a supertype of `record A`. Hence, the final result is:

`correct; correct; check(record B := record A,` U`)`

As it turns out, no additional rewriting rules are needed in order to produce human-readable error messages in the presence of subtyping. The above term is reduced to an error message `assignment-incompatible: record B := record A` by applying several previously presented rules, including rule `[Er8]` to produce the actual message. Figure 18 shows the slice computed for this example program. Observe that the positional information contains precisely the records and fields involved in the creation of the error message. Note in particular that the subtype-relationship `B extends A` is *not* in the slice because it is irrelevant for the given error message, which has to do with the *absence* of an a subtype-relationship `A extends B`.

5.4 Mini-ML

In order to determine how our techniques apply to languages with polymorphic types, we conducted an experiment with an existing ASF+SDF specification for Mini-ML [Clément et al. 1986] written by Hendriks [Bergstra et al. 1989; Hendriks 1991]. Mini-ML is a subset of ML [Milner et al. 1990] that contains the essential elements of ML's type system as far as type-

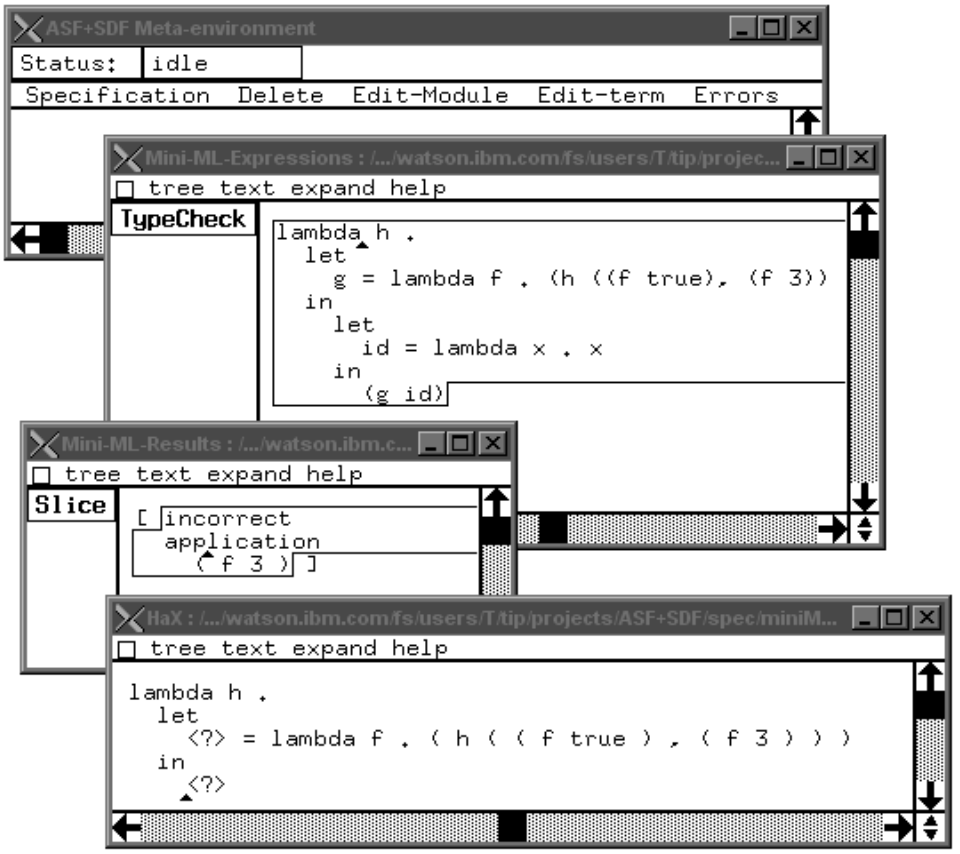

Fig. 19. View of the generated Mini-ML environment.

checking is concerned, including generic type variables, function types and cartesian product types. Hendriks' specification is similar in spirit to ASF+SDF specifications for imperative languages, but contains machinery for dealing with the intricacies of the ML type system, such as instantiating type schemes, determining if two types can be unified, and updating the type environment as a result of checking an expression. Hendriks' type-checker computes the most general type for a type-correct Mini-ML expression, and a list of type errors found in the expression.

Figure 19 shows a snapshot of the generated Mini-ML environment in which a small Mini-ML program is checked. The example program contains a type error because the parameter `f` of a lambda expression is applied to arguments of different types: a boolean constant `true`, and a number `3`. Invoking the type-checker on this program yields a type error: "`incorrect application (f 3)`" which appears in a separate window. Pressing the `Slice` button results in the appearance of another window that contains the slice associated with this error message. Observe that several subterms that did not contribute to the error message do not occur in the slice, in

particular, the name of the identifier declared in the outer `let` expression, and the entire inner `let` expression do not occur in the slice.

Hendriks' Mini-ML specification is not written in the abstract interpretation style that we advocate, but in a more traditional style that relies on explicitly traversing syntactic constructs. We conducted the above experiment without making any changes to the specification. While in the case of the above example and many other examples, dependence tracking computes an acceptable slice, in some cases slices are computed that seem larger than necessary. It is our strong conjecture that slice quality can be improved by applying the techniques of Section 4.7.

6. RELATED WORK

6.1 Using Origin Tracking for Computing Positional Information

The work presented in this paper is closely related to earlier work by the same authors. The CLaX type-checker [Dinesh and Tip 1992] 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* [van Deursen et al. 1993] 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. At an informal level, the difference between the two techniques can be summarized as follows: origin tracking determines relationships that reflect how parts of the original term *recur* in the final term, whereas dependence tracking determines which parts of the original term are responsible for the appearance of parts of the final term. Both techniques have their strengths and weaknesses. Origin relations are more precise, but not every term has an origin. Dependence tracking relates each symbol in the final term to a subcontext of the original term, but these relationships may be less precise than origins in cases where many function symbols contribute to the occurrence of a symbol. We discuss these trade-offs in some more detail below.

Similar to dependence tracking, origin tracking relies on the notion of *residuation*, which captures how subterms are moved around, copied, or erased due to the application of rewriting rules. However, unlike dependence tracking, origin tracking does *not* track the creation of function symbols. Instead, relationships are established between subterms that correspond to syntactically equal subterms that occur in both the left-hand side and the right-hand side of a rewriting rule, as well as relationships between the root of the *redex* (the subterm that is being rewritten) and the root of the *reduct* (the subterm that replaces the redex).

The use of origin tracking for associating positional information with type errors was explored in Dinesh [1994; 1996]. 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 typechecker must contain fragments that literally occur in the program source; otherwise, positional information is unavailable. In Dinesh [1994; 1996], 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 nonemptiness of origins. Modification of the specification resulted in adequate positional information for type errors.

By contrast, the approach taken in this paper does not require any modification to specifications in principle. In practice, certain specification styles produce more accurate results than others, and we found that avoiding a small number of common idioms in specifications can significantly improve the results. In Section 4.7, we discussed several of these techniques, including avoiding overspecification and explicit list traversals. It should be emphasized, however, that application of these techniques is completely optional.

Another approach to providing positional information for type errors is pursued by van Deursen [1994a; 1994b]. Van Deursen investigates a restricted class of algebraic specifications called Primitive Recursive Schemes (PRSes). 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 van Deursen et al. [1993] by taking this additional structure into account, which enables the computation of more precise origins.

Fraer [1996] uses a variation on origin tracking [Bertot 1991a; 1991b; 1992] 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.

6.2 Other Applications of Dependence Tracking

The dependence tracking relation we use for obtaining positional information was developed by Field and Tip [1994; 1998] for the purpose of computing program slices. A *program slice* [Weiser 1979; 1984; Tip 1995b] 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 Tip [1995b].

By applying dependence tracking to different rewriting systems, various types of slices can be obtained. In Field et al. [1995] programs are translated to an intermediate graph representation named PIM [Field 1992; Bergstra et al. 1996]. 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 trade-offs to be made. In Tip [1995a], dynamic program slices are obtained by applying dependence tracking to a previously written specification for a CLaX-interpreter.

6.3 Relationship to Inference-Based Type-Checking Approaches

There are some interesting connections between rewriting-based static semantics specifications, and more traditional static semantics specifications based on inference rules (e.g., see Kahn's natural semantics [Kahn 1987]). Both approaches rely on the notion of a type-environment, which maps each identifier to its type, and construct an initial type environment by scanning the program's declarations. An important difference between the two frameworks is the fact that rewriting-based approaches encode the type-environment directly into the term structure, whereas inference rule based approaches represent the environment as a separate entity. Another difference is that while conditional rewriting rules and conditional inference rules seem similar, proving conditions of inference rules generally involves an exhaustive search that may involve backtracking, whereas conditional rewriting rules only involve normalization followed by a check for syntactic (in)equality without backtracking.

In terms of expressive power, there appears to be little difference between the two approaches. Although significantly more work has been done on type-checking complex type systems such as that of ML in the inference-based setting (e.g., see Damas and Milner [1982], Kahn [1987], and Cardelli [1987]), Hendriks [Bergstra et al. 1989; Hendriks 1991] created an ASF+SDF specification of the static semantics of Mini-ML [Clément et al. 1986], an ML subset that contains the essential elements of ML's type system as far as type-checking is concerned, including generic type variables, function types and cartesian product types.

In Section 5.4, we presented some experiments with an existing type-checker for an ML subset. Providing accurate positional information for inference errors in ML has long been known to be a difficult problem. Several proposals that rely on adapting or extending the underlying type system or inference algorithm have been presented (e.g., see Bernstein and Stark [1995], Wand [1986], and Johnson and Walz [1986]). Duggan and Bent [1996] presented an approach for explaining the types inferred by a type inference algorithm (in the context of ML and Haskell) that relies on an adaptation of the unification algorithm to keep track of the individual reasoning steps that are applied. While the work by Duggan and Bent is similar in spirit to ours in the sense that information is inferred from the applied reasoning steps, it is different in the sense that the problem of finding source locations of type *errors* is not addressed directly. Moreover,

our approach is to present error locations by highlighting areas in the program's source text, whereas Duggan and Bent only provide explanations why a particular type is inferred for a designated variable.

Fraer [1997] adapted dependence tracking to natural semantics. Fraer's definitions mirror those of Field and Tip [1994; 1998], but his implementation approach is quite different. Instead of implementing dependence tracking directly in the inference engine, inference rules are *instrumented* with additional arguments that store dependence information. This instrumentation is performed according to *instrumentation schemas* that analyze the syntactic structure of the inference rule to determine how dependence information should be combined and propagated. An important advantage of a transformational approach is that it does not require a specialized inference engine, and that the implementation of dependence tracking benefits directly from any improvements to the underlying engine. While Fraer's work is mostly concerned with program verification and determining the origin of verification errors, his dependence tracking implementation is completely application-independent, and can in principle be applied to type-checking problems such as the ones we consider.

6.4 Miscellaneous

The slice notion presented in the current paper differs from the traditional program slices [Weiser 1979; Tip 1995b] 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 notion of a slice in the present paper 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.

Heering [1996] 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 explicit traversal of syntactic structures, which adversely affects slice accuracy. However, this would require the extension of dependence tracking to higher-order rewriting systems.

Flanagan et al. [1996] developed MrSpidey, an interactive debugger for Scheme that 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.

7. 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 [Klop 1992]. In this model, a type-check function rewrites a program's abstract syntax tree to a list of type errors. Dependence tracking [Field and Tip 1994; 1998] is used to associate a *slice* [Weiser 1979; Tip 1995b] of the program with each error message. Unlike previous approaches for automatic determination of error locations [Dinesh 1994; 1996; Dinesh and Tip 1992; van Deursen 1994a; 1994b; Bertot 1991a; 1991b; 1992], ours does not rely on a specific specification style, nor does it require additional specification-level information for tracking locations. The computed slices have the interesting semantic property that 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 [Klint 1993; van Deursen et al. 1996], and conducted experiments with a significant subset of Pascal, with features of object-oriented type systems such as subtyping, and with an existing type-checker specification for a subset of ML. The positional information conveyed by the computed slices is generally quite good, but depends on the style in which the specification is written. We have explored the effect of different specification styles in depth, and identified a number of specification idioms that result in a loss of slice accuracy. For each of these idioms, we have identified alternative specification approaches that yield more accurate slices. Interestingly, these changes often result in more declarative specifications that are more concise and easier to read.

There are a number of directions in which our work can be extended. It would be interesting to rewrite the Mini-ML type-checker discussed in Section 5.4 in an abstract interpretation style, and to conduct experiments to see how slice quality is affected. We would also like to apply our techniques to a complete object-oriented language such as Java [Gosling et al. 1996].

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for many helpful suggestions.

REFERENCES

ALT, M., ASSMANN, U., AND VAN SOMEREN, H. 1994. Cosy compiler phase embedding with the cosy compiler model. In *Compiler Construction '94*, P. A. Fritzson, Ed. Springer-Verlag, New York, NY, 278–293.

BERGSTRA, J A AND KLOP, J W. 1986. Conditional rewrite rules: Confluence and termination. *J. Comput. Syst. Sci. 32*, 3 (June), 323–362.

BERGSTRA, J., DINESH, T., FIELD, J., AND HEERING, J. 1996. A complete transformational toolkit for compilers. In *Proceedings of the European Symposium on Programming* (Linköping, Sweden, Apr.). Springer-Verlag, New York, NY.

BERGSTRA, J. A. 1989. *Algebraic Specification*. ACM Press Frontier Series. ACM Press, New York, NY.

BERNSTEIN, K. L. AND STARK, E. W. 1995. Debugging type errors. State University of New York at Stony Brook, Stony Brook, NY.

BERTOT, Y. 1991a. Occurrences in debugger specifications. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (SIGPLAN '91, Toronto, Ontario, Canada, June 26–28), D. S. Wise, Chair. ACM Press, New York, NY, 327–337.

BERTOT, Y. 1991b. Une automatisation du calcul des résidus en sémantique naturelle. Ph.D. Dissertation.

BERTOT, Y. 1992. Origin functions in lambda-calculus and term rewriting systems. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming* (CAAP '92), J.-C. Raoult, Ed. Springer-Verlag, New York, NY.

CARDELLI, L. 1987. Basic polymorphic typechecking. *Sci. Comput. Program. 8*, 2 (Apr. 1), 147–172.

CLÉMENT, D., DESPEYROUX, J., DESPEYROUX, T., AND KAHN, G. 1986. A simple applicative language: Mini-ml. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (Cambridge, MA, Aug.). 13–27.

DAMAS, L. AND MILNER, R. 1982. Principal type schemes for functional programs. In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages* (Albuquerque, NM). 207–212.

DINESH, T. B. 1994. Type checking revisited: Modular error handling. In *Semantics of Specification Languages, Workshops in Computing*, D. J. Andrews, J. F. Groote, and C. A. Middelburg, Eds. Springer-Verlag, Berlin, Germany, 216–231.

DINESH, T. B. 1996. Typechecking with modular error handling. In *Language Prototyping: An Algebraic Specification Approach*, A. von Duersen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co, Inc., Singapore, 85–104.

DINESH, T. B. AND TIP, F. 1992. Animators and error reporters for generated programming environments. Rep. CS-R9253. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

DINESH, T. B. AND TIP, F. 1997a. A case-study of a slicing-based approach for locating type errors. In *Proceedings of the 2nd International Workshop on Theory and Practice of Algebraic Specifications* (ASF+SDF'97, Amsterdam, The Netherlands, Sept.).

DINESH, T. B. AND TIP, F. 1997b. A slicing-based approach for locating type errors. In *Proceedings of 1st Usenix Conference on Domain-Specific Languages* (DSL'97, Santa Barbara, CA, Oct.). USENIX Assoc., Berkeley, CA, 77–88.

DUGGAN, D. AND BENT, F. 1996. Explaining type inference. *Sci. Comput. Program. 27*, 1, 37–83.

FIELD, J. 1992. 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*. 98–107.

FIELD, J. AND TIP, F. 1994. 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*, M. Hermenegildo and J. Penjam, Eds. Springer-Verlag, New York, NY, 415–431.

FIELD, J. AND TIP, F. 1998. Dynamic dependence in term rewriting systems and its application to program slicing. *Inf. Softw. Technol. 40*, 609–636.

FIELD, J., RAMALINGAM, G., AND TIP, F. 1995. Parametric program slicing. In *Papers of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '95, San Francisco, CA, Jan. 22–25), R. K. Cytron and P. Lee, Chairs. ACM Press, New York, NY, 379–392.

FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., WEIRICH, S., AND FELLEISEN, M. 1996. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation* (PLDI '96, Philadelphia, PA, May 21–24), C. N. Fischer, Chair. ACM Press, New York, NY, 23–32.

FOKKINK, W., KAMPERMAN, J., AND WALTERS, P. 1998. Within ARM's reach: compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Trans. Program. Lang. Syst. 20*, 3, 679–706.

FRAER, R. 1996. Tracing the origins of verification conditions. In *Proceedings of the AMAST '96 Conferenence* (AMAST'96, Munich, Germany, July). Springer-Verlag, New York, NY.

FRAER, R. 1997. Analyse de programmes annotés par des assertions. Ph.D. Dissertation.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, MA.

HEERING, J. 1996. Second-order term rewriting specification of static semantics. In *Language Prototyping: An Algebraic Specification Approach*, A. von Duersen, J. Heering, and P. Klint, Eds. World Scientific Publishing Co, Inc., Singapore, 295–306.

HEERING, J., HENDRIKS, P. R. H., KLINT, P., AND REKERS, J. 1989. The syntax definition formalism SDF—reference manual. *SIGPLAN Not. 24*, 11 (Nov.), 43–75.

HENDRIKS, P. 1991. Implementation of modular algebraic specifications. Ph.D. Dissertation. Dept. of Computer Science, Univ. of Amsterdam, The Netherlands.

HUDAK, P., PEYTON JONES, S., WADLER, P., BOUTEL, B., FAIRBAIRN, J., FASEL, J., GUZMÁN, M. M., HAMMOND, K., HUGHES, J., JOHNSSON, T., KIEBURTZ, D., NIKHIL, R., PARTAIN, W., AND PETERSON, J. 1992. Report on the programming language Haskell: A non-strict, purely functional language, version 1.2. *SIGPLAN Not. 27*, 5 (May), 1–164.

JOHNSON, G. F. AND WALZ, J. A. 1986. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th ACM Conference on Principles of Programming Languages* (POPL '86, St. Petersburg, FL, Jan.). ACM, New York, NY, 44–57.

KAHN, G. 1987. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87* (Passau, Federal Republic of Germany, Feb. 19-21), F. J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds. Springer-Verlag, Berlin, Germany, 22–39.

KAMPERMAN, J. 1996. Compilation of term rewriting systems. Ph.D. Dissertation. Dept. of Computer Science, Univ. of Amsterdam, The Netherlands.

KAMPERMAN, J. AND WALTERS, H. 1996. Minimal term rewriting systems. In *Proceedings of the 11th Workshop on Recent Trends in Data Type Specification: Specification of Abstract Data Types Joint With the 8th COMPASS Workshop* (Oslo, Norway, Sept. 19-23, 1995). Springer-Verlag, New York, NY, 274–290.

KLINT, P. 1993. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol. 2*, 2 (Apr.), 176–201.

KLOP, J. W. 1992. Term rewriting systems. In *Handbook of Logic in Computer Science-Background: Computational Structures*, S. Abramsky, D. M. Gabbay, and S. E. Maibaum, Eds. Osborne Handbooks of Logic in Computer Science, vol. 2. Oxford University Press, Inc., New York, NY, 1–116.

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

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML-Revised*. MIT Press, Cambridge, MA.

PLASMEIJER, M. J. AND VAN EEKELEN, M. C. J. D. 1994. Concurrent Clean 1.0 Language Manual, draft version. Katholieke Universiteit Nijmegen, Nijmegen, The Netherlands.

STROUSTRUP, B. 1997. *The C++ Programming Language*. 3rd ed. Addison-Wesley, Reading, MA.

THE COMPARE CONSORTIUM. 1991. Description of the cosy-prototype.

TIP, F. 1995a. 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), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds. Springer-Verlag, New York, NY, 516–530.

TIP, F. 1995b. A survey of program slicing techniques. *J. Program. Lang. 3*, 3, 121–189.

VAN DEN BRAND, M., KLINT, P., AND OLIVIER, P. 1999. Compilation and memory management for asf+sdf. In *Proceedings of the Eighth International Conference on Compiler Construction*

(CC'99, Amsterdam, The Netherlands), S. Jähnichen, Ed. Springer-Verlag, New York, NY, 198–213.

VAN DEURSEN, A. 1994. Executable language definitions—Case studies and origin tracking techniques. Ph.D. Dissertation. Dept. of Computer Science, Univ. of Amsterdam, The Netherlands.

VAN DEURSEN, A. 1994. Origin tracking in primitive recursive schemes. Rep. CS-R9401. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

VAN DEURSEN, A., KLINT, P., AND TIP, F. 1993. Origin tracking. *J. Symb. Comput. 15*, 523–545.

VON DUERSEN, A., HEERING, J., AND KLINT, P., EDS. 1996. *Language Prototyping: An Algebraic Specification Approach*. World Scientific Publishing Co, Inc., Singapore.

WALTERS, H. R. AND KAMPERMAN, J. F. T. 1996. Epic: An equational language–abstract machine and supporting tools. In *Proceedings of the Seventh International Conference on Rewriting Techniques and Applications* (RTA'96, New Brunswick, NJ, July). Springer-Verlag, New York, NY, 424–427.

WAND, M. 1986. Finding the source of type errors. In *Proceedings of the 13th ACM Conference on Principles of Programming Languages* (POPL '86, St. Petersburg, FL, Jan.). ACM, New York, NY, 38–43.

WEISER, M. 1979. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. Dissertation. University of Michigan, Ann Arbor, MI.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10*, 4, 352–357.

Received: September 1998; revised: February 1999 and May 2000; accepted: August 2000