Generic Techniques for Source-Level Debugging and Dynamic Program Slicing

Frank Tip*

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
tip@cwi.nl

Abstract. Algebraic specifications have been used successfully as a formal basis for software development. This paper discusses how the *origin* and *dynamic dependence* relations implicitly defined by an algebraic specification can be used to define powerful language-specific tools. In particular, the generation of tools for source-level debugging and dynamic program slicing from specifications of interpreters will be addressed.

1 Introduction

Algebraic specifications [4] have been used successfully for the generation of a variety of software development tools, such as typecheckers [9], interpreters [10], and program analysis tools [11, 12, 21]. The present paper discusses how two previously developed language-independent techniques, origin tracking [8] and dynamic dependence tracking [13], can be used to derive powerful language-specific debugging tools from algebraic specifications of interpreters. In particular, we show that—in addition to "standard" debugger features such as single-stepping, state inspection, and breakpoints—a variation of dynamic program slicing [1, 17] can be defined with surprisingly little effort. The main contribution of this paper is to show that the information required to construct such debugging tools is to a very large extent *language-independent* and *implicitly* present in a language's specification.

We assume that specifications are executed by conditional term rewriting [16]. Specifically, an algebraic specification of an interpreter expresses the execution of a program as the rewriting of a term consisting of a function `execute` applied to the abstract syntax tree (AST) of that program. Rewriting this term produces a sequence of terms that effectively represent the consecutive internal states of the interpreter. Origin tracking is a method for tracing occurrences of the *same* subterm in a sequence of terms, and is used for the definition of single-stepping and breakpoints. Dependence tracking establishes certain minimal dependence relations between terms in a rewriting sequence, and is used to obtain dynamic slices.

2 Specification of an interpreter

We illustrate our ideas by way of a simple language **L** that features assignments, **if** statements, **while** statements, and statement sequences. **L**-expressions are constructed

* Supported in part by the European Union under ESPRIT project # 5399 (Compiler Generation for Parallel Machines—COMPARE).

from constants, variables, arithmetic operators '+', '-', and '*', and the equality test operator '='. Fig. 1 shows an algebraic specification of an **L**-interpreter. The execution of an **L**-program P corresponds to the *rewriting* of the *term* `execute`(t_P) according to this specification, where t_P is the term that constitutes the AST of P; this will ultimately result in a (term that represents a) list containing the final value of each variable.

/* top-level function for execution of programs */

[L1] `execute`(**declare** *DeclSeq* **begin** *StatSeq* **end**) = `exec`(*StatSeq*, `create`(*DeclSeq*, ϵ_e))

/* functions for creation and manipulation of environments */

[L2] `create`(ϵ_d, *Env*) = *Env*
[L3] `create`(*Var*;*DeclSeq*, *Env*) = `create`(*DeclSeq*, *Var* $\mapsto$ `0`; *Env*)
[L4] `lookup`(*Var* $\mapsto$ *Constant*;*Env*, *Var*) = *Constant*
[L5] `lookup`(*Var* $\mapsto$ *Constant*;*Env*, *Var*$'$) = `lookup`(*Env*, *Var*$'$) **when** *Var* $\neq$ *Var*$'$
[L6] `update`(*Var* $\mapsto$ *Constant*;*Env*, *Var*, *Constant*$'$) = *Var* $\mapsto$ *Constant*$'$;*Env*
[L7] `update`(*Var* $\mapsto$ *Constant*;*Env*, *Var*$'$, *Constant*$'$) = *Var* $\mapsto$ *Constant*; `update`(*Env*, *Var*$'$, *Constant*$'$) **when** *Var* $\neq$ *Var*$'$

/* evaluation of expressions */

[L8] `eval`(*Constant*, *Env*) = *Constant*
[L9] `eval`(*Var*, *Env*) = `lookup`(*Var*, *Env*)
[L10] `eval`((*Exp* + *Exp*$'$), *Env*) = `intadd`(`eval`(*Exp*, *Env*), `eval`(*Exp*$'$, *Env*))
[L11] `eval`((*Exp* - *Exp*$'$), *Env*) = `intsub`(`eval`(*Exp*, *Env*), `eval`(*Exp*$'$, *Env*))
[L12] `eval`((*Exp* * *Exp*$'$), *Env*) = `intmul`(`eval`(*Exp*, *Env*), `eval`(*Exp*$'$, *Env*))
[L13] `eval`((*Exp* = *Exp*$'$), *Env*) = `inteq`(`eval`(*Exp*, *Env*), `eval`(*Exp*$'$, *Env*))

/* execution of (lists of) statements */

[L14] `exec`(ϵ_s, *Env*) = *Env*
[L15] `exec`(*Var* := *Exp*;*StatSeq*, *Env*) = `exec`(*StatSeq*, `update`(*Env*, *Var*, `eval`(*Exp*, *Env*)))
[L16] `exec`(**if** *Exp* **then** *StatSeq* **else** *StatSeq*$'$ **end**;*StatSeq*$''$, *Env*) = `exec`(*StatSeq*$''$, `exec`(*StatSeq*, *Env*))
when `eval`(*Exp*, *Env*) $\neq$ `0`
[L17] `exec`(**if** *Exp* **then** *StatSeq* **else** *StatSeq*$'$ **end**;*StatSeq*$''$, *Env*) = `exec`(*StatSeq*$''$, `exec`(*StatSeq*$'$, *Env*))
when `eval`(*Exp*, *Env*) = `0`
[L18] `exec`(**while** *Exp* **do** *StatSeq* **end**; *StatSeq*$'$, *Env*) = `exec`(**while** *Exp* **do** *StatSeq* **end**; *StatSeq*$'$, `exec`(*StatSeq*, *Env*))
when `eval`(*Exp*, *Env*) $\neq$ `0`
[L19] `exec`(**while** *Exp* **do** *StatSeq* **end**; *StatSeq*$'$, *Env*) = `exec`(*StatSeq*$'$, *Env*)
when `eval`(*Exp*, *Env*) = `0`

Fig. 1. Algebraic specification of an **L**-interpreter.

Term rewriting is a cyclic process where each cycle involves determining a subterm t and a rule $l = r$ such that t and l match. This is the case if a substitution σ can be found that maps every variable X in l to a term $\sigma(X)$ such that $t \equiv \sigma(l)$ (σ distributes over function symbols). For rewrite rules without conditions, the cycle is completed by replacing t by the instantiated right-hand side $\sigma(r)$. A term for which no rule is applicable to any of its subterms is called a *normal form*; the process of rewriting a term to its normal form (if it exists) is referred to as *normalizing*. A conditional rewrite rule (e.g., **[L16]**) 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 conditions (of the form $t_1 = t_2$) succeed iff the resulting normal forms are syntactically equal, negative conditions ($t_1 \neq t_2$) if they are syntactically different.

The specification of Fig. 1 is based on the manipulation of an environment, i.e., a list containing the current value of each variable. **[L1]** defines the top-level function `execute` in terms of two other functions, `create` and `exec`. The former, `create`, uses the declarations of the program to create an initial environment, where each variable has the value `0` (**[L2]** and **[L3]**)[2]. The latter, `exec`, specifies the execution of a list of statements; it "uses" the functions `lookup` (**[L4]**–**[L5]**) for retrieving a value from an environment, and `update` (**[L6]**–**[L7]**) for updating the value of a variable in an environment. Rules **[L8]**–**[L13]** define a recursive function `eval` for evaluating **L**-expressions. The specification of the primitive operations `intadd`, `intsub`, `intmul`, and `inteq` is omitted here. **[L14]** states that executing the empty list of statements does not affect the environment. In **[L15]**–**[L19]**, the cases are specified where the statement list is non-empty. **[L15]** defines the execution of an assignment in terms of the evaluation of its right-hand side expression, and an update of the environment. In **[L16]**–**[L17]** the execution of a non-empty statement list beginning with an **if**–**then**–**else** construct is defined by conditional rules; **[L16]** and **[L17]** correspond to situations where the control predicate evaluates to any non-zero value and zero, respectively. The execution of a **while** statement is specified in a similar way (**[L18]**–**[L19]**).

Fig. 2 **(a)** shows an example **L**-program. By applying the equations of Fig. 1, the environment of Fig. 2 **(b)** is produced.

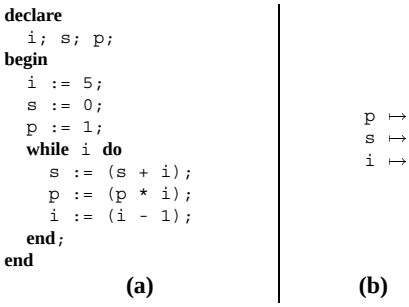


Fig. 2. **(a)** Example **L**-program. **(b)** Environment obtained by executing the program of **(a)** according to the specification of Fig. 1.

3 Basic techniques

In this section, we will briefly present origin tracking and dynamic dependence tracking. Due to space limitations, only an informal description of these techniques is presented here; the reader is referred to [8, 13, 21] for formal definitions.

3.1 Origin tracking

In the discussion below, it is assumed that a term S is rewritten to a term T in zero or more steps: $S \rightarrow^* T$. In [8, 21], the origin relation is formally defined as a relation

[2] This specification assumes that every variable is properly declared.

between subterms of S and subterms of T; associated with every subterm T' of T is a set of subterms, *OriginOf*(T'), of the initial term S—the *origin of* T'. The principal properties of the origin relation are that: (i) relations involve *equal* terms (in the sense of rewriting): for each subterm $S' \in$ *OriginOf*(T') we have that $S' \rightarrow^* T'$, and that (ii) relations are defined in an inductive manner. For a reduction of length zero, the origin relation is the identity relation; for a multi-step reduction $S \rightarrow^* T \xrightarrow{r} U$, the origin of a subterm U' of U is defined in terms of the origins of subterms of T, and the structure of the applied rule, r.

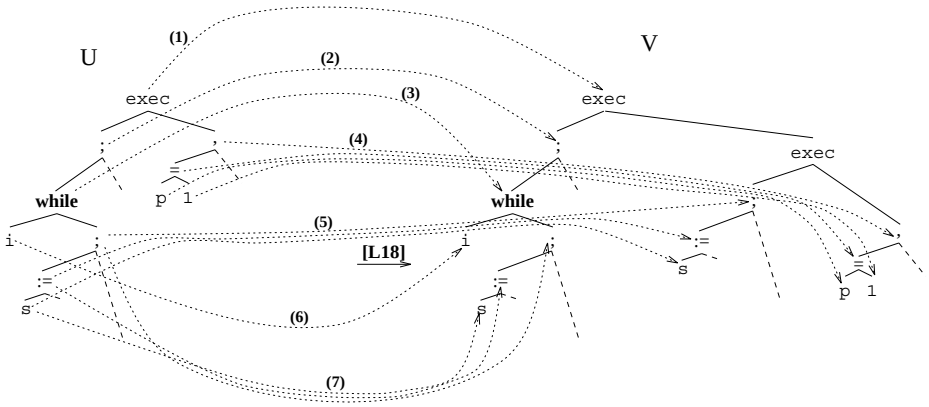


Fig. 3. Origin relations.

As an example, Fig. 3 depicts an application of **[L18]** to a **while**-term. Dotted lines in the figure indicate origin relations. The relation labeled **(1)** is the relation between the roots of U and V—such a relation is always present. Variables that occur in both the left-hand side and the right-hand side of **[L18]** cause more origin relations to appear—variable *Exp* gives rise to the relation labeled **(6)**, variable *StatSeq* to the sets of relations labeled **(5)** and **(7)**, and variable *Env* to the relations labeled **(4)**. The relation labeled **(3)** is caused by the occurrence of a *subterm* **while** *Exp* **do** *StatSeq* **end** in both the left-hand side and the right-hand side of **[L18]**. Relation **(2)** is also caused by a common subterm.

Note that the rightmost `exec` function symbol in term V is not related to any symbol in U—its origin is the empty set. In general, a term will have a non-empty origin if it was derived directly from a subterm of the initial term (here: the program's AST). In [8, 21], a number of sufficient constraints on specifications is stated that guarantee that origin sets of subterms with a specific root function symbol, or of a specific sort, contain at least one, or exactly one element. The specification of Fig. 1 satisfies the constraints necessary to guarantee that each "statement" subterm will have an origin set containing exactly one element. For specifications that do not conform to these constraints, the origin relation of [7, Chapter 7] may be used, which is applicable to any specification of a compositional nature.

3.2 Dynamic dependence tracking

Consider the following simple rules for integer arithmetic:

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

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

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

By carefully studying this example reduction, we can make the following observations:

- The outer context `intsub(3, •)` of T_0 ('•' denotes a missing subterm) is not affected at all, and therefore reappears in T_1 and T_2.
- The occurrence of variables *X*, *Y*, and *Z* in both the left-hand side and the right-hand side of **[A2]** causes the respective subterms `0`, `1`, and `2` of the underlined subterm of T_0 to reappear in T_1.
- Variable *X* only occurs in the left-hand side of **[A1]**. Consequently, the subterm (of T_1) `intmul(1, 2)` 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 [13, 21]. Notions of *creation* and *residuation* are defined for single rewrite-steps. The former involves function symbols produced by rewrite rules whereas the latter corresponds to situations where symbols are copied, erased, or not affected by rewrite rules[3]. Fig. 4 shows all residuation and creation relations for the example reduction discussed above.

Roughly speaking, the dynamic dependence relation for a multi-step reduction ρ consists of the transitive closure of creation and residuation relations for the individual rewrite steps in ρ. In [13, 21], the dynamic dependence relation is defined as a relation on *contexts*, i.e., connected sets of function symbols in a term. The fact that C is a *subcontext* of a term T is denoted $C \sqsubseteq T$. For any reduction ρ that transforms a term T into a term T', a *term slice* with respect to some $C' \sqsubseteq T'$ is defined as the subcontext $C \sqsubseteq T$ that is found by tracing back the dynamic dependence relations from C'. The term slice C satisfies the following properties:

1. C can be rewritten to a term $D' \sqsupseteq C'$ via a reduction ρ', and
2. ρ' is a subreduction of the original reduction ρ. Intuitively, ρ' contains a subset of the rule applications in ρ.

[3] The notions of creation and residuation become more complicated in the presence of so-called *left-nonlinear* rules and *collapse rules*. This is discussed at greater length in [13, 21].

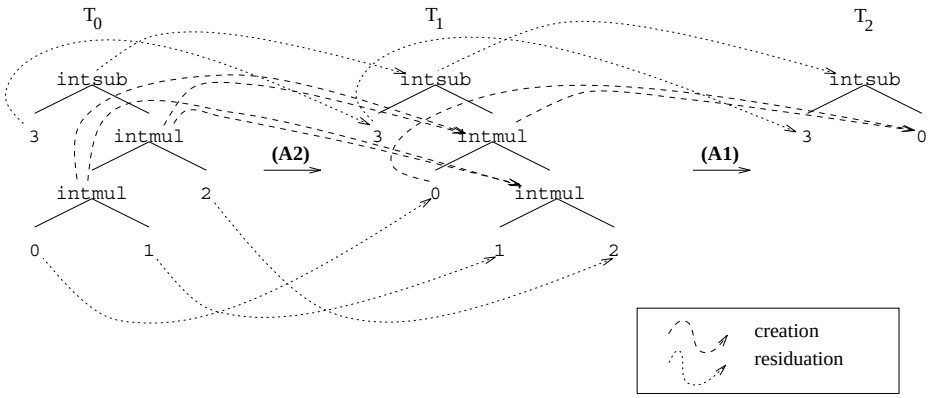


Fig. 4. Example of creation and residuation relations.

In cases where no confusion arises, we will write $C = \mathit{SliceOf}(C')$ to indicate that C is the term slice with respect to C' for some reduction $\rho : T \rightarrow^* T'$, $C \sqsubseteq T$, and $C' \sqsubseteq T'$.

Returning to the example, we can determine the term slice with respect to the entire term T_2 by tracing back all creation and residuation relations to T_0; the reader may verify that *SliceOf*(`intsub(3, 0)`) = `intsub(3, intmul(intmul(0, •), •))`.

3.3 Application to specifications of interpreters

Fig. 5 depicts some of the relations established by origin tracking and dynamic dependence tracking as a result of executing the program of Fig. 2. The figure shows the initial term S, the final term T and an intermediate term U that occur in the process of executing the program according to the specification of Fig. 1. The intermediate term U corresponds to the situation where the **while** loop is entered for the first time.

Subterms of U and S that are related by the origin relation are indicated by dotted lines in Fig. 5. Also shown in Fig. 5 is a subcontext S' of S that is related to the subterm U' of U via the dynamic dependence relation. Observe that S' *excludes* the right-hand sides of two of the assignment statements in the program. One of the key properties of the dynamic dependence relation is that replacing these right-hand sides by *any* **L**-expression will yield a term that can be rewritten (via a subreduction of `r`) to a term that contains a subcontext $\mathtt{p} \mapsto \mathtt{1}$.

Although origin and dependence relations are computed in a similar manner, using similar information as input, the nature of these relations is different. This is mainly due to the fact that these relations were designed with different objectives in mind. Origin information always involves *equal* terms. In the example of Fig. 5, origin track-

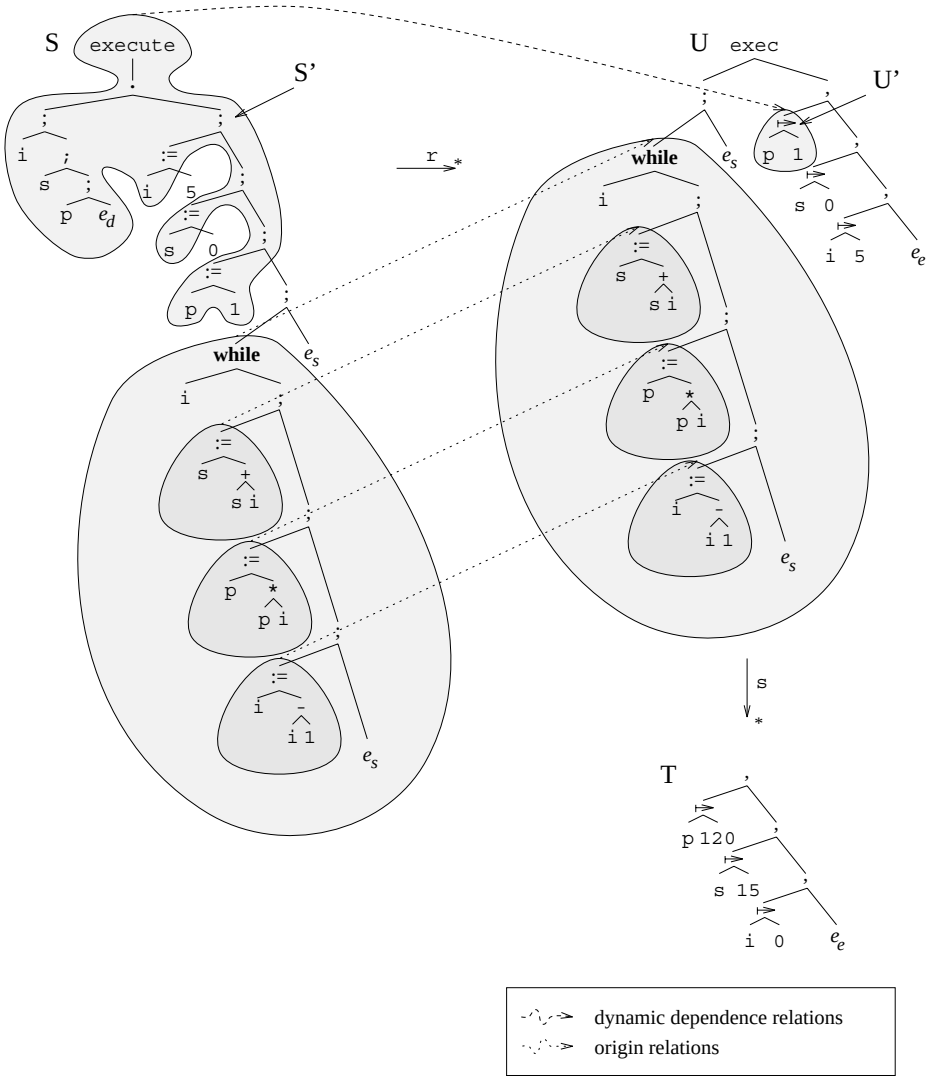


Fig. 5. Illustration of origin and dynamic dependence relations.

ing establishes relations between a number of syntactically[4] equal terms; in this case corresponding to the statements of the program. Equality (via convertibility of terms) also plays an important role in the notion of dependence tracking. Dynamic dependence

[4] For the purpose of debugging, origin relations rarely involve terms that are not syntactically equal. Examples of origin relations involving terms not syntactically equal are mainly to be found in the area of error-reporting [10, 9, 7].

relations are in principle defined for any subcontext of any term that occurs in a rewriting process: associated with a subcontext s is the minimal subcontext of the initial term that was necessary for "creating" a term that contains s. In the sequel, we are primarily interested in the dynamic dependence relations for subcontexts that represent values computed by a program (such as the subterm U' in Fig. 5). It will be shown below that for these subcontexts, the dynamic dependence relation will compute information that is similar to the notion of a dynamic program slice [1, 17].

3.4 Implementation

Origin tracking and dynamic dependence tracking have been implemented in the rewrite engine of the ASF+SDF Meta-environment [15]. All function symbols of all terms that arise in a rewriting process are *annotated* with their associated origin and dependence information; this information is efficiently represented by way of bit-vectors. Whenever a rewrite rule is applied to a term t, and a new term t' is created, origin and dependence information is *propagated* from t to t'. These propagations are expressed in terms of operations on sets. In [21], it is argued that the cost of performing these propagation steps is at worst linear in the size of the initial term of the reduction.

4 Definition of debugger features

Below, we describe how a number of debugger features can be defined using the techniques of the previous section. We will primarily concentrate on the mechanisms needed for *defining* debugger features, and ignore issues related to a debugger's user-interface.

4.1 Single stepping/visualization

Step-wise execution of a program at the source code level is the basic feature of any debugger.

Observe that in the specification of Fig. 1, the execution of a statement corresponds to the rewriting of a term of the following form:

`exec`(*Stat*;*StatList*, *Env*)

where *Stat* represents any statement, *StatList* any list of statements, and *Env* any environment. Consequently, the fact that *some* statement is executed can be detected by matching the above *pattern* against the current redex[5].

Origin tracking can be used to determine *which* statement is currently being executed. We assume that the rewriting process is suspended whenever a redex T *matches* the above pattern. At this point, the subterm T' of T that is matched against variable *Stat* is determined. The origin of T', *OriginOf*(T'), will consist of the subtree of the program's AST that represents the currently executed statement. Thus, program execution can be visualized at the source level by highlighting this subterm of the AST.

[5] We will use the term "redex" (short for reducible expression) to denote the subterm that has been matched against some equation. For conditional rules, it is assumed that no conditions have been evaluated yet.

4.2 Breakpoints

Another standard source-level debugger feature is the *breakpoint*. The idea is that the user selects a statement s in the program, and execution continues until s is reached.

A breakpoint on a statement s can be implemented as follows. Let T_s be the subterm of the AST that corresponds to s. Then the rewriting process should be suspended when: (i) a redex T matches the pattern `exec`(*Stat*;*StatList*, *Env*) (indicating that *some* statement is being executed), and (ii) $T_s \in$ *OriginOf*(T'), where T' is the subterm of T matched against variable *Stat*.

4.3 State inspection

At any moment that execution is suspended, either while single-stepping or due to a breakpoint, one may wish to inspect the values of variables or, more generally, arbitrary source-level expressions.

State inspection may be implemented as follows. We assume that execution was suspended at the moment that some statement was executed, i.e., a redex T matches the pattern `exec`(*Stat*;*StatList*, *Env*) Let T_z be the subterm of T that was matched against variable *Env*. Then an arbitrary source-level expression e (with an AST T_e) can be evaluated by *rewriting* the term `eval`(T_e, T_z) according to the specification of Fig. 1. The result of this rewriting process will be a term representing the "current" value of expression e.

4.4 Watchpoints

Watchpoints [18] are a generalization of breakpoints. The user supplies a source-level expression e (with AST T_e), and execution continues until the value of e changes.

A watchpoint may be implemented as follows. First, an initial value u (with AST T_u) of expression e is computed (using the technique of Sec. 4.3) and stored by the debugger. Whenever a statement is executed, the current value v (with AST T_v) of e is determined and is compared with u by rewriting a term `inteq`(T_u, T_v). Execution (i.e., the rewriting process) is suspended when this test fails (i.e., yields the value zero).

4.5 Data breakpoints

A *data breakpoint* [22] is yet another variation on the breakpoint theme. A data breakpoint on a variable v (with AST T_v) is effective when v is referenced (or modified).

Data breakpoints can be implemented by suspending the rewriting process when a redex matches the pattern `lookup`($T_v \mapsto$ *Constant*;*Env*, T_v) (for a data breakpoint on a reference to v), or `update`($T_v \mapsto$ *Constant*;*Env*, T_v, *Constant*′) (for a data breakpoint on an update to v).

4.6 Call stack inspection

In the presence of procedures, the notion of an "environment" needs to be generalized to a stack of activation records, where each record contains the values of the local variables

and parameters for a procedure call. Call-stack inspection can be defined in way that is similar to the techniques of Sec. 4.3, by visualizing the procedure calls in each record. One can easily imagine a tool that allows interactive traversal of the stack of activation records, and enables one to inspect the values of arbitrary source-level expressions in each scope.

5 Dynamic program slicing

Myriad variations on the notion of a *dynamic program slice* [1, 17] can be found in the literature [20]. For the purposes of this paper, we define a *dynamic slice with respect to the current value of a variable* v to be the parts of the program that are necessary for obtaining the current value of v. To see why dynamic slicing is useful for debugging, consider a situation where an incorrect value is computed for v—only the statements in the dynamic slice with respect to v had an effect on the value of v. This allows one to ignore many statements in the process of localizing a bug[6].

Below we pursue a two-phase approach for computing dynamic slices. Sec. 5.1 discusses the nature of the "raw" information provided by the dynamic dependence relation of Sec. 3.2. In Sec. 5.2, we present an heuristic approach for post-processing this information, in order to obtain dynamic slices similar to those of [1, 17].

5.1 Pure term slices

We assume that execution is suspended at a moment that some statement was executed, i.e., a redex T matches the pattern `exec(`*Stat*`;`*StatList*`,`*Env*`)`. Let T_z be the subterm of T matched against *Env*, and let T_p be the subterm of T_z that constitutes the variable-value pair for variable x. Then, the dynamic dependence relation of Sec. 3.2 will associate with T_p a minimal set of function symbols, *SliceOf*(T_p), in the program's AST.

Fig. 6 **(a)** shows a (textual representation of) the term slice that is determined for the final value of variable `p` as obtained by executing the example program of Fig. 2. Observe that the two holes in this term slice can be replaced by *any* **L**-expression without affecting the computation of the value `120` for variable `p`.

One may wonder why the assignments to variable `s` are not completely omitted in the term slice of Fig. 6 **(a)**. This is best understood by keeping in mind that *any* hole in a term slice may be replaced by *any* syntactically valid **L**-term. Note that the assignments to `s` cannot be replaced by any other assignment; e.g., they can certainly not be replaced by any assignment to `p`.

5.2 Post-processing of term slices

While term slices provide information that is semantically sound, they may contain a certain amount of "clutter", in the form of uninteresting information. An example of such information are the two partial assignments to `s` in the term slice of Fig. 6 **(a)**.

[6] Even in cases where a statement is missing inadvertently, dynamic slices may provide useful information. In such a case, it is likely that more statements show up in the slice than one would expect.

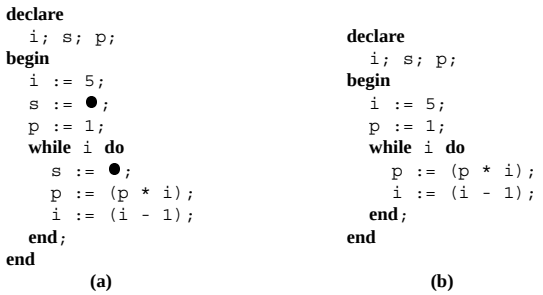


Fig. 6. **(a)** Term slice with respect to the final value of `p`. **(b)** Post-processed slice with respect to the final value of `p`.

In order obtain dynamic slices similar to those in [1, 17], one may *post-process* term slices by: (i) transforming any statement whose right-hand side is irrelevant into an irrelevant statement (rule **[P1]**), and (ii) removing irrelevant statements from statement lists (rule **[P2]**). A specification of this post-processing is shown in Fig. 7. Rewriting the term slice of Fig. 6 **(a)** according to this specification yields the slice of Fig. 6 **(b)**.

[P1] *Var* := ● = ●
[P2] ● ; *StatSeq* = *StatSeq*

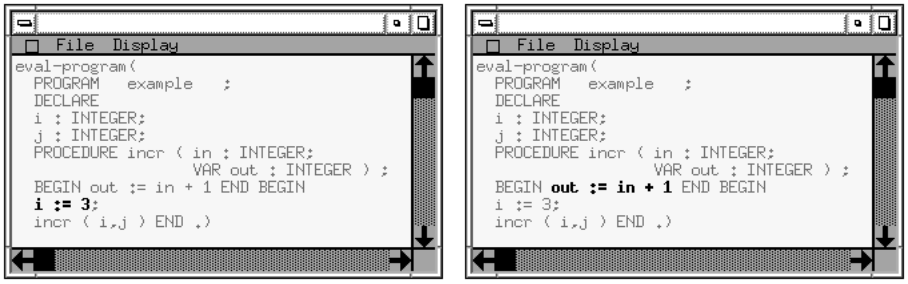
Fig. 7. Specification for post-processing of term slices.

The specification of Fig. 7 is minimal—it only removes irrelevant assignments. In practice, one would like more sophisticated post-processing that, for example, removes all irrelevant declarations from the program. Post-processing becomes nontrivial in the presence of procedures, where situations may occur in which different parameters are omitted at different call sites.

6 Practical experience

To a large extent, the ideas in this paper have been implemented using the ASF+SDF Meta-environment [15], a programming environment generator. In particular, origin tracking, dynamic dependence tracking, and the matching of language-specific patterns have been implemented successfully.

Fig. 8 shows some snapshots of a language-specific single-stepping tool for the language CLaX [10, 19], a substantial subset of Pascal that features procedures with nested scopes, unstructured control flow, and multi-dimensional arrays. This tool has been implemented according to the techniques of Sec. 4.1.



(a) **(b)**

Fig. 8. Generated language-specific single-stepping tool.

Fig. 9 shows a screen dump of a dynamic slicing tool for the language CLaX, that was created using the technique of Sec. 5. In this figure, the dynamic slice with respect to the final value of variable `product` is shown, both in pure "term slice" form (here, '<?>' indicates a missing subterm), and in post-processed form.

7 Related work

The work that is most closely related to ours was done in the context of the PSG system [3]. A generator for language-specific debuggers was described in [2]. Language-specific compilers are generated by compiling denotational semantics definitions to a functional language. A standard, language-independent interpreter is used to execute the generated functional language fragments. The behavior of a debugger is specified using a set of built-in debugging concepts. In particular, trace functions are provided for the visualization of execution. Other notions enable one to inspect the state of the interpreter, and to define breakpoints. Bahlke et al. write that 'correspondences between the AST and the terms of the functional language are established in both directions'. These correspondences are used to determine a language-specific notion of a step. However, the nature of these "correspondences" is not described, making it impossible to conclude how powerful these correspondences are, or what constraints on specifications they imply[7]. By contrast, our method for keeping track of correspondences, origin tracking [8], is well-defined, and has proven to be sufficiently powerful for realistic languages [19]. A second difference between the work by Bahlke et al. is the information that is used to define debugger features. In our approach, debugger features are defined in terms of specification-level patterns in conjunction with language-independent origin information. That is, the specification of the interpreter and the specification of debugger features are uniform. It is unclear to what extent the debugging concepts of [2] are similar to the interpreter's specification. Finally, Bahlke et al. do not consider more advanced debugger features such as watchpoints, data breakpoints, and dynamic slices.

[7] The subset of Pascal that is considered in [2] does not contain **goto** statements. It is unclear what complications these statements would cause.

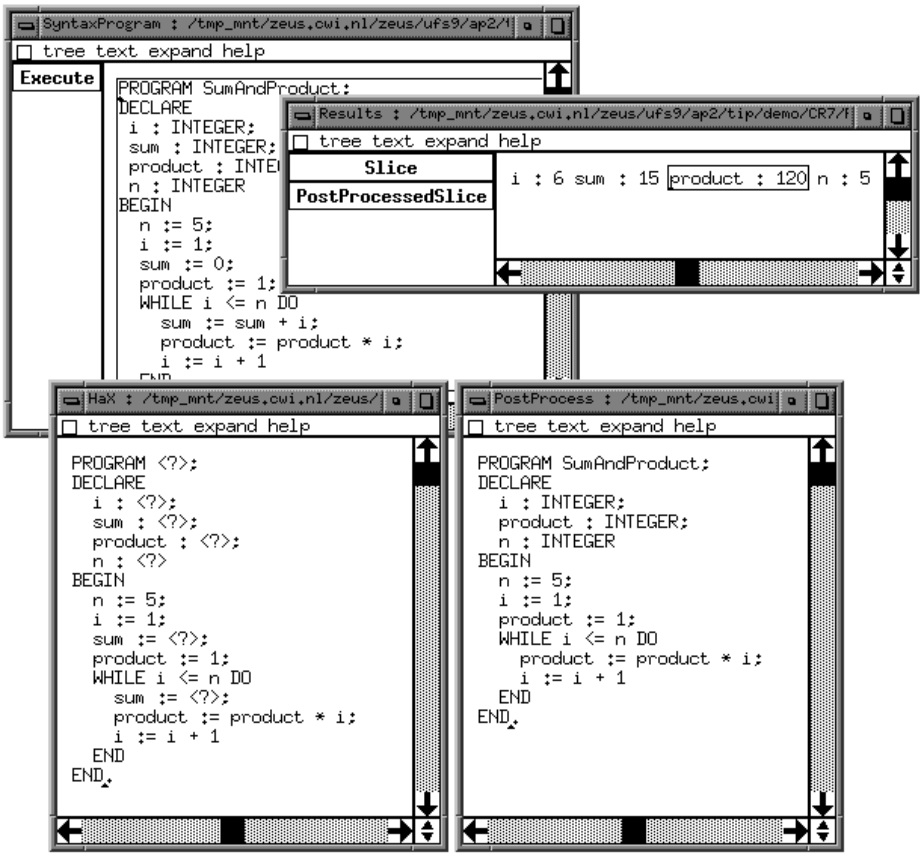


Fig. 9. Generated language-specific dynamic slicing tool.

Bertot [6] contributes a technique called *subject tracking* to the specification language Typol, which is based on natural semantics [14], for animation and debugging purposes. A key property of Typol specifications is that the meaning of a language construct is expressed in terms of its sub-constructs. A special variable, *Subject*, serves to indicate the language construct currently processed. This variable may be manipulated by the specification writer, when different animation or debugging behavior is required. Bertot does not consider other debugger features besides single-stepping, animation, and simple breakpoints.

Berry [5] presents an approach where animators are generated from structured operational semantics definitions. These specifications are augmented with *semantic display rules* that determine how to perform animation when a particular semantic rule is being processed. Various views of the execution of a program can be obtained by defining the appropriate display rules. Static views consist of parts of the AST of a program, and dynamic views are constructed from the program state during execution. As an example of a dynamic view, the evaluation of a control predicate may be visualized as the actual

truth value it obtains during execution. Although Berry considers highly sophisticated animation features, he does not consider debugger features such as breakpoints and dynamic program slices.

8 Conclusions and future work

We have presented a generic approach for deriving debugging and dynamic program slicing tools from algebraic specifications. The main conclusion of this paper is that the information needed for implementing such tools is to a very large extent *language-independent* and *implicitly* present in the language's specification. The three "building blocks" we used to define debugger features are:

1. matching of patterns,
2. rewriting of terms, and
3. computation of origin/dependence information.

The first two items consist of functionality that is, at least in principle, already provided by any rewriting engine. As was described in Sec. 3, the information used in the third item can be computed automatically, as a side-effect of rewriting.

The only additional *language-dependent* information that is required to define debugging and slicing features consists of the specification of a set of language-specific patterns, and the actions that should be performed when a match with such a pattern occurs.

The emphasis of this paper has been on generic techniques for constructing debugging tools; we have ignored all aspects that have to do with user-interfacing. In the future, we plan to develop a formalism in which one can *specify* such tools together with their user-interfaces.

Acknowledgments

I am grateful to Paul Klint and T.B. Dinesh for their comments on a draft of this paper.

References

1. AGRAWAL, H., AND HORGAN, J. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* (1990), pp. 246–256. *SIGPLAN Notices* 25(6).
2. BAHLKE, R., MORITZ, B., AND SNELTING, G. A generator of language-specific debugging systems. In *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques* (1987), pp. 92–101. SIGPLAN Notices 22(7).
3. BAHLKE, R., AND SNELTING, G. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems 8*, 4 (1986), 547–576.
4. BERGSTRA, J., HEERING, J., AND KLINT, P., Eds. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

5. BERRY, D. *Generating Program Animators from Programming Language Semantics*. PhD thesis, University of Edinburgh, 1991.
6. BERTOT, Y. Occurrences in debugger specifications. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 327–337. *SIGPLAN Notices* 26(6).
7. DEURSEN, A. V. *Executable Language Definitions—Case Studies and Origin Tracking Techniques*. PhD thesis, University of Amsterdam, 1994.
8. DEURSEN, A. V., KLINT, P., AND TIP, F. Origin tracking. *Journal of Symbolic Computation 15* (1993), 523–545.
9. DINESH, T. Type checking revisited: Modular error handling. In *International Workshop on Semantics of Specification Languages* (1993).
10. DINESH, T., AND TIP, F. Animators and error reporters for generated programming environments. Report CS-R9253, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1992.
11. FIELD, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1992), pp. 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.
12. FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995). To appear.
13. FIELD, J., AND TIP, F. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming* (1994), M. Hermenegildo and J. Penjam, Eds., vol. 844, Springer-Verlag, pp. 415–431.
14. KAHN, G. Natural semantics. In *Fourth Annual Symposium on Theoretical Aspects of Computer Science* (1987), F. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds., vol. 247 of *LNCS*, Springer-Verlag, pp. 22–39.
15. KLINT, P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology 2*, 2 (1993), 176–201.
16. KLOP, J. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1992, pp. 1–116.
17. KOREL, B., AND LASKI, J. Dynamic slicing of computer programs. *Journal of Systems and Software 13* (1990), 187–195.
18. STALLMAN, R., AND PESCH, R. *Using GDB, A guide to the GNU Source-Level Debugger*. Free Software Foundation/Cygnus Support, 1991. Version 4.0.
19. TIP, F. Animators for generated programming environments. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *LNCS*, Springer-Verlag, pp. 241–254.
20. TIP, F. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), 1994.
21. TIP, F. *Generation of Program Analysis Tools*. PhD thesis, University of Amsterdam, 1995.
22. WAHBE, R., LUCCO, S., AND GRAHAM, S. Practical data breakpoints: Design and implementation. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation* (Albuquerque, NM, 1993), pp. 1–12. *SIGPLAN Notices* 28(6).

This article was processed using the LaTeX macro package with LLNCS style