Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing

John Field[1] and Frank Tip[2] *

[1] IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY, 10598, USA; jfield@watson.ibm.com
[2] CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands; tip@cwi.nl

Abstract. *Program slicing* is a useful technique for debugging, testing, and analyzing programs. A program slice consists of the parts of a program which (potentially) affect the values computed at some point of interest. With rare exceptions, program slices have hitherto been computed and defined in ad-hoc and language-specific ways. The principal contribution of this paper is to show that general and semantically well-founded notions of slicing and *dependence* can be derived in a simple, uniform way from *term rewriting systems* (TRSs). Our slicing technique is applicable to any language whose semantics is specified in TRS form. Moreover, we show that our method admits an efficient implementation.

1 Introduction

1.1 Overview

Program slicing is a useful technique for debugging, testing, and analyzing programs. A program slice consists of the parts of a program which (potentially) affect the values computed at some point of interest, referred to as the *slicing criterion*. As originally defined by Weiser [19], a slicing criterion was the value of a variable at a particular program point and a slice consisted of an "executable" subset of the program's original statements. Numerous variations on the notion of slice have since been proposed, as well as many different techniques to compute them [16], but all reduce to determining *dependence* relations among program components. Unfortunately, with rare exceptions, the notion of "dependence" has been defined in an ad-hoc and language-specific manner, resulting in algorithms for computing slices that are notoriously difficult to understand, especially in the presence of pointers, procedures, and unstructured control flow. The contributions of this paper are as follows:

We define a general notion of slice that applies to any unconditional term rewriting system (TRS). Our definition uses a relation on *contexts* derived from the reduction relation on terms. This relation makes precise the *dynamic dependence* of function symbols in terms in a reduction sequence on symbols in previous terms in that sequence. Our

* Supported in part by the European Union under ESPRIT project # 5399 (Compiler Generation for Parallel Machines—COMPARE). Part of this work was done when the second author was at IBM.

notion of dependence does not require labeled terms [2, 3, 13, 14], and is distinguished by its ability to treat (normally problematic) TRSs with left-nonlinear rules.

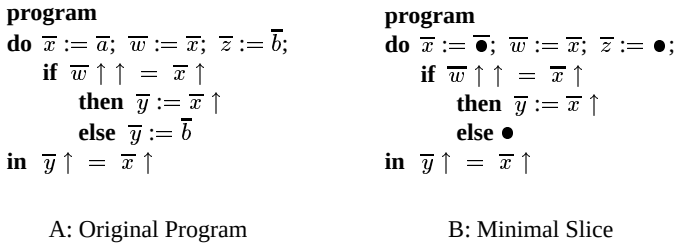
Our notion of slicing subsumes most of those defined in previous work on program slicing. The distinction traditionally made between "static" and "dynamic" slicing can be modeled by reduction of open or closed terms, respectively. Partial instantiation of open terms yields a useful intermediate notion of *constrained* slicing. Although Venkatesh defines a similar notion abstractly [17], he does not indicate how to *compute* such slices.

We describe an algorithm by which slices can be efficiently computed in practice by systematically transforming the original TRS to gather dependence information. The overhead required to compute this information is linear in the size of the initial term.

In [9], we present proofs that our definitions yield *minimal* (with respect to the reduction) and *sound* slices which have been omitted here because of space limitations. In a forthcoming companion paper, we will show how our techniques can be applied to standard programming languages, and compare these techniques to other algorithms in the literature. Here, we will concentrate primarily on technical foundations.

1.2 Motivating Examples

Consider the program in Figure 1A below, written in a tiny imperative programming language, **P**. The semantics of **P** are similar to those of many imperative programming languages with pointers. A **do** construct is executed by evaluating its statement list, and using the computed values to evaluate its **in** expression. Expressions of the form '$\overline{x}$' are atoms, and play the dual role of basic values and addresses which may be assigned to using ':='. Addresses are explicitly dereferenced using '$\uparrow$'. The distinguished atoms $\overline{t}$ and $\overline{f}$ represent boolean values.



A: Original Program B: Minimal Slice

Fig. 1. Example **P** Program.

We evaluate **P** programs by applying the *rewriting rules* of Figure 2 to the *term* consisting of the program's syntax tree until no further rules are applicable. This reduction process produces a sequence of terms ending with a *normal form* that denotes the result of the evaluation. The program in Figure 1A reduces to the normal form '**result** $\overline{t}$'. Figure 1B depicts the slice of the example program with respect to this normal form. The symbol '•' represents subterms of the program that do not affect its result.

It should be clear that a program slice is valuable for understanding which program components depend critically on the slicing criterion—even in the small example of Figure 1, this is not immediately obvious. Slicing information can be used to determine

what statements might have to be changed in order to correct an error or to alter the value of the criterion. The techniques we describe also allow the programmer the option of binding various inputs to values or leaving them undefined, allowing the effects of various initial conditions to be precisely traced. This significant capability is unique to our approach, and derives from its generality. In addition, by defining different (TRS-based) semantics for the same language, different sorts of slices can be derived. For instance, by using variants of the semantics in [7], we can compute both traditional "static" and "dynamic" [16] slices for the same language.

[P1]	$\overline{X} = \overline{X}$	$\rightarrow \overline{t}$	
[P2]	$\overline{a} = \overline{b}$	$\rightarrow \overline{f}$	for all constants a, b such that $a \neq b$
[P3]	**if** $\overline{t}$ **then** X **else** Y	$\rightarrow X$	
[P4]	**if** $\overline{f}$ **then** X **else** Y	$\rightarrow Y$	
[P5]	**do** X **in** $\overline{Y}$	$\rightarrow \overline{Y}$	
[P6]	**do** X **in** $Y = Z$	$\rightarrow$ (**do** X **in** Y) $=$ (**do** X **in** Z)	
[P7]	**do** $X; A := E$ **in** $(B \uparrow)$	$\rightarrow$ **if** (**do** $X; A := E$ **in** B) $=$ (**do** X **in** A) **then** (**do** X **in** E) **else do** X **in** ((**do** $A := E$ **in** B) $\uparrow$)	
[P8]	**do** $A := E$ **in** $(B \uparrow)$	$\rightarrow$ **if** (**do** $A := E$ **in** B) $= A$ **then** E **else** ((**do** $A := E$ **in** B) $\uparrow$)	
[P9]	**do** X; **if** A **then** B **else** C **in** E	$\rightarrow$ **if** (**do** X **in** A) **then** (**do** $X; B$ **in** E) **else** (**do** $X; C$ **in** E)	
[P10]	**do if** A **then** B **else** C **in** E	$\rightarrow$ **if** A **then** (**do** B **in** E) **else** (**do** C **in** E)	
[P11]	**do** X **in if** A **then** B **else** C	$\rightarrow$ **if** (**do** X **in** A) **then** (**do** X **in** B) **else** (**do** X **in** C)	
[P12]	**program** $\overline{X}$	$\rightarrow$ **result** $\overline{X}$	

Fig. 2. Rewriting Semantics of **P**.

We believe that our notion of a slice should also prove useful as an adjunct to theorem-proving systems, since it yields certain universally quantified equations from derivations of equations on closed terms. Consider, for example, the simple TRS **B** in Figure 3, which defines a few boolean identities ('$\wedge$' denotes conjunction, '$\oplus$' exclusive-or). Figure 4 shows how **B**-term ff $\wedge$ (tt $\oplus$ tt) can be reduced to ff. Observe that in deriving the theorem ff $\wedge$ (tt $\oplus$ tt) $=$ ff, we actually derive the *more general* theorem $P \wedge (\text{tt} \oplus \text{tt}) = \text{ff}$, for arbitrary P. From the point of view of slicing, the slice with respect to the normal form ff is the subcontext $\bullet \wedge (\text{tt} \oplus \text{tt})$ of the initial term. To determine such a slice, we must pay careful attention to the behavior of *non-linear* rules such as **[B4]** and **[P1]**, which many authors on reduction-theoretic properties of TRSs do not treat. In the sequel, we show how minimal slices can be obtained by examining the manner in which rules *create* new function symbols, and *residuate*, or "move around" old ones.

[B1]	$X \wedge (Y \oplus Z) \longrightarrow (X \wedge Y) \oplus (X \wedge Z)$	**[B3]**	$X \wedge \text{ff} \longrightarrow \text{ff}$
[B2]	$X \wedge \text{tt} \longrightarrow X$	**[B4]**	$X \oplus X \longrightarrow \text{ff}$

Fig. 3. Boolean TRS **B**.

Note that the reduction of Figure 4 is not the only one which yields the normal form 'ff'. In this case, the same slice will be computed for *any* **B**-reduction starting with T_0. For *orthogonal* TRSs [12], it can be shown that slices are *always* independent of the order in which rules are applied. However, in general, slices may depend on the particular reduction used.

$$\underline{\mathrm{ff} \wedge (\mathrm{tt} \oplus \mathrm{tt})} \equiv T_0 \xrightarrow{[B1]} (\underline{\mathrm{ff} \wedge \mathrm{tt}}) \oplus (\mathrm{ff} \wedge \mathrm{tt}) \equiv T_1 \xrightarrow{[B2]} \mathrm{ff} \oplus (\underline{\mathrm{ff} \wedge \mathrm{tt}}) \equiv T_2$$
$$\xrightarrow{[B2]} \underline{(\mathrm{ff} \oplus \mathrm{ff})} \equiv T_3 \xrightarrow{[B4]} \mathrm{ff} \equiv T_4$$

Fig. 4. A **B**-reduction; redexes are underlined.

1.3 Definition of a Slice

In general, we will define a slice as a certain *context* contained in the initial term of some reduction. Intuitively, a context may be viewed as a connected (in the sense of a tree) subset of function symbols taken from a term. For instance, if $f(g(a, b),\ c) \equiv T$ is a term, then one of several contexts contained in T is $g(\bullet, b) \equiv C$. C contains an omitted subterm, or *hole*[1], denoted by '$\bullet$'. This hole results from deleting the subterm 'a' of T. We denote the fact that C is a subcontext of T by $C \sqsubseteq T$; contexts as well as terms may contain subcontexts.

In a slice, holes denote subterms that are irrelevant to the computation of the criterion. Figure 1B depicts the *minimal subcontext of the original program that yields the slicing criterion via a "subreduction" of the original reduction*. Informally, the holes in the slice could be replaced by *any* **P**-expression and the same criterion could be produced by a **P**-reduction.

Definition 1.1 below makes precise our notion of slice. We will formalize the notion of "subreduction" of a sequence of reduction steps ρ using a set *Project*$*^{\rho}$, which is a collection of triples of the form $\langle C, \rho', C' \rangle$. Informally, such a triple denotes the fact that context C reduces to context C' by a reduction ρ' derived from rule applications that also occur in ρ. We discuss *Project*$*^{\rho}$ further in Section 6 and provide a complete definition in [9].

Definition 1.1 (Slice) *Let $\rho : T \longrightarrow^* T'$ be a reduction. Then a <u>slice</u> with respect to a subcontext C' of T' is a subcontext C of T with the property that there exists a reduction ρ' such that $\rho' : C \longrightarrow^* D'$ for some $D' \sqsupseteq C'$ and $\langle C, \rho', D' \rangle \in$ Project$*^{\rho}$. Slice C is <u>minimal</u> if there is no slice with respect to criterion C' that contains fewer function symbols.*

Definition 1.1 is rendered pictorially in Figure 5.

The notion of TRS-based slice we define in the sequel can be used for any language whose operational semantics is defined by a TRS. Many languages whose semantics are traditionally defined via extended lambda-calculi or using structural operational semantics also have corresponding rewriting semantics [1, 7]. In [8], it is shown how many traditional program constructs may be modeled by an appropriate TRS.

[1] Some authors require that contexts contain exactly one hole; we will not.

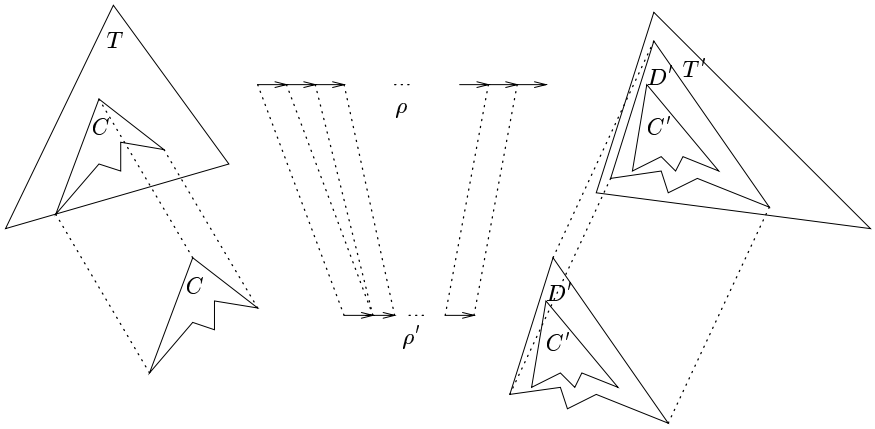


Fig. 5. Depiction of the definition of a slice.

2 Basic Definitions

In this section, we make precise the notion of a *context* introduced informally in the previous section. This notion will be the cornerstone of our formalization of slicing and dependence. Instead of deriving contexts from the usual definition of a term, we view terms as a special class of contexts. Contexts will be defined as connected fragments of *trees* decorated with function symbols and variables. We begin with a few preliminary definitions, most of which are standard.

2.1 Signatures, Paths, Context Domains

A *signature* Σ is a finite set of *function symbols*; associated with each function symbol $f \in \Sigma$ is a natural number $arity(f)$, its number of arguments. We will use a denumerable set of *variables* $\mathcal{V}$ such that $\Sigma \cap \mathcal{V} = \emptyset$. By convention, for each variable $X \in \mathcal{V}$, $arity(X) = 0$. Lower-case letters of the form $f, g, h, \cdots$ will denote function symbols and upper-case letters of the form $X, Y, Z, \cdots$ will represent variables.

A *path* is a sequence of positive integers that designates a particular function symbol or subtree by encoding a walk from the tree's root. The empty path, '$()$', designates the root of a tree; path $(i_1\ i_2 \cdots i_m)$ designates the i_m^{th} subtree (counted from left to right) of the subtree indicated by path $(i_1\ i_2 \cdots i_{(m-1)})$. The operation '$\cdot$' denotes path concatenation. Path p is a *prefix* of path q if there exists an r such that $q = p \cdot r$; this is notated $p \preceq q$. If $r \neq ()$ then $p \prec q$. A *context domain* P is a set of paths designating a connected fragment of a tree. This means that P must (i) possess a unique root, *root*(P), such that for all $p \in P$, *root*$(P) \preceq p$, and (ii) have no "gaps," i.e., for all p, q, r such that $p \prec q \prec r$ and $p, r \in P$ it must be the case that $q \in P$.

2.2 Contexts

We can now define a context as a total mapping from a context domain to function symbols and variables:

Definition 2.1 (Context) *Let Σ be a signature, $\mathcal{V}$ be a set of variables, and $\mathcal{P}$ be a context domain. Let μ be a total mapping from $\mathcal{P}$ to $(\Sigma \cup \mathcal{V})$ and p be a path. Then a pair $\langle p, \mu \rangle$ is a $\Sigma\mathcal{V}$-context if and only if:*

(i) *For all $q \in \mathcal{P}$ and $s \in \Sigma \cup \mathcal{V}$ such that $\mu(q) = s$, $q \cdot i \in \mathcal{P}$ for some i implies that $i \leq$ arity(s).*
(ii) *If $\mathcal{P} \neq \emptyset$, then $p =$ root$(\mathcal{P})$.*

Clause (i) of Definition 2.1 ensures that a child of a function symbol f must have an ordinal number less than or equal to the arity of f. Clause (ii) ensures that the root of the context is the same as the root of its underlying domain, except when the domain is empty; in the latter case, we will say that the context is *empty*. The definition is specifically designed to admit empty contexts, which will be important in the sequel for describing the behavior of *collapse rules*, i.e., rewriting rules whose right hand sides are single variables. Given context $C \equiv \langle p, \mu \rangle$, *root*$(C)$ denotes the path p, and $\mathcal{O}(C)$ the domain of μ. The path corresponding to a "missing child" in a context will be referred to as a *hole occurrence*; an empty context is also a hole. We will use $Cont(\Sigma, \mathcal{V})$ to denote the set of all $\Sigma\mathcal{V}$-contexts.

For any context C and a path p, $p \leftarrow C$ denotes an isomorphic context rooted at p obtained by *rerooting* C. This notation is used to represent contexts textually; e.g., $p \leftarrow f(\bullet, g(a, \bullet))$ represents a context rooted at p with two holes ('$\bullet$'), binary function symbols f and g and a constant a. $p \leftarrow \bullet$ represents an empty context rooted at p.

A context C is a *term* if: (i) C has no hole occurrences, and (ii) *root*$(C) = ()$. Although the restriction of *root*(C) to be $()$ is not strictly necessary, it results in a definition that agrees most closely with that used by other authors. We will use $Term(\Sigma)$ to denote the set of terms over signature Σ. Letters $C, D, \cdots$ will generally denote arbitrary contexts, and $S, T, \cdots$ terms. Whenever convenient, we ignore the distinction between a variable X and the term consisting of that variable. Some convenient operations on contexts are introduced next in an informal way; formal definitions can be found in [9].

For a context C, and $\mathcal{S}$ a subset of $\Sigma \cup \mathcal{V}$, $\mathcal{O}_{\mathcal{S}}(C)$ denotes the set of paths to elements of $\mathcal{S}$ in C; $\mathcal{O}_{\{s\}}(C)$ is abbreviated by $\mathcal{O}_s(C)$. The set of variable occurrences in a $\Sigma\mathcal{V}$-context C, i.e., $\mathcal{O}_{\mathcal{V}}(C)$, is denoted $vars(C)$, and *vars*$_1(C)$ is the set of variables which occur exactly once in C.

Two contexts are *compatible* if all paths common to both of their domains are mapped to the same symbol. If C and D are compatible, C is a *subcontext* of D, notated $C \sqsubseteq D$, if and only if one of the following holds: (i) C and D are nonempty and $\mathcal{O}(C) \subseteq \mathcal{O}(D)$, (ii) C and D are empty and $C \equiv D$, or (iii) C is empty, D is nonempty, *root*$(C) = q \cdot i \in \mathcal{O}(D)$, and $q \in \mathcal{O}(D)$. The third clause states that an empty context C is a subcontext of a nonempty context D *only* if its root is "sandwiched" between adjacent nodes in D. This property will greatly simplify a number of definitions in the sequel. Contexts D and E are *disjoint* if and only if there exists no context C such that $C \sqsubseteq D$ and $C \sqsubseteq E$. If C and D are contexts such that *root*$(D) \in \mathcal{O}(C)$, $C[D]$ denotes the context C where the subcontext at *root*(D) is *replaced* by D.

A context *forest* is a set of mutually disjoint contexts. Forest $\mathcal{S}$ is a *subforest* of forest $\mathcal{T}$, notated $\mathcal{S} \sqsubseteq \mathcal{T}$, if and only if for all contexts $C \in \mathcal{S}$, there exists a context $D \in \mathcal{T}$ such that $C \sqsubseteq D$. Some convenient set-like operations on context forests can be defined

as follows: Let $\mathcal{S}$ and $\mathcal{T}$ be compatible context forests. Then their *union*, notated $\mathcal{S} \sqcup \mathcal{T}$, is the smallest forest $\mathcal{U}$ such that $\mathcal{S} \sqsubseteq \mathcal{U}$ and $\mathcal{T} \sqsubseteq \mathcal{U}$; their *difference*, notated $\mathcal{S} - \mathcal{T}$, is the smallest forest $\mathcal{U}$ such that $\mathcal{U} \sqsubseteq \mathcal{S}$ and $\mathcal{S} \sqsubseteq (\mathcal{T} \sqcup \mathcal{U})$. If $\mathcal{P}$ is a set of paths, C / P is the forest containing subcontexts of C rooted at paths in $\mathcal{P}$. The notion of context replacement is easily generalized to a forest $\mathcal{S}$. We will feel free to refer to a singleton forest $\{C\}$ by its element C when no confusion arises; e.g., "$C \sqcup D$".

3 Term Rewriting and Related Relations

In this section, we formalize standard term rewriting-related notions using operations on contexts; we then define the important related ideas of *creation* and *residuation*, which are derived from the rewriting relation. We will first consider only *left-linear* TRSs; this restriction will be removed in in Section 5.

3.1 Substitutions and Term Rewriting Systems

A *substitution* is a finite partial map from $\mathcal{V}$ to $Cont(\Sigma, \mathcal{V})$, where Σ is a signature and $\mathcal{V}$ a set of variables. Applying a substitution σ to a context C corresponds to replacing each subcontext $C_X \sqsubseteq C$ consisting solely of a variable X by the context ($root(C_X) \leftarrow \sigma(X)$), for all X on which σ is defined. A *term rewriting system* $\mathcal{R}$ over a signature Σ is any set of pairs $\langle L, R\rangle$ such that L and R are terms over Σ, and $vars(R) \subseteq vars(L)$; $\langle L, R\rangle$ is called a *rewrite rule* and is commonly notated $L \to R$. For $\alpha \equiv L \to R \in \mathcal{R}$ we define $L_\alpha = L$ and $R_\alpha = R$. A rewrite rule α is *left-linear* if $vars(L_\alpha) = vars_1(L_\alpha)$. If $\mathcal{R}$ is a TRS, then we define an $\mathcal{R}$*-contraction* $\mathcal{A}$ to be a triple $\langle p, \alpha, \sigma\rangle$, where p is a path, α is a rule of $\mathcal{R}$, and σ is a substitution.

We use $P_\mathcal{A}$, $\alpha_\mathcal{A}$, $L_\mathcal{A}$, $R_\mathcal{A}$, and $\sigma_\mathcal{A}$ to denote p, α, $L(\alpha_\mathcal{A})$, $R(\alpha_\mathcal{A})$, and σ, respectively. Moreover, $\overline{L_\mathcal{A}}$ and $\overline{R_\mathcal{A}}$ will denote the contexts $(P_\mathcal{A} \leftarrow L_\mathcal{A})$ and $(P_\mathcal{A} \leftarrow R_\mathcal{A})$, respectively. The $\mathcal{R}$*-contraction relation*, $\longrightarrow_\mathcal{R}$, is defined by requiring that $T \longrightarrow_\mathcal{R} T'$ if and only if a contraction $\mathcal{A}$ exists such that $T \equiv T[\sigma_A(\overline{L_\mathcal{A}})]$ and $T' \equiv T[\sigma_A(\overline{R_\mathcal{A}})]$ The subcontext $\sigma_A(\overline{L_A})$ of C is an α_A*-redex*, and the context $\sigma_A(\overline{R_A})$ is an α_A*-reduct*; these contexts are abbreviated respectively by $Redex_A$ and $Reduct_A$. As usual, $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$. A *reduction* ρ is a sequence of contractions $\mathcal{A}_1\mathcal{A}_2 \ldots \mathcal{A}_n$ such that if ρ is nonempty, there exist terms $T_0, T_1, \ldots, T_n$ where:

$$T_0 \xrightarrow{\mathcal{A}_1} T_1 \xrightarrow{\mathcal{A}_2} T_2 \cdots T_{n-1} \xrightarrow{\mathcal{A}_n} T_n$$

This reduction is abbreviated by $\rho : T_0 \longrightarrow T_n$. A reduction ρ is a reduction *of* term T if there exists T' such that $\rho : T \longrightarrow^* T'$. The reduction of length 0 is denoted by ϵ; for all terms T, we adopt the convention that $\epsilon : T \longrightarrow^* T$.

Given the definitions above, the **B**-reduction depicted in Figure 4 may be described formally by the following sequence of contractions:

$\langle(), [\mathbf{B1}], [X := \text{ff}, Y := \text{tt}, Z := \text{tt}]\rangle$; $\langle(1), [\mathbf{B2}], [X := \text{ff}]\rangle$; $\langle(2), [\mathbf{B2}], [X := \text{ff}]\rangle$; $\langle(), [\mathbf{B4}], [X := \text{ff}]\rangle$

Most of the new relations defined in the sequel are parameterized with a reduction $\rho\mathcal{A}$, in which the final contraction is highlighted. Several definitions are concerned with

the last contraction $\mathcal{A}$ only; however, when our definitions are generalized in Section 5, the "history" contained in ρ will become relevant. Whenever we define a truly *inductive* relation on $\rho\mathcal{A}$, we will append a '$*$' to the name of the relation.

3.2 Residuation and Creation

In order to formalize our notion of slice, we must first reformulate the standard notion of *residual* and the somewhat less standard notion of *creation* in terms of contexts. Each of these will use Definition 3.1, which formalizes how an application of a contraction $\mathcal{A}$ has the effect of "copying," "moving," or "deleting" contexts bound to variable instances in $L_\mathcal{A}$ when $R_\mathcal{A}$ is instantiated. The elements of the set $\mathit{VarPairs}^{\rho\mathcal{A}}$ are pairs $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle$ of context forests, such that contexts $C_1 \in \mathcal{S}_1$ and $C_2 \in \mathcal{S}_2$ are corresponding subcontexts of the context bound to some variable in $\alpha_\mathcal{A}$.

Definition 3.1 (*VarPairs*) *Let ρA be a reduction. Then*

$$\begin{aligned}\mathit{VarPairs}^{\rho\mathcal{A}} \triangleq \{\ \langle \mathcal{S}_1, \mathcal{S}_2 \rangle \ \mid\ & X \in \mathcal{V},\ C \sqsubseteq (() \leftarrow \sigma_\mathcal{A}(X)),\ q = \mathit{root}(C), \\ & \mathcal{S}_1 = \{(p_L \cdot q \leftarrow C) \mid p_L \in \mathcal{O}_X(\overline{L_\mathcal{A}})\}, \\ & \mathcal{S}_2 = \{(p_R \cdot q \leftarrow C) \mid p_R \in \mathcal{O}_X(\overline{R_\mathcal{A}})\}\ \}\end{aligned}$$

In left-linear systems, for any pair $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}$, $\mathcal{S}_1$ is always a singleton. This will not, however, be the case when we generalize the definition for left-nonlinear systems.

Definition 3.2 is the standard notion of *residual*, in relational form. For a contraction $\mathcal{A} : C \longrightarrow C'$, *Resid* associates each subcontext of C that is not affected by $\mathcal{A}$ with the corresponding subcontext of C'. Moreover, for each $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}$, $C_1 \in \mathcal{S}_1$, and $C_2 \in \mathcal{S}_2$, C_1 is related to C_2. If $\mathcal{S}_2$ is empty, this will have the effect that no pairs are added to $\mathit{Resid}^{\rho\mathcal{A}}$.

Definition 3.2 (*Resid*) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{aligned}\mathit{Resid}^{\rho\mathcal{A}} \triangleq \{\ \langle D_1, D_2 \rangle \mid\ & D_1 \in \mathcal{S}_1,\ D_2 \in \mathcal{S}_2,\ \langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}\ \} \cup \\ & \{\ \langle D, D \rangle \mid\ D \textit{ and } \mathit{Redex}_\mathcal{A} \textit{ are disjoint}\ \}\end{aligned}$$

Figure 6 depicts *Resid* and several other definitions we will encounter in the sequel, as they apply to the initial and final contractions of the reduction in Figure 4, involving the left-linear rule **[B1]** and the left-*non*linear rule **[B4]** of TRS **B**, respectively.

Definition 3.3 describes the *creating* and the *created* contexts associated with a contraction $\mathcal{A}$. Intuitively, if contraction $\mathcal{A}$ is applied to term T, the creating context is the minimal subcontext of T needed for the left-hand side of $\mathcal{A}$'s rule to match; the created context is the corresponding minimal context "constructed" by the right-hand side of the rule. The former is defined as the context derived by subtracting from $\mathit{Redex}_\mathcal{A}$ all contexts $D_1 \in \mathcal{S}_1$ such that $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}$. The latter is the context derived by subtracting from $\mathit{Reduct}_\mathcal{A}$ all contexts $D_2 \in \mathcal{S}_2$ such that $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}$.

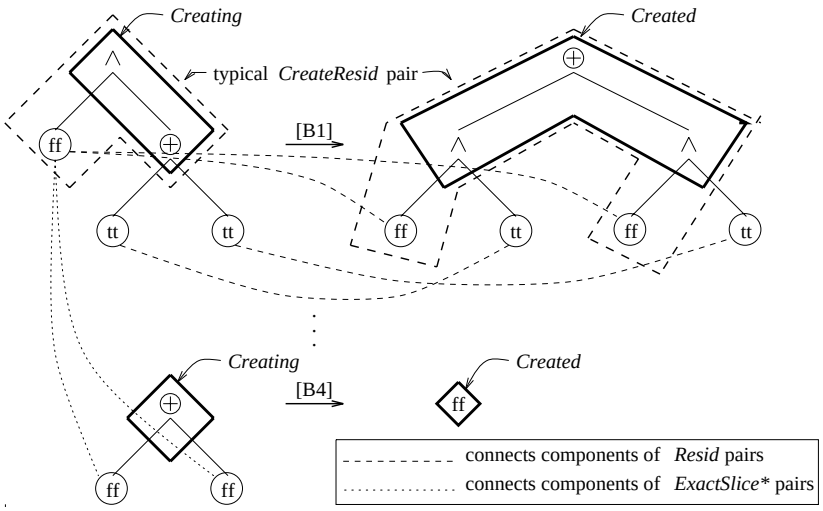


Fig. 6. Illustration of selected relations and contexts derived from **B**-reduction of Figure 4.

Definition 3.3 (*Creating* **and** *Created***)** *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\mathit{Creating}^{\rho\mathcal{A}} \triangleq \begin{cases} \mathit{Redex}_{\mathcal{A}} - \bigsqcup\{\mathcal{S}_1 \mid \langle \mathcal{S}_1, \mathcal{S}_2\rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}\} & \text{when } L_{\mathcal{A}} \notin \mathcal{V} \\ P_{\mathcal{A}} \leftarrow \bullet & \text{otherwise} \end{cases}$$

$$\mathit{Created}^{\rho\mathcal{A}} \triangleq \begin{cases} \mathit{Reduct}_{\mathcal{A}} - \bigsqcup\{\mathcal{S}_2 \mid \langle \mathcal{S}_1, \mathcal{S}_2\rangle \in \mathit{VarPairs}^{\rho\mathcal{A}}\} & \text{when } R_A \notin \mathcal{V} \\ P_{\mathcal{A}} \leftarrow \bullet & \text{otherwise} \end{cases}$$

While $\mathit{Creating}^{\rho\mathcal{A}}$ and $\mathit{Created}^{\rho\mathcal{A}}$ could have been defined in a more direct way from the structure of $L_{\mathcal{A}}$, $R_{\mathcal{A}}$, and $P_{\mathcal{A}}$ without using $\mathit{VarPairs}^{\rho\mathcal{A}}$ at all, the approach we take here will be much easier to generalize when we consider left-nonlinear systems.

Combining Definitions 3.2 and 3.3, we arrive at the relation *CreateResid*, formalized in Definition 3.4. Every pair of *terms* $\langle T, T'\rangle \in \mathit{CreateResid}$ has the property that $T \longrightarrow T'$.

Definition 3.4 (*CreateResid***)** *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{aligned} \mathit{CreateResid}^{\rho\mathcal{A}} \triangleq \{\, \langle C_1, C_2\rangle \mid\ & R \subseteq \mathit{Resid}^{\rho\mathcal{A}}, \\ & C_1 = \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\, C \mid \langle C, C'\rangle \in R \,\}, \\ & C_2 = \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\, C' \mid \langle C, C'\rangle \in R \,\} \ \ \} \end{aligned}$$

Note that it is *impossible* to have both $\langle C_1, D\rangle \in \mathit{Resid}^{\mathcal{A}}$ and $\langle C_2, D\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$, for any $C_1,\ C_2,\ D$.

4 A Dynamic Dependence Relation

In this section, we will derive our dynamic dependence relation, *Slice*$*$, using the concepts introduced in Section 3. We first consider the somewhat more restrictive relation *ExactSlice*, which is the union of $\mathit{Resid}^{\rho\mathcal{A}}$ and $\mathit{CreateResid}^{\rho\mathcal{A}}$.

Definition 4.1 (*ExactSlice*) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\mathit{ExactSlice}^{\rho\mathcal{A}} \triangleq \mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}}$$

The name of *ExactSlice* comes from the fact that its transitive closure is a subrelation of relation *Slice*$*$ shown below. For the empty reduction, *Slice*$*$ is defined as the identity relation. For a criterion D, the inductive case determines the minimal super-context $D' \sqsupseteq D$ for which there is a C such that $\langle C, D'\rangle \in \mathit{ExactSlice}^{\rho\mathcal{A}}$; then the slice for this C in reduction ρ is determined. Operation '$\cdot$' in Definition 4.2 denotes relational join.

Definition 4.2 (*Slice*$*$) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{array}{lll} \mathit{Slice}*^{\epsilon} & \triangleq \{\, \langle C, C\rangle \mid C \in \mathit{Cont}(\Sigma) \,\} & \\ \mathit{Slice}*^{\rho\mathcal{A}} & \triangleq \mathit{Slice}*^{\rho} \cdot \{\, \langle C, D\rangle \mid & \text{there exists a minimal } D' \sqsupseteq D \\ & & \text{such that } \langle C, D'\rangle \in \mathit{ExactSlice}^{\rho\mathcal{A}} \,\} \end{array}$$

4.1 Example

Consider the following **B**-reduction ρ:

$$S = (\mathrm{ff} \wedge \underline{(\mathrm{ff} \wedge \mathrm{tt})}) \wedge \mathrm{tt} \xrightarrow{\text{[B2]}} \underline{(\mathrm{ff} \wedge \mathrm{ff})} \wedge \mathrm{tt} \xrightarrow{\text{[B3]}} \mathrm{ff} \wedge \mathrm{tt} \xrightarrow{\text{[B2]}} \mathrm{ff} = T$$

Definition 4.2 yields the following relations between subcontexts of S and T:

$$\mathit{Slice}*^{\rho} = \{\, \langle (1) \leftarrow \bullet \wedge (\mathrm{ff} \wedge \mathrm{tt}), () \leftarrow \mathrm{ff}\rangle, \quad \langle () \leftarrow \bullet, () \leftarrow \bullet\rangle \,\}$$

Thus, the slice with respect to $() \leftarrow \mathrm{ff} \sqsubseteq T$ is $(1) \leftarrow \bullet \wedge (\mathrm{ff} \wedge \mathrm{tt}) \sqsubseteq S$. This is the minimal context for which there exists a subreduction of ρ that yields the criterion. In this case, the projection consists of the first two contractions.

The above example also illustrates why *Slice*$*$ is defined on contexts rather than on context domains: collapse rules require special treatment in order to produce minimal slices. Note that the example exhibits two applications of collapse rule **[B2]**. Intuitively, the first one created the criterion, whereas the second one merely affected its *location*. We achieve this differentiation by: (i) having a collapse rule create an empty context $P_{\mathcal{A}} \leftarrow \bullet$ instead of the context consisting of the function symbol at path $P_{\mathcal{A}}$ (the approach of [12]), and (ii) defining an empty context $p \leftarrow \bullet$ to be a subcontext of a nonempty context *only* if the latter "surrounds" the former.

5 Nonlinear Rewriting Systems

Unfortunately, our previous definitions do not extend trivially to left-nonlinear TRSs, because they do not account for the fact that non-linearities in the left-hand side of a rule constrain the set of contexts for which the rule is applicable. For example, when rule **[B4]** of TRS **B** of Figure 3 is applied to $\mathrm{ff} \oplus \mathrm{ff}$, this results in a contraction $\mathcal{A} : T \equiv \mathrm{ff} \oplus \mathrm{ff} \longrightarrow \mathrm{ff} \equiv T'$. Our previous definitions yield $C = () \leftarrow (\bullet \oplus \bullet) \sqsubseteq T$ as the slice with respect to criterion $D = () \leftarrow \mathrm{ff} \sqsubseteq T'$. This is not a valid slice, because some 'instantiations' of C do not reduce to a context containing D, e.g., $() \leftarrow \mathrm{tt} \oplus \mathrm{ff}$

does not. A related problem is that multiple contexts may be related to a criterion in the presence of left-nonlinear rules; this conflicts with our objective that a slice with respect to a context consists of a single context.

A simple solution to deal with nonlinear TRSs consists of restricting *VarPairs* to variables which occur at most once in the left-hand side of a rule. However, this would yield non-minimal slices. For instance, for the reduction of Figure 4 the non-minimal slice $\mathrm{ff} \wedge (\mathrm{tt} \oplus \mathrm{tt})$ would be computed. The immediate cause for this inaccuracy is the fact that the subcontexts $(1) \leftarrow \mathrm{ff}$ and $(2) \leftarrow \mathrm{ff}$ of T_3 are deemed responsible for the creation of term T_4. However, in this case, they are *reducts of the same subcontext* $C = (1) \leftarrow \mathrm{ff} \sqsubseteq T_0$. This being the case, C may be replaced by an *arbitrary* context without affecting the applicability of the left-nonlinear rule.

We can account for this fact by modifying the *VarPairs* relation as follows. If, for a rule α, all occurrences of a variable X in L_α are matched against reducts of some subcontext C, this gives rise to *residuation* (if X occurs in R_α). All other cases cause *creation*: the subcontexts matched against X in L_α are *creating*, and the subcontexts matched against X in R_α are *created*. Note that the former situation occurs trivially for all variables that occur exactly once in L_α. The notion of contexts being 'reducts' of the same context is formalized by way of the transitive and reflexive closure *ExactSlice*$*$ of relation *ExactSlice*. The revised definition of *VarPairs* is shown below.

Definition 5.1 (*VarPairs* for non-linear TRSs) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{aligned}\mathit{VarPairs}^{\rho\mathcal{A}} \triangleq \{\, \langle \mathcal{S}_1, \mathcal{S}_2 \rangle \;\mid\; & X \in \mathcal{V},\ C \sqsubseteq (() \leftarrow \sigma_A(X)),\ q = \mathit{root}(C), \\ & \textit{there exists a unique } D \textit{ such that for all } p_L \in \mathcal{O}_X(\overline{L_A}), \\ & \qquad \langle D, (p_L \cdot q \leftarrow C) \rangle \in \mathit{ExactSlice}*^{\rho}, \\ & \mathcal{S}_1 = \{(p_L \cdot q \leftarrow C) \mid p_L \in \mathcal{O}_X(\overline{L_\mathcal{A}})\}, \\ & \mathcal{S}_2 = \{(p_R \cdot q \leftarrow C) \mid p_R \in \mathcal{O}_X(\overline{R_\mathcal{A}})\} \;\}\end{aligned}$$

(where ExactSlice$*$ is the transitive and reflexive closure of relation ExactSlice.)

In Figure 6, certain pairs in the *ExactSlice*$*$ relation are depicted using dotted lines. Note that the two 'ff' subterms in the term matched by left-nonlinear rule **[B4]** have the *same ExactSlice*$*$ "origin" in the initial term. Definition 5.1 thus implies that the 'ff' subterms are components of *VarPairs*. Consequently, the *Creating* context for the **[B4]** contraction does not include the 'ff' subterms. Taken together, these facts allow us to conclude that the final term of the reduction of Figure 6 does *not* depend on the 'ff' subterm of the initial term of the reduction.

5.1 Example: Slicing in a non-linear system

Definitions 4.2 and 5.1 yield the following dependences for the example of Figure 4:

$$\mathit{Slice}*^{\rho} = \{\, \langle () \leftarrow \bullet, () \leftarrow \bullet \rangle,\ \langle () \leftarrow \bullet \wedge (\mathrm{tt} \oplus \mathrm{tt}), () \leftarrow \mathrm{ff} \rangle \,\}$$

Consequently, the minimal slice $\bullet \wedge (\mathrm{tt} \oplus \mathrm{tt}) \sqsubseteq T_0$ is computed for criterion $() \leftarrow \mathrm{ff} \sqsubseteq T_4$.

6 Formal Properties of Slices

In this section, we state some important formal properties of slices. We begin with further details of our notion of "subreduction".

6.1 Projections

Recall from Section 1.3 that we use the set *Project*$*^{\rho}$ to formalize subreductions of ρ. *Project*$*^{\rho}$ is a collection of triples of the form $\langle C, \rho', C' \rangle$, each of which denotes the fact that context C reduces to context C' by a reduction ρ' composed of contractions derived from those in ρ. Although we defer a full definition of this notion to [9], the idea is illustrated by the following $\mathbf{B}$-reduction:

$$T_0 \equiv \underline{\boxed{(\mathrm{ff} \wedge \mathrm{ff})} \wedge (\mathrm{tt} \oplus \mathrm{tt})} \xrightarrow{\mathcal{A}_{[B1]}} (\underline{(\mathrm{ff} \wedge \mathrm{ff})} \wedge \mathrm{tt}) \oplus ((\mathrm{ff} \wedge \mathrm{ff}) \wedge \mathrm{tt})$$
$$\xrightarrow{\mathcal{A}_{[B3]}} (\boxed{\mathrm{ff}} \wedge \mathrm{tt}) \oplus (\boxed{(\mathrm{ff} \wedge \mathrm{ff})} \wedge \mathrm{tt}) \equiv T_2$$

As usual, we have underlined each redex. We use $\mathcal{A}_{[B1]}$ and $\mathcal{A}_{[B3]}$ to denote the contractions that use rules [B1] and [B3], respectively. Some typical elements of the set *Project*$*^{\mathcal{A}_{[B1]}\mathcal{A}_{[B3]}}$ are:

$$\langle\ (1) \leftarrow \mathrm{ff} \wedge \mathrm{ff}\ ,\ \mathcal{A}_{[B1]}\mathcal{A}_{[B3]}\ ,\ (2\ 1) \leftarrow \mathrm{ff} \wedge \mathrm{ff}\ \rangle \quad \langle\ (1) \leftarrow \mathrm{ff} \wedge \mathrm{ff}\ ,\ \mathcal{A}_{[B1]}\mathcal{A}_{[B3]}\ ,\ (1\ 1) \leftarrow \mathrm{ff}\ \rangle$$

These triples correspond to the two "paths through the reduction" taken by the boxed subterm of T_0. One residual is contracted in a subsequent step, the other is not. The final elements of the projection triples corresponding to subterms in T_2 are also boxed.

6.2 Context Rewriting

In the discussion above, we were rather cavalier about extending the rewriting relation to contexts. To be more precise, we need the following definitions: A *variable instantiation* of a context C is a term T that can be obtained from C by replacing each hole with a variable that does not occur in C, and rerooting it to $()$. A variable instantiation is a *linear instantiation* if each hole is replaced by a *distinct* variable. A context C rewrites to a context C', notated $C \longrightarrow^* C'$, if and only if $T \longrightarrow^* T'$, where T is a linear instantiation of C and T' is a variable instantiation of C'. Note that we do not require that C and C' have the same roots. Also note that context reduction is *not* defined as the transitive closure of a single-step contraction relation on contexts; this is necessary to correctly account for the way in which a reduction causes distinct holes to be moved and copied, particularly in the case of left-nonlinear rules.

6.3 Theorems

We can now state some theorems describing the most important properties of slices. In [9], we provide full proofs; most of the proofs take the form of induction on the length of reductions with (sometimes tedious) case analyses in the single-step case. We first show that the slicing relation is single-valued, i.e., that *Slice*$*$ is one-to-one mapping from contexts to contexts:

Theorem 1 (Uniqueness of Slices). *Let $\rho : T \longrightarrow^* T'$ be a reduction, and let $D \sqsubseteq T'$. Then there exists a unique $C \sqsubseteq T$ such that $\langle C, D\rangle \in$ Slice$*^{\rho}$.*

Given Theorem 1, we will be able to write $C =$ *Slice*$*^{\rho}(D)$ instead of $\langle C, D\rangle \in$ *Slice*$*^{\rho}$.

The next theorem demonstrates that slices effectively preserve the topology of their corresponding criteria. This is important in showing that slices are minimal projections.

Theorem 2 (Inclusion Theorem). *Let $\rho : T \longrightarrow^* T'$ be a reduction, and let $D' \sqsubseteq D \sqsubseteq T'$. Then* Slice$*^{\rho}(D') \sqsubseteq$ *Slice*$*^{\rho}(D)$.

The following theorem states that a slice is the *minimal* initial component of some projection triple whose final component contains the slicing criterion:

Theorem 3 (Minimality). *If $\langle C, D\rangle \in$ Slice$*^{\rho}$ for some reduction ρ, then C is a minimal (with respect to '$\sqsubseteq$') element of the set $\{\, C' \mid D \sqsubseteq D',\ \langle C', \rho', D'\rangle \in$ Project$*^{\rho}\,\}$.*

Our last theorem shows that slices are *sound*, in the sense that they reduce to a supercontext of the slicing criterion (by a reduction derived from the original reduction by projection).

Theorem 4 (Soundness). *If $\langle C, D\rangle \in$ Slice$*^{\rho}$ for some reduction ρ, then there exists a context $D' \sqsupseteq D$ such that $C \longrightarrow^* D'$.*

Together, Theorems 3 and 4 imply that our construction of slices agrees with Definition 1.1.

7 Implementation

In principle, one could implement slicing by storing information about every step of a reduction ρ, and then computing *Slice*$*^{\rho}$ based on this information. In practice, such an approach is infeasible since it would require space and time proportional to the length of ρ for each choice of criterion. Since our reasons for investigating dependence relations are eminently practical, we will use an alternate method that allows slices to be computed as a "side-effect" of the reduction process, in a way that efficiently yields slices with respect to any chosen criterion. During the reduction process, our method will maintain (i) the slices for all function symbols in a term, and (ii) for every pair of adjacent function symbols, the *difference* between their individual slices, and the slice with respect to their (context) union. In [9], we will show that this is sufficient information to derive slices with respect to any non-empty criterion. An important issue is that the difference information for two adjacent function symbols is not necessarily equal to the slice with respect to the empty context 'between' them. The implementation method discussed below is based on the following starting-points:

– *Information about slices is coded into terms* by surrounding every function symbol with a special 'wrapper' symbol. If P is the slice associated with $f(\bullet, \cdots, \bullet)$, and Q is the difference information for this 'node' and its parent, this will be notated $\langle f(\bullet, \cdots, \bullet), P, Q\rangle$.

– *The computation of slices is coded into TRSs* by transforming the rules such that: (i) the original behavior of the TRS can be reproduced, and (ii) the 'slice' and 'difference' arguments of the wrappers are manipulated in accordance with *Slice*$*$.

Figure 7 below sketches the global organization of our approach. For any reduction $\rho : C \longrightarrow^* D$, and $\langle C_1, D_1\rangle, \langle C_2, D_2\rangle \in$ *Slice*$*^{\rho}$, we will *transform* C into an equivalent context C' where each symbol is surrounded by a wrapper with the appropriate information. In the transformed TRS, a reduction $\rho' : C' \longrightarrow^* D'$ will exist. The steps in ρ' are either rewrite steps equivalent to steps in ρ, or 'administrative' steps for manipulating slice and difference information. Context D can be extracted from D' using a simple transformation. More important is the fact that for every subcontext of D, its slice can be extracted from the corresponding subcontext of D'.

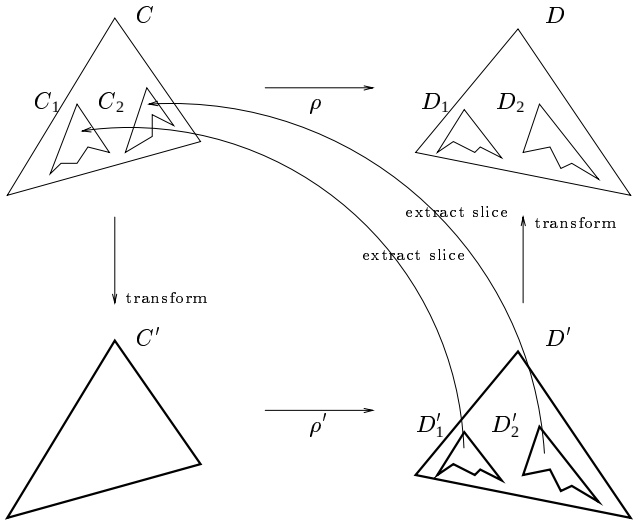


Fig. 7. Global organization of the implementation of *Slice*$*$.

Slices and difference information are represented by *context domains* (i.e. sets of paths). Details of the requisite administrative functions for manipulating this information depend on the actual set representation, and are irrelevant for the purposes of this paper. Although sets can be represented as terms, and the administrative steps as rewrite steps, recent experiments show that implementing these steps by non-rewriting means is more efficient.

Figure 8 shows the transformed version $\mathbf{B}'$ of system **B** of Figure 3. In this system, *Union*, *Add*, *Collect*, and *Strip* perform administrative steps: *Union* denotes set union on context domains, *Add* updates the difference information of the root of a term, *Merge* merges the information in the bindings of a non-linearly matched variables, *Collect* recursively collects the information in such variables, and *Strip* derives the corresponding **B**-term from a $\mathbf{B}'$-term by removing all slice information. Details of these operations are discussed in [9].

$$
\begin{array}{ll}
\textbf{[B1}'\textbf{]} & X\ \langle\wedge, V_{()}, W_{()}\rangle\ (Y\ \langle\oplus, V_{(2)}, W_{(2)}\rangle\ Z) \longrightarrow \\
& \quad (X\ \langle\wedge, G, G\rangle\ Y)\ \langle\oplus, G, G\rangle\ (X\ \langle\wedge, G, G\rangle\ Z) \\
& \quad \textbf{where}\ G = \mathit{Union}(\mathit{Union}(V_{()},\ V_{(2)}),\ W_{(2)}) \\
\textbf{[B2}'\textbf{]} & X\ \langle\wedge, V_{()}, W_{()}\rangle\ \langle\mathrm{tt}, V_{(2)}, W_{(2)}\rangle \longrightarrow \mathit{Add}(X,\ \mathit{Union}(V_{()},\ \mathit{Union}(V_{(2)},\ W_{(2)}))) \\
\textbf{[B3}'\textbf{]} & X\ \langle\wedge, V_{()}, W_{()}\rangle\ \langle\mathrm{ff}, V_{(2)}, W_{(2)}\rangle \longrightarrow \langle\mathrm{ff}, \mathit{Union}(V_{()},\ \mathit{Union}(V_{(2)},\ W_{(2)})), W_{()}\rangle \\
\textbf{[B4}'\textbf{]} & X\ \langle\oplus, V_{()}, W_{()}\rangle\ X' \longrightarrow \langle\mathrm{ff}, \mathit{Union}(V_{()},\ \mathit{Collect}(X,\ X',\ V_{()},\ V_{()})), W_{()}\rangle \\
& \quad \textbf{when}\ \mathit{Strip}(X) = \mathit{Strip}(X')
\end{array}
$$

Fig. 8. Transformed Boolean TRS $\mathbf{B}'$.

The **B**-term $\mathrm{ff} \wedge (\mathrm{tt} \oplus \mathrm{tt})$ will be transformed to the $\mathbf{B}'$-term:

$$\langle\mathrm{ff}, \{\ (1)\ \}, \emptyset\rangle\ \langle\wedge, \{\ ()\ \}, \emptyset\rangle\ (\langle\mathrm{tt}, \{\ (2\ 1)\ \}, \emptyset\rangle\ \langle\oplus, \{\ (2)\ \}, \emptyset\rangle\ \langle\mathrm{tt}, \{\ (2\ 2)\ \}, \emptyset\rangle)$$

Reduction of this term according to system $\mathbf{B}'$ results in the term:

$$\langle\mathrm{ff}, \{\ (),\ (2),\ (2\ 1),\ (2\ 2)\ \}, \emptyset\rangle$$

The "context" domain of this result corresponds to the slice $\bullet \wedge (\mathrm{tt} \oplus \mathrm{tt})$ we computed previously. In general, the domain of a slice with respect to a context is equal to the union of the function symbol domains and the difference domains for each node in the criterion, except that the difference domain at the *root node* of the criterion is omitted.

For each reduction step in the untransformed TRS, there is a corresponding non-administrative step in the transformed TRS. Administrative steps are bounded by the size of domain sets, which are in turn bounded by the size of the initial term. Each union operation can be implemented efficiently using bit vectors, whose size is proportional to the size of the initial term. The number of unions per reduction step is bounded by the number of function symbols that need to be matched. Consequently, the overhead is *linear* in the size of the initial term. We have implemented dependence tracking in the ASF+SDF system [11]; the overhead required to compute slices has proved quite tolerable in practice.

8 Related Work

The term "slice" was first coined by Weiser [19], and defined for imperative programming languages using dataflow analysis. Subsequent work, beginning with that of Ottenstein and Ottenstein [15], has focused on use of *program dependence graphs* [6] for computing slices. Cartwright and Felleisen [4] and Venkatesh [17] discuss the denotational foundations of dependence and slicing, respectively for similar classes of languages; however, they do not provide an operational means to *compute* slices. [16] provides a survey of current work on program slicing.

A number of authors have considered various "labeling" or "tracking" schemes which propagate auxiliary information in conjunction with reduction systems; these schemes are similar in some respects to the method we will use to implement slicing. Bertot [2, 3] defines an *origin function*, which is a generalization of the classic notions of residual and descendant in the lambda-calculus and TRSs. He applies this idea to the implementation of source-level program debuggers for languages implemented using *natural semantics* [10]. Van Deursen, Klint and Tip [5], addressing similar problems,

define a slightly expanded class of "origin" information for the larger class of *conditional* TRSs. However, slicing is not considered in these papers, nor do these "tracking" algorithms propagate information appropriate for computing slices.

In [12] (page 85), Klop presents a "tracing relation" which is very similar to our dynamic dependence notion, and observes that it can be used to distinguish the *needed prefix* and the *non-needed part* of a term. In our terminology, the needed part is the slice with respect to the entire normal form, and the non-needed parts correspond to the "holes" in this slice. In other words, replacing the non-needed parts by arbitrary subterms will result in the same normal form. There are two main differences with our work. First, Klop's tracing relation is only defined for *orthogonal* TRSs. This ensures that the normal form resulting from replacing non-needed parts is *exactly* the same as the normal form of the original term. Second, for collapse rules the top symbol of the reduct is considered to be "created". As we discussed earlier, this gives rise to slices being non-minimal. Finally, Klop does not study the use of tracing relations for program slicing, nor does he give an algorithm to compute his relation efficiently in practice.

In certain respects, our technique is the dual of *strictness analysis* in lazy functional programming languages, particularly the work of Wadler and Hughes [18] using *projections*. Strictness analysis is used to characterize those subcomponents of a function's input domain that are always needed to compute a result; we instead determine subcomponents of a *particular* input that are *not* needed. However, there are significant differences: strictness analysis is concerned with domain-theoretic *approximations* of values, usually requires computation by fixpoint iteration, and rarely addresses more than a few core functional primitives. By contrast, we perform exact analysis on a particular input (although we can effectively perform some approximate analyses by reduction of open terms), compute our results algebraically, and can address any construct expressible in TRS form.

Maranget [13, 14] provides a comprehensive study of lazy and optimal reductions in orthogonal TRSs using labeled terms. Although Maranget's label information could in principle be used to compute slices, he does not discuss such an application, nor does he provide any means by which such labels could be used to implement slicing. Like Klop, Maranget also only considers *orthogonal* TRSs. Our approach covers a larger class of TRSs, and provides a purely *relational* definition of slice which does not require labeling.

9 Future Work

An important question for future work is to define classes of TRSs for which slices are independent of the reduction actually used. While orthogonal systems certainly have this property, we believe it should be possible to characterize non-orthogonal systems for which this property also holds.

Acknowledgments

We are grateful to G. Ramalingam and Jan Heering for commenting on earlier drafts of this paper.

References

1. ACETO, L., BLOOM, B., AND VAANDRAGER, F. Turning SOS rules into equations. In *Proc. IEEE Symp. on Logic in Computer Science* (Santa Cruz, CA, June 1992), pp. 113–124.
2. BERTOT, Y. Occurrences in debugger specifications. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation* (Toronto, June 1991), pp. 327–336.
3. BERTOT, Y. Origin functions in λ-calculus and term rewriting systems. In *Proc. Seventeenth CAAP* (1992), J.-C. Raoult, Ed., pp. 49–64. (Springer-Verlag LNCS 581).
4. CARTWRIGHT, R., AND FELLEISEN, M. The semantics of program dependence. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (Portland, OR, 1989), pp. 13–27.
5. DEURSEN, A. VAN, KLINT, P., AND TIP, F. Origin tracking. *J. Symbolic Computation 15* (1993), 523–545.
6. FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems 9*, 3 (July 1987), 319–349.
7. FIELD, J. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proc. Seventeenth ACM Symp. on Principles of Programming Languages* (San Francisco, January 1990), pp. 1–15.
8. FIELD, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (San Francisco, June 1992), pp. 98–107. Published as Yale University Technical Report YALEU/DCS/RR–909.
9. FIELD, J., AND TIP, F. Dynamic dependence in term rewriting systems and its application to program slicing. Report CS-R94xx, CWI, Amsterdam, 1994. Forthcoming.
10. KAHN, G. Natural semantics. In *Fourth Annual Symp. on Theoretical Aspects of Computer Science* (1987), vol. 247 of *LNCS*, Springer-Verlag, pp. 22–39.
11. KLINT, P. A meta-environment for generating programming environments. *ACM Trans. on Software Engineering and Methodology 2*, 2 (1993), 176–201.
12. KLOP, J. Term rewriting systems. Tech. Rep. CS-R9073, CWI, Amsterdam, The Netherlands, 1990.
13. MARANGET, L. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proc. Eighteenth ACM Symp. on Principles of Programming Languages* (Orlando, FL, January 1991), pp. 255–269.
14. MARANGET, L. *La Stratégie Paresseuse*. PhD thesis, Université de Paris VIII, 1992. (in French).
15. OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. The program dependence graph in a software development environment. In *Proc. ACM SIGPLAN/SIGSOFT Symp. on Practical Programming Development Environments* (April 1984), pp. 177–184. SIGPLAN Notices 19(5).
16. TIP, F. A survey of program slicing methods. Forthcoming CWI technical report, 1994.
17. VENKATESH, G. The semantic approach to program slicing. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (Toronto, June 1991), pp. 80–91.
18. WADLER, P., AND HUGHES, R. Projections for strictness analysis. In *Proc. Conf. on Functional Programming and Computer Architecture* (Portland, OR, September 1987), pp. 385–406. (Springer-Verlag LNCS 274).
19. WEISER, M. Program slicing. *IEEE Trans. on Software Engineering SE-10*, 4 (1989), 352–357.

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