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

John Field and Frank Tip

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

Abstract

Program slicing is a useful technique for debugging, testing, and analyzing programs. A program slice consists of the parts of a program that (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.

Viewed more abstractly, our techniques yield a method for automatically deriving certain minimal equational theorems on *open* terms as a consequence of deriving a single theorem about a closed term. Our techniques can thus be used to augment the capabilities of equational theorem proving systems.

Key Words & Phrases: program slicing, term rewriting, dependence analysis, origin tracking.

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 [32], 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 [29], 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:

*Some of the material in this paper appeared in preliminary form in [15] and [27].

- 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 notion of dependence does not require labeled terms [6, 7, 22, 23], and is distinguished by its ability to treat (normally problematic) TRSs with left-nonlinear rules.

- Our notion of slicing is more general than those defined in previous work. 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 [30], he does not indicate how to *compute* such slices.

- We describe the implementation of a practical algorithm to perform dynamic dependence tracking. The algorithm operates by systematically transforming the original TRS to obtain an "instrumented" version which gathers dependence information during the reduction process.

- Finally, for the case of left-linear systems, we present proofs that our definitions yield *minimal* and *sound* slices.

In a companion paper [14], we show how extensions of the techniques discussed here can be used to implement slicing in a standard programming language, and compare these techniques to other algorithms in the literature. In [28], the dynamic dependence relation defined in this paper is used for providing dynamic slicing facilities in generic source-level debugging tools. In this paper, we will concentrate primarily on technical foundations.

1.2 Motivating Examples

Consider the program in Figure 1A. It is written in a tiny imperative programming language, **P**, whose syntax is given in Figure 2. While extremely limited in its computational power, **P** contains constructs that are representative of features found in real language:

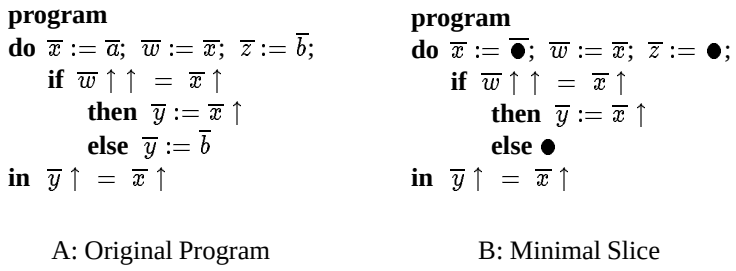
- Expressions of the form $\overline{x}$ are atoms, and play the dual role of basic values and addresses which may be assigned to using ':='. The distinguished atoms $\overline{t}$ and $\overline{f}$ represent boolean values.

- Addresses must be explicitly dereferenced using '$\uparrow$' to yield the value associated with the address.

- A **P do** statement is executed by first evaluating its compound statement operand, which has the effect of assigning values to one or more variables. Those values are then used to evaluate the **in** expression. (Note that this is *not* a loop construct, but is executed only once.)

- Conditional (**if**) constructs come in two forms: one that operates on statements, and one that operates on expressions.

We evaluate **P** programs by applying the *rewriting rules* of Figure 3 to the *term* consisting of the program's syntax tree until no further rules are applicable. This reduction process produces a sequence of



A: Original Program B: Minimal Slice

Figure 1: Example **P** Program.

terms ending with a *normal form* that denotes the result of the evaluation. The program in Figure 1A, for instance, 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 '$\bullet$' represents subterms of the program that do not affect its result, a concept that we will formalize in the sequel. It should be clear that a program slice is valuable for understanding which program components the slicing criterion depends on to compute its value. 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 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 of Bergstra et al. [5], we can compute both traditional "static" and "dynamic" [29] slices for the same language. Details of how this can be done for a realistic language (a subset of C) may be found in [14].

In addition to applications in program development tools, we believe that our notion of a slice should 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 4, which defines a few boolean identities ('$\wedge$' denotes conjunction, '$\oplus$' exclusive-or). Figure 5 shows how **B**-term $\mathrm{ff} \wedge (\mathrm{tt} \oplus \mathrm{tt})$ can be reduced to ff. Observe that in deriving the theorem $\mathrm{ff} \wedge (\mathrm{tt} \oplus \mathrm{tt}) = \mathrm{ff}$, we actually derive the *more general* theorem $P \wedge (\mathrm{tt} \oplus \mathrm{tt}) = \mathrm{ff}$, for arbitrary P. From the point of view of slicing, the slice with respect to the normal form ff is the *subcontext* $\bullet \wedge (\mathrm{tt} \oplus \mathrm{tt})$ of the initial term (we will define the notion of context precisely in the sequel). To determine such a slice, we must pay careful attention to the behavior of *nonlinear* rules such as **[B4]** and **[P1]**, which many authors on reduction-theoretic properties of TRSs do not treat. In the sequel, we show how slices can be obtained by examining the manner in which rules *create* new function symbols, and *residuate*, or "move around" old ones.

Note that the reduction of Figure 5 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. Although in general slices may differ depending on the particular reduction used, for the particularly well-behaved class of *orthogonal* TRSs [21], it is easy to show that the slices computed are always the same regardless of the order in which rules are

(atom tags)	X	::=	t
		$\mid$	f
		$\mid$	a
		$\mid$	b
		$\vdots$	$\vdots$
(expressions)	E	::=	$\overline{X}$
		$\mid$	$E\uparrow$
		$\mid$	$E = E$
		$\mid$	**if** E **then** E **else** E
		$\mid$	**do** S **in** E
(simple statements)	L	::=	$E := E$
		$\mid$	**if** E **then** S **else** S
(compound statements)	S	::=	L
		$\mid$	$S; L$
(programs)	P	::=	**program** E

Figure 2: Syntax of **P**

[P1]	$\overline{X} = \overline{X}$	$=$	$\overline{t}$
[P2]	$\overline{a} = \overline{b}$	$=$	$\overline{f}$ for all constants a, b such that $a \neq b$
[P3]	**if** $\overline{t}$ **then** E_1 **else** E_2	$=$	E_1
[P4]	**if** $\overline{f}$ **then** E_1 **else** E_2	$=$	E_2
[P5]	**if** $\overline{t}$ **then** S_1 **else** S_2	$=$	S_1
[P6]	**if** $\overline{f}$ **then** S_1 **else** S_2	$=$	S_2
[P7]	**do** S **in** $\overline{X}$	$=$	$\overline{X}$
[P8]	**do** S **in** $E_1 = E_2$	$=$	(**do** S **in** E_1) = (**do** S **in** E_2)
[P9]	**do** S **in if** E_1 **then** E_2 **else** E_3	$=$	**if** (**do** S **in** E_1) **then** (**do** S **in** E_2) **else** (**do** S **in** E_3)
[P10]	**do** $E_1 := E_2$ **in** $(E_3 \uparrow)$	$=$	**if** (**do** $E_1 := E_2$ **in** E_3) = E_1 **then** E_2 **else** ((**do** $E_1 := E_2$ **in** E_3) $\uparrow$)
[P11]	**do** $S; E_1 := E_2$ **in** $(E_3 \uparrow)$	$=$	**if** (**do** $S; E_1 := E_2$ **in** E_3) = (**do** S **in** E_1) **then** (**do** S **in** E_2) **else do** S **in** ((**do** $E_1 := E_2$ **in** E_3) $\uparrow$)
[P12]	**do if** E **then** S_1 **else** S_2 **in** E	$=$	**if** E **then** (**do** S_1 **in** E) **else** (**do** S_2 **in** E)
[P13]	**do** S_1; **if** E_1 **then** S_2 **else** S_3 **in** E_2	$=$	**if** (**do** S_1 **in** E_1) **then** (**do** $S_1; S_2$ **in** E_2) **else** (**do** $S_1; S_3$ **in** E_2)
[P14]	**program** $\overline{X}$	$=$	**result** $\overline{X}$

Figure 3: Rewriting Semantics of **P**.

[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]**	$X \oplus X \longrightarrow \text{ff}$

Figure 4: Boolean TRS **B**.

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

Figure 5: A **B**-reduction; redexes are underlined.

applied.

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 example, if $f(g(a,b),\ c)$ is a term, then one of several contexts contained in T is $g(\bullet, b)$. The context 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 a context C is a subcontext of a term T by $C \sqsubseteq T$; naturally, contexts as well as terms may contain subcontexts.

In a slice, holes denote subterms that are irrelevant to the computation of the criterion. Informally, this means that replacing any hole in the slice would still allow the original criterion to be produced by a "subreduction" derived from the original reduction. Definition 1.1 below (which is rendered pictorially in Figure 6) makes these ideas precise.

Definition 1.1 (Slice) *Let $\rho : T \longrightarrow^* T'$ be a reduction. Then a* slice *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 $D' \sqsupseteq E'$, $E' \doteq C'$, and $\langle C, \rho', D' \rangle \in$ Project$*^{\rho}$. Slice C is* minimal *if there is no slice with respect to criterion C' that contains fewer function symbols.*

Project$*^{\rho}$ denotes the set of subreductions derived from ρ. Such sets contain collections of triples of the form $\langle C, \rho', C' \rangle$. Roughly, 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 will discuss *Project*$*^{\rho}$ further in Section 5. The operator '$\doteq$' is used to indicate that two contexts are isomorphic (but may be "rooted" in different terms or different subterms of the same term).

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 [2, 13]. Bergstra et al. [5] show how many traditional program constructs may be modeled equationally, and implemented using a TRS.

2 Basic Definitions

In this section, we make precise the notion of a *context* introduced informally in the previous section. This idea will be the cornerstone of our formalization of slicing and dependence. Instead of deriving contexts

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

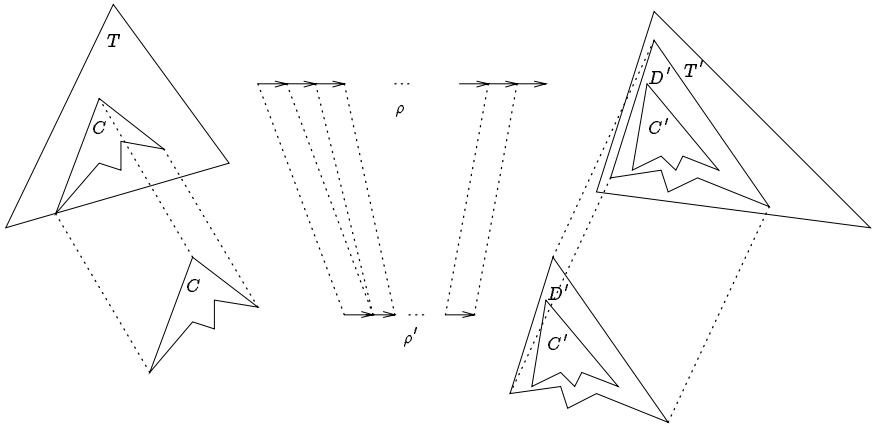


Figure 6: Depiction of the definition of a slice.

from terms, 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 $p \prec q$. If $p \preceq q$, then $q \dot{-} p$ denotes the path r such that $p \cdot r = q$. If neither $p \preceq q$ nor $q \preceq p$ then p and q are *disjoint*, notated $p \mid q$.

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* $i \leq \mathit{arity}(s)$.

(ii) *If* $\mathcal{P} \neq \emptyset$, *then* $p = \mathit{root}(\mathcal{P})$.

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 $C = \langle p, \mu \rangle$, $\mathit{root}(C)$ denotes the path p, and $\mathcal{O}(C)$ the domain of μ. We will use $\mathit{Cont}(\Sigma, \mathcal{V})$ to denote the set of all $\Sigma\mathcal{V}$-contexts.

Given a path p and context C, if either C is empty and $p = \mathit{root}(C)$, or if $p \notin \mathcal{O}(C)$ and there exists path $q \in \mathcal{O}(C)$ such that $p = q \cdot i$, $\mu(q) = f$, and $i \leq \mathit{arity}(f)$, then we will refer to p as a *hole occurrence* of C. A hole occurrence thus corresponds to a child "missing" from a context. The set of hole occurrences in a context C will be denoted by $\mathcal{O}_\bullet(C)$.

We will use the operator '$\equiv$' to denote identity of 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 useful for defining contexts textually; e.g., $p \leftarrow f(\bullet, g(a, \bullet))$ represents a context rooted at p with two holes, binary function symbols f and g and a constant a. $p \leftarrow \bullet$ represents an empty context rooted at p. We will say that contexts C and D are *isomorphic*, notated $C \doteq D$, if $((\ ) \leftarrow C) \equiv ((\ ) \leftarrow D)$

A context C is a *term* if: (i) C has no hole occurrences, and (ii) $\mathit{root}(C) = ()$. Although the restriction on the root of C is not strictly necessary, it results in a definition that agrees most closely with that used by other authors. We will use $\mathit{Term}(\Sigma, \mathcal{V})$ to denote the set of $\Sigma\mathcal{V}$-terms. 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.

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 $\mathit{vars}(C)$, and $\mathit{vars}_1(C)$ 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, $\mathit{root}(C) = q \cdot i \in \mathcal{O}(D)$, 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 $\mathit{root}(D) \in (\mathcal{O}(C) \cup \mathcal{O}_\bullet(C))$, $C[D]$ denotes the context C where the subcontext at $\mathit{root}(D)$ is *replaced* by D. A context C is *elementary* iff $|\mathcal{O}(C)| = 1$.

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 7.

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, L does not consist of a sole variable, 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 5 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 7, 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 Context Rewriting

In order to generalize term rewriting to context rewriting, a few auxiliary definitions are needed. 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. 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 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.

3.3 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*

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

An example of *VarPairs* will be presented in Section 4.1.

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 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*

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

The reflexive, transitive closure of Resid is defined by

$$\begin{aligned} \mathit{Resid}*^{\epsilon} &\triangleq \{\, \langle C, C \rangle \mid C \in \mathit{Cont}(\Sigma)\ \} \\ \mathit{Resid}*^{\rho\mathcal{A}} &\triangleq \mathit{Resid}*^{\rho} \cdot \mathit{Resid}^{\rho\mathcal{A}} \end{aligned}$$

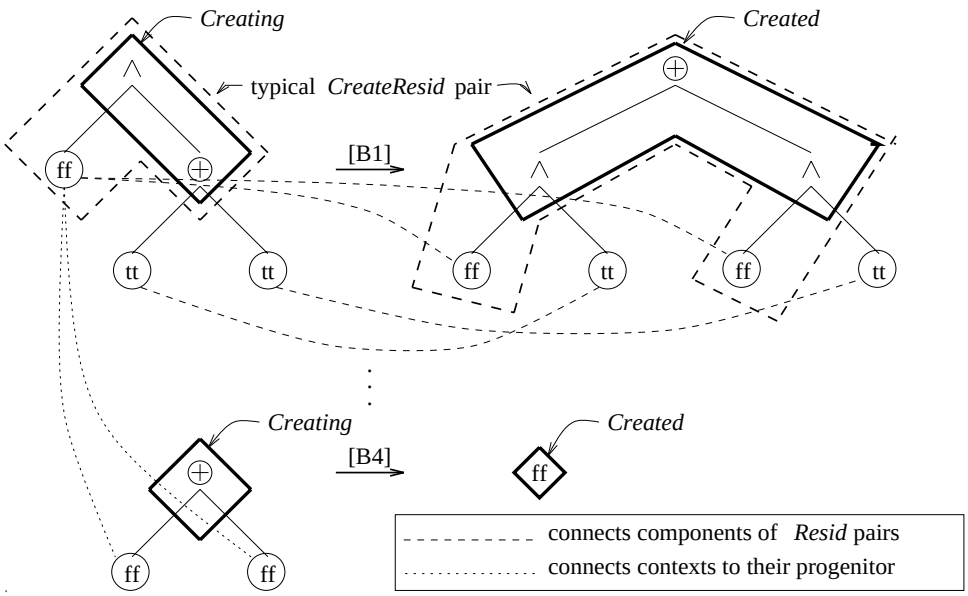


Figure 7: Illustration of selected relations and contexts derived from **B**-reduction of Figure 5.

Figure 7 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 5, 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 $\langle \mathcal{S}_1, \mathcal{S}_2 \rangle \in \mathit{VarPairs}^{\rho,\mathcal{A}}$.

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

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

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

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*

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

Note that it is *impossible* to have both $\langle C_1, D\rangle \in \mathit{Resid}^{\rho\mathcal{A}}$ and $\langle C_2, D\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$, for any nonempty C_1, C_2, D; these relations may, however, overlap on empty contexts.

4 A Dynamic Dependence Relation

In this section, we will derive our dynamic dependence relation, *Slice*$*$, using the concepts introduced in Section 3. 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 $\langle C, D'\rangle \in (\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}})$; then the slice for this C in reduction ρ is determined. Operation '$\cdot$' in Definition 4.1 denotes relational join.

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

$$\begin{aligned} \mathit{Slice}*^{\epsilon} &\triangleq \{\, \langle C, C\rangle \mid C \in \mathit{Cont}(\Sigma)\,\} \\ \mathit{Slice}*^{\rho\mathcal{A}} &\triangleq \mathit{Slice}*^{\rho} \cdot \{\, \langle C, D\rangle \mid \textit{there exists a minimal } D' \sqsupseteq D \\ &\qquad\qquad \textit{such that } \langle C, D'\rangle \in (\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}})\,\} \end{aligned}$$

Since $\mathit{Resid}^{\rho\mathcal{A}}$ and $\langle C_2, D\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ only overlap for empty contexts, it is easy to see that the slice with respect to any nonempty criterion is uniquely defined. Empty contexts may have multiple slices, which arise from the application of collapse rules.

4.1 Example

In the example that follows, we will frequently use set comprehension to avoid unwieldy notation. We will consider the following **B**-reduction $\rho = \mathcal{A}_1\mathcal{A}_2\mathcal{A}_3$:

$$S = (\mathrm{ff} \wedge \underline{(\mathrm{ff} \wedge \mathrm{tt})}) \wedge \mathrm{tt} \xrightarrow{\mathcal{A}_1} \underline{(\mathrm{ff} \wedge \mathrm{ff})} \wedge \mathrm{tt} \xrightarrow{\mathcal{A}_2} \mathrm{ff} \wedge \mathrm{tt} \xrightarrow{\mathcal{A}_3} \mathrm{ff} = T$$

Note that for contraction $\mathcal{A}_1$, an application of rule **B2**, we have $P_{\mathcal{A}_1} = (1\ 2)$, $\overline{L_{\mathcal{A}_1}} = (1\ 2) \leftarrow X \wedge \mathrm{tt}$, $\overline{R_{\mathcal{A}_1}} = (1\ 2) \leftarrow X$, $\mathit{Redex}_{\mathcal{A}_1} = (1\ 2) \leftarrow \mathrm{ff} \wedge \mathrm{tt}$, and $\mathit{Reduct}_{\mathcal{A}_1} = (1\ 2) \leftarrow \mathrm{ff}$. This results in the following relations

for $\mathcal{A}_1$:

$$
\begin{array}{lcl}
\mathit{VarPairs}^{\mathcal{A}_1} & = & \{\ \langle \{\ (1\ 2\ 1) \leftarrow \mathrm{ff}\ \}, \{\ (1\ 2) \leftarrow \mathrm{ff}\ \}\rangle,\ \langle \{\ (1\ 2\ 1) \leftarrow \bullet\ \}, \{\ (1\ 2) \leftarrow \bullet\ \}\rangle\ \} \\
\mathit{Resid}^{\mathcal{A}_1} & = & \{\ \ \langle (1\ 2\ 1) \leftarrow \mathrm{ff}, (1\ 2) \leftarrow \mathrm{ff}\rangle,\ \langle (1\ 2\ 1) \leftarrow \bullet, (1\ 2) \leftarrow \bullet\rangle,\ \langle () \leftarrow \bullet, () \leftarrow \bullet\rangle \\
 & & \quad \langle (1\ 2) \leftarrow \bullet, (1\ 2) \leftarrow \bullet\rangle\ \} \cup \{\ \langle C, C\rangle \mid C \sqsubseteq () \leftarrow (\mathrm{ff} \wedge \bullet) \wedge \mathrm{tt}\ \} \\
 & & \\
\mathit{Creating}^{\mathcal{A}_1} & = & (1\ 2) \leftarrow (\bullet \wedge \mathrm{tt}) \\
\mathit{Created}^{\mathcal{A}_1} & = & (1\ 2) \leftarrow \bullet \\
 & & \\
\mathit{CreateResid}^{\mathcal{A}_1} & = & \{\ \ \langle (1\ 2) \leftarrow (\bullet \wedge \mathrm{tt}), (1\ 2) \leftarrow \bullet\rangle,\ \langle (1) \leftarrow \bullet \wedge (\mathrm{ff} \wedge \mathrm{tt}), (1) \leftarrow \bullet \wedge \mathrm{ff}\rangle, \\
 & & \quad \langle (1) \leftarrow \mathrm{ff} \wedge (\mathrm{ff} \wedge \mathrm{tt}), (1) \leftarrow \mathrm{ff} \wedge \mathrm{ff}\rangle, \\
 & & \quad \langle () \leftarrow (\mathrm{ff} \wedge (\mathrm{ff} \wedge \mathrm{tt})) \wedge \bullet, () \leftarrow (\mathrm{ff} \wedge \mathrm{ff}) \wedge \bullet\rangle, \\
 & & \quad \langle () \leftarrow (\mathrm{ff} \wedge (\mathrm{ff} \wedge \mathrm{tt})) \wedge \mathrm{tt}, () \leftarrow (\mathrm{ff} \wedge \mathrm{ff}) \wedge \mathrm{tt}\rangle\ \}
\end{array}
$$

For contraction $\mathcal{A}_2$, which is an application of **B3**, we have $P_{\mathcal{A}_2} = (1)$, $\overline{L_{\mathcal{A}_2}} = (1) \leftarrow X \wedge \mathrm{ff}$, $\overline{R_{\mathcal{A}_2}} = (1) \leftarrow \mathrm{ff}$, $\mathit{Redex}_{\mathcal{A}_2} = (1) \leftarrow \mathrm{ff} \wedge \mathrm{ff}$, and $\mathit{Reduct}_{\mathcal{A}_2} = (1) \leftarrow \mathrm{ff}$. Therefore, we have:

$$
\begin{array}{lcl}
\mathit{VarPairs}^{\mathcal{A}_1\mathcal{A}_2} & = & \{\ \langle \{\ (1\ 1) \leftarrow \mathrm{ff}\ \}, \emptyset\rangle,\ \langle \{\ (1\ 1) \leftarrow \bullet\ \}, \emptyset\rangle\ \} \\
\mathit{Resid}^{\mathcal{A}_1\mathcal{A}_2} & = & \{\ \langle () \leftarrow \bullet, () \leftarrow \bullet\rangle,\ \langle (1) \leftarrow \bullet, (1) \leftarrow \bullet\rangle\ \} \cup \{\ \langle C, C\rangle \mid C \sqsubseteq () \leftarrow \bullet \wedge \mathrm{ff}\ \} \\
 & & \\
\mathit{Creating}^{\mathcal{A}_1\mathcal{A}_2} & = & (1) \leftarrow (\bullet \wedge \mathrm{ff}) \\
\mathit{Created}^{\mathcal{A}_1\mathcal{A}_2} & = & (1) \leftarrow \mathrm{ff} \\
 & & \\
\mathit{CreateResid}^{\mathcal{A}_1\mathcal{A}_2} & = & \{\ \ \langle (1) \leftarrow (\bullet \wedge \mathrm{ff}), (1) \leftarrow \mathrm{ff}\rangle,\ \langle () \leftarrow (\bullet \wedge \mathrm{ff}) \wedge \bullet, () \leftarrow \mathrm{ff} \wedge \bullet\rangle, \\
 & & \quad \langle () \leftarrow (\bullet \wedge \mathrm{ff}) \wedge \mathrm{tt}, () \leftarrow \mathrm{ff} \wedge \mathrm{tt}\rangle\ \}
\end{array}
$$

For the third contraction, $\mathcal{A}_3$, an application of **B2**, we have $P_{\mathcal{A}_3} = ()$, $\overline{L_{\mathcal{A}_3}} = () \leftarrow X \wedge \mathrm{tt}$, $\overline{R_{\mathcal{A}_3}} = () \leftarrow X$, $\mathit{Redex}_{\mathcal{A}_3} = () \leftarrow \mathrm{ff} \wedge \mathrm{tt}$, and $\mathit{Reduct}_{\mathcal{A}_3} = () \leftarrow \mathrm{ff}$. The following relations are computed for $\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3$:

$$
\begin{array}{lcl}
\mathit{VarPairs}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & \{\ \langle \{\ (1) \leftarrow \mathrm{ff}\ \}, \{\ () \leftarrow \mathrm{ff}\ \}\rangle,\ \langle \{\ (1) \leftarrow \bullet\ \}, \{\ () \leftarrow \bullet\ \}\rangle\ \} \\
\mathit{Resid}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & \{\ \langle (1) \leftarrow \mathrm{ff}, () \leftarrow \mathrm{ff}\rangle,\ \langle (1) \leftarrow \bullet, () \leftarrow \bullet\rangle,\ \langle () \leftarrow \bullet, () \leftarrow \bullet\rangle\ \} \\
 & & \\
\mathit{Creating}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & () \leftarrow (\bullet \wedge \mathrm{tt}) \\
\mathit{Created}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & () \leftarrow \bullet \\
 & & \\
\mathit{CreateResid}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} & = & \{\ \langle () \leftarrow (\bullet \wedge \mathrm{tt}), () \leftarrow \bullet\rangle\ \}
\end{array}
$$

From the above and Definition 4.1 it follows that we have the following dynamic dependence relations between subcontexts of S and T:

$$
\mathit{Slice}*^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} = \{\ \langle (1) \leftarrow \bullet \wedge (\mathrm{ff} \wedge \mathrm{tt}), () \leftarrow \mathrm{ff}\rangle,\quad \langle () \leftarrow \bullet, () \leftarrow \bullet\rangle\ \}
$$

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 subreduction consists of the first two contractions. We will formalize the notion of a subreduction in Section 5.

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 [21]), 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 Projections, Subreductions

In this section, we formalize the notion of a *projection* of some reduction on a subcontext of its initial term. It will be convenient to define simultaneously the initial and final contexts to which a projection corresponds along with the the projected "subreduction" itself. We therefore define the set of *projection triples* as follows:

Definition 5.1 (Projection Triples) *Let $\mathcal{R}$ be a TRS over signature Σ. Then the set of $\mathcal{R}$* projection triples *is inductively defined as follows:*

$$
\begin{array}{lcll}
\mathit{Project}{*}^{\epsilon} & \triangleq & \{\, \langle B, \epsilon, B\rangle \mid B \in \mathit{Cont}(\Sigma) \,\} & \\
\mathit{Project}{*}^{\rho,\mathcal{A}} & \triangleq & \{\langle B, \sigma\mathcal{A}, D'\rangle \mid \langle B, \sigma, C\rangle \in \mathit{Project}{*}^{\rho},\ \langle C, D\rangle \in \mathit{CreateResid}^{\rho,\mathcal{A}},\ D' \sqsubseteq D \,\} \cup & \text{(i)} \\
 & & \{\langle B, \sigma, D'\rangle \mid \langle B, \sigma, C\rangle \in \mathit{Project}{*}^{\rho},\ \langle C, D\rangle \in \mathit{Resid}^{\rho,\mathcal{A}},\ D' \sqsubseteq D \,\} & \text{(ii)}
\end{array}
$$

The interesting cases in Definition 5.1 are numbered. Intuitively, these cases behave as follows:

- In case (i), the context D' that constitutes the third element of the triple is entirely contained in a context D that is involved in a $\mathit{CreateResid}^{\rho,\mathcal{A}}$–relation. In this case, contraction $\mathcal{A}$ is deemed applicable to D', and the construction continues recursively with the context C that contracted to D, and reduction ρ.

- In case (ii), D' is a subcontext of some context D that residuated from a context C. In this case, contraction $\mathcal{A}$ was not applicable to D', and the construction continues recursively with the context C from which D residuated, and reduction ρ.

Note that *each* residual D of a context C gives rise to the construction of a new projection triple. This reflects the fact that different residuals of a context may be reduced differently, causing the construction of different subreductions.

Informally, the occurrence of a triple $\langle B, \sigma, D'\rangle$ in relation $\mathit{Project}{*}^{\rho}$ indicates that context B reduces to a context D that "contains" D'. Moreover, it does so by a reduction σ that is composed of a subset of the contractions in the original reduction ρ. In Section 6, we will prove this property of projections. In addition, we will show that $\mathit{Slice}{*}^{\rho}$ computes slices that correspond to minimal projections by effectively selecting the minimal supercontext D of D' (in each construction step) for which there exists a pair $\langle C, D\rangle$ in $(\mathit{Resid}^{\rho,\mathcal{A}} \cup \mathit{CreateResid}^{\rho,\mathcal{A}})$.

As an example of the behavior of *Project*$*$, consider the **B**-reduction in Fig. 8. As usual, we have underlined each redex. We use $\mathcal{A}_{[\mathrm{B1}]}$ and $\mathcal{A}_{[\mathrm{B3}]}$ to denote the contractions that use rules **[B1]** and **[B3]**,

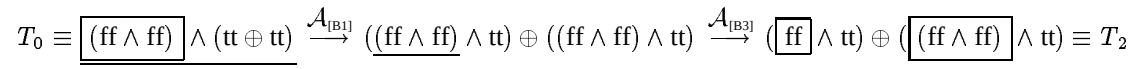


Figure 8: Example of projections.

respectively. Some typical, minimal elements of the set $\mathit{Project}_*^{\mathcal{A}_{[B1]}\mathcal{A}_{[B3]}}$ are:

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

Observe that the last two of these projection triples 'apply to' the subcontext $(1) \leftarrow (\mathrm{ff} \wedge \mathrm{ff})$ of T_0; this subcontext is shown boxed in Fig. 8. The projections of the boxed subcontext of T_0 are also shown boxed (in T_2). Clearly, these triples correspond to the two different "paths through the reduction" taken by the boxed subterm of T_0. One residual is contracted in a subsequent step, the other is not.

6 Formal Properties of Slices

We can now state some theorems describing the most important properties of slices. We will prove these theorems for left-linear TRSs only.

In order to prove that *Slice*$_*$ is a many-to-one mapping for non-empty contexts (that is, each context has a unique slice), we will first prove a few lemma's.

Lemma 6.1 *Let $B \xrightarrow{\rho}{}^* C \xrightarrow{\mathcal{A}} D$ be a reduction. Then for any non-empty $D' \sqsubseteq D$ there is at most one $C' \sqsubseteq C$ such that $\langle C', D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$. Moreover, if it exists, this C' will be non-empty.*

Proof. Let $D' \sqsubseteq D$ be a non-empty context such that $\langle C', D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$ for some $C' \sqsubseteq C$.

There are two cases:

1. ($\mathit{root}(D') \preceq \mathit{root}(\mathit{Created}^{\rho\mathcal{A}})$ or $\mathit{root}(D') \mid \mathit{root}(\mathit{Created}^{\rho\mathcal{A}})$) and D' and $\mathit{Created}^{\rho\mathcal{A}}$ are disjoint.

   Then it follows from Definition 3.2 that $C' \equiv D'$ is the unique subcontext of C such that $\langle C', D' \rangle \in \mathit{Resid}^{\rho\mathcal{A}}$. This C' is non-empty because D' is non-empty.

2. $D' \equiv (p_R \cdot q \leftarrow A)$ where $p_R \in \mathcal{O}_X(\overline{R_\mathcal{A}})$, $A \sqsubseteq (() \leftarrow \sigma_\mathcal{A}(X))$, and $q = \mathit{root}(A)$ for some variable X.

   From left-linearity it follows that there is a *unique* path p_L such that $\{\, p_L \,\} = \mathcal{O}_X(\overline{L_\mathcal{A}})$. From Definitions 3.1 and 3.2 it follows that $C' \equiv (p_L \cdot q \leftarrow A)$ is the unique subcontext of C such that $\langle C', D' \rangle \in \mathit{Resid}^{\rho'\mathcal{A}}$. Since rerooting a context does not affect its (non-)emptyness, C' will be a non-empty context. □

Lemma 6.2 *Let $B \xrightarrow{\rho}{}^{*} C \xrightarrow{\mathcal{A}} D$ be a reduction. Then for any non-empty $D' \sqsubseteq D$ there is at most one $C' \sqsubseteq C$ such that $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho,\mathcal{A}}$. Moreover, if it exists, this C' will be non-empty.*

Proof. Let $D' \sqsubseteq D$ be a non-empty context such that $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho,\mathcal{A}}$ for some $C' \sqsubseteq C$.

From Definition 3.4 it follows that there exists a unique subset R of $\mathit{Resid}^{\rho,\mathcal{A}}$ such that:

$$D' \equiv \mathit{Created}^{\rho,\mathcal{A}} \sqcup \bigsqcup \{\, E' \mid \langle E, E' \rangle \in R \,\}$$

and also that there exists a unique context

$$C' \equiv \mathit{Creating}^{\rho,\mathcal{A}} \sqcup \bigsqcup \{\, E \mid \langle E, E' \rangle \in R \,\}$$

such that $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho,\mathcal{A}}$. Since the left-hand side of a rewrite rule is not a single variable, $\mathit{Creating}^{\rho,\mathcal{A}}$ is non-empty, causing this C' to be non-empty as well. □

Lemma 6.3 *Let $B \xrightarrow{\rho}{}^{*} C \xrightarrow{\mathcal{A}} D$ be a reduction. It is impossible to have $\langle C_1, D' \rangle \in \mathit{Resid}^{\rho,\mathcal{A}}$ and $\langle C_2, D' \rangle \in \mathit{CreateResid}^{\rho,\mathcal{A}}$ for any $C_1,\ C_2 \sqsubseteq C$ and any non-empty $D' \sqsubseteq D$.*

Proof. Assume that $\langle C_2, D' \rangle \in \mathit{CreateResid}^{\rho,\mathcal{A}}$ for some $C_2 \sqsubseteq C$, and some non-empty $D' \sqsubseteq D$. From Definition 3.4 it follows that $\mathit{Creating}^{\rho,\mathcal{A}} \sqsubseteq C_2$ and $\mathit{Created}^{\rho,\mathcal{A}} \sqsubseteq D'$.

From Definitions 3.2 and 3.3 it follows that for any pair $\langle C_1, D_1 \rangle \in \mathit{Resid}^{\rho,\mathcal{A}}$ with $C_1 \sqsubseteq C$, $D_1 \sqsubseteq D$, we have that C_1 and $\mathit{Creating}^{\rho,\mathcal{A}}$ are disjoint and D_1 and $\mathit{Created}^{\rho,\mathcal{A}}$ are disjoint.

From $\mathit{Creating}^{\rho,\mathcal{A}} \sqsubseteq C_2$ and $\mathit{Creating}^{\rho,\mathcal{A}} \not\sqsubseteq C_1$ it follows that $C_2 \not\equiv C_1$. A similar argument can be used to demonstrate that $D' \not\equiv D_1$. □

Lemma 6.4 *Let $\rho : B \xrightarrow{\rho'}{}^{*} C \xrightarrow{\mathcal{A}} D$ be a reduction, and let D'' be a non-empty subcontext of D. Then there exists a unique minimal $D' \sqsubseteq D$ such that $D' \sqsupseteq D''$ and $\langle C', D' \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$ for some non-empty $C' \sqsubseteq C$. Moreover,*

$$\langle B', D'' \rangle \in \mathit{Slice}{*}^{\rho} \Leftrightarrow \langle B', D' \rangle \in \mathit{Slice}{*}^{\rho} \Leftrightarrow \langle B', C' \rangle \in \mathit{Slice}{*}^{\rho'}$$

where $B' \sqsubseteq B$.

Proof. The theorem holds trivially if $\langle C', D'' \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$, for some $C' \sqsubseteq C$.

Assume that there exists no $C' \sqsubseteq C$ such that $\langle C', D'' \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$. From Definitions 3.1 and 3.2, it follows that D'' and $\mathit{Created}^{\rho}$ are not disjoint—otherwise, D'' would be involved in a Resid^{ρ}-relation. From the fact that D'' is not involved in a $\mathit{CreateResid}^{\rho}$ relation either, it follows that:

- $\mathit{Created}^{\rho} \not\sqsubseteq D''$, and/or
- $D'' - \mathit{Created}^{\rho} \equiv \bigsqcup \{\, E' \mid \langle E, E' \rangle \in R \,\}$, for some $R \subset \mathit{Resid}^{\rho}$ such that there exist $\langle A, B \rangle \in R$, $\langle A, B' \rangle \in \mathit{Resid}^{\rho}$ for which $\langle A, B' \rangle \notin R$.

Define:

$$R' \triangleq R \cup \{ \langle A, B' \rangle \mid \langle A, B \rangle \in R,\ \langle A, B' \rangle \in \mathit{Resid}^{\rho} \}$$
$$D' \triangleq D'' \sqcup \mathit{Created}^{\rho} \sqcup \{ E' \mid \langle E, E' \rangle \in R' \}$$

Clearly, D' is the minimal supercontext of D'' for which $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho}$, where

$$C' \equiv \mathit{Creating}^{\rho} \sqcup \bigsqcup \{ E \mid \langle E, E' \rangle \in R' \} \sqsubseteq C$$

Since $\mathit{Creating}^{\rho}$ is always non-empty, C' is non-empty as well.

From Definition 4.1 it follows that

$$\langle B', D'' \rangle \in \mathit{Slice}*^{\rho} \Leftrightarrow \langle B', D' \rangle \in \mathit{Slice}*^{\rho} \Leftrightarrow \langle B', C' \rangle \in \mathit{Slice}*^{\rho'}$$

where $B' \sqsubseteq B$. □

Theorem 6.5 (Uniqueness of Slices) *Let $\rho : B \xrightarrow{\rho}{}^{*} D$ be a reduction, and let $D' \sqsubseteq D$ be non-empty. Then there exists a unique non-empty $B' \sqsubseteq B$ such that $\langle B', D' \rangle \in \mathit{Slice}*^{\rho}$.*

Proof. By induction on the length of the reduction ρ.

For $\rho \equiv \epsilon$, we have $\{ B' \mid \langle B', D' \rangle \in \mathit{Slice}*^{\epsilon} \} = \{ D' \}$ according to Definition 4.1.

For the inductive case, assume that $\rho \equiv \rho' \mathcal{A}$ such that $B \xrightarrow{\rho'}{}^{*} C \xrightarrow{\mathcal{A}} D$, and let $D' \sqsubseteq D$ be a non-empty context. According to Lemma 6.4, we may assume without loss of generality that $\langle C', D' \rangle \in (\mathit{Resid}^{\rho' \mathcal{A}} \cup \mathit{CreateResid}^{\rho' \mathcal{A}})$, for some $C' \sqsubseteq C$.

According to Lemma 6.3, we have *either* $\langle C', D' \rangle \in \mathit{Resid}^{\rho' \mathcal{A}}$ *or* $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho' \mathcal{A}}$.

Lemmas 6.1 and 6.2 state that both *Resid* and *CreateResid* map any non-empty context $D' \sqsubseteq D$ to a unique non-empty $C' \sqsubseteq C$. By induction, there exists a unique non-empty $B' \sqsubseteq B$ such that $\langle B', C' \rangle \in \mathit{Slice}*^{\rho'}$.

From Definition 4.1 it follows that this B' is the unique non-empty subcontext of B such that $\langle B', D' \rangle \in \mathit{Slice}*^{\rho}$. □

Given Theorem 6.5, we will be able to write $C = \mathit{Slice}*^{\rho}(D)$ instead of $\langle C, D \rangle \in \mathit{Slice}*^{\rho}$, for non-empty D.

The following lemma states that slices may be computed by repeatedly determining a unique related subcontext of the previous context in the reduction.

Lemma 6.6 *Let $\rho : B \xrightarrow{\rho'}{}^{*} C \xrightarrow{\mathcal{A}} D$ be a reduction, and let $D'' \sqsubseteq D$ be a non-empty context. Moreover, let D' be the unique minimal super-context of D'' for which there exists a non-empty context $C' \sqsubseteq C$ such that $\langle C', D'' \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$. Then*

$$\mathit{Slice}*^{\rho}(D') = \mathit{Slice}*^{\rho}(D'') = \mathit{Slice}*^{\rho'}(C')$$

Proof. Follows immediately from Definition 4.1, Lemma 6.4, and Theorem 6.5. □

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

Lemma 6.7 (Inclusion Lemma) *Let $\rho\mathcal{A}$ be a reduction such that $B \xrightarrow{\rho}{}^{*} C \xrightarrow{\mathcal{A}} D$, and let $D'' \sqsubseteq D' \sqsubseteq D$ be non-empty contexts such that there exist pairs $\langle C', D'\rangle \in (\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}})$, and $\langle C'', D''\rangle \in (\mathit{Resid}^{\rho\mathcal{A}} \cup \mathit{CreateResid}^{\rho\mathcal{A}})$. Then $C'' \sqsubseteq C'$.*

Proof. There are three cases:

1. $\langle C', D'\rangle \in \mathit{Resid}^{\rho\mathcal{A}}$ and $\langle C'', D''\rangle \in \mathit{Resid}^{\rho\mathcal{A}}$.

   From Definition 3.2 it follows that there are two cases:

   (a) ($\mathit{root}(D') \preceq \mathit{Created}^{\rho\mathcal{A}}$ or $\mathit{root}(D') \mid \mathit{Created}^{\rho\mathcal{A}}$), D' and $\mathit{Created}^{\rho\mathcal{A}}$ are disjoint, $\mathit{root}(D'') \preceq \mathit{Created}^{\rho\mathcal{A}}$, and D'' and $\mathit{Created}^{\rho\mathcal{A}}$ are disjoint. Then $C' \equiv D'$ and $C'' \equiv D''$, and therefore $C'' \sqsubseteq C'$.

   (b) $D' \equiv (p_R \cdot q' \leftarrow A')$, $D'' \equiv (p_R \cdot q'' \leftarrow A'')$ where $p_R \in \mathcal{O}_X(\overline{R_\mathcal{A}})$, $A' \sqsubseteq (() \leftarrow \sigma_\mathcal{A}(X))$, $A'' \sqsubseteq (() \leftarrow \sigma_\mathcal{A}(X))$, $q' = \mathit{root}(A')$, $q'' = \mathit{root}(A'')$, for some variable X.

   From Definitions 3.1 and 3.2 and left-linearity, it follows that there exists a unique occurrence p_L of X in $L_\mathcal{A}$ such that $C' \equiv (p_L \cdot q' \leftarrow A')$, $C'' \equiv (p_L \cdot q'' \leftarrow A'')$.

   $D'' \sqsubseteq D'$ implies $q'' \preceq q'$, $A'' \sqsubseteq A'$ and therefore $C'' \sqsubseteq C'$.

2. $\langle C', D'\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ and $\langle C'', D''\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$.

   From Definition 3.4 it follows that

$$\begin{array}{lcl} D' & \equiv & \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\ E' \mid \langle E, E'\rangle \in R'\ \} \\ D'' & \equiv & \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\ E' \mid \langle E, E'\rangle \in R''\ \} \\ C' & \equiv & \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\ E \mid \langle E, E'\rangle \in R'\ \} \\ C'' & \equiv & \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\ E \mid \langle E, E'\rangle \in R''\ \} \end{array}$$

   for R', $R'' \subseteq \mathit{Resid}^{\rho\mathcal{A}}$ such that for all $\langle E, E'\rangle \in R', R''$, both E and E' are are elementary. From $D'' \sqsubseteq D'$ it follows that $R'' \subseteq R'$ and therefore that $C'' \sqsubseteq C'$.

3. $\langle C', D'\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}$ and $\langle C'', D''\rangle \in \mathit{Resid}^{\rho\mathcal{A}}$

   According to Definition 3.4, we have that

$$\begin{array}{lcl} D' & \equiv & \mathit{Created}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\ E' \mid \langle E, E'\rangle \in R\ \} \\ C' & \equiv & \mathit{Creating}^{\rho\mathcal{A}} \sqcup \bigsqcup\{\ E \mid \langle E, E'\rangle \in R\ \} \end{array}$$

   for some $R \subseteq \mathit{Resid}^{\rho\mathcal{A}}$ such that for all $\langle E, E'\rangle \in R$, both E and E' are elementary. From $D'' \sqsubseteq D'$, and the disjointness of $\mathit{Created}^{\rho\mathcal{A}}$ and D'' it follows that there exists a subset $R' \subseteq R$ such that

$$D'' \equiv \bigsqcup\{\ E' \mid \langle E, E'\rangle \in R'\ \}$$

   Using an argument similar to that in case 1, it follows that

$$C'' \equiv \bigsqcup\{\ E \mid \langle E, E'\rangle \in R'\ \}$$

   Consequently $C'' \sqsubseteq C'$.

Note that the case where $\langle C', D' \rangle \in \mathit{Resid}^{\rho,\mathcal{A}}$ and $\langle C'', D'' \rangle \in \mathit{CreateResid}^{\rho,\mathcal{A}}$ is impossible, given $D'' \sqsubseteq D'$. □

Lemma 6.8 states that if $D''SubContextD'\ E''$ is the smallest supercontext of D'' involves in a *Resid* or *CreateResid*-relationship, and E' is the smallest supercontext of D' involves in a *Resid* or *CreateResid*-relationship, then $E'' \sqsubseteq E'$.

Lemma 6.8 *Let $\rho : B \xrightarrow{\rho}{}^{*} D$ be a reduction, let $D'' \sqsubseteq D' \sqsubseteq D$, and let D'' be non-empty. Furthermore, let E'' be the unique minimal supercontext of D'' such that $\langle C'', E'' \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$ for some C'', and let E' be the unique minimal supercontext of D' such that $\langle C', E' \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$ for some C'. Then $E'' \sqsubseteq E'$.*

Proof. The following cases can be distinguished:

1. $E'' = D''$ and $E' \sqsupseteq D'$. Trivial.
2. $E'' \sqsupset D''$ and $E' = D'$. From $E'' \neq D''$ it follows that D'' and $\mathit{Created}^{\rho}$ are not disjoint, and that $\langle C'', E'' \rangle \in \mathit{CreateResid}^{\rho}$. From $D'' \sqsubseteq D'$ and $D' = E'$ it follows that $\mathit{Created}^{\rho} \sqsubseteq D'$. Define $R',\ R'' \subseteq \mathit{Resid}^{\rho}$ as follows:
$$R' \equiv D' - \mathit{Created}^{\rho}$$
$$R'' \equiv D'' - \mathit{Created}^{\rho}$$
From $D'' \sqsubseteq D'$ it follows that $R'' \subseteq R'$. From Definition 3.4 it follows that:
$$E'' = \mathit{Created}^{\rho} \sqcup \bigsqcup \{\ Y\ \mid\ \langle X, Y \rangle \in \hat{R}''\ \}$$
where
$$\hat{R}'' = \{\ \langle X, Y' \rangle\ \mid\ \langle X, Y \rangle \in R'',\ \langle X, Y' \rangle \in \mathit{Resid}^{\rho}\ \}$$
From $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho}$ it follows that $\hat{R}'' \subseteq R'$, and therefore that $E'' \sqsubseteq E'$.
3. $E'' \sqsupset D''$ and $E' \sqsupset D'$. It follows that D'' and $\mathit{Created}^{\rho}$ are not disjoint, and that D' and $\mathit{Created}^{\rho}$ are not disjoint. Define $R',\ R'' \subseteq \mathit{Resid}^{\rho}$ as follows:
$$R' \equiv D' - \mathit{Created}^{\rho}$$
$$R'' \equiv D'' - \mathit{Created}^{\rho}$$
Then, $D'' \sqsubseteq D'$ implies that $R'' \subseteq R'$. Define:
$$\hat{R}' \equiv \{\ \langle X, Y' \rangle\ \mid\ \langle X, Y \rangle \in R',\ \langle X, Y' \rangle \in \mathit{Resid}^{\rho}\ \}$$
$$\hat{R}'' \equiv \{\ \langle X, Y' \rangle\ \mid\ \langle X, Y \rangle \in R'',\ \langle X, Y' \rangle \in \mathit{Resid}^{\rho}\ \}$$
Then $\hat{R}'' \subseteq \hat{R}'$. From Definition 3.4, we have that:
$$E'' = \mathit{Created}^{\rho} \sqcup \bigsqcup \{\ Y\ \mid\ \langle X, Y \rangle \in \hat{R}''\ \}$$
$$E' = \mathit{Created}^{\rho} \sqcup \bigsqcup \{\ Y\ \mid\ \langle X, Y \rangle \in \hat{R}'\ \}$$
Hence, $E'' \sqsubseteq E'$.

□

Theorem 6.9 (Inclusion Theorem) *Let $\rho : B \xrightarrow{\rho}{}^{*} D$ be a reduction, let $D'' \sqsubseteq D' \sqsubseteq D$, and let D'' be non-empty. Then $\mathit{Slice}{*}^{\rho}(D'') \sqsubseteq \mathit{Slice}{*}^{\rho}(D')$.*

Proof. Follows directly from Lemmas 6.7 and 6.8. □

Lemma 6.10 formally justifies the relationship between a reduction and the components of a projection triple. Since the reduction component of a projection triple in $\mathit{Project}{*}^{\rho}$ is derived by construction from the contractions of ρ, it can justifiably be deemed a subreduction of ρ.

Lemma 6.10 *Let ρ be a reduction, and let $\langle B, \sigma, D' \rangle \in \mathit{Project}{*}^{\rho}$. Then $B \xrightarrow{\sigma}{}^{*} D$ such that there exists an $E' \sqsubseteq D$ for which $E' \doteq D'$.*

Proof. By induction on the length of ρ.

According to Definition 5.1, $\langle B, \sigma, D' \rangle \in \mathit{Project}{*}^{\epsilon}$ implies that $\sigma \equiv \epsilon$ and that $B \equiv D'$. From this it follows trivially that $B \xrightarrow{\epsilon}{}^{*} D$, for $D \equiv D'$.

For the inductive case, assume that $\rho \equiv \rho' \mathcal{A}$, and $\langle B, \sigma, D'' \rangle \in \mathit{Project}{*}^{\rho}$. From Definition 5.1 it follows that two cases can be distinguished:

1. $D'' \sqsubseteq D'$, $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho}$, and $\langle B, \sigma', C' \rangle \in \mathit{Project}{*}^{\rho}$.

   By induction, there exists a reduction $B \xrightarrow{\sigma'}{}^{*} C$ for some $C \sqsupseteq C'$.

   From Definition 3.4 it follows that $C' \xrightarrow{\mathcal{A}} D'$. Therefore $B \xrightarrow{\sigma'\mathcal{A}}{}^{*} C[E] = D$, where $E \equiv (\mathit{root}(C') \leftarrow D')$. Since D' and E are isomorphic, and $D'' \sqsubseteq D'$, it follows that $D'' \doteq E'$ for some $E' \sqsubseteq D$.

2. $D'' \sqsubseteq D'$, $\langle C', D' \rangle \in \mathit{Resid}^{\rho}$, and $\langle B, \sigma', C' \rangle \in \mathit{Project}{*}^{\rho}$.

   By induction, there exists a reduction $B \xrightarrow{\sigma'}{}^{*} C$ for some $C \sqsupseteq C'$.

   From Definition 3.2 it follows that $C' \doteq D'$. Hence, $B' \xrightarrow{\sigma'}{}^{*} C[E] = D$, where $E \equiv (\mathit{root}(C') \leftarrow D')$. Since D' and E are isomorphic, and $D'' \sqsubseteq D'$, it follows that $D'' \doteq E'$ for some $E' \sqsubseteq D$. □

The lemma below establishes a connection between the relations $\mathit{Slice}{*}^{\rho}$ and $\mathit{Project}{*}^{\rho}$.

Lemma 6.11 *Let ρ be a reduction such that $B \xrightarrow{\rho}{}^{*} D$, and let $B' = \mathit{Slice}{*}^{\rho}(D')$ for some non-empty $D' \sqsubseteq D$. Then there exists a triple $\langle B', \sigma, D' \rangle \in \mathit{Project}{*}^{\rho}$.*

Proof. By induction on the length of reduction ρ.

Let $\rho \equiv \epsilon$. From Definition 4.1 it follows that $B' = \mathit{Slice}{*}^{\epsilon}(D')$ implies $B' \equiv D'$. Moreover, from Definition 5.1 it follows that $\langle B', \epsilon, D' \rangle \in \mathit{Project}{*}^{\epsilon}$ implies $B' \equiv D'$ as well, so that the lemma trivially holds.

For the inductive case, assume that $\rho \equiv \rho'\mathcal{A}$ such that $B \xrightarrow{\rho'}{}^{*} C \xrightarrow{\mathcal{A}} D$, and let $B' = \mathit{Slice}{*}^{\rho}(D'')$, for some non-empty $D'' \sqsubseteq D$. According to Lemmas 6.4 and 6.6 there exists a unique $D' \sqsupseteq D''$ such that $\langle C', D' \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$ and $\mathit{Slice}{*}^{\rho}(D'') = \mathit{Slice}{*}^{\rho}(D') = \mathit{Slice}{*}^{\rho'}(C') = B'$.

By induction there exists a triple $\langle B', \sigma', C' \rangle \in \mathit{Project}{*}^{\rho'}$. From Lemma 6.3, it follows that there are two cases:

1. $\langle C', D' \rangle \in \mathit{Resid}^{\rho}$. Since $D'' \sqsubseteq D'$ it follows from Definition 5.1 that that $\langle B', \sigma', D' \rangle \in \mathit{Project}{*}^{\rho}$.
2. $\langle C', D' \rangle \in \mathit{CreateResid}^{\rho}$. Since $D'' \sqsubseteq D'$ it follows from Definition 5.1 that $\langle B', \sigma'\mathcal{A}, D' \rangle \in \mathit{Project}{*}^{\rho}$. □

The soundness theorem states that the *Slice*∗ relation computes slices that comply with Definition 1.1.

Theorem 6.12 (Soundness) *Let ρ be a reduction such that $B \xrightarrow{\rho}{}^{*} D$. Moreover, let $B' = \mathit{Slice}{*}^{\rho}(D'')$ for some non-empty $D'' \sqsubseteq D$. Then there exists a reduction σ such that:*

1. *$\langle B', \sigma, D'' \rangle \in \mathit{Project}{*}^{\rho}$, and*
2. *$B' \xrightarrow{\sigma}{}^{*} D'$ such that there exists an $E'' \sqsubseteq D'$ for which $E'' \doteq D''$.*

Proof. Follows immediately from Lemmas 6.10 and 6.11. □

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

Theorem 6.13 (Minimality) *Let ρ be a reduction such that $B \xrightarrow{\rho}{}^{*} D$, and let $B'_s = \mathit{Slice}{*}^{\rho}(D'_s)$ for some non-empty D'_s. Then $\langle B'_p, \sigma, D'_p \rangle \in \mathit{Project}{*}^{\rho}$ and $D'_p \sqsupseteq D'_s$ together imply that $B'_p \sqsupseteq B'_s$.*

Proof. By induction on the length of reduction ρ.

For $\rho \equiv \epsilon$, Definition 5.1 states that $\langle B'_p, \sigma, D'_p \rangle \in \mathit{Project}{*}^{\epsilon}$ implies $\sigma \equiv \epsilon$ and $B'_p \equiv D'_p$. Moreover, according to Definition 4.1 we have that $B'_s = \mathit{Slice}{*}^{\epsilon}(D'_s)$ implies $B'_s \equiv D'_s$. Therefore $D'_p \sqsupseteq D'_s$ implies that $B'_p \sqsupseteq B'_s$.

For the inductive case, assume that $\rho \equiv \rho'\mathcal{A}$, let $B'_s = \mathit{Slice}{*}^{\rho}(D'_s)$ for some non-empty D'_s, and let $\langle B'_p, \sigma, D'_p \rangle \in \mathit{Project}{*}^{\rho}$ such that $D'_p \sqsupseteq D'_s$. Then by Definition 5.1, there exists a $D_p \sqsupseteq D'_p$ such that $\langle C_p, D_p \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho'\mathcal{A}})$, and $\langle B'_p, \sigma', C_p \rangle \in \mathit{Project}{*}^{\rho'}$.

According to Lemmas 6.4 and 6.6, there exists a unique minimal super-context D_s of D'_s such that $\langle C_s, D_s \rangle \in (\mathit{Resid}^{\rho} \cup \mathit{CreateResid}^{\rho})$ and:

$$\mathit{Slice}{*}^{\rho}(D'_s) = \mathit{Slice}{*}^{\rho}(D_s) = \mathit{Slice}{*}^{\rho'}(C_s) = B_s$$

By induction, $\langle B'_p, \sigma', C_p \rangle \in \mathit{Project}{*}^{\rho'}$ and $C_p \sqsupseteq C_s$ together imply that $B_p \sqsupseteq B_s$. Consequently it suffices to show that $C_p \sqsupseteq C_s$.

From (i) the fact that D_s is the *minimal* super-context of D'_s that is related in a $\mathit{CreateResid}^{\rho}$-relation, (ii) the fact that D_p is *some* supercontext of D'_s that is involved in a $\mathit{CreateResid}^{\rho}$-relation, and (iii) Definition 3.4, it follows that $D_p \sqsupseteq D_s$. According to Lemma 6.7, we therefore have $C_p \sqsupseteq C_s$. This concludes the proof of the minimality theorem. □

Together, Theorems 6.12 and 6.13 imply that our construction of slices agrees with Definition 1.1.

7 Nonlinear Rewriting Systems[1]

Unfortunately, our previous definitions do not extend trivially to left-nonlinear TRSs, because they do not account for the fact that nonlinearities 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 4 is applied to ff $\oplus$ ff, this results in a contraction $\mathcal{A} : T \equiv \text{ff} \oplus \text{ff} \longrightarrow \text{ff} \equiv T'$. Our previous definitions yield $C = () \leftarrow (\bullet \oplus \bullet) \sqsubseteq T$ as the slice with respect to criterion $D = () \leftarrow \text{ff} \sqsubseteq T'$. This is not a valid slice, because some 'instantiations' of C do not reduce to a context containing D, e.g., $() \leftarrow \text{tt} \oplus \text{ff}$ 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 consist of a single context.

A simple solution for nonlinear TRSs would be to restrict *VarPairs* to variables which occur at most once in the left-hand side of a rule. However, this would yield larger slices than necessary. For instance, for the reduction of Figure 5 the non-minimal slice ff $\wedge$ (tt $\oplus$ tt) would be computed. The immediate cause for this inaccuracy is the fact that the subcontexts $(1) \leftarrow \text{ff}$ and $(2) \leftarrow \text{ff}$ of T_3 are deemed responsible for the creation of term T_4. However, they are *residuals of the same subcontext* $C = (1) \leftarrow \text{ff} \sqsubseteq T_0$. 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 a set of "equivalent" contexts $\mathcal{S}$ that are residuals of a common context (one that occurs earlier in the reduction sequence), then the contexts in $\mathcal{S}$ are deemed to be residuated by α (assuming X occurs in R_α). All other cases cause *creation*: those subcontexts matched against X that are not residuals of a common context are deemed *creating*, and the corresponding subcontexts matched against X in R_α are *created*.

7.1 Formal Definitions for Nonlinear Systems

If a context D is created at some point in a reduction, and D has a residual C which occurs later in the reduction, we will say that D is a *progenitor* of C. This concept will be useful for formulating an adequate notion of slice for nonlinear TRSs. Formally, we have:

Definition 7.1 (Progenitor) *Let T be a term, σ and τ be reductions such that $\sigma\tau : U \longrightarrow^* T$ for some term U, and D be a subcontext of T. Then we will say that a context C is a σ,τ-*progenitor *of D if $\langle C, D\rangle \in \mathit{Resid}*^\tau$, and either $C \sqsubseteq \mathit{Created}^\sigma$ or $\sigma = \epsilon$.*

We will say that a context forest $\mathcal{S}$ has *common* σ,τ–progenitor C if for all $D \in \mathcal{S}$, D has σ,τ–progenitor C. Note that an empty context may have more than one progenitor, due to collapse rules, which have the effect of combining existing empty contexts as well as creating new ones. Also note that the progenitor of a context C created by the last step of reduction ρ has ρ,ϵ–progenitor C.

We can now revise Definition 3.1 to account for common residuals in subterms matched nonlinearly:

[1] The definition of the *Slice*$*$ relation for nonlinear systems in [15] contained an error. The definitions in this section therefore supersede the earlier ones.

Definition 7.2 (*VarPairs* for nonlinear TRSs) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{aligned}\mathit{VarPairs}^{\rho\mathcal{A}} \triangleq \{\ \langle \mathcal{S}_1, \mathcal{S}_2\rangle \mid\ & X \in \mathcal{V},\\ & C \sqsubseteq (() \leftarrow \sigma_A(X)) \ \textit{ or } \ C = (() \leftarrow \bullet),\\ & q = \mathit{root}(C),\\ & \mathcal{S}_1 = \{(p_L \cdot q \leftarrow C) \mid p_L \in \mathcal{O}_X(\overline{L_\mathcal{A}})\},\\ & \mathcal{S}_2 = \{(p_R \cdot q \leftarrow C) \mid p_R \in \mathcal{O}_X(\overline{R_\mathcal{A}})\},\\ & \mathcal{S}_1 \textit{ has a common } \sigma, \tau\textit{–progenitor},\\ & \sigma\tau = \rho\ \}\end{aligned}$$

For linear TRSs, Definition 7.2 reduces to Definition 3.1, since $\mathcal{S}_1$ is always a singleton and thus has a trivial common progenitor.

In nonlinear TRSs, certain empty contexts at the "edge" of *Creating* and *Resid* have a creating effect that does not occur in the linear case; the definition of *Slice*$*$ for nonlinear systems must therefore be modified accordingly. More specifically, in the linear case, the empty contexts between *Creating* and *Resid* are irrelevant to the applicability of the redex. However, in the nonlinear case, they are indeed relevant, since if these "glue" contexts were not empty, the nonlinear match would not occur (unless, as with all contexts matched nonlinearly, the edge contexts have a common progenitor).

The following definition computes the union of the slices with respect to relevant edge empty contexts:

Definition 7.3 (*EdgeSlices*) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{aligned}\mathit{EdgeSlices}^{\rho\mathcal{A}} \triangleq \bigsqcup\{\ C \mid\ & \langle \mathcal{S}_1, \mathcal{S}_2\rangle \in \mathit{VarPairs}^{\rho\mathcal{A}},\\ & (p \leftarrow \bullet) \in (\mathcal{S}_1 \cap \mathcal{O}_\bullet(\mathit{Creating}^{\rho\mathcal{A}})),\\ & D \textit{ is a } \sigma, \tau\textit{–progenitor of } (p \leftarrow \bullet),\\ & D \textit{ is not a common } \sigma, \tau\textit{–progenitor of } \mathcal{S}_1,\\ & \langle C, D\rangle \in \mathit{CreatedSlice}{*}^{\sigma},\\ & \sigma\tau = \rho\ \}\end{aligned}$$

(The relation *CreatedSlice*$*$, defined formally below, is a subrelation of *Slice*$*$ in which the second elements are *created* by the last step of the reduction; this yields a slice specific to the progenitor in the definition when more than one progenitor exists). Definition 7.3 yields the union of slices with respect to empty context criteria at the "edge" between *Creating* and *Resid* that are not derived from a progenitor common to all the contexts associated with a given variable. Note that $\mathit{EdgeSlices}^{\rho\mathcal{A}}$ is always empty for linear TRSs, since for such systems, the forest $\mathcal{S}_1$ in the definition is always a singleton.

Our definition of *Slice*$*$ in the nonlinear case is essentially the same as that for the linear case, except that we must add the information in *EdgeSlices* where appropriate:

Definition 7.4 (*Slice*$*$ for nonlinear TRSs) *Let $\rho\mathcal{A}$ be a reduction. Then*

$$\begin{aligned}\mathit{Slice}{*}^{\epsilon} &\triangleq \{\ \langle C, C\rangle \mid C \in \mathit{Cont}(\Sigma)\ \}\\ \mathit{Slice}{*}^{\rho\mathcal{A}} &\triangleq \mathit{ResidSlice}{*}^{\rho\mathcal{A}} \cup \mathit{CreatedSlice}{*}^{\rho\mathcal{A}}\end{aligned}$$

where

$$\begin{array}{lcl}
\mathit{ResidSlice}*^{\rho\mathcal{A}} & \triangleq & \mathit{Slice}*^{\rho} \cdot \mathit{Resid}^{\rho\mathcal{A}} \\
\mathit{CreatedSlice}*^{\rho\mathcal{A}} & \triangleq & \{\ \langle C, E\rangle \mid \langle C', D\rangle \in \mathit{Slice}*^{\rho}, \\
 & & \quad \textit{there exists a minimal } E' \sqsupseteq E \textit{ such that } \langle D, E'\rangle \in \mathit{CreateResid}^{\rho\mathcal{A}}, \\
 & & \quad E \textit{ and } \mathit{Created}^{\rho\mathcal{A}} \textit{ are not disjoint}, \\
 & & \quad C = C' \sqcup \mathit{EdgeSlices}^{\rho\mathcal{A}}\ \}
\end{array}$$

Definition 7.4 is complicated by the necessity of splitting the pure residuation case from the creation case—the two cases *both* apply only when created and residuated information overlap exactly; i.e., when $\mathcal{A}$ is a collapse rule application.

While Definition 7.4, along with the auxiliary definitions, may appear rather complicated, testing whether two contexts have a common progenitor can be performed cheaply in practice if reduction is implemented using *term graph rewriting* techniques [4, 20]. Graph rewriting causes terms that are created by contraction of sets of residuals of previous reductions to be shared in a graphical data structure. If such an implementation is used, testing whether two contexts have a common progenitor reduces to determining whether the contexts are represented by a common shared subgraph.

7.2 Example: Slicing in a nonlinear system

Recall the reduction used in the example of Fig. 5:

$$\begin{array}{lll}
\underline{\mathrm{ff} \wedge (\mathrm{tt} \oplus \mathrm{tt})} \equiv T_0 & \xrightarrow{\mathcal{A}_1} & (\underline{\mathrm{ff} \wedge \mathrm{tt}}) \oplus (\mathrm{ff} \wedge \mathrm{tt}) \equiv T_1 \xrightarrow{\mathcal{A}_2} \mathrm{ff} \oplus (\underline{\mathrm{ff} \wedge \mathrm{tt}}) \equiv T_2 \\
 & \xrightarrow{\mathcal{A}_3} & \underline{(\mathrm{ff} \oplus \mathrm{ff})} \equiv T_3 \xrightarrow{\mathcal{A}_4} \mathrm{ff} \equiv T_4
\end{array}$$

We have denoted the contractions in the reduction above by $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, and $\mathcal{A}_4$.

Applying the definitions of the previous section to this example, we find that the most interesting step is the contraction $\mathcal{A}_4$, which uses nonlinear rule [B4]. In Figure 7, the two 'ff' subterms in the term matched by contraction $\mathcal{A}_4$ have the *same* progenitor in the initial term, indicated by dotted lines. Definition 7.2 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 7 does *not* depend on the 'ff' subterm of the initial term of the reduction.

It is instructive to observe the effect of the formal definitions of Section 7.1 with respect to contraction $\mathcal{A}_4$. In order to determine whether the contexts bound to the nonlinearly matched variable X are derived from a common source, we must first consider the common progenitors of the contexts in $\mathit{VarPairs}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\mathcal{A}_4}$, which are:

$$\begin{array}{lcll}
\{\ (1) \leftarrow \mathrm{ff},\ (2) \leftarrow \mathrm{ff}\ \} & \sqsubseteq & T_3 & \text{has common } \epsilon, \mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\text{–progenitor} \quad (1) \leftarrow \mathrm{ff} \ \sqsubseteq \ T_0 \\
\{\ (1) \leftarrow \bullet,\ (2) \leftarrow \bullet\ \} & \sqsubseteq & T_3 & \text{has common } \epsilon, \mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\text{–progenitor} \quad (1) \leftarrow \bullet \ \sqsubseteq \ T_0
\end{array}$$

Since the contexts bound to the nonlinearly matched variable X (namely, $(1) \leftarrow \mathrm{ff}$, $(2) \leftarrow \mathrm{ff}$, $(1) \leftarrow \bullet$, and $(2) \leftarrow \bullet$) have a common progenitor, they are included in *VarPairs*:

$$\begin{array}{lcl}
\mathit{VarPairs}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\mathcal{A}_4} & = & \{\ \langle\{(1) \leftarrow \mathrm{ff}, (2) \leftarrow \mathrm{ff}\}, \emptyset\rangle,\ \ \langle\{(1) \leftarrow \bullet, (2) \leftarrow \bullet\}, \emptyset\rangle\ \} \\
\mathit{Resid}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\mathcal{A}_4} & = & \langle() \leftarrow \bullet, () \leftarrow \bullet\rangle
\end{array}$$

Using *VarPairs*, we can eliminate the nonlinearly matched contexts from *Creating* and *Created*:

$$\begin{array}{lcl} \mathit{Creating}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\mathcal{A}_4} & = & () \leftarrow \bullet \oplus \bullet \\ \mathit{Created}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\mathcal{A}_4} & = & () \leftarrow \mathrm{ff} \end{array}$$

However, before we can compute the *Slice*$*$ relation, we must consider slices with respect to the "edge" empty contexts $(1) \leftarrow \bullet$ and $(2) \leftarrow \bullet$ which separate *Creating* from elements of *Resid.* Their progenitor information is as follows:

$$(1) \leftarrow \bullet \;\sqsubseteq\; T_3 \quad \left\{ \begin{array}{ll} \text{has } \mathcal{A}_1\mathcal{A}_2, \mathcal{A}_3\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_2 \\ \text{has } \mathcal{A}_1, \mathcal{A}_2\mathcal{A}_3\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_1 \\ \text{has } \epsilon, \mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_0 \end{array} \right.$$

$$(2) \leftarrow \bullet \;\sqsubseteq\; T_3 \quad \left\{ \begin{array}{ll} \text{has } \mathcal{A}_1\mathcal{A}_2\mathcal{A}_3, \epsilon\text{–progenitor} & (2) \leftarrow \bullet \;\sqsubseteq\; T_3 \\ \text{has } \mathcal{A}_1, \mathcal{A}_2\mathcal{A}_3\text{–progenitor} & (2) \leftarrow \bullet \;\sqsubseteq\; T_1 \\ \text{has } \epsilon, \mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\text{–progenitor} & (1) \leftarrow \bullet \;\sqsubseteq\; T_0 \end{array} \right.$$

$(1) \leftarrow \bullet$ and $(2) \leftarrow \bullet$ each have three progenitors because the collapse rule **[B2]** (applied in contractions $\mathcal{A}_2$ and $\mathcal{A}_3$) has the effect of combining the empty contexts above and below the matched part of the redex, as well as creating a "new" empty context.

For the purpose of computing $\mathit{EdgeSlices}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\mathcal{A}_4}$, we need consider only those progenitors not common to both $(1) \leftarrow \bullet \sqsubseteq T_3$ and $(2) \leftarrow \bullet \sqsubseteq T_3$. These are: $(1) \leftarrow \bullet \sqsubseteq T_2$, $(1) \leftarrow \bullet \sqsubseteq T_1$, $(2) \leftarrow \bullet \sqsubseteq T_3$, and $(2) \leftarrow \bullet \sqsubseteq T_1$. The *CreatedSlice*$*$ subrelations relevant to the latter contexts are as follows:

$$\begin{array}{lcl} \{\, \langle () \leftarrow \bullet \wedge (\bullet \oplus \bullet), (1) \leftarrow \bullet \rangle,\ \langle () \leftarrow \bullet \wedge (\bullet \oplus \bullet), (2) \leftarrow \bullet \rangle \,\} & \subseteq & \mathit{CreatedSlice}*^{\mathcal{A}_1} \\ \langle () \leftarrow \bullet \wedge (\mathrm{tt} \oplus \bullet), (1) \leftarrow \bullet \rangle & \subseteq & \mathit{CreatedSlice}*^{\mathcal{A}_1\mathcal{A}_2} \\ \langle () \leftarrow \bullet \wedge (\bullet \oplus \mathrm{tt}), (2) \leftarrow \bullet \rangle & \subseteq & \mathit{CreatedSlice}*^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3} \end{array}$$

Taking the context union of the *CreatedSlice*$*$ information above, we get:

$$\begin{array}{lcl} \mathit{EdgeSlices}^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\mathcal{A}_4} & = & () \leftarrow \bullet \wedge (\bullet \oplus \bullet) \;\sqcup\; () \leftarrow \bullet \wedge (\mathrm{tt} \oplus \bullet) \;\sqcup\; () \leftarrow \bullet \wedge (\bullet \oplus \mathrm{tt}) \\ & = & () \leftarrow \bullet \wedge (\mathrm{tt} \oplus \mathrm{tt}) \end{array}$$

Combining the information computed above and using Definition 7.4, we finally have:

$$\mathit{Slice}*^{\mathcal{A}_1\mathcal{A}_2\mathcal{A}_3\mathcal{A}_4} \;=\; \{\, \langle () \leftarrow \bullet, () \leftarrow \bullet \rangle,\ \langle () \leftarrow \bullet \wedge (\mathrm{tt} \oplus \mathrm{tt}), () \leftarrow \mathrm{ff} \rangle \,\}$$

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

7.3 Nonlinear systems and Optimal Slices

Although the approach to nonlinear slicing developed in the previous section is sound, it does not always yield minimal slices. To see this, consider the **B** reductions in Fig. 9. Although both ρ_1 and ρ_2 start and end at the same term, using the definitions of Section 7.1, the slice with respect to criterion T_5 is $(\bullet \wedge \mathrm{ff}) \wedge \mathrm{tt} \oplus \mathrm{tt}$, whereas the slice with respect to criterion T_5' is $(\mathrm{ff} \wedge \mathrm{ff}) \wedge \mathrm{tt} \oplus \mathrm{tt}$, i.e., the entire initial term.

$$\rho_1: \quad (\underline{\mathrm{ff} \wedge \mathrm{ff}}) \wedge (\mathrm{tt} \oplus \mathrm{tt}) \equiv T_0 \xrightarrow{[B3]} \mathrm{ff} \wedge (\mathrm{tt} \oplus \mathrm{tt}) \equiv T_1 \longrightarrow^* \underline{\mathrm{ff} \oplus \mathrm{ff}} \equiv T_4 \xrightarrow{[B4]} \mathrm{ff} \equiv T_5$$

$$\rho_2: \quad \underline{(\mathrm{ff} \wedge \mathrm{ff}) \wedge (\mathrm{tt} \oplus \mathrm{tt})} \equiv T_0 \xrightarrow{[B1]} ((\mathrm{ff} \wedge \mathrm{ff}) \wedge \mathrm{tt}) \oplus ((\mathrm{ff} \wedge \mathrm{ff}) \wedge \mathrm{tt}) \equiv T_1'$$
$$\longrightarrow^* (\underline{\mathrm{ff} \wedge \mathrm{ff}}) \oplus (\mathrm{ff} \wedge \mathrm{ff}) \equiv T_2' \xrightarrow{[B3]} \mathrm{ff} \oplus (\underline{\mathrm{ff} \wedge \mathrm{ff}}) \equiv T_3' \xrightarrow{[B3]} \underline{\mathrm{ff} \oplus \mathrm{ff}} \equiv T_4' \xrightarrow{[B4]} \mathrm{ff} \equiv T_5'$$

Figure 9: Sensitivity of nonlinear slicing to reduction strategy

The difference in the slices results from the order in which redexes were contracted in the two reductions. In ρ_1, the $(\mathrm{ff} \wedge \mathrm{ff}) \equiv S_0$ subterm of T_0 is contracted immediately, and two residuals of its contractum, ff, subsequently appear in term T_4. In ρ_2, however, S_0 is not immediately contracted. Instead, the reduction produces an intermediate term T_2' containing two residuals of S_0. These residuals are contracted in subsequent steps, ultimately yielding the term T_4'. However, unlike T_4, the two 'ff' subterms of T_4' are *not* residuals of any previous term. Since the 'ff' subterms of T_4 have a common progenitor, the definitions of Section 7.1 allow information common to the slices of the ff subterms of T_4 to be omitted when the nonlinear rule **[B4]** is applied. In the case of T_4', however, the 'ff' subterms have no common progenitor, and thus no information can be omitted.

It should be clear from the example of Fig. 9 that the notion of progenitor is dependent upon reduction order. One way to avoid the problems illustrated by Fig. 9 is to use an *innermost* reduction strategy, in which all redexes are contracted before they are residuated. However, if we do not wish to impose restrictions on allowable reduction strategies, we must take into account the behavior of reductions such as ρ_2, in which terms that have no common progenitor *could have had* a common progenitor if the redexes were contracted in a different order.

Put another way, we must treat sets of terms that are all "derived in the same way" from a set of residuals with a common progenitor as equivalent to sets of terms with a true common progenitor. Maranget [22, 23] defines a notion of equivalence modulo permutation of redexes that could be used for determining when classes of terms are or could have been residuals of a common term. However, if reduction is implemented using term graph rewriting techniques, terms that have common progenitors and terms that *could have* common progenitors are indistinguishable. In the case of the example in Fig. 9, both term T_4 and term T_4' would be represented by identical graphs in which the 'ff' subterms would be shared.

Unfortunately, even graph rewriting does not eliminate the possibility of computing suboptimal slices for nonlinear systems. Consider, for instance, the following TRS **E**:

[E1]	$\mathrm{f}(X) \longrightarrow \mathrm{eq}(\mathrm{g}(X), \mathrm{h}(X))$	**[E4]**	$\mathrm{k}(\mathrm{a}) \longrightarrow \mathrm{b}$
[E2]	$\mathrm{h}(X) \longrightarrow \mathrm{k}(X)$	**[E5]**	$\mathrm{eq}(X, X) \longrightarrow \mathrm{c}$
[E3]	$\mathrm{g}(X) \longrightarrow \mathrm{k}(X)$		

Note in particular that rule **[E5]** is nonlinear. Now consider the following **E**-reduction:

$$\rho: \quad \mathrm{f(a)} \xrightarrow{[E1]} \mathrm{eq(g(a),\ h(a))} \xrightarrow{[E2]} \mathrm{eq(k(a),\ h(a))} \xrightarrow{[E3]} \mathrm{eq(k(a),\ k(a))}$$
$$\xrightarrow{[E4]} \mathrm{eq(b,\ k(a))} \xrightarrow{[E4]} \mathrm{eq(b,\ b)} \equiv T \xrightarrow{[E5]} \mathrm{c}$$

In principle, we ought to be able to determine that the slice with respect to the final term '*c*' of ρ is $\mathrm{f}(\bullet)$,

since we can attain the same final term by omitting the fourth and fifth reduction steps entirely. However it is difficult to see how any information short of maintaining the entire reduction history could be used to determine that this is the case. In particular, note that the 'b' subterms of the intermediate term T in ρ do not have a common progenitor, nor are they derived in an "equivalent" way from the sets of residuals. Therefore, we cannot use information about the derivations of the 'b' subterms in isolation as a means for allowing common slice information to be omitted when rule **[E5]** is applied.

We are led to conclude that short of maintaining information about an entire reduction history, the only systematic way to treat nonlinear rules is to eliminate information associated with nonlinearly-matched subterms possessing a common progenitor (generalized using graph reduction techniques to account for "potential progenitors"). It is conceivable, however, that a restricted class of reduction systems or reduction strategies could eliminate the problems exhibited in the example of Fig. 9. We leave it to future work to explore these possibilities further.

8 Implementation

8.1 Overview

In principle, one could implement slicing by storing information about every step of a reduction ρ, 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. We will instead describe an alternate method that allows slices to be produced as a "side-effect" of the reduction process, in a way that efficiently computes slices with respect to any chosen criterion. This technique has been implemented in the SML language [24] for a class of conditional term rewriting systems; this class subsumes the unconditional linear and nonlinear systems discussed in this paper.

Our technique will use a variant of the *term graph* representation [4, 20]. In such a representation, every function symbol in a term corresponds to a unique graph node. However, since shared subgraphs (corresponding to sets of residuals) may be created during the reduction process, a graph node may represent *several* function symbols in the corresponding term.

Our graph representation will use *two* distinct node types, as follows:

elementary nodes: These correspond to the usual term graph nodes. Each such node will contain information used to compute slices for the elementary contexts (i.e., function symbols—contexts of the form $f(\bullet, \cdots, \bullet)$) represented by the node.

empty context nodes: These represent empty contexts in the term denoted by the graph. Each empty context node will contain information used to compute the union of all slices for the empty contexts that the node represents (recall that $Slice*^{\rho}$ is not necessarily single-valued on empty contexts). Empty context nodes with non-null slice information are created as a result of applying *collapse* rules. Such rules are implemented using special *indirection* elementary nodes [26], which are subsequently transformed into empty context nodes.

A slice with respect to any non-empty context D is determined by computing the union of the slices with respect to all elementary and empty contexts that are a subcontext of D. These slices are in turn obtained using information from the elementary and context nodes in the graph representing the term. Slices for

$$
\begin{array}{lcl}
\textit{OriginSet} & ::= & \textit{Path}\ \ \textbf{set} \\
\textit{Path} & ::= & \textbf{int}\ \ \textbf{list} \\
\textit{Operator} & ::= & \nabla \\
 & | & \cdots \\
 & \vdots & \\
\textit{EmptyContextInfo} & ::= & \textit{OriginSet} \\
\textit{EmptyContextNode} & ::= & \mathcal{C}\langle\ \textit{ElementaryNode}\ \ \textbf{ref},\ \textit{EmptyContextInfo}\ \rangle \\
\textit{ElementaryNode} & ::= & \mathcal{F}\langle\ \textit{Operator},\ \textit{TermGraph}\ \ \textbf{list},\ \textit{OriginSet}\ \rangle \\
\textit{TermGraph} & ::= & \textit{EmptyContextNode}\ \ \textbf{ref}
\end{array}
$$

Figure 10: Type definitions for data structures implementing terms in linear TRSs.

the elementary and empty contexts are represented by sets of paths in the initial term; these sets can be represented by bit vectors to allow unions to be computed efficiently.

Given a TRS, our technique "compiles" each rewriting rule into a fragment of executable code that carries out a corresponding transformation on a term graph representation. This code makes use of several auxiliary functions to perform unions of the sets of paths used to represent slices, to process indirection and context nodes, and to implement equality tests for nonlinear rules. The set of rewriting rules can then be applied and ordered according to a variety of user-specified *strategies*; however, here we will focus solely on the implementation of individual rules.

In the next section, we provide details on the implementation of slicing for linear TRSs. We will then extend those ideas to nonlinear TRSs. Rather than providing a full algorithm for compiling, rewriting, and extracting slices (most aspects of which amount to tedious inductive application of simple translation rules to terms, rewriting rules, and rewriting strategies), we will instead illustrate its key points by example.

8.2 Implementing Linear Systems

In describing our implementation technique, we will use a simplified dialect of SML in which basic data types may take the form of fixed-arity *constructors*, positive integers, lists, sets, and *references* (i.e., pointers) to the other data types. A reference is mutable, i.e., the value to which it refers may be updated in place. If the value to which a reference points is updated, all other instances of the same reference will "see" the update; thus references can be regarded as edges in a directed graph. All the other basic types are immutable; once instantiated, they cannot be altered. For linear systems, our data structures will be constructed from the recursively-defined types given in Figure 10.

An *OriginSet* is a set of *Path* data structures, the contents of which denote a context of the initial term

which comprises a slice. A *Path* will be represented by a list of integers, as defined in Section 2. An *EmptyContextNode* is represented by the $\mathcal{C}\langle\,\cdot,\,\cdot\,\rangle$ constructor; such a node is comprised of a reference to an elementary node and auxiliary information about the empty context node (*EmptyContextInfo*). For linear systems, *EmptyContextInfo* is simply an *OriginSet* representing the union of all slices for the empty contexts represented by the empty context node (the definition for *EmptyContextInfo* will become slightly more complex when we consider nonlinear systems in the next section). An *ElementaryNode* is represented by the $\mathcal{F}\langle\,\cdot,\,\cdot,\,\cdot\,\rangle$ constructor; such a node is comprised of an *Operator*, a list of *TermGraph* children, and an *OriginSet* containing the slice information for the elementary contexts represented by the elementary node. Operators are represented by nullary constructors, which we will depict without argument delimiters ('$\langle$' and '$\rangle$') for brevity. In addition to the signature-specific operators determined by the TRS, there is also a distinguished *indirection* operator, ∇, which will be used in conjunction with collapse rules. Finally, a *TermGraph* is simply a reference to an empty context node.

8.2.1 Compiling Reduction Rules

Figure 11 depicts the compiled code corresponding to the first three rules (i.e., the linear rules) of the system **B** of Figure 4.

In the code depicted in Figure 11 and in other code in the sequel, **let** expressions extract component values from composite data structures through the use of *patterns* containing free variables. The wildcard pattern, '_', is used when a component's value is not of interest. The **ref** keyword is used in expressions or patterns respectively to build or extract mutable values. An expression may be matched against a sequence of patterns using a **case** expression; values bound to variables on the left-hand side of the '$\Rightarrow$' symbol in a clause of the **case** expression may be used in the expression returned by the right-hand side. References are updated using ':='. Sets are manipulated using ordinary set notation. **skip** represents the null statement. Each instance of a constructor, list, or set is a *newly allocated* instance distinct from other instances.

The function *oneStepRewrite*(*termGraph*) uses one of the three rewriting rules of *termGraph*, if applicable, to transform the root of the graph in place. If none of the rules are applicable, the graph is left untransformed. It should not be difficult to see that the computations performed on path sets in Figure 11 correspond to the components of Definition 4.1.

Note in Figure 11 that rule **[B2]** is a collapse rule. As a result, an indirection node is "created" when the rule is applied. The function *compressOneIndirection*(*termGraph*) depicted in Figure 12 is used to "compress" indirections by merging the origin sets for the indirection elementary context into the origin sets for the empty context nodes "above" and "below" the indirection.

8.2.2 Reducing the Graph

Figure 13 depicts the reduction of the **B**-term $T = \mathrm{ff} \wedge (\mathrm{tt} \oplus \mathrm{tt})$ to the term $T' = \mathrm{ff} \oplus \mathrm{ff}$ using our graph representation. At each step, the subgraph delimited by a dashed line is transformed by the *oneStepRewrite* function. In the graph representing T, each path set annotating an elementary or empty context node is initialized to the singleton set containing the path of the corresponding function symbol or empty context in T. In the second graph of the reduction depicted in Figure 13, note that two references to the *same* subgraph are created. This shared subgraph corresponds to the two residuals of the ff subterm in T. Finally, note that the last two steps apply rule **[B2]**, a collapse rule. This causes two *indirection* elementary nodes to be created.

```
fun oneStepRewrite(termGraph) =
    let ref C⟨ elemRef, _ ⟩ = termGraph
    and ref elemNode = elemRef in
        case elemNode of
            F⟨ ∧, [X, ref C⟨ ref F⟨ ⊕, [Y, Z], o1 ⟩, o2 ⟩ ], o3 ⟩ ⇒          (* Rule [B1] *)
                let o' = o1 ∪ o2 ∪ o3 in
                    elemRef := F⟨ ⊕,
                                  [ ref C⟨ ref F⟨ ∧, [X, Y], o' ⟩, o' ⟩,
                                    ref C⟨ ref F⟨ ∧, [X, Z], o' ⟩, o' ⟩ ],
                                  o' ⟩
                end

        |   F⟨ ∧, [X, ref C⟨ ref F⟨ tt, [ ], o1 ⟩, o2 ⟩ ], o3 ⟩ ⇒          (* Rule [B2] *)
                let o' = o1 ∪ o2 ∪ o3 in
                    elemRef := F⟨ ∇, [X], o' ⟩
                end

        |   F⟨ ∧, [X, ref C⟨ ref F⟨ ff, [ ], o1 ⟩, o2 ⟩ ], o3 ⟩ ⇒          (* Rule [B3] *)
                let o' = o1 ∪ o2 ∪ o3 in
                    elemRef := F⟨ ff, [ ], o' ⟩
                end

        |   _ ⇒ skip
    end
```

Figure 11: Compiled single-step reduction function for the linear TRS comprised of the first three rules of **B**.

```
fun compressOneIndirection(termGraph) =
    let ref contNode = termGraph
    and C⟨ elemRef, o1 ⟩ = contRef in
        case elemRef of
            ref F⟨ ∇, [ ref C⟨ childElemNode, o2 ⟩ ], o3 ⟩ ⇒
                let o' = o1 ∪ o2 ∪ o3 in
                    contNode := C⟨ childElemNode, o' ⟩
                end

        | _ ⇒ skip
    end
```

Figure 12: Compression function for indirection nodes in linear TRSs.

8.2.3 Compressing Indirections

The indirection nodes created during the reduction depicted in Figure 13 inhibit the application of further reduction rules. As a result, the administrative function *compressOneIndirection*, depicted in 12, is used where necessary to "compress" away the the indirections to yield a new graph to which further reduction rules can be applied. An indirection node can be thought of as a "proto" empty context node; each application of *compressOneIndirection* thus merges path sets for adjacent contexts during compression to compute the union of slices with respect to the single empty context node that exists after compression.

Figure 14 depicts the compression of the two indirection nodes in the last graph of Figure 13 (representing the term $T' = \mathrm{ff} \oplus \mathrm{ff}$) using function *compressOneIndirection*. In practice, indirections can be compressed on demand as required by the rule matching process, which avoids traversing the entire graph between each reduction step. However, to ensure that shared chains of indirections are not repeatedly compressed, function *compressOneIndirection* must be applied "bottom-up", that is in reverse topological sort order relative to the root of the subgraph being reduced.

8.2.4 Computing the Slice

To compute the slice with respect to any context represented by the final term of a reduction, one merely takes the union of path sets in the subgraph of the final graph that represents the slicing criterion. Thus, for example, the path sets corresponding to elementary or empty contexts in the subterm $\mathrm{ff} \oplus \bullet$ of the final term of the reduction represented by the last graph in Figure 14 are

$$\{(\,),(2)\}$$
$$\{(\,),(1),(2),(2,1)\}$$
$$\{(1)\}$$

Their union is

$$\{(\,),(1),(2),(2,1)\}$$

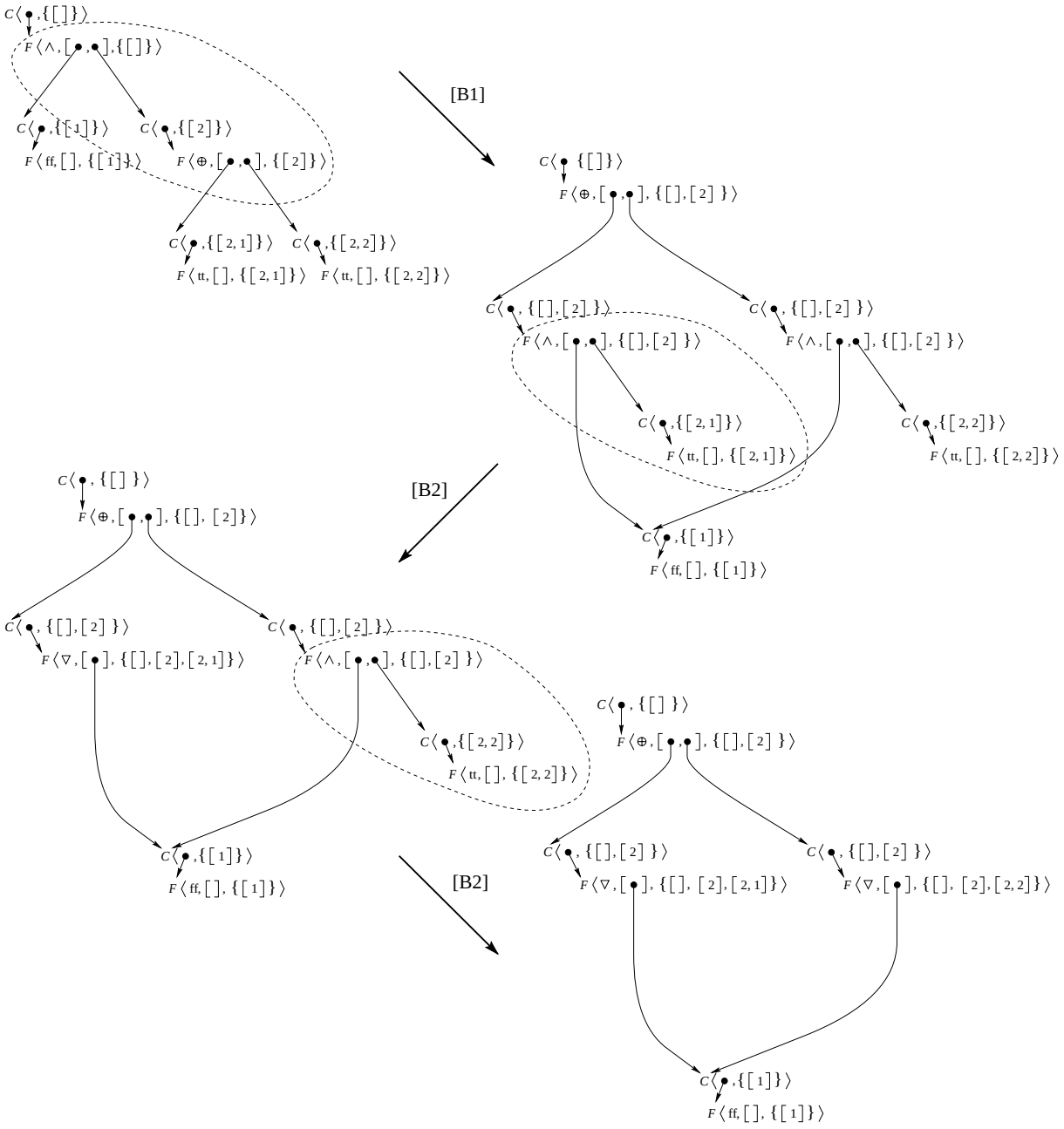


Figure 13: Graph reduction corresponding to the first three steps of the **B**-reduction of Figure 5. The subgraph transformed at each step by *oneStepRewrite* is delimited by a dashed line.

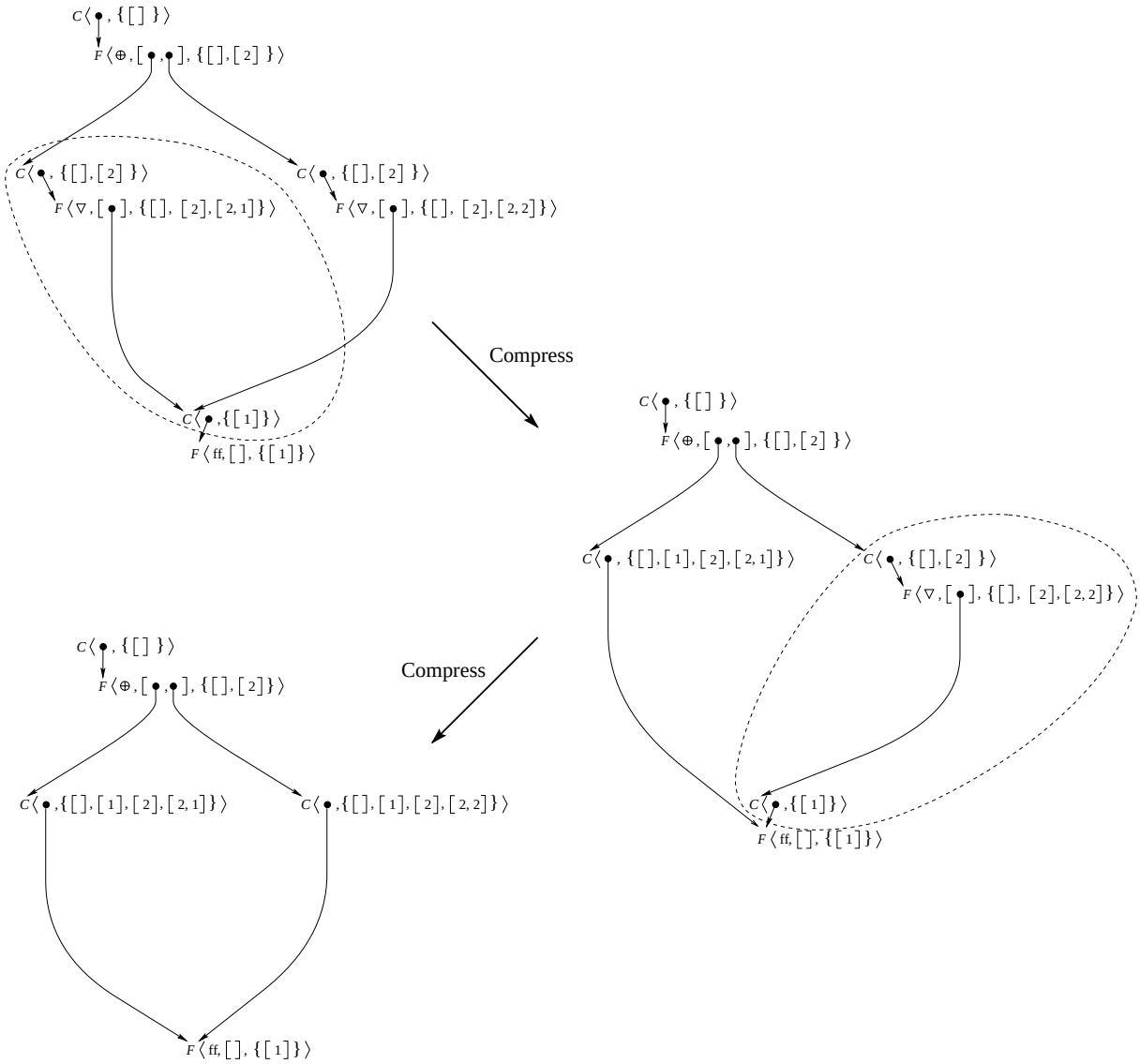


Figure 14: Compression of indirection nodes generated in Figure 13. The subgraph transformed at each step by *compressOneIndirection* is delimited by a dashed line.

$$
\begin{array}{lcl}
\textit{OriginSet} & ::= & \textit{Path}\ \textbf{set} \\
\textit{Path} & ::= & \textbf{int}\ \textbf{list} \\
\textit{Operator} & ::= & \nabla \\
 & | & \cdots \\
 & \vdots & \\
\textit{EmptyContextInfo} & ::= & \mathcal{O}\langle\ \textit{OriginSet},\ \textit{IndirectionList}\ \rangle \\
\textit{IndirectionList} & ::= & (\textit{ElementaryNode}\ \ \textbf{ref})\ \ \textbf{list} \\
\textit{EmptyContextNode} & ::= & \mathcal{C}\langle\ \textit{ElementaryNode}\ \ \textbf{ref},\ \textit{EmptyContextInfo}\ \rangle \\
\textit{ElementaryNode} & ::= & \mathcal{F}\langle\ \textit{Operator},\ \textit{TermGraph}\ \ \textbf{list},\ \textit{OriginSet}\ \rangle \\
\textit{TermGraph} & ::= & \textit{EmptyContextNode}\ \ \textbf{ref}
\end{array}
$$

Figure 15: Type definitions for data structures implementing terms in nonlinear TRSs.

which represents the context

$$\mathrm{ff} \wedge \mathrm{tt} \oplus \bullet$$

in the initial term.

8.3 Implementing Nonlinear Systems

8.3.1 Additional Data Structures

The type definitions for the data structures used to implement nonlinear systems are depicted in Figure 15. These definitions are almost identical to those used for linear systems. The only difference in this case is that *EmptyContextInfo* is now defined as a pair, built with the constructor $\mathcal{O}\langle\cdot,\cdot\rangle$. The first component of the pair is an *OriginSet,* and represents the same information as the *OriginSet* used to represent *EmptyContextInfo* data in the linear case. The second component of the pair is an *IndirectionList* containing a list of references to indirection *ElementaryNode*s. The indirections in the list are those that are "compressed away" in compression operations performed on the empty context with which the list is associated.

8.3.2 Reducing the Graph

Figure 16 depicts the updated single step rewriting function, *oneStepRewrite*$'$ used for the full system **B**. Note that the code for first three (nonlinear) rules is essentially unchanged from the linear version; the contents of the indirection list components of the *EmptyContextInfo* structures are ignored.

However, the code implementing the nonlinear rule

$$\textbf{[B4]} \qquad X \oplus X \longrightarrow \text{ff}$$

is more interesting. First, note that the two instances of the nonlinear variable X have been replaced by distinct instances X_1 and X_2. The function *termsEqual* is used to check whether the term representation of its term graph arguments are the same (we will not supply the details of *termsEqual*, which in its simplest form can be implemented as a recursive traversal over the "flattened" tree images of its term graph arguments). If the nonlinearly-matched term graphs are not equal, then the rule is not applied. If, however, the terms are equal, then another auxiliary function, *originsOfEqualTerms*, is used to compute the origin sets to be associated with the created contexts on the rule's right-hand side. We will discuss this function further in Section 8.3.4.

8.3.3 Compressing Indirections

The indirection compression function for nonlinear systems, *compressOneIndirection′*, is depicted in Figure 17. This function performs the same operation on origin sets as its linear counterpart in Figure 12. However, the nonlinear version also creates a new indirection list by first appending a reference to the compressed indirection node to the indirection list for the parent empty context node, then concatenating the indirection list for the child empty context node to the result (the list concatenation operation is denoted by '@'). The indirection list in the *EmptyContextInfo* structure thus contains all of the indirection nodes involved in any contiguous chain of applications of the compression function.

Figure 18 depicts the nonlinear analogue of the linear indirection compression operations depicted in Figure 14.

8.3.4 Computing the Origin Sets for Equal Terms

The code for *originsOfEqualTerms* and several related auxiliary functions are depicted in Figure 19. This function computes the union of origin sets representing pairs of equal elementary or empty contexts that do not have a common *progenitor* (see Section 7.1. Such subcontexts are treated as part of the *creating* context for a nonlinear rule.

Those elementary or empty contexts with a common progenitor are represented in term graph rewriting systems by the *same* graph node. In our implementation, the identity of two nodes can be determined by testing the references that point to them for equality.

originsOfEqualTerms compares its empty context reference arguments for equality. If the references are identical, then the origin set computed by the function is empty (i.e., the contexts are not part of the creating context since they have a common progenitor). If the references are not identical, then the functions *originsOfEqualElemRefs* and *originsOfReversedIndLists* are invoked to process the corresponding elementary nodes and the indirection lists. The indirection lists are *reversed* before being processed by *originsOfReversedIndLists*, since the lists do not necessarily have the same length, and any indirections shared by both lists must be part of a common list tail.

originsOfEqualElemRefs compares two references to elementary nodes. As in the empty context case, the empty origin set is returned if the references point to the same node. Otherwise, the argument lists are processed recursively using *originsOfEqualTermLists* (we will not depict its implementation here), and the result combined with the origin sets for the two elementary nodes.

```
fun oneStepRewrite′(termGraph) =
    let ref C⟨ elemRef, _ ⟩ = termGraph
    and ref elemNode = elemRef in
        case elemNode of
            F⟨ ∧, [X, ref C⟨ ref F⟨ ⊕, [Y, Z], o1 ⟩, O⟨ o2, _ ⟩ ⟩ ], o3 ⟩ ⇒      (* Rule [B1] *)
                let o′ = o1 ∪ o2 ∪ o3 in
                    elemRef := F⟨ ⊕,
                                  [ ref C⟨ ref F⟨ ∧, [X, Y], o′ ⟩, O⟨ o′, [ ] ⟩ ⟩,
                                    ref C⟨ ref F⟨ ∧, [X, Z], o′ ⟩, O⟨ o′, [ ] ⟩ ⟩ ],
                                  o′ ⟩
                end

        | F⟨ ∧, [X, ref C⟨ ref F⟨ tt, [ ], o1 ⟩, O⟨ o2, _ ⟩ ⟩ ], o3 ⟩ ⇒          (* Rule [B2] *)
                let o′ = o1 ∪ o2 ∪ o3 in
                    elemRef := F⟨ ∇, [X], o′ ⟩
                end

        | F⟨ ∧, [X, ref C⟨ ref F⟨ ff, [ ], o1 ⟩, O⟨ o2, _ ⟩ ⟩ ], o3 ⟩ ⇒          (* Rule [B3] *)
                let o′ = o1 ∪ o2 ∪ o3 in
                    elemRef := F⟨ ff, [ ], o′ ⟩
                end

        | F⟨ ⊕, [X1, X2 ], o1 ⟩ ⇒                                            (* Rule [B4] *)
                if termsEqual(X1, X2)
                    then
                        let o′ = o1 ∪ originsOfEqualTerms(X1, X2) in
                            elemRef := F⟨ ff, [ ], o′ ⟩
                        end
                    else skip

        | _ ⇒ skip
    end
```

Figure 16: Compiled single-step reduction function for the nonlinear full nonlinear TRS **B**.

```
fun compressOneIndirection'(termGraph) =
    let ref contNode = termGraph
    and C⟨ elemRef, O⟨ o1, indList1 ⟩ ⟩ = contRef in
        case elemRef of
            ref F⟨ ∇, [ ref C⟨ childElemNode, o2 ⟩ ], O⟨ o3, indList2 ⟩ ⟩ ⇒
                let o' = o1 ∪ o2 ∪ o3 in
                    contNode := C⟨ childElemNode, O⟨ o', indList1 @ [ elemRef ] @ indList2 ⟩ ⟩
                end

        | _ ⇒ skip
    end
```

Figure 17: Compression function for indirection nodes in nonlinear TRSs.

Finally, *originsOfReversedIndLists* processes reversed indirection lists by comparing their head elements (each of which is a reference to an indirection node) and ignoring the origins for those nodes whose references point to the same node. The auxiliary function *originsOfAllInds*, not depicted here, simply returns the union of the origin sets associated with all the indirection nodes in its argument list. Note that the operator ' :: ' is the list constructor.

8.3.5 Computing the Slice

Figure 20 depicts the result of applying rule **[B4]** to the compressed graph depicted in Figure 18. Note that the two indirections depicted, which are traversed by *originsOfEqualTerms* when computing the origin information for the nonlinearly-matched subgraphs, are *not* shared. Thus their origin information is merged into the created subgraph representing the final term, ff.

The slice with respect to the final term of the reduction, ff, is represented by the origin set associated with the single elementary node representing the term. This set is

$$\{(\,),(2),(2,1),(2,2)\}$$

and represents the context

$$\bullet \wedge \mathrm{tt} \oplus \mathrm{tt}$$

in the initial term.

8.3.6 Nonlinear Variables on Rule Right-Hand Sides

Our prototypical TRS **B** lacks a nonlinear rule in which the nonlinear variable appears on the term's right-hand side. In such cases, Definition 7.4 requires that the graph representing the term bound to the nonlinear variable on the right-hand side of the rule be constructed from two distinct classes of subgraphs:

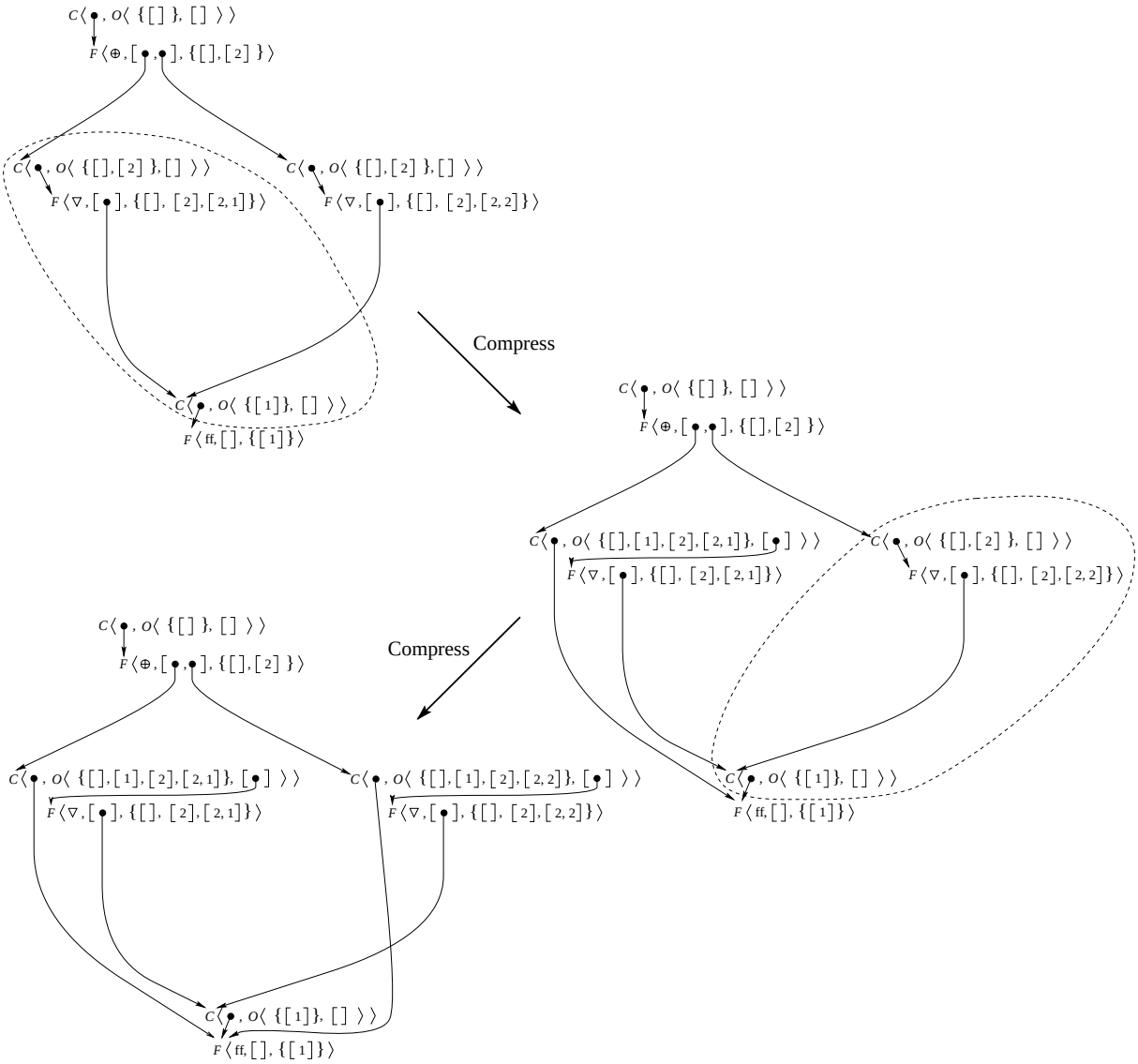


Figure 18: Compression of indirection nodes, nonlinear case. The subgraph transformed at each step by *compressOneIndirection′* is delimited by a dashed line.

```
fun originsOfEqualTerms(termGraph_1, termGraph_2) =
    if termGraph_1 = termGraph_2
        then ∅
        else
            let ref C⟨ elemRef_1, indList_1 ⟩ = termGraph_1
            and ref C⟨ elemRef_2, indList_2 ⟩ = termGraph_2 in
                originsOfEqualElemRefs(elemRef_1, elemRef_2)
                ∪ originsOfIndLists(reverse(indList_1), reverse(indList_2))
            end


fun originsOfEqualElemRefs(elemRef_1, elemRef_2) =
    if elemRef_1 = elemRef_2
        then ∅
        else
            let ref F⟨ _, operandList_1, o_1 ⟩ = elemRef_1
            and ref F⟨ _, operandList_2, o_2 ⟩ = elemRef_2 in
                o_1 ∪ o_2 ∪ originsOfEqualTermLists(operandList_1, operandList_2)
            end


fun originsOfReversedIndLists([ ], [ ]) =
    ∅

| originsOfReversedIndLists(indListHd_1 :: indListTl_1, [ ]) =
    originsOfAllInds(elemRef_1 :: indListTl_1)

| originsOfReversedIndLists([ ], indListHd_2 :: indListTl_2) =
    originsOfAllInds(elemRef_2 :: indListTl_2)

| originsOfReversedIndLists(elemRef_1 :: indListTl_1, elemRef_2 :: indListTl_2) =
    if elemRef_1 = elemRef_2
        then originsOfReversedIndLists(indListTl_1, indListTl_2)
        else
            let ref F⟨ _, _, o_1 ⟩ = elemRef_1
            and ref F⟨ _, _, o_2 ⟩ = elemRef_2 in
                o_1 ∪ o_2 ∪ originsOfReversedIndLists(indListTl_1, indListTl_2)
```

Figure 19: Functions for computing origin sets for pairs of equal terms.

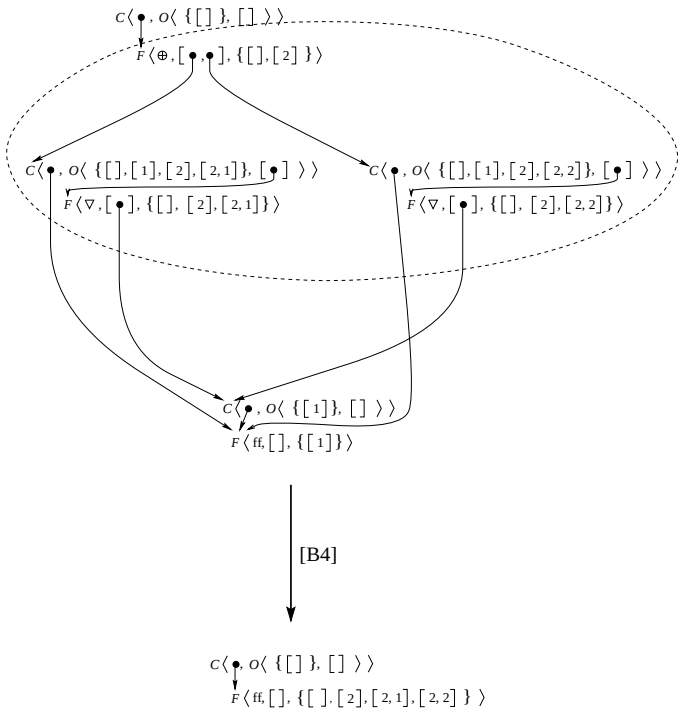


Figure 20: Graph reduction corresponding to the last step of the **B**-reduction of Figure 5. The "creating" context for this step is delimited by a dashed line.

- Shared *residuated* subgraphs, each of which represents a set of corresponding contexts in the equal terms, all of the latter of which have a common progenitor.
- A *created* subgraph corresponding to the remaining (non-residuated) context common to the equal terms.

8.3.7 Improving the Implementation of Equality Functions

As described in Figure 19, the function *originsOfEqualTerm* effectively traverses the "flattened" tree image of the graph. Unfortunately, it is easy to construct examples in which the size of the tree image of a term graph is *exponentially* larger than its graph representation. To avoid the possibility of incurring severe efficiency penalties when performing equality tests on such graphs, a variant of the Hopcroft/Karp algorithm for determining equivalence of finite automata [18] (described in [3, pp. 143–145]) can be used to partition the graph nodes representing equal terms into equivalence classes. The recursive traversal required by the origin set processing functions for equal subterms can then be carried out on a data structure representing the induced graph of equivalence classes (whose size is bounded by the sum of the sizes of the graphs representing the equal terms), rather than on the tree images of the term's graphs.

8.4 Complexity Issues

If path sets are implemented by bitmaps, the extra administrative steps required to track dynamic dependence information during each reduction step can be implemented in time 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 per reduction step of performing path set unions is *linear* in the size of the initial term.

If indirections are compressed in reverse topological order relative to the root of any subgraph to be transformed by a reduction step, then the number of compression steps necessary to eliminate all indirection nodes in any graph is proportional to the number of graph edges linking empty context nodes to indirection nodes. In any reduction sequence, the number of such edges generated is proportional to the sum of the size of the initial graph and the number of reduction steps in the sequence. Thus the overhead of indirection compression per reduction step is constant when amortized over any reduction sequence longer than the size of the initial term; otherwise the overhead is bounded by the size of the initial term.

In nonlinear systems, it is in principle possible to create chains of indirection nodes whose length is proportional to the length of the entire reduction. Traversing the indirection lists for such chains when processing the origin sets for equal terms could thus incur an overhead per reduction step linear in the size of the entire reduction. In practice (e.g., modeling C semantics in [14]), this phenomenon has not proved to be a problem. Long chains of indirections are rarely constructed, and it is even more unusual for such chains to be subsequently traversed during equality tests. If this phenomenon were to occur in practice, slices for nonlinear systems could be computed without the overhead of indirection traversal by using the *EmptyContextInfo* representation for nonlinear systems, albeit at the cost of computing larger than necessary slices in some cases.

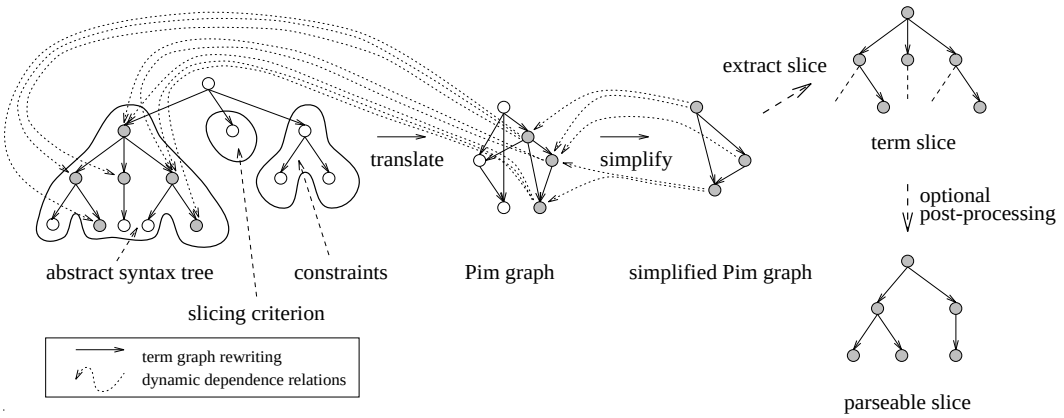


Figure 21: Overview of the parametric slicing approach of [14].

9 Applications

Dynamic dependence tracking has been applied successfully to a number of different application domains.

9.1 Parametric Program Slicing

In [14], dynamic dependence tracking is applied to Pim, [5], an intermediate program representation whose behavior is defined by an equational logic. A subsystem of the full Pim system can be used not only to execute programs, but also to perform various kinds of analysis and optimizations by simplification of a program's Pim representation.

To compute the slice of a program with respect to the final value of a variable x, a term is constructed that "encodes" (i) the abstract syntax tree (AST) of the program possibly containing meta-variables denoting unknown values or inputs, (ii) the variable x that represents the slicing criterion, and (iii) a (possibly empty) set of additional constraints on meta-variables. Next, the AST is translated to a graph comprising its Pim representation. This translation is assumed to be defined by a rewriting system (although it need not necessarily be implemented that way). The resulting graph is then simplified by repeated application of sets of rewriting rules derived from the Pim logic. The graph that results from the reduction process represents the final value of variable x (in terms of the unconstrained meta-variables). Tracing back the dynamic dependence relations from this graph to the AST of the program yields the desired slice, as a subcontext of the AST. If this slice is required to be a parseable representation of the program, an optional post-processing step is performed. The approach is illustrated in Figure 21.

The approach of [14] is distinguished by the fact that changes to the behavior of the slicing algorithm can be accomplished through simple changes in the rewriting rules that define the semantics of the program representation. Thus, e.g., different notions of dependence may be specified, properties of language-specific datatypes can be exploited, and various time, space, and precision tradeoffs may be made. This flexibility enables the generalization of the traditional notions of static and dynamic slices to that of a *constrained* slice, where any subset of the inputs of a program may be supplied.

9.2 Dynamic Program Slicing

In [28], dependence tracking is applied to the algebraic specification of an interpreter for the language ClaX, a Pascal subset. This specification consists of a set of conditional rewriting rules, which can be executed by conditional term rewriting. The specification starts with a term representing the program's AST, and constructs a term that represents the stack of activation records containing the current value for each variable. Execution of a statement is modeled by rewriting the activation stack appropriately. A dynamic slices with respect to the current value of a variable v is computed by tracing back the dynamic dependence relations from the subterm representing v's value. The resulting slices can be postprocessed (i.e., $\bullet$-subterms are transformed away) for the sake of enhancing readability.

9.3 Locating Type Errors

Dinesh and Tip [11] present an approach where the behavior of a type checker is algebraically specified by way of a set of conditional equations. This type checker specification is executed by way of conditional term rewriting. The rewriting rules of this system express the type checking process by transforming a program's abstract syntax tree (AST) into a list of error messages. Dynamic dependence tracking is used to associate a *slice* P_e with each type error e that occurs when type-checking program P. This slice serves as the positional information associated with an error message, but it has an interesting semantic property: type-checking P_e is guaranteed to produce the same type error e.

The approach of [11] has been implemented for the language ClaX, a significant subset of Pascal. Figure 22 shows a snapshot of the generated error reporting tool. A discussion of the engineering issues that came up during the implementation of the ClaX environment can be found in [10].

9.4 Tracing the Origins of Verification Conditions

In his Ph.D. thesis [17], Fraer presents a variation of dynamic dependence tracking for systems based on inference rules instead of rewrite rules. Fraer uses dependence tracking for tracing the origins of verification conditions generated by a Verification Condition Generator (VCG) [16]. A VCG takes as input an imperative program annotated with Hoare logic assertions (pre/postconditions and loop invariants) and outputs a list of verification conditions that are submitted as input to a theorem prover. Dependence tracking is used in situations where the proof of some verification condition fails, in order to determine what program components or verification conditions need to be modified. Fraer implements dependence tracking by instrumenting the inference rules and terms to propagate dependence information.

10 Related Work

The term "slice" was first coined by Weiser [32], and defined for imperative programming languages using dataflow analysis. Subsequent work, beginning with that of Ottenstein and Ottenstein [25], has focused on use of *program dependence graphs* [12] for computing slices. Cartwright and Felleisen [8] and Venkatesh [30] 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. [29] provides a survey of current work on program slicing.

Figure 22: The ClaX environment. The top window is a program editor with two buttons attached to it for invoking a type checker and an interpreter, respectively. The middle window shows a list of four type errors reported by the type checker, in which the error message "in-call expected-arg VAR INTEGER found-arg REAL" is selected, indicating a mismatch between formal and actual parameter types in a procedure call. The bottom window shows the slice computed for this error message, containing all program components that contributed to the selected type error. The '<?>' symbols in the slice denote placeholders for program constructs not contributing to the error message.

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 [6, 7] 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* [19]. Van Deursen, Klint and Tip [9], 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 [21] (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 [31] 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 [22, 23] 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.

Abadi et al. independently developed a technique similar to dependence tracking for several variations of the λ-calculus [1]. The motivation of their work is incremental evaluation: their work is stated as a technique for "caching" results of rewriting λ-terms in order to avoid the subsequent rewriting of similar terms. In the approach of [1], some or all of the function symbols in a λ-term are labeled with a unique label, and additional rules are introduced for pushing labels outward so that the "standard" reductions can take place. After normalizing a λ-term, the labels that remain in the normal form are precisely those of the context (prefix, using their terminology) that was needed to perform the reduction (i.e., the term slice). The approach by Abadi et al. is more limited than ours in a number of respects:

- The approach of [1] heavily relies on an *outermost* reduction strategy. If an innermost rewriting strategy were to be used, all labels would be pushed outward before they could be eliminated, and the computed slice would consist of the entire term. Our approach is not limited in this respect.

- Abadi et al. only consider a number of variations on the λ-calculus, which are particularly well-behaved TRSs. Although their approach could easily be extended to other TRSs that are orthogonal [21], its practical applications are severely limited by the inability to deal with nonlinear rewriting rules and collapse rules.

11 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.

We have begun efforts to extend our techniques to *conditional* rewriting systems and lambda calculi (i.e., systems with variable-binding constructs). A preliminary implementation of these ideas has proved promising, but a formal study remains to be done.

Acknowledgments

We are grateful to G. Ramalingam for his comments and for his significant contributions to the implementation the dependence tracking technique. We also thank Jan Heering for commenting on earlier drafts of this paper.

References

[1] ABADI, M., LAMPSON, B., AND LÉVY, J.-J. Analysis and caching of dependencies. In *Proceedings of 1996 ACM SIGPLAN International Conference on Functional Programming* (1996), pp. 83–91. ACM SIGPLAN Notices 31(6).

[2] 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.

[3] AHO, A., HOPCROFT, J., AND ULLMAN, J. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[4] BARENDREGT, H., VAN EEKELEN, M., GLAUERT, J., KENNAWAY, J., PLASMEIJER, M., AND SLEEP, M. Term graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages* (Eindhoven, The Netherlands, 1987), Springer-Verlag, pp. 141–158. Lecture Notes in Computer Science 259.

[5] BERGSTRA, J., DINESH, T., FIELD, J., AND HEERING, J. Toward a complete transformational toolkit for compilers. *ACM Trans. on Programming Languages and Systems 5*, 19 (September 1997), 639–684.

[6] BERTOT, Y. Occurrences in debugger specifications. In *Proc. ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation* (Toronto, June 1991), pp. 327–336.

[7] BERTOT, Y. Origin functions in λ-calculus and term rewriting systems. In *Proc. Seventeenth CAAP* (1992), J.-C. Raoult, Ed., vol. 581 of *LNCS*, Springer-Verlag, pp. 49–64.

[8] 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.

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

[10] DINESH, T. B., AND TIP, F. A case study of a slicing-based approach for locating type errors. In *Proceedings of the Second International Workshop on Theory and Practice of Algebraic Specifications (ASF+SDF'97)* (Amsterdam, The Netherlands, September 1997), M. P. A. Sellink, Ed., electronic Workhops in Computing, Springer-Verlag.

[11] DINESH, T. B., AND TIP, F. A slicing-based approach for locating type errors. In *Proceedings of the USENIX Conference on Domain-Specific Languages (DSL'97)* (Santa Barbara, CA, October 15–17 1997), pp. 77–88.

[12] 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.

[13] 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.

[14] FIELD, J., RAMALINGAM, G., AND TIP, F. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[15] 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* (September 1994), M. Hermenegildo and J. Penjam, Eds., vol. 844, Springer-Verlag, pp. 415–431.

[16] FRAER, R. Reasoning with executable specifications. In *Proceedings of the Sixth International Joint Conference on Theory and Practice of Software Development* (Aarhus, Denmark, May 1995), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., vol. 915 of *LNCS*, Springer-Verlag, pp. 531–545.

[17] FRAER, R. *Analyse de Programmes Annotés par des Assertions*. PhD thesis, L'Université de Nice, Sophia Antipolis, November 1997. In French.

[18] HOPCROFT, J., AND KARP, R. An algorithm for testing the equivalence of finite automata. Tech. Rep. TR-71-114, Dept. of Computer Science, Cornell University, Ithaca, NY, 1971.

[19] KAHN, G. Natural semantics. In *Fourth Annual Symp. on Theoretical Aspects of Computer Science* (1987), vol. 247 of *LNCS*, Springer-Verlag, pp. 22–39.

[20] KENNAWAY, J. R., KLOP, J. W., SLEEP, M. R., AND DE VRIES, F. J. On the adequacy of graph rewriting for simulating term rewriting. *ACM Trans. on Programming Languages and Systems 16*, 3 (1994), 493–523.

[21] 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.

[22] 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.

[23] MARANGET, L. *La Stratégie Paresseuse*. PhD thesis, Université de Paris VIII, 1992. (in French).

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

[25] 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).

[26] PEYTON JONES, S. L. *The Implementation of Functional Programming Languages*. Prentice Hall International, Englewood Cliffs, NJ, 1987.

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

[28] TIP, F. Generic techniques for source-level debugging and dynamic program slicing. In *Proceedings of the Sixth International Joint Conference on Theory and Practice of Software Development* (Aarhus, Denmark, May 1995), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., vol. 915 of *LNCS*, Springer-Verlag, pp. 516–530.

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

[30] 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.

[31] WADLER, P., AND HUGHES, R. Projections for strictness analysis. In *Proc. Conf. on Functional Programming and Computer Architecture* (Portland, OR, September 1987), vol. 274 of *LNCS*, Springer-Verlag, pp. 385–406.

[32] WEISER, M. Program slicing. *IEEE Trans. on Software Engineering SE-10*, 4 (1989), 352–357.