Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 177 (2007) 123–136
www.elsevier.com/locate/entcs

Static Slicing of Rewrite Systems[1]

Diego Cheda[2] Josep Silva[2] Germán Vidal[2]

DSIC, Technical University of Valencia
Camino de Vera S/N, 46022 Valencia, Spain

Abstract

Program slicing is a method for decomposing programs by analyzing their data and control flow. Slicing-based techniques have many applications in the field of software engineering (like program debugging, testing, code reuse, maintenance, etc). Slicing has been widely studied within the imperative programming paradigm, where it is often based on the so called *program dependence graph*, a data structure that makes explicit both the data and control dependences for each operation in a program. Unfortunately, the notion of "dependence" cannot be easily adapted to a functional context. In this work, we define a novel approach to *static* slicing (i.e., independent of a particular input data) for first-order functional programs which are represented by means of rewrite systems. For this purpose, we introduce an appropriate notion of dependence that can be used for computing program slices. Also, since the notion of static slice is generally undecidable, we introduce a complete approximation for computing static slices which is based on the construction of a *term dependence graph*, the counterpart of program dependence graphs.

Keywords: Program slicing, rewrite systems

1 Introduction

Program slicing [13] is a method for decomposing programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of those program statements which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. In imperative programming, slicing criteria are usually given by a pair $(program\_line, variable)$.

Example 1.1 *Consider the program in Figure 1 to compute the number of characters and lines of a text. A slice of this program w.r.t. the slicing criterion* $(12, chars)$ *would contain the black sentences (while the gray sentences are discarded). This slice contains all those parts of the program which are necessary to compute the value of variable chars at line* 12.

[1] This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2005-09207-C03-02, by the ICT for EU-India Cross-Cultural Dissemination Project ALA/95/23/2003/077-054, by LERNet AML/19.0902/97/0666/II-0472-FA and by the *Vicerrectorado de Innovación y Desarrollo de la UPV* under project TAMAT ref. 5771.

[2] Email: {dcheda,jsilva,gvidal}@dsic.upv.es

doi:10.1016/j.entcs.2007.01.010

```
(1)    lineCharCount(str)
(2)      i:=1;
(3)      lines:=0;
(4)      chars:=0;
(5)      while (i<length(str)) do
(6)        if (str[i] == CR)
(7)        then lines := lines + 1;
(8)             chars := chars + 1;
(9)        else chars := chars + 1;
(10)       i = i + 1;
(11)   return lines;
(12)   return chars;
```

Fig. 1. Example program `lineCharCount`

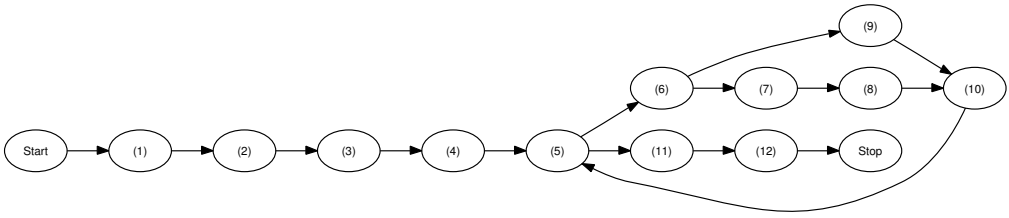

Fig. 2. Control flow graph of `lineCharCount`

In order to extract a slice from a program, the dependences between its statements must be computed first. The *control flow graph* (CFG) is a data structure which makes the control dependences for each operation in a program explicit. For instance, the CFG of the program in Fig. 1 is depicted in Fig. 2.

However, the CFG does not generally suffice for computing program slices because it only stores control dependences and, for many applications (such as debugging), data dependences are also necessary. For this reason, in imperative programming, program slices are usually computed from a *program dependence graph* (PDG) [4,6] that makes explicit both the data and control dependences for each operation in a program. A PDG is an oriented graph where the nodes represent statements in the source code and the edges represent data and control dependences. As an example, the PDG of the program in Fig. 1 is depicted in Fig. 3 where solid arrows represent control dependences and dotted arrows represent data dependences.

Program dependences can be traversed backwards or forwards (from the slicing criterion), which is known as *backward* or *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. A complete survey on slicing can be found, e.g., in [12].

While PDGs are good to represent the data and control flow behavior of imperative programs, their level of granularity (i.e., considering all function bodies as a whole) is not appropriate for representing dependences in functional programming.

In this work, we present a new notion of *dependence* in term rewriting that can be used to give an appropriate definition of static slicing. Unfortunately, the

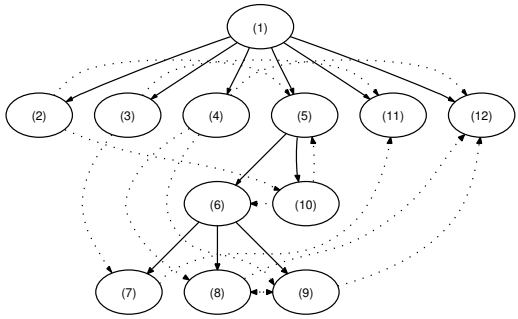

Fig. 3. Program dependence graph of `lineCharCount`

computation of static slices is generally undecidable—since one should consider all possible computations for a given program—and, thus, we also introduce a complete algorithm to compute static slices which is based on the construction of *term dependence graphs*, a new formalism to represent both data and control flow dependences of first-order functional programs denoted by term rewriting systems.

The rest of the paper is organized as follows. In the next section, we recall some notions on term rewriting that will be used throughout the paper. Then, in Section 3, we present our approach to static slicing within a functional context. Section 4 introduces a new data structure, called term dependence graph, than can be used to compute static slices. Finally, Section 5 discusses some related work and concludes.

2 Preliminaries

For completeness, here we recall some basic notions of term rewriting. We refer the reader to [3] for details.

A set of rewrite rules (or oriented equations) $l \to r$ such that l is a nonvariable term and r is a term whose variables appear in l is called a *term rewriting system* (TRS for short); terms l and r are called the left-hand side and the right-hand side of the rule, respectively. We assume in the following that all rules are numbered uniquely. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}$ are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. We restrict ourselves to finite signatures and TRSs. We denote the domain of terms and *constructor terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectively, where $\mathcal{V}$ is a set of variables with $\mathcal{F} \cap \mathcal{V} = \emptyset$.

A TRS $\mathcal{R}$ is *constructor-based* if the left-hand side of its rules have the form $f(s_1, \ldots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \ldots, n$. The root symbol of a term t is denoted by $root(t)$. A term t is *operation-rooted* (resp. *constructor-rooted*) if $root(t) \in \mathcal{D}$ (resp. $root(t) \in \mathcal{C}$). The set of variables appearing in a term t is denoted by $\mathcal{V}ar(t)$. A term t is *linear* if every variable of $\mathcal{V}$ occurs at most once in t. $\mathcal{R}$ is left-linear (resp. right-linear) if l (resp. r) is linear for all rules $l \to r \in \mathcal{R}$. In this paper, we restrict ourselves to left-linear constructor-based TRSs, which we often call *programs*.

As it is common practice, a *position* p in a term t is represented by a sequence of natural numbers, where Λ denotes the root position. Positions are used to address the nodes of a term viewed as a tree: $t|_p$ denotes the *subterm* of t at position p and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s. A term t is *ground* if $\mathcal{V}ar(t) = \emptyset$. A *substitution* σ is a mapping from variables to terms such that its domain $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is finite. The identity substitution is denoted by *id*. Term t' is an *instance* of term t if there is a substitution σ with $t' = \sigma(t)$. A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. In the following, we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

A rewrite step is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l \to r)$ and a substitution σ with $t|_p = \sigma(l)$, $s = t[\sigma(r)]_p$ (p and R will often be omitted in the notation of a reduction step). The instantiated left-hand side $\sigma(l)$ is called a *redex*. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to s$. We denote by $\to^+$ the transitive closure of $\to$ and by $\to^*$ its reflexive and transitive closure. Given a TRS $\mathcal{R}$ and a term t, we say that t *evaluates* to s iff $t \to^* s$ and s is in normal form.

3 Static Slicing of Rewrite Systems

In this section, we introduce our notion of *dependence* in term rewriting, which is then used to give an appropriate definition of static slicing. First, we define the *program position* of a term, which uniquely determines its location in the program.

Definition 3.1 (position, program position) *Positions are represented by a sequence of natural numbers, where Λ denotes the empty sequence (i.e., the root position). They are used to address subterms of a term viewed as a tree:*

$$t|_\Lambda = t \;\; \textit{for all term } t \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \qquad d(\overline{t_n})|_{i.w} = t_i|_w \;\; \textit{if} \;\; i \in \{1, \ldots, n\},\; d/n \in \mathcal{F}$$

A program position is a pair (k, w) that addresses the (possibly variable) subterm $r|_w$ in the right-hand side r of the k-th rule $l \to r$ of $\mathcal{P}$. Given a program $\mathcal{P}$, $\mathcal{P}os(\mathcal{P})$ denotes the set of all program positions of the terms in the right-hand sides of $\mathcal{P}$.

Definition 3.2 (labeled term) *Given a program P, a labeled term is a term in which each (constructor or defined) function or variable symbol is labeled with a set of program positions from $\mathcal{P}os(\mathcal{P})$. The domain of labeled terms can be inductively defined as follows:*

- *a^P is a labeled term, with $a \in \mathcal{F} \cup \mathcal{V}$ and $P \subseteq \mathcal{P}os(\mathcal{P})$;*
- *if $d/n \in \mathcal{F}$, $P \subseteq \mathcal{P}os(\mathcal{P})$ and $t_1, \ldots, t_n$ are labeled terms, then $d^P(t_1, \ldots, t_n)$ is also a labeled term.*

In the remainder of this paper, we assume the following considerations:

- The right-hand sides of program rules are labeled.
- Labels do not interfere with the standard definitions of pattern matching, instance, substitution, rewrite step, derivation, etc (i.e., labels are ignored).

- The application of a (labeled) substitution σ to a term t is redefined so that, for each binding $x \mapsto d^{P_0}(t_1^{P_1}, \ldots, t_n^{P_n})$ of σ, if variable x^P occurs in t, then it is replaced by $d^{P \cup P_0}(t_1^{P_1}, \ldots, t_n^{P_n})$ in $\sigma(t)$.

The next example shows why we need to associate a set of program positions with every term and not just a single position:

Example 3.3 *Consider the following labeled program:*[3]

$$\begin{array}{lll} (\mathtt{R1}) & \mathtt{main} & \to \mathtt{g}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{Z}^{\{(\mathtt{R1},1.1)\}})) \\ (\mathtt{R2}) & \mathtt{g(x)} & \to \mathtt{x}^{\{(\mathtt{R2},\Lambda)\}} \\ (\mathtt{R3}) & \mathtt{f(x)} & \to \mathtt{x}^{\{(\mathtt{R3},\Lambda)\}} \end{array}$$

together with the derivation:

$$\begin{array}{ll} \underline{\mathtt{main}}^{\{\}} & \to_{\Lambda,\mathtt{R1}} \mathtt{g}^{\{(\mathtt{R1},\Lambda)\}}(\underline{\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{Z}^{\{(\mathtt{R1},1.1)\}})}) \\ & \to_{1,\mathtt{R3}} \underline{\mathtt{g}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{Z}^{\{(\mathtt{R3},\Lambda),(\mathtt{R1},1.1)\}})} \\ & \to_{\Lambda,\mathtt{R2}} \mathtt{Z}^{\{(\mathtt{R2},\Lambda),(\mathtt{R3},\Lambda),(\mathtt{R1},1.1)\}} \end{array}$$

Here, one may argue that the list of program positions of Z *in the final term of the derivation should only contain the pair* $(\mathtt{R1}, 1.1)$ *(i.e., the only occurrence of* Z *in the right-hand side of the first rule). However, for program slicing, it is also relevant to know that* Z *has been propagated through the variable* x *that appears in the right-hand sides of both the second and third rules.*

Observe that main *is labeled with an empty set of program positions since it does not appear in the right-hand side of any program rule.*

Before introducing our notion of "dependence", we need the following definition.[4] Here, we let "$\bullet$" be a fresh constructor symbol not occurring in $\mathcal{F}$ (which is not labeled). We use this symbol to denote a *missing* subterm.

Definition 3.4 (subreduction) *Let $\mathcal{D} : t_0 \to_{p_1,R_1} \ldots \to_{p_n,R_n} t_n$ be a derivation for t_0 in a program $\mathcal{P}$, with $t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. We say that $\mathcal{D}' : t'_0 \to_{p'_1,R'_1} \ldots \to_{p'_m,R'_m} t'_m$, $m \leq n$, is a subreduction of $\mathcal{D}$ if the following conditions hold:*

(i) *$t'_0 = t_0[\bullet]_p$ for some position p,*

(ii) *t'_m is a normal form, and*

(iii) *all the elements in the sequence $(p'_1, R'_1), \ldots, (p'_m, R'_m)$ also appear in the sequence $(p_1, R_1), \ldots, (p_n, R_n)$ and in the same order.*

Roughly, we say that a derivation is a subreduction of another derivation if

- the initial terms are equal except possibly for some missing subterm,

[3] In the examples, data constructor symbols start with uppercase letters while defined functions and variables start with lowercase letters. Furthermore, we underline the selected redex at each reduction step.

[4] A similar, though more general, notion of subreduction can be found in [5].

- both derivations end with a normal form (i.e., an irreducible term), and
- the same steps, and in the same order, are performed in both derivations except for some steps that cannot be performed because of the missing subterm in the initial term (and its descendants).

Example 3.5 *Consider the following labeled program:*

$$\begin{array}{lll} (\mathtt{R1}) & \mathtt{main} & \to \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{A}^{\{(\mathtt{R1},1.1)\}}), \mathtt{g}^{\{(\mathtt{R1},2)\}}(\mathtt{B}^{\{(\mathtt{R1},2.1)\}})) \\ (\mathtt{R2}) & \mathtt{f(A)} & \to \mathtt{D}^{\{(\mathtt{R2},\Lambda)\}} \\ (\mathtt{R3}) & \mathtt{g(x)} & \to \mathtt{x}^{\{(\mathtt{R3},\Lambda)\}} \end{array}$$

together with the derivation

$$\begin{array}{rl} \mathcal{D}: & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\underline{\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{A}^{\{(\mathtt{R1},1.1)\}})}, \mathtt{g}^{\{(\mathtt{R1},2)\}}(\mathtt{B}^{\{(\mathtt{R1},2.1)\}})) \\ \to_{1,\mathtt{R2}} & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}^{\{(\mathtt{R2},\Lambda)\}}, \underline{\mathtt{g}^{\{(\mathtt{R1},2)\}}(\mathtt{B}^{\{(\mathtt{R1},2.1)\}})}) \\ \to_{2,\mathtt{R3}} & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}^{\{(\mathtt{R2},\Lambda)\}}, \mathtt{B}^{\{(\mathtt{R3},\Lambda),(\mathtt{R1},2.1)\}}) \end{array}$$

Then, for instance, the following derivations:

$$\begin{array}{rl} \mathcal{D}_1: & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\underline{\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{A}^{\{(\mathtt{R1},1.1)\}})}, \mathtt{g}^{\{(\mathtt{R1},2)\}}(\bullet)) \\ \to_{1,\mathtt{R2}} & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}^{\{(\mathtt{R2},\Lambda)\}}, \underline{\mathtt{g}^{\{(\mathtt{R1},2)\}}(\bullet)}) \\ \to_{2,\mathtt{R3}} & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}^{\{(\mathtt{R2},\Lambda)\}}, \bullet) \end{array}$$

$$\begin{array}{rl} \mathcal{D}_2: & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\underline{\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{A}^{\{(\mathtt{R1},1.1)\}})}, \bullet) \\ \to_{1,\mathtt{R2}} & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}^{\{(\mathtt{R2},\Lambda)\}}, \bullet) \end{array}$$

$$\begin{array}{rl} \mathcal{D}_3: & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{f}^{\{(\mathtt{R1},1)\}}(\bullet), \underline{\mathtt{g}^{\{(\mathtt{R1},2)\}}(\mathtt{B}^{\{(\mathtt{R1},2.1)\}})}) \\ \to_{2,\mathtt{R3}} & \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{f}^{\{(\mathtt{R1},1)\}}(\bullet), \mathtt{B}^{\{(\mathtt{R3},\Lambda),(\mathtt{R1},2.1)\}}) \end{array}$$

are subreductions of $\mathcal{D}$.

Now, we can introduce our notion of dependence in term rewriting. Informally speaking, given a function call $f(\overline{v_n})$ that evaluates to a constructor term v, we say that a subterm of v, say s_2, depends on a term s_1 (that appears in the program) if

- there exists a derivation of the form $f(\overline{v_n}) \to^* t \to^* v$ where s_1 is a subterm of t and
- if s_1 is replaced in t by the fresh symbol $\bullet$, then the root symbol of s_2 is not computed anymore in the considered value v.

Definition 3.6 (dependence) *Given a program P, a constructor term s_2 depends on the term s_1 w.r.t. function f of P, in symbols $s_1 \rightsquigarrow_f s_2$, iff there exists a*

derivation of the form $f(\overline{v_n}) \to^* t \to^* v$, *where* $\overline{v_n}$ *and* v *are constructor terms,* $t|_p = s_1$ *and* $v|_q = s_2$ *for some positions* p *and* q, *and there is a subreduction* $t' \to^* v'$ *of a suffix* $t \to^* v$ *of the derivation* $f(\overline{v_n}) \to^* t \to^* v$ *with* $t' = t[\bullet]_p$ *such that* $root(v'|_q) \neq root(s_2)$.

Example 3.7 *Consider again the program of Example 3.5. Here,* $\mathtt{A} \rightsquigarrow_{\mathtt{main}} \mathtt{D}$ *because we have a derivation*

$$\mathtt{main} \to \overbrace{\mathtt{C}(\underline{\mathtt{f}(\mathtt{A})}, \mathtt{g}(\mathtt{B})) \to \mathtt{C}(\mathtt{D}, \underline{\mathtt{g}(\mathtt{B})}) \to \mathtt{C}(\mathtt{D}, \mathtt{B})}^{\mathcal{D}}$$

with $\mathtt{C}(\mathtt{D},\mathtt{B})|_1 = \mathtt{D}$, $\mathtt{C}(\mathtt{f}(\mathtt{A}),\mathtt{g}(\mathtt{B}))|_{1.1} = \mathtt{A}$, *and in the following subreduction of* $\mathcal{D}$:

$$\mathtt{C}(\mathtt{f}(\bullet), \underline{\mathtt{g}(\mathtt{B})}) \to \mathtt{C}(\mathtt{f}(\bullet), \mathtt{B})$$

we have $root(\mathtt{C}(\mathtt{f}(\bullet),\mathtt{B})|_1) = \mathtt{f} \neq \mathtt{D}$.

Note that we are not fixing any particular strategy in the definition of dependence. Our aim is to produce static slices which are independent of any evaluation strategy. Now, we introduce the basic concepts of our approach to static slicing.

For the definition of slicing criterion, we recall the notion of slicing patterns:

Definition 3.8 (slicing pattern [8]) *The domain Pat of slicing patterns is defined as follows:*

$$\pi \in Pat \ ::= \ \bot \mid \top \mid c(\overline{\pi_k})$$

where $c/k \in \mathcal{C}$ *is a constructor symbol of arity* $k \geq 0$, $\bot$ *denotes a subexpression of the value whose computation is not relevant and* $\top$ *a subexpression which is relevant.*

Slicing patterns are similar to the *liveness patterns* which are used to perform dead code elimination in [7]. Basically, they are *abstract* terms that can be used to denote the *shape* of a constructor term by ignoring part of its term structure. For instance, given the constructor term $\mathtt{C}(\mathtt{A},\mathtt{B})$, we can use (among others) the following slicing patterns $\top$, $\bot$, $\mathtt{C}(\top,\top)$, $\mathtt{C}(\top,\bot)$, $\mathtt{C}(\bot,\top)$, $\mathtt{C}(\bot,\bot)$, $\mathtt{C}(\mathtt{A},\top)$, $\mathtt{C}(\mathtt{A},\bot)$, $\mathtt{C}(\top,\mathtt{B})$, $\mathtt{C}(\bot,\mathtt{B})$, $\mathtt{C}(\mathtt{A},\mathtt{B})$, depending on the available information and the relevant fragments of $\mathtt{C}(\mathtt{A},\mathtt{B})$.

Given a slicing pattern π, the concretization of an abstract term is formalized by means of function γ, so that $\gamma(\pi)$ returns the set of terms that can be obtained from π by replacing all occurrences of both $\top$ and $\bot$ by any constructor term. This usually leads to an infinite set, e.g., $\gamma(\mathtt{C}(\mathtt{A},\top)) = \gamma(\mathtt{C}(\mathtt{A},\bot)) = \{\mathtt{C}(\mathtt{A},\mathtt{A}), \mathtt{C}(\mathtt{A},\mathtt{B}), \mathtt{C}(\mathtt{A},\mathtt{D}), \mathtt{C}(\mathtt{A},\mathtt{C}(\mathtt{A},\mathtt{A})), \mathtt{C}(\mathtt{A},\mathtt{C}(\mathtt{A},\mathtt{B})), \ldots\}$.

Definition 3.9 (slicing criterion) *Given a program* $\mathcal{P}$, *a slicing criterion for* $\mathcal{P}$ *is a pair* (f,π) *where* f *is a function symbol and* π *is a slicing pattern.*

Now, we introduce our notion of static slice. Basically, given a slicing criterion (f,π), a program slice is given by the set of program positions of those terms in the program that affect the computation of the relevant parts—according to π—of the possible values of function f. Formally,

Definition 3.10 (slice) *Let* $\mathcal{P}$ *be a program and* (f,π) *a slicing criterion for* $\mathcal{P}$. *Let* P_π *be the set of positions of* π *which do not address a symbol* $\bot$. *Then, the slice*

of $\mathcal{P}$ w.r.t. (f, π) is given by the following set of program positions:

$$\bigcup \{(k,w) \mid (k,w) \in P,\ t^P \rightsquigarrow_f v|_q,\ v \in \gamma(\pi) \text{ and } q \in P_\pi\}$$

Observe that a slice is a subset of the program positions of the original program that uniquely identifies the program (sub)terms that belong to the slice.

Example 3.11 *Consider again the program of Example 3.5 and the slicing pattern* $(\mathtt{main}, \mathtt{C}(\top,\bot))$.

- *The concretizations of* $\mathtt{C}(\top,\bot)$ *are* $\gamma(\mathtt{C}(\top,\bot)) = \{\mathtt{C}(\mathtt{A},\mathtt{A}),\ \mathtt{C}(\mathtt{A},\mathtt{B}),\ \mathtt{C}(\mathtt{A},\mathtt{D}),\ \mathtt{C}(\mathtt{B},\mathtt{A}),\ \mathtt{C}(\mathtt{B},\mathtt{B}),\ \mathtt{C}(\mathtt{B},\mathtt{D}),\ \mathtt{C}(\mathtt{D},\mathtt{A}),\ \mathtt{C}(\mathtt{D},\mathtt{B}),\ \mathtt{C}(\mathtt{D},\mathtt{D}),\ \ldots\}$.
- *The set of positions of* $\mathtt{C}(\top,\bot)$ *which do not address a symbol* $\bot$ *are* $P_\pi = \{\Lambda,\ 1\}$.
- *Clearly, only the value* $\mathtt{C}(\mathtt{D},\mathtt{B}) \in \gamma(\mathtt{C}(\top,\bot))$ *is computable from* `main`.
- *Therefore, we are interested in the program positions of those terms such that either* $\mathtt{C}(\mathtt{D},\mathtt{B})$ *(i.e.,* $\mathtt{C}(\mathtt{D},\mathtt{B})|_\Lambda$*) or* $\mathtt{D}$ *(i.e.,* $\mathtt{C}(\mathtt{D},\mathtt{B})|_1$*) depend on them.*
- *The only computations from* `main` *to a concretization of* $\mathtt{C}(\top,\bot)$ *(i.e., to a term from* $\gamma(\mathtt{C}(\top,\bot))$*) are thus the following:*

$$\begin{aligned}
\mathcal{D}_1 : \underline{\mathtt{main}}^{\{\}} &\rightarrow_{\Lambda,\mathtt{R1}} \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\underline{\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{A}^{\{(\mathtt{R1},1.1)\}})}, \mathtt{g}^{\{(\mathtt{R1},2)\}}(\mathtt{B}^{\{(\mathtt{R1},2.1)\}})) \\
&\rightarrow_{1,\mathtt{R2}} \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}^{\{(\mathtt{R2},\Lambda)\}}, \underline{\mathtt{g}^{\{(\mathtt{R1},2)\}}(\mathtt{B}^{\{(\mathtt{R1},2.1)\}})}) \\
&\rightarrow_{2,\mathtt{R3}} \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}^{\{(\mathtt{R2},\Lambda)\}}, \mathtt{B}^{\{(\mathtt{R3},\Lambda),(\mathtt{R1},2.1)\}})
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}_2 : \underline{\mathtt{main}}^{\{\}} &\rightarrow_{\Lambda,\mathtt{R1}} \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{A}^{\{(\mathtt{R1},1.1)\}}), \underline{\mathtt{g}^{\{(\mathtt{R1},2)\}}(\mathtt{B}^{\{(\mathtt{R1},2.1)\}})}) \\
&\rightarrow_{2,\mathtt{R3}} \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\underline{\mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{A}^{\{(\mathtt{R1},1.1)\}})}, \mathtt{B}^{\{(\mathtt{R3},\Lambda),(\mathtt{R1},2.1)\}}) \\
&\rightarrow_{1,\mathtt{R2}} \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}^{\{(\mathtt{R2},\Lambda)\}}, \mathtt{B}^{\{(\mathtt{R3},\Lambda),(\mathtt{R1},2.1)\}})
\end{aligned}$$

In this example, it suffices to consider only one of them, e.g., the first one. Now, in order to compute the existing dependences, we show the possible suffixes of $\mathcal{D}_1$ together with their associated subreductions (here, for clarity, we ignore the

program positions):

$$
\begin{aligned}
\textit{Suffix}: \quad & \mathtt{C}(\underline{\mathtt{f}(\mathtt{A})}, \mathtt{g}(\mathtt{B})) \to_{1,\mathtt{R2}} \mathtt{C}(\mathtt{D}, \underline{\mathtt{g}(\mathtt{B})}) \to_{2,\mathtt{R3}} \mathtt{C}(\mathtt{D}, \mathtt{B}) \\
\textit{Subreductions}: \quad & \bullet \\
& \mathtt{C}(\mathtt{f}(\bullet), \underline{\mathtt{g}(\mathtt{B})}) \to_{2,\mathtt{R3}} \mathtt{C}(\mathtt{f}(\bullet), \mathtt{B}) \\
& \mathtt{C}(\bullet, \underline{\mathtt{g}(\mathtt{B})}) \to_{2,\mathtt{R3}} \mathtt{C}(\bullet, \mathtt{B}) \\
& \mathtt{C}(\underline{\mathtt{f}(\mathtt{A})}, \mathtt{g}(\bullet)) \to_{1,\mathtt{R2}} \mathtt{C}(\mathtt{D}, \underline{\mathtt{g}(\bullet)}) \to_{2,\mathtt{R3}} \mathtt{C}(\mathtt{D}, \bullet) \\
& \mathtt{C}(\underline{\mathtt{f}(\mathtt{A})}, \bullet) \to_{1,\mathtt{R2}} \mathtt{C}(\mathtt{D}, \bullet)
\end{aligned}
$$

$$
\begin{aligned}
\textit{Suffix}: \quad & \mathtt{C}(\mathtt{D}, \underline{\mathtt{g}(\mathtt{B})}) \to_{2,\mathtt{R3}} \mathtt{C}(\mathtt{D}, \mathtt{B}) \\
\textit{Subreductions}: \quad & \bullet \\
& \mathtt{C}(\bullet, \underline{\mathtt{g}(\mathtt{B})}) \to_{2,\mathtt{R3}} \mathtt{C}(\bullet, \mathtt{B}) \\
& \mathtt{C}(\mathtt{D}, \underline{\mathtt{g}(\bullet)}) \to_{2,\mathtt{R3}} \mathtt{C}(\mathtt{D}, \bullet) \\
& \mathtt{C}(\mathtt{D}, \bullet)
\end{aligned}
$$

$$
\begin{aligned}
\textit{Suffix}: \quad & \mathtt{C}(\mathtt{D}, \mathtt{B}) \\
\textit{Subreductions}: \quad & \bullet \\
& \mathtt{C}(\bullet, \mathtt{B}) \\
& \mathtt{C}(\mathtt{D}, \bullet)
\end{aligned}
$$

Therefore, we have the following dependences (we only show the program positions of the root symbols, since these are the only relevant program positions for computing the slice):

· *From the first suffix and its subreductions:*

$$
\begin{aligned}
& \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{f}(\mathtt{A}), \mathtt{g}(\mathtt{B})) \rightsquigarrow_{\mathtt{main}} \mathtt{C}(\mathtt{D}, \mathtt{B}) \\
& \mathtt{A}^{\{(\mathtt{R1},1.1)\}} \rightsquigarrow_{\mathtt{main}} \mathtt{D} \\
& \mathtt{f}^{\{(\mathtt{R1},1)\}}(\mathtt{A}) \rightsquigarrow_{\mathtt{main}} \mathtt{D}
\end{aligned}
$$

· *From the second suffix and its subreductions:*

$$
\begin{aligned}
& \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D}, \mathtt{g}(\mathtt{B})) \rightsquigarrow_{\mathtt{main}} \mathtt{C}(\mathtt{D}, \mathtt{B}) \\
& \mathtt{D}^{\{(\mathtt{R2},\Lambda)\}} \rightsquigarrow_{\mathtt{main}} \mathtt{D}
\end{aligned}
$$

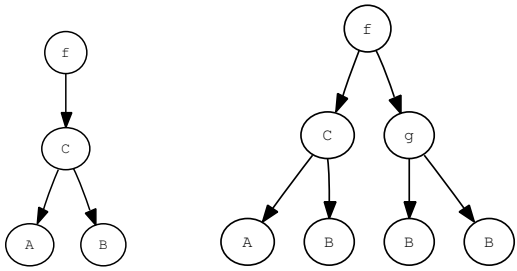

Fig. 4. Tree terms of $\mathtt{f}(\mathtt{C}(\mathtt{A},\mathtt{B}))$ and $\mathtt{f}(\mathtt{C}(\mathtt{A},\mathtt{B}),\mathtt{g}(\mathtt{B},\mathtt{B}))$

· *From the third suffix and its subreductions:*

$$\begin{aligned} \mathtt{C}^{\{(\mathtt{R1},\Lambda)\}}(\mathtt{D},\mathtt{B}) &\leadsto_{\mathtt{main}} \mathtt{C}(\mathtt{D},\mathtt{B}) \\ \mathtt{D}^{\{(\mathtt{R2},\Lambda)\}} &\leadsto_{\mathtt{main}} \mathtt{D} \end{aligned}$$

Therefore, the slice of the program w.r.t. $(\mathtt{main},\mathtt{C}(\top,\bot))$ *returns the following set of program positions* $\{(\mathtt{R1},\Lambda),(\mathtt{R1},\mathtt{1}),(\mathtt{R1},\mathtt{1.1}),(\mathtt{R2},\Lambda)\}$.

Clearly, the computation of all terms that depend on a given constructor term is undecidable. In the next section, we present a decidable approximation based on the construction of a graph that approximates the computations of a program.

4 Term Dependence Graphs

In this section, we sketch a new method for approximating the dependences of a program which is based on the construction of a data structure called *term dependence graph*. We first introduce some auxiliary definitions.

Definition 4.1 (Tree term) *We consider that terms are represented by trees in the usual way. Formally, the tree term T of a term t is a tree with nodes labeled with the symbols of t and directed edges from each symbol to the root symbols of its arguments (if any).*

For instance, the tree terms of the terms $\mathtt{f}(\mathtt{C}(\mathtt{A},\mathtt{B}))$ and $\mathtt{f}(\mathtt{C}(\mathtt{A},\mathtt{B}),\mathtt{g}(\mathtt{B},\mathtt{B}))$ are depicted in Fig. 4.

We introduce two useful functions that manipulate tree terms. First, function *Term* from nodes to terms is used to extract the term associated to the subtree whose root is the given node of a tree term:

$$Term(T,n) \;=\; \begin{cases} n & \text{if } n \text{ has no children in } T \\ n(\overline{Term(T,n_k)}) & \text{if } n \text{ has } k \text{ children } \overline{n_k} \text{ in } T \end{cases}$$

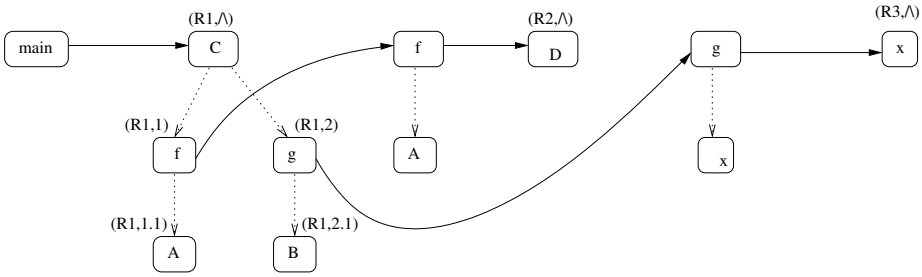

Fig. 5. Term dependence graph of the program in Example 3.5

Now, function $Term_{abs}$ is analogous to function $Term$ but replaces inner operation-rooted subterms by fresh variables:

$$Term_{abs}(T,n) \;=\; \begin{cases} n & \text{if } n \text{ has no children in } T \\ n(\overline{Term'_{abs}(T,n_k)}) & \text{if } n \text{ has } k \text{ children } \overline{n_k} \text{ in } T \end{cases}$$

$$Term'_{abs}(T,n) \;=\; \begin{cases} x & \text{if } n \text{ is a function symbol,} \\ & \text{where } x \text{ is a fresh variable} \\ Term_{abs}(T,n) & \text{otherwise} \end{cases}$$

Now, we can finally introduce the main definition of this section.

Definition 4.2 (Term dependence graph) *Let $\mathcal{P}$ be a program. A term dependence graph for $\mathcal{P}$ is built as follows:*

(i) *the tree terms of all left- and right-hand sides of $\mathcal{P}$ belong to the term dependence graph, where edges in these trees are labeled with S (for Structural);*

(ii) *we add an edge, labeled with C (for Control), from the root symbol of every left-hand side to the root symbol of the corresponding right-hand side;*

(iii) *finally, we add an edge, labeled with C, from every node n of the tree term T_r of the right-hand side of a rule to the topmost node m of the tree term T_l of a left-hand side of a rule whenever $Term_{abs}(T_r,n)$ and $Term(T_l,m)$ unify.*

Intuitively speaking, the term dependence graph stores a path for each possible computation in the program. A similar data structure is introduced in [1], where it is called *graph of functional dependencies* and is used to detect unsatisfiable computations by *narrowing* [11].

Example 4.3 *The term dependence graph of the program of Example 3.5 is shown in Fig. 5.*[5] *Here, we depict C arrows as solid arrows and S arrows as dotted arrows. Observe that only the symbols in the right-hand sides of the rules are labeled with program positions.*

Clearly, the interest in term dependence graphs is that we can compute a *complete* program slice from the term dependence graph of the program. Usually, the slice will

[5] For simplicity, we make no distinction between a node and the label of this node.

not be correct since the graph is an approximation of the program computations and, thus, some paths in the graph would not have a counterpart in the actual computations of the program.

Algorithm 1 *Given a program $\mathcal{P}$ and a slicing criterion (f, π), a slice of $\mathcal{P}$ w.r.t. (f, π) is computed as follows:*

(i) *First, the term dependence graph of $\mathcal{P}$ is computed according to Def. 4.2.*

For instance, we start with the term dependence graph of Fig. 5 for the program of Example 3.5.

(ii) *Then, we identify in the graph the nodes $\mathcal{N}$ that correspond to the program positions P_π of π which do not address the symbol $\bot$. For this purpose, we start from a node in the left-hand side of a rule labeled with symbol f and follow its C-path (i.e., a path made of C arrows). If the last node of this path is a constructor constant c and $\pi = c$ or $\pi = \top$, then this node belongs to $\mathcal{N}$. Otherwise (i.e., it is a constructor-rooted term), we collect all topmost constructor symbols that agree with π and follow the C-paths that start from every maximal operation-rooted subterm in order to continue inspecting the associated values.*

For instance, given the slicing criterion $(\mathtt{main}, \mathtt{C}(\top, \bot))$ and the term dependence graph of Fig. 5, the nodes $\mathcal{N}$ that correspond to the program positions of $\mathtt{C}(\top, \bot)$ which do not address the symbol $\bot$ are shown with a bold box in Fig. 6.

(iii) *Finally, we collect*
- *the program positions of the nodes (associated with the right-hand side of a program rule) in every C-path that ends in a node of $\mathcal{N}$ and either starts in f or there is no node labeled with f in the path,*
- *the program positions of the* descendants *$\mathcal{M}$ of the above nodes (i.e., all nodes which are reachable following the S arrows), and*
- *the program positions of the nodes which are reachable from $\mathcal{M}$ following the C arrows, its descendants, and so on, until no new nodes are collected.*

Therefore, in the example above, the slice will contain the following program positions:
- *the program positions $(\mathtt{R1}, \Lambda), (\mathtt{R1}, \mathtt{1}), (\mathtt{R2}, \Lambda)$ associated with the paths that end in a node with a bold box;*
- *the program positions of their descendants, i.e., $(\mathtt{R1}, \mathtt{1.1})$;*
- *and no more program positions, since there is no node reachable from the node labeled with* `A`.

Trivially, this algorithm always terminates. The completeness of the algorithm (i.e., that all the program positions of the slice according to Definition 3.10 are collected) can be proved by showing that all possible computations can be traced using the term dependence graph.

However, the above algorithm for computing static slices is not correct since there may be C-paths in the term dependence graph that have no counterpart in

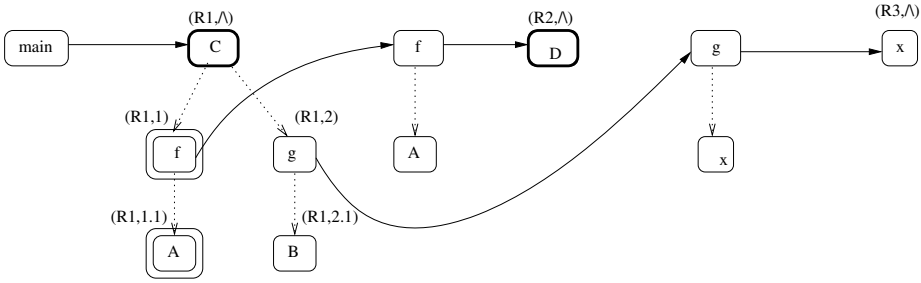

Fig. 6. Slice of the program in Example 3.5

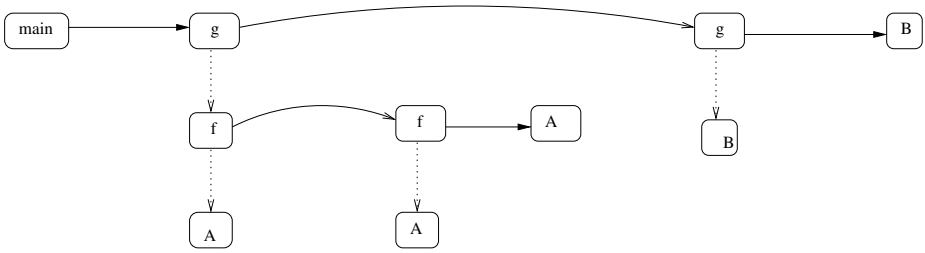

Fig. 7. Term dependence graph of the program in Example 4.4

the computations of the original program. The following example illustrates this point.

Example 4.4 *Consider the following program:*

$$\begin{aligned} &(\texttt{R1}) \quad \texttt{main} \rightarrow \texttt{g(f(A))} \\ &(\texttt{R2}) \quad \texttt{g(B)} \rightarrow \texttt{B} \\ &(\texttt{R3}) \quad \texttt{f(A)} \rightarrow \texttt{A} \end{aligned}$$

The associated term dependence graph is shown in Fig. 7. From this term dependence graph, we would infer that there is a computation from `main` *to* `B` *while this is not true.*

5 Related Work and Discussion

The first attempt to adapt PDGs to the functional paradigm has been recently introduced by Rodrigues and Barbosa [10]. They have defined the *functional dependence graphs* (FDG), which represent control relations in functional programs. However, the original aim of FDGs was the component identification in functional programs and thus they only consider high level functional program entities (i.e., the lowest level of granularity they consider are functions).

In a FDG, a single node often represents a complex term (indeed a complete function definition) and, hence, the information about control dependences of its subterms is not stored in the graph. Our definition of term dependence graph solves this problem by representing terms as trees and thus considering a lower level of

granularity for control dependences between subterms.

As mentioned before, our term dependence graph shares many similarities with the loop checks of [1]. Roughly speaking, [1] defines a directed graph of functional dependencies as follows: for every rule $l \to r$, there is an $\mathcal{R}$-arrow from l to every subterm of r (where inner arguments are replaced by fresh variables); also, u-arrows are added from every term in the right-hand side of an $\mathcal{R}$-arrow to every term in the left-hand side of an $\mathcal{R}$-arrow with which it unifies. In this way, every possible computation path can be followed in the directed graph of functional dependencies. Later, [2] introduced the computation of similar relations (the so called dependency pairs) to analyze the termination of term rewriting systems.

As for future work, we plan to formally prove the completeness of the slices computed by Algorithm 1. We also want to identify and define more dependence relations in the term dependence graph in order to augment its precision w.r.t. Definition 3.6. Then, we want to extend the framework to cover higher-order features. Finally, we plan to implement the slicing algorithm and integrate it in a Curry slicer [9] to perform static slicing of functional and functional logic programs.

References

[1] M. Alpuente, M. Falaschi, M.J. Ramis, and G. Vidal. Narrowing Approximations as an Optimization for Equational Logic Programs. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93, Tallinn (Estonia)*, pages 391–409. Springer LNCS 714, 1993.

[2] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[3] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[4] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[5] J. Field and F. Tip. Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing. *Information and Software Technology*, 40(11-12):609–634, 1998.

[6] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimization. In *Proc. of the 8th Symp. on the Principles of Programming Languages (POPL'81), SIGPLAN Notices*, pages 207–218, 1981.

[7] Y.A. Liu and S.D. Stoller. Eliminating Dead Code on Recursive Data. *Science of Computer Programming*, 47:221–242, 2003.

[8] C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 123–134. ACM Press, 2004.

[9] C. Ochoa, J. Silva, and G. Vidal. Lighweight Program Specialization via Dynamic Slicing. In *Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 1–7. ACM Press, 2005.

[10] N. Rodrigues and L.S. Barbosa. Component Identification Through Program Slicing. In *Proc. of Formal Aspects of Component Software (FACS 2005)*. Elsevier ENTCS, 2005.

[11] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[12] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[13] M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.