A Framework for Computing Finite SLD Trees[☆]

Naoki Nishida[a], Germán Vidal[b,*]

[a] *Graduate School of Information Science, Nagoya University*
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
[b] *MiST, DSIC, Universitat Politècnica de València*
Camino de Vera, s/n, 46022 Valencia, Spain

Abstract

The search space of SLD resolution, usually represented by means of a so-called SLD tree, is often infinite. However, there are many applications that must deal with possibly infinite SLD trees, like partial evaluation or some static analyses. In this context, being able to construct a finite representation of an infinite SLD tree becomes useful.

In this work, we introduce a framework to construct a finite data structure representing the (possibly infinite) SLD derivations for a goal. This data structure, called *closed* SLD tree, is built using four basic operations: unfolding, flattening, splitting, and subsumption. We prove some basic properties for closed SLD trees, namely that both computed answers and calls are preserved. We present a couple of simple strategies for constructing closed SLD trees with different levels of abstraction, together with some examples of its application. Finally, we illustrate the viability of our approach by introducing a test case generator based on exploring closed SLD trees.

Keywords: logic programming, semantics, program analysis

[☆]This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.
This paper is published in "*Naoki Nishida, Germán Vidal: A Framework for Computing Finite SLD Trees. Journal of Logic and Algebraic methods in Programming, 2015.*" DOI: `http://dx.doi.org/10.1016/j.jlamp.2014.11.006`.

[*]Corresponding author.
Email addresses: `nishida@is.nagoya-u.ac.jp` (Naoki Nishida), `gvidal@dsic.upv.es` (Germán Vidal)

1. Introduction

In the context of logic programming, *partial evaluation* (also known as partial deduction [25]) is a well known technique to specialize programs. Intuitively speaking, given a logic program P and a finite set of atoms $\mathcal{A} = \{a_1, \ldots, a_n\}$, one should construct finite—possibly *incomplete*—SLD trees for the atomic goals $\leftarrow a_1, \ldots, \leftarrow a_n$, such that every leaf in these trees is either successful, a failure, or only contains atoms that are instances of $\{a_1, \ldots, a_n\}$. This condition, called *closedness* condition [25], ensures that the computed trees are *self-contained*, which in turn guarantees the correctness of the approach.

In this work, we aim at generalizing this idea by introducing a general framework to construct finite representations of (possibly infinite) SLD trees, so that they can also be used in other contexts. Indeed, there are many problems in computer science that require dealing with (a finite representation of) an infinite search space. Besides partial evaluation, static analyses and model checking techniques, for instance, aim at exploring all possible computations starting from a goal or from a class of goals, some of which are usually infinite. Furthermore, there are many approaches (see, e.g., [1, 2, 3, 9, 8, 15, 16, 27, 29, 38]) that advocate a *transformational approach* in which a program with imperative, object-oriented or concurrent features—which are often difficult to analyze—are compiled into a simpler, rule-based intermediate language (usually ignoring some details or abstracting away some features), where rigorous program analyses can be defined and implemented in a simpler way. One of the most popular such rule-based languages is Prolog. Therefore, our framework may contribute to the use of Prolog as a target language within the above transformational approach to program analysis.

In particular, we introduce an extension of standard SLD trees that are built using four basic operations: unfolding (based on SLD resolution), flattening (i.e., generalizing some atoms in a goal), splitting (i.e., partitioning a goal into a number of subgoals that are then evaluated independently) and subsumption (a sort of memoization). When all the leaves of the tree are either successful, a failure or a subsumed goal, we speak of a *closed* SLD tree. Besides formalizing this notion, our main contributions are the following:

- Given a program and a goal, we show that a closed SLD tree can always be constructed using an appropriate strategy. Here, we present a couple of strategies that produce closed SLD trees with different levels of abstraction.

- We prove that the computed answers of a standard (possibly infinite)

SLD tree and those of an associated closed SLD tree (no matter the considered strategy) are the same. Thus, the abstraction involved in producing closed SLD trees does not affect the computed answers represented by the tree.

- We also prove that, for every call in a standard SLD tree, there is a (possibly more general) call in any associated closed SLD tree. This property guarantees the usefulness of closed SLD trees for those program analysis where computing (an approximation of) the call patterns of a goal is required.

- Then, we show how the success set of a closed SLD tree can be represented in a compact way by means of equations. This might be useful, e.g., for program comprehension.

- Finally, we illustrate the viability of our approach by introducing a fully automatic test case generator based on exploring closed SLD trees.

This paper is organized as follows. After some preliminaries, Section 3 introduces the basic operations involved in the construction of closed SLD trees and states a number of properties for them. Specific strategies for constructing closed SLD trees are introduced and illustrated by means of examples in Section 4. In Section 5, the design of a test case generator is introduced. Finally, Section 6 discusses some related work and Section 7 concludes and points out some directions for further research.

2. Preliminaries

In this section, we briefly present some basic notions from logic programming; we refer the interested reader to [24] for a detailed introduction to this paradigm.

We consider a first-order language with a fixed vocabulary of predicate symbols, function symbols, and variables denoted by Π, $\mathcal{F}$, and $\mathcal{V}$, respectively. We let $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denote the set of *terms* constructed using symbols from $\mathcal{F}$ and variables from $\mathcal{V}$. An *atom* has the form $p(t_1, \ldots, t_n)$ with $p/n \in \Pi$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ for $i = 1, \ldots, n$. A *definite clause* has the form $head \leftarrow body$, where *head* is an atom and $body \equiv (a_1 \wedge \cdots \wedge a_n)$ is a goal[1] (a conjunction of atoms); the empty goal is denoted by *true*. We usually denote goals with capital letters (e.g., $G, C, \ldots$) and atoms with small letters (e.g.,

[1] We use "$\equiv$" to denote the identity on syntactic objects, and identify goals with (possibly empty) conjunctions of atoms.

$a, b, \ldots$). A *definite program* is a finite set of definite clauses. In the following, we focus on definite programs and will refer to them just as programs (analogously to clauses and goals). $\mathcal{V}ar(s)$ denotes the set of variables in the syntactic object s.

Substitutions and their operations are defined as usual. In particular, a substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ denotes a (partial) mapping σ such that $\sigma(x) = t_i$ if $x = x_i$, $i = 1, \ldots, n$, and $\sigma(x) = x$ otherwise. The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is called the *domain* of a substitution σ. The empty substitution is denoted by id. A substitution σ is idempotent if $\sigma = \sigma \cdot \sigma$, where "$\cdot$" denotes the standard composition operator on substitutions. The *restriction* $\theta\restriction_V$ of a substitution θ to a set of variables V is defined as follows: $x\theta\restriction_V = x\theta$ if $x \in V$, and $x\theta\restriction_V = x$ otherwise. We say that $\theta = \sigma\ [V]$ if $\theta\restriction_V = \sigma\restriction_V$. This notation is extended to sets of substitutions in the natural way: $\Theta_1 = \Theta_2\ [V]$ implies that there is a substitution $\theta_1 \in \Theta_1$ iff there is a substitution $\theta_2 \in \Theta_2$ such that $\theta_1 = \theta_2\ [V]$. A syntactic object s_1 is *more general* than a syntactic object s_2, denoted $s_1 \leq s_2$, if there exists a substitution θ such that $s_2 = s_1\theta$. A substitution θ is a *unifier* of two syntactic objects t_1 and t_2 iff $t_1\theta = t_2\theta$; furthermore, θ is the *most general unifier* of t_1 and t_2, denoted by $\mathsf{mgu}(t_1 = t_2)$ if, for every other unifier σ of t_1 and t_2, we have that $\theta \leq \sigma$. The mgu operator is naturally extended to a conjunction of equations. A *variable renaming* is a substitution that is a bijection on $\mathcal{V}$. Two syntactic objects t_1 and t_2 are *variants* (or equal up to variable renaming), denoted $t_1 \approx t_2$, if $t_1 = t_2\rho$ for some variable renaming ρ.

Computations in logic programming are formalized by means of SLD resolution. The notion of *computation rule* $\mathcal{R}$ is used to select an atom within a goal for its evaluation. Given a program P, a goal $G \equiv (a_1 \wedge \cdots \wedge a_n)$, and a computation rule $\mathcal{R}$, we say that $G \leadsto_{P,\mathcal{R},\sigma} G'$ is an *SLD resolution step* for G with P and $\mathcal{R}$ if $\mathcal{R}(G) = a_i$, $1 \leq i \leq n$, is the selected atom, $h \leftarrow b_1 \wedge \cdots \wedge b_m$ is a renamed apart clause (i.e., a clause with fresh variables not used before in the computation) of P, $\sigma = \mathsf{mgu}(a_i = h)$, and $G' \equiv (a_1 \wedge \cdots \wedge a_{i-1} \wedge b_1 \wedge \cdots \wedge b_m \wedge a_{i+1} \wedge \cdots \wedge a_n)\sigma$; we often omit P, $\mathcal{R}$, and/or σ in the notation of an SLD resolution step when they are clear from the context.

Let P be a program, G_0 a goal and $\mathcal{R}$ a computation rule. An *SLD derivation* for G_0 with P and $\mathcal{R}$ is a (finite or infinite) sequence $G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of renamed apart clauses of P, a sequence $\mathcal{R}(G_0)$, $\mathcal{R}(G_1)$, $\mathcal{R}(G_2), \ldots$ of selected atoms, and a sequence $\theta_1, \theta_2, \ldots$ of most general unifiers such that $G_{i-1} \leadsto_{\theta_i} G_i$ using C_i for all $i > 0$. We often use $G_0 \leadsto_\theta^* G_n$ as a shorthand for $G_0 \leadsto_{\theta_1} G_1 \leadsto_{\theta_2} \cdots \leadsto_{\theta_n} G_n$ with $\theta = \theta_1 \cdots \theta_n$ (where $\theta = id$ if $n = 0$). An SLD derivation $G \leadsto_\theta^* G'$ is

successful when $G' \equiv \mathit{true}$; in this case, we say that $\theta\restriction_{\mathcal{V}ar(G)}$ is the *computed answer substitution*.

As it is common practice, SLD derivations are represented by a (possibly infinite) finitely branching tree called *SLD tree*.

3. Closed SLD Trees

In this section, we present a framework for constructing a data structure that extends standard SLD trees by using four basic operations: unfolding, flattening, splitting, and subsumption. When this tree is *closed*—which implies that it is finite—we prove that the tree still represents all the computed answers of the original (possibly infinite) SLD tree. Therefore, in this work we aim at preserving the answers computed in *successful* derivations.

Definition 1 (success set). Let P be a program, G a goal and $\mathcal{R}$ a computation rule. The success set $\mathcal{SS}^{\mathcal{R}}_P(G)$ of G with P and $\mathcal{R}$ is defined as follows:

$$\mathcal{SS}^{\mathcal{R}}_P(G) = \{\theta\restriction_{\mathcal{V}ar(G)} |\ \ G \rightsquigarrow^*_{P,\mathcal{R},\theta} \mathit{true}\}$$

We often ignore the computation rule and write $\mathcal{SS}_P(G)$ instead since the computed answers are the same (up to variable renaming) for any computation rule [24].

3.1. Basic Operations

Let us now introduce the basic operations of our framework for constructing closed SLD trees. The first operation, *unfolding*, consists in applying an SLD resolution step to a goal labelling a leaf of the tree built so far.

Definition 2 (unfolding). Let P be a program and G be a goal. Let $G \rightsquigarrow_{P,\mathcal{R},\sigma} G'$ be an SLD resolution step for some computation rule $\mathcal{R}$. Then, we say that G' in an unfolding of G using $\mathcal{R}$.

We note that an unfolding operator may take the "history" into account (e.g., to avoid unfolding growing calls to a given predicate, according to some order). We refer the reader to, e.g., [12, 21], for unfolding operators that avoid infinite unfolding in the literature of partial evaluation. Also, in Section 5, we consider an unfolding rule that selects the leftmost atom in a goal which does not embed a previously selected atom in the same derivation.

The second operation, *flattening*, basically amounts to generalize a goal by replacing some subterms by fresh variables, and then adding the corresponding equalities to the derived goal. We let $\widehat{\theta}$ denote the *equational representation* of a substitution θ, i.e., if $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ then

$\widehat{\theta} = (x_1 = t_1 \wedge \cdots \wedge x_n = t_n)$. Moreover, we assume that every program implicitly contains a clause for syntactic equality of the form $x = x \leftarrow \mathit{true}$.

Definition 3 (flattening). Let $G \equiv (G_1 \wedge G_2\theta \wedge G_3)$ be a goal, where G_i, $1 \leq i \leq 3$, are (possibly empty) conjunctions of atoms and $\theta \neq id$ is a substitution which is not a renaming and such that $\mathcal{D}om(\theta) \subseteq \mathcal{V}ar(G_2)$ and $\mathcal{D}om(\theta) \cap (\mathcal{V}ar(G_1) \cup \mathcal{V}ar(G_3)) = \emptyset$. Then, we say that the goal $(G_1 \wedge \widehat{\theta} \wedge G_2 \wedge G_3)$ is a flattening of G.

Example 4. Consider the goal $G \equiv \mathtt{p}(\mathtt{f}(\mathtt{X}), \mathtt{Y}) \wedge \mathtt{q}(\mathtt{a}, \mathtt{f}(\mathtt{b}))$. Then, we have that the goal

$$\mathtt{p}(\mathtt{f}(\mathtt{X}), \mathtt{Y}) \wedge (\mathtt{Z} = \mathtt{a}) \wedge (\mathtt{W} = \mathtt{b}) \wedge \mathtt{q}(\mathtt{Z}, \mathtt{f}(\mathtt{W}))$$

is a flattening of G, while

$$\mathtt{p}(\mathtt{f}(\mathtt{X}), \mathtt{Y}) \wedge (\mathtt{X} = \mathtt{a}) \wedge (\mathtt{W} = \mathtt{b}) \wedge \mathtt{q}(\mathtt{X}, \mathtt{f}(\mathtt{W}))$$

is not since $\mathtt{X} \in \mathcal{V}\mathtt{ar}(\mathtt{p}(\mathtt{f}(\mathtt{X}), \mathtt{Y}))$. Also, given the goal $G \equiv \mathtt{p}(\mathtt{X}, \mathtt{X})$, we have that the goal $(\mathtt{A} = \mathtt{X}) \wedge (\mathtt{B} = \mathtt{X}) \wedge \mathtt{p}(\mathtt{A}, \mathtt{B})$ is a flattening of G.

The flattening operation is sound and complete, i.e., no precision is lost by applying flattening steps in an SLD tree:

Theorem 5 (correctness of flattening). *Let P be a program and G a goal. If G' is a flattening of G, then $\mathcal{SS}_P(G) = \mathcal{SS}_P(G')$ $[\mathcal{V}ar(G)]$ (up to variable renaming).*

Proof. Let us assume that $G \equiv (G_1 \wedge G_2\theta \wedge G_3)$, where G_i, $1 \leq i \leq 3$, are conjunctions of atoms and $\theta \neq id$ is a substitution which is not a renaming and such that $\mathcal{D}om(\theta) \subseteq \mathcal{V}ar(G_2)$ and $\mathcal{D}om(\theta) \cap (\mathcal{V}ar(G_1) \cup \mathcal{V}ar(G_3)) = \emptyset$, with $G' \equiv (G_1 \wedge \widehat{\theta} \wedge G_2 \wedge G_3)$. By considering a left-to-right computation rule, we have that any successful derivation of G with P is prefixed by

$$(G_1 \wedge G_2\theta \wedge G_3) \leadsto^*_\sigma (G_2\theta \wedge G_3)\sigma$$

iff the corresponding derivation for G' with P is prefixed by

$$(G_1 \wedge \widehat{\theta} \wedge G_2 \wedge G_3) \leadsto^*_\sigma (\widehat{\theta} \wedge G_2 \wedge G_3)\sigma$$

Furthermore, since $\mathcal{D}om(\theta) \cap \mathcal{V}ar(G_1) = \emptyset$ and clauses are renamed apart in a derivation, we have that $\mathcal{D}om(\theta) \cap \mathcal{D}om(\sigma) = \emptyset$ and, thus, $\mathsf{mgu}(\widehat{\theta}\sigma) = \{x \mapsto x\theta\sigma \mid x \in \mathcal{D}om(\theta)\}$. Assume $\delta = \{x \mapsto x\theta\sigma \mid x \in \mathcal{D}om(\theta)\}$ with $\mathcal{D}om(\delta) \cap \mathcal{V}ar(G_3) = \emptyset$ since $\mathcal{D}om(\theta) \cap \mathcal{V}ar(G_3) = \emptyset$. Therefore, we have

$(\widehat{\theta} \wedge G_2 \wedge G_3)\sigma \leadsto^*_\delta (G_2\theta \wedge G_3)\sigma$. Hence, under a left-to-right computation rule $\mathcal{R}_L$, we have $\mathcal{SS}_P^{\mathcal{R}_L}(G) = \mathcal{SS}_P^{\mathcal{R}_L}(G')$ $[\mathcal{V}ar(G)]$ since $\mathcal{D}om(\delta) \cap \mathcal{V}ar(G) = \emptyset$. Finally, by the independence of the computation rule [24], we have that $\mathcal{SS}_P^{\mathcal{R}}(G) = \mathcal{SS}_P^{\mathcal{R}'}(G')$ $[\mathcal{V}ar(G)]$ also holds for any arbitrary computation rules $\mathcal{R}$ and $\mathcal{R}'$. □

Our next operation is called *splitting*. It has been already used in different contexts to break up a conjunction of atoms (see, e.g., the use of splitting in the partial evaluation of logic programs [14]). Let us first recall some terminology from [14]. Given a set S, we let $\mathcal{M}(S)$ denote all multisets composed of elements of S, and let $=_r$ denote syntactic identity up to reordering on conjunctions. Given a multiset M, we use the notation $\wedge_{C \in M} C$ to denote a conjunction constructed from the elements of M, taking their multiplicity into account. For example, for the multiset $M = \{\{B, B, D\}\}$, $\wedge_{C \in M} C$ refers to either of the conjunctions $B \wedge B \wedge D$, $B \wedge D \wedge B$, or $D \wedge B \wedge B$.

Definition 6 (partitioning function [14]). Let $\mathcal{C}$ denote the set of all conjunctions of atoms over the given alphabet. A *partitioning function* is a mapping $p : \mathcal{C} \mapsto \mathcal{M}(\mathcal{C})$ such that for any $B \in \mathcal{C}$, $B =_r \wedge_{C \in p(B)} C$.

The splitting operation then uses a partitioning function to split up a goal into a number of subgoals:

Definition 7 (splitting). Let G be a goal and let p be a partition function. Then, the splitting of G using p gives rise to a collection of subgoals $G_1, \ldots, G_n$, $n \geq 1$, where $p(G) = \{\{G_1, \ldots, G_n\}\}$.

Analogously to the unfolding operator, a splitting operator or partition function may also take into account the history of a computation in order to produce the best partition. Actually, splitting is the only real source of abstraction in our framework. Whenever we split a goal, the subgoals will be evaluated independently and, thus, some variable bindings will not be shared anymore between these goals. Therefore, splitting should be applied carefully in order to minimize the loss of precision. Observe, nevertheless, that the global tree data structure still contains enough information to extract all (and only) the computed answers of the original program (see Theorem 11 below). The loss of precision is thus only *local* to single derivations, where some atoms may be more general and perform some unnecessary steps (which are then discarded when composing them with the answers coming from other subgoals using the parallel composition operator, see below).

Example 8. Consider the goal $\mathtt{p}(\mathtt{X}) \wedge \mathtt{q}(\mathtt{Y},\mathtt{Z}) \wedge \mathtt{r}(\mathtt{X})$. A splitting operator may return the subgoals $\mathtt{p}(\mathtt{X}) \wedge \mathtt{q}(\mathtt{Y},\mathtt{Z})$ and $\mathtt{r}(\mathtt{X})$. By considering a left-to-right computation rule as in Prolog, this splitting may involve performing some unnecessary steps for $\mathtt{r}(\mathtt{X})$ since the bindings from the first subgoal will not be applied to $\mathtt{r}(\mathtt{X})$.

A better splitting operator would instead return the subgoals: $\mathtt{p}(\mathtt{X}) \wedge \mathtt{r}(\mathtt{X})$ and $\mathtt{q}(\mathtt{Y},\mathtt{Z})$, so that we avoid the independent evaluation of shared variables.

The correctness of splitting is a consequence of the fact that SLD resolution is compositional [13, 28]. Let us now recall the definition of *parallel composition* of substitutions, denoted by $\Uparrow$ in [28].

Definition 9 (parallel composition [28]). Let θ_1 and θ_2 be two idempotent substitutions. Then, we define $\Uparrow$ as follows:

$$\theta_1 \Uparrow \theta_2 = \begin{cases} \mathsf{mgu}(\widehat{\theta_1} \wedge \widehat{\theta_2}) & \text{if } \widehat{\theta_1} \wedge \widehat{\theta_2} \text{ has a solution (a unifier)} \\ \mathit{fail} & \text{otherwise} \end{cases}$$

Parallel composition is then extended to sets of substitutions in the natural way: $\Theta_1 \Uparrow \Theta_2 = \{\theta_1 \Uparrow \theta_2 \mid \theta_1 \in \Theta_1,\ \theta_2 \in \Theta_2,\ \theta_1 \Uparrow \theta_2 \neq \mathit{fail}\}$.

In order to state and prove the correctness of splitting, we first need a compositionality result for logic programs. The compositionality of SLD resolution in logic programming can then be stated as follows:

Theorem 10 (compositionality [13, 28]). *Let P be a program and $G_1 \wedge G_2$ be a goal, where G_1, G_2 are (possibly empty) conjunctions of atoms. Then, we have $\mathcal{SS}_P(G_1 \wedge G_2) = \mathcal{SS}_P(G_1) \Uparrow \mathcal{SS}_P(G_2)$.*

The correctness of the splitting operation is thus a straightforward consequence of the compositionality of SLD resolution:

Theorem 11 (correctness of splitting). *Let P be a program and G be a goal. Let p be a partitioning function with $p(G) = \{\{G_1, \ldots, G_n\}\}$, $n > 0$. Then, $\mathcal{SS}_P(G) = \mathcal{SS}_P(G_1) \Uparrow \cdots \Uparrow \mathcal{SS}_P(G_n)$, up to variable renaming.*

Proof. Immediate by Theorem 10.

Our last operation is called *subsumption*. Subsumption is used to check if a leaf of the tree built so far is a variant of a previous goal in this tree so that we can avoid its evaluation. Therefore, subsumption allows us to introduce a sort of memoization.

Definition 12. Let G be a goal and let $\mathcal{S}$ be a set of goals. We say that G is subsumed by $\mathcal{S}$ if there exists a goal $G' \in \mathcal{S}$ such that $G \approx G'$ (i.e., they are equal up to variable renaming).

Let us note that, in many related works (particularly, in the area of partial evaluation), a goal G is said to be subsumed by another goal G' if G is an *instance* of G'. However, in our framework, a similar effect can be achieved by first applying flattening and then splitting and subsumption. We prefer to keep these operations independent so that the resulting framework is more flexible and customizable.

The correctness of subsumption is an easy consequence of the fact that the success set of a goal and that of a renaming of this goal is the same up to variable renaming. In the following, we say that two substitutions θ_1 and θ_2 are *variants*, in symbols $\theta_1 \approx \theta_2$, iff $\theta_1 = \mathsf{mgu}(\widehat{\theta_2}\rho)$ for some variable renaming ρ. This notion is naturally extended to sets as follows: $\Theta_1 \approx \Theta_2$ implies that there is a substitution $\theta_1 \in \Theta_1$ iff there is a substitution $\theta_2 \in \Theta_2$ such that $\theta_1 \approx \theta_2$.

Theorem 13 (correctness of subsumption). *Let P be a program, G a goal and G' a variant of G (i.e., $G \approx G'$). Then, $\mathcal{SS}_P(G) \approx \mathcal{SS}_P(G')$.*

Proof. The proof is a straightforward consequence of Corollary 3.19 in [5] since, for each SLD derivation from G, we can construct a *similar* SLD derivation for G', and vice versa. Roughly speaking, two SLD derivations are similar if the initial goals are variants of each other, the length of the derivations is the same, and for each SLD resolution step, the same atoms are selected and variants of the same clauses are considered. □

3.2. Extended SLD Trees

Now, we formalize a new data structure that extends the original notion of SLD tree to also allow the application of the operations introduced so far. In the following, we denote these data structures with the symbol τ. Also, we denote by Goal the domain of goals (including *true* and *fail*) and by Subs the domain of substitutions, respectively. Moreover, we let $\underline{\mathsf{Goal}}$ denote the domain of *marked* goals, that we graphically depict with an underscore, i.e., $\underline{\mathsf{Goal}} = \{\underline{G} \mid G \in \mathsf{Goal}\}$.

Definition 14 (extended SLD tree). Let P be a program, G_0 a goal, and $\mathcal{R}$ a computation rule. An extended SLD tree τ for G_0 with P and $\mathcal{R}$ is an acyclic, directed rooted node- and edge-labeled graph. It is denoted

by a pair $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N} = \mathsf{Goal} \cup \underline{\mathsf{Goal}}$ is the set of nodes and $\mathcal{E} \subseteq \mathcal{N} \times \{\mathsf{unf}, \mathsf{flat}, \mathsf{sub}, \mathsf{split}\} \times \mathcal{N}$ is a set of labelled directed edges.[2]

We note that our extended SLD trees share some similarities with the AND-OR trees introduced in [10], since we also combine OR steps (unfolding with respect to all matching clauses) and AND steps (splitting).

In the following, we use $G \in \tau$ to denote that there exists a node labelled with a goal G in the tree τ. Marked nodes are graphically denoted by underlining them. We use the notation $\tau[G]$ to denote that the tree τ contains a *leaf* labelled with G which is not marked (we use $\tau[\underline{G}]$ when it is marked). We also use this notation for the edges that lead to leaves, e.g., $\tau[\underline{G} \rightarrow^{l} G']$ denotes that τ contains an edge from (marked) node G to (non-marked) leaf G' labelled with l. As we will see below in the transition rules, we use this notation either as a condition on a tree τ or as a modification of τ.

Let P be a program, G_0 a goal, and $\mathcal{R}$ a computation rule. An extended SLD tree for G_0 with P and $\mathcal{R}$ is built as follows.

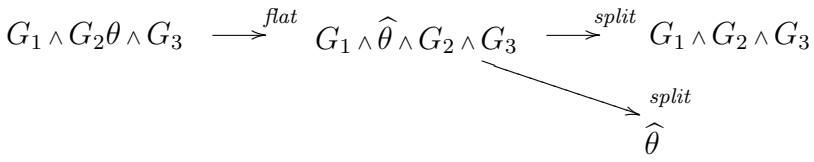
- The initial tree contains no edges and only one non-marked node labelled with G_0.
- The initial tree is expanded using the transition rules shown in Fig. 1, where the program and the computation rule are considered global parameters and they are not explicitly shown.
- Rules success and failure just terminates a branch when the special atoms *true* (success) or *fail* (failure) are derived.
- The subsumption rule checks whether there is a previous node *in the same root-to-leaf path* such that G and G' are variants. We require the path $G' \rightarrow^{+} G$ (here, $\rightarrow^{+}$ denotes a path of one or more steps) to include at least one unfolding step in order to avoid introducing a loop with no real evaluation. We note that, in some approaches, subsumption is rather denoted with an edge from the subsumed node to the subsuming node. We prefer our definition in order to keep the data structure a tree. Note also that, in this rule, the added leaf *true* plays no role, only the label of the edge is relevant.
- The flattening rule proceeds as expected, by deriving a flattening of a given goal. Here, we further require that the equations introduced by a

[2]Note, however, that when the label is unf or sub we label the step with some additional information (see below). Here, we keep this simpler formulation for simplicity.

flattening step cannot be unfolded until a goal only contains equations. This condition is necessary to avoid introducing incorrect loops of the form

$$G_1 \wedge G_2\theta \wedge G_3 \xrightarrow{flat} G_1 \wedge \widehat{\theta} \wedge G_2 \wedge G_3 \xrightarrow{sub} \cdots \xrightarrow{unf} \cdots \xrightarrow{unf} G_1 \wedge G_2\widehat{\theta} \wedge G_3$$

In contrast, any sensible strategy would apply splitting immediately after flattening:



- The splitting rule proceeds as expected, and it is parametric w.r.t. a given partitioning function.
- Finally, in the unfolding rule, the SLD resolution steps must be all and only the SLD resolution steps for G with P and $\mathcal{R}$ (i.e., analogously to standard SLD trees, a goal cannot be *partially* unfolded).

Example 15. Let us consider the following program $P_{\texttt{len}}$:

$$\texttt{len}([\,],\texttt{0}). \qquad \texttt{inc}(\texttt{0},\texttt{s}(\texttt{0})).$$
$$\texttt{len}([\texttt{X}|\texttt{R}],\texttt{Z}) \leftarrow \texttt{len}(\texttt{R},\texttt{Y}) \wedge \texttt{inc}(\texttt{Y},\texttt{Z}). \qquad \texttt{inc}(\texttt{s}(\texttt{N}),\texttt{s}(\texttt{M})) \leftarrow \texttt{inc}(\texttt{N},\texttt{M}).$$

and the initial goal $G \equiv \texttt{len}(\texttt{A},\texttt{B})$. While the standard SLD tree for goal G with program $P_{\texttt{len}}$ is trivially infinite, we can easily construct a finite extended SLD tree for G with $P_{\texttt{len}}$ (shown in Fig. 2).

Observe that the rules of Fig. 1 overlap, i.e., given an extended SLD tree with some unmarked leaves, several rules can be applicable at the same time. Actually, these transition rules define *all* possible extended SLD trees. In practice, as we will see below, one should introduce a particular strategy that builds just one such extended SLD tree. Also, note that any standard SLD tree is also an extended SLD tree by only considering the rules success, failure and unfolding from Fig. 1.

Given a standard SLD tree τ for a goal G, the computed answers of G are obtained by composing the substitutions labelling the paths from G to

$$\text{(success)} \qquad \frac{}{\tau[true] \longrightarrow \tau[\underline{true}]}$$

$$\text{(failure)} \qquad \frac{G \not\equiv true \text{ and } \nexists G' \text{ such that } G \rightsquigarrow_{P,\mathcal{R},\sigma} G'}{\tau[G] \longrightarrow \tau[\underline{fail}]}$$

$$\text{(subsumption)} \qquad \frac{\exists G' \in \tau : \ G' \rightarrow^{+} G \text{ in } \tau \text{ includes at least one unfolding step and } G\rho = G' \text{ with } \rho \text{ a renaming substitution}}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{sub}_{(G',\rho)} \underline{true}]}$$

$$\text{(flattening)} \qquad \frac{G' \text{ is a flattening of } G}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{flat} G']}$$

$$\text{(splitting)} \qquad \frac{p(G) = \{\{G_1, \ldots, G_n\}\} \text{ for some partitioning function } p,\ n > 1}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{split} G_1, \ldots, \underline{G} \rightarrow^{split} G_n]}$$

$$\text{(unfolding)} \qquad \frac{G \rightsquigarrow_{P,\mathcal{R},\sigma_1} G_1, \ldots, G \rightsquigarrow_{P,\mathcal{R},\sigma_n} G_n}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\sigma_1)} G_1, \ldots, \underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\sigma_n)} G_n]}$$

Figure 1: Construction of Extended SLD trees

true in τ. Now, we introduce an analogous function that extracts the set of computed answers that are represented by an extended SLD tree. In the following, besides the standard composition of substitutions, we also consider its extension to sets of substitutions as follows. Given a set of substitutions Θ and a substitution σ, we let $\sigma \cdot \Theta = \{\sigma \cdot \theta \mid \theta \in \Theta\}$. Moreover, for simplicity, we do not distinguish between a substitution σ and the singleton set $\{\sigma\}$ when no confusion can arise. Also, we set $\sigma \cdot \{\,\} = \{\,\}$ and $\{\,\} \Uparrow \Theta = \Theta \cdot \{\,\} = \{\,\}$.

Definition 16 (success set of an extended SLD tree). Let P be a program and G_0 be a goal. Let τ be an extended SLD tree for G_0 with P. Then, the success set of G_0 using τ, in symbols $\mathcal{SS}_\tau(G_0)$, is defined as follows:

$$\mathcal{SS}_\tau(G_0) = \{\sigma \restriction_{\mathcal{V}ar(G_0)} \mid \sigma \in ss_\tau(G_0)\}$$

where the definition of the auxiliary function ss is shown in Fig. 3.[3]

In practice, the success set of an extended SLD tree can be obtained, e.g., by following a depth-first exploration of the search space, similarly to an

[3]Here, we do not distinguish whether a node is marked or not since it is not relevant.

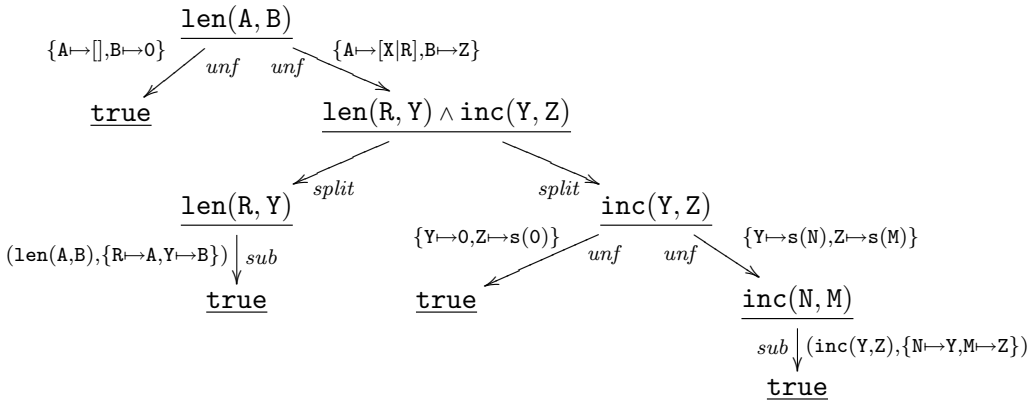


Figure 2: Extended SLD tree for $\mathtt{len(A,B)}$ with $P_{\mathtt{len}}$.

$$ss_\tau(G) = \begin{cases} \{id\} & \text{if } G \equiv true \\ \{\,\} & \text{if } G \equiv fail \\ \rho \cdot ss_\tau(G') & \text{if } G \rightarrow^{sub}_{(G',\rho)} true \in \tau \\ ss_\tau(G') & \text{if } G \rightarrow^{flat} G' \in \tau \\ ss_\tau(G_1) \Uparrow \cdots \Uparrow ss_\tau(G_n) & \text{if } G \rightarrow^{split} G_1, \ldots, G \rightarrow^{split} G_n \in \tau \\ \sigma_1 \cdot ss_\tau(G_1) \cup \cdots \cup \sigma_n \cdot ss_\tau(G_n) & \text{if } G \rightarrow^{unf}_{\sigma_1} G_1, \ldots, G \rightarrow^{unf}_{\sigma_n} G_n \in \tau \end{cases}$$

Figure 3: Auxiliary function ss

standard SLD tree. The main differences are that subsumption steps imply moving back to a previous node of the tree (rather than to its first child) and that splitting involves computing all possible combinations of the different subtrees. Clearly, a fixpoint computation is also possible; we refer the reader to the fixpoint computation for success set equations shown in Section 3.3, since the procedure here would be analogous.

Example 17. Let us consider the extended SLD tree τ for the initial goal $G \equiv \mathtt{len(A,B)}$ shown in Figure 2, where $G' \equiv \mathtt{inc(Y,Z)}$. The associated success set is computed using the function ss_τ following a depth-first strategy

as follows:

$$
\begin{aligned}
ss_\tau(G) &\supseteq \{\mathtt{A} \mapsto [], \mathtt{B} \mapsto \mathtt{0}\} \cdot \{\mathtt{id}\} \ni \{\mathtt{A} \mapsto [], \mathtt{B} \mapsto \mathtt{0}\} \\
ss_\tau(G) &\supseteq \{\mathtt{A} \mapsto [\mathtt{X}|\mathtt{R}], \mathtt{B} \mapsto \mathtt{Z}\} \cdot (\{\mathtt{R} \mapsto \mathtt{A}, \mathtt{Y} \mapsto \mathtt{B}\} \cdot \mathtt{ss}_\tau(\mathrm{G}) \Uparrow \mathtt{ss}_\tau(\mathrm{G}')) \\
&\ni \{\mathtt{A} \mapsto [\mathtt{X}|\mathtt{R}], \mathtt{B} \mapsto \mathtt{Z}\} \cdot (\{\mathtt{R} \mapsto \mathtt{A}, \mathtt{Y} \mapsto \mathtt{B}\} \cdot \{\mathtt{A} \mapsto [], \mathtt{B} \mapsto \mathtt{0}\} \Uparrow \{\mathtt{Y} \mapsto \mathtt{0}, \mathtt{Z} \mapsto \mathtt{s(0)}\}) \\
&= \{\mathtt{A} \mapsto [\mathtt{X}|\mathtt{R}], \mathtt{B} \mapsto \mathtt{Z}\} \cdot (\{\mathtt{R} \mapsto [], \mathtt{Y} \mapsto \mathtt{0}\} \Uparrow \{\mathtt{Y} \mapsto \mathtt{0}, \mathtt{Z} \mapsto \mathtt{s(0)}\}) \\
&= \{\mathtt{A} \mapsto [\mathtt{X}|\mathtt{R}], \mathtt{B} \mapsto \mathtt{Z}\} \cdot \{\mathtt{R} \mapsto [], \mathtt{Y} \mapsto \mathtt{0}, \mathtt{Z} \mapsto \mathtt{s(0)}\} \\
&= \{\mathtt{A} \mapsto [\mathtt{X}], \mathtt{B} \mapsto \mathtt{s(0)}\} \\
&\ldots
\end{aligned}
$$

In general, however, we are not interested in arbitrary extended SLD trees but only in *closed* SLD trees:

Definition 18 (closed SLD tree). Let τ be an extended SLD tree. We say that τ is closed iff all nodes are marked.

Trivially, closed SLD trees are finite. Moreover, their associated success set is the same as the original (possibly infinite) SLD tree for the same goal and program:

Theorem 19. *Let P be a program and G be a goal. Let τ be a closed SLD tree for G with P. Then, $\mathcal{SS}_P(G) = \mathcal{SS}_\tau(G)$ up to variable renaming.*

Proof. Essentially, the proof of this claim is a consequence of Theorems 5, 11 and 13. Let us now consider $\mathcal{SS}_P(G) \subseteq \mathcal{SS}_\tau(G)$ and $\mathcal{SS}_P(G) \supseteq \mathcal{SS}_\tau(G)$ independently.

($\subseteq$) We prove that, for all computed answer $\theta \in \mathcal{SS}_P(G)$, there exists $\theta \in \mathcal{SS}_\tau(G)$; for simplicity, we consider that clauses are standardized apart (i.e., renamed with fresh variable) using the same renamings.

Let us consider that $G_0 \leadsto_\theta^* \mathit{true}$ with P (where θ is assumed restricted to the variables of G_0 for simplicity). Now, we introduce the following auxiliary functions:

- we let $|G|$ denote the *weight* of goal G, i.e., the number of constant and function symbols in the atoms of G (not including the equalities introduced by flattening steps);
- we let $sh(G)$ be the number of occurrences of repeated variables in G (again, not including the equalities introduced by flattening steps).

Then, given a derivation $\mathcal{D} \equiv (G_0 \leadsto_{\theta_1} \cdots \leadsto_{\theta_n} \mathit{true})$, we let $\mathcal{W}(\mathcal{D}) = n' + |G_0| + sh(G_0)$, where $n' \leq n$ is the number of steps in which an atom has been unfolded (not including the last steps for unifying the equalities introduced by flattening, if any).

Now, we prove the claim by induction on $\mathcal{W}(\mathcal{D})$. We distinguish the following cases depending on the rule applied in the closed SLD tree to G_0:

- If a step with rule success is performed, the proof is done.
- If an unfolding step is applied, say $G_0 \rightarrow^{unf}_{\sigma} G_1$, the claim follows trivially by induction since the same step is performed in both trees and, thus, $\mathcal{W}(G_1 \leadsto^* \mathit{true}) < \mathcal{W}(G_0 \leadsto^* \mathit{true})$, with $\sigma \cdot \mathcal{SS}_\tau(G_1) \subseteq \mathcal{SS}_\tau(G_0)$.
- Consider now that a flattening step $G_0 \rightarrow^{flat} G_0'$ is applied in the closed SLD tree. By Theorem 5, we have that $\mathcal{SS}_P(G_0) = \mathcal{SS}_P(G_0')\ [\mathcal{V}ar(G_0)]$. Therefore, now we consider the goal G_0' and an associated derivation $G_0' \leadsto^* \mathit{true}$ in P. Trivially, $\mathcal{W}(G_0' \leadsto^* \mathit{true}) < \mathcal{W}(G_0 \leadsto^* \mathit{true})$ since both derivations are identical except for the additional equalities and either $|G_0'| < |G_0|$ or $sh(G_0') < sh(G_0)$. Hence, the claim follows by induction since $\mathcal{SS}_\tau(G_0) = \mathcal{SS}_\tau(G_0')$.
- Consider now that a closed SLD tree applies a splitting step $G_0 \rightarrow^{split} G_0^1, \ldots, G_0 \rightarrow^{split} G_0^m$. By Theorem 11, we have that $\mathcal{SS}_P(G_0) = \mathcal{SS}_P(G_0^1) \Uparrow \cdots \Uparrow \mathcal{SS}_P(G_0^m)$. Then, we now consider the independent SLD derivations $G_0^i \leadsto^*_{\theta_i} \mathit{true}$ in P, $i = 1, \ldots, m$. By induction (since $\mathcal{W}(G_0^i \leadsto^*_{\theta_i} \mathit{true}) < \mathcal{W}(G_0 \leadsto^* \mathit{true})$), we have that $\theta_i \in \mathcal{SS}_\tau(G_0^i)$. Therefore, by definition, we have $\theta_1 \Uparrow \cdots \Uparrow \theta_m \in \mathcal{SS}_\tau(G_0)$ and the claim follows.
- Consider finally that a subsumption step $G_0 \rightarrow^{sub}_{(G';\rho)} \mathit{true}$ is applied, with $\mathcal{SS}_\tau(G_0) = \rho \cdot \mathcal{SS}_\tau(G')$. Here, we proceed analogously as in the previous cases depending on the rule applied to G' in the closed SLD tree. This is safe by Theorem 13 and the fact that the path $G' \rightarrow^+ G$ in τ must contain at least one unfolding step (thus at some point the length of the derivation decreases and induction applies).

($\supseteq$) Assume now that $\theta \in \mathcal{SS}_\tau(G)$. By definition of function $\mathcal{SS}_\tau$, we can construct a derivation of the form $G \rightarrow^* \mathit{true}$ by replacing subsumption edges of the form $G_1 \rightarrow^{sub}_{(G',\rho)} \mathit{true}$ in τ by $G_1 \rightarrow^{sub}_{(G',\rho)} G'$ and, then, considering the edges issuing from G' in τ. The proof proceeds now by induction on the number of steps in such a derivation. We distinguish the following cases (analogously to the definition of function $\mathcal{SS}_\tau$):

- If $G \equiv \mathit{true}$, the proof is done (note that the case $G \equiv \mathit{fail}$ should not be considered since, in this case, no computed answer is obtained).
- If $G \rightarrow^{sub}_{(G',\rho)} G'$, then we have $\rho \cdot \mathcal{SS}_\tau(G') = \mathcal{SS}_\tau(G)$. Then, the claim follows by Theorem 13 and by induction, since the length of $G' \rightarrow^* \mathit{true}$ is strictly smaller than the length of $G \rightarrow^* \mathit{true}$.
- If $G \rightarrow^{flat} G'$ then, by Theorem 5, we have $\mathcal{SS}_P(G) = \mathcal{SS}_P(G')$. Therefore, the claim follows by induction since the number of steps in $G' \rightarrow^* \mathit{true}$ is strictly smaller than the number of steps in $G \rightarrow^* \mathit{true}$ and $\mathcal{SS}_\tau(G) = \mathcal{SS}_\tau(G')$.
- If $G \rightarrow^{split} G_1, \ldots, G \rightarrow^{split} G_n$ then, by Theorem 5, we have $\mathcal{SS}_P(G) = \mathcal{SS}_P(G_1) \Uparrow \cdots \Uparrow \mathcal{SS}_P(G_n)$. Therefore, the claim follows by induction since the number of steps in each derivation $G_i \rightarrow^* \mathit{true}$ is strictly smaller than the number of steps in $G \rightarrow^* \mathit{true}$ and $\mathcal{SS}_\tau(G) = \mathcal{SS}_\tau(G_1) \Uparrow \cdots \Uparrow \mathcal{SS}_\tau(G_n)$, $n > 1$.
- Finally, if $G \rightarrow^{unf}_{\sigma} G'$, the same step should also be performed in the original SLD tree and, thus, the claim follows by induction since $\sigma \cdot \mathcal{SS}_\tau(G') \subseteq \mathcal{SS}_\tau(G)$.

□

Therefore, the computed answers of the original program can still be obtained from an associated closed SLD tree. In some sense, our closed SLD trees are in between the original program (which also represents all possible answers implicitly) and the real (possibly infinite) SLD tree. Depending on the program and the accuracy of the extended SLD tree built, we will be closer to the original program or to the standard SLD tree. Nevertheless, our purpose is not to compute the answers for a given goal, but to use the finite representation of the search space for program analysis and transformation techniques.

Although Theorem 19 is essential to state the correctness of closed SLD trees, it might be insufficient in those cases where we need to analyze not only the computed answers, but also the calls that arise during the evaluation of a goal (which is essential, e.g., for some static analyses or model checking techniques).

Definition 20 (calls). Let P be a program, G a goal, and τ an (extended) SLD tree for G with P and some computation rule $\mathcal{R}$. We say that an atom a is a *call* in a derivation of G with P and $\mathcal{R}$ iff there is a path from G to G'

in τ and $G' \equiv (C_1 \wedge a \wedge C_2)$ for some (possibly empty) conjunctions C_1, C_2. We denote by $calls_\tau(G)$ the set of all calls in the derivations for G in τ.

Now, we can state the property mentioned above as follows:

Theorem 21. *Let P be a program, G a goal, and $\mathcal{R}$ a computation rule. Let τ be a (possibly infinite) SLD tree for G with P and $\mathcal{R}$ and let τ' be a closed SLD tree for G with P and $\mathcal{R}$. Then, for each call $a \in calls_\tau(G)$, there exists a call $a' \in calls_{\tau'}(G)$ such that $a' \leq a$.*

Proof. This result is a consequence of the fact that the same computation rule is used in both trees and the following easy property: if a (possibly infinite) SLD derivation $G_0 \leadsto G_1 \leadsto \cdots$ can be proved for some goal G_0 with P and $\mathcal{R}$ using the clauses $C_1, C_2, \ldots$, a corresponding derivation $G'_0 \leadsto G'_1 \leadsto \cdots$ can also be proved for $G' \leq G$ with P and $\mathcal{R}$ using the same clauses $C_1, C_2, \ldots$ and so that the same atoms are selected (though they might be more general in G'_i). In other words, generalizing some terms of a goal cannot discard any of the original derivations (but might introduce some more derivations that were not possible, of course).

Now, we prove a slightly more general claim: Let P be a program, G a goal, and $\mathcal{R}$ a computation rule. Let τ be a (possibly infinite) SLD tree for G with P and $\mathcal{R}$ and let τ' be a closed SLD tree for G' with P and $\mathcal{R}$, where $G' \leq G$ (but the predicate symbols are the same, i.e., only terms can be generalized). Then, for each call $a \in calls_\tau(G)$, there exists a call $a' \in calls_{\tau'}(G')$ such that $a' \leq a$.

Let us consider a call $a \in calls_\tau(G)$. Therefore, there exists a derivation $G \equiv G_0 \leadsto G_1 \leadsto \cdots \leadsto C_1, a, C_2$ for G with P and R in τ. We prove the above claim by induction on the length of the derivation $G_0 \leadsto G_1 \leadsto \cdots \leadsto C_1, a, C_2$. We distinguish the following cases, depending on the rule applied in τ' to the corresponding goal G':

- For the cases success and failure, the claim follows trivially.
- If an unfolding step is applied, both trees would proceed analogously, so $G'_0 \rightarrow^{unf} G'_1$. Then, it is straightforward that $G'_1 \leq G_1$ since $G'_0 \leq G_0$. Therefore, the claim follows by induction.
- When a flattening step is performed, we have that

  $$G'_0 \equiv C_1 \wedge C_2\theta \wedge C_3 \rightarrow^{flat} C_1 \wedge \widehat{\theta} \wedge C_2 \wedge C_3 \equiv G'_1$$

  Now, since we assumed that $\widehat{\theta}$ cannot be unfolded until all other atoms are fully evaluated, we trivially have that $calls_{\tau'}(G'_1) = calls(G''_1)$, with

$G_1'' \equiv C_1 \wedge C_2 \wedge C_3$. Now, we consider $G_1'' \leq G_0' \leq G_0$ and the proof proceeds by choosing some of these cases again. Observe that the number of possible flattenings is finite and, thus, at some point we should apply a different rule so that the claim will follow by induction.

- If an splitting step is performed: $G_0' \rightarrow^{split} G_{01}', \ldots, G_0' \rightarrow^{split} G_{0m}'$. First, let us consider the partitioning function p applied in this splitting step. Trivially, for all call $a \in calls_\tau(G_0)$, there exists $a' \in calls_\tau(G_{01}) \cup \cdots \cup calls_\tau(G_{0m})$ with $a' \leq a$ and $p(G_0) = \{\{G_{01}, \ldots, G_{0m}\}\}$, since the bindings coming from the evaluation of previous subgoals are not applied anymore. On the other hand, we have $calls_{\tau'}(G_0') = calls_{\tau'}(G_{01}') \cup \ldots \cup calls_{\tau'}(G_{0m}')$. Therefore, the claim follows by induction considering the derivations for G_{0i} in τ since $G_{0i}' \leq G_{0i}$, $i = 1, \ldots, m$, $m > 1$.

- Finally, if a subsumption step $G_0' \rightarrow^{sub}_{(G',\rho)} \mathit{true}$ is performed, then we have $calls_{\tau'}(G_0') \approx calls_{\tau'}(G')$ and, thus, for all $a \in calls_{\tau'}(G_0')$ there exists $a' \in calls_{\tau'}(G')$ with $a \leq a'$ and $a' \leq a$. Trivially, we have $G' \leq G_0' \leq G_0$. Now, we choose again some of these cases depending on the applied rule in τ'. As in the previous proof, this is safe by the fact that at least one unfolding step must be applied between subsumption steps and, then, induction applies. □

3.3. Success Set Equations

Finally, we show that the success set represented by a closed SLD tree can be compactly represented using equations. We follow a similar construction as the one in [26] for rewrite systems. First, our equational notation considers the following three operators:[4]

- Composition ($\cdot$), i.e., the standard composition on substitutions extended to sets, as mentioned before.

- Alternative ($+$), where an expression like $ss_1 + ss_2$ denotes the union of the success sets denoted by ss_1 and ss_2.

- Parallel composition ($\Uparrow$), i.e., the operator introduced in Definition 9.

Now, we introduce a technique to extract the success set equations of a goal from a given closed SLD tree. Intuitively speaking, the process starts by partitioning the closed SLD tree into those subtrees that are rooted by a

[4] As for the operator precedence, we assume that composition has a higher priority than parallel composition, which has a higher priority than alternative.

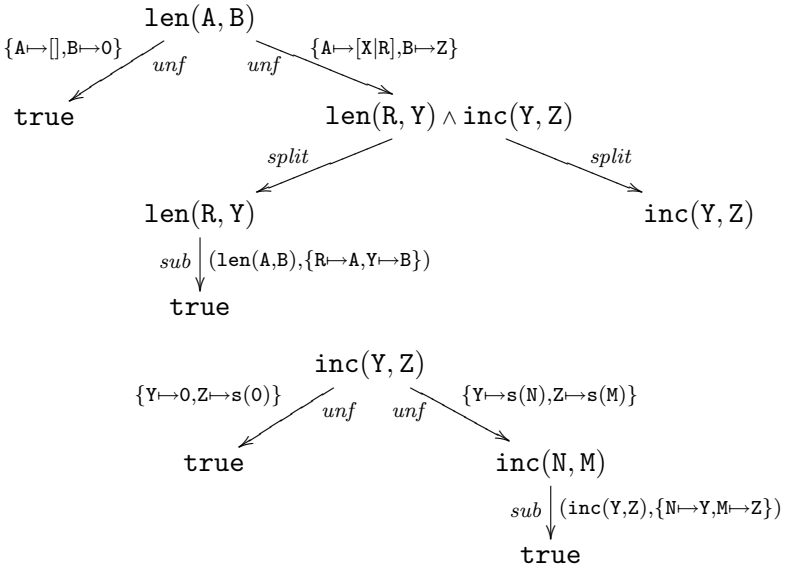


Figure 4: Subtrees of the extended SLD tree in Fig. 2

node which has been used to perform a subsumption step. For instance, for the tree τ of Fig. 2, we consider the two subtrees shown in Fig. 4 (note that $\mathtt{inc(Y, Z)}$ occurs in both trees). Then, an equation is produced for each subtree following these ideas: substitutions along derivations with SLD resolution steps are just composed; the success sets of the different branches issuing from a goal are put together using the alternative operator; flattening steps are ignored; splitting steps involve computing the parallel composition of the success sets of the different branches; finally, for subsumption steps, we compose the current set with the substitution labeling the step and, then, with the success set of the previous variant goal.

In the following, given a closed SLD tree τ, we denote with $\mathit{subtrees}(\tau)$ the set of subtrees of τ that are obtained by partitioning τ into those subtrees that are rooted by a node which has been used to perform a subsumption step in the tree (i.e., that occurs in the label (G', ρ) of some subsumption edge). Note that the nodes used for subsumption are duplicated in these subtrees (as the root of one of them, and a leaf in the other one).

Definition 22 (success set equations). Let P be a program and G_0 a goal. Let τ_0 be a closed SLD tree for G_0 with P. Let $\mathcal{T} = \mathit{subtrees}(\tau_0)$. Then, we produce a success set equation $[F_G = eq_\tau(G)]$ for each tree $\tau \in \mathcal{T}$ with $\mathsf{root}(\tau) = G$, where the definition of the auxiliary function eq_τ is shown in Fig. 5.

$$eq_\tau(G) = \begin{cases} id & \text{if } G \equiv true \\ fail & \text{if } G \equiv fail \\ F_G & \text{if } G \text{ is a leaf different from } true \text{ and } fail \\ \rho \cdot F_{G'} & \text{if } G \rightarrow^{sub}_{(G',\rho)} true \in \tau \\ eq_\tau(G') & \text{if } G \rightarrow^{flat} G' \in \tau \\ eq_\tau(G_1) \Uparrow \cdots \Uparrow eq_\tau(G_n) & \text{if } G \rightarrow^{split} G_1, \ldots, G \rightarrow^{split} G_n \in \tau \\ \sigma_1 \cdot eq_\tau(G_1) + \cdots + \sigma_n eq_\tau(G_n) & \text{if } G \rightarrow^{unf}_{\sigma_1} G_1, \ldots, G \rightarrow^{unf}_{\sigma_n} G_n \in \tau \end{cases}$$

Figure 5: Auxiliary function eq_τ

Note that we have $\sigma \cdot fail = fail$, $fail \Uparrow \Theta = \Theta \Uparrow fail = fail$, and $fail \cup \Theta = \Theta \cup fail = \Theta$.

Here, we denote with $\mathsf{sols}(F_G)$ the (possibly infinite) set of solutions of the success set equation F_G for some goal G.

Let us consider a set of success set equations $F_{G_1} = r_1, \ldots, F_{G_n} = r_n$ associated to the SLD derivations starting from a goal G_1. A procedure to enumerate the substitutions in $\mathsf{sols}(F_{G_1})$ can proceed as follows:

1. Initialization. $F^0_{G_1} = \cdots = F^0_{G_n} = \emptyset$.
2. Iterative process. For all $i > 0$, we compute the following sets:

$$F^i_{G_1} = r_1[F_G \mapsto F^{i-1}_G] \qquad \ldots \qquad F^i_{G_n} = r_n[F_G \mapsto F^{i-1}_G]$$

where $r_j[F_G \mapsto F^{i-1}_G]$ denotes the expression that results from r_j by replacing every occurrence of F_G by F^{i-1}_G, with $j = 1, \ldots, n$ and $G \in \{G_1, \ldots, G_n\}$.

Then, we have $\mathsf{sols}(F_{G_1}) = \bigcup_{i>0} F^i_{G_1}$, where the $F^i_{G_1}$ are computed as above.

We do not formally prove the correctness of the above procedure for computing $\mathsf{sols}(F_G)$ since it is rather standard.

Example 23. Let us consider the following slight variation of the program in Example 15:

$\mathtt{len}([\,],[\,]).$ $\qquad$ $\mathtt{inc}(\mathtt{0},\mathtt{s}(\mathtt{0})).$

$\mathtt{len}([\mathtt{X}|\mathtt{R}],\mathtt{Z}) \leftarrow \mathtt{len}(\mathtt{R},\mathtt{Y}) \wedge \mathtt{inc}(\mathtt{Y},\mathtt{Z}).$ $\qquad$ $\mathtt{inc}(\mathtt{s}(\mathtt{N}),\mathtt{s}(\mathtt{M})) \leftarrow \mathtt{inc}(\mathtt{N},\mathtt{M}).$

and the initial goal $G \equiv \mathtt{len}(\mathtt{A},\mathtt{B})$. We consider a closed SLD tree which is essentially the same as that of Figure 2, that we decompose into two subtrees

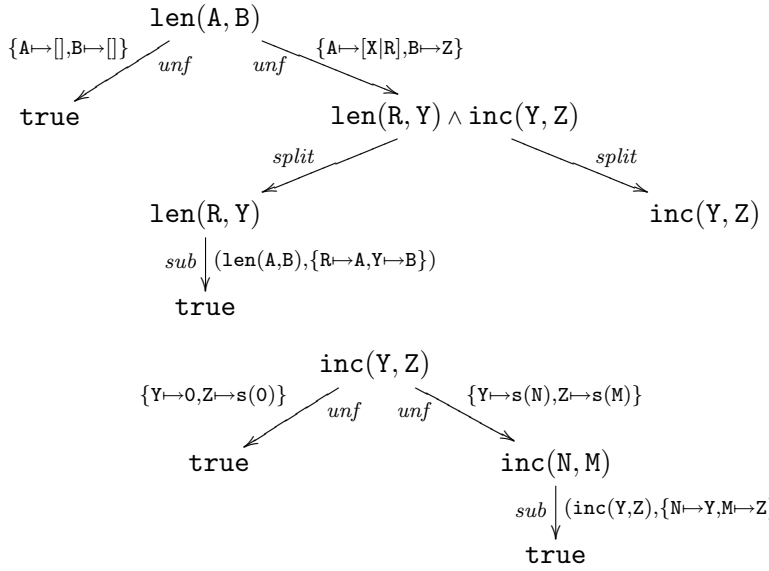


Figure 6: Subtrees from Example 23

as shown in Figure 6. Then, the success set equations using the subtrees of Fig. 6 are the following:

$$\begin{array}{rcl} F_G & = & \{\mathtt{A} \mapsto [\,], \mathtt{B} \mapsto [\,]\} \\ & + & \{\mathtt{A} \mapsto [\mathtt{X}|\mathtt{R}], \mathtt{B} \mapsto \mathtt{Z}\} \cdot (\{\mathtt{R} \mapsto \mathtt{A}, \mathtt{Y} \mapsto \mathtt{B}\} \cdot F_G \Uparrow F_{G'}) \\ F_{G'} & = & \{\mathtt{Y} \mapsto \mathtt{0}, \mathtt{Z} \mapsto \mathtt{s}(\mathtt{0})\} \\ & + & \{\mathtt{Y} \mapsto \mathtt{s}(\mathtt{N}), \mathtt{Z} \mapsto \mathtt{s}(\mathtt{M})\} \cdot \{\mathtt{N} \mapsto \mathtt{Y}, \mathtt{M} \mapsto \mathtt{Z}\} \cdot F_{G'} \end{array}$$

where $G \equiv \mathtt{len}(\mathtt{A}, \mathtt{B})$ and $G' \equiv \mathtt{inc}(\mathtt{Y}, \mathtt{Z})$.

Informally speaking, the (infinite) solutions of these equations can be enumerated iteratively as follows. One starts with $F^0_G = \{\,\}$ and $F^0_{G'} = \{\,\}$. Then, we compute the next iteration $i > 0$ as follows:

$$\begin{array}{rcl} F^i_G & = & \{\mathtt{A} \mapsto [\,], \mathtt{B} \mapsto [\,]\} \\ & + & \{\mathtt{A} \mapsto [\mathtt{X}|\mathtt{R}], \mathtt{B} \mapsto \mathtt{Z}\} \cdot (\{\mathtt{R} \mapsto \mathtt{A}, \mathtt{Y} \mapsto \mathtt{B}\} \cdot F^{i-1}_G \Uparrow F^{i-1}_{G'}) \\ F^i_{G'} & = & \{\mathtt{Y} \mapsto \mathtt{0}, \mathtt{Z} \mapsto \mathtt{s}(\mathtt{0})\} \\ & + & \{\mathtt{Y} \mapsto \mathtt{s}(\mathtt{N}), \mathtt{Z} \mapsto \mathtt{s}(\mathtt{M})\} \cdot \{\mathtt{N} \mapsto \mathtt{Y}, \mathtt{M} \mapsto \mathtt{Z}\} \cdot F^{i-1}_{G'} \end{array}$$

Therefore, we have the following finite sequence:[5]

$$F^1_G = \{\{\mathtt{A} \mapsto [\,], \mathtt{B} \mapsto [\,]\}\} \qquad F^2_G = F^1_G$$

[5]We restrict substitutions to $\mathcal{V}ar(\mathtt{len}(\mathtt{A}, \mathtt{B}))$ for conciseness.

Actually, it is easy to see that, in this case, the only solution of F_G is $\{\mathtt{A} \mapsto [\,], \mathtt{B} \mapsto [\,]\}$ because the parallel composition $\{\mathtt{R} \mapsto \mathtt{A}, \mathtt{Y} \mapsto \mathtt{B}\} \cdot F_G \Uparrow F_{G'}$ has no solutions, since variable $\mathtt{Y}$ is bound to $[\,]$ in the leftmost expression, $\{\mathtt{R} \mapsto \mathtt{A}, \mathtt{Y} \mapsto \mathtt{B}\} \cdot F_G$, and to $\mathtt{0}$ in the rightmost one, $F_{G'}$. Indeed, one can conclude that the program was incorrect: the bug is in the first clause of `len`, which should be $\mathtt{len}([\,], \mathtt{0})$, as in Example 15.

The success set equations can be useful, e.g., for program comprehension or formally analyzing the success set of a program.

4. Construction of Closed SLD Trees

In this section, we introduce some strategies that allow us to construct closed SLD trees with different levels of abstraction. We illustrate their application through some examples.

4.1. Maximal Abstraction

Our first strategy considers a maximal abstraction so that the closed SLD tree will be very small. As a counterpart, the information represented by the tree is not so different from just considering the text of the original program. Intuitively speaking, it constructs a sort of *call graph* for the program.

The rules for constructing a closed SLD tree with this strategy can be found in Fig. 7. Here, we assume that the rules are tried by following a textual order and discarding the application of all the other rules as soon as a rule is applicable. We do it for simplicity; otherwise some additional premises would be required. Basically, this strategy applies splitting in order to only consider atomic goals and, moreover, applies flattening to always consider (a renaming of) the atoms in the body of the program clauses without the instantiation coming from the unification with the head of the clause.

Straightforwardly, this strategy produces a closed SLD tree since the number of (variants of) the atoms in the bodies of the clauses is finite.

Example 24 (improving termination analysis). Consider the following simple program $P_{\mathtt{loop}}$:

$$\begin{array}{lll} \mathtt{nat(0)}. & & \mathtt{loop(a)}. \\ \mathtt{nat(s(X))} \leftarrow \mathtt{nat(X)} \wedge \mathtt{loop(a)}. & & \mathtt{loop(b)} \leftarrow \mathtt{loop(b)}. \end{array}$$

and the initial goal $G \equiv \mathtt{nat(X)}$. By using the maximal abstraction, we get the closed SLD tree of Figure 8.[6] Now, one can analyze the transitions in this

[6]We skip the flattening steps since they are not necessary, i.e., we already derive a variant of the body atoms by unfolding. We often skip these steps in this section when they are not necessary to ensure termination.

$$\text{(success)} \qquad \frac{}{\tau[true] \longrightarrow \tau[true]}$$

$$\text{(failure)} \qquad \frac{G \not\equiv true \text{ and } \nexists G' \text{ such that } G \rightsquigarrow_{P,\mathcal{R},\sigma} G'}{\tau[G] \longrightarrow \tau[\underline{fail}]}$$

$$\text{(subsumption)} \qquad \frac{\begin{array}{c}\exists G' \in \tau : \ G' \rightarrow^{+} G \text{ in } \tau \text{ includes at least one unfolding step} \\ \text{and } G\rho = G' \text{ with } \rho \text{ a renaming substitution}\end{array}}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{sub}_{(G',\rho)} \underline{true}]}$$

$$\text{(flattening)} \qquad \frac{\sigma = \theta \restriction \mathcal{V}ar(G')}{\tau[\underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\theta)} G'\theta] \longrightarrow \tau[\underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\theta)} \underline{G'\theta} \rightarrow^{flat} \widehat{\sigma} \wedge G']}$$

$$\text{(splitting)} \qquad \frac{G \equiv a_1 \wedge \cdots \wedge a_n \text{ and } n > 1}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{split} a_1, \ldots, \underline{G} \rightarrow^{split} a_n]}$$

$$\text{(unfolding)} \qquad \frac{G \rightsquigarrow_{P,\mathcal{R},\sigma_1} G_1, \ldots, G \rightsquigarrow_{P,\mathcal{R},\sigma_n} G_n}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\sigma_1)} G_1, \ldots, \underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\sigma_n)} G_n]}$$

Figure 7: Closed SLD trees using maximal abstraction

tree (as in, e.g., [32]) in order to prove termination rather than the clauses of the original program. Trivially, one will infer that the program terminates for `nat(X)` (and its instances) while inspecting the source program (without assuming any particular class of input goals) one would conclude that it is not terminating because of the clause `loop(b)` ← `loop(b)`. Therefore, our technique might be useful for improving goal-dependent analyses.

Example 25 (partial evaluation). Consider the following program $P_{\texttt{ilist}}$:

```
ilist([ ], I, [ ]).
ilist([X|R], I, [XI|RI]) ← ilist(R, I, RI) ∧ add(I, X, XI).
add(0, Y, Y).
add(s(X), Y, s(Z)) ← add(X, Y, Z).
```

and the initial goal $G \equiv$ `ilist(A, s(0), C)`. The standard SLD tree for G with program $P_{\texttt{ilist}}$ (using, e.g., a left-to-right computation rule) is trivially infinite. Fig. 9 shows the closed SLD tree for G with $P_{\texttt{ilist}}$ using the maximal abstraction strategy, where predicates `ilist` and `add` are abbreviated to `i` and `a`, respectively, and ρ_1, ρ_2 are the obvious variable renamings. Moreover,

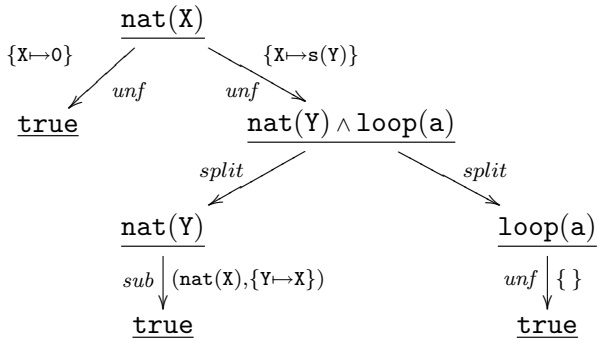


Figure 8: Closed SLD tree for $\texttt{nat(X)}$ with $P_{\texttt{loop}}$ using maximal abstraction.

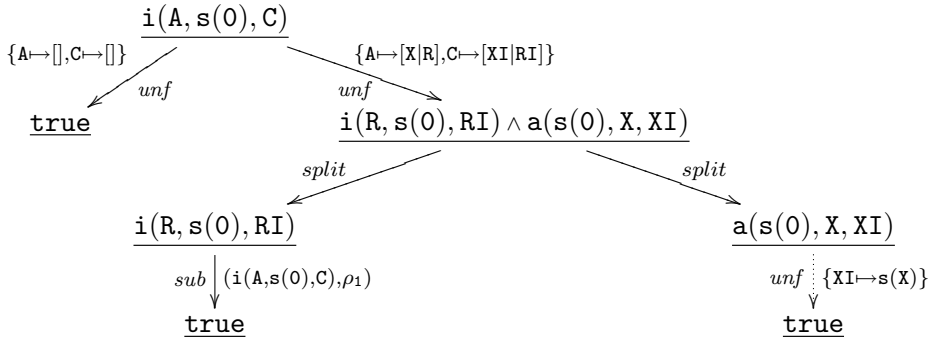


Figure 9: Closed SLD tree for $\texttt{ilist(X, Y)}$ with $P_{\texttt{ilist}}$ using maximal abstraction.

to simplify the presentation, we skip some steps that are denoted with a dotted arrow.

This is a well-known example for partial evaluation. The maximal abstraction strategy, however, is not successful in this case, since it would produce a residual program that is basically identical to the original one.

4.2. Depth-k Abstraction

Our next strategy is parametric w.r.t. a pair (c, d), where c is the maximum number of atoms in a goal and d is the maximum depth (number of nested function symbols) in the arguments of predicate calls. By restricting the number of atoms in a goal and the depth of terms, we ensure a finite domain of goals (up to variable renaming) and, thus, the construction of a closed SLD tree.

The depth-k abstraction strategy w.r.t. a pair (c, d) is formalized in Fig. 10. As before, we assume that the rules are tried by following a textual

order and discarding the application of all the other rules as soon as a rule is applicable. This strategy is similar to the previous one, but delay flattening and splitting until they become necessary to ensure the conditions on the number of atoms in a goal and the depth of the terms:

- Flattening. We denote by $depth(t)$ the usual function that returns the maximum number of nested function symbols in a term t. We extend it to atoms and goals as follows:

$$\begin{array}{rcl} depth(p(t_1,\ldots,t_n)) & = & \max(\{depth(t_i) \mid i = 1,\ldots,n\}) \\ depth(a_1 \wedge \cdots \wedge a_m) & = & \max(\{depth(a_i) \mid i = 1,\ldots,m\}) \end{array}$$

  Then, given a goal G and a natural number d, we let $\mathit{flat}(G,d)$ return the pair (θ, G') such that $G'\theta = G$ and $depth(G) \leq d$. Here, the function flat only generalizes the terms that violate the depth condition (i.e., we assume that there is no overgeneralization).

  Observe that this strategy is correct since we assumed that the equations introduced by a flattening step cannot be unfolded until a goal only contains equations (see page 11).

- Given a goal $G \equiv (a_1 \wedge \cdots \wedge a_n)$, an extended SLD tree τ and a natural number c with $n > c$, $split_\tau(G,c)$ applies some partition function to return a number of subgoals $\{\{G_1,\ldots,G_m\}\}$ so that none of them has more than c atoms. Here, one can consider different criteria, like trying to ensure that atoms with shared variables are kept together in the same subgoal or using strategies that look for the best matching conjunction in order to ease subsequent subsumption steps (as in [14]).

As mentioned above, this strategy always produces a closed SLD tree since the number of (variants of) atoms and goals is finite.

Example 26 (partial evaluation). Consider again the program $P_{\texttt{ilist}}$ of Example 25 and the initial goal $G \equiv \texttt{ilist(A, s(0), C)}$. Fig. 11 shows the closed SLD tree for G with $P_{\texttt{ilist}}$ and the left-to-right computation rule, using the depth-k strategy and the pair $(2,2)$, where ρ_1, ρ_2, ρ_3 are the obvious renaming mappings.

Unfortunately, the tree is closed but it is still not appropriate for partial evaluation, since the call to `add` has not been unfolded. Luckily, this strategy can be easily refined, e.g., by introducing an unfolding operator that selects the leftmost atom that is not a variant of a previously unfolded atom in the same derivation (a well-known unfolding strategy in the partial evaluation literature). In this way, we can obtain the much simpler closed SLD tree

$$\text{(success)} \qquad \frac{}{\tau[true] \longrightarrow \tau[true]}$$

$$\text{(failure)} \qquad \frac{G \not\equiv true \text{ and } \nexists G' \text{ such that } G \rightsquigarrow_{P,\mathcal{R},\sigma} G'}{\tau[G] \longrightarrow \tau[\underline{fail}]}$$

$$\text{(subsumption)} \qquad \frac{\begin{array}{c}\exists G' \in \tau : \ G' \rightarrow^{+} G \text{ in } \tau \text{ includes at least one unfolding step} \\ \text{and } G\rho = G' \text{ with } \rho \text{ a renaming substitution}\end{array}}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{sub}_{(G',\rho)} \underline{true}]}$$

$$\text{(flattening)} \qquad \frac{depth(G) > d \text{ and } flat(G,d) = (\theta, G')}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{flat} \widehat{\theta} \wedge G']}$$

$$\text{(splitting)} \qquad \frac{G \equiv a_1 \wedge \cdots \wedge a_n,\ n > c \geq 1, \text{ and } split_\tau(G,c) = \{\{G_1, \ldots, G_n\}\}}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{split} G_1, \ldots, \underline{G} \rightarrow^{split} G_n]}$$

$$\text{(unfolding)} \qquad \frac{G \rightsquigarrow_{P,\mathcal{R},\sigma_1} G_1, \ldots, G \rightsquigarrow_{P,\mathcal{R},\sigma_n} G_n}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\sigma_1)} G_1, \ldots, \underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\sigma_n)} G_n]}$$

Figure 10: Closed SLD trees using depth-k abstraction w.r.t. (c, d)

shown in Fig. 12. The nice thing with this tree is that there are no splitting steps and, therefore, both the success set and the calls are preserved (considering the same computation rule in the original SLD tree, i.e., selecting first `add` and then `ilist`). Moreover, partial evaluation may produce an optimal residual program from this tree:[7]

```
inc([ ], s(0), [ ]).
inc([X|R], s(0), [s(X)|RI]) ← inc(R, s(0), RI).
```

Example 27 (program comprehension and debugging). Our last example shows the application of our framework to detect an error in a program.

[7]Actually, the second term, `s(0)`, would also be deleted from the atoms by renaming. We skip this step for simplicity.

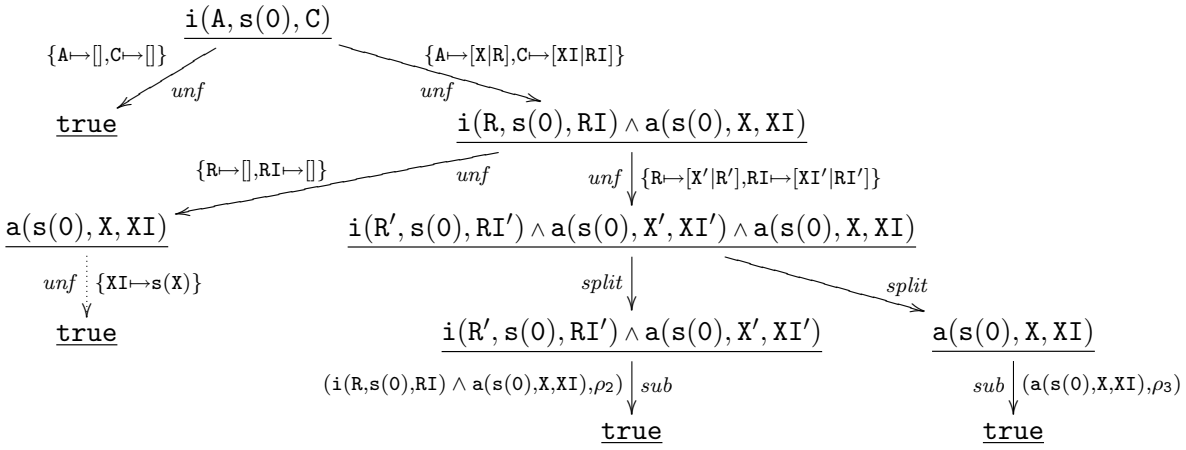


Figure 11: Closed SLD tree for $\texttt{ilist}(\texttt{X},\texttt{Y})$ with $P_{\texttt{ilist}}$ using depth-k abstraction.

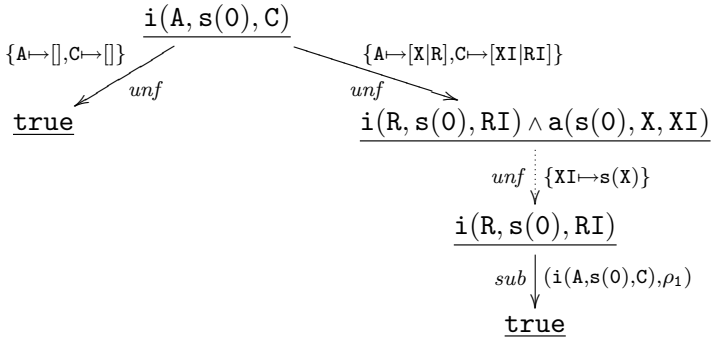


Figure 12: Refined closed SLD tree for $\texttt{ilist}(\texttt{X},\texttt{Y})$ with $P_{\texttt{ilist}}$ using depth-k abstraction.

Consider the following logic program $P_{\texttt{max}}$ to compute the maximum of a list:

$$\begin{array}{ll}
\texttt{max}([\texttt{X}|\texttt{R}],\texttt{Y}) \leftarrow \texttt{max}(\texttt{R},\texttt{X},\texttt{Y}). & \texttt{max}([],\texttt{M},\texttt{M}).\\
 & \texttt{max}([\texttt{X}|\texttt{R}],\texttt{M},\texttt{Y}) \leftarrow \texttt{g}(\texttt{X},\texttt{M}) \wedge \texttt{max}(\texttt{R},\texttt{X},\texttt{Y}).\\
 & \texttt{max}([\texttt{X}|\texttt{R}],\texttt{M},\texttt{Y}) \leftarrow \texttt{l}(\texttt{X},\texttt{M}) \wedge \texttt{max}(\texttt{R},\texttt{M},\texttt{Y}).\\
\texttt{g}(\texttt{s}(\texttt{X}),\texttt{0}). & \texttt{l}(\texttt{0},\texttt{s}(\texttt{X})).\\
\texttt{g}(\texttt{s}(\texttt{X}),\texttt{s}(\texttt{Y})) \leftarrow \texttt{g}(\texttt{X},\texttt{Y}). & \texttt{l}(\texttt{s}(\texttt{X}),\texttt{s}(\texttt{Y})) \leftarrow \texttt{l}(\texttt{X},\texttt{Y}).
\end{array}$$

and an initial goal of the form $\texttt{max}(\texttt{L},\texttt{M})$. For simplicity, we only show the subtree starting with $G \equiv \texttt{g}(\texttt{X},\texttt{M}) \wedge \texttt{max}(\texttt{R},\texttt{X},\texttt{Y})$ in Figure 13, where a leftmost selection strategy is considered. By also considering the subtree for $G' \equiv \texttt{l}(\texttt{X},\texttt{M}) \wedge \texttt{max}(\texttt{R},\texttt{X},\texttt{Y})$ which is rather similar, we get the following success set

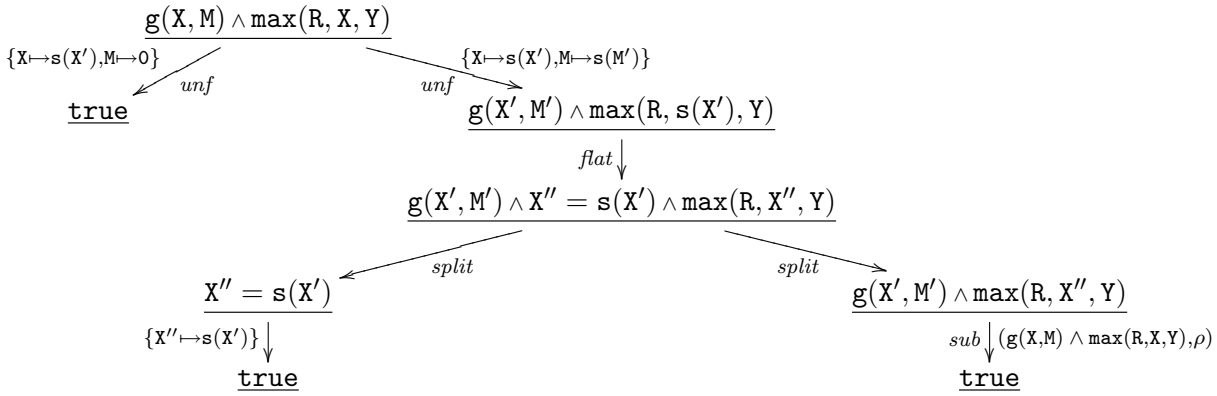


Figure 13: Fragment of a closed SLD tree for $\mathtt{max}(\mathtt{L},\mathtt{M})$ with $P_{\mathtt{max}}$ using depth-k abstraction.

equations:

$$\begin{array}{rcl} F_G & = & \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}'), \mathtt{M} \mapsto \mathtt{0}\} \\ & + & \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}'), \mathtt{M} \mapsto \mathtt{s}(\mathtt{M}')\} \cdot (\{\mathtt{X}'' \mapsto \mathtt{s}(\mathtt{X}')\} \Uparrow \{\rho \cdot F_G\}) \\ F_{G'} & = & \{\mathtt{X} \mapsto \mathtt{0}, \mathtt{M} \mapsto \mathtt{s}(\mathtt{M}')\} \\ & + & \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}'), \mathtt{M} \mapsto \mathtt{s}(\mathtt{M}')\} \cdot (\{\mathtt{X}'' \mapsto \mathtt{s}(\mathtt{X}')\} \Uparrow \{\rho' \cdot F_{G'}\}) \end{array}$$

Now, just by inspecting these equations, one can easily infer that the program is incorrect since it does not consider the case when there are repeated elements in the list, as can be seen by looking at the base cases of these equations. Actually, the solutions have the form:

$$\begin{array}{rcl} F_G & = & \{\{\mathtt{X} \mapsto \mathtt{s}(\mathtt{X}'), \mathtt{M} \mapsto \mathtt{0}\}, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{s}(\mathtt{X}'')), \mathtt{M} \mapsto \mathtt{s}(\mathtt{0})\}, \ldots\} \\ F_{G'} & = & \{\mathtt{X} \mapsto \mathtt{0}, \mathtt{M} \mapsto \mathtt{s}(\mathtt{M}')\}, \{\mathtt{X} \mapsto \mathtt{s}(\mathtt{0}), \mathtt{M} \mapsto \mathtt{s}(\mathtt{s}(\mathtt{M}''))\}, \ldots\} \end{array}$$

5. Test Case Generation

In this section, we present a test case generation procedure based on the construction of closed SLD trees. For this purpose, we consider an extension of the depth-k strategy presented in Section 4.2. Moreover, we do not follow the standard leftmost computation rule, but introduce a more refined unfolding rule based on *homeomorphic embedding*. Intuitively, we say that atom a_i *embeds* atom a_j, denoted by $a_i \trianglerighteq a_j$, when a_j can be obtained from a_i by deleting symbols (see [21] for a survey on the use of embedding for ensuring termination in online partial evaluation).

Definition 28 (homeomorphic embedding). The homeomorphic embedding relation $\trianglerighteq$ is defined as the smallest relation satisfying $x \trianglerighteq y$ for all $x, y \in \mathcal{V}$, and $s \equiv f(s_1, \ldots, s_m) \trianglerighteq g(t_1, \ldots, t_n) \equiv t$, if and only if

1. $f \equiv g$ (with $m = n$) and $s_i \trianglerighteq t_i$ for all $i = 1, \ldots, n$ or
2. $s_j \trianglerighteq t$, for some j, $1 \leq j \leq m$.

For instance, $p(f(X), h(a, b))$ embeds $p(X, a)$, but $p(f(X), h(a, b))$ does not embed $p(X, f(a))$.

Definition 29 (covering ancestors [11]**).** Given an SLD resolution step

$$a_1 \wedge \cdots \wedge a_i \wedge \cdots \wedge a_n \ \leadsto_\sigma \ (a_1 \wedge \cdots \wedge a_{i-1} \wedge a'_1 \wedge \cdots \wedge a'_m \wedge a_{i+1} \wedge \cdots \wedge a_n)\sigma$$

with selected atom a_i using clause $a \leftarrow a'_1, \ldots, a'_m$, $\sigma = \mathsf{mgu}(a_i = a)$, we say that a_i is the *parent* of atoms $a'_1\sigma, \ldots, a'_m\sigma$ in this step. The ancestor relation is just the transitive closure of the parent relation.

The *covering ancestors* of a query atom in an SLD derivation is the subset of its ancestors with the same predicate symbol.

Here, we consider an unfolding rule that takes a goal and applies an SLD resolution step by selecting the leftmost atom that does not embed a covering ancestor. This strategy is terminating by Kruskal theorem [20]; eventually, all goals are successful, a failure, or they contain no selectable atom (because all of them embed a previously selected atom). We formalize this unfolding strategy as follows (analogously to, e.g., [11]):

Definition 30 (unfolding strategy, $\overset{\triangleright}{\leadsto}$). Let P be a program and $G_0 \leadsto^*_\theta G_n$ be an SLD derivation such that $G_n \leadsto G_{n+1}$ for some computation rule (i.e., G_n is not the end of the derivation). Then, we have $G_n \overset{\triangleright}{\leadsto}_\sigma G_{n+1}$ if the SLD resolution step $G_n \leadsto_{P,\mathcal{R},\sigma} G_{n+1}$ holds for some computation rule $\mathcal{R}$ and the selected atom, a, is the leftmost atom such that there is no covering ancestor b in $G_0 \leadsto^*_\theta G_n$ with $a \trianglerighteq b$ (if any).

It is not difficult to prove (e.g., from the results in [11]) that our unfolding strategy guarantees that the number of unfolding steps in a derivation is finite.

Regarding splitting, we follow the approach of [35, 36] that is based on identifying *non-regular* predicates to determine when splitting is required to ensure termination. For this purpose, a syntactic characterization that allows us to identify which predicate calls might give rise to infinitely growing conjunctions is introduced. In the following, we say that the *call graph* of a program P is a directed graph that contains the predicate symbols of P as vertices and an edge from predicate p/n to predicate q/m for each clause $p(t_1, \ldots, t_n) \leftarrow \mathit{body}$ and atom $q(s_1, \ldots, s_m)$ of body.

Definition 31 (strongly regular logic programs [35]). Let P be a logic program and let $CG_1, \ldots, CG_n$ be the strongly connected components (SCC) in the call graph of P. We say that P is strongly regular if there is no clause $p(t_1, \ldots, t_n) \leftarrow \mathit{body}$ such that body contains two atoms $q(s_1, \ldots, s_m)$ and $r(l_1, \ldots, l_k)$ such that q/m and r/k belong to the same SCC of p/n.

Intuitively speaking, strongly regular programs cannot produce infinitely growing conjunctions in an extended SLD tree if the usual dynamic computation rules are considered (like the one above which is based on unfolding the leftmost atom that does not embed a previously unfolded atom).

When a program P is not strongly regular, we identify the predicates that are responsible for violating the strongly regular condition: a predicate p/n is *non-regular* if there is a clause $p(t_1, \ldots, t_n) \leftarrow \mathit{body}$ and body contains two atoms with predicates q/m and r/k that belong to the same SCC of p/n.

Example 32. Consider the following Prolog program (from the DPPD library [22]):

```
applast(L,X,Last) :- append(L,[X],LX), last(Last,LX).
last(X,[X]).
last(X,[H|T]) :- last(X,T).
append([],L,L).
append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).
```

Here, there are three SCCs, {`applast/3`}, {`append/3`} and {`last/2`}, but no clause violates the strongly regular condition. In contrast, the following program (also from the DPPD library [22]):

```
flipflip(XT,YT) :- flip(XT,TT), flip(TT,YT).
flip(leaf(X),leaf(X)).
flip(tree(L,I,R),tree(FR,I,FL)) :- flip(L,FL), flip(R,FR).
```

is not strongly regular. Here, we have two SCCs, {`flipflip/2`} and {`flip/2`}, and the second clause of `flip/2` violates the strongly regular condition. As a consequence, we have that `flip/2` is a non-regular predicate.

The splitting operator then proceeds as follows:

Definition 33 (regular splitting [35]). Let G be a goal and p a partition function with $p(G) = \{\{G_1, \ldots, G_n\}\}$, $n \geq 1$. Let S be a set of non-regular predicates. We say that the sequence of subgoals $G_1, \ldots, G_n$ is a regular splitting of G, in symbols $\{\{G_1, \ldots, G_n\}\} \in \mathsf{split}_S(G)$, if the following conditions hold:

$$\text{(success)} \qquad \frac{}{\tau[true] \longrightarrow \tau[true]}$$

$$\text{(failure)} \qquad \frac{G \not\equiv true \text{ and } \nexists G' \text{ such that } G \rightsquigarrow_{P,\mathcal{R},\sigma} G'}{\tau[G] \longrightarrow \tau[\underline{fail}]}$$

$$\text{(subsumption)} \qquad \frac{\begin{array}{c}\exists G' \in \tau : \ G' \rightarrow^{+} G \text{ in } \tau \text{ includes at least one unfolding step} \\ \text{and } G\rho = G' \text{ with } \rho \text{ a renaming substitution}\end{array}}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{sub}_{(G',\rho)} \underline{true}]}$$

$$\text{(unfolding)} \qquad \frac{G \overset{\rhd}{\rightsquigarrow}_{\sigma_1} G_1, \ldots, G \overset{\rhd}{\rightsquigarrow}_{\sigma_n} G_n, \ n > 0}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\sigma_1)} G_1, \ldots, \underline{G} \rightarrow^{unf}_{(P,\mathcal{R},\sigma_n)} G_n]}$$

$$\text{(flattening)} \qquad \frac{depth(G) > d \text{ and } flat(G,d) = (\theta, G')}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{flat} \widehat{\theta} \wedge G']}$$

$$\text{(splitting)} \qquad \frac{\{\{G_1, \ldots, G_n\}\} \in \mathsf{split}_S(G)}{\tau[G] \longrightarrow \tau[\underline{G} \rightarrow^{split} G_1, \ldots, \underline{G} \rightarrow^{split} G_n]}$$

Figure 14: Closed SLD trees using depth-k abstraction and regular splitting

- every goal G_i contain at least one atom;
- every goal G_i contains at most one call to a non-regular predicate in S.

In general, there might be more than one regular splitting for a given goal. Here, we consider that $\mathsf{split}_S(G)$ returns any of them.[8]

The construction of closed SLD trees—using a maximum depth d and a set of non-regular predicates S—is formalized in Fig. 14.

As usual, we assume that the rules are tried by following a textual order and discarding the application of all the other rules as soon as a rule is applicable. Observe that the unfolding rule appears now before flattening and splitting. This is safe since the unfolding rule of Definition 30 does not allow infinite derivations. Therefore, in this calculus we might have longer "standard" SLD derivations, so that flattening and/or splitting is only applied when it is required to ensure termination.

[8]In the implemented tool, we just traverse G from left to right and start a new subgoal every time a call to a non-regular predicate in S is found.

On top of this algorithm for constructing closed SLD trees, we have implemented a test case generation procedure. Basically, given an initial goal G, it proceeds as follows:

- The closed tree is explored following a depth-first strategy.
- Every time a non-failing leaf G' is reached, if the path $G \rightarrow_\theta^+ G'$ includes no splitting steps, we output a test case $G\theta\gamma$, where γ is a substitution that grounds the *input* parameters of the predicate (according to some mode). The additional grounding provided by γ is required if we are interested in terminating test cases (assuming that those goals with ground input arguments always terminate, a reasonable assumption). Note that subsumption steps are not followed, i.e., goals that are reduced by subsumption are considered leaves too. This is sensible in our context since considering the rest of the derivation will not produce test cases covering additional clauses.
- When a splitting step is performed, say we have $G \rightarrow_\sigma^* G'$ and then $G' \rightarrow^{split} G'_1, \ldots G' \rightarrow^{split} G'_n$, we only return a test case $G\theta\gamma$ when we have computed substitutions $\sigma_1, \ldots, \sigma_n$ for $G'_1, \ldots, G'_n$, $\sigma_1 \Uparrow \cdots \Uparrow \sigma_n \neq \mathit{fail}$, and $\theta = \sigma \cdot (\sigma_1 \Uparrow \cdots \Uparrow \sigma_n)$, where γ is a grounding substitution as before. The different combinations are explored using Prolog's backtracking mechanism.

The implemented prototype tool is publicly available through a web interface from `http://kaz.dsic.upv.es/tcgen.html`. In the current version, the user should provide a logic program, an initial goal (typically with variable arguments in order to test all possibilities), the maximum term depth (in principle, a value between 1 and 3 should be fine in most examples), and the set of non-regular predicates. The grounding substitution is not computed by the tool, but should be trivial to produce: replacing all variables in input arguments by fresh constants will suffice.

To the best of our knowledge, previous approaches (e.g., [4, 17]) to test case generation based on partial evaluation, the closest to our approach, do not claim full coverage w.r.t. any coverage criteria. In contrast, our approach is a good starting point to ensure full coverage in the sense that—as a consequence of Theorem 19—the execution of the generated test cases will use all the clauses that are reachable in any successful execution of the program. Of course, we might produce test cases that do not produce a successful computation (e.g., because a non-leftmost unfolding is used in the construction of closed SLD trees) and it might also happen that some test cases are redundant or cover the same clauses more than once.

Nevertheless, our purpose in this section was to illustrate a promising application of closed SLD trees. Extending the tool for dealing with full Prolog, defining refined strategies that minimize the number of test cases while still ensuring full coverage, etc., is out of the scope of this paper, and is the subject of ongoing work.

6. Related Work

The operations of unfolding, flattening, splitting and subsumption have been already used—perhaps with a different formulation from that in this paper—in a number of program analysis and transformation techniques. We first present an overview of these operators in the literature:

- Unfolding is a fundamental operation based on applying SLD resolution [24]. Unfolding is also an essential component of many program transformation strategies and, particularly, of the fold/unfold framework (see [30] and references therein). Also, refined unfolding operators that take the history (i.e., the previously unfolded atoms in the SLD tree) into account in order to preserve termination can be found, e.g., in [12, 21]. We make no particular assumption on unfolding and, thus, refined unfolding operators can also be applied in our setting.

- Flattening is a well-known operation in functional logic programming [18], where it has been used to transform functional programs into Prolog programs by flattening nested function calls. Basically, for each conditional equation of the form $l = r \Leftarrow s_1 = t_1, \ldots, s_n = t_n$, if r contains nested function calls, e.g., $f(g(X), Y)$, we replace it with $f(Z, Y)$, where Z is a fresh variable, and then add the equation $g(X) = Z$ to the conditional part. Equations in the condition are flattened analogously. Similar flattening procedures have been used, e.g., in [6, 7, 33] to implement functional logic languages via SLD-resolution. Our flattening operation is the natural extension of this notion of flattening to logic programs.

- Subsumption, in our framework, amounts to considering that goal variants are computationally equivalent. As mentioned in the paper, this result is formalized, e.g., in Corollary 3.19 in [5], where the notion of similar SLD derivations is introduced. In partial evaluation, however, all instances of a goal are also considered subsumed [25]. In contrast, in our framework, a similar effect can be achieved by first applying flattening and then splitting and subsumption. We prefer to keep these operations independent so that the resulting framework is more flexible and customizable.

- Finally, goal splitting has been more recently introduced in [23, 14] in the context of so-called *conjunctive partial evaluation*. Our definition of splitting is very general but the refined algorithms in [14] for computing the best splitting (in the sense of minimizing the loss of information) would also be applicable in our setting.

On the other hand, the idea of constructing a finite (possibly approximate) search space of a goal is pervasive among the areas of program analysis and transformation. There are, however, a number of differences w.r.t. previous approaches. Let us consider in the following the closest ones.

Firstly, OLDT resolution, introduced by Tamaki and Sato [34], aims at defining a complete (and, for programs defining finite relations only, terminating) refinement to SLD. OLDT is based on tabulation, so that calls are stored in a table, together with the answers produced so far; in this way, repeated evaluation of the same atom may only involve a table lookup rather than applying SLD resolution once and again. Although we share some similarities with OLDT, there are also notable differences. On the one hand, OLDT only guarantees completeness and termination for finite-model programs [34, Theorem 4.3]; indeed, [34] presents no result regarding the finiteness of OLDT trees (despite the fact that a depth-k abstraction is part of their technique), while we have presented several strategies that guarantee the construction of complete and finite closed SLD trees. More importantly, in comparison to OLDT, our approach is *more declarative*, since we have a notion of closed SLD tree that is independent of a particular strategy; in contrast, the definition of OLDT resolution already includes a particular abstraction strategy (analogous to our depth-k abstraction). Furthermore, we clearly distinguish two separate stages: construction of a closed SLD tree, and extraction of computed answers. These two stages are combined in [34], so OLDT trees are in general more complex than our closed SLD trees.

Secondly, in contrast to many static analyses based on abstract interpretation, our approach implies no approximation at the level of computed answers, i.e., both an infinite SLD tree and an associated closed SLD tree still represent the same computed answers. In some sense, our closed SLD trees are in between the original source program and a standard SLD tree. The better the precision, the closer to the standard SLD tree. A related technique is the construction of justifications in tabled logic programming [31]. However, justifications aim at *explaining* successful/failing SLD derivations rather than representing the computed answer substitutions, as our closed SLD trees do. In particular, cyclic paths in the graph of a justification are not relevant for successful derivations, while cycles are essential in our closed SLD trees.

The fold/unfold framework (see, e.g., [30]) also shares some similarities with our work. On the one hand, this framework considers many elemental transformation rules, in particular including the ones that we have in our framework (or slight variations). Therefore, the fold/unfold framework is more general, and both our approach and standard partial evaluation can be seen as instances. However, this generality comes with a price: the fold/unfold framework usually requires some user interaction. Actually, our approach (as well as partial evaluation and many other related techniques) can be seen as particular strategies within the fold/unfold framework that are amenable to full automation.

The closest approach is of course that of *partial evaluation* of logic programs [25] and, particularly, a generalization known as *conjunctive partial evaluation* [23, 14]. Basically, conjunctive partial evaluation in logic programming aims at finding a sequence of goals $\mathcal{G} = \{G_1, \ldots, G_n\}$ and a sequence of associated finite (possibly partial) SLD trees $\tau_1, \ldots, \tau_n$ for the initial goals $G_1, \ldots, G_n$, such that every leaf in these trees is either successful, a failure, or only contains conjunctions that are instances of the conjunctions in $\mathcal{G}$; this is called the *closedness condition*, and guarantees the correctness of conjunctive partial evaluation. The partially evaluated program is obtained by producing a clause—called a *resultant*—of the form $G_i\theta \leftarrow G$ for each SLD derivation $G_i \leadsto_\theta G$ in τ_i, $i = 1, \ldots, n$. In contrast to the original definition of partial evaluation in logic programming (where renaming of resultants is not always required), every resultant $G_i\theta \leftarrow G$ is finally renamed to $ren_{\mathcal{G}}(G_i\theta) \leftarrow ren_{\mathcal{G}}(G)$ so that $ren_{\mathcal{G}}(G_i\theta)$ becomes an atom.

In practice, however, the process starts with a program P and an initial goal G. The key issue is then: how can we obtain a set of goals $\{G_1, \ldots, G_n\}$ and a set of finite SLD trees $\tau_1, \ldots, \tau_n$, so that G is an instance of some G_i, $i \in \{1, \ldots, n\}$, and the closedness condition holds? For this purpose, two operations are fundamental:

- *Splitting.* In general, the number of atoms in the goals of the SLD trees might keep growing and, thus, we need to split these goals into shorter conjunctions. By setting a bound on the number of atoms, we can avoid this source of non-termination.

- *Abstraction.* Even if the size of conjunctions is bounded, one can still have infinitely many conjunctions so that every new conjunction is not an instance of (i.e., closed with) any previous conjunction. Here, an abstraction operator (usually based on the notion of *most specific generalization*) is required. For instance, one can replace a conjunction by another one which is more general.

Given a program P and an initial goal G, typical algorithms usually proceeds as follows:

1. Initialization: $\mathcal{G}_0 = \{G\}$, $i = 1$.
2. Repeat
   (a) compute finite (possibly incomplete) SLD trees for the goals of $\mathcal{G}_i$;
   (b) for each leaf in these trees that is not closed: apply splitting and/or abstraction (if termination cannot be guaranteed) and add the results to $\mathcal{G}_i$ thus producing a new set $\mathcal{G}_{i+1}$.
3. Until $\mathcal{G}_i = \mathcal{G}_{i+1}$.

Stage 2(a) is called the *local level* and ensuring that SLD trees are finite is known as the *local termination* problem. For this purpose, one can use some well-founded or well-quasi ordering [12, 21]. This problem is not critical since stopping the construction of the SLD tree at any arbitrary point is still correct (though the quality of the specialization might vary). Stage 2(b) is called the *global level* and ensuring that the number of iterations is kept finite is known as the *global termination* problem. Here, some ordering is also used to detect a potential source of non-termination, as in the local level. This problem, however, is critical since the iterations cannot be simply stopped because the resulting partial evaluation would not be closed and, thus, it would be incorrect. Here, splitting and abstraction are necessary, as mentioned before.

The first stage of conjunctive partial evaluation, i.e., the construction of the (possibly incomplete) SLD trees so that the closedness condition holds, can be recast in our setting as follows:

- Given a program P and a goal G, we start by constructing a first SLD tree τ as in step 2(a) above.
- Then, for each leaf in τ which is closed w.r.t. G (i.e., which an instance of G), we apply flattening and splitting so that a variant of G is obtained.
- For each non-closed leaf, we apply the same splitting steps as in step 2(b). Abstraction steps, in turn, can be mimicked by flattening and splitting so that a more general goal is obtained.
- In this way, rather than a new set of goals, we produce an extended SLD tree whose frontier contains either leaves (a success, a failure or a variant of a previous goal) or new goals that should be further evaluated.

The process follows along the same lines starting now from the goals that require further evaluation.

Basically, rather than constructing a collection of independent SLD trees, in our approach we build a single (extended) SLD tree so that the relation between these trees (and the initial goal) is made explicit. Hence, our formulation is conceptually clearer since a single evaluation tree (basically an SLD tree) is built. Moreover, besides partial evaluation, other applications are possible (like, e.g., program comprehension, model checking or test case generation).

Finally, some of the ideas in this paper can also be found in our previous work [37, 26], where rewrite systems are considered instead. However, besides the paradigm shift, which is not trivial at all,[9] in this paper we also introduce a new, *parametric* notion of closed SLD tree (i.e., the notion of extended SLD tree that includes all possible closed SLD trees, which are then built using particular strategies); moreover, we also prove some new properties for our closed SLD trees and introduce novel specific strategies with different levels of abstraction for their construction.

7. Concluding Remarks

We have introduced in this paper the notion of a closed SLD tree, a particular extension of SLD trees that are always finite. These trees are built using the basic operations of unfolding, flattening, splitting, and subsumption. We have proved some basic properties for closed SLD trees, namely that the computed answers are preserved and, when the same computation rule is considered, the call patterns are also preserved (though possibly less instantiated). Moreover, an equational representation of the success set can be obtained from a closed SLD tree, which might be useful, e.g., for program comprehension. We have introduced some concrete strategies for building closed SLD trees and have shown its application through some examples. Finally, we have illustrated the usefulness of our approach by introducing a simple test case generator.

As for future work, we consider the improvement of the test case generator presented in Section 5. Another challenging topic for future work is the definition of an abstract-check-refine strategy similar to that commonly used in model checking techniques (see, e.g., [19]). In this context, one starts with a simple strategy (say, maximal abstraction) and, then, checks if some undesirable error happens in the closed SLD tree. If the tree is error-free,

[9] Actually, considering logic programs would correspond to dealing with conditional term rewrite systems, while [26] only considers unconditional ones.

the process stops successfully. Otherwise, one tries to check if the error is a real one or a false positive. In the second case, one should consider a more accurate, refined abstraction, and then the same process starts again. Here, the challenge is being able to construct the refined closed SLD tree incrementally rather than doing it from scratch. Compositionality is a nice property that may help in this task.

References

[1] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, D. Zanardini, Termination analysis of Java bytecode, in: G. Barthe, F. S. de Boer (eds.), Proc. of FMOODS'08, vol. 5051 of Lecture Notes in Computer Science, Springer, 2008, pp. 2–18.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode, in: Proc. of FMCO'07, Springer LNCS 5382, 2008, pp. 113–132.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Cost analysis of object-oriented bytecode programs, Theor. Comput. Sci. 413 (1) (2012) 142–159.

[4] E. Albert, M. Gómez-Zamalloa, J. M. Rojas, Resource-Driven CLP-Based Test Case Generation, in: G. Vidal (ed.), Proc. of the 21st International Logic-Based Program Synthesis and Transformation, LOPSTR 2011. Revised and Selected Papers, vol. 7225 of Lecture Notes in Computer Science, Springer, 2012, pp. 25–41.

[5] K. Apt, From Logic Programming to Prolog, Prentice Hall, 1997.

[6] R. Barbuti, M. Bellia, G. Levi, M. Martelli, LEAF: A language which integrates logic, equations and functions, in: Logic Programming: Functions, Relations, and Equations, 1986, pp. 201–238.

[7] P. Bosco, E. Giovannetti, C. Moiso, Narrowing vs. SLD-resolution, Theoretical Computer Science 59 (1988) 3–23.

[8] M. Brockschmidt, C. Otto, J. Giesl, Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting, in: Proc. of RTA 2011, vol. 10 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 155–170.

[9] M. Brockschmidt, C. Otto, C. von Essen, J. Giesl, Termination Graphs for Java Bytecode, in: S. Siegler, N. Wasser (eds.), Verification, Induction, Termination Analysis, vol. 6463 of Lecture Notes in Computer Science, Springer, 2010, pp. 17–37.

[10] M. Bruynooghe, A Practical Framework for the Abstract Interpretation of Logic Programs, J. Log. Program. 10 (2) (1991) 91–124.

[11] M. Bruynooghe, D. De Schreye, B. Martens, A General Criterion for Avoiding Infinite Unfolding during Partial Deduction of Logic Programs, in: V. Saraswat, K. Ueda (eds.), Proc. 1991 Int'l Symp. on Logic Programming, 1991, pp. 117–131.

[12] M. Bruynooghe, D. De Schreye, B. Martens, A General Criterion for Avoiding Infinite Unfolding, New Generation Computing 11 (1) (1992) 47–79.

[13] F. de Boer, J. Kok, C. Palamidessi, J. Rutten, From Failure to Success: Comparing a Denotational and a Declarative Semantics for Horn Clause Logic, Theor. Comput. Sci. 101 (2) (1992) 239–263.

[14] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, M. Sørensen, Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments, Journal of Logic Programming 41 (2&3) (1999) 231–277.

[15] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, R. Thiemann, Automated termination proofs for Haskell by term rewriting, ACM Trans. Program. Lang. Syst. 33 (2) (2011) 7.

[16] J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, C. Fuhs, Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs, in: PPDP'12, ACM, 2012, pp. 1–12.

[17] M. Gómez-Zamalloa, E. Albert, G. Puebla, Test case generation for object-oriented imperative languages in CLP, TPLP 10 (4-6) (2010) 659–674.

[18] M. Hanus, The integration of functions into logic programming: From theory to practice, Journal of Logic Programming 19&20 (1994) 583–628.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, G. Sutre, Lazy abstraction, in: Proc. of POPL, 2002, pp. 58–70.

[20] J. Kruskal, Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture, Transactions of the American Mathematical Society 95 (1960) 210–225.

[21] M. Leuschel, Homeomorphic Embedding for Online Termination of Symbolic Methods, in: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, Springer LNCS 2566, 2002, pp. 379–403.

[22] M. Leuschel, The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks, available at URL: `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html` (2007).

[23] M. Leuschel, D. De Schreye, A. de Waal, A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration, in: M. Maher (ed.), Proc. of JICSLP'96, The MIT Press, Cambridge, MA, 1996, pp. 319–332.

[24] J. Lloyd, Foundations of Logic Programming, Springer-Verlag, Berlin, 1987, second edition.

[25] J. Lloyd, J. Shepherdson, Partial Evaluation in Logic Programming, Journal of Logic Programming 11 (1991) 217–242.

[26] N. Nishida, G. Vidal, A Finite Representation of the Narrowing Space, in: G. Gupta, R. Peña (eds.), Proc. of the 23th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13), Springer LNCS 8901, 2014, pp. 54–71.

[27] C. Otto, M. Brockschmidt, C. von Essen, J. Giesl, Automated Termination Analysis of Java Bytecode by Term Rewriting, in: C. Lynch (ed.), Proc. of RTA 2010, vol. 6 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010, pp. 259–276.

[28] C. Palamidessi, Algebraic Properties of Idempotent Substitutions, in: M. Paterson (ed.), Proc. of 17th Int'l Colloquium on Automata, Languages and Programming, Springer LNCS 443, 1990, pp. 386–399.

[29] J. Peralta, J. Gallagher, Imperative Program Specialisation: An Approach Using CLP, in: A. Bossi (ed.), Proc. of the 9th International Workshop on Logic Programming Synthesis and Transformation (LOPSTR'99), Springer LNCS 1817, 2000, pp. 102–117.

[30] A. Pettorossi, M. Proietti, Transformation of Logic Programs: Foundations and Techniques, Journal of Logic Programming 19,20 (1994) 261–320.

[31] A. Roychoudhury, C. Ramakrishnan, I. Ramakrishnan, Justifying proofs using memo tables, in: PPDP 2000, ACM, 2000, pp. 178–189.

[32] P. Schneider-Kamp, J. Giesl, T. Ströder, A. Serebrenik, R. Thiemann, Automated termination analysis for logic programs with cut, TPLP 10 (4-6) (2010) 365–381.

[33] H. Tamaki, Semantics of a Logic Programming Language with a Reducibility Predicate, in: Proc. of First IEEE Int'l Symp. on Logic Programming, IEEE, New York, 1984, pp. 259–264.

[34] H. Tamaki, T. Sato, OLD Resolution with Tabulation, in: Proc. of the 3rd Int'l Conference on Logic Programming (ICLP'86), Springer LNCS 225, 1986, pp. 84–98.

[35] G. Vidal, A Hybrid Approach to Conjunctive Partial Evaluation of Logic Programs, in: M. Alpuente (ed.), Proc. of LOPSTR'10, vol. 6564 of Lecture Notes in Computer Science, Springer, 2011, pp. 200–214.

[36] G. Vidal, Annotation of logic programs for independent AND-parallelism by partial evaluation, TPLP 12 (4-5) (2012) 583–600.

[37] G. Vidal, Closed symbolic execution for verifying program termination, in: Proc. of the 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012), IEEE, 2012, pp. 34–43, available from URL `http://users.dsic.upv.es/~gvidal`.

[38] G. Vidal, Towards Erlang Verification by Term Rewriting, in: G. Gupta, R. Peña (eds.), Proc. of the 23th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13), Springer LNCS 8901, 2014, pp. 109–126.