ELSEVIER The Journal of Logic Programming 42 (2000) 1–46

THE JOURNAL OF LOGIC PROGRAMMING

www.elsevier.com/locate/jlpr

Operational and goal-independent denotational semantics for Prolog with cut

Fausto Spoto[*]

Dipartimento di Informatica, Università di Pisa, Corso Italia, 40, 56100 Pisa, Italy

Received 7 July 1998; received in revised form 10 November 1998; accepted 9 February 1999

Abstract

In this paper we propose an operational and a denotational semantics for Prolog. We deal with the control rules of Prolog and the cut operator. Our denotational semantics provides a goal-independent semantics. This means that the behaviour of a goal in a program is defined as the evaluation of the goal in the denotation (semantics) of the program. We show how our denotational semantics can be specialised into a computed answer semantics and into a call pattern semantics. Our work provides a basis for a precise abstract interpretation of Prolog programs. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Prolog; Logic programming; Abstract interpretation; Cut; Divergence

1. Introduction

Prolog is a well-known programming language which implements the logic programming paradigm. However, for historical as well as for efficiency issues, Prolog is *not* logic programming. For instance, the depth-first search strategy of Prolog entails that the SLD tree is not always fully explored by the Prolog interpreter. Similarly, the presence of many meta-logical predicates like cut, set operations and database operations breaks the declarative paradigm of logic programming. Consider for instance the following Prolog program:

```
min([H|T],M):-min(T,M),M <= H,!.
min([H|T],H).
```

This program computes the minimum element of a list of integers. As it is easy to check, the second clause is correct only because a cut is contained in the first one.

[*] Corresponding author. Tel.: +39-050-887248; fax: +39-050-887226.
E-mail address: spoto@di.unipi.it (F. Spoto)

This means that if we do not take the cut into account we would conclude that `min([2,1],2)` belongs to the success set of the program, while it is not true.

Note that meta-logical features like the cut can even affect the *abstract* behaviour of a program. Consider for instance the following program:

```
select_vars_in_term(X,[X]):- var(X),!.
select_vars_in_term(A,[]):- atom(A),!.
select_vars_in_term(F,L):- F=..[_Name|Args],
  select_vars_in_list(Args,L).

select_vars_in_list([],[]).
select_vars_in_list([H|T],A):-
  select_vars_in_term(H,H1), select_vars_in_list(T,T1),
  append(H1,T1,A).
```

If we take into account the control information, it is easy to conclude that in the third clause `F` can never be a free variable nor an atom. This allows us to provide an optimised code which checks only whether `F` is a number.

The contribution of this paper is to provide a general semantical framework that can be instantiated as a denotational semantics able to model the fact that `min([2,1],2)` does *not* belong to the success set of the first program and that `F` can never be a free variable nor an atom in the third clause of the second program. The general framework can precisely model several operational aspects of Prolog in a denotational setting. It is built upon a very *concrete* denotational semantics that can be specialised into a more *abstract* one as soon as we specify the observable property of interest. In this paper, we specifically consider the observable properties of computed answers and call patterns. In a companion paper [23], we show how to abstract the general framework into a semantics that models the abstract behaviour of the observable property. For instance, we could be interested in the groundness analysis of computed answers, or in the type analysis of call patterns. A feature of our semantics is that it is goal-independent, i.e., it provides a denotation for a program. Every query can be evaluated in such a denotation. This feature allows a goal-independent analysis of logic programs. For instance, to compute groundness or sharing information at some program points, we need this information to be correct independently from the query.

The paper is organised as follows. Section 4 introduces an operational semantics for logic programming with control rules of Prolog and the cut operator. This semantics constructs the portion of the SLD tree which is actually visited by an interpreter. It admits a top-down definition only. In Section 5 we introduce a tree semantics which deals with SLD trees with parametric control information. This way, we model in a compositional (*denotational*) way the operational behaviour of programs. We show that this semantics admits equivalent top-down, bottom-up, goal-dependent and goal-independent formulations. In Section 6 we show that each formulation of the tree semantics can be abstracted into the operational semantics of Section 4. Section 7 describes an alternative presentation of the tree semantics of Section 5, where control information is *compiled* rather than *declared*, though still in a parametric way. This transformation can be seen as the first, loss-free step toward the definition of observable specific abstract denotational semantics. Actually, Sections 8 and 9 show two abstractions of the semantics of Section 7, the first into a computed answer denotational semantics and the second into a call pattern denotational semantics. These two semantics admit a bottom-up formulation only.

All proofs not contained in this paper can be found in Ref. [22].

2. Related works

There exist many formalisations for subsets of Prolog [1,3,9,24,10,11,16,17,21], and even a formalisation for full Prolog [2]. We must therefore justify why a new semantics for a subset of Prolog (more precisely, Prolog with the cut but without database and set operations) is needed. A common weakness of previous approaches (except Refs. [1,2]), when used for abstract interpretation, is that they do not provide a general framework for the abstract analysis of generic properties of logic programs. Namely, they consider computed answers as the unique observable property of interest. How to generalise their approaches to call patterns, resultants or even SLD refutations is not clear.

The only paper that defines an operational semantics is Ref. [2]. The main problem with operational semantics is that it is goal-dependent. The operational semantics is then no longer adequate as a basis for goal-independent analysis. If we are concerned with goal-independent global analysis, a denotational semantics is definitely better than an operational one. This is because a denotational semantics is able to model the behaviour of a program independently from the query. With the denotation, we can evaluate all possible query patterns and collect the information about the behaviour of the program for those patterns. Since the evaluation of a query in the denotation of a program is computationally inexpensive, when compared to the abstract execution of the query in the program, this approach is convenient when we have to deal with many query patterns. The usual argument against denotational semantics is that it is not very good in dealing with all the operational subtleties. This is not a real issue in global analysis since this deals with abstraction, so that normally there is a safe way of disregarding those subtleties without losing correctness and precision.

Denotational semantics for various subsets of Prolog are defined in Refs. [9,10,16]. However, these semantics are not adequate for abstraction, since they use functions as denotations for predicates. It is not clear how such functions can be abstracted. Moreover, the choice of functions as denotation for predicates prevents, from our point of view, any easy way for using these semantics as the basis for abstract interpretation. This is because abstract interpretation must provide an *effective* algorithm for the analysis of the abstract property. This result is obtained with our semantics since we use syntactical objects (substitutions with some control decoration) as denotational domains rather than functions.

The only denotational semantics developed for abstract interpretation is presented in Ref. [17]. The problem with this is that it follows a goal-dependent approach. Indeed, the semantics for Prolog is based on sequences of substitutions, decorated with some information about cut and divergence. For instance, if a sequence ends with a cut mark, then this means that a cut has been executed and the following substitutions can no longer be observed. This is obtained through the use of a concatenation operator on sequences that drops the second sequence if the first ends with a cut mark. Using this approach, it is not possible to compute the denotation just for the more general goals.

In Ref. [21], a totally different approach to the semantics of Prolog is described. Here it is shown how Prolog can be compiled into Milner's CCS [20]. Properties of the original Prolog program are then related to properties of the resulting CCS program. It is not clear how to abstract the resulting CCS program in such a way that the abstracted program could provide some information about the original Prolog program.

An interesting and recent approach is developed in Ref. [24]. The authors define in a metric fashion a general operational and a general denotational semantics which model the behaviour of OR-parallel logic programs with a commit operator which behaves as a cut operator if the code is executed in a sequential way. Their proposal uses binary strings as history and cutpoints. This makes the construction a little complex. For instance, our denotational semantics for computed answers (Section 8) does not use histories and cutpoints. This is because we use the technique of *control compilation*. Intuitively, control compilation means that every computed answer substitution is endowed with an observability condition that says when exactly that substitution is observable. Control compilation was used for the first time in Ref. [1], where a semantics for logic programs with Prolog control was obtained by *compiling* a Prolog program into an ask/tell language, so that the semantics of the Prolog program can be viewed as the semantics of a concurrent logic program. Our approach generalises this through the use of *observability* constraints. Moreover, we show how the cut operator can be easily handled in this context.

Our semantics can be considered as the natural evolution of a series of previous proposals. The first is Ref. [11], where divergence is modelled in a fixpoint framework. A more adequate semantics for modelling divergence is presented in Ref. [3] as an abstraction of a more concrete semantics given in terms of resultants. A resultant H :- B, where B is a non-empty set of goals, is abstracted into a *divergent atom* $\widetilde{H}$, representing a computation which is still in progress and which can possibly remove all the following atoms observed in the computation. Note that the semantic domain consists of sequences rather than sets of atoms, so as to model the relation between atoms implied by the backtracking semantics of Prolog. A further development of the semantics in Ref. [3] can be found in Refs. [18,19], where the s-semantics [4] is extended to deal with the cut operator. The semantic domain is very complex, since any constraint is associated with a history and a cutpoint. In the present work we show how this information can safely be discarded.

3. Preliminaries

We assume the reader familiar with basic algebraic structures [8]. A sequence is an ordered collection of elements with repetitions. We will write $\mathsf{Seq}(\mathbb{E})$ for the set of (possibly empty) sequences of elements of $\mathbb{E}$. We will write $\mathsf{Seq}^+(\mathbb{E})$ for the set of non-empty sequences of elements of $\mathbb{E}$. $::$ denotes sequence concatenation. Non-empty sequences are denoted by variables with a tilde sign on them. For instance, $\tilde{s}$ represents a *non*-empty sequence. The empty sequence is explicitly written as ε. If $\tilde{s}$ is a sequence then $\#\tilde{s}$ is the length of the sequence. Note that we consider $\mathbb{E} \subseteq \mathsf{Seq}(\mathbb{E})$, i.e., a single element is a sequence of length one.

Abstract interpretation [6,7] is a formal technique for relating semantics at different level of abstractions. One way of formalising abstract interpretation is by means of Galois connections:

Definition 1. Given two posets $\langle P, \sqsubseteq\rangle$ and $\langle P^a, \sqsubseteq^a\rangle$, a Galois connection between them is a pair $\langle \alpha, \gamma\rangle$ of total maps such that the following condition holds: for all $p \in P$ and $p^a \in P^a$ we have

$$\alpha(p) \sqsubseteq^a p^a \quad \text{if and only if} \quad p \sqsubseteq \gamma(p^a).$$

α and γ are the abstraction and the concretization maps of the connection.

We know from a theorem in Ref. [7] that, given two complete lattices $\langle P, \sqsubseteq\rangle$ and $\langle P^a, \sqsubseteq^a\rangle$, if $\langle \alpha, \gamma\rangle$ is a Galois connection between them, $\phi : P \mapsto P$ and $\phi^a : P^a \mapsto P^a$ are two monotonic operators and $\alpha(\bot) = \bot^a$, then the local correctness condition implies the global one, i.e.:

- $\alpha \circ \phi \sqsubseteq^a \phi^a \circ \alpha$ implies $\alpha(\mathsf{lfp}(\phi)) \sqsubseteq^a \mathsf{lfp}(\phi^a)$,
- $\alpha \circ \phi = \phi^a \circ \alpha$ implies $\alpha(\mathsf{lfp}(\phi)) = \mathsf{lfp}(\phi^a)$.

Given $\phi : P \mapsto P$ and a Galois connection $\langle \alpha, \gamma\rangle$ between P and P^a, the best approximation of ϕ on P^a is given by $\phi^a = \lambda x.\alpha(\phi(\gamma(x)))$.

If α is a continuous map between $\langle P, \sqsubseteq\rangle$ and $\langle P^a, \sqsubseteq^a\rangle$, then α induces a map γ such that $\langle \alpha, \gamma\rangle$ is a Galois connection between $\langle P, \sqsubseteq\rangle$ and $\langle P^a, \sqsubseteq^a\rangle$. This means that γ need not be explicitly specified, once α is given.

We recall that two posets can be extended to complete lattices in such a way that any monotonic map α between them can be extended to a continuous map $\widetilde{\alpha}$ between their extensions. The same extension leads to the following result which will allow us to get simpler proofs:

Proposition 2. *Let $\langle F, \sqsubseteq\rangle$ and $\langle F^a, \sqsubseteq^a\rangle$ be two posets and $T : F \mapsto F$, $T^a : F^a \mapsto F^a$ and $\alpha : F \mapsto F^a$ be three monotonic maps such that $\alpha(\bot) = \bot^a$ and $\alpha \circ T = T^a \circ \alpha$. Then we can extend F and F^a to two complete lattices $\widetilde{F}$ and $\widetilde{F^a}$, respectively, and α, T and T^a to continuous maps $\widetilde{\alpha} : \widetilde{F} \mapsto \widetilde{F^a}$, $\widetilde{T} : \widetilde{F} \mapsto \widetilde{F}$ and $\widetilde{T^a} : \widetilde{F^a} \mapsto \widetilde{F^a}$ such that $\langle \widetilde{\alpha}, \gamma\rangle$, for a suitable γ, is a Galois connection between $\widetilde{F}$ and $\widetilde{F^a}$, $\widetilde{\alpha}(\bot) = \bot^a$ and the correctness condition $\widetilde{\alpha} \circ \widetilde{T} = \widetilde{T^a} \circ \widetilde{\alpha}$ holds.*

The relevance of the above proposition is that we do not need to be concerned with the infinite elements of the two lattices, neither do we have to define the semantic operators and the abstraction map on them. Roughly speaking, F consists of the finite elements of $\widetilde{F}$ and F^a consists of the finite elements of $\widetilde{F^a}$.

3.1. Prolog

In the following we will use an *abstract* syntax for Prolog programs, which simplifies the semantic operators. Moreover, this allows us to look at Prolog as an instance of the general CLP scheme [15]. The translation from Prolog into our syntax is straightforward and can be understood by noting that the Prolog clause `q(X):-p(X),!,s(X).` is translated into `q`(x) `:- cut (p`(x)`) and s`(x). Finally, our abstract syntax assumes all predicates to be unary. This constraint simplifies the definition of the semantics without loss of generality. The extension of that definition to the general case is anyway straightforward. A clause has the form $\mathtt{p}(x)$:- G_1 or $\cdots$ or G_n., with $n \geqslant 1$, where $G_1, \dots, G_n$ are goals, defined by the grammar:

$$G ::= c|\mathtt{p}(x)|G \texttt{ and } G|\texttt{exists } x.G|\texttt{cut}(G)|\texttt{cut}_d(G),$$

where c is a constraint, x is a program variable and d is a non-negative integer. The construct $\texttt{cut}_d()$ embeds an explicit cut point mark d. The expression $\mathtt{p}(x)$, where p is a predicate symbol, is called *procedure call.*

We require that if G_1 contains a procedure call then, in the construct G_1 and G_2, G_2 is *cut-free*, i.e., it does not contain $\texttt{cut}()$ or $\texttt{cut}_d()$ constructs. The set of goals is denoted by $\mathbb{G}$. Note that when we transform a Prolog program in our abstract syntax we do not need any $\texttt{cut}_d(G)$ constructs. However, we need $\texttt{cut}(G)$ constructs, called *open cuts*, for expressing the scope of a cut, and exists $x.G$ constructs, for expressing existential variables.

The constraint c is taken from the constraint domain over which the CLP language is defined. This constraint domain is assumed to fulfill the following definition [14]:

Definition 3. A basic constraint is an element of a lattice $\langle \mathscr{B}, \leqslant, \vee, \wedge, \mathit{true}, \mathit{false} \rangle$, where $\vee$ is the least upper bound operator, $\wedge$ is the greatest lower bound operator, *true* is the top of the lattice and *false* is the bottom of the lattice. We assume $\mathscr{B}$ contains the element $\delta_{x,y}$, for each pair of variables x and y. For instance, $\delta_{x,y}$ represents the constraint identifying the variables x and y. Moreover, we assume there is a family of monotonic operators $\exists_x$ on the set of constraints, representing the restriction of a constraint obtained by hiding all the information related to the variable x.

A goal G is called divergent if and only if it contains a procedure call. The notation $\mathsf{div}(G)$ means that G is a divergent goal. Its formal definition can be given by straightforward induction on the structure of G. A goal which is not divergent is said to be convergent.

A goal represents a state in the refutation procedure. If we use a leftmost selection rule, we can recover the partial answer of a goal as the conjunction of the constraints which precede the first procedure call. Formally, the partial answer of a goal is given by the function con which is defined as

$$\begin{aligned}
\mathsf{con}(c) &= c\\
\mathsf{con}(\mathtt{p}(x)) &= \mathit{true}\\
\mathsf{con}(\texttt{exists } x.G) &= \exists_x \mathsf{con}(G)\\
\mathsf{con}(\texttt{cut}(G)) &= \mathsf{con}(\texttt{cut}_d(G)) = \mathsf{con}(G)\\
\mathsf{con}(G_1 \texttt{ and } G_2) &= \begin{cases} \mathsf{con}(G_1) & \text{if } \mathsf{div}(G_1)\\ \mathsf{con}(G_1) \wedge \mathsf{con}(G_2) & \text{if not } \mathsf{div}(G_1). \end{cases}
\end{aligned}$$

4. Operational semantics

In this section we define an operational semantics for the subset of Prolog we are taking into account. This semantics will be a goal-dependent semantics which mimics the behaviour of a Prolog interpreter. Namely, it constructs the sequence of trees, where the ith tree is the snapshot of the portion of SLD tree already visited by a Prolog interpreter after i steps of resolution.

We define our semantical domain of trees. We require that if a node does not contain procedure calls, then it has no children.

Definition 4. A tree is an element of the set

$$\mathbb{T} = \{(G,\varepsilon) \mid G \in \mathbb{G}\} \cup \{(G,\tilde{t}) \mid G \in \mathbb{G}, \mathsf{div}(G) \text{ and } \tilde{t} \in \mathsf{Seq}(\mathbb{T})\}.$$

We will make induction on trees on the basis of their structure. Moreover, we will make induction on sequences of trees using the following well-founded complete ordering relation: $\tilde{t}'$ precedes $\tilde{t}''$ if $\tilde{t}'' = \tilde{t}' :: \tilde{t}$ or $\tilde{t}'' = \tilde{t} :: \tilde{t}'$ for a suitable sequence $\tilde{t}$. (G,ε) precedes $(G,\tilde{t})$ and $(G,\tilde{t}_1)$ precedes $(G,\tilde{t}_2)$ if and only if $\tilde{t}_1$ precedes $\tilde{t}_2$.

Trees are also ordered with respect to inclusion, which is a complete ordering relation:

$$\begin{aligned}
&(G,\varepsilon) \subseteq (G,\varepsilon)\\
&(G,\varepsilon) \subseteq (G,\tilde{t})\\
&(G,\tilde{t}) \subseteq (G,\tilde{t}') \quad \text{if } \tilde{t} \subseteq \tilde{t}'\\
&\tilde{t}_1 :: \tilde{t}_2 \subseteq \tilde{t}'_1 :: \tilde{t}'_2 \quad \text{if } \tilde{t}_1 \subseteq \tilde{t}'_1 \text{ and } \tilde{t}_2 \subseteq \tilde{t}'_2.
\end{aligned}$$

Note that if $\tilde{t} \subseteq \tilde{t}'$ then $\tilde{t}$ and $\tilde{t}'$ must be formed by the same number of trees. This condition can be relaxed, leading to the following complete ordering relation:

$$\begin{aligned}
(G,\varepsilon) &\subseteq_l (G,\varepsilon)\\
(G,\varepsilon) &\subseteq_l (G,\tilde{t})\\
(G,\tilde{t}) &\subseteq_l (G,\tilde{t}') \quad \text{if } \tilde{t} \subseteq_l \tilde{t}'\\
t_1 :: \cdots :: t_n &\subseteq_l t'_1 :: \cdots :: t'_n :: \tilde{t} \quad \text{if } t_i \subseteq_l t'_i \quad \text{for } i = 1,\ldots,n.
\end{aligned}$$

Note that $\tilde{t} \subseteq \tilde{t}'$ implies $\tilde{t} \subseteq_l \tilde{t}'$. $(G,\tilde{t})$ is called a tree *for* G. A tree is called *i-cut-closed* if and only if every $\mathtt{cut}_d()$ construct is at height at least $d - i$ from the root. We assume that the root has height 0, its children have height 1 and so on. Roughly speaking, an i-cut-closed tree is a tree such that, when attached to the leaves of another tree, its cuts can affect only the last i levels of that tree. The cuts of a 0-cut-closed tree cannot affect that tree at all. A tree is called *cut-closed* when it is 0-cut-closed.

Fig. 1 shows a tree for $\mathtt{p}(x)$. Note that all the goals which are not leaves contain a procedure call. Assume a goal contains a $\mathtt{cut}_d(G)$ construct. Consider the portion of the tree formed by the nodes which are on the right of the goal containing $\mathtt{cut}_d(G)$ along a path toward the root which has exactly a d length. Such a portion of the tree is the scope of the cut. For instance, in Fig. 1 the goal $\mathtt{cut}_1(x = 2)$ has the goal $x = 3$ in its scope, as shown by the curly line. A goal of the form $\mathtt{cut}(G)$ can be seen as an abridged form of $\mathtt{cut}_\infty(G)$. Hence all the goals which are on the right of the path which connects the goal to the root of the tree are in the scope of the construct.

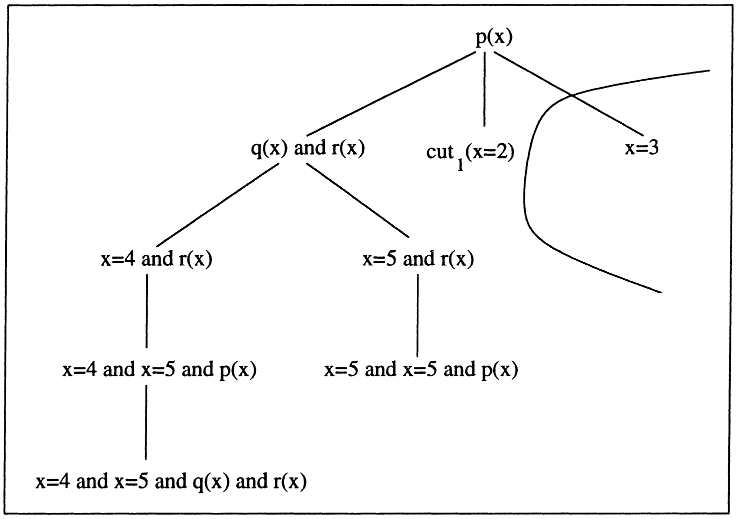

Fig. 1. A tree for p(x).

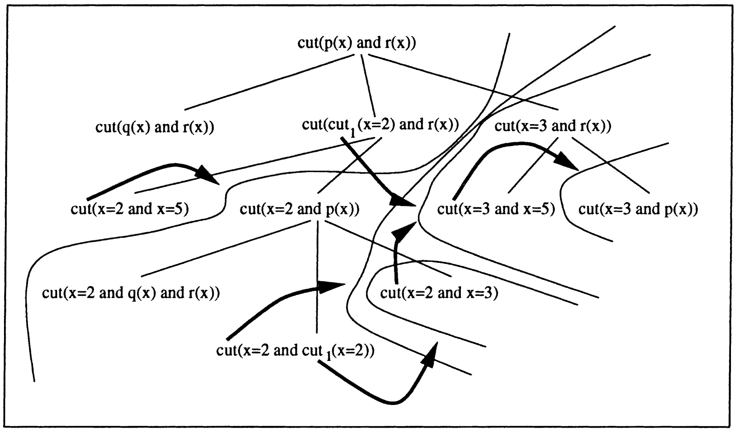

Fig. 2. A tree for cut (p(x) and r(x)).

For instance, consider Fig. 2. Every cut(G) or cut$_d$(G) with not div(G) is given as scope the portion of the tree related to it by an arrow.

Given a program P, we say that a tree T is P-directed if and only if the children of every node are related to their parent by the procedure definitions of P. To formalise this definition, we define two auxiliary functions. choices(G, P) is the number of alternatives for the leftmost execution of the goal G in the program P. $\imath(G, P, i)$ is the goal obtained from G by substituting the leftmost procedure call of G with its ith definition in P. Note that we define $\imath(G, P, i)$ for the case div(G) only. In order to give a formal definition of these functions, we need an auxiliary map τ which removes all cut() and cut$_d$() constructs from a goal.

Definition 5. We define the following maps:

$$\begin{aligned}
\tau(c) &= c\\
\tau(\mathtt{p}(x)) &= \mathtt{p}(x)\\
\tau(G_1 \ \mathtt{and}\ G_2) &= \tau(G_1)\ \mathtt{and}\ \tau(G_2)\\
\tau(\mathtt{exists}\ x.G) &= \mathtt{exists}\ x.\tau(G)\\
\tau(\mathtt{cut}_d(G)) &= \tau(\mathtt{cut}(G)) = \tau(G).
\end{aligned}$$

$$\begin{aligned}
\mathsf{choices}(c,P) &= 0\\
\mathsf{choices}(G_1\ \mathtt{and}\ G_2,P) &= \begin{cases}\mathsf{choices}(G_2,P) & \text{if not } \mathsf{div}(G_1)\\ \mathsf{choices}(G_1,P) & \text{if } \mathsf{div}(G_1)\end{cases}\\
\mathsf{choices}(\mathtt{exists}\ x.G,P) &= \mathsf{choices}(G,P)\\
\mathsf{choices}(\mathtt{cut}_d(G),P) &= \mathsf{choices}(\mathtt{cut}(G),P) = \mathsf{choices}(G,P)\\
\mathsf{choices}(\mathtt{p}(x),P) &= n,
\end{aligned}$$

where $\mathtt{p}(y)$:- G_1 `or` $\cdots$ `or` G_n is the definition of p in the program P.

$$\begin{aligned}
\imath(G_1\ \mathtt{and}\ G_2,P,i) &= \begin{cases}\tau(G_1)\ \mathtt{and}\ \imath(G_2,P,i) & \text{if not } \mathtt{div}(G_1)\\ \imath(G_1,P,i)\ \mathtt{and}\ G_2 & \text{if } \mathsf{div}(G_1)\end{cases}\\
\imath(\mathtt{exists}\ x.G,P,i) &= \mathtt{exists}\ x.\imath(G,P,i)\\
\imath(\mathtt{cut}_d(G),P,i) &= \mathtt{cut}_{d+1}(\imath(G,P,i))\\
\imath(\mathtt{cut}(G),P,i) &= \mathtt{cut}(\imath(G,P,i))\\
\imath(\mathtt{p}(x),P,i) &= \begin{cases}G_i & \text{if } x=y\\ \mathtt{exists}\ y.(\delta_{y,x}\ \mathtt{and}\ G_i) & \text{if } x\neq y,\end{cases}
\end{aligned}$$

where $\mathtt{p}(y)$:- G_1 `or` ... `or` G_n. is the definition of `p` in the program P where all `cut()` constructs have been replaced by $\mathtt{cut}_1()$ constructs.

Note that in the definition of $\imath(\mathtt{p}(x),P,i)$ we do not require an explicit renaming. Instead, we use an approach based on cylindrification. We introduce the binding $\delta_{y,x}$, that equates the variables x and y. Then we remove any reference to y, by considering it as an existential variable. This approach cannot lead to any name clash between x and an existential variable of the same name contained in the definition of `p`, since the scope of such an existential variable would be limited by an `exists` x. construct.

Definition 6. Given a program P, we say that (G,ε) is P-directed for every goal G, while $(G,t_1::\cdots::t_n)$ is P-directed if and only if $n\leqslant \mathsf{choices}(G,P)$, t_i is P-directed for $i=1,\ldots,n$ and, letting G_i be the root of t_i, $G_i=\imath(G,P,i)$ for every $i=1,\ldots,n$. Strongly P-directed trees are defined in the same way, substituting the condition $n\leqslant\mathsf{choices}(G,P)$ with the condition $n=\mathsf{choices}(G,P)$.

Example 7. The tree shown in Fig. 1 is P-directed for the following program P:

$\mathtt{p}(x)$:- $(\mathtt{q}(x)$ `and` $\mathtt{r}(x))$ `or` $\mathtt{cut}(x=2)$ `or` $x=3$.
$\mathtt{q}(x)$:- $x=4$ `or` $x=5$.
$\mathtt{r}(x)$:- $x=5$ `and` $\mathtt{p}(x)$.

Note that the same tree is not strongly P-directed for the same program. Actually, the goal $x = 4$ and $x = 5$ and $\mathtt{p}(x)$ is expanded only through its first child, while the others' alternatives for $\mathtt{p}(x)$ are not tried. On the contrary, the tree shown in Fig. 2 is strongly P-directed for the program P above. This is because at every choice point either all alternatives or none are tried.

By definition of ι, if $(\mathtt{p}(x), t_1 :: \cdots :: t_n)$ is a strongly P-directed tree for $\mathtt{p}(x)$ and $\mathtt{p}(y)$:- G_1 or ... or G_n. is the definition of $\mathtt{p}$ in the program P where all cut() constructs have been replaced by $\mathtt{cut}_1()$ constructs, then t_i is a strongly P-directed tree for G_i, if $x = y$, or it is a strongly P-directed tree for exists $y.(\delta_{y,x}$ and $G_i)$, if $x \neq y$.

Let us come back to the definition of our operational semantics on trees. Given a program P, we define a function

$$\mathsf{expand}_P : \mathbb{T} \mapsto \mathbb{T}$$

such that $\mathsf{expand}_P(T)$ is the tree obtained by making an SLD resolution step from T, using the leftmost selection rule and the depth first search rule of Prolog and taking the cut operator into account. We assume that the tree T does not contain open cuts, but only constructs $\mathtt{cut}_d()$ with an explicit scope information d. This is because open cuts are useful only in a compositional or denotational approach, as we will see later. We will use the following definition $\mathsf{expandable}_P(T) = (\mathsf{expand}_P(T) \neq T)$.

We first define a map which selects the set of cut conditions in a goal. It will be used in the definition of expand_P in order to check whether a cut has been executed in a tree and makes the following nodes of the tree, in a depth-first ordering, non-observable. This map, which we call cuts, selects all cut conditions which are convergent and are not preceded by a procedure call. This means that these cuts have been actually executed in a left to right execution of the goal. If the cut which has been executed has an explicit, positive scope information (i.e., it has the form $\mathtt{cut}_d(G)$ with $d \geqslant 1$), then this scope information is extracted by the cuts map and a pair ⟨condition/scope⟩ is built. Otherwise, only the cut condition is remembered. Hence the range of the cuts map is a set $\mathscr{B} \cup (\mathscr{B} \times \mathbb{N})$ of constraints and of pairs ⟨constraint/positive integer⟩. Note that the case of an open cut of the form $\mathtt{cut}(G)$ cannot arise in this operational semantics, but will arise in the tree semantics which we will define in Section 5.

Definition 8. On $\mathscr{B} \cup (\mathscr{B} \times \mathbb{N})$ we define the following operations:

$$\begin{aligned} c \bullet \{\langle c_1, d_1\rangle, \ldots, \langle c_n, d_n\rangle, c'_1, \ldots, c'_m\} \\ = \{\langle c \wedge c_1, d_1\rangle, \ldots, \langle c \wedge c_n, d_n\rangle, c \wedge c'_1, \ldots, c \wedge c'_m\} \end{aligned} \tag{1}$$

$$\exists\!\!\exists_x \{\langle c_1, d_1\rangle, \ldots, \langle c_n, d_n\rangle, c'_1, \ldots, c'_m\} = \{\langle \exists_x c_1, d_1\rangle, \ldots, \langle \exists_x c_n, d_n\rangle, \exists_x c'_1, \ldots, \exists_x c'_m\} \tag{2}$$

$$\bigsqcup \{\langle c_1, d_1\rangle, \ldots, \langle c_n, d_n\rangle, c'_1, \ldots, c'_m\} = (\vee_i c_i) \vee (\vee_i c'_i). \tag{3}$$

The formal definition of cuts can now be given:

Definition 9. Given a goal G, the set of its cut conditions is defined as

$$\mathsf{cuts}(c) = \mathsf{cuts}(\mathtt{p}(x)) = \emptyset$$

$$\mathsf{cuts}(G_1 \ \mathtt{and}\ G_2) = \begin{cases} \mathsf{cuts}(G_1) \cup (\mathsf{con}(G_1) \bullet \mathsf{cuts}(G_2)) & \text{if not } \mathsf{div}(G_1) \\ \mathsf{cuts}(G_1) & \text{if } \mathsf{div}\ (G_1) \end{cases}$$

$$\mathsf{cuts}(\mathtt{exists}\ x.G) = \exists\!\!\!\exists_x \mathsf{cuts}(G)$$

$$\mathsf{cuts}(\mathtt{cut}_d(G)) = \begin{cases} \mathsf{cuts}(G) \cup \{\langle \mathsf{con}(G), d\rangle\} & \text{if not } \mathsf{div}(G) \text{ and } d > 0 \\ \mathsf{cuts}(G) & \text{otherwise} \end{cases}$$

$$\mathsf{cuts}(\mathtt{cut}(G)) = \begin{cases} \mathsf{cuts}(G) \cup \{\mathsf{con}(G)\} & \text{if not } \mathsf{div}(G) \\ \mathsf{cuts}(G) & \text{if } \mathsf{div}(G). \end{cases}$$

We extend the map cuts from goals to sequences of trees. Roughly speaking, we should make the union of the function cuts applied to all the goals of the tree. However, if a cut with an explicit scope information is such that its scope is limited and cannot go beyond the tree, we must not consider this cut as a cut condition, because it is not visible outside the tree. Hence, to define the set of cut conditions of a tree, we first compute the set of the cut conditions of the root. Then we make the union of this set with the set of cut conditions of the subtrees, provided that the scope information has been decreased by a unity and is still positive. This way, cut conditions with a scope information are kept only if their scope goes beyond the tree. This is achieved through a *shaking* function ς:

Definition 10. On $\mathscr{B} \cup (\mathscr{B} \times \mathbb{N})$ we define the *shaking* function

$$\varsigma(\{\langle c_1, d_1\rangle, \ldots, \langle c_n, d_n\rangle, c'_1, \ldots, c'_m\}) = \{c'_1, \ldots, c'_m\} \cup \bigcup_{d_i > 1} \{\langle c_i, d_i - 1\rangle\}. \tag{4}$$

The map cuts is extended to trees as

$$\mathsf{cuts}(G, \varepsilon) = \mathsf{cuts}(G)$$

$$\mathsf{cuts}(G, \tilde{t}) = \mathsf{cuts}(G) \cup \varsigma \mathsf{cuts}(\tilde{t})$$

$$\mathsf{cuts}(t_1 :: \cdots :: t_n) = \bigcup_i \mathsf{cuts}(t_i).$$

Example 11. We have $\mathsf{cuts}(\mathtt{exists}\ x.\mathtt{cut}_2(y = \mathtt{f}(x,z))) = \{\langle \exists_x (y = \mathtt{f}(x,z)), 2\rangle\}$ while $\mathsf{cuts}(\mathtt{exists}\ x.\mathtt{cut}_2(\mathtt{p}(y))) = \emptyset$, since $\mathtt{p}(y)$ is divergent.

Moreover, consider the tree T of Fig. 1. We have $\mathsf{cuts}(T) = \emptyset$, since the scope of the $\mathtt{cut}_1(x = 2)$ construct cannot go beyond the root of the tree. However, if we had $\mathtt{cut}_2(x = 2)$ instead of $\mathtt{cut}_1(x = 2)$, we would have $\mathsf{cuts}(T) = \{\langle x = 2, 1\rangle\}$. If we had $\mathtt{cut}(x = 2)$ instead of $\mathtt{cut}_1(\mathtt{x} = 2)$, we would have $\mathsf{cuts}(T) = \{x = 2\}$.

If T is the SLD tree already visited by a Prolog interpreter, $\bigsqcup \mathsf{cuts}(T) \neq \mathit{false}$ if and only if a cut has been executed in the construction of T, avoiding possible brothers of T to be visited. With this intuition in mind, we can define the transition map expand_P for a program P as follows:

Definition 12. Given a program P and a tree T, $\mathsf{expand}_P(T)$ is defined as

$$\mathsf{expand}_P(G,\varepsilon) = \begin{cases} (G,\varepsilon) & \text{if } \mathsf{con}(G) = \mathit{false} \text{ or not } \mathsf{div}(G) \\ (G,(\imath(G,P,1),\varepsilon)) & \text{otherwise} \end{cases}$$

$$\mathsf{expand}_P(G,t_1 :: \cdots :: t_n) = \begin{cases} (G,t_1 :: \cdots :: t_{n-1} :: \mathsf{expand}_P(t_n)) \\ \quad \text{if } \mathsf{expandable}_P(t_n) \\ \\ (G,t_1 :: \cdots :: t_n :: (\imath(G,P,n+1),\varepsilon)) \\ \quad \text{if not } \mathsf{expandable}_P(t_n), \\ \quad \bigsqcup \mathsf{cuts}(t_n) = \mathit{false} \\ \quad \text{and } \mathsf{choices}(G,P) > n \\ \\ (G,t_1 :: \cdots :: t_n) \\ \quad \text{otherwise.} \end{cases}$$

The first case of the definition deals with a tree which is formed by its root only. If this root does not contain procedure calls, or is inconsistent, the computation stops. Otherwise, the first child is added to the root. Note that the definition of $\imath$ embeds a leftmost selection rule. We specify a depth first search rule since only the first child is attached. The second case of the definition deals with a tree which has already some subtrees. In this case, we try to expand the rightmost subtree. If this is not possible, we check whether there exists another child of the root goal and whether the subtree rooted at the previous child has executed a cut. In this case we stop the computation. Otherwise, we add a new brother or continue the computation. Note that we use the selection and search rules of Prolog in this case too.

Definition 13. The operational semantics of a goal G in a program P is defined as $\mathcal{O}_{G,P} = \mathsf{lub}_{i\geqslant 0}\mathcal{O}_{G,P,i}$, where $\mathcal{O}_{G,P,0} = (G,\varepsilon)$ and $\mathcal{O}_{G,P,i+1} = \mathsf{expand}_P(\mathcal{O}_{G,P,i})$.

The lub is computed w.r.t. the $\subseteq_l$ relation in the completion of $\mathbb{T}$ into a complete lattice w.r.t. $\subseteq_l$. Note that $T \subseteq_l \mathsf{expand}_P(T)$, as it can be easily shown (see Lemma 23). Hence $\{\mathcal{O}_{G,P,i}\}_{i\geqslant 0}$ is by construction a sequence of P-directed trees for G, increasing w.r.t. $\subseteq_l$.

5. A tree semantics

The semantics we described in the previous section can be seen as an abstract version of an interpreter for logic programming which uses the selection and the search rules of Prolog and the cut operator. It is goal-dependent and it does not enjoy any compositionality property. We define now a different semantics, that deals with SLD trees, enriched with parametric control information, in the form of *observability constraints*. It is still able to consider the selection and the search rules of Prolog and the cut operator. Moreover, it is a compositional semantics and can be constructed in a top-down and in a bottom-up way, as well as in a goal-dependent and in a goal-independent way. The importance of this semantics is that it enjoys both the properties of an operational semantics (top-down and goal-dependent) and the properties of a denotational semantics (bottom-up and goal-independent), depending on the way it is constructed. We will show that all these ways of constructing the semantics are equivalent. This equivalence will be used to prove the relationship between the denotational semantics which will be defined in Sections 8 and 9 and the operational semantics defined in Section 4. Actually, this tree semantics can be seen as our concrete semantics, over which more abstract semantics (computed answer semantics, call patterns semantics and so on) can be defined. Since this tree semantics will be closely related to our operational semantics, we have a general correctness result between the operational semantics and the various abstract semantics which can be defined.

A node which belongs to an SLD tree may not belong to the tree derived by using the SLD resolution together with the control rules of Prolog and the cut operator. This second tree is typically a strict subset of the whole, *declarative*, SLD tree. In the operational semantics defined above, we decided which portion of the SLD tree is observable and which is not during the same construction of the SLD tree. If we want to do this in a goal-independent way, we cannot make this decision before the actual evaluation of the goal. Instead, control information must be specified in a goal-independent way, in order to get a goal-independent, compositional semantics which is able to deal with the control rules of Prolog and the cut operator.

We already know what a tree is (Definition 4). We define here some operators on trees.

Definition 14. The following operators are defined on the set of sequences of trees.

- **Instantiation.**[1] If not $\mathsf{div}(G)$ then

$$G\boxed{\circ}(G',\varepsilon) = (G \texttt{ and } G',\varepsilon)$$

$$G\boxed{\circ}(G',\tilde{t}) = (G \texttt{ and } G',\tau(G)\boxed{\circ}\tilde{t})$$

$$G\boxed{\circ}(\tilde{t}_1 :: \tilde{t}_2) = G\boxed{\circ}\tilde{t}_1 :: G\boxed{\circ}\tilde{t}_2.$$

- **Product.** Let G_2 be the root of T. We define

$$(G_1,\varepsilon)\boxtimes T = \begin{cases} (G_1 \texttt{ and } G_2,\varepsilon) & \text{if } \mathsf{div}(G_1) \\ G_1\boxed{\circ}\, T & \text{if not } \mathsf{div}(G_1). \end{cases}$$

$$(G_1,\tilde{t}_1) \boxtimes T = (G_1 \texttt{ and } G_2, \tilde{t}_1\boxtimes T)$$

$$(\tilde{t}_1 :: \tilde{t}_2) \boxtimes T = \tilde{t}_1\boxtimes T :: \tilde{t}_2\boxtimes T.$$

- **Cylindrification.** Let x be a variable. We define

$$\boxed{\exists}_x(G,\varepsilon) = (\texttt{exists } x.G,\varepsilon)$$

$$\boxed{\exists}_x(G,\tilde{t}) = (\texttt{exists } x.G,\ \boxed{\exists}_x\,\tilde{t})$$

$$\boxed{\exists}_x(\tilde{t}_1 :: \tilde{t}_2) = \boxed{\exists}_x\tilde{t}_1 :: \boxed{\exists}_x\tilde{t}_2.$$

[1] In this definition and in many others which will follow, the inductive case is given for a generic partition of a sequence (of trees, in this case). In all these cases, it can be easily shown that the result is the same for every possible choice of the partition.

- **Cut.** Let d be a non-negative integer. We define

$$!_d(G,\varepsilon) = (\mathtt{cut}_d(G),\varepsilon)$$
$$!_d(G,\tilde{t}) = (\mathtt{cut}_d(G),!_{d+1}(\tilde{t}))$$
$$!_d(\tilde{t}_1 :: \tilde{t}_2) = !_d\tilde{t}_1 :: !_d\tilde{t}_2.$$

Moreover, we define

$$!(G,\varepsilon) = (\mathtt{cut}(G),\varepsilon)$$
$$!(G,\tilde{t}) = (\mathtt{cut}(G),!\tilde{t})$$
$$!(\tilde{t}_1 :: \tilde{t}_2) = !\tilde{t}_1 :: !\tilde{t}_2.$$

- **Uncut.** $¡\tilde{t} = ¡_0\tilde{t}$, where

$$¡_i(G,\varepsilon) = (¡_iG,\varepsilon)$$
$$¡_i(G,\tilde{t}) = (¡_iG,¡_{i+1}\tilde{t})$$
$$¡_i(\tilde{t}_1 :: \tilde{t}_2) = ¡_i\tilde{t}_1 :: ¡_i\tilde{t}_2$$

and

$$¡_i(c) = c$$
$$¡_i(\mathtt{p}(x)) = \mathtt{p}(x)$$
$$¡_i(G_1 \mathtt{\ and\ } G_2) = ¡_i(G_1) \mathtt{\ and\ } ¡_i(G_2)$$
$$¡_i(\mathtt{exists\ } x.G) = \mathtt{exists\ } x.¡_iG$$
$$¡_i(\mathtt{cut}_d(G)) = \mathtt{cut}_d(¡_iG)$$
$$¡_i(\mathtt{cut}(G)) = \mathtt{cut}_i(¡_iG).$$

- **Expansion.** Let G' be a goal. We define

$$\phi^{G'}(G,\tilde{t}) = (G',(G,\tilde{t})).$$

- **Root swapping.** Let G' be a goal. We define

$$\psi^{G'}(G,\tilde{t}) = (G',\tilde{t}).$$

- **Sum.**

$$(G,\tilde{t}_1) \boxplus (G,\tilde{t}_2) = (G,\tilde{t}_1 :: \tilde{t}_2).$$

Figs. 3–6 show some examples of the operations defined above.

Interpretations give a (possibly partial) information for every predicate symbol:

Definition 15. A tree interpretation I is a function which maps any predicate symbol $\mathtt{p}$ into a cut-closed tree for $\mathtt{p}(\alpha)$. α is a distinguished variable which is not allowed in the syntax of the clauses. Given a program P, a tree interpretation I is (strongly) P-directed if and only if $I(\mathtt{p})$ is (strongly) P-directed for every predicate symbol $\mathtt{p}$. We

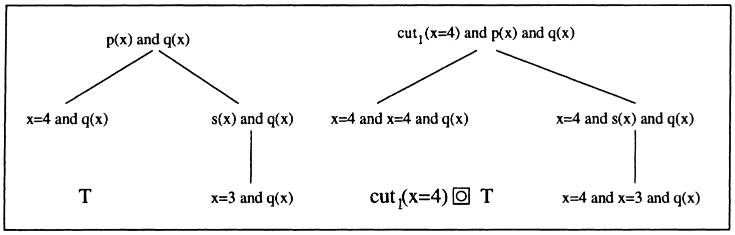

Fig. 3. A tree and its instantiation with the goal $\mathtt{cut}_1(x = 4)$.

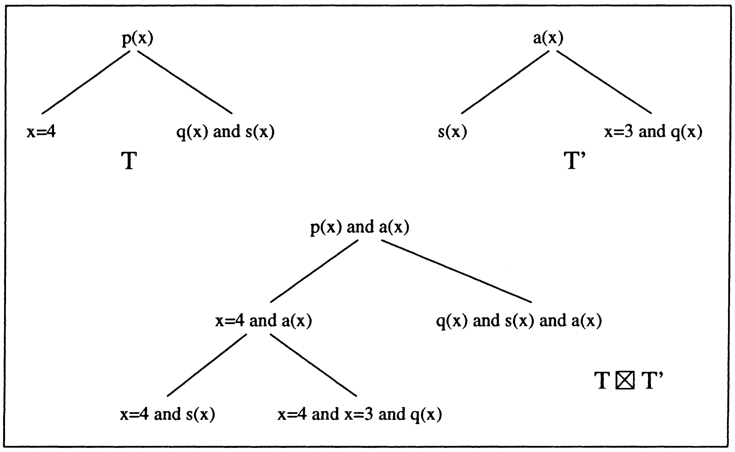

Fig. 4. Two trees and one of their products.

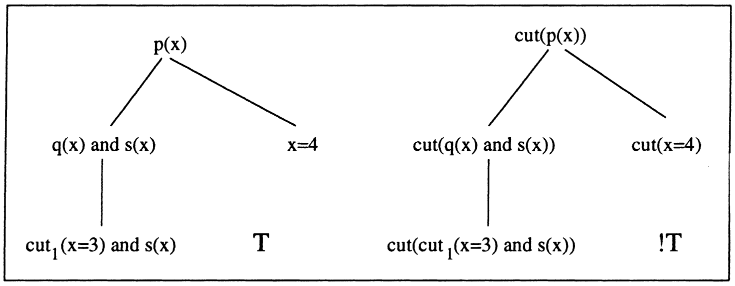

Fig. 5. A tree and the result of the ! operation applied to it.

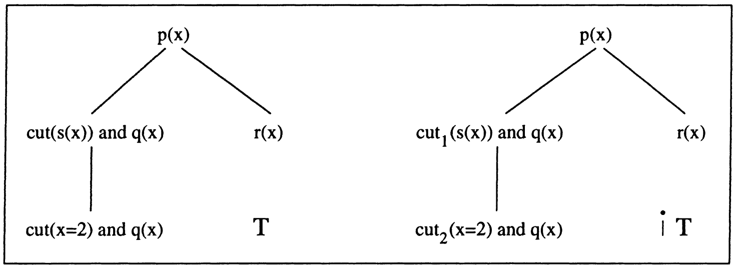

Fig. 6. A tree and the result of the uncut operation applied to it.

define $I_1 \subseteq I_2$ if and only if $I_1(\mathtt{p}) \subseteq I_2(\mathtt{p})$ for every predicate symbol p. The bottom element of this poset is the interpretation $I_0^{\mathbb{T}}$ such that $I_0^{\mathbb{T}}(\mathtt{p}) = (\mathtt{p}(\alpha), \varepsilon)$ for every predicate symbol p.

We define $\mathscr{T}_P^{\mathbb{T}}[\![G]\!]I$ as the tree for G computed using the information contained in I in order to interpret the procedure calls contained in G:

Definition 16. Given a program P, the tree for a goal in a given strongly P-directed tree interpretation I is given by

$$\begin{aligned}
\mathscr{T}_P^{\mathbb{T}}[\![c]\!]I &= (c, \varepsilon)\\
\mathscr{T}_P^{\mathbb{T}}[\![G_1 \text{ and } G_2]\!]I &= \mathscr{T}_P^{\mathbb{T}}[\![G_1]\!]I \boxtimes \mathscr{T}_P^{\mathbb{T}}[\![G_2]\!]I\\
\mathscr{T}_P^{\mathbb{T}}[\![\mathtt{exists}\, x.G]\!]I &= \exists\!\!\!\exists_x \mathscr{T}_P^{\mathbb{T}}[\![G]\!]I\\
\mathscr{T}_P^{\mathbb{T}}[\![\mathtt{p}(x)]\!]I &= I(\mathtt{p})[x/\alpha]\\
\mathscr{T}_P^{\mathbb{T}}[\![\mathtt{cut}_d(G)]\!]I &= !_d \mathscr{T}_P^{\mathbb{T}}[\![G]\!]I\\
\mathscr{T}_P^{\mathbb{T}}[\![\mathtt{cut}(G)]\!]I &= ! \mathscr{T}_P^{\mathbb{T}}[\![G]\!]I,
\end{aligned}$$

where

$$\begin{aligned}
(\mathtt{p}(\alpha), \varepsilon)[x/\alpha] &= (\mathtt{p}(x), \varepsilon)\\
(\mathtt{p}(\alpha), \exists\!\!\!\exists_x ((\delta_{x,\alpha}, \varepsilon) \boxtimes \tilde{t}))[x/\alpha] &= (\mathtt{p}(x), \tilde{t})\\
(\mathtt{p}(\alpha), \exists\!\!\!\exists_y ((\delta_{y,\alpha}, \varepsilon) \boxtimes \tilde{t}))[x/\alpha] &= (\mathtt{p}(x),\ (\exists\!\!\!\exists_y ((\delta_{y,x}, \varepsilon) \boxtimes \tilde{t}))) \qquad \text{if } x \neq y.
\end{aligned}$$

Note that the definition of $[x/\alpha]$ is sufficient for our purposes since we assume I to be P-directed. It can be shown that $\mathscr{T}_P^{\mathbb{T}}[\![G]\!]I_0^{\mathbb{T}} = (G, \varepsilon)$ for every goal G, by straightforward induction.

$\mathscr{D}^P[\![G]\!]I$ is similar to $\mathscr{T}_P^{\mathbb{T}}[\![G]\!]I$. However, procedure calls are *extended*, in the sense that they are replaced with the denotation ($\mathscr{T}_P^{\mathbb{T}}[\![\,]\!]$) of their body:

Definition 17. The extended tree for a goal G in a tree interpretation I with respect to a program P is[2]

$$\begin{aligned}
\mathscr{D}^P[\![c]\!]I &= (c, \varepsilon)\\
\mathscr{D}^P[\![G_1 \text{ and } G_2]\!]I &= \mathscr{D}^P[\![G_1]\!]I \boxtimes \mathscr{D}^P[\![G_2]\!]I
\end{aligned}$$

[2] Since $\boxplus$ is associative, the exact ordering of the computation of a sum like $T_1 \boxplus \cdots \boxplus T_n$ is irrelevant. The same remark holds for Eqs. (25), (26) and (32).

$$\mathscr{D}^P[\![\texttt{exists}\, x.G]\!]I = \boxminus_x \mathscr{D}^P[\![G]\!]I$$
$$\mathscr{D}^P[\![\texttt{cut}_d(G)]\!]I = !_d \mathscr{D}^P[\![G]\!]I$$
$$\mathscr{D}^P[\![\texttt{cut}(G)]\!]I = !\mathscr{D}^P[\![G]\!]I$$

$$\mathscr{D}^P[\![\mathtt{p}(x)]\!]I = \begin{cases} \psi^{\mathtt{p}(x)}\boxminus_y\left((\delta_{y,x},\varepsilon)\boxtimes¡\left(\begin{array}{l}\left[\phi^{\mathtt{p}(y)}\mathscr{T}_P^{\mathbb{T}}[\![G_1]\!]I\right]\boxplus\cdots\\ \cdots\boxplus\left[\phi^{\mathtt{p}(y)}\mathscr{T}_P^{\mathbb{T}}[\![G_n]\!]I\right]\end{array}\right)\right)\\ \text{if } x \neq y,\\ \\ ¡\left(\phi^{\mathtt{p}(x)}\mathscr{T}_P^{\mathbb{T}}[\![G_1]\!]I\boxplus\cdots\boxplus\phi^{\mathtt{p}(x)}\mathscr{T}_P^{\mathbb{T}}[\![G_n]\!]I\right)\\ \text{if } x = y. \end{cases}$$

where $\mathtt{p}(y)$ `:-` G_1 `or` $\cdots$ `or` G_n. is the definition of p in the program P.

Note that the definition of $\mathscr{D}^P[\![\mathtt{p}(x)]\!]I$ closes the scopes of the cuts contained in the definition of p given by P. This is because a cut is local to the procedure definition it belongs to.

It can be shown by structural induction on goals that $\mathscr{T}_P^{\mathbb{T}}[\![G]\!]I$ and $\mathscr{D}^P[\![G]\!]I$ are trees. Namely, every G_1 `and` G_2 construct with $\mathsf{div}(G_1)$ that they contain in some node is such that G_2 contains neither $\texttt{cut}_d()$ nor $\texttt{cut}()$ constructs. Moreover, it can be easily checked that $\mathscr{T}_P^{\mathbb{T}}[\![\mathtt{p}(x)]\!]I$ and $\mathscr{D}^P[\![\mathtt{p}(x)]\!]I$ are cut-closed for every interpretation I.

The unfolding of a tree T with respect to a tree interpretation I is the tree obtained by substituting the procedure calls in the leaves of T by their denotation contained in I. Formally:

Definition 18. Given a program P, the unfolding of a tree with respect to a strongly P-directed tree interpretation I is defined as

$$(G,\varepsilon) \bowtie^P I = \mathscr{D}^P[\![G]\!]I$$
$$(G,\tilde{t}) \bowtie^P I = (G, \tilde{t} \bowtie^P I)$$
$$(\tilde{t}_1 :: \tilde{t}_2) \bowtie^P I = (\tilde{t}_1 \bowtie^P I) :: (\tilde{t}_2 \bowtie^P I)$$

and is extended to tree interpretations as $(I_1 \bowtie^P I_2)(\mathtt{p}) = I_1(\mathtt{p}) \bowtie^P I_2$. We define the unfolding immediate consequence operator as

$$T_P^{\mathbb{T}}(I) = I_0^{\mathbb{T}} \bowtie^P I. \tag{5}$$

Note that $I_0^{\mathbb{T}}$ is strongly P-directed and it can be shown that $\bowtie^P$ is closed on the set of strongly P-directed interpretations (see Ref. [22]). Hence we can define two sequences of strongly P-directed interpretations as:

$$I_0 = I_0^{\mathbb{T}}, \quad I_{i+1} = I_0^{\mathbb{T}} \bowtie^P I_i = T_P^{\mathbb{T}}(I_i) \tag{6}$$

and

$$J_0 = I_0^{\mathbb{T}}, \quad J_{i+1} = J_i \bowtie^P I_0^{\mathbb{T}}. \tag{7}$$

Definition 19. In the completion of $(\mathbb{T}, \subseteq)$ we can define

$$\mathscr{S}_P^{bu} = \mathsf{lub}_i\, I_i, \qquad \mathscr{S}_P^{td} = \mathsf{lub}_i\, J_i.$$

$\mathscr{S}_P^{bu}$ is the bottom-up unfolding semantics of P, while $\mathscr{S}_P^{td}$ is the top-down unfolding semantics of P. Moreover, given a goal G we define

$$K_0^G = (G, \varepsilon), \qquad K_{i+1}^G = K_i^G \bowtie^P I_0^{\mathbb{T}}, \tag{8}$$

and we say that $\mathsf{lub}_i\, K_i^G$ is the goal-dependent semantics of G in P.

Note that $\{J_i\}_{i \geqslant 0}$ and $\{K_i^G\}_{i \geqslant 0}$ are increasing sequences (of interpretations and of trees, respectively) w.r.t. the $\subseteq$ relation. This is a consequence of the definition of $\bowtie^P$ itself. This entails that the lubs of such sequences exist on the completion of $(\mathbb{T}, \subseteq)$. We will prove now that the same holds for the sequence $\{I_i\}_{i \geqslant 0}$.

Proposition 20. *If $I_1 \subseteq I_2$ then $T_P^{\mathbb{T}}(I_1) \subseteq T_P^{\mathbb{T}}(I_2)$.*

Therefore, $\mathsf{lub}_i\, I_i$ exists on the completion of $(\mathbb{T}, \subseteq)$.

The proposition above tells us that $T_P^{\mathbb{T}}$ is a monotonic operator w.r.t. $\subseteq$. Hence its extension is continuous and we conclude that $I_i = \left(T_P^{\mathbb{T}}\right)^i$ for every $i \geqslant 0$ (Eq. (6)) entails $\mathsf{lub}_i\, I_i = \mathtt{lfp}\ T_P^{\mathbb{T}}$, i.e., $\mathscr{S}_P^{bu} = \mathtt{lfp}\ T_P^{\mathbb{T}}$. This means that $T_P^{\mathbb{T}}$ can be used to compute the goal-independent semantics of the program P.

The following theorem:

Theorem 21. *Given a program P and a goal G, we have*

(1) $\mathscr{S}_P^{bu} = \mathscr{S}_P^{td}$;

(2) $\mathsf{lub}_i\, K_i^G = \mathscr{T}_P^{\mathbb{T}}[\![G]\!]\mathscr{S}_P^{bu} = \mathscr{T}_P^{\mathbb{T}}[\![G]\!]\mathscr{S}_P^{td}$

shows that the two approaches are equivalent, i.e., they lead to the same denotation for a goal in a program. Therefore, we can define *the* tree semantics of a program P as

$$\mathscr{S}_P^{\mathbb{T}} = \mathscr{S}_P^{bu} = \mathscr{S}_P^{tp}.$$

This result is extremely important because the goal-dependent semantics will be related to our operational semantics of Section 4 (Theorem 30), while the goal-independent one will be related to our denotational semantics of Section 8 (Theorems 40, 46 and 53) and Section 9 (Theorems 40, 55 and 56).

Let us compare our tree semantics with the semantics defined in Ref. [10]. First of all, they do not provide any goal-independent top-down definition of the semantics of logic programming with Prolog selection and search rule and the cut operator. Their denotational semantics is defined in a goal-independent bottom-up way only. Furthermore, they relate to every goal in a program a set of substitutions. So their semantics is too abstract to be taken as concrete semantics for observables like call patterns. On the contrary, our tree semantics is extremely concrete, allowing to observe the whole SLD tree. Therefore, every observable that is more abstract than the SLD tree can be modelled as an abstract interpretation of our tree semantics.

6. From the tree semantics to the operational semantics

We want now to relate the denotational tree semantics defined in the previous section with the operational semantics defined in Section 4.

Our operational semantics constructs a sequence of trees, where the ith tree $\mathcal{O}_{G,P,i}$ is the subset of the SLD tree visited by a Prolog interpreter after i resolution steps. Our tree semantics, on the contrary, deals with whole SLD trees, enriched with information necessary to extract the subset which is visited (*observed*) by a Prolog interpreter. Roughly speaking, if a connection has to be established between these two semantics, then the operational trees are exactly the subset of the trees constructed by the tree semantics formed by nodes whose observability is not *false* and whose parent is satisfiable. This idea will be formalised in this section. First of all, we have to characterise the trees constructed by our operational semantics.

We define the set of P-Prolog trees as the set of P- directed trees which are built by using the selection and search rules of Prolog and by considering the cut operator. Namely:

Definition 22. Given a Prolog program P and a goal G, (G, ε) is a P-Prolog tree for G. $(G, t_1 :: \cdots :: t_n)$ is a P-Prolog tree if and only if it is P-directed, t_i is a P-Prolog tree for every $i = 1, \ldots, n$, not $\mathsf{expandable}_P(t_i)$ and $\bigsqcup \mathtt{cuts}(t_i) = \mathit{false}$ for every $i = 1, \ldots, n-1$.

The set of P-Prolog trees is closed by expansion:

Lemma 23. *Given a program P and a P-Prolog tree T, $\mathsf{expand}_P(T)$ is a P-Prolog-tree such that $T \subseteq_l \mathsf{expand}_P(T)$.*

Consider the sequence $\{K_i^G\}_{i \geqslant 0}$ of Eq. (8). The ith tree of such a sequence is an approximation of the SLD tree for the goal G in the program P built using a leftmost selection rule. The search rule and the cut operator are not taken into account. However, those trees contain all the information needed to select their portion which is actually visited by a given search rule and by taking the cut operator into account. This means that we can define a map α_{obs} which, given a strongly P-directed tree, selects the subset of its nodes which are actually visited by a Prolog interpreter. Using the map α_{obs}, we can define the semantics of a goal G in a program P as $\alpha_{obs}(\mathsf{lub}_i\, K_i^G)$. This means that we first compute the goal-dependent tree semantics of the goal G in the program P and then we abstract this semantics with respect to the control of Prolog. We will show that $\mathsf{lub}_i\, \mathcal{O}_{G,P,i} = \alpha_{obs}(\mathsf{lub}_i\, K_i^G)$. This result can be seen as a correctness result of our denotational tree semantics defined in Section 5 with respect to our operational semantics defined in Section 4. α_{obs} abstracts from the tree semantics all the information which is needed in order to get a compositional design. The result is exactly our operational semantics. In the following we will prove the result claimed above.

We start by defining a map $\alpha_{\mathbb{DT}}$ which adds to every node of a tree its *observability condition*. This condition must specify when a node is visited by a Prolog interpreter. These trees decorated with observability conditions are called *decorated trees*. Hence the name $\alpha_{\mathbb{DT}}$. Then we will define a map $\alpha_{\mathbb{T}}$ which, given a tree obtained by the function $\alpha_{\mathbb{DT}}$, selects the subset of its nodes which are actually visited by a Prolog interpreter. This subset is formed exactly by the set of nodes which have a true observability condition and whose parent is not inconsistent. Note that control information is embedded only in $\alpha_{\mathbb{DT}}$. Namely, $\alpha_{\mathbb{T}}$ is independent of every choice of control.

The definition of $\alpha_{\mathbb{DT}}$ needs the introduction of the concept of observability constraint. We already know the basic constraints of Definition 3. An observability constraint can be seen as a set of basic constraints. In order for an observability constraint to be satisfied in a constraint store, we require the set of basic constraints which form the observability constraint to be *individually true* in the constraint store. We formalise these concepts below:

Definition 24. The set $\mathcal{O}$ of observability constraints is defined as the minimal set containing $\mathcal{B}$ and such that if $S \subseteq \mathcal{O}$ then $\sqcup S$ and $\sqcap S$ belong to $\mathcal{O}$ and such that if $o \in \mathcal{O}$ then $-o \in \mathcal{O}$. We will often write $o_1 \sqcap \ldots \sqcap o_n$ for $\sqcap\{o_1, \ldots, o_n\}$ and similarly for $\sqcup$. We assume an injection function $\propto\ obs : \mathcal{B} \mapsto \mathcal{O}$ such that $b\ \propto\ obs$ is the basic constraint b seen as an observability constraint. $\propto\ obs$ has the greatest precedence: $o \sqcap b\ \propto\ obs$ stands for $o \sqcap (b\ \propto\ obs)$. If $b' \in \mathcal{B}$ and $o \in \mathcal{O}$ then $b' \bullet o$ is defined as

$$\begin{aligned} b' \bullet (b\ \propto\ obs) &= (b' \wedge b)\ \propto\ obs \quad \text{if } b \in \mathcal{B} \\ b' \bullet \sqcap S &= \sqcap\, \{b' \bullet o \mid o \in S\} \\ b' \bullet \sqcup S &= \sqcup\, \{b' \bullet o \mid o \in S\} \\ b' \bullet (-o) &= -\, (b' \bullet o), \end{aligned}$$

while $\exists_x o$ is defined as

$$\begin{aligned} \exists_x (b\ \propto\ obs) &= (\exists_x b)\ \propto\ obs \\ \exists_x (\sqcap S) &= \sqcap\, \{\exists_x o \mid o \in S\} \\ \exists_x (\sqcup S) &= \sqcup\, \{\exists_x o \mid o \in S\} \\ \exists_x (-o) &= -\, (\exists_x o). \end{aligned}$$

An observability constraint o is true if and only if $o \in \mathcal{B}$ and $o \neq false$, or $o = \sqcap S$ and every $o \in S$ is true, or $o = \sqcup S$ and there exists $o \in S$ which is true, or $o = -o'$ and o' is not true. If $o \in \mathcal{O}$ is not true, we say that it is false. Observability constraints are ordered as $o_1 \leqslant o_2$ if and only if for every $b \in \mathcal{B}$ if $b \bullet o_1$ is true then $b \bullet o_2$ is true. $\leqslant$ induces an equivalence relation on the set of observability constraints defined as $o_1 \equiv o_2$ if and only if $o_1 \leqslant o_2$ and $o_2 \leqslant o_1$. On the set of equivalence classes induced by $\equiv$, $\leqslant$ is a partial ordering relation. The top element w.r.t. $\leqslant$ is $\sqcap\{\}$ and the bottom element w.r.t. $\leqslant$ is $\sqcup\{\}$. We will write $true_{\mathcal{O}}$ for $\sqcap\{\}$ and $false_{\mathcal{O}}$ for $\sqcup\{\}$. In the following, an observability constraint will always stand for its equivalence class. Hence we will write $=$ instead of $\equiv$.

Note that every observability constraint is either true or false. Moreover, the satisfiability of every basic constraint which is used to build an observability constraint is checked individually. For instance, the observability constraint $x = 2 \sqcap x = 4$ is true, since $(x = 2) \neq \mathit{false}$ and $(x = 4) \neq \mathit{false}$. Note, however, that $x = 2 \sqcap x = 4 \leqslant x = 2$. Actually, in this case the strict inclusion holds: $x = 2 \sqcap x = 4 < x = 2$. This is because $(x = 2) \bullet (x = 2 \sqcap x = 4) = (x = 2) \sqcap \mathit{false}$ which is false, while $(x = 2) \bullet (x = 2) = (x = 2)$ which is true.

It can be easily shown that $\sqcap$ is monotonic, commutative and reductive, $\sqcup$ is monotonic, commutative and extensive, $-$ is counter-monotonic and $\bullet$ is monotonic in its second argument. Moreover, De Morgan's rules hold: $-(o_1 \sqcap o_2) \equiv -o_1 \sqcup -o_2$ and $-(o_1 \sqcup o_2) \equiv -o_1 \sqcap -o_2$.

Consider sets contained in $\mathcal{O} \cup (\mathcal{O} \times \mathbb{N})$. We can extend Eqs. (1)–(3) in the following way:

$$b \bullet \{\langle o_1 d_1\rangle \ldots, \langle o_n d_n\rangle, o'_1, \ldots, o'_m\}$$
$$= \{\langle b \bullet o_1 d_1\rangle, \ldots, \langle b \bullet o_n d_n\rangle, b \bullet o'_1, \ldots, b \bullet o'_m\},$$

where $b \in \mathscr{B}$,

$$\boldsymbol{\exists}_x\{\langle o_1, d_1\rangle, \ldots, \langle o_n, d_n\rangle, o'_1, \ldots, o'_m\} = \{\langle \exists_x o_1, d_1\rangle, \ldots, \langle \exists_x o_n, d_n\rangle, \exists_x o'_1, \ldots, \exists_x o'_m\} \tag{9}$$

and

$$\bigsqcup\{\langle o_1, d_1\rangle, \ldots, \langle o_n, d_n\rangle, o'_1, \ldots, o'_m\} = (\sqcup_i o_i) \sqcup (\sqcup_i o'_i).$$

We are now able to define a map $\beta^{\mathbb{T}}$ which, given a tree T, yields the set of its block conditions. Every block condition can be derived from a cut contained in T or from a path in T which must be expanded (i.e., from a *divergent* leaf of T):

Definition 25. Given $\tilde{t} \in \mathsf{Seq}(\mathbb{T})$, we define $\beta^{\mathbb{T}}(\tilde{t})$ as

$$\beta^{\mathbb{T}}(G, \varepsilon) = \begin{cases} \mathsf{cuts}(G) \propto \mathit{obs} \cup \{\mathsf{con}(G) \propto \mathit{obs}\} & \text{if } \mathsf{div}(G) \\ \mathsf{cuts}(G) \propto \mathit{obs} & \text{if not } \mathsf{div}(G) \end{cases}$$

$$\beta^{\mathbb{T}}(G, \tilde{t}) = \mathtt{cuts}(G) \propto \mathit{obs} \cup \varsigma(\beta^{\mathbb{T}}(\tilde{t}))$$

$$\beta^{\mathbb{T}}(\tilde{t}_1 :: \tilde{t}_2) = \beta^{\mathbb{T}}(\tilde{t}_1) \cup \left(-\bigsqcup \beta^{\mathbb{T}}(\tilde{t}_1)\right) \times \beta^{\mathbb{T}}(\tilde{t}_2),$$

where

$$o \times \{\langle o_1, d_1\rangle, \ldots, \langle o_n, d_n\rangle, o'_1, \ldots, o'_m\}$$
$$= \{\langle o \sqcap o_1, d_1\rangle, \ldots, \langle o \sqcap o_n, d_n\rangle, o \sqcap o'_1, \ldots, o \sqcap o'_m\} \tag{10}$$

and

$$\{\langle b_1, d_1\rangle, \ldots, \langle b_n, d_n\rangle, b'_1, \ldots, b'_m\} \propto \mathit{obs}$$
$$= \{\langle b_1 \propto \mathit{obs}, d_1\rangle, \ldots, \langle b_n \propto \mathit{obs}, d_n\rangle, b'_1 \propto \mathit{obs}, \ldots, b'_m \propto \mathit{obs}\}. \tag{11}$$

The first case of the above definition selects the cuts in the root of the tree. Since the root is actually a leaf, we add a divergence condition if it contains a procedure call. The second case selects the cuts in the root of the tree, which cannot be a leaf. Moreover, we recursively apply the map $\beta^{\mathbb{T}}$ to the children of the root. However, we want to discard cuts which are *internal* to the subtrees, in the sense that their scope does not allow them to reach possible brothers of the root. This is done through the use of a *shaking* function ς, which is a straightforward extension of the map ς of Eq. (4):

$$\varsigma(\{\langle o_1, d_1\rangle, \ldots, \langle o_n, d_n\rangle, o'_1, \ldots, o'_m\}) = \{o'_1, \ldots, o'_m\} \cup \bigcup_{d_i > 1} \{\langle o_i, d_i - 1\rangle\}. \tag{12}$$

The third case is applied to a sequence of length at least two. The sequence is split in two portions and the function $\beta^{\mathbb{T}}$ is recursively applied to each portion. However, the block conditions contained in the second portion are in the scope of the block conditions contained in the first portion of the sequence. Hence they are observable only if the block conditions of the first portion are not observable. This explains the use of the $\times$ operation.

We are now able to define the first abstraction of a sequence of trees. This abstraction maps every tree to a decorated tree, as defined below:

Definition 26. A decorated tree is an element of the set

$$\mathbb{DT} = \left\{ (o + G, \varepsilon) \middle| \begin{array}{l} o \in \mathcal{O}, \\ G \in \mathbb{G} \end{array} \right\} \cup \left\{ (o + G, \tilde{t}) \middle| \begin{array}{c} o \in \mathcal{O},\ G \in \mathbb{G}, \\ \mathsf{div}(G),\ \tilde{t} \in \mathsf{Seq}(\mathbb{DT}) \end{array} \right\}.$$

Nodes of a decorated tree contain pairs $o + G$ of an observability constraint and a goal, rather than simply a goal. The observability constraint is meant to represent under which condition the goal is observable. On the set of decorated trees we define an instantiation operation as

$$\begin{aligned} o\boxdot(o_1 + G_1, \varepsilon) &= (o \sqcap o_1 + G_1, \varepsilon) \\ o\boxdot(o_1 + G_1, \tilde{t}) &= (o \sqcap o_1 + G_1, o\boxdot\tilde{t}) \\ o\boxdot(\tilde{t}_1 :: \tilde{t}_2) &= (o\boxdot\tilde{t}_1 :: o\boxdot\tilde{t}_2). \end{aligned} \tag{13}$$

We map a tree into a decorated tree by applying block conditions to the nodes contained in their scope. This is achieved through the map $\alpha_{\mathbb{DT}} : \mathsf{Seq}^+(\mathbb{T}) \mapsto \mathsf{Seq}^+(\mathbb{DT})$ defined as:

$$\begin{aligned} \alpha_{\mathbb{DT}}(G, \varepsilon) &= (true_{\mathcal{O}} + G, \varepsilon) \\ \alpha_{\mathbb{DT}}(G, \tilde{t}) &= (true_{\mathcal{O}} + G, \alpha_{\mathbb{DT}}(\tilde{t})) \\ \alpha_{\mathbb{DT}}(\tilde{t}_1 :: \tilde{t}_2) &= \alpha_{\mathbb{DT}}(\tilde{t}_1) :: -\bigsqcup \beta^{\mathbb{T}}(\tilde{t}_1)\boxdot\alpha_{\mathbb{DT}}(\tilde{t}_2). \end{aligned} \tag{14}$$

Example 27. Consider for instance the tree shown in Fig. 7. Its abstraction into a decorated tree, induced by the $\alpha_{\mathbb{DT}}$ map, is shown in Fig. 8. Note that the observability constraint for the second child of the first child of the root is $-(x = 3)$. This is because, when it is possible to assume $x = 3$, then the cut of its left brother is executed and it gets cut. If it is not cut, then it is contained in a possibly infinite path, since it contains some procedure calls. Hence, in such a case, the nodes to its right must be non-observable. This explains why the observability part of the second child of the root requires that it must be possible to assume $x = 3$ (note that $x = 3 \equiv -(-(x = 3)))$. Actually, in such a case the cut of the first child of the first child of the root is executed, the second child of the first child of the root is cut and the possible divergence it contains cannot lead to an infinite path. The third child of the root is observable when it is possible to assume that $x = 3$ but it is not possible to assume that $x = 6$ (note that $-(-(x = 3)) \sqcap -(-(-(x = 3)) \sqcap x = 6) \equiv (x = 3 \sqcap -(x = 6)))$. Therefore, it is observable only when we start with a constraint store in which $x = 3$. Actually, this is the only case in which the cut of the first child of the first child of the root is executed and the second child of the root cannot be contained in an infinite path.

The abstraction through $\alpha_{\mathbb{DT}}$ expresses the observability of the nodes of a tree in a parametric way. This means that we do not decide which nodes are observable and which are not. We simply compute a constraint for every node. This constraint says exactly when the node is observable. Given an initial constraint store, we can decide which nodes are observable and which are not. We simply check observability constraints in the initial constraint store. Assume we start with an empty constraint store. The observable nodes of a tree are the nodes whose observability constraint

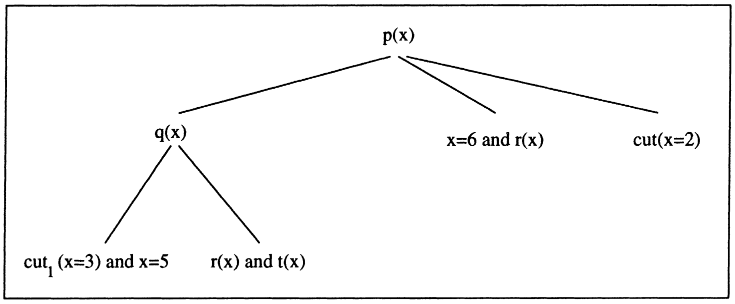

Fig. 7. A tree. Cut and divergence conditions are implicitly expressed.

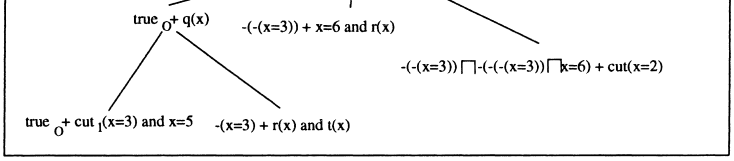

Fig. 8. The abstraction through $\alpha_{\mathbb{DT}}$ of the tree shown in Fig. 7. Cut and divergence conditions are now explicitly expressed.

is not false and whose parent is not inconsistent. This leads to the definition of the following map $\alpha_{\mathbb{T}}$. Assuming we start with an empty constraint store, $\alpha_{\mathbb{T}}$ gives us the set of nodes of the tree which are visited by a Prolog interpreter before leaving the tree. Formally:

Definition 28. We define $\alpha_{\mathbb{T}} : \mathsf{Seq}^+(\mathbb{DT}) \mapsto \mathsf{Seq}(\mathbb{T})$ as

$$\alpha_{\mathbb{T}}(o + G, \tilde{t}) = \begin{cases} \varepsilon & \text{if } o \text{ is } \mathit{false} \\ (G, \varepsilon) & \text{if } o \text{ is } \mathit{true} \text{ and } \mathsf{con}(G) = \mathit{false} \\ (G, \alpha_{\mathbb{T}}(\tilde{t})) & \text{if } o \text{ is } \mathit{true} \text{ and } \mathsf{con}(G) \neq \mathit{false}. \end{cases}$$

$$\alpha_{\mathbb{T}}(\tilde{t}_1 :: \tilde{t}_2) = \alpha_{\mathbb{T}}(\tilde{t}_1) :: \alpha_{\mathbb{T}}(\tilde{t}_2)$$

and $\alpha_{obs} : \mathsf{Seq}^+(\mathbb{T}) \mapsto \mathsf{Seq}^+(\mathbb{T})$ as [3]

$$\alpha_{obs}(\tilde{t}) = \alpha_{\mathbb{T}}(\alpha_{\mathbb{DT}}(\tilde{t})).$$

Consider the decorated tree of Fig. 8. Its abstraction through the $\alpha_{\mathbb{T}}$ map is the tree shown in Fig. 9. Note that it is contained, w.r.t. $\subseteq_l$, in the tree shown in Fig. 7. This is a general result, as it can be easily seen from the definitions of $\alpha_{\mathbb{DT}}$ and $\alpha_{\mathbb{T}}$: It can be seen that if $\alpha_{obs}(t_1 :: \cdots :: t_n) = d_1 :: \cdots :: d_m$ then $m \leqslant n$ and $d_i \subseteq_l t_i$ for every $i = 1, \ldots, m$. d_i is the observable part of t_i, for $i = 1, \ldots, m$, and the trees $t_{m+1}, \ldots, t_n$ are not observable.

We can prove that the abstraction through α_{obs} of a sequence of strongly P-directed trees is a sequence of P-Prolog trees.

Proposition 29. *Let $\tilde{t}$ be a sequence of strongly P-directed trees, and let $\alpha_{obs}(\tilde{t}) = d_1 :: \cdots :: d_m$. Then d_i is a P-Prolog tree for $i = 1, \ldots, m$ and $\bigsqcup \mathtt{cuts}(d_i) = \mathit{false}$ and not $\mathsf{expandable}_P(d_i)$ for $i = 1, \ldots, m - 1$.*

The following result states that our operational semantics, as defined in Section 4, is an abstraction of the denotational tree semantics defined in Section 5.

Theorem 30. *Given a goal G and a program P, consider the sequence $\{K_i^G\}_{i \geqslant 0}$ defined by Eq.* (8). *On the completion of the set of trees we have*

$$\bigcup_{i \geqslant 0} \mathcal{O}_{G,P,i} = \alpha_{obs}\left(\bigcup_{i \geqslant 0} K_i^G\right).$$

Note that the equality $\mathcal{O}_{G,P,i} = \alpha_{obs}(K_i^G)$, for $i \geqslant 0$, in general, does not hold.

The above theorem allows us to conclude that the denotational tree semantics is strictly related to our operational semantics. Intuitively, the denotational semantics contains more information than the operational one, in order to be

[3] α_{obs} never yields an empty sequence because the observability constraint of the root of the first tree in $\alpha_{\mathbb{DT}}(\tilde{t})$ is always $\mathit{true}_{\mathcal{O}}$.

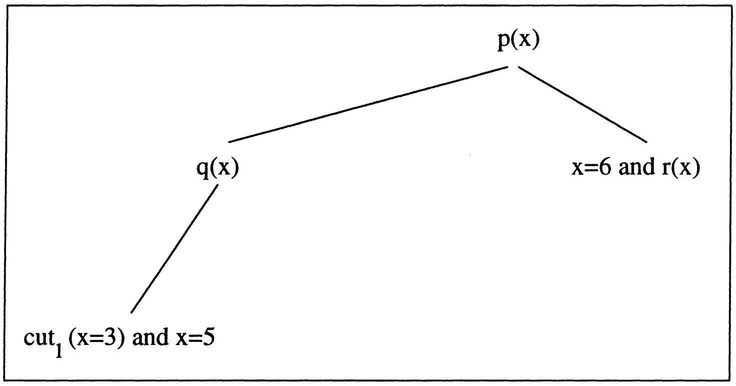

Fig. 9. The abstraction through α_T of the decorated tree shown in Fig. 8.

defined in a compositional fashion. This information is removed by the abstraction map α_{obs}. Since we know that all possible constructions of the denotational tree semantics give the same denotation for a given goal (Theorem 21), Theorem 30 can be extended to every possible way of construction of the denotational tree semantics.

We have proved that the denotational tree semantics is a compositional version of our operational semantics, in the sense that it admits a compositional definition and is strictly related to our operational semantics (Theorem 30). Therefore, we can use it to define a more abstract denotational semantics for Prolog with cut. Actually, our denotational semantics can be seen as a semantics which observes execution traces. We can use it even if we want to observe more abstract observables, like computed answers and call patterns. However, we want to show that a more *economical* denotational semantics can be defined as soon as we have chosen an observable property. By economical we mean that a semantics will deal with simpler domains than SLD trees. Such a semantics will be an abstract interpretation of the tree semantics of Section 5. This abstraction will be done in two steps. The first step, described in the following section, is independent of the chosen observable. It is only an alternative presentation of the tree semantics. The second step, described in Section 8 for computed answers and in Section 9 for call patterns, is dependent on the chosen observable.

7. Decorated tree semantics

In this section we define the abstract semantics induced by the abstraction map $\alpha_{\mathbb{DT}}$ of Eq. (14).

The map $\alpha_{\mathbb{DT}}$ is not onto. For instance, $\alpha_{\mathbb{DT}}(T)$ can never be a decorated tree whose root has an observability condition different from $true_O$. Hence we select the image of $\mathbb{T}$ through $\mathbb{DT}$: $\alpha_{\mathbb{DT}}(\mathbb{T})$. It can be easily checked that $\alpha_{\mathbb{DT}}$ is onto and one to one from $\mathbb{T}$ into $\alpha_{\mathbb{DT}}(\mathbb{T})$. This means that $\mathbb{T}$ and $\alpha_{\mathbb{DT}}(\mathbb{T})$ are isomorphic. Hence the abstract semantics induced by $\alpha_{\mathbb{DT}}$ is an alternative presentation of the tree semantics of Section 5. The abstract counterparts of the operators defined in Definition 14 can be defined in the classical way, as the abstraction by $\alpha_{\mathbb{DT}}$ of the concrete operators applied to the concretization of the abstract arguments. Note that the concretization of a tree in $\alpha_{\mathbb{DT}}(\mathbb{T})$ is simply the tree from which the observability constraints have been removed.

We introduce this intermediate semantics since it is better suited for an observable specific abstraction than the tree semantics defined in Section 5. Actually, this new formulation of the semantics applies cut and divergence conditions to their scope, rather than declaring them as it was done in Section 5. This means that the cut or divergence conditions which can affect a given node of the SLD tree are explicitly recorded in the node, rather than being derivable from an inspection of the SLD tree only. Therefore, any further abstraction need not save all cut conditions, but only those whose scope is open. This means that a simpler abstract interpretation can be based on this new version of the tree semantics. This technique can be called *control compilation*, while the technique of Section 5 can be called *control interpretation*. Note that the denotation obtained through a bottom-up construction using the $T_P^{\mathbb{T}}$ operator does not contain open cuts, due to the use of the ¡ operator (see Definitions 17 and 18). This means that the chain $\{(T_P^{\mathbb{T}})^i\}_{i\geqslant 0}$ does not contain cut() constructs. This is very important when finiteness of analysis is a must, for instance when one wants to build an analysis framework based on our semantics. Indeed, this means that every fixpoint approximation obtained through the immediate consequence operator of the analysis will not contain any information about open cuts or the scope of the cuts. This, in turn, means that the fixpoint is reached in less iterations.

Section 7.1 shows the optimal abstract counterparts of the operators of Definition 14 induced by the abstraction map $\alpha_{\mathbb{DT}}$. Section 7.2 shows a partial ordering relation on $\alpha_{\mathbb{DT}}(\mathbb{T})$ which can be lifted to every further abstraction of the semantics of Section 7.1. Moreover, it shows that this last semantics is an exact abstract interpretation of the bottom-up version of the semantics described in Section 5.

7.1. Control compilation

In this section, we want to show how the abstract operators on $\mathbb{DT}$ can be directly defined, without using the concretization-abstraction approach. This will be very important in order to define further abstractions of the semantics induced by $\alpha_{\mathbb{DT}}$.

We will define a bottom-up version of this semantics only. We will show that for that version we do not need the $!_d$ operator and that we can simplify the definition of the immediate consequence operator w.r.t. the definition of $T_P^{\mathbb{T}}$ of Eq. (5).

We define a map $o^{\mathbb{DT}}$ which, given a sequence of decorated trees, yields the set of its convergent leaves, as a set of observability constraints. Hence, $\bigsqcup o^{\mathbb{DT}}(\tilde{t})$ is true if and only if there is a consistent and observable convergent leaf in $\tilde{t}$. We define a map $\kappa^{\mathbb{DT}}$ which selects the set of cut conditions contained in a sequence of decorated trees and a map $\delta^{\mathbb{DT}}$ which selects the set of divergent leaves of a sequence of trees. Therefore, $\bigsqcup\delta^{\mathbb{DT}}(\tilde{t})$ is true if and only if a divergent leaf of $\tilde{t}$ is consistent and observable. This means that $\tilde{t}$ might contain the first portion of an infinite path. Finally, computation stops in a sequence of trees if this sequence executes a cut whose scope reaches the roots of the trees, or if a divergent leaf exists in the sequence. This leads to the definition of a map $\beta^{\mathbb{DT}}$. Formally:

Definition 31. We define the following maps:

$$\begin{aligned}
\mathsf{o}^{\mathbb{DT}}(o+G,\varepsilon) &= \begin{cases} \emptyset & \text{if } \mathsf{div}(G) \\ \{o \sqcap \mathsf{con}(G) \propto obs\} & \text{if not } \mathsf{div}(G) \end{cases} \\
\mathsf{o}^{\mathbb{DT}}(o+G,\tilde{t}) &= \mathsf{o}^{\mathbb{DT}}(\tilde{t}) \\
\mathsf{o}^{\mathbb{DT}}(\tilde{t}_1 :: \tilde{t}_2) &= \mathsf{o}^{\mathbb{DT}}(\tilde{t}_1) \cup \mathsf{o}^{\mathbb{DT}}(\tilde{t}_2).
\end{aligned} \tag{15}$$

$$\begin{aligned}
\kappa^{\mathbb{DT}}(o+G,\varepsilon) &= o \times (\mathsf{cuts}(G) \propto obs) \\
\kappa^{\mathbb{DT}}(o+G,\tilde{t}) &= o \times (\mathtt{cuts}(G) \propto obs) \cup \varsigma\kappa^{\mathbb{DT}}(\tilde{t}) \\
\kappa^{\mathbb{DT}}(\tilde{t}_1 :: \tilde{t}_2) &= \kappa^{\mathbb{DT}}(\tilde{t}_1) \cup \kappa^{\mathbb{DT}}(\tilde{t}_2),
\end{aligned}$$

where $\times$ was defined by Eq. (10), $\propto obs$ was defined by Eq. (11) and ς was defined by Eq. (12).

$$\begin{aligned}
\delta^{\mathbb{DT}}(o+G,\varepsilon) &= \begin{cases} \emptyset & \text{if not } \mathsf{div}(G) \\ \{o \sqcap \mathsf{con}(G) \propto obs\} & \text{if } \mathsf{div}(G) \end{cases} \\
\delta^{\mathbb{DT}}(o+G,\tilde{t}) &= \delta^{\mathbb{DT}}(\tilde{t}) \\
\delta^{\mathbb{DT}}(\tilde{t}_1 :: \tilde{t}_2) &= \delta^{\mathbb{DT}}(\tilde{t}_1) \cup \delta^{\mathbb{DT}}(\tilde{t}_2).
\end{aligned}$$

$$\beta^{\mathbb{DT}}(\tilde{t}) = \kappa^{\mathbb{DT}}(\tilde{t}) \cup \delta^{\mathbb{DT}}(\tilde{t}).$$

The following proposition gives some monotonicity results for the maps just defined on $\mathsf{Seq}(\mathbb{DT})$.

Proposition 32. *Given $\tilde{t}_1, \tilde{t}_2 \in \mathsf{Seq}(\mathbb{T})$ such that $\tilde{t}_1 \subseteq \tilde{t}_2$, we have*
(i) $\bigsqcup \mathsf{o}^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(\tilde{t}_1)) \leqslant \bigsqcup \mathsf{o}^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(\tilde{t}_2))$;
(ii) $\bigsqcup \delta^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(\tilde{t}_1)) \geqslant \bigsqcup \delta^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(\tilde{t}_2))$;
(iii) $\bigsqcup \varsigma^i\kappa^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(\tilde{t}_1)) \leqslant \bigsqcup \varsigma^i\kappa^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(\tilde{t}_2))$ *for every* $i \geqslant 0$;
(iv) $\bigsqcup \beta^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(\tilde{t}_1)) \geqslant \bigsqcup \beta^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(\tilde{t}_2))$.

The results above should be read as follows. If $\tilde{t}_1 \subseteq \tilde{t}_2$ then $\tilde{t}_2$ contains more convergent leaves than $\tilde{t}_1$ (point i). $\tilde{t}_2$ diverges less than $\tilde{t}_1$ (point ii). $\tilde{t}_2$ executes more cuts than $\tilde{t}_1$ (point iii). Point (ii) prevails over point (iii) (point iv).

We define now the abstract counterparts of the operators of Definition 14 induced by the $\alpha_{\mathbb{DT}}$ map.

Definition 33. Given a goal G such that not $\mathsf{div}(G)$, we define

$$\begin{aligned}
G\boxed{\circ}(o+G',\varepsilon) &= (\mathsf{con}(G) \bullet o + G \ \mathtt{and}\ G') \\
G\boxed{\circ}(o+G',\tilde{t}) &= (\mathsf{con}(G) \bullet o + G \ \mathtt{and}\ G', \tau(G)\boxed{\circ}\tilde{t}) \\
G\boxed{\circ}(\tilde{t}_1 :: \tilde{t}_2) &= G\boxed{\circ}\tilde{t}_1 :: G\boxed{\circ}\tilde{t}_2.
\end{aligned}$$

Let $o_2 + G_2$ be the root of $T_2 \in \mathbb{DT}$. We define:

$$\begin{aligned}
(o_1 + G_1, \varepsilon)\boxtimes T_2 &= \begin{cases} (o_1 + G_1 \texttt{ and } G_2, \varepsilon) & \text{if } \mathsf{div}(G_1) \\ o_1\boxdot(G_1\boxed{\circ}T_2) & \text{if not } \mathsf{div}(G_1) \end{cases} \\
(o_1 + G_1, \tilde{t}_1)\boxtimes T_2 &= (o_1 + G_1 \texttt{ and } G_2, \tilde{t}_1 \boxtimes T_2) \qquad (16)\\
(\tilde{t}_1 :: \tilde{t}_2)\boxtimes T_2 &= \tilde{t}_1 \boxtimes T_2 :: -\bigsqcup \xi^{\mathbb{DT}}(\tilde{t}_1, T_2)\boxdot(\tilde{t}_2 \boxtimes T_2),
\end{aligned}$$

where

$$\begin{aligned}
\xi^{\mathbb{DT}}((o + G, \varepsilon), T) &= \begin{cases} \emptyset & \text{if } \mathsf{div}(G) \\ o \times (\mathsf{con}(G) \bullet \beta^{\mathbb{DT}}(T)) & \text{if not } \mathsf{div}(G) \end{cases} \\
\xi^{\mathbb{DT}}((o + G, \tilde{t}), T) &= \varsigma\xi^{\mathbb{DT}}(\tilde{t}, T) \qquad (17)\\
\xi^{\mathbb{DT}}(\tilde{t}_1 :: \tilde{t}_2, T) &= \xi^{\mathbb{DT}}(\tilde{t}_1, T) \cup \xi^{\mathbb{DT}}(\tilde{t}_2, T)
\end{aligned}$$

and $\boxdot$ was introduced in Eq. (13).

$$\begin{aligned}
\exists\!\!\exists_x(o + G, \varepsilon) &= (\exists_x o + \texttt{exists } x.G, \varepsilon) \\
\exists\!\!\exists_x(o + G, \tilde{t}) &= (\exists_x o + \texttt{exists } x.G, \exists\!\!\exists_x \tilde{t}) \qquad (18)\\
\exists\!\!\exists_x(\tilde{t}_1 :: \tilde{t}_2) &= \exists\!\!\exists_x\tilde{t}_1 :: \exists\!\!\exists_x\tilde{t}_2.
\end{aligned}$$

$$\begin{aligned}
!(o + G, \varepsilon) &= (o + \texttt{cut}(G), \varepsilon) \\
!(o + G, \tilde{t}) &= (o + \texttt{cut}(G), !\tilde{t}) \qquad (19)\\
!(\tilde{t}_1 :: \tilde{t}_2) &= !\tilde{t}_1 :: -\bigsqcup \mathsf{o}^{\mathbb{DT}}(\tilde{t}_1)\boxdot!\tilde{t}_2.
\end{aligned}$$

$$¡(\tilde{t}) = ¡_0(\tilde{t}), \quad \text{where}$$

$$\begin{aligned}
¡_i(o + G, \varepsilon) &= (o + ¡_i G, \varepsilon) \\
¡_i(o + G, \tilde{t}) &= (o + ¡_i G, ¡_{i+1}\tilde{t}) \\
¡_i(\tilde{t}_1 :: \tilde{t}_2) &= ¡_i\tilde{t}_1 :: ¡_i\tilde{t}_2.
\end{aligned}$$

$$\phi^G(T) = (true_\emptyset + G, T),$$

$$\begin{aligned}
\psi^G(o + G', \varepsilon) &= (o + G, \varepsilon) \\
\psi^G(o + G', \tilde{t}) &= (o + G, \tilde{t}).
\end{aligned}$$

$$(o + G, \tilde{t}_1)\boxplus(o + G, \tilde{t}_2) = \left(o + G, \tilde{t}_1 :: -\bigsqcup \beta^{\mathbb{DT}}(\tilde{t}_1)\boxdot\tilde{t}_2\right). \qquad (20)$$

The following proposition shows that every operator we have just defined above is correct w.r.t. the corresponding operator of Definition 14:

Proposition 34.
(i) *Given* $\tilde{t} \in \mathsf{Seq}(\mathbb{T})$ *and* $G \in \mathbb{G}$ *such that not* $\mathsf{div}(G)$, *if* $G = \tau(G)$ *when* $\#\tilde{t} > 1$, *we have*[4] $\alpha_{\mathbb{DT}}(G\boxed{\circ}\tilde{t}) = G\boxed{\circ}\alpha_{\mathbb{DT}}(\tilde{t})$.

[4] Note that the condition on G is not restrictive, since we use $\boxed{\circ}$ in the definition of $\boxtimes$ only, and in such a case we always have $\#\tilde{t} = 1$.

(ii) *Given* $\tilde{t} \in \mathsf{Seq}(\mathbb{T})$ *and* $T \in \mathbb{T}$, *with* T *cut-closed, we have*[5] $\alpha_{\mathbb{DT}}(\tilde{t} \boxtimes T) = \alpha_{\mathbb{DT}}(\tilde{t}) \boxtimes \alpha_{\mathbb{DT}}(T)$.
(iii) *Given* $\tilde{t} \in \mathsf{Seq}(\mathbb{T})$, *we have* $\exists\!\!\exists_x \alpha_{\mathbb{DT}}(\tilde{t}) = \alpha_{\mathbb{DT}}(\exists\!\!\exists_x \tilde{t})$.
(iv) *Given* $\tilde{t} \in \mathsf{Seq}(\mathbb{T})$, *we have* $\alpha_{\mathbb{DT}}(!\tilde{t}) = !\alpha_{\mathbb{DT}}(\tilde{t})$.
(v) *Given* $\tilde{t} \in \mathsf{Seq}(\mathbb{T})$ *and* $i \geqslant 0$, *we have* $\alpha_{\mathbb{DT}}(¡_i \tilde{t}) = ¡_i \alpha_{\mathbb{DT}}(\tilde{t})$, *assuming* $i > 0$ *when* $\#\tilde{t} > 1$. [6]
(vi) $\phi^G(\alpha_{\mathbb{DT}}(T)) = \alpha_{\mathbb{DT}}(\phi^G(T))$, *with* $G \in \mathbb{G}$ *and* $T \in \mathbb{T}$.
(vii) $\psi^G(\alpha_{\mathbb{DT}}(T)) = \alpha_{\mathbb{DT}}(\psi^G(T))$, *with* $G \in \mathbb{G}$ *and* $T \in \mathbb{T}$.
(viii) *Given* $(G, \tilde{t}_1), (G, \tilde{t}_2) \in \mathbb{T}$, *we have* $\alpha_{\mathbb{DT}}((G, \tilde{t}_1) \boxplus (G, \tilde{t}_2)) = \alpha_{\mathbb{DT}}(G, \tilde{t}_1) \boxplus \alpha_{\mathbb{DT}}(G, \tilde{t}_2)$.

The last operation which is used in the semantics of Section 5 is the substitution operation $[x/\alpha]$ (Definition 16). It is defined (and used) only on strongly P-directed trees for $\mathsf{p}(\alpha)$. Hence its abstract counterpart can be defined only on the abstraction of strongly P-directed trees for $\mathsf{p}(\alpha)$. The unique strongly P-directed tree for $\mathsf{p}(\alpha)$ of height one is $(\mathsf{p}(\alpha), \varepsilon)$. We define the substitution operator on its abstraction as

$$(true_{\mathcal{C}} + \mathsf{p}(\alpha), \varepsilon)[x/\alpha] = (true_{\mathcal{C}} + \mathsf{p}(x), \varepsilon).$$

Proposition 34, points (ii) and (iii), tells us that the abstraction of a strongly P-directed tree for $\mathsf{p}(\alpha)$ of height at least two must have the form

$$(true_{\mathcal{C}} + \mathsf{p}(\alpha), \exists\!\!\exists_y\, ((true_{\mathcal{C}} + \delta_{y,\alpha}, \varepsilon) \boxtimes \alpha_{\mathbb{DT}}(\tilde{t}))). \tag{21}$$

Therefore, if we define

$$(true_{\mathcal{C}} + \mathsf{p}(\alpha), \exists\!\!\exists_x\, ((true_{\mathcal{C}} + \delta_{x,\alpha}, \varepsilon) \boxtimes \alpha_{\mathbb{DT}}(\tilde{t})))[x/\alpha] = (true_{\mathcal{C}} + \mathsf{p}(x), \alpha_{\mathbb{DT}}(\tilde{t})) \tag{22}$$

and

$$\begin{aligned}&(true_{\mathcal{C}} + \mathsf{p}(\alpha), \exists\!\!\exists_y\, ((true_{\mathcal{C}} + \delta_{y,\alpha}, \varepsilon) \boxtimes \alpha_{\mathbb{DT}}(\tilde{t})))[x/\alpha]\\ &\quad = (true_{\mathcal{C}} + \mathsf{p}(x), \exists\!\!\exists_y\, ((true_{\mathcal{C}} + \delta_{y,x}, \varepsilon) \boxtimes \alpha_{\mathbb{DT}}(\tilde{t})))\end{aligned} \tag{23}$$

when $x \neq y$, we get a substitution operator which is defined exactly on the abstraction of the trees on which the concrete substitution operator is defined and such that

$$\alpha_{\mathbb{DT}}(T[x/\alpha]) = \alpha_{\mathbb{DT}}(T)[x/\alpha], \tag{24}$$

whenever they are defined (see Definition 16).

Now we have almost all the abstract counterparts of the operators introduced in Definition 14. Actually, we have not defined the abstract counterpart of the $!_d$ operation. This is because we are interested only in a bottom-up version of the abstract semantics on $\mathbb{DT}$. This abstract semantics will be obtained by substituting the abstract operators defined in this section with the operators introduced in Definition 14. We show now how a bottom-up-only version of the semantics defined in Section 5

[5] Note that the condition on T is not restrictive since, every time we use $\boxtimes$, the second argument is cut-closed (see Definitions 16 and 17 and the comments after this last definition). Note that in the use of $\boxtimes$ in the first case of the definition for $\mathscr{D}^P[\![\mathsf{p}(x)]\!]I$ we know that $\mathscr{T}_P^{\mathbb{T}}[\![G_i]\!]I$ is cut-closed for every $i = 1, \ldots, n$. Moreover, the operators $\phi^{\mathsf{p}(y)}$, $\boxplus$ and $¡$ cannot introduce cuts whose scope goes beyond the root of the tree they belong to. Then even in this case the second argument of $\boxtimes$ is cut-closed.

[6] Note that this condition is not restrictive since we use $¡$ only on sequences of trees formed by exactly one tree (see Definition 17).

does not use the $!_d$ operator. Moreover, its definition can be simplified w.r.t. that given at the end of Section 5.

Consider the bottom-up definition of the tree semantics of Section 5. It is defined as the least fixpoint of the immediate consequence operator $T_P^{\mathbb{T}}(I) = I_0^{\mathbb{T}} \bowtie^P I$. By definition of $I_0^{\mathbb{T}}$, we have $T_P^{\mathbb{T}}(I)(\mathtt{p}) = \mathscr{D}^P[\![\mathtt{p}(\alpha)]\!]I$. Since α is a distinguished variable, not present in programs, we have:

$$T_P^{\mathbb{T}}(I)(\mathtt{p}) = \psi^{\mathtt{p}(\alpha)}\exists\!\!\!\exists_y\left((\delta_{y,\alpha},\varepsilon)\boxtimes_{\mathsf{i}}\left(\begin{array}{c}\left[\phi^{\mathtt{p}(y)}\mathscr{T}_P^{\mathbb{T}}[\![G_1]\!]I\right]\boxplus\cdots\\ \cdots\boxplus\left[\phi^{\mathtt{p}(y)}\mathscr{T}_P^{\mathbb{T}}[\![G_n]\!]I\right]\end{array}\right)\right), \tag{25}$$

where $\mathtt{p}(y)$:- G_1 `or` $\cdots$ `or` G_n. is the definition of p in the program P. Since programs do not contain $\mathtt{cut}_d()$ constructs, we conclude that $T_P^{\mathbb{T}}$ can be defined without using the $!_d$ operation.

Eq. (25) gives rise to a $T_P^{\mathbb{DT}}$ operator as soon as we substitute the operators of Section 5 with the corresponding operators defined in this section. This must be done even in the definition of the $\mathscr{T}_P^{\mathbb{T}}[\![]\!]$ operator. This means that we get an immediate consequences operator on $\mathbb{DT}$ as

$$T_P^{\mathbb{DT}}(I)(\mathtt{p}) = \psi^{\mathtt{p}(\alpha)}\exists\!\!\!\exists_y\left((true_{\emptyset}+\delta_{y,\alpha},\varepsilon)\boxtimes_{\mathsf{i}}\left(\begin{array}{c}\left[\phi^{\mathtt{p}(y)}\mathscr{T}_P^{\mathbb{DT}}[\![G_1]\!]I\right]\boxplus\cdots\\ \cdots\boxplus\left[\phi^{\mathtt{p}(y)}\mathscr{T}_P^{\mathbb{DT}}[\![G_n]\!]I\right]\end{array}\right)\right) \tag{26}$$

where I is a decorated tree interpretation, i.e., a map from predicate symbols into decorated trees, and

$$\begin{aligned}
\mathscr{T}_P^{\mathbb{DT}}[\![c]\!]I &= (true_{\emptyset}+c,\varepsilon)\\
\mathscr{T}_P^{\mathbb{DT}}[\![G_1 \mathtt{\ and\ } G_2]\!]I &= \mathscr{T}_P^{\mathbb{DT}}[\![G_1]\!]I\boxtimes\mathscr{T}_P^{\mathbb{DT}}[\![G_2]\!]I\\
\mathscr{T}_P^{\mathbb{DT}}[\![\mathtt{exists}\ x.G]\!]I &= \exists\!\!\!\exists_x\mathscr{T}_P^{\mathbb{DT}}[\![G]\!]I\\
\mathscr{T}_P^{\mathbb{DT}}[\![\mathtt{p}(x)]\!]I &= I(\mathtt{p})[x/\alpha]\\
\mathscr{T}_P^{\mathbb{DT}}[\![\mathtt{cut}(G)]\!]I &= !\mathscr{T}_P^{\mathbb{DT}}[\![G]\!]I.
\end{aligned}$$

By Proposition 34 and by Eq. (24), we have the following theorem:

Theorem 35. *Let I be a tree interpretation and G be a goal. We have*
(i) $\alpha_{\mathbb{DT}}(\mathscr{T}_P^{\mathbb{T}}[\![G]\!]I) = \mathscr{T}_P^{\mathbb{DT}}[\![G]\!]\alpha_{\mathbb{DT}}(I)$,
(ii) $\alpha_{\mathbb{DT}}(T_P^{\mathbb{T}}(I)) = T_P^{\mathbb{DT}}(\alpha_{\mathbb{DT}}(I))$,
where $\alpha_{\mathbb{DT}}(I)(\mathtt{p}) = \alpha_{\mathbb{DT}}(I(\mathtt{p}))$.

7.2. A partial ordering relation on $\alpha_{\mathbb{DT}}(\mathbb{T})$

We want to define, now, a relation $\leqslant$ on $\mathsf{Seq}^+(\mathbb{DT})$ which will turn out to be a partial ordering relation on $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))$. Note that $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))\subseteq\mathsf{Seq}^+(\mathbb{DT})$. If we prove that $\alpha_{\mathbb{DT}}$ is both strict and monotonic w.r.t. $\leqslant$ and that $T_P^{\mathbb{DT}}$ is monotonic, Theorem 35 allows us to conclude that the semantics defined as the least fixpoint of the $T_P^{\mathbb{DT}}$ operator (w.r.t. $\leqslant$) is the abstraction by $\alpha_{\mathbb{DT}}$ of $\mathscr{S}_P^{\mathbb{T}} = \mathtt{lfp}\ T_P^{\mathbb{T}}$.

The definition of a partial ordering relation on $\mathsf{Seq}(\mathbb{DT})$, such that if $\tilde{t}_1\subseteq\tilde{t}_2$ then $\alpha_{\mathbb{DT}}(\tilde{t}_1)\leqslant\alpha_{\mathbb{DT}}(\tilde{t}_2)$, is not trivial. Indeed, the sequence of decorated trees $\alpha_{\mathbb{DT}}(\tilde{t}_2)$ is *bigger* than the sequence of decorated trees $\alpha_{\mathbb{DT}}(\tilde{t}_1)$. This means that every node in

$\alpha_{\mathbb{DT}}(\tilde{t}_1)$ has a companion node in $\alpha_{\mathbb{DT}}(\tilde{t}_2)$ in the same position relative to the root. However, the observability conditions of these two nodes can be very different. We will show that the following relation is the partial ordering relation on $\alpha_{\mathbb{DT}}(\mathsf{Seq}(\mathbb{T}))$ we are looking for:

Definition 36. On $\mathsf{Seq}(\mathbb{DT})$ we define the relation $\leqslant$ as follows:

$$(o+G,\varepsilon) \leqslant (o+G,\varepsilon)$$

$$(o+G,\varepsilon) \leqslant (o+G,\tilde{t})$$

$$(o+G,\tilde{t}') \leqslant (o+G,\tilde{t}'') \quad \text{if and only if} \quad \tilde{t}' \leqslant \tilde{t}'',$$

moreover, $\tilde{t} \leqslant \tilde{t}'$ if for every partition $\tilde{t}_1 :: \tilde{t}_2$ of $\tilde{t}$ we can find a partition $\tilde{t}'_1 :: \tilde{t}'_2$ of $\tilde{t}'$ such that $\tilde{t}_1 \leqslant \tilde{t}'_1$ and $\tilde{t}_2 \leqslant -\bigsqcup \delta^{\mathbb{DT}}(\tilde{t}_1) \boxdot \tilde{t}'_2$.

Note that $\leqslant$ is not a partial ordering relation on $\mathsf{Seq}(\mathbb{DT})$. For instance, in general it is not reflexive. However, it is a partial ordering relation on $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T})) \subseteq \mathsf{Seq}^+(\mathbb{DT})$:

Proposition 37. *The relation* $\leqslant$ *of Definition* 36 *is a partial ordering relation on* $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))$.

Note that $\leqslant$ on $\alpha_{\mathbb{DT}}(\mathbb{T}) \subseteq \alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))$ admits many distinct minimal elements. Namely, every decorated tree of the form $(true_{\mathcal{O}} + G, \varepsilon)$ is minimal on $\alpha_{\mathbb{DT}}(\mathbb{T})$.

We lift $\leqslant$ to a partial ordering on decorated tree interpretations by defining $I_1 \leqslant I_2$ if and only if for every predicate symbol p we have $I_1(\mathrm{p}) \leqslant I_2(\mathrm{p})$. Since, if I is a decorated tree interpretation then $I(\mathrm{p})$ is a decorated tree for $\mathrm{p}(\alpha)$, we conclude that $\leqslant$ on decorated tree interpretations admits exactly one minimal element, which is indeed the bottom element of the partial ordering on decorated tree interpretations. This element will be denoted by $I_0^{\mathbb{DT}}$ and is such that

$$I_0^{\mathbb{DT}}(\mathrm{p}) = (true_{\mathcal{O}} + \mathrm{p}(\alpha), \varepsilon).$$

In the completion of the domain of decorated tree interpretations we can define

$$\mathcal{S}_P^{\mathbb{DT}} = \mathsf{lub}_{i \geqslant 0}\left(T_P^{\mathbb{DT}}\right)^i.$$

We want to relate the tree semantics $\mathcal{S}_P^{\mathbb{T}}$ with the decorated tree semantics $\mathcal{S}_P^{\mathbb{DT}}$. This will be done through the classical methodology of abstract interpretation.

It turns out that the extension of $\alpha_{\mathbb{DT}}$ on tree interpretations is strict (see Definition 15 for the definition of $I_0^{\mathbb{T}}$):

$$\alpha_{\mathbb{DT}}(I_0^{\mathbb{T}})(\mathrm{p}) = \alpha_{\mathbb{DT}}(I_0^{\mathbb{T}}(\mathrm{p})) = \alpha_{\mathbb{DT}}(\mathrm{p}(\alpha),\varepsilon) = (true_{\mathcal{O}} + \mathrm{p}(\alpha),\varepsilon) = I_0^{\mathbb{DT}}[\mathrm{p}]. \qquad (27)$$

We prove now that $\alpha_{\mathbb{DT}}$ is monotonic on sequences of trees. This will imply that it is monotonic on tree interpretations too.

Proposition 38. *Let* $\tilde{t}_1, \tilde{t}_2 \in \mathsf{Seq}(\mathbb{T})$ *be such that* $\tilde{t}_1 \subseteq \tilde{t}_2$. *We have* $\alpha_{\mathbb{DT}}(\tilde{t}_1) \leqslant \alpha_{\mathbb{DT}}(\tilde{t}_2)$.

Note that, as a simple consequence of Definition 36, the converse of Proposition 38 holds, i.e., if $\alpha_{\mathbb{DT}}(\tilde{t}_1) \leqslant \alpha_{\mathbb{DT}}(\tilde{t}_2)$ then $\tilde{t}_1 \subseteq \tilde{t}_2$. Therefore, we conclude that

$(\mathsf{Seq}(\mathbb{T}), \subseteq)$ is isomorphic to $(\alpha_{\mathbb{DT}}(\mathsf{Seq}(\mathbb{T})), \leqslant)$. This allows us to conclude that $T_P^{\mathbb{DT}}$ is monotonic w.r.t. $\leqslant$:

Proposition 39. *If $I_1 \leqslant I_2$ are two decorated tree interpretations such that $I_1 = \alpha_{\mathbb{DT}}(I'_1)$ and $I_2 = \alpha_{\mathbb{DT}}(I'_2)$, where I'_1 and I'_2 are tree interpretations, then $T_P^{\mathbb{DT}}(I_1) \leqslant T_P^{\mathbb{DT}}(I_2)$.*

Proof.

$$\begin{aligned} I_1 \leqslant I_2 &\Rightarrow I'_1 \subseteq I'_2 \\ (\text{Proposition 20}) &\Rightarrow T_P^{\mathbb{T}}(I'_1) \subseteq T_P^{\mathbb{T}}(I'_2) \\ (\text{Proposition 38}) &\Rightarrow \alpha_{\mathbb{DT}}(T_P^{\mathbb{T}}(I'_1)) \leqslant \alpha_{\mathbb{DT}}(T_P^{\mathbb{T}}(I'_2)) \\ (\text{Theorem 35}) &\Rightarrow T_P^{\mathbb{DT}}(I_1) \leqslant T_P^{\mathbb{DT}}(I_2). \end{aligned}$$

Now we know that $T_P^{\mathbb{T}}$ is monotonic w.r.t. $\subseteq$ (Proposition 20), that $T_P^{\mathbb{DT}}$ is monotonic w.r.t. $\leqslant$ (Proposition 39), that $\alpha_{\mathbb{DT}}$ is monotonic from $\subseteq$ into $\leqslant$ (Proposition 38), that $\alpha_{\mathbb{DT}}$ is strict (Eq. (27)) and that $T_P^{\mathbb{DT}}$ is correct w.r.t. $T_P^{\mathbb{T}}$ (Theorem 35). From the theory we conclude that on the completions of the posets of tree interpretations and of decorated tree interpretations we have:

Theorem 40. *Given a program P, the following equalities hold*:

$$\alpha_{\mathbb{DT}}(\mathscr{S}_P^{\mathbb{T}}) = \alpha_{\mathbb{DT}}\big(\mathtt{lfp}\ (T_P^{\mathbb{T}})\big) = \mathtt{lfp}\ T_P^{\mathbb{DT}} = \mathsf{lub}_i\big(T_P^{\mathbb{DT}}\big)^i = \mathscr{S}_P^{\mathbb{DT}}.$$

According to the above theorem, the semantics $\mathscr{S}_P^{\mathbb{DT}}$ defined by the operators presented in Section 7.1 is the abstraction by $\alpha_{\mathbb{DT}}$ of the tree semantics described in Section 5. The abstraction compiles the cut and divergence conditions.

The semantics $\mathscr{S}_P^{\mathbb{DT}}$ is the starting point for further observable-specific abstractions.

8. Denotational semantics for computed answers

In this section we show how to obtain a computed answer semantics as an abstraction of the semantics defined in the last section into a computed answer semantics.

Consider a decorated tree $T \in \mathbb{DT}$ (or, equivalently, the abstraction through $\alpha_{\mathbb{DT}}$ of a tree $T \in \mathbb{T}$). If we are interested in the set of its computed answers, we can abstract this tree by collecting the set of convergent leaves whose observability is true. However, we want a compositional semantics. Hence we have to select all the convergent leaves. This is because a false observability constraint like $-(x = 4)$ can become true if we instantiate x. For instance, $(x = 5) \bullet (-(x = 4)) = -(x = 5 \wedge x = 4)$ which is true. Consider the semantical operators in Definition 33. The observability constraints are modified by the product operation (Eq. (16)) through the function $\xi^{\mathbb{DT}}$ which (Eq. (17)) needs the convergent leaves of a tree and the block condition $\beta^{\mathbb{DT}}$ of trees. Moreover, the ! operation (Eq. (19)) is defined through the convergence condition $\mathrm{o}^{\mathbb{DT}}$. This condition can be easily recovered from the set of convergent leaves of a tree (see its definition given by Eq. (15)). Finally, the sum operation (Eq. (20)) is defined through the block condition of trees. In conclusion, our abstraction of a decorated tree must be concrete enough to allow for the definition of a block condition. Hence we have to select the divergent leaves of a decorated tree, as well as the cut conditions of a decorated tree. Since cuts have been *compiled*, we only need the cut conditions whose scope is open, i.e., the cut() constructs.

Every node of a decorated tree is formed by an observability constraint and by a goal. The abstraction of this goal can give rise to a convergent constraint, corresponding to a leaf which does not contain procedure calls, to a divergent constraint, corresponding to a leaf containing procedure calls, and to cut constraints, corresponding to cut conditions with an open scope. Note that cut constraints can derive from the internal nodes as well as from the leaves of a tree. The observability constraint of a node is maintained in its abstraction. Hence every node $o + G$ is abstracted into a set of *conditional constraints* $o + b$, where $b \in \mathscr{B}$ is the basic constraint obtained from G, as we will explain later. Depending on the kind of information that a conditional constraint represents, we have three kinds of conditional constraints:

- $\mathscr{CA}^c = \{o +^c b \mid o \in \mathscr{O},\ b \in \mathscr{B}\}$ (convergent conditional constraints);
- $\mathscr{CA}^d = \{o +^d b \mid o \in \mathscr{O},\ b \in \mathscr{B}\}$ (divergent conditional constraints);
- $\mathscr{CA}^! = \{o +^! b \mid o \in \mathscr{O},\ b \in \mathscr{B}\}$ (cut conditional constraints).

In the following, we will drop the word *conditional* and we will simply talk of constraints, rather than conditional constraints. Note that this does not introduce any confusion with basic constraints $b \in \mathscr{B}$ and observability constraints $o \in \mathscr{O}$. We define $\mathbb{CA} = \mathscr{CA}^c \cup \mathscr{CA}^d \cup \mathscr{CA}^!$. We will use two selectors on $\mathbb{CA}$: given $c \in \mathbb{CA}$, $|c|_1$ is the observability constraint of c, while $|c|_2$ is the basic constraint of c.

Every decorated tree will be abstracted into a sequence of constraints in $\mathbb{CA}$. Note that we use sequences rather than sets of constraints. This is usual when one takes control into account. Actually, sequences allow us to represent the order in which the nodes of an SLD tree are visited by an interpreter.

The cut constraints contained in a goal can be extracted through the function cut of Definition 9. However, we need only the cut conditions with an open scope (the others are assumed to have been compiled). Moreover, we need a sequence of constraints rather than a set of cut conditions. Therefore, we define a map cutsseq which is very similar to the cuts map of Definition 9, except for the considerations above. Formally:

Definition 41. Given a goal G, the sequence of its cut constraints is defined as

$$\mathsf{cutsseq}(c) = \mathsf{cutsseq}(\mathsf{p}(x)) = \varepsilon$$

$$\mathsf{cutsseq}(G_1 \ \mathtt{and}\ G_2) = \begin{cases} \mathsf{cutsseq}(G_1)::\mathsf{con}(G_1)\circledcirc\mathsf{cutsseq}(G_2) \\ \quad \text{if not } \mathsf{div}(G_1) \\ \\ \mathsf{cutsseq}(G_1) \\ \quad \text{if } \mathsf{div}(G_1) \end{cases}$$

$$\mathsf{cutsseq}(\mathtt{exists}\ x.G) = \exists\!\!\!\exists_x \mathsf{cutsseq}(G)$$

$$\mathsf{cutsseq}(\mathtt{cut}_d(G)) = \mathsf{cutsseq}(G)$$

$$\mathsf{cutsseq}(\mathtt{cut}(G)) = \begin{cases} \mathsf{cutsseq}(G)::\mathit{true}_{\mathscr{O}} +^! \mathsf{con}(G) & \text{if not } \mathsf{div}(G) \\ \mathsf{cutsseq}(G) & \text{if } \mathsf{div}(G), \end{cases}$$

where

$$\begin{aligned}\exists\!\!\exists_x(o +^c b) &= \exists_x o +^c \exists_x b\\ \exists\!\!\exists_x(o +^d b) &= \exists_x o +^d \exists_x b\\ \exists\!\!\exists_x(o +^! b) &= \exists_x o +^! \exists_x b\\ \exists\!\!\exists_x(\tilde{s}_1 :: \tilde{s}_2) &= (\exists\!\!\exists_x \tilde{s}_1) :: (\exists\!\!\exists_x \tilde{s}_2).\end{aligned} \tag{28}$$

and, given $b \in \mathscr{B}$:

$$\begin{aligned}b \circledcirc (o +^c b') &= (b \bullet o) +^c (b \wedge b')\\ b \circledcirc (o +^d b') &= (b \bullet o) +^d (b \wedge b')\\ b \circledcirc (o +^! b') &= (b \bullet o) +^! (b \wedge b')\\ b \circledcirc (\tilde{s}_1 :: \tilde{s}_2) &= (b \circledcirc \tilde{s}_1) :: (b \circledcirc \tilde{s}_2).\end{aligned} \tag{29}$$

Note that the constructs $\mathtt{cut}_d()$ do not give rise to cut constraints, since we assume that they have been compiled. Moreover, note that the observability part of the constraint generated by the cutsseq map is always $true_{\mathscr{O}}$. Now we are able to define the abstraction of a sequence of decorated trees.

Definition 42. We define a map $\alpha_{\mathbb{CA}} : \mathsf{Seq}^+(\mathbb{DT}) \mapsto \mathsf{Seq}^+(\mathbb{CA})$ as follows:

$$\begin{aligned}\alpha_{\mathbb{CA}}(o + G, \varepsilon) &= \begin{cases} o \odot \mathsf{cutsseq}(G) :: (o +^d \mathsf{con}(G)) & \text{if } \mathsf{div}(G)\\ o \odot \mathsf{cutsseq}(G) :: (o +^c \mathsf{con}(G)) & \text{if not } \mathsf{div}(G)\end{cases}\\ \alpha_{\mathbb{CA}}(o + G, \tilde{t}) &= o \odot \mathsf{cutsseq}(G) :: \alpha_{\mathbb{CA}}(\tilde{t})\\ \alpha_{\mathbb{CA}}(\tilde{t}_1 :: \tilde{t}_2) &= \alpha_{\mathbb{CA}}(\tilde{t}_1) :: \alpha_{\mathbb{CA}}(\tilde{t}_2),\end{aligned} \tag{30}$$

where, given $o \in \mathscr{O}$:

$$\begin{aligned}o \odot (o' +^c b) &= (o \sqcap o' +^c b)\\ o \odot (o' +^d b) &= (o \sqcap o' +^d b)\\ o \odot (o' +^! b) &= (o \sqcap o' +^! b)\\ o \odot (\tilde{s}_1 :: \tilde{s}_2) &= (o \odot \tilde{s}_1) :: (o \odot \tilde{s}_2).\end{aligned} \tag{31}$$

Example 43. Consider the decorated tree shown in Fig. 8. Its abstraction through the $\alpha_{\mathbb{CA}}$ map is the sequence of constraints

$$\begin{gathered} true_{\mathscr{O}} +^c (x = 3 \wedge x = 5) :: -(x = 3) +^d true :: -(-(x = 3)) +^d x = 6 ::\\ :: (-(-(x = 3)) \sqcap -(-(-(x = 3)) \sqcap x = 6)) +^! x = 2 ::\\ :: (-(-(x = 3)) \sqcap -(-(-(x = 3)) \sqcap x = 6)) +^c x = 2.\end{gathered}$$

Note that the cut condition $\mathtt{cut_1(x = 3)}$ is not contained in the above sequence. Since its scope is closed, it has been *compiled* by the $\alpha_{\mathbb{DT}}$ map. Therefore, we do not need it in the abstraction through $\alpha_{\mathbb{CA}}$. The cut condition $\mathtt{cut}(x = 2)$ has an open scope. Hence we maintain it in the abstraction by $\alpha_{\mathbb{CA}}$. Note that that goal gives rise both to a cut constraint and to a convergent constraint.

We want now to define the abstract counterparts, w.r.t. the $\alpha_{\mathbb{CA}}$ map, of the operators defined in Section 7.1. We already defined $\odot$, the abstract counterpart of $\boxdot$

(Eq. (31)), $\circledcirc$, the abstract counterpart of $\boxed{\circ}$ (Eq. (29)) and $\exists\!\!\exists_x$, the abstract counterpart of $\exists\!\!\exists_x$ on $\mathsf{Seq}(\mathbb{DT})$ (Eq. (28)). The abstract counterparts of the conditions and of the other operators are defined below:

Definition 44. Given $\tilde{s}, \tilde{s}_1, \tilde{s}_2 \in \mathsf{Seq}(\mathbb{CA})$ we define

$$\begin{aligned}
\mathsf{o}(\tilde{s}) &= \bigsqcup_{o+^c b \in \tilde{s}} (o \sqcap b \ \propto \ obs),\\
\delta(\tilde{s}) &= \bigsqcup_{o+^d b \in \tilde{s}} (o \sqcap b \ \propto \ obs),\\
\kappa(\tilde{s}) &= \bigsqcup_{o+^! b \in \tilde{s}} (o \sqcap b \ \propto \ obs),\\
\beta(\tilde{s}) &= \delta(\tilde{s}) \sqcup \kappa(\tilde{s}),\\
\xi(\tilde{s}_1, \tilde{s}_2) &= \bigsqcup_{o+^c b \in \tilde{s}_1} (o \sqcap (b \bullet \beta(\tilde{s}_2))),
\end{aligned}$$

$$\begin{aligned}
&(o +^c b) \otimes \tilde{s} = o \odot (b \circledcirc \tilde{s})\\
&(o +^d b) \otimes \tilde{s} = o +^d b\\
&(o +^! b) \otimes \tilde{s} = o +^! b\\
&(\tilde{s}_1 :: \tilde{s}_2) \otimes \tilde{s} = (\tilde{s}_1 \otimes \tilde{s}) :: -\xi(\tilde{s}_1, \tilde{s}) \odot (\tilde{s}_2 \otimes \tilde{s}),
\end{aligned}$$

$$\begin{aligned}
&!(o +^c b) = (o +^! b) :: (o +^c b)\\
&!(o +^d b) = o +^d b\\
&!(o +^! b) = o +^! b\\
&!(\tilde{s}_1 :: \tilde{s}_2) = !\tilde{s}_1 :: -\mathsf{o}^{\mathbb{DT}}(\tilde{s}_1) \odot !\tilde{s}_2,
\end{aligned}$$

$$\begin{aligned}
&¡(o +^c b) = o +^c b\\
&¡(o +^d b) = o +^d b\\
&¡(o +^! b) = \varepsilon\\
&¡(\tilde{s}_1 :: \tilde{s}_2) = ¡(\tilde{s}_1) :: ¡(\tilde{s}_2),
\end{aligned}$$

$$\begin{aligned}
\phi^G(\tilde{s}) &= \mathsf{cutsseq}(G) :: \tilde{s},\\
\psi^G(\tilde{s}) &= \mathsf{cutsseq}(G) :: \tilde{s},\\
\tilde{s}_1 \oplus \tilde{s}_2 &= \tilde{s}_1 :: -\beta(\tilde{s}_1) \odot \tilde{s}_2,\\
[x/\alpha]\tilde{s} &= \exists\!\!\exists_\alpha(\delta_{\alpha,x} \circledcirc \tilde{s}).
\end{aligned}$$

All these operators are correct w.r.t. the ones defined in Definition 33.

Proposition 45. *Given* $\tilde{t} \in \mathsf{Seq}(\mathbb{DT})$, *we have*

(i) $\bigsqcup \mathsf{o}^{\mathbb{DT}}(\tilde{t}) = \mathsf{o}(\alpha_{\mathbb{CA}}(\tilde{t}))$;
(ii) $\bigsqcup \delta^{\mathbb{DT}}(\tilde{t}) = \delta(\alpha_{\mathbb{CA}}(\tilde{t}))$;
(iii) *if* $\tilde{t}$ *is formed by i-cut-closed decorated trees, then* $\bigsqcup \varsigma^i \kappa^{\mathbb{DT}}(\tilde{t}) = \kappa(\alpha_{\mathbb{CA}}(\tilde{t}))$;
(iv) *if* $\tilde{t}$ *is formed by i-cut-closed decorated trees, then* $\bigsqcup \varsigma^i \beta^{\mathbb{DT}}(\tilde{t}) = \beta(\alpha_{\mathbb{CA}}(\tilde{t}))$;

(v) *given* $o \in \mathcal{O}$ *we have* $\alpha_{\mathbb{CA}}(o \boxdot \tilde{t}) = o \odot \alpha_{\mathbb{CA}}(\tilde{t})$;
(vi) $\alpha_{\mathbb{CA}}(b \boxed{\circ} \tilde{t}) = b \circledcirc \alpha_{\mathbb{CA}}(\tilde{t})$;
(vii) $\alpha_{\mathbb{CA}}(\boxed{\exists}_x \tilde{t}) = \boxed{\exists}_x \, \alpha_{\mathbb{CA}}(\tilde{t})$;
(viii) *given* $T \in \mathbb{DT}$ *cut-closed, we have* $\bigsqcup \xi^{\mathbb{DT}}(\tilde{t}, T) = \xi(\alpha_{\mathbb{CA}}(\tilde{t}), \alpha_{\mathbb{CA}}(T))$;
(ix) *given* $T \in \mathbb{DT}$ *cut-closed, we have* $\alpha_{\mathbb{CA}}(\tilde{t} \boxtimes T) = \alpha_{\mathbb{CA}}(\tilde{t}) \otimes \alpha_{\mathbb{CA}}(T)$;
(x) $\alpha_{\mathbb{CA}}(!\tilde{t}) = !\alpha_{\mathbb{CA}}(\tilde{t})$;
(xi) *given* $i \geqslant 0$, *we have* $\alpha_{\mathbb{CA}}(¡_i \tilde{t}) = ¡\alpha_{\mathbb{CA}}(\tilde{t})$;
(xii) $\alpha_{\mathbb{CA}}(\phi^G(T)) = \phi^G(\alpha_{\mathbb{CA}}(T))$;
(xiii) *given* $T \in \mathbb{DT}$, *whose root* $true_{\mathcal{O}} + G$ *is such that* $\mathtt{cutsseq}(G) = \varepsilon$ *and whose height is at least* 2, *and a goal* $G' \in \mathbb{G}$, *we have* [7] $\alpha_{\mathbb{CA}}(\psi^{G'}(T)) = \psi^{G'}(\alpha_{\mathbb{CA}}(T))$;
(xiv) *given* $T_1 = (o + G, \tilde{t}_1)$ *and* $T_2 = (o + G, \tilde{t}_2)$ *on* $\mathbb{DT}$ *such that* $\mathtt{cutsseq}(G) = \varepsilon$, *we have* [8]: $\alpha_{\mathbb{CA}}(T_1 \boxplus T_2) = \alpha_{\mathbb{CA}}(T_1) \oplus \alpha_{\mathbb{CA}}(T_2)$;
(xv) *given* $T \in \mathbb{DT}$ *whose root is* $true_{\mathcal{O}} + \mathtt{p}\alpha$, *we have* [9] $\alpha_{\mathbb{CA}}(T[x/\alpha]) = \alpha_{\mathbb{CA}}(T)[x/\alpha]$.

Note that, for our purposes (Eq. (26)), the functions ϕ^G and ψ^G are always called with $G = \mathtt{p}(x)$ for some variable x. Since $\mathtt{cutsseq}(\mathtt{p}(x)) = \varepsilon$, Proposition 45 tells us that in situations in which the concrete operators ϕ^G and ψ^G are used, their abstract counterparts coincide with the identity function on $\mathsf{Seq}(\mathbb{CA})$.

The abstract immediate consequence operator induced by the $\alpha_{\mathbb{CA}}$ map is obtained from the $T_P^{\mathbb{DT}}$ operator of Eq. (26) substituting the abstract operators for the concrete ones:

$$T_P^{\mathbb{CA}}(I)(\mathtt{p}) = \boxed{\exists}_y \left(\delta_{y,\alpha} \circledcirc ¡\left(\mathcal{T}_P^{\mathbb{CA}}[\![G_1]\!]I \oplus \cdots \oplus \mathcal{T}_P^{\mathbb{CA}}[\![G_n]\!]I\right)\right) \tag{32}$$

for every predicate symbol $\mathtt{p}$, where I is a computed answer interpretation, i.e., a map from predicate symbols into $\mathsf{Seq}(\mathbb{CA})$, and

$$\begin{aligned}
\mathcal{T}_P^{\mathbb{CA}}[\![c]\!]I &= (true_{\mathcal{O}} + c)\\
\mathcal{T}_P^{\mathbb{CA}}[\![G_1 \ \mathtt{and}\ G_2]\!]I &= \mathcal{T}_P^{\mathbb{CA}}[\![G_1]\!]I \otimes \mathcal{T}_P^{\mathbb{CA}}[\![G_2]\!]I\\
\mathcal{T}_P^{\mathbb{CA}}[\![\mathtt{exists}\ x.G]\!]I &= \boxed{\exists}_x \mathcal{T}_P^{\mathbb{CA}}[\![G]\!]I\\
\mathcal{T}_P^{\mathbb{CA}}[\![\mathtt{p}(x)]\!]I &= I(\mathtt{p})[x/\alpha]\\
\mathcal{T}_P^{\mathbb{CA}}[\![\mathtt{cut}(G)]\!]I &= !\mathcal{T}_P^{\mathbb{CA}}[\![G]\!]I.
\end{aligned} \tag{33}$$

A straightforward consequence of Proposition 45 is the following theorem:

Theorem 46. *Let G be a goal, P a program and I a strongly P-directed decorated tree interpretation. We have*
(i) $\alpha_{\mathbb{CA}}(\mathcal{T}_P^{\mathbb{DT}}[\![G]\!]I) = \mathcal{T}_P^{\mathbb{CA}}[\![G]\!]\alpha_{\mathbb{CA}}(I)$,
(ii) $\alpha_{\mathbb{CA}}(T_P^{\mathbb{DT}}(I)) = T_P^{\mathbb{CA}}(\alpha_{\mathbb{CA}}(I))$,
where $\alpha_{\mathbb{CA}}(I)[\mathtt{p}] = \alpha_{\mathbb{CA}}(I(\mathtt{p}))$.

[7] Actually, this is the only way in which we use the ψ^G operation (see Eq. (26). We use the function ψ^G with trees whose root is $true_{\mathcal{O}} + \mathtt{exists}\ y.\delta_{y,\alpha}\ \mathtt{and}\ \mathtt{p}(y)$ and whose height is at least 2).

[8] This is the way we use the $\boxplus$ operator (see Eq. (26)).

[9] Note that the condition on T is not restrictive since we apply the substitution operator only on trees whose root is $true_{\mathcal{O}} + \mathtt{p}(\alpha)$ (see Eqs. (21)–(23)).

In order to prove that the semantics defined in this section is an abstract interpretation of the semantics defined on decorated trees in Section 7.1, we need some more results.

We first show that the partial ordering on $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))$ of Definition 36 induces a partial ordering on $\mathsf{Seq}^+(\mathbb{CA})$ which can be lifted to a partial ordering on computed answer interpretations. The bottom element of such a lifted partial ordering will be the computed answer interpretation $I_0^{\mathbb{CA}}$ which behaves as

$$I_0^{\mathbb{CA}}(\mathsf{p}) = (true_{\mathcal{O}} +^d true).$$

We want a partial ordering $\sqsubseteq$ on $\mathsf{Seq}(\mathbb{CA})$ such that $\alpha_{\mathbb{CA}}$ is monotonic from $\alpha_{\mathbb{DT}}(\mathsf{Seq}(\mathbb{T}))$ into $\mathsf{Seq}(\mathbb{CA})$. Therefore, consider a decorated tree $(true_{\mathcal{O}} + G, \varepsilon)$ with not $\mathsf{div}(G)$. If $(true_{\mathcal{O}} + G, \varepsilon) \leqslant \tilde{t}$, we must have $\tilde{t} = (true_{\mathcal{O}} + G, \varepsilon)$. This suggests that we must define $\sqsubseteq$ in such a way that $\alpha_{\mathbb{CA}}(true_{\mathcal{O}} + G, \varepsilon) \sqsubseteq \alpha_{\mathbb{CA}}(true_{\mathcal{O}} + G, \varepsilon)$, i.e., in such a way that $\mathsf{cutsseq}(G) :: (true_{\mathcal{O}} +^c \mathsf{con}(G)) \sqsubseteq \mathsf{cutsseq}(G) :: (true_{\mathcal{O}} +^c \mathsf{con}(G))$. Therefore, let

1. $o +^! b \sqsubseteq o +^! b$,
2. $o +^c b \sqsubseteq o +^c b$,
3. $\tilde{s} \sqsubseteq \tilde{s}'$ if for every partition $\tilde{s}_1 :: \tilde{s}_2$ of $\tilde{s}$ there exists a partition $\tilde{s}'_1 :: \tilde{s}'_2$ of $\tilde{s}'$ such that $\tilde{s}_1 \sqsubseteq \tilde{s}'_1$ and $\tilde{s}_2 \sqsubseteq \tilde{s}'_2$.

If $\mathsf{div}(G)$, we have $(true_{\mathcal{O}} + G, \varepsilon) \leqslant (true_{\mathcal{O}} + G, \varepsilon)$ as well as $(true_{\mathcal{O}} + G, \varepsilon) \leqslant (true_{\mathcal{O}} + G, \tilde{t})$. Therefore, $\sqsubseteq$ must be defined in such a way that $\mathsf{cutsseq}(G) :: (true_{\mathcal{O}} +^d \mathsf{con}(G)) \sqsubseteq \mathsf{cutsseq}(G) :: (true_{\mathcal{O}} +^d \mathsf{con}(G))$, as well as that $\mathsf{cutsseq}(G) :: (true_{\mathcal{O}} +^d \mathsf{con}(G)) \sqsubseteq \mathsf{cutsseq}(G) :: \alpha_{\mathbb{CA}}(\tilde{t})$. Using the following lemma:

Lemma 47. *Let G be a divergent goal and let $1 \leqslant i \leqslant \mathtt{choices}(G, P)$. We have $\mathsf{con}(G) \geqslant \mathsf{con}\imath(G, P, i)$ and $\mathsf{con}(G) \geqslant \bigsqcup \mathtt{cuts}(\imath(G, P, i))$.*

We conclude that the relation we need is

4. $o +^d b \leqslant \tilde{s}$ if and only if every $o' +^c b'$ (or $o' +^d b'$ or $o' +^! b'$) in $\tilde{s}$ is such that $o \geqslant o'$ and $b \geqslant b'$.

Consider now $\tilde{t}, \tilde{t}' \in \mathsf{Seq}(\mathbb{DT})$ such that $\tilde{t} \leqslant \tilde{t}'$. This means that $\tilde{t} = \tilde{t}_1 :: \tilde{t}_2$ and $\tilde{t}' = \tilde{t}'_1 :: \tilde{t}'_2$ with $\tilde{t}_1 \leqslant \tilde{t}'_1$ and $\tilde{t}_2 \leqslant -\bigsqcup \delta^{\mathbb{DT}}(\tilde{t}_1) \boxdot \tilde{t}'_2$. We must define $\sqsubseteq$ in such a way that $\alpha_{\mathbb{CA}}(\tilde{t}) = \alpha_{\mathbb{CA}}(\tilde{t}_1) :: \alpha_{\mathbb{CA}}(\tilde{t}_2) \sqsubseteq \alpha_{\mathbb{CA}}(\tilde{t}'_1) :: \alpha_{\mathbb{CA}}(\tilde{t}'_2) = \alpha_{\mathbb{CA}}(\tilde{t}')$. Since $\alpha_{\mathbb{CA}}(-\bigsqcup \delta^{\mathbb{DT}}(\tilde{t}_1) \boxdot \tilde{t}'_2) = -\delta\ (\alpha_{\mathbb{CA}}(\tilde{t}_1)) \odot \alpha_{\mathbb{CA}}(\tilde{t}_2)$ (by Proposition 45, points (ii) and (v)), we conclude that the relation we need is

3′. $\tilde{s} \sqsubseteq \tilde{s}'$ if for every partition $\tilde{s}_1 :: \tilde{s}_2$ of $\tilde{s}$ there exists a partition $\tilde{s}'_1 :: \tilde{s}'_2$ of $\tilde{s}'$ such that $\tilde{s}_1 \sqsubseteq \tilde{s}'_1$ and $\tilde{s}_2 \sqsubseteq -\delta(\tilde{s}_1) \odot \tilde{s}'_2$.

Note that condition 3 can be safely substituted with condition 3′. This is because condition 3′ entails that

$$\mathsf{cutsseq}(G) :: (true_{\mathcal{O}} +^c \mathsf{con}(G)) \sqsubseteq \mathsf{cutsseq}(G) :: (true_{\mathcal{O}} +^c \mathsf{con}(G)),$$

since $\delta(\mathtt{cutsseq}(G)) = false_{\mathcal{O}}$.

The definition below formalises our ideas.

Definition 48. We define the relation $\sqsubseteq$ on $\mathsf{Seq}(\mathbb{CA})$ as the minimal relation on $\mathsf{Seq}(\mathbb{CA})$ such that:

1. $o +^c b \sqsubseteq o +^c b$;
2. $o +^d b \sqsubseteq \tilde{s}$ if and only if every $o' +^c b'$ and $o' +^d b'$ and $o' +^! b'$ in $\tilde{s}$ is such that $o \geqslant o'$ and $b \geqslant b'$;
3. $o +^! b \sqsubseteq o +^! b$;
4. $\tilde{s} \sqsubseteq \tilde{s}'$ if for every partition $\tilde{s}_1 :: \tilde{s}_2$ of $\tilde{s}$ we can find a partition $\tilde{s}'_1 :: \tilde{s}'_2$ of $\tilde{s}'$ such that $\tilde{s}_1 \sqsubseteq \tilde{s}'_1$ and $\tilde{s}_2 \sqsubseteq -\delta(\tilde{s}_1) \odot \tilde{s}'_2$.

$\sqsubseteq$ is a partial ordering relation on $\alpha_{\mathbb{CA}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T})))$, as we are going to show. Note that this is sufficient for our purposes, since the operators on non-empty sequences of decorated trees are the abstraction through $\alpha_{\mathbb{DT}}$ of the operators on non-empty sequences of trees and the operators on non-empty sequences of conditional constraints are the abstraction through $\alpha_{\mathbb{CA}}$ of the operators on non-empty decorated trees.

We first need the following lemma:

Lemma 49. *Given* $\tilde{s} \in \alpha_{\mathbb{CA}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T})))$ *such that* $\tilde{s} = \tilde{s}_1 :: \tilde{s}_2$, *we have* $\tilde{s}_2 = -\delta(\tilde{s}_1) \odot \tilde{s}_2$.

A non-empty sequence $\tilde{s} \in \mathsf{Seq}^+(\mathbb{CA})$, such that every partition $\tilde{s}_1 :: \tilde{s}_2$ is such that $\tilde{s}_2 = -\delta(\tilde{s}_1) \odot \tilde{s}_2$, will be called *well formed.*

Proposition 50. $\sqsubseteq$ *is a partial ordering relation on the set of well formed sequences* (*and therefore*, *by Lemma* 49, *on* $\alpha_{\mathbb{CA}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T})))$).

The minimum of such a partial ordering is the sequence $true_{\emptyset} +^d true$, as it is easy to check. The $\sqsubseteq$ relation on sequences of constraints is lifted to a $\sqsubseteq$ relation on computed answer interpretations as $I_1 \sqsubseteq I_2$ if and only if for every predicate symbol p we have $I_1(\mathsf{p}) \sqsubseteq I_2(\mathsf{p})$. In the following, we will always consider computed answer interpretations which give for every predicate symbol a sequence in $\alpha_{\mathbb{CA}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}(\mathbb{T})))$. This way, $\sqsubseteq$ becomes a partial ordering relation on this subset of computed answer interpretations. The minimum of such a partial ordering is the computed answer interpretation $I_0^{\mathbb{CA}}$ which is such that $I_0^{\mathbb{CA}}(\mathsf{p}) = true_{\emptyset} +^d true$ for every predicate symbol p. $\alpha_{\mathbb{CA}}$ is strict:

$$\alpha_{\mathbb{CA}}(I_0^{\mathbb{DT}}(\mathsf{p})) = \alpha_{\mathbb{CA}}(true_{\emptyset} + \mathsf{p}(\alpha), \varepsilon) = true_{\emptyset} +^d true = I_0^{\mathbb{CA}}(\mathsf{p}), \tag{34}$$

for every predicate symbol p. We show now that $\alpha_{\mathbb{CA}}$ is monotonic:

Proposition 51. *Let* $\tilde{t}_1, \tilde{t}_2 \in \mathsf{Seq}(\mathbb{DT})$ *be two strongly P-directed decorated trees such that the observability condition of a node is greater than the observability conditions of its children and such that* $\tilde{t}_1 \leqslant \tilde{t}_2$. *We have* $\alpha_{\mathbb{CA}}(\tilde{t}_1) \sqsubseteq \alpha_{\mathbb{CA}}(\tilde{t}_2)$.

Since every sequence in $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))$ is such that the observability condition of a node is greater than the observability conditions of its children (see the definition of $\alpha_{\mathbb{DT}}$ given by Eq. (14)), we conclude that $\alpha_{\mathbb{CA}}$ is monotonic on $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))$.

Consider the sequence $\{(T_P^{\mathbb{DT}})^i\}_{i \geqslant 0}$ of interpretations over $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))$. We know that it is increasing w.r.t. the relation $\leqslant$ on decorated trees of Definition 36. By the strictness of $\alpha_{\mathbb{CA}}$ (Eq. (34)) and by Theorem 46, we conclude that

$$\alpha_{\mathbb{CA}}\left(\left(T_P^{\mathbb{DT}}\right)^i\right) = \left(T_P^{\mathbb{CA}}\right)^i \quad \text{for every } i \geqslant 0.$$

By monotonicity of $\alpha_{\mathbb{CA}}$ (Proposition 51), we conclude that

$$\left(T_P^{\mathbb{CA}}\right)^i \sqsubseteq \left(T_P^{\mathbb{CA}}\right)^{i+1} \quad \text{for every } i \geqslant 0.$$

Therefore, in the completion of the partial orders $\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T}))$ and $\alpha_{\mathbb{CA}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(\mathbb{T})))$ there exists $\mathsf{lub}_i\left(T_P^{\mathbb{CA}}\right)^i$. Given a program P, we define

$$\mathscr{S}_P^{\mathbb{CA}} = \mathsf{lub}_i\left(T_P^{\mathbb{CA}}\right)^i.$$

Since the extension of $\alpha_{\mathbb{CA}}$ is continuous, we have

Theorem 52. *Given a program P, the following equalities hold*:

$$\mathscr{S}_P^{\mathbb{CA}} = \mathsf{lub}_i\left(T_P^{\mathbb{CA}}\right)^i = \mathsf{lub}_i\, \alpha_{\mathbb{CA}}\left(\left(T_P^{\mathbb{DT}}\right)^i\right) = \alpha_{\mathbb{CA}}\left(\mathsf{lub}_i\left(T_P^{\mathbb{DT}}\right)^i\right) = \alpha_{\mathbb{CA}}\left(\mathscr{S}_P^{\mathbb{DT}}\right).$$

The above result allows us to conclude that the computed answer semantics described in this section can be actually used to collect the computed answers of a program P, taking control into account:

Theorem 53. *Given a program P and a goal G, the set of consistent computed answers for G in P is given by*

$$\{b \mid b \neq \mathit{false}, o \text{ is true and } o +^c b \text{ belongs to } \mathscr{T}_P^{\mathbb{CA}}[\![G]\!]\mathscr{S}_P^{\mathbb{CA}}\}.$$

Proof. By Theorem 30 we know that the portion of the SLD tree actually visited by an interpreter which uses control rules of Prolog and the cut operator is given by the portion of $\alpha_{\mathbb{DT}}(\mathscr{T}_P^{\mathbb{DT}}[\![G]\!]\mathscr{S}_P^{\mathbb{T}})$ formed by the nodes whose observability constraint is true and whose parent is not inconsistent. By Theorems 35 and 40, this means that the set of consistent computed answers of G in P is given by the consistent leaves of $\mathscr{T}_P^{\mathbb{DT}}[\![G]\!]\mathscr{S}_P^{\mathbb{DT}}$ whose observability constraint is true. By definition of $\alpha_{\mathbb{CA}}$, this is the set of constraints $o +^c b$ which belong to $\alpha_{\mathbb{CA}}(\mathscr{T}_P^{\mathbb{DT}}[\![G]\!]\mathscr{S}_P^{\mathbb{DT}})$ and such that o is true and $b \neq \mathit{false}$. By Theorems 46 and 52 we have the thesis. □

Compare our semantics with that presented in Ref. [3]. Our approach is an evolution of theirs. The main difference is that we use an observability condition in every constraint, while in their approach the observability of a constraint depends on the constraints which are on its left in the sequence. As said before, the use of observability constraints (i.e., control compilation) is very important for doing a finite abstract analysis which consider the cut operator. Note also that Ref. [3] does not consider the cut operator.

8.1. An example

We compute the denotational computed answer semantics of the following Prolog program:

```
min([H|T],M):- min(T,M),M <= H,!.
min([H|T],H).
```

which in our syntax is written as the following program P:

$$\begin{array}{l}
\texttt{min}(x) : \texttt{-} \\
\qquad\qquad \texttt{exists } y.\texttt{exists } h.\texttt{exists } t.\texttt{exists } m. \\
\qquad\qquad (\texttt{cut}(x) = \langle [h|t], m\rangle \texttt{ and } y = \langle t, m\rangle \texttt{ and } \texttt{min}(y) \texttt{ and } m \leqslant h) \\
\qquad\qquad \texttt{or} \\
\qquad\qquad \texttt{exists } h.\texttt{exists } t.x = \langle [h|t], h\rangle.
\end{array}$$

Note that we are assuming that our basic constraint system (Definition 3) contains comparison constraints like $m \leqslant h$. [10]

We have (all variables except α must be considered existentially quantified):

$$\begin{aligned}
\left(T_P^{\mathbb{CA}}\right)^0[\texttt{min}] &= true_{\emptyset} +^d true \\
\left(T_P^{\mathbb{CA}}\right)^1[\texttt{min}] &= true_{\emptyset} +^d \alpha = \langle [h_1|t], m\rangle :: -(\alpha = \langle [h_1|t], m\rangle) +^c \alpha = \langle [h_1|t], h_1\rangle \\
\left(T_P^{\mathbb{CA}}\right)^2[\texttt{min}] &= true_{\emptyset} +^d \alpha = \langle [h_1, h_2|t], m\rangle :: \\
&\quad -(\alpha = \langle [h_1, h_2|t], m\rangle) +^c \alpha = \langle [h_1, h_2|t], h_2\rangle \wedge h_2 \leqslant h_1 :: \\
&\quad -(\alpha = \langle [h_1, h_2|t], m\rangle) \sqcap -(\alpha = \langle [h_1, h_2|t], h_2\rangle \wedge h_2 \leqslant h_1) +^c \\
&\quad +^c \alpha = \langle [h_1|t], h_1\rangle
\end{aligned}$$

and, in general, letting K_i denote $\alpha = \langle [h_1, \ldots, h_n|t], m\rangle$, we have

$$\left(T_P^{\mathbb{CA}}\right)^n[\texttt{min}] = true_{\emptyset} +^d K_n :: C_n :: \cdots :: C_1,$$

where C_i, $1 \leqslant i \leqslant n$, is the constraint:

$$\begin{aligned}
&-K_i \sqcap \sqcap_{j=i+1,\ldots,n} - \left(\alpha = \langle [h_1, \ldots, h_j|t], h_j\rangle \wedge \wedge h_j = \min_{s=1,\ldots,j} h_s\right) +^c \\
&+^c \alpha = \langle [h_1, \ldots, h_i|t], h_i\rangle \wedge h_i = \min_{s=1\ldots i} h_s.
\end{aligned}$$

We conclude that

$$\mathscr{S}_P^{\mathbb{CA}}[\texttt{min}] = \cdots :: C'_n :: \cdots :: C'_1,$$

where C'_i, $i \geqslant 1$, is the constraint:

$$\begin{aligned}
&\sqcap_{j=i+1,\ldots,n} - \left(\alpha = \langle [h_1, \ldots, h_j|t], h_j\rangle \wedge h_j = \min_{s=1,\ldots,j} h_s\right) +^c \\
&+^c \alpha = \langle [h_1, \ldots, h_i|t], h_i\rangle \wedge h_i = \min_{s=1\ldots i} h_s.
\end{aligned}$$

Roughly speaking, the constraint C'_i says that (right hand side of the $+^c$ separator) the ith element (from the head) of a list is selected as the list minimum if and only if the list has at least i elements, the ith is the minimum of the elements from 1 to i and (observability condition, on the left hand side of the $+^c$ separator) whether the list is not longer than i or it is longer than i but no element following the ith one is the minimum of the elements from 1 to i.

Consider now the call `min([5,1,4,3]`,x). Since `[5,1,4,3]` is formed by four elements only, $|C'_i|_2$ with $i \geqslant 5$ is false. Thus it cannot contribute to any computed answer. Moreover, $|C'_4|_2$ is false because it is not true that `3` is the minimum element

[10] For simplicity, we do not consider here the fact that a Prolog predicate like `M <= H` gives rise to an error when `M` or `H` are not bound to numbers. In all our examples, `min` will be called with the first argument bound to a list of integers. Note, however, that error conditions could be handled in our framework.

among `[5,1,4,3]`. Similarly, $|C'_3|_2$ is false. $|C'_1|_2$, on the contrary, is satisfiable, since `[5,1,4,3]` is formed by at the least one element and `5` is the minimum element of `[5]`. However, the observability constraint $|C'_1|_1$ is false since it requires that if `[5,1,4,3]` is formed by more than one element (and it is) then every other element different from `5` must not be the minimum of the elements of the list from the head to it. This is not our case, since `1` is the minimum of `[5,1]`. Therefore, this means that C'_1 cannot contribute to any computed answer. Consider now C'_2. $|C'_2|_2$ is satisfiable since `[5,1,4,3]` is formed by at least two elements and `1` is the minimum of `[5,1]`. Moreover, the observability constraint is true. This is because if you consider `[5,1,4,3]` as a list of at least i elements, with $i \geqslant 3$, whether this is not possible or, if it is, the ith element is not the minimum of the part of the list from the head to the ith element, since such a minimum is `1`. Then C'_2 is observable and computes a consistent computed answer, which is $\langle [\mathtt{5},\mathtt{1},\mathtt{4},\mathtt{3}], x\rangle = \langle [h_1, h_2|t], h_2\rangle \wedge \mathtt{1} = \min\{\mathtt{5},\mathtt{1}\}$ that is, by simplifying and by projecting on x, $x = \mathtt{1}$. This is the unique consistent and observable computed answer. Note that the operational behaviour of the cut operator of Prolog has been correctly modelled.

9. Denotational semantics for call patterns

In this section we sketch how a denotational semantics for call patterns can be defined. This semantics will take the control rules of Prolog with cut into account. Its construction is similar to the construction of the denotational semantics for computed answers of Section 8. Therefore, we will only introduce the abstraction map and the semantical operators, together with a correctness result.

We abstract a decorated tree as we did in Section 8. We need the cut conditions, the divergent leaves of a tree and the convergent leaves of a tree. Moreover, if a node $o + G$ of the tree is such that $\mathsf{div}(G)$, we have to abstract this node into a call pattern conditional constraint, representing the fact that a call pattern for the leftmost procedure call in G is executed if o is true. This means that we need the following sets of conditional constraints:

- $\mathscr{CP}^c = \{o +^c b \mid o \in \mathscr{O},\ b \in \mathscr{B}\}$ (convergent conditional constraints);
- $\mathscr{CP}^d = \{o +^d b \mid o \in \mathscr{O},\ b \in \mathscr{B}\}$ (divergent conditional constraints);
- $\mathscr{CP}^! = \{o +^! b \mid o \in \mathscr{O},\ b \in \mathscr{B}\}$ (cut conditional constraints);
- $\mathscr{CP}^p = \{o +^p b, \mathtt{p} \mid o \in \mathscr{O},\ b \in \mathscr{B}$ and $\mathtt{p}$ is a predicate symbol$\}$ (call pattern conditional constraints).

Again, we will always drop the word *conditional.* Note that call pattern constraints contain the predicate symbol which is actually called. We define $\mathbb{CP} = \mathscr{CP}^c \cup \mathscr{CP}^d \cup \mathscr{CP}^! \cup \mathscr{CP}^p$.

Eqs. (28), (29) and (31) are extended by defining

$$\begin{aligned} \exists\!\!\exists_x o +^p b, \mathtt{p} &= \exists_x o +^p \exists_x b, \mathtt{p} \\ b \circledcirc (o +^p b', \mathtt{p}) &= (b \bullet o) +^p (b \wedge b'), \mathtt{p} \\ o \odot (o' +^p b, \mathtt{p}) &= (o \sqcap o') +^p b, \mathtt{p}. \end{aligned}$$

Definition 54. We define a map $\alpha_{\mathbb{CP}} : \mathsf{Seq}^+(\mathbb{DT}) \mapsto \mathsf{Seq}^+(\mathbb{CP})$ as follows (compare this with Eq. (30)):

$$\alpha_{\mathbb{CP}}(o+G,\varepsilon)=\begin{cases} o\odot\mathsf{cutsseq}(G)::(o+^{p}\mathsf{con}(G),\mathtt{p})::(o+^{d}\mathsf{con}(G)) \\ \quad\text{if } \mathsf{div}(G) \text{ and } \mathtt{p} \text{ is the leftmost} \\ \quad\text{procedure call present in } G \\ \\ o\odot\mathsf{cutsseq}(G)::(o+^{c}\mathsf{con}(G)) \\ \quad\text{if not } \mathsf{div}(G) \end{cases}$$

$$\alpha_{\mathbb{CP}}(o+G,\tilde{t})=o\odot\mathsf{cutsseq}(G)::(o+^{p}\mathsf{con}(G),\mathtt{p})::\alpha_{\mathbb{CP}}(\tilde{t})$$
$$\text{(where } \mathtt{p} \text{ is the leftmost procedure call present in } G)$$
$$\alpha_{\mathbb{CP}}(\tilde{t}_1::\tilde{t}_2)=\alpha_{\mathbb{CP}}(\tilde{t}_1)::\alpha_{\mathbb{CP}}(\tilde{t}_2).$$

Consider for instance the decorated tree shown in Fig. 8. Its abstraction through the $\alpha_{\mathbb{CP}}$ map is the sequence of constraints

$$\begin{gathered} true_{\emptyset}+^{p}\ true,\ \mathtt{p}::\ true_{\emptyset}+^{p}\ true,\ \mathtt{q}::\ true_{\emptyset}+^{c}(x=3\wedge x=5):: \\ ::-(x=3)+^{p}\ true,\mathtt{r}::-(x=3)+^{d}\ true::-(-(x=3))+^{p}\ x=6,\mathtt{r}:: \\ -(-(x=3))+^{d}\ x=6:: \\ ::(-(-(x=3))\sqcap-(-(-(x=3))\sqcap x=6))+^{!}x=2:: \\ ::(-(-(x=3))\sqcap-(-(-(x=3))\sqcap x=6))+^{c}\ x=2. \end{gathered}$$

This means that there is a call pattern for p which is always observable and with *true* as partial answer. That there are two call patterns for r, observable in the constraint stores in which it is not possible to assume that $x=3$, and with *true* and $x=6$ as partial answer, respectively. Note that there is no call pattern for t, since we are considering a leftmost selection rule. Moreover, note that this denotation is strictly more concrete than the denotation for computed answers, already computed for the same decorated tree.

The definitions of o, δ, κ, β and ξ are left unchanged. We add the following equalities to Definition 44:

$$\begin{aligned} o+^{p}b,\mathtt{p}\otimes\tilde{s}&=o+^{p}b,\mathtt{p}, \\ !(o+^{p}b,\mathtt{p})&=o+^{p}b,\mathtt{p}, \\ ¡(o+^{p}b,\mathtt{p})&=o+^{p}b,\mathtt{p}. \end{aligned}$$

We can still use the identity map (this time on $\mathsf{Seq}^{+}(\mathbb{CP})$) as the abstract counterpart of ϕ^{G} and ψ^{G}. Finally, the map $\oplus$ is simply extended to $\mathsf{Seq}^{+}(\mathbb{CP})$, though its definition remains unchanged. The same happens to the substitution operation. Eqs. (33) and (32) are extended to the new domain $\mathsf{Seq}^{+}(CP)$, using the extended operators. This way, we obtain the maps $T_P^{\mathbb{CP}}$ and $\mathscr{T}_P^{\mathbb{CP}}[\![]\!]$. Now, I is a call pattern interpretation, i.e., a map from predicate symbols into $\mathsf{Seq}^{+}(\mathbb{CP})$.A result similar to Theorem 46 holds:

Theorem 55. *Let G be a goal and I be a decorated tree interpretation such that* $\mathscr{T}_P^{\mathbb{DT}}[\![G]\!]I$ *is defined. We have*
(i) $\alpha_{\mathbb{CP}}(\mathscr{T}_P^{\mathbb{DT}}[\![G]\!]I)=\mathscr{T}_P^{\mathbb{CP}}[\![G]\!]\alpha_{\mathbb{CP}}(I)$,
(ii) $\alpha_{\mathbb{CP}}(T_P^{\mathbb{DT}}(I))=T_P^{\mathbb{CP}}(\alpha_{\mathbb{CP}}(I))$,
where $\alpha_{\mathbb{CP}}(I)[\mathrm{p}]=\alpha_{\mathbb{CP}}(I(\mathrm{p}))$.

A partial order can be defined on $\alpha_{\mathbb{CP}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(T)))$ by modifying condition 2 of Definition 48:

2. $o +^d b \sqsubseteq \tilde{s}$ if and only if every $o' +^c b'$ and $o' +^d b'$ and $o' +^! b'$, and $(o' +^p b', \mathtt{p})$ in $\tilde{s}$ is such that $o \geqslant o'$ and $b \geqslant b'$,

and by adding the condition

5. $o +^p b, \mathtt{p} \sqsubseteq o +^p b, \mathtt{p}$ for every predicate symbol p.

This partial order can be extended to call pattern interpretations which yield for every predicate symbol a sequence in $\alpha_{\mathbb{CP}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(T)))$. The minimum of such a partial order is the interpretation $I_0^{\mathbb{CP}}$ such that

$$I_0^{\mathbb{CP}}[\mathtt{p}] = \mathit{true}_{\emptyset} +^p \ \mathit{true}, \ \mathtt{p} :: \ \mathit{true}_{\emptyset} +^d \ \mathit{true}$$

for every predicate symbol p.[11]

It can be shown that $(T_P^{\mathbb{CP}})^i \sqsubseteq (T_P^{\mathbb{CP}})^{i+1}$ for every $i \geqslant 0$. Therefore, on the completion of $\alpha_{\mathbb{CP}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}(T)))$ we can define

$$\mathscr{S}_P^{\mathbb{CP}} = \mathsf{lub}_i (T_P^{\mathbb{CP}})^i.$$

It can be shown that

$$\mathscr{S}_P^{\mathbb{CP}} = \alpha_{\mathbb{CP}}(\mathscr{S}_P^{\mathbb{DT}})$$

(compare this with Theorem 52) and that $\mathscr{S}_P^{\mathbb{CP}}$ can actually be used to compute the set of call patterns arising from the execution of a goal in a program:

Theorem 56. *Given a program P and a goal G, the set of consistent call patterns for G in P is given by*

$$\{b, \mathtt{p} \mid b \neq \ \mathit{false}, \ o \text{ is true and } o +{}^d b, \mathtt{p} \text{ belongs to } \mathscr{T}_P^{\mathbb{CP}}[\![G]\!]\mathscr{S}_P^{\mathbb{CP}}\}.$$

Compare our goal-independent bottom-up semantics for call patterns with the semantics proposed in Ref. [13]. They propose a goal-independent, bottom-up semantics for resultants. Since call patterns are an abstraction of resultants, their semantics can be used for computing call patterns. This abstraction is explicitly described in Ref. [12], where the authors provide a concrete semantics for call patterns and its abstraction for program analysis. The main difference with our approach is that they use sets of clauses rather than sequences of constraints as the semantical domain. Our choice was motivated by the fact that clauses are more complex objects than constraints. Moreover, their approach does not consider divergence nor the cut operator. An advantage of their approach is that the use of clauses leads easily to an OR-compositional semantics for call patterns (in the sense of Ref. [5]).

9.1. An example

Consider the following Prolog program P:

[11] Note that $\mathit{true}_{\emptyset} +^d \ \mathit{true} \sqsubseteq \ \mathit{true}_{\emptyset} +^p \ \mathit{true}, \ \mathtt{p} :: \ \mathit{true}_{\emptyset} +^d \ \mathit{true}$, but $\mathit{true}_{\emptyset} +^d \ \mathit{true}$ does not belong to $\alpha_{\mathbb{CP}}(\alpha_{\mathbb{DT}}(\mathsf{Seq}^+(T)))$.

```
int(0) :- !.
int(s(X)) :- int(X).
```

which in our abstract syntax becomes

$$\texttt{int}(x) \texttt{ :- } \texttt{cut}(x = 0) \texttt{or exists } y.x = s(y) \texttt{ and } \texttt{int}(y).$$

We have

$$\begin{aligned}
\left(T_P^{\mathbb{CP}}\right)^0[\texttt{int}] &= \mathit{true}_\emptyset +^p \mathit{true}, \texttt{int} :: \mathit{true}_\emptyset +^d \ \mathit{true}\\
\left(T_P^{\mathbb{CP}}\right)^1[\texttt{int}] &= \mathit{true}_\emptyset +^p \mathit{true}, \texttt{int} :: \mathit{true}_\emptyset +^c \alpha = 0 ::\\
&\quad :: -(\alpha = 0) +^p \exists_y.\alpha = \mathtt{s}(y), \texttt{int} :: -(\alpha = 0) +^d \exists_y.\alpha = \mathtt{s}(y)\\
\left(T_P^{\mathbb{CP}}\right)^2[\texttt{int}] &= \mathit{true}_\emptyset +^p \mathit{true}, \texttt{int} :: \mathit{true}_\emptyset +^c \alpha = 0 ::\\
&\quad :: -(\alpha = 0) +^p \exists_y.\alpha = \mathtt{s}(y), \texttt{int} :: -(\alpha = 0) +^c \alpha = \mathtt{s}(0) ::\\
&\quad :: -(\alpha = 0) \sqcap -(\alpha = \mathtt{s}(0)) +^p \exists_y.\alpha = \mathtt{s}(\mathtt{s}(\mathtt{y})), \texttt{int} ::\\
&\quad :: -(\alpha = 0) \sqcap -(\alpha = \mathtt{s}(0)) +^p \exists_y.\alpha = \mathtt{s}(\mathtt{s}(y))
\end{aligned}$$

and, in general,

$$\begin{aligned}
\left(T_P^{CP}\right)^n[\texttt{int}] &= \mathit{true}_\emptyset +^p \mathit{true}, \texttt{int} :: \mathit{true}_\emptyset +^c \alpha = 0 ::\\
&\quad :: -(\alpha = 0) +^p \exists_y.\alpha = \mathtt{s}(y), \texttt{int} :: -(\alpha = 0) +^c \alpha = \mathtt{s}(0) :: \cdots\\
&\quad \cdots :: \sqcap_{0 \leqslant i \leqslant n-2} \left(-(\alpha = \mathtt{s}^i(0))\right) +^p \exists_y.\alpha = \mathtt{s}^{n-1}(y), \texttt{int} ::\\
&\quad :: \sqcap_{0 \leqslant i \leqslant n-2} \left(-(\alpha = \mathtt{s}^i(0))\right) +^c \alpha = \mathtt{s}^{n-1}(0) ::\\
&\quad :: \sqcap_{0 \leqslant i \leqslant n-1} \left(-(\alpha = \mathtt{s}^i(0))\right) +^p \exists_y.\alpha = \mathtt{s}^n(y), \texttt{int}\\
&\quad :: \sqcap_{0 \leqslant i \leqslant n-1} \left(-(\alpha = \mathtt{s}^i(0))\right) +^d \exists_y.\alpha = \mathtt{s}^n(y)
\end{aligned}$$

i.e.

$$\begin{aligned}
\mathscr{S}_P^{\mathbb{CP}}[\texttt{int}] &= \mathit{true}_\emptyset +^d \mathit{true}, \texttt{int} :: \mathit{true}_\emptyset +^c \alpha = 0 ::\\
&\quad :: -(\alpha = 0) +^p \exists_y.\alpha = \mathtt{s}(y), \texttt{int} :: -(\alpha = 0) +^c \alpha = \mathtt{s}(0) :: \cdots\\
&\quad \cdots :: \sqcap_{0 \leqslant i \leqslant n-2} \left(-(\alpha = \mathtt{s}^i(0))\right) +^d \exists_y.\alpha = \mathtt{s}^{n-1}(y), \texttt{int} ::\\
&\quad :: \sqcap_{0 \leqslant i \leqslant n-2} \left(-(\alpha = \mathtt{s}^i(0))\right) +^c \alpha = \mathtt{s}^{n-1}(0) :: \cdots
\end{aligned}$$

This means that only the first call pattern for the goal $\texttt{int}(x)$ in the program P is observable, with a partial computed answer equal to *true*. The other call patterns, with a partial computed answer of $\exists_y.x = \mathtt{s}^i(y)$ for $i \geqslant 1$, are not observable.

10. Conclusions and future work

This paper shows several semantics modelling the Prolog search and selection rules and the cut operator. The derivation of a more abstract semantics from a more concrete semantics is often done through the methodology of abstract interpretation. In this case, abstract interpretation has proved itself to be very useful for a non-trivial application.

Our approach deals naturally with negation as finite failure, which can easily be implemented through a clever use of the cut operator. Simple extensions of our approach can deal with many Prolog built-in's and even with error conditions. The overhead needed for reaching an improved precision essentially consists in the use of observability constraints.

Our denotational semantics can be used as an effective base for precise program analysis. The abstraction of observability constraints requires their downward and upward approximation. This problem has been already introduced in Ref. [19]. In Ref. [23] the authors show how this approximation can be done for a given abstract domain. Assuming that the abstract property at hand is actually affected by control, Ref. [23] shows that there is a general way for computing the upward and downward approximations we need. Moreover, it shows how to deal with the infinitely growing fixpoint computation that can arise from the use of sequences rather then sets. Roughly speaking, Ref. [23] shows that repeated elements in a sequence are useless and therefore can be safely removed. This leads to a finite analysis framework if the abstract domain itself is finite.

Acknowledgements

We acknowledge the anonymous referees for their constructive criticisms. Moreover, we acknowledge doctor Pat Hill for providing a quiet environment during the final revision of this paper.

References

[1] R. Barbuti, M. Codish, R. Giacobazzi, G. Levi, Modelling Prolog control, Journal of Logic and Computation 3 (1993) 579–603.

[2] E. Börger, A logical operational semantics of full Prolog, in: E. Börger, H. Kleine, H. Büning, M. Richter (Eds.), CSL 89, Third Workshop on Computer Science Logic, Lecture Notes in Computer Science, vol. 440, Springer, Berlin, 1990, pp. 36–64.

[3] A. Bossi, M. Bugliesi, M. Fabris, Fixpoint semantics for Prolog, in: D.S. Warren (Ed.), Proceedings of the Tenth International Conference on Logic Programming, MIT Press, Cambridge, MA, 1993, pp. 374–389 .

[4] A. Bossi, M. Gabbrielli, G. Levi, M. Martelli, The s-semantics approach: Theory and applications, Journal of Logic Programming 19 (20) (1994) 149–197.

[5] M. Codish, S.K. Debray, R. Giacobazzi, Compositional analysis of modular logic programs, in: Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages, ACM, New York, 1993, pp. 451–464.

[6] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the Fourth ACM Symposium on Principles of Programming Languages, 1977, pp. 238–252.

[7] P. Cousot, R. Cousot, Abstract interpretation and applications to logic programs, Journal of Logic Programming 13 (23) (1992) 103–179.

[8] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, Cambridge University Press, Cambridge, 1990.

[9] A. deBruin and E. deVink, Continuation semantics for Prolog with cut, in: J. Diaz and F. Orejas (Eds.), Proceedings of CAAP 89, Lecture Notes in Computer Science, vol. 351, Springer, Berlin, 1989, pp. 178–192.

[10] S.K. Debray, P. Mishra, Denotational and operational semantics for Prolog, Journal of Logic Programming 5 (1988) 61–91.

[11] Melvin Fitting, A deterministic Prolog fixpoint semantics, Journal of Logic Programming 2 (2) (1985) 111–118.

[12] M. Gabbrielli, R. Giacobazzi, Goal independency and call patterns in the analysis of logic programs, in: J. Urban, E. Deaton, D. Oppenheim, H. Berghel (Eds.), Proceedings of the Nineth ACM Symposium on Applied Computing, Phoenix AZ, ACM, New York, 1994, pp. 394–399.

[13] M. Gabbrielli, M.C. Meo, Fixpoint semantics for partial computed answer substitutions and call patterns, in: H. Kirchner, G. Levi (Eds.), Algebraic and Logic Programming, Proceedings of the Third International Conference, Lecture Notes in Computer Science, vol. 632, Springer, Berlin, 1992, pp. 84–99.

[14] L. Henkin, J.D. Monk, A. Tarski, Cylindric Algebras. Parts I and II, North-Holland, Amsterdam, 1971.

[15] J. Jaffar, M.J. Maher, Constraint logic programming: A survey, Journal of Logic Programming 19-20 (1994) 503–581.

[16] N.D. Jones, A. Mycroft, Stepwise development of operational and denotational semantics for Prolog, in: S.-Å. Tärnlund (Ed.), Proceedings of the Second International Conference on Logic Programming, 1984, pp. 281–288.

[17] B. Le Charlier, S. Rossi, P. Van Hentenryck, An abstract interpretation framework which accurately handles Prolog search rule and the cut, in: M. Bruynooghe (Ed.), Proceedings of the 1994 International Symposium on Logic Programming, MIT Press, Cambridge, MA, 1994, pp. 157–171.

[18] G. Levi, D. Micciancio, Analysis of pure Prolog programs, in: M.I. Sessa (Ed.), Proceedings GULP-PRODE '95, 1995 Joint Conference on Declarative Programming, 1995.

[19] G. Levi, F. Spoto, Accurate analysis of Prolog with cut, in: P. Lucio, M. Martelli, M. Navarro (Eds.), Proceeding APPIA-GULP-PRODE'96, 1996, pp. 481–492.

[20] R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[21] B.J. Ross, A. Smaill, An algebraic semantics of Prolog program termination, in: Koichi Furukawa (Ed.), Proceedings of the Eighth International Conference on Logic Programming, MIT Press, Cambridge, MA, 1991, pp. 316–330.

[22] F. Spoto, Operational and goal-independent denotational semantics for Prolog with cut: proofs. Available at the web address http://strudel.di.unipi.it/papers/papers.html.

[23] F. Spoto, Giorgio Levi, Abstract interpretation of Prolog programs, in: A.M. Haeberer (Ed.), Proceedings of the Seventh International Conference on Algebraic Methodology and Software Technology, AMAST'98, Lecture Notes in Computer Science, vol. 1548, Amazonia, Manaus, Brazil, January, Springer, Berlin, 1999, pp. 455–470.

[24] E. Todoran, J.I. den Hartog, E.P. de Vink, Comparative metric semantics for commit in or-parallel logic programming, in: J. Mauszyński (Ed.), Logic Programming, Proceedings of the 1997 International Symposium on Logic Programming, MIT Press, Cambridge, MA, 1997, pp. 101–116.