A Semantics-Based Determinacy Analysis for Prolog with Cut

Torben Æ. Mogensen

DIKU
University of Copenhagen
Universitetsparken 1
DK2100 København Ø
Denmark
email: torbenm@diku.dk
phone: (+45) 35321404
fax: (+45) 35321401

Abstract. At PEPM'91 Sahlin presented a determinacy analysis for Prolog with cut. The analysis was, however, not justified in any kind of semantics for Prolog, so correctness was argued with respect to an intuitive idea of the behaviour of Prolog programs.
We now start with a denotational semantics of Prolog and derive a variant of Sahlins analysis as an abstraction of this. We observe that we avoid some problems Sahlin had, and in addition get rid of some redundant domain elements, which our semantics show can not be distinguished by any context.
To obtain better precision in the abstract interpretation we do fixed-point iteration in two steps: First we find the least equivalence class using a powedomain ordering, then we find the least fixed-point within that equivalence class using the subset ordering. We believe this method is original.

1 Introduction.

We first describe a denotational semantics for Prolog with cut, based on the semantics presented by Jones and Mycroft in [4]. The semantics models the observable behaviour of Prolog programs.

When a goal is solved, several things can happen:

The program runs forever. This is called *infinite failure.*
or The programs stops and says "`no`". This is called *finite failure.*
or The programs stops and outputs an answer substitution and can then continue to find other solutions, allowing any of the three behaviours for this search.

This naturally leads to a structure where the behavior of a goal is modeled by a stream of substitutions. The stream can be infinite or it can be terminated by [] (representing finite failure) or $\perp$ (representing infinite failure). To model cut we also allow a stream to terminate in **!**, representing that a cut has occurred before a finite failure occurs. This is sufficient, as we can only observe a cut when a failure occurs.

We will in section 2 show a denotational semantics for Prolog with cut. Then we in section 3 will discuss various issues about abstracting this semantics for doing abstract interpretation, including a (we believe) novel way of improving the precision of analyses that uses powerdomains. In section 4 we use determinacy analysis as an example of an abstract interpretation based on the semantics and compare this to Sahlins analysis.

2 A denotational semantics for Prolog with cut.

2.1 Syntactic domains.

A Prolog program consists of a set of predicates, each of which consists of a list of clauses. Each clause consist of a left-hand side atom (with the same functor and arity as the predicate) and a right-hand side which is a goal. A goal can either be `true`, `fail`, **!**(cut), a conjunction, a disjunction or a call to a predicate (an atom).

$$\begin{array}{lll}
p & \in \mathit{Prog} & = \mathit{Pred}^* \\
r & \in \mathit{Pred} & = \mathit{Clause}^+ \\
l & \in \mathit{Clause} & = \mathit{Atom}\ \texttt{:-}\ \mathit{Goal} \\
m & \in \mathit{Atom} & = \mathit{Functor}(\mathit{Term}^*_{,}) \\
g & \in \mathit{Goal} & = \texttt{true} \mid \texttt{fail} \mid \textbf{!} \mid \mathit{Goal}\texttt{,}\mathit{Goal} \mid \mathit{Goal}\texttt{;}\mathit{Goal} \mid \mathit{Atom} \\
t & \in \mathit{Term} & = \mathit{Var} \mid \mathit{Functor}(\mathit{Term}^*_{,}) \\
v & \in \mathit{Var} & = \text{a countably infinite set of variables} \\
f & \in \mathit{Functor} & = \text{a finite set of functor symbols}
\end{array}$$

2.2 Semantic domains.

We need a domain for substitutions, for unifiers, for streams, for booleans, for answers and for predicate environments.

$$\begin{array}{ll}
\theta \in \mathit{Subst} & = \mathit{Var} \to \mathit{Term} \\
u \in \mathit{Unifier} & = \mathbb{1} + \mathit{Subst} \\
s \in \mathit{Stream} & = (\mathbb{1} + \mathbb{1} + \mathit{Subst} \times \mathit{Stream})_\perp \\
\rho \in \mathit{Env} & = \mathit{Functor} \to \mathit{Term}^* \to \mathit{Stream}
\end{array}$$

Substitutions extend in the natural way to $\mathit{Term} \to \mathit{Term}$ and $\mathit{Term}^* \to \mathit{Term}^*$.

We use the notation [] to mean the element in the first summand of *Stream* (finite failure). The second summand represents the occurrence of a cut followed by finite failure. We denote this by **!**. Elements in the third summand are represented as $\theta : s$, where θ is a substitution and s is a stream. We use $[\theta_1 \ldots \theta_n]$ as a shorthand for $\theta_1 : \ldots : \theta_n : []$, $[\theta_1 \ldots \theta_n]_{\textbf{!}}$ for $\theta_1 : \ldots : \theta_n : \textbf{!}$ and $[\theta_1 \ldots \theta_n]_\perp$ for $\theta_1 : \ldots : \theta_n : \perp$. The element in the left summand of *Unifier* is written as `fail`.

We use $\circ$ to compose substitutions. We also need

$$MGU : Term^* \times Term^* \rightarrow Unifier$$

which finds the most general unifier of two lists (of equal length) of terms, and

$$? : Unifier \times Stream \rightarrow Stream$$

which is defined by

$$\begin{array}{ll} \texttt{fail} & ?\ s = [] \\ \Theta & ?\ s = s \end{array}$$

The ? operator is used to return the empty (failed) stream when unification fails.

$$+\!\!+ : Stream \times Stream \rightarrow Stream$$

concatenates two streams. It is defined by

$$\begin{array}{lll} \bot & +\!\!+\ s & = \bot \\ \mathbf{!} & +\!\!+\ s & = \mathbf{!} \\ [] & +\!\!+\ s & = s \\ \theta : s_1 & +\!\!+\ s_2 & = \theta : (s_1 +\!\!+ s_2) \end{array}$$

It is easy to show that $+\!\!+$ is associative with neutral element (both left and right) []. Note that a cut turns off backtracking, which we model by ignoring the stream after $+\!\!+$.

We need a function to hide the occurrence of cut:

$$hide : Stream \rightarrow Stream$$

$$\begin{array}{ll} hide\ \bot & = \bot \\ hide\ \mathbf{!} & = [] \\ hide\ [] & = [] \\ hide\ (\theta : s) & = \theta : hide\ s \end{array}$$

Finally, we need the functions

$$mapcompose : Subst \rightarrow Stream \rightarrow Stream$$

defined by

$$\begin{array}{ll} mapcompose\ \theta\ \bot & = \bot \\ mapcompose\ \theta\ [] & = [] \\ mapcompose\ \theta\ (\theta_1 : s) & = (\theta_1 \circ \theta) : (mapcompose\ \theta\ s) \end{array}$$

and

$$concatmap : (Subst \rightarrow Stream) \rightarrow Stream \rightarrow Stream$$

defined by

$$\begin{array}{ll} concatmap\ f\ \bot & = \bot \\ concatmap\ f\ \mathbf{!} & = \mathbf{!} \\ concatmap\ f\ [] & = [] \\ concatmap\ f\ (\theta : s) & = (f\ \theta) +\!\!+ (concatmap\ f\ s) \end{array}$$

2.3 Semantic functions.

We are now ready to define the semantic functions.

$$\mathit{Solve} : \mathit{Goal} \to \mathit{Env} \to \mathit{Subst} \to \mathit{Stream}$$

$$\begin{aligned}
&\mathit{Solve}[\![\texttt{true}]\!]\rho\theta &&= [\theta] \\
&\mathit{Solve}[\![\texttt{fail}]\!]\rho\theta &&= [] \\
&\mathit{Solve}[\![\texttt{!}]\!]\rho\theta &&= [\theta]_{\texttt{!}} \\
&\mathit{Solve}[\![g_1, g_2]\!]\rho\theta &&= \mathit{concatmap}\ (\mathit{Solve}[\![g_2]\!]\rho)\ (\mathit{Solve}[\![g_1]\!]\rho\theta) \\
&\mathit{Solve}[\![g_1; g_2]\!]\rho\theta &&= \mathit{Solve}[\![g_1]\!]\rho\theta +\!\!+ \mathit{Solve}[\![g_2]\!]\rho\theta \\
&\mathit{Solve}[\![r(\bar{t})]\!]\rho\theta &&= \mathit{mapcompose}\ \theta\ (\rho\ r\ \theta(\bar{t}))
\end{aligned}$$

$$\mathit{Pred} : \mathit{Functor} \to \mathit{Env} \to \mathit{Term}^* \to \mathit{Stream}$$

$$\begin{aligned}
\mathit{Pred}[\![r]\!]\rho\bar{t} = \ &\mathit{hide}\ (\mathit{Clausolve}[\![c_1]\!]\rho\bar{t} +\!\!+ \ldots +\!\!+ \mathit{Clausolve}[\![c_n]\!]\rho\bar{t}) \\
&\mathit{where}\ \ c_1, \ldots, c_n \text{ are the clauses of the predicate with name } r
\end{aligned}$$

$$\mathit{Clausolve} : \mathit{Clause} \to \mathit{Env} \to \mathit{Term}^* \to \mathit{Stream}$$

$$\mathit{Clausolve}[\![r(\overline{t_1})\ \texttt{:-}\ g]\!]\rho\bar{t} = \ \mathit{MGU}(\overline{t_1}, \bar{t})\ ?\ \mathit{Solve}[\![g]\!]\rho(\mathit{MGU}(\overline{t_1}, \bar{t}))$$

To solve a top-level goal g in a program p, we do

$$\begin{aligned}
&\mathit{hide}\ (\mathit{Solve}[\![g]\!]\rho I) \\
&\quad \mathit{where}\ I \text{ is the identity substitution} \\
&\quad \mathit{and}\ \ \rho\, r_i = \mathit{Pred}[\![r_i]\!]\rho \ \text{ for all predicates } r_i \text{ in } p
\end{aligned}$$

2.4 Comparison to other semantics for Prolog with cut

The advantage of using a denotational semantics instead of an operational semantics is that abstract interpretations can be derived by abstraction of the domains in the semantics, something that is not as directly possible with operational semantics. We have chosen a direct style semantics based on streams in the style of Jones and Mycroft [4] as opposed to continuation semantics (as in [2]) because the number of solutions produced by a goal is directly available in the stream-based semantics but only indirectly in a continuation based semantics by the number of times the continuation is called. On the other hand, a continuation semantics is easier to relate to an operational model, as is done in [2]. The semantics of Debray & Mishra [3] uses a mixture of streams and continuations. While this is closer to the pure stream-based semantics of Jones and Mycroft, the presence of continuations makes abstraction harder.

3 Abstract interpretation

We now describe how the domains used in the semantics can be abstracted to give approximative compile-time information about programs using abstract interpretation. The approach is fairly standard, see e.g. [1] with a few exceptions, which we note below.

3.1 Powerdomains

As usual in abstract interpretation we work with sets of values, which are then abstracted into elements in our abstract domain. We extend the semantic operations to work point-wise over the elements of a set, e.g.

$$S_1 \overline{+\!\!+} S_2 = \{s_1 +\!\!+ s_2 \mid s_1 \in S_1, s_2 \in S_2\}$$

If we want the singleton function to be continous, we can't use subset ordering and we must put some restrictions on the sets. There are various choices of orderings and restrictions, which leads to various different *powerdomains*. We have chosen the *Plotkin* (or *convex* or *Egli-Milner*) powerdomain since this distinguishes more sets than the other traditional powerdomain constructions.

The Plotkin powerdomain. Normally, the Plotkin powerdomain is restricted to closed convex sets. In this case the sets can be partially ordered by the Egli-Milner ordering: $A \sqsubseteq_{EM} B$ iff $\forall a \in A, \exists b \in B, a \sqsubseteq b$ and $\forall b \in B, \exists a \in A, a \sqsubseteq b$. We will, however, want to distinguish between more sets than this restriction allows. So we just restrict ourselves to closed (but not necesarily convex) sets.

A set is closed if whenever a (not necessarily ordered) sequence of values from the set has a limit, then that limit is also in the set. The definition of limits of general sequences can be found in appendix A.

When working with arbitrary closed sets, the Egli-Milner ordering is not a partial order, but just a preorder. Hence, we define $A \equiv_{EM} B$ iff $A \sqsubseteq_{EM} B$ and $B \sqsubseteq_{EM} A$. Now we can define:

Definition 1. The *closed Plotkin powerdomain* $\overline{\mathcal{P}}(D)$ of the domain D is the set of all non-empty closed subsets of D quotiented by $\equiv_{EM}$ and ordered by $\sqsubseteq_{EM}$.

Definition 2. An *EM-fixed-point* of a continous function $f : \overline{\mathcal{P}}(D) \to \overline{\mathcal{P}}(D)$ is a set $A \in \overline{\mathcal{P}}(D)$ such that $f(A) \equiv_{EM} A$. A *minimal* EM-fixed-point of f is an EM-fixed-point A of f such that for any EM-fixed-point B of f. $A \sqsubseteq_{EM} B$.

Note that if A is a (minimal) EM-fixed-point of f, all elements of A's equivalence class will be so too. All minimal EM-fixed-points of f will be in the same equivalence class.

Finding precise fixed-points. We want to have more information from an analysis than is given by the equivalence class of the minimal EM-fixed-points. We note that since a function maps any element of the equivalence class of its minimal EM-fixed-points into another element of the same, we can consider it as a function over that equivalence class. If the function is a point-wise extension $\overline{f}$ of a function f over the original domain, it will be continous with respect to the subset ordering within that equivalence class, and we can talk about the least fixed-point w.r.t the subset ordering of $\overline{f}$ within that equivalence class. Appendix A shows a formal development of this construction. For now, we just note that we will define the least fixed point of $\overline{f}$ by a two-step process:

1. We find the equivalence class A of minimal EM-fixed-points of $\overline{f}$.
2. In that equivalence class, we find the (unique) least fixed-point of $\overline{f}$ with respect to the subset ordering.

For finite domains this can be done by fixed-point iteration. We start by the minimal element $x_0 = \{\bot\}$ of the closed Plotkin powerdomain and iterate $x_{i+1} = \overline{f}(x_i)$ until $x_{i+1} \equiv_{EM} x_i$. We have now found a minimal EM-fixed-point of $\overline{f}$. We now go to the smallest (by subset) element of the equivalence class and call this a_0. We now iterate $a_{i+1} = \overline{f}(a_i)$ until $a_{i+1} = a_i$. We have now found the (in some sense) least fixed-point of $\overline{f}$.

3.2 Abstraction

Using this domain structure for abstract interpretation, we must first for each semantic domain D make an abstract domain A_D.

A_D is related to D by an abstraction function $\alpha_D : \mathcal{P}(D) \rightarrow A_D$ and a concretization function $\gamma_D : A_D \rightarrow \overline{\mathcal{P}}(D)$ such that $\alpha_D \circ \gamma_D = id_{A_D}$ and $\gamma_D \circ \alpha_D \sqsubseteq_{EM} id_{\mathcal{P}(D)}$. Furthermore, if a and a' are in the same equivalence class in $\overline{\mathcal{P}}(D)$ and $a \subseteq a'$, then $\gamma_D(\alpha_D a) \subseteq \gamma_D(\alpha_D a')$.

We define a (pre)order $\preceq$ on A_D by $d \preceq d'$ iff $\gamma_D d \sqsubseteq_{EM} \gamma_D d'$ and a partial order $\leq$ by $d \leq d'$ iff $\gamma_D d \subseteq \gamma_D d'$.

We must then make abstract versions of the operations on the domains, such that the abstract operations approximate the standard point-wise operations, e.g. an abstract concatenation operator $+\!\!+^\sharp$, such that

$$\gamma_{Stream}(a_1 +\!\!+^\sharp a_2) \sqsupseteq_{EM} \gamma_{Stream}(a_1) \overline{+\!\!+} \gamma_{Stream}(a_2)$$

and if $\gamma_{Stream}(a_1 +\!\!+^\sharp a_2) \equiv_{EM} \gamma_{Stream}(a_1) \overline{+\!\!+} \gamma_{Stream}(a_2)$ then also

$$\gamma_{Stream}(a_1 +\!\!+^\sharp a_2) \supseteq \gamma_{Stream}(a_1) \overline{+\!\!+} \gamma_{Stream}(a_2)$$

The abstract operations must be continous over the orderings $\preceq$ and $\leq$.

The semantic equations in the denotational semantics now define a set of equations for the abstract interpretation by replacing constants and functions over the semantic domains by their abstracted version. These equations can be solved by a fixed-point iteration as described in section 3.1.

4 Determinacy analysis.

Dan Sahlin presented at PEPM'91 a determinacy analysis for Prolog with cut [6]. This analysis was, however, not semantically well founded (it relied on intuition for the correctness of the very complex rules used in the analysis). Furthermore, the analysis required a non-trivial termination proof, as the fixed-point iteration did not start at the bottom element of the domain.

We can obtain a determinacy analysis similar to Sahlins by a simple abstraction of the denotational semantics, making a correctness proof relatively simple. The abstraction approach also allows us to fine-tune the precision of the analysis by choosing slightly different abstraction functions.

Abstract domains for determinacy analysis. In determinacy analysis we want to find out how many times a goal can succeed, in particular if it can do so more than once. To this purpose we abstract away from the substitutions and only consider the structure of streams:

$$\begin{array}{ll} A_{Subst} & = \mathbb{1} \\ A_{Unifier} & = \mathbb{1} \\ A_{Stream} & = \overline{\mathcal{P}}(AStream) \\ AStream & = (\mathbb{1} + \mathbb{1} + AStream)_\perp \end{array}$$

For elements of *AStream* we use the notation n, where n is a natural number to mean a []-terminated list with n elements. $n_{\mathbf{!}}$ represents a **!**-terminated list with n elements and $n_\perp$ represents a $\perp$-terminated list with n elements. Hence, $0_\perp$ is the bottom element of *AStream.* We use ∞ to represent the infinite list (the limit element of $\perp$-terminated lists). The abstraction and concretization functions are

$$\alpha_{Stream} S \;=\; \{\beta\, s \mid s \in S\}$$

$$\begin{array}{ll} \beta\,\perp & = 0_\perp \\ \beta\,\mathbf{!} & = 0_{\mathbf{!}} \\ \beta\,[] & = 0 \\ \beta\,(\theta : s) & = \begin{cases} (n+1) & , if\ \beta\, s = n \\ (n+1)_\perp & , if\ \beta\, s = n_\perp \\ (n+1)_{\mathbf{!}} & , if\ \beta\, s = n_{\mathbf{!}} \\ \infty & , if\ \beta\, s = \infty \end{cases} \end{array}$$

$$\gamma_{Stream} S \;=\; \bigcup\{\delta\, s \mid s \in S\}$$

$$\begin{array}{ll} \delta\, 0_\perp & = \{\perp\} \\ \delta\, 0_{\mathbf{!}} & = \{\mathbf{!}\} \\ \delta\, 0 & = \{[]\} \\ \delta\,(n+1)_\perp & = \{\theta : s \mid \theta \in Subst, s \in \delta\, n_\perp\} \\ \delta\,(n+1)_{\mathbf{!}} & = \{\theta : s \mid \theta \in Subst, s \in \delta\, n_{\mathbf{!}}\} \\ \delta\,(n+1) & = \{\theta : s \mid \theta \in Subst, s \in \delta\, n\} \end{array}$$

Note that applying β to an infinite stream requires solving the equation $x = x+1$, which is solved by $x = \infty$. By letting the pattern $(n+1)$ match ∞, we get the set of infinite streams as the concretization of ∞.

A_{Stream} is, however, of infinite height, and hence unsuitable for abstract interpretation. So we need to make a further approximation. An obvious choice (also made by Sahlin) is to combine all streams that are longer than 1. This can be done by limiting *AStream* to

$$\{0_\perp, 0_!, 0, 1_\perp, 1_!, 1, 2_\perp, 2_!, 2\}$$

where $2_\perp$ abstract all $\perp$-terminated streams with two or more elements and infinite streams (which are limits point of chains of $\perp$-terminated streams). Similarly, the set containing all []-terminated (or **!**-terminated) streams is not closed unless we include all infinite streams as well (the limit of the sequence $1, 2, \ldots$ is ∞). Hence, we let 2 and $2_!$ abstract all [] and **!**-terminated streams *and* infinite streams (making $2_\perp \sqsubseteq 2$ and $2_\perp \sqsubseteq 2_!$). The new abstraction and concretization functions are

$$\alpha_{Stream} S \;=\; \{\beta\, s \mid s \in S\}$$

$$\begin{array}{ll}
\beta\,\perp & = 0_\perp \\
\beta\,! & = 0_! \\
\beta\,[] & = 0 \\
\beta\,(\theta : s) = & \begin{cases} 1 & , if\;\; \beta\, s = 0 \\ 1_\perp & , if\;\; \beta\, s = 0_\perp \\ 1_! & , if\;\; \beta\, s = 0_! \\ 2 & , if\;\; \beta\, s = 1 \;\mathit{or}\; \beta\, s = 2 \\ 2_\perp & , if\;\; \beta\, s = 1_\perp \;\mathit{or}\; \beta\, s = 2_\perp \\ 2_! & , if\;\; \beta\, s = 1_! \;\mathit{or}\; \beta\, s = 2_! \end{cases}
\end{array}$$

$$\gamma_{Stream} S \;=\; \bigcup\{\delta\, s \mid s \in S\}$$

$$\begin{array}{ll}
\delta\, 0_\perp & = \{\perp\} \\
\delta\, 0_! & = \{\mathtt{1}\} \\
\delta\, 0 & = \{[]\} \\
\delta\, 1_\perp & = \{[\theta]_\perp \mid \theta \in \mathit{Subst}\} \\
\delta\, 1_! & = \{[\theta]_! \mid \theta \in \mathit{Subst}\} \\
\delta\, 1 & = \{[\theta] \mid \theta \in \mathit{Subst}\} \\
\delta\, 2_\perp & = \{[\theta_1, \ldots, \theta_n]_\perp \mid \theta_1, \ldots, \theta_n \in \mathit{Subst}\} \cup \{\theta_1 : \theta_2 : \ldots \mid \theta_1, \theta_2, \ldots \in \mathit{Subst}\} \\
\delta\, 2_! & = \{[\theta_1, \ldots, \theta_n]_! \mid \theta_1, \ldots, \theta_n \in \mathit{Subst}\} \cup \{\theta_1 : \theta_2 : \ldots \mid \theta_1, \theta_2, \ldots \in \mathit{Subst}\} \\
\delta\, 2 & = \{[\theta_1, \ldots, \theta_n] \mid \theta_1, \ldots, \theta_n \in \mathit{Subst}\} \cup \{\theta_1 : \theta_2 : \ldots \mid \theta_1, \theta_2, \ldots \in \mathit{Subst}\}
\end{array}$$

Note that in the definition of β, any of $2, 2_\perp$ or $2_!$ are solutions if the argument is the infinite stream. We will include all of these in the abstraction of a set that contains the infinite stream.

What remains is to make abstract versions of the operators *MGU*, ?, ++, *hide*, *mapcompose* and *concatmap*. $MGU^\sharp$ is trivial, it just returns the unit element of the $A_{Unifier}$ domain. ? is defined by

$$u\ ?\ S \ = \ S \cup \{0\}$$

mapcompose$^\sharp$ ignores the substitution and returns its second argument unchanged. The other operators are shown below.

$$S_1 \mathbin{+\!\!+}^\sharp S_2 \ = \ \{s_1 \mathbin{+\!\!+}^\natural s_2 \mid s_1 \in S_1, s_2 \in S_2\}$$

where $\mathbin{+\!\!+}^\natural$ is defined by the table

$s_1 \mathbin{+\!\!+}^\natural s_2$									
$s_1 \backslash s_2$	$0_\perp$	$0_!$	0	$1_\perp$	$1_!$	1	$2_\perp$	$2_!$	2
$0_\perp$	$0_\perp$	$0_\perp$	$0_\perp$	$0_\perp$	$0_\perp$	$0_\perp$	$0_\perp$	$0_\perp$	$0_\perp$
$0_!$	$0_!$	$0_!$	$0_!$	$0_!$	$0_!$	$0_!$	$0_!$	$0_!$	$0_!$
0	$0_\perp$	$0_!$	0	$1_\perp$	$1_!$	1	$2_\perp$	$2_!$	2
$1_\perp$	$1_\perp$	$1_\perp$	$1_\perp$	$1_\perp$	$1_\perp$	$1_\perp$	$1_\perp$	$1_\perp$	$1_\perp$
$1_!$	$1_!$	$1_!$	$1_!$	$1_!$	$1_!$	$1_!$	$1_!$	$1_!$	$1_!$
1	$1_\perp$	$1_!$	1	$2_\perp$	$2_!$	2	$2_\perp$	$2_!$	2
$2_\perp$	$2_\perp$	$2_\perp$	$2_\perp$	$2_\perp$	$2_\perp$	$2_\perp$	$2_\perp$	$2_\perp$	$2_\perp$
$2_!$	$2_!$	$2_!$	$2_!$	$2_!$	$2_!$	$2_!$	$2_!$	$2_!$	$2_!$
2	$2_\perp$	$2_!$	2	$2_\perp$	$2_!$	2	$2_\perp$	$2_!$	2

hide$^\sharp$ is simple:

$$\mathit{hide}^\sharp\ S \ = \ \{\mathit{hide}^\natural \mid s \in S\}$$

where

$$\begin{aligned}
&\mathit{hide}^\natural\ 0_! &&= 0 \\
&\mathit{hide}^\natural\ 1_! &&= 1 \\
&\mathit{hide}^\natural\ 2_! &&= 2 \\
&\mathit{hide}^\natural\ x &&= x \qquad \text{otherwise}
\end{aligned}$$

concatmap$^\sharp : A_{Stream} \to A_{Stream} \to A_{Stream}$ is more interesting.

$$\mathit{concatmap}^\sharp\ A\ S = \bigcup\{\mathit{ccmap}\, A\, s \mid s \in S\}$$

$$\begin{aligned}
&\mathit{ccmap}\ A\ 0_\perp &&= \{0_\perp\} \\
&\mathit{ccmap}\ A\ 0_! &&= \{0_!\} \\
&\mathit{ccmap}\ A\ 0 &&= \{0\} \\
&\mathit{ccmap}\ A\ 1_\perp &&= \{a \mathbin{+\!\!+}^\natural 0_\perp \mid a \in A\} \\
&\mathit{ccmap}\ A\ 1_! &&= \{a \mathbin{+\!\!+}^\natural 0_! \mid a \in A\} \\
&\mathit{ccmap}\ A\ 1 &&= A \\
&\mathit{ccmap}\ A\ 2_\perp &&= \{a \mathbin{+\!\!+}^\natural a' \mathbin{+\!\!+}^\natural 0_\perp \mid a, a' \in A\} \cup \{a \mathbin{+\!\!+}^\natural a' \mid a \in A, a' \in \mathit{ccmap}\, A\, 2_\perp\} \\
&\mathit{ccmap}\ A\ 2_! &&= \{a \mathbin{+\!\!+}^\natural a' \mathbin{+\!\!+}^\natural 0_! \mid a, a' \in A\} \cup \{a \mathbin{+\!\!+}^\natural a' \mid a \in A, a' \in \mathit{ccmap}\, A\, 2_!\} \\
&\mathit{ccmap}\ A\ 2 &&= \{a \mathbin{+\!\!+}^\natural a' \mid a, a' \in A\} \cup \{a \mathbin{+\!\!+}^\natural a' \mid a \in A, a' \in \mathit{ccmap}\, A\, 2\}
\end{aligned}$$

This definition requires least fixed-point solutions to the last three equations for *ccmap*. These are easy (if costly) to compute, as the domains are finite. Tabulation may not be practical as there are $2^9 \times 9 = 4608$ argument combinations. Sahlin uses what amounts to a tabulation of inverse images: for each element a in A_{Stream} he lists a set of (A, s) pairs such that for any argument (A', s') that has a in its result there exist a pair (A, s) such that $A \subseteq A'$ and $s = s'$. This reduces the size of the tables considerably, but requires searching which may end up being as slow as fixed-point iteration.

Note that *ccmap* is *not* distributive in its first argument.

The abstract version of the *Solve* function maps a goal and an abstract environment to an abstract *Stream*. An abstract environment maps predicate names to abstract *Streams*. The abstract interpretation will construct an abstract environment ρ as the least fixed-point of the equations $Pred^\sharp [\![r_i]\!] \rho$ for all predicates r_i in the program. The abstract semantic functions follow directly from the definitions of the abstract domains and operators.

4.1 Comparison to Sahlins analysis.

Sahlins analysis is not based on a semantics of the language, so it must be argued correct by referring to intuition about the behaviour of programs. In particular, the large number of complex rules for the conjunction operator take some effort to check for correctness and especially completeness. Both of these issues can in our analysis be handled by proving each semantic operator correct with respect to the concretization function.

Sahlin used subset ordering and did not start his fixed-point iteration from the least element (which would be the empty set), but the set $\{0_\perp\}$ so he needed a complex termination proof and had no notion of a "least fixed-point". By using the Egli-Milner ordering the set $\{0_\perp\}$ is indeed the least element, so a least fixed-point equivalence class is found in finite time, after which the least (subset-wise) fixed-point within the equivalence class is found. The fixed-point we obtain this way is at least as precise as the fixed-point obtained by Sahlins analysis. They will (all else being equal) be in the same Egli-Milner equivalence class, but we are guaranteed minimality within the equivalence class, which is not obviously the case with Sahlins method.

Sahlins analysis distinguishes infinite failure with and without cut. As a consequence, he has elements $0_{\perp !}$, $1_{\perp !}$ and $2_{\perp !}$ indicating that a cut has occurred either while producing the solutions or afterwards while looping. These elements can, however, not be distinguished from $0_\perp$, $1_\perp$ and $2_\perp$ by any context. Hence, no optimization seems plausible which may use the extra information. In consequence, Sahlins analysis is needlessly complex, both in terms of running time and conceptual complexity.

Sahlins use of tabulation of inverse images for the equivalent of *ccmap* was almost mandatory, as a direct tabulation would have to handle $2^{12} \times 12 = 49152$ argument combinations, which would be ridiculous to construct by hand, and quite impractical even if constructed by machine. By using a semantics based approach we do not have to do a tedious and error-prone tabulation by hand, but can either compute *ccmap* on the fly or indeed construct a table by machine. While 4608 elements is a nontrivial size for a table, it is bound to speed computation and it is not unrealistic to do.

4.2 Extensions

The semantics shown above does not include negation by failure or the related conditional ($G_1 -> G_2 ; G_3$) construction. This can be justified by the fact that the effects of both of these can be achieved using cut, so they are in fact redundant. It would, however, be convenient to have these in the semantics. To some extent, it is unproblematical to include negation, e.g. by an operator *not* on streams:

$$\begin{aligned} &\mathit{not}\ [] &&= [I] \\ &\mathit{not}\ (\Theta_1 : s) &&= [] \end{aligned}$$

Where I is the identity substitution. Here we have, however, glossed over what happens if a cut has occurred while solving the negated goal. What, for example, should *not* **!** be? There are two reasonable answers: $\mathit{not}\ \mathbf{!} = I : \mathbf{!}$ and $\mathit{not}\ \mathbf{!} = [I]$. In the first case, the scope of the cut is the entire clause, as if the cut was outside the negation. In the second case, the scope of the cut is limited to the negated clause. None of the referenced semantics for Prolog with cut include negation, so no help can be found there, nor does any of half a dozen books on Prolog at our library shed any light on this. Trying out the situation in SICStus Prolog [8] reveals that the compiler will not accept cuts appearing syntactically within negated goals or in the condition part of the conditional construct. With this restriction, the above definition of *not* is complete, and a similar definition for the conditional construct can be made:

$$\begin{aligned} &[] &&\rightarrow s_1 \ ; s_2 = s_2 \\ &(\Theta_1 : s) &&\rightarrow s_1 \ ; s_2 = s_1 \end{aligned}$$

The determinacy analysis as described ignores substitutions. But substitutions provide valuable information about the number of solutions a goal can have. In particular, knowing that a term is ground will often allow us to deduce that at most one solution will be returned. Conversely, that a variable is unbound will sometimes mean that a goal can never fail.

While it is easy to make an abstraction of the denotational semantics for groundness analysis, one important use of this information in determinacy analysis, namely indexing, requires special attention.

If a predicate has non-overlapping left-hand sides and the argument is sufficiently ground, at most one of the left-hand sides will match. We want to be able to use this in the analysis. In the semantics, the stream returned by a predicate call is the concatenation of the streams returned by the clauses in the predicate. We can not directly model the desired behaviour by an abstraction of the concatenation operator, as this doesn't have any information about what the left-hand sides look like. A possible solution is first do the groundness analysis and then use the information gathered by this to insert cuts in predicates where indexing is possible. Another solution is to rewrite the denotational semantics so it matches the left-hand sides of all the clauses in a predicate using a single extended *MGU* operator. This can then be abstracted to take indexing into account. While this approach is more flexible (as it e.g. allows the groundness analysis to be done at the same time as the determinacy analysis), it does complicate the denotational semantics, which ideally should be as simple as possible.

Appendix A

Here we describe some theory of the Plotkin powerdomain and a proof that an equivalence class in the closed Plotkin powerdomain is a complete lattice partially ordered by set inclusion.

We first state a few theorems and definitions from [5] and [7].

Definition 3. A subset of a domain D is *open* if it is upwards closed and whenever it contains an element that is the limit of an ascending chain, it contains at least one element of the chain.

Definition 4. A *finitely branching generating tree* is a possibly infinite finitely branching tree with nodes labeled by elements of D such that the label of a node is $\sqsupseteq$ the labels of its ancestor nodes. The set generated by a finitely branching generating tree is the set of labels of leaf nodes plus the limits of the chains of labels formed from infinite paths in the tree. A subset of a domain D is *finitely generable* if it is generated by a finitely branching generating tree. Note that the empty set is not finitely generable.

Definition 5. For subsets A, B of D, we say that $A \underset{\sim}{\sqsubseteq}_{EM} B$ iff

1. For all open sets $U \subseteq D$, $A \cap U \neq \emptyset$ implies $B \cap U \neq \emptyset$.
2. For all open sets $U \subseteq D$, $B \setminus U \neq \emptyset$ implies $A \setminus U \neq \emptyset$.

$\underset{\sim}{\sqsubseteq}_{EM}$ is reflexive and transitive, but not anti-symmetric. Hence, we define an equivalence relation $\approx_{EM}$: $A \approx_{EM} B$ iff $A \underset{\sim}{\sqsubseteq}_{EM} B$ and $B \underset{\sim}{\sqsubseteq}_{EM} A$.

Definition 6. The *Plotkin powerdomain* $\mathcal{P}(D)$ of the domain D is the set of all finitely generable subsets of D quotiented by $\approx_{EM}$ and partially ordered by $\underset{\sim}{\sqsubseteq}_{EM}$.

Theorem 7. *If* $X \sqsubseteq_{EM} Y$ *then* $X \underset{\sim}{\sqsubseteq}_{EM} Y$.

Definition 8. An element $a \in D$ is *finite* iff whenever a is the least upper bound of a directed set, a must be an element of that set. If a is finite then $\uparrow a = \{b \in D \mid a \sqsubseteq b\}$ is an open set.

Definition 9. A domain D is *ω-algebraic* if the set of finite elements of D is enumerable and for all $x \in D$ the set $\downarrow x = \{d \mid d \sqsubseteq x,\ d \text{ is finite}\}$ is directed and has least upper bound x.

We will assume D is ω-algebraic, so we will use the above as a theorem.

Theorem 10. *If $A \subseteq D$ is a finite set of finite elements, then every minimal upper bound of A is a finite element.*

Definition 11. The *limit* of a (not necessarily ascending) sequence of elements from D is defined by $\lim_{n\to\infty} x_n = a$ iff for all finite elements $e \in D$: $a \in \uparrow e \Rightarrow \exists k > 0 : \forall j > k : x_j \in \uparrow e$ and $a \notin \uparrow e \Rightarrow \exists k > 0 : \forall j > k : x_j \notin \uparrow e$

Not all sequences have a limit, but

Theorem 12. *Any sequence has a converging sub-sequence.*

Theorem 13. *All non-empty closed subsets of D are finitely generable. All non-empty finite subsets of D are closed.*

Theorem 14. *If A is finitely generable, then $A \approx_{EM} Cl(A)$, where $Cl(A)$ is the closure of A.*

Definition 15. A set A is *convex* if whenever $x \sqsubseteq y \sqsubseteq z$ and $x, z \in A$ then $y \in A$.

Theorem 16. *If A is finitely generable, then $Con(Cl(A))$, where $Con(A)$ is the convex closure of A, is closed and $A \approx_{EM} Con(Cl(A))$. $Con(Cl(A))$ is the least convex closed set containing A.*

Theorem 17. *If A and B are finitely generable, then $A \approx_{EM} B$ iff $Con(Cl(A)) = Con(Cl(B))$.*

Since we have restricted ourselves to closed sets, we have $A = Cl(A)$ for all the sets in $\overline{\mathcal{P}}(D)$ and hence we have that $A \approx_{EM} B$ iff $Con(A) = Con(B)$ for $A, B \in \overline{\mathcal{P}}(D)$.

We use a few lemmas:

Lemma 18. *For any $a \in D$, the set $\triangle a = \{x \in D \mid x \not\sqsubseteq a\}$ is an open set.*

Proof: follows from the transitivity of $\sqsubseteq$ and the continuity of the function

$$f_a(d) = \begin{cases} \bot & d \sqsubseteq a \\ \top & d \not\sqsubseteq a \end{cases}$$

$\square$

Lemma 19.

i *For any $a \in A$ there exist a minimal $a' \in A$ such that $a' \sqsubseteq a$.*

ii *If $A \approx_{EM} B$ and a is a minimal element of A, then a is a minimal element of B.*

iii *If $A \in \overline{\mathcal{P}}(D)$ then for any $a \in A$ there exist a maximal $a'' \in A$ such that $a \sqsubseteq a''$.*

iv *If $A, B \in \overline{\mathcal{P}}(D)$, $A \approx_{EM} B$ and a is a maximal element of A, then a is a maximal element of B.*

Proof:

i If a is minimal in A we are done. Otherwise, there must exist $a_1 \sqsubset a$, $a_1 \in A$. If a_1 is minimal we are done, otherwise we find a smaller a_2 etc. Since there are no infinite strictly descending chains in D, this process must eventually terminate.

ii For any $a \in A$, $A \setminus \triangle a \neq \emptyset$. Since $B \underset{\sim}{\sqsubseteq}_{EM} A$, we have $B \setminus \triangle a \neq \emptyset$ and hence there exist a $b \in B$, $b \sqsubseteq a$. Using the same reasoning we find $a' \in A$, $a' \sqsubseteq b$. Since a is minimal in A, $a' = a$ and thus $a = b$. If we assume b is not minimal, we have $b' \sqsubset b$ and we can find an element $a'' \in A$, $a'' \sqsubseteq b$. But this is in contradiction with our assumption that a is minimal in A. Hence, $b = a$ must be minimal in B.

iii Let us assume the converse. Then the set $\upharpoonright a$ can contain no element that is maximal in A. Now assume that we have a totally ordered subset S of $\upharpoonright a \cap A$. Since A is closed under limits, the least upper bound of S is in A and since $\sqcup S \sqsupseteq a$ it must be in $\upharpoonright a \cap A$. Hence, for every totally ordered subset of $\upharpoonright a \cap A$ there is an element in $\upharpoonright a \cap A$ which is maximal w.r.t. the subset. This is the definition of an inductively ordered set, and by Zorns lemma one such has a maximal element. This must also be a maximal element of A itself, as no element in $A \setminus \upharpoonright a$ can be larger than it. Hence, we have reached a contradiction.

iv We first take the case where a is a finite element. Since $a \in A$ we have $\upharpoonright a \cap A \neq \emptyset$ and hence $\upharpoonright a \cap B \neq \emptyset$. Thus there must exist $b \in B$, $a \sqsubseteq b$. If b is finite, we can use the same argument to find an element $a' \sqsupseteq b$ in A, and hence we have $a = a' = b$. If b is not finite, the set $\upharpoonright a$ must contain a countable infinity of finite elements a_i smaller than b (using definition 9 and theorem 10). If we look at the open sets $\upharpoonright a_i$, only the one where $a_i = a$ will have a non-empty intersection with A (since all $a_i \sqsupseteq a$ and a is maximal in A). Hence (since $A \approx_{EM} B$), for all the a_i, $a_i \neq a$, $\upharpoonright a_i$ will have an empty intersection with B. Since b is in all of the $\upharpoonright a_i$ we have a contradiction. Hence, b can not be non-finite, and hence we have b is finite and $a = b$.
Now assume a is maximal in A and not finite. Let $a_1, a_2, \ldots$ form an ascending chain of values from $\downharpoonleft a$ with $\bigsqcup_{i \in N} a_i = a$. $\upharpoonright a_i$ is an open set containing a. Hence, $B \cap \upharpoonright a_i \neq \emptyset$. We now form a sequence of elements $b_i \in B \cap \upharpoonright a_i$. By theorem 12, the sequence b_i has a converging sub-sequence. Since $a_i \sqsubseteq b_i$, the limit b of this sub-sequence is greater than or equal to a and since B is closed, $b \in B$. Hence, we have found a $b \in B$ such that $a \sqsubseteq b$. By a similar reasoning, we can find $a' \in A$ such that $b \sqsubseteq a'$ and hence (since a is maximal in A), $a = a' = b$.

$\square$

We now have that all equivalent closed sets contain the same minimal and maximal elements. Also, (by i and iii) any element of a closed set can be wedged between a minimal and a maximal element. Hence, the convex closure of the set containing just the minimal and maximal elements will contain the set itself, and hence also its convex closure. Since the intersection of any collection of equivalent sets contain the minimal and maximal elements of the sets, it too will have the same closure. Also, the intersection of any collection of closed sets is again closed. Hence, we have that the intersection of any collection of equivalent sets is equivalent to all the sets in the collection. Also, if a collection of sets have the same convex closure, their union will also have the same convex closure. However, the union need not be a closed set. The closure of the union under limits of sequences will, however, be closed and have the same convex closure (since the convex closure is a closed set).

We also have:

Theorem 20. *If A and B are closed stes, then $A \underset{\sim}{\sqsubseteq}_{EM} B$ iff $A \sqsubseteq_{EM} b$.*

Proof: 'If' comes from theorem 7. For 'only if', we recall that $A \underset{\sim}{\sqsubseteq}_{EM} B$ means that for any open set U, $A \cap U \neq \emptyset \Rightarrow B \cap U \neq \emptyset$ and $B \setminus U \neq \emptyset \Rightarrow A \setminus U \neq \emptyset$. If $b \in B$ then $B \setminus \triangle b \neq \emptyset$ and hence $A \setminus \triangle b \neq \emptyset$. Thus there must be an element $a \in A$ such that $a \sqsubseteq b$. If $a \in A$ is a finite element, $A \cap \uparrow a \neq \emptyset$ and hence $B \cap \uparrow a \neq \emptyset$ and hence there must be $b \in B$ such that $a \sqsubseteq b$. If $a \in A$ is not finite we look at a chain of elements a_i from $\downarrow a$. Each $\uparrow a_i$ contains a and hence has a non-empty intersection with A and consequently with B. We now find a sequence of $b_i \in B \cap \uparrow a_i$. This sequence has a convergent subsequence with a limit b in B (since B is closed). Furthermore, since $a_i \sqsubseteq b_i$, the limit must be greater than or equal to a. Henec we have found an elememt $b \in B$ such that $a \sqsubseteq b$.

$\square$

Hence, we have our main theorem:

Theorem 21. *Given $A \in \overline{\mathcal{P}}(D)$, the equivalence class of A forms a complete lattice ordered by inclusion. The meet of arbitrary collections is defined and equal to the intersection of these. The join operation is union followed by closure. The convex closure of A, $Cl(A)$ is the top element of the lattice. The bottom element is the set containing all minimal and maximal elements of A.*

By Tarskis fixed-point theorem, any monotone function over a complete lattice has a unique least fixed-point defined by the meet of all post-fixed-points: $lfp(f) = \bigcap\{x \mid x \sqsupseteq f(x)\}$. Since point-wise extensions of continous functions are monotone w.r.t. the subset ordering, the least fixed-point of a function within the equivalence class of its minimal EM-fixed-points is well defined.

References

1. S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Chichester: Ellis Horwood, 1987.
2. A. de Bruin and E.P. de Vink, *Continuation Semantics for PROLOG with Cut*, Proceedings of TAPSOFT'89, Springer LNCS 351, pp. 178-192, 1989.
3. S. K. Debray and P. Mishra, *Denotational and Operational Semantics for Prolog*, Journal of Logic Programming **5**, pp. 61-91, 1988.
4. N. D. Jones and A. Mycroft, *Stepwise Development of Operational and Denotational Semantics for PROLOG*, 1984 International Symposium on Logic Programming, pp. 281-288, IEEE Computer Society Press 1984.
5. G. D. Plotkin, *A Powerdomain Construction*, SIAM J. of Computing, Vol 5, No.3, September 1976.
6. D. Sahlin, *Determinacy Analysis for Full Prolog*, Proceedings of PEPM'91, pp. 23-30, ACM Press 1991.
7. D. Schmidt, *Denotational Semantics, a Methodology for Language Development*, Wm. C. Brown Publishers, 1988.
8. *SICStus Prolog User's Manual (v. 2.1)*, SICS, 1983

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