

A structured alternative to Prolog with simple compositional semantics

ANTÓNIO PORTO

LIACC / Department of Computer Science, Faculty of Sciences, University of Porto, Portugal
(e-mail: ap@dcc.fc.up.pt)

Abstract

Prolog's very useful expressive power is not captured by traditional logic programming semantics, due mainly to the cut and goal and clause order. Several alternative semantics have been put forward, exposing operational details of the computation state. We propose instead to redesign Prolog around structured alternatives to the cut and clauses, keeping the expressive power and computation model but with a compositional denotational semantics over much simpler states—just variable bindings. This considerably eases reasoning about programs, by programmers and tools such as a partial evaluator, with safe unfolding of calls through predicate definitions.

An if-then-else across clauses replaces most uses of the cut, but the cut's full power is achieved by an until construct. Disjunction, conjunction and until, along with unification, are the primitive goal types with a compositional semantics yielding sequences of variable-binding solutions. This extends to programs via the usual technique of a least fixpoint construction. A simple interpreter for Prolog in the alternative language, and a definition of until in Prolog, establish the identical expressive power of the two languages. Many useful control constructs are derivable from the primitives, and the semantic framework illuminates the discussion of alternative ones.

The formalisation rests on a term language with variable abstraction as in the λ -calculus. A clause is an abstraction on the call arguments, a continuation, and the local variables. It can be inclusive or exclusive, expressing a local case bound to a continuation by either a disjunction or an if-then-else. Clauses are open definitions, composed (and closed) with simple functional application (β -reduction). This paves the way for a simple account of flexible module composition mechanisms.

Cube, a concrete language with the exposed principles, has been implemented on top of a Prolog engine and successfully used to build large real-world applications.

KEYWORDS: Prolog, cut, compositional semantics, denotational semantics

1 Introduction

Practitioners of logic programming have always relied on Prolog's *cut* for expressive power, not just efficiency, although it destroys the logical reading of programs and, even worse, is not amenable to a simple compositional reasoning. The ordering of goals and clauses is also crucial in practice, not just for the meaning of cuts but to get desired *sequences* of solutions, another departure from the ideal set-theoretic semantics. Real programs also often rely on solutions being partial (rather than ground) instances of a call. This aspect has been given extensive formal treatment in the *s*-semantics approach (Bossi et al. 1994), and solution order

has been captured in an algebra over solution streams (Seres et al. 1999), but proposed formalisations of the cut have serious problems for effective applicability. Most have the inherent complexity of resorting to a detailed operational state, e.g. the decorated SLD tree (Debray and Mishra 1988; Spoto 2000), success and failure continuations (de Bruin and de Vink 1989) or a π -calculus version of such (Li 1994). Others are simpler but at the cost of generality by restricting the use of cut, e.g. (Andrews 2003) where the “firm” cut has just the power of (a limited form of) if-then-else.

In contrast to previous attempts at tackling these problems, our novel approach avoids a direct characterisation of cut’s behaviour and offers instead a few linguistic alternatives, which provide Prolog’s programming flavour but are compositional in a simple semantic domain based on variable binding states. One such alternative is **if-then-else**, whose semantics underpins most uses of the cut. Available in Prolog only within a clause body, we can use it across several clauses, via a simple reformulation of clause syntax yielding proper compositional abstractions. But **if-then-else** does not capture the full power of Prolog’s cut, needed for more generic pruning of solutions. For this we propose an **until** construct, that together with conjunction, disjunction and unification provide the primitive ingredients of the formal machinery. With these we can define other control constructs such as **if-then-else**, **not** or **var**, and write a very compact interpreter for Prolog programs, thus showing, since **until** is easily expressed in Prolog, the equivalent expressive power of the two languages.

Our linguistic proposal is seemingly simple, both syntactically and (especially) semantically. We regard it in relation to the cut as structured programming historically stood to **goto**, making us firmly believe in its methodological impact on good programming practice. The exhibited compositionality allows programmers to reason locally, i.e. disregarding a goal’s context in the code, in terms of *sequences* of variable binding solutions for the goal, from an initial variable binding setting. Substantially eased is the task of writing important tools such as debuggers, abstract interpreters or partial evaluators, as the unfolding of a call through its predicate definition is sound, being context-independent, in contrast to Prolog.

Our design principles led us to implement a concrete language, called **Cube**, on top of a Prolog engine and successfully use it, for a number of years and by many programmers, to build complex real-world applications such as the online academic management system at our previous Faculty (Porto 2003).

The formalism in the paper is relatively straightforward. We consider a term syntax with variable scoping (abstraction), for a rigorous account of clause composition and dynamic variable creation. A (goal) term denotes a *behaviour* function from an initial *setting*—a variable scope and a substitution—into a corresponding *outcome* capturing the stream of alternative solutions—a sequence of settings ending (if finite) with a termination status of finite failure or divergence. This is recognisable as an abductive semantics for valuations of goals’ variables, a natural way for programmers to understand their code. Disjunction, conjunction and **until** correspond respectively to notions of sum, product and pruning of outcomes. The semantics of programs with procedure definitions and calls is given (as usual) as the least fixpoint of a suitable continuous call step transformer of program interpretations.

2 The cut, if-then-else and clause syntax

Consider this Prolog code for a predicate to delete repeated elements in a list (for variable-tail lists we prefer the notation $X.L$ to $[X|L]$):

```
dre( X.L, D ) :- X in L, !, dre( L, D ).
dre( X.L, X.D ) :- dre( L, D ).
dre( [], [] ).
```

This is a fairly typical case of a definition whose understanding, although simple, relies crucially on the order of clauses and the effect of the cut. Even a novice Prolog programmer will recognise here the implicit pattern of an if-then-else, with the if-then part stated in the first clause and else in the next one(s). Indeed, one can replace the first two clauses with a single one having an if-then-else body:

```
dre( X.L, Y ) :- X in L -> dre( L, Y ) ; Y=(X.D), dre( L, D ).
```

Although another alternative with cut and disjunction would work just as well,

```
dre( X.L, Y ) :- X in L, !, dre( L, Y ) ; Y=(X.D), dre( L, D ).
```

we must realise that the if-then-else body can soundly replace appropriate calls for `dre` when doing partial evaluation of a program, whereas the one with the cut cannot, as the cut would apply to a different definition scope. One should, therefore, definitely prefer a structural if-then-else to its unstructured implementation with cut. For all but very simple definitions, however, it is inconvenient to trade multiple clauses for one single clause body, following the general principle of keeping local definitions concise in order to ease reasoning about program behaviour. What is needed, then, is a more direct structural syntax for clauses that yields an if-then-else meaning, giving us the expressive power to cascade if-then-elses across clauses. Here is our concrete alternative syntax for the original first clause of `dre`:

```
dre( X.L, D ) <- X in L <> dre( L, D ).
```

We call this an *iff-clause* due to the analogy with a biconditional (under an implication), in this case $(X \text{ in } L) \rightarrow (\text{dre}(X.L,D) \leftrightarrow \text{dre}(L,D))$. Procedurally, if a call unifies with the head `dre(X.L,D)` and the $(X \text{ in } L)$ condition succeeds, then solving `dre(L,D)` is the only way to solve the call; otherwise, the call's solutions must come from the next clauses. Such a clause therefore represents an if-then-else statement abstracted on the else part, to be plugged with the statement corresponding to the continuation of the definition. This compositionality for clauses is formalised in the paper by considering an extended syntax incorporating variable abstraction and application as in the λ -calculus. To make the abstract nature of clauses even more apparent we introduce syntax to lump them under a single occurrence of the procedure name, as follows (sugared syntax on the left, unsugared on the right).

<pre>dre :: X.L, D <- X in L <> dre(L, D) .. X.L, X.D <> dre(L, D) .. [], [] !.</pre>	<pre>dre :: X.L, D <- X in L <> dre(L, D) .. X.L, X.D <- true <> dre(L, D) .. [], [] <- true <> true</pre>
---	--

The programming style promoted by using clauses is to split the definition of a procedure into *cases*, typically related to certain patterns of arguments. Given the semantic possibility of multiple solutions, the central decision to be made when thinking about a case is whether it should be *exclusive*, i.e. precluding subsequent cases from being considered, or *inclusive*, when alternative solutions may come from subsequent cases. In *Cube* this is expressed by choosing one of two clause formats, an *iff-clause* $A \leftarrow C \langle \rangle B$ or an *if-clause* $A \leftarrow B$ (akin to $A : -B$ in Prolog), standing respectively for an implicit if-then-else or a disjunction, with the *else* or second disjunct a placeholder for the rest of the procedure definition. The choice is exemplified by comparing a multi-solution procedure for producing members of a list with one that just checks for membership of a given item (as in Prolog, H stands for $H \leftarrow \text{true}$).

```

member                has_member
:: X._, X              :: X._, X !
.. _.L, X <> member( L,X ).  .. _.L, X <> has_member( L,X ).

```

The syntax promotes quick recognition of whether a clause is exclusive or inclusive, through the presence or absence of a *single* occurrence of $\langle \rangle$ (or its sugared variant $!$). The syntax remains very close to that of Prolog; the whole point is to provide a much cleaner semantics, as the paper will show, with minor syntactic adjustments.

3 The true power of cut: until

The power of if-then-else, and therefore of iff-clauses, is enough to define many other useful control constructs found in Prolog, such as `once`, `not` or `var`. If, however, we want to define operations that can stop the production of solutions after possibly more than one, if-then-else is no longer enough and we need a more powerful `until` construct. A simple and instructive way to convey its meaning is to write its definition in Prolog:

```
Solve until Stop :- Solve, ( Stop, ! ; true ).
```

The solutions for `(Solve until Stop)` are the initial ones of `Solve` for which `Stop` fails, plus the first (if ever) for which `Stop` succeeds, and then no more. Notice that we need to use a cut in a conjunctive disjunct of a disjunctive conjunct, precisely the kind of context where a cut achieves its full power.

Having `until` as a basic primitive, along with conjunction, disjunction and unification, we can implement if-then-else. We take this opportunity to propose the notation `(If -> Then -; Else)`, avoiding Prolog's bad overload of `“;”` for both disjunction and the else separator. We use `until` as an infix operator binding tighter than the comma, i.e. $(A \text{ until } C, B) \equiv ((A \text{ until } C), B)$. The implementation uses an auxiliary variable `R` to convey the result of the test (`t` for taking the ‘then’ branch, `e` (else) otherwise), bindable only once ($X \text{ until true} \equiv \text{once } X$).

```

( If -> Then -; Else ) <> ( If, R=t ; R=e ) until true,
                          ( R=t, Then ; R=e, Else ).

```

Another useful derived construct is `unless`, akin to `until` but failing rather than succeeding after the stop condition holds. We implement it with a result variable that is bound only upon a successful test, then causing final failure:

Solve unless Stop <> Solve until (Stop, R=f), R=s.

With `unless` we can write e.g. a clean local read-process repeat-fail loop:

```
( repeat, read(Item) ) unless Item=end_of_file, process(Item), fail
```

To show that `until` really holds the full power of Prolog's cut we write an interpreter for Prolog in our cut-free language. The presentation is simplified by the use of iff-clauses and `unless`, knowing that both are implementable with `until`. We assume a unary procedure `system` for identifying system calls (`true` being one).

```
execute
:: G <> exec(G,R) unless R=fail.
exec
:: (A,B), R <> exec(A,RA), ( RA=fail, R=fail ; exec(B,R) )
.. (A;B), R <> exec(A,R) ; exec(B,R)
.. (!), R <> R=succ ; R=fail
.. G, succ <- system(G)
           <> G
.. G, R <> ( clause(G,B), exec(B,R) ) unless R=fail.
```

The main idea is that `exec` may succeed with two results in its second argument: `succ` signals true success, coming from the base `true` (a `system` call) or a cut's initial success; backtracking past a cut, though, actually succeeds again but with the `fail` result; this fake success is propagated through conjunctions (from or bypassing the second conjunct) and disjunctions (any branch), eventually exiting the `exec(B,R)` call of a `clause` body B, at which point the `unless` condition succeeds for the first time, immediately calling off the production of more solutions for G from any pending choices in `exec` or `clause` (similarly for the top goal in the `execute` clause).

We now turn to the problem of formally characterising the compositional semantics behind this reconstruction of Prolog along structured principles, aiming at a precise and clear understanding of program behaviour.

4 Syntax

It helps in the formalisation to consider an *abstract* syntax embodying the structural principles onto which the concrete syntax maps. We adopt a simple abstract syntax that is universal, given its suitability to encode the syntax of various formal calculi in the absence of predefined semantic roles for its constructs.

4.1 Terms

We use variables and a special null as basic terms, and three constructors: for pairing two terms (which, together with the null, provide lists), for applying a constant to a list (giving us rooted terms) and for abstracting a variable in a term (scoping).

Definition 1

The identifiers are a set of *constants* \mathcal{C} and a totally ordered countably infinite set \mathcal{V} of *variables* disjoint from \mathcal{C} . The *terms* are the smallest set \mathcal{T} satisfying

$$\mathcal{T} = \{\square\} \cup \mathcal{V} \cup (\mathcal{T} \times \mathcal{T}) \cup (\mathcal{C} \times \mathcal{L}) \cup (\{\lambda\} \times \mathcal{V} \times \mathcal{T}) \quad (1)$$

where \square is the *null*, $(t, t') \in (\mathcal{T} \times \mathcal{T})$ is a *pair*, $(c \triangleleft l) \in (\mathcal{C} \times \mathcal{L})$ is an *application*, $(\lambda v \cdot t) \in (\{\lambda\} \times \mathcal{V} \times \mathcal{T})$ is an *abstraction*, and the *lists* are the smallest set $\mathcal{L} \subset \mathcal{T}$ satisfying $\mathcal{L} = \{\square\} \cup (\mathcal{T} \times \mathcal{L})$.

Notice that formally a constant $c \in \mathcal{C}$ is not a term, but c in the concrete syntax corresponds to the term $c \triangleleft \square$. The familiar notation $c(t_1, \dots, t_n)$ for structured terms in the concrete syntax corresponds abstractly to $c \triangleleft (t_1, (\dots, (t_n, \square) \dots))$, also represented in our metalanguage by $c \triangleleft [t_1, \dots, t_n]$, following Prolog's tradition. This inevitably reminds us of the `univ (=..)` system predicate in Prolog, which does indeed correspond to the construction/deconstruction of an application term from/into its two components.

Abstraction terms have the usual intrinsic syntactic property of scoped variable capture, seen in the next definition.

Definition 2

The *free variables* \widehat{t} of a term $t \in \mathcal{T}$ are defined recursively as follows.

$$\begin{aligned} \widehat{c \triangleleft l} &= \widehat{l} & \widehat{\square} &= \emptyset \\ \widehat{a, b} &= \widehat{a} \cup \widehat{b} & \widehat{v} &= \{v\} \quad \text{if } v \in \mathcal{V} \\ \widehat{\lambda v \cdot t} &= \widehat{t} \setminus \{v\} \end{aligned}$$

In this universal syntax we are able to encode type-free λ -calculus terms using $\{\lambda v \cdot e\} = \lambda v \cdot \{e\}$ and $\{(a)b\} = (\{a\}, \{b\})$, predicate calculus formulae with $\{\forall v F\} = \forall \triangleleft [\lambda v \cdot \{F\}]$ (assuming $\forall \in \mathcal{C}$), etc., with no predefined semantics for the syntax on its own. Only for a certain intended use—in context—may terms and constants acquire a particular meaning, formalised in a semantics.

In the sequel we shall use both abstract and concrete syntax, according to contextual convenience.

4.2 Clauses, procedures and programs

The true power of any programming paradigm comes from the ability to define procedures and interpret certain expressions as procedure calls, so we proceed with a syntactic characterisation of procedures in our framework.

A procedure definition is built from sequences of clauses (in the concrete syntax) and captured in a single term (in the abstract syntax) that provides the semantics of calls to the procedure. We use certain terms to encode open (partial) definitions, such as clauses, and a simple syntactical composition for building larger open definitions from smaller ones. Closing an open definition is a simple operation.

The main intuition is that a procedure call is an *application* term with a root constant and a list of arguments, whereas a closed procedure definition for the root constant is an *abstraction* term to be applied (in the λ -calculus β -reduction sense) to the call's argument list, resulting in a term to be further evaluated.

For example, the following definition, taken as closed,

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

associates `int` to the abstraction term $\lambda A \cdot (\lambda X \cdot (\lambda A = [0]; \lambda X \cdot (\lambda A = [s(X)], \text{int}(X))))$. The term is very reminiscent of Clark's completion (Clark 1978), as should be expected in this example. But whereas Clark's construction always uses disjunction when composing clauses, ours may use *if-then-else* rather than disjunction, when composing exclusive rather than inclusive clauses. Let us look, then, at the process of building a complete procedure definition.

A clause stands for (a base case of) a *partial* definition—an abstraction term abstracting away the argument list and the alternative continuation of the definition, plus any local variables, under which we find either a disjunction or an *if-then-else*, respectively for *if*-clauses or *iff*-clauses, as this general translation shows:

$$\begin{array}{ll} a_1, \dots, a_n \leftarrow B & \lambda A \cdot \lambda D \cdot (\lambda v_1 \cdots \lambda v_k \cdot (\lambda A = [a_1, \dots, a_n], B) ; D) \\ a_1, \dots, a_n \leftarrow C \langle \rangle B & \lambda A \cdot \lambda D \cdot (\lambda v_1 \cdots \lambda v_k \cdot (\lambda A = [a_1, \dots, a_n], C \rightarrow B -; D)) \end{array}$$

with v_1, \dots, v_k the variables in each clause. Take as an example the following two clauses for a binary procedure:

```
X, a <- r(X).
1, b <> true.
```

The *if*-clause is equivalent to the following partial definition

$$p_1 = \lambda A \cdot \lambda D \cdot (\lambda X \cdot (\lambda A = [X, a], r(X)); D),$$

while the *iff*-clause represents this other one

$$p_2 = \lambda A \cdot \lambda D \cdot (\lambda A = [1, b] \rightarrow \text{true} -; D).$$

The effect of putting one clause after another can be defined as a generic syntactic composition of partial definitions,

$$p' \text{ after } p = \lambda A \cdot \lambda D \cdot (\lambda A \bullet ((p' \bullet A) \bullet D))$$

where $(\lambda x \cdot t) \bullet a$ stands for the replacement by a of all free occurrences of x in t , i.e. the equivalent of applying the β -rule in the λ -calculus to $(\lambda x \cdot t)a$.

In our example we can compose in two ways:

$$\begin{array}{ll} p_2 \text{ after } p_1 & = \lambda A \cdot \lambda D \cdot (\lambda X \cdot (\lambda A = [X, a], r(X)); (\lambda A = [1, b] \rightarrow \text{true} -; D)), \\ p_1 \text{ after } p_2 & = \lambda A \cdot \lambda D \cdot (\lambda A = [1, b] \rightarrow \text{true} -; (\lambda X \cdot (\lambda A = [X, a], r(X)); D)). \end{array}$$

Typically clauses are composed in their textual order. In a modular version of the language one may wish to compose a generic definition after a more specific one (defaults after overriding exceptions) or the other way around (specific cases uncovered by general ones).

Closing a partial definition is simple. With $\llbracket \text{fail} \rrbracket(s) = \emptyset$ (e.g. `fail` \equiv `a=b`),

$$\text{close } p = \lambda A \cdot (\lambda A \bullet \text{fail}).$$

After closing a definition, the abstraction of the definition's continuation has vanished. There is one outer abstraction on the argument list, and any remaining abstractions are for local clause variables.

A *program* is formally a mapping from constants to appropriate abstraction terms standing for closed procedure definitions. Cases for different argument arities under

the same constant are possible, as in Prolog; grouping them together, rather than by constant/arity pairs, is just a technical convenience.

5 Semantics

We are interested in defining *denotational* semantics for our terms, capturing their solution-producing behaviour when invoked as goals (*tasks*, we prefer to call them).

What are the basic intuitions for defining the semantics? A task generally has variables, and its behaviour results in solutions for them, expressed as constraints on the variables' possible valuations (as ground terms), a notion we call a *setting*. A task is launched in the context of an initial setting (a previously executed task may share some of the variables) and any solution is a possibly more constrained setting satisfying the initial one. A task may produce more than one solution, in a well-defined order. The sequence of solutions may be infinite, or else the task ends up by either finitely failing or diverging in the search for more solutions.

5.1 Settings, outcomes and behaviours

For constructing our semantic domain, then, we wish to define a setting providing partial information on some variables' possible valuations. It should satisfy the very abstract characterisation we gave in (Monteiro and Porto 1998) of a structure with a partial order of entailment and consistency completeness, adequate for arbitrary constraint logic programming languages. In this paper we restrict ourselves to a Prolog-like language handling only identity constraints through unification, and define settings accordingly.

Formalising a setting of identity constraints may vary in the degree of abstraction. In the WAM (Ait-Kaci 1991) implementation of Prolog we can see settings as accumulated equations of variables to terms, but these are too concrete. Striving to approach full abstraction, we opt instead for substitutions expressing the solved form of those equalities, actually the view commonly held by programmers. However, our settings will formally differ from standard substitutions, to cater for two needs (clarified ahead): ideal infinite terms, and an explicit scope of variables.

Definition 3

The *ideal terms* ${}^I\mathcal{T}$ are the largest (not smallest) solution set for \mathcal{T} in equation (1). Definition 2 of free variables carries over from \mathcal{T} to ${}^I\mathcal{T}$ by assuming the smallest set satisfying the equations. The *substitutions* Σ are the mappings $\sigma : V \rightarrow {}^I\mathcal{T}$ from a finite set of variables $V \subset_{\text{fin}} \mathcal{V}$ to ideal terms where they do not occur free, $v' \in \widehat{\sigma(v)} \Rightarrow v' \notin \text{dom}(\sigma)$, and not mapping a variable to a smaller one (they are totally ordered), $\sigma(v) \in \mathcal{V} \Rightarrow v > \sigma(v)$. Considering $\widehat{\sigma} = \{\widehat{\sigma(v)} \mid v \in \text{dom}(\sigma)\}$ the free variables of a substitution $\sigma \in \Sigma$, the *settings* \mathcal{S} are the pairs $V:\sigma$ of a variable *scope* $V \subset_{\text{fin}} \mathcal{V}$ and a substitution $\sigma \in \Sigma$ under the scope, with $\text{dom}(\sigma) \cup \widehat{\sigma} \subseteq V$.

The difference ${}^I\mathcal{T} \setminus \mathcal{T}$ between ideal and regular terms are the so-called *infinite* terms. Their possible appearance in substitutions reflects most Prolog implementations, that by omitting the expensive occurs check in unification may generate

solutions corresponding to infinite *rational trees*, as indeed proposed by Prolog’s inventor (Colmerauer 1993). The definition of substitution conveys the idea that the implicit equations are sufficiently unfolded into a “solved form”. For example, $\sigma = \{X = f(Y), Y = a\}$ is not a substitution, as the domain variable Y occurs in $\sigma(X)$. The requirement on variable order aids in the determinacy of settings, by following the WAM’s policy when binding a pair of free variables.

Now we come to the notion of *outcome*, to express the deterministic result of executing a task in the context of a given setting, as a sequence of solutions and termination status. For example, the task `member(1.X,Y)` launched in the setting $\{X, Y\} : \{X = [2]\}$ yields a first solution $\{X, Y\} : \{X = [2], Y = 1\}$, then after backtracking a second one $\{X, Y\} : \{X = [2], Y = 2\}$, and if retried again finitely fails. A semantics based on sequences of solutions is not new, having been proposed for algebraic logic programming (Seres et al. 1999); ours differs in the form of those solutions (settings) and the inclusion of termination status.

Definition 4

The *outcomes* are $\mathcal{O} = \mathcal{O}^f \cup \mathcal{O}^s$, with $\mathcal{O}^f = \{\emptyset, \infty\}$ the *final* outcomes and \mathcal{O}^s the *successful* outcomes—the greatest set satisfying $\mathcal{O}^s = \{s.o \mid s \in \mathcal{S}, o \in \mathcal{O}\}$.

The symbols \emptyset and ∞ represent the final outcomes of, respectively, finite failure and divergence. A successful outcome is a non-empty sequence of solutions, either infinite or terminated by a final outcome.

We want to capture the variability of settings in which a task is executed, affecting its outcome. It becomes relevant to define the entailment relation on settings, to support the intuition that the outcome of a task can only have solutions with equal or stronger constraints than the setting at the start, i.e. entailing it.

Definition 5

For any ideal term t and substitution σ , let $t[\sigma]$ denote the ideal term obtained by replacing in t any occurrence of a variable $v \in \text{dom}(\sigma) \cap \hat{t}$ by $\sigma(v)$. The *entailment* $\sigma' \vdash \sigma$ between substitutions is defined by $\sigma(v)[\sigma'] = \sigma'(v)$ for every $v \in \text{dom}(\sigma) \subseteq \text{dom}(\sigma')$. For settings, $V' : \sigma' \vdash V : \sigma$ whenever $V' \supseteq V$ and $\sigma' \vdash \sigma$. We say that $o \in \mathcal{O}$ is an outcome *upon* a setting $s \in \mathcal{S}$, written $o \vdash s$, if all the solutions entail it, i.e. with $S(o)$ denoting the *set* of settings in o we have $s' \in S(o) \Rightarrow s' \vdash s$.

Just before finally defining behaviours for tasks, we remark that the setting in which a task is executed must have a scope covering the task’s free variables, but possibly also some other variables from the task’s original lexical scope (typically a program clause). It becomes convenient to index behaviours by sets of variables covered by, rather than equal to, the scopes of the involved settings.

Definition 6

$\mathcal{S}_V = \{V' : \sigma \in \mathcal{S} \mid V \subseteq V' \subset_{\text{fin}} \mathcal{V}\}$ are the settings *covering* V . The *behaviours* are $\mathcal{B} = \bigcup_{V \subset_{\text{fin}} \mathcal{V}} \mathcal{B}_V$, with \mathcal{B}_V , the behaviours *for* V , being the mappings $\beta : \mathcal{S}_V \rightarrow \mathcal{O}$ from settings covering V to outcomes upon them, i.e. such that $\beta(s) \vdash s$.

The denotational semantics for terms, taken as tasks, is a mapping $\llbracket \cdot \rrbracket : \mathcal{T} \rightarrow \mathcal{B}$ into behaviours for the terms’ free variables, i.e. $\llbracket t \rrbracket \in \mathcal{B}_{\hat{t}}$. Notice that, according

to the given definitions, the denotation of a task yields outcomes for initial settings whose scope is a superset of the task's free variables. The denotation mapping $\llbracket \cdot \rrbracket$ must satisfy certain equations for a class of *special* terms \mathcal{T}^* that have a predefined compositional way of being interpreted as tasks, whereas the other terms are interpreted in the context of a given program. We now proceed to introduce the members of \mathcal{T}^* , along with their fixed denotation equations. To lighten the presentation we use concrete infix operator syntax (rather than abstract) for such terms.

5.2 Disjunction

The denotational semantics of a disjunctive term $(a;b) \in \mathcal{T}^*$ is given through a semantic *sum* operation $\oplus : \mathcal{O} \times \mathcal{O} \rightarrow \mathcal{O}$, as follows.

$$\begin{aligned} \llbracket a;b \rrbracket(s) &= \llbracket a \rrbracket(s) \oplus \llbracket b \rrbracket(s) & \infty \oplus o' &= \infty \\ & & \emptyset \oplus o' &= o' \\ (s.o) \oplus o' &= s.(o \oplus o') \end{aligned}$$

Each disjunctive sub-term is evaluated in the same initial setting—the essence of the backtracking process that implements this semantics. Failure of the first disjunct leads to collecting the solutions of the second, and divergence of the first naturally extends to the whole disjunction.

5.3 Conjunction

For the semantics of a conjunctive term $(a,b) \in \mathcal{T}^*$ we use a semantic *product* operation $\otimes : \mathcal{O} \times \mathcal{B} \rightarrow \mathcal{O}$ that relies on the sum, as follows.

$$\begin{aligned} \llbracket a,b \rrbracket(s) &= \llbracket a \rrbracket(s) \otimes \llbracket b \rrbracket & \infty \otimes \beta &= \infty \\ & & \emptyset \otimes \beta &= \emptyset \\ (s.o) \otimes \beta &= \beta(s) \oplus (o \otimes \beta) \end{aligned}$$

The evaluation of the second conjunct is performed upon each solution of the first (yielding stronger solutions). As expected, both failure and divergence of the first are absorbing.

5.4 Until

The intended behaviour of a term $(a \text{ until } b) \in \mathcal{T}^*$ is to provide the solutions of a but checking, upon each one, whether b has a successful outcome, in which case the corresponding solution is the last to be provided. This is achieved by a *pruning* operation $\oslash : \mathcal{O} \times \mathcal{B} \rightarrow \mathcal{O}$, as follows.

$$\begin{aligned} \llbracket a \text{ until } b \rrbracket(s) &= \llbracket a \rrbracket(s) \oslash \llbracket b \rrbracket & \infty \oslash \beta &= \infty \\ & & \emptyset \oslash \beta &= \emptyset \\ (s.o) \oslash \beta &= \begin{cases} \infty & \text{if } \beta(s) = \infty \\ s.(o \oslash \beta) & \text{if } \beta(s) = \emptyset \\ s'.\emptyset & \text{if } \beta(s) = s'.o' \end{cases} \end{aligned}$$

Notice how the first successful solution s' of the pruning condition is taken as the final global solution (it entails the solution s for the pruned task), discarding further solutions from both the condition (σ') and the pruned task (σ). As expected, failure of the pruned task and divergence of either task are absorbing.

5.5 Unless and if-then-else (revisited)

Although from the perspective of minimal semantic ingredients **unless** and **if-then-else** are not primitive constructs, being definable through **until**, it is enlightening to see them defined directly in our semantic framework.

For **unless** we need a very simple variation on the pruning operator of **until**, obtained by replacing, in the last line of the definition above, $s'.\emptyset$ with just \emptyset .

The implementation of **if-then-else** given in section 3 matches this definition:

$$\llbracket \text{if } \rightarrow \text{ then } - ; \text{ else } \rrbracket(s) = \begin{cases} \infty & \text{if } \llbracket \text{if} \rrbracket(s) = \infty \\ \llbracket \text{else} \rrbracket(s) & \text{if } \llbracket \text{if} \rrbracket(s) = \emptyset \\ \llbracket \text{then} \rrbracket(s') & \text{if } \llbracket \text{if} \rrbracket(s) = s'.\sigma \end{cases}$$

Ahead in section 6 we discuss an alternative meaning adopted by other languages.

5.6 Unification - the prime mover

Any computational engine using the given compositional interpretation for the three operators must also define the denotational semantics for some terms that act as the basis for change, building stronger settings from previous ones. Here we assume as basic just the operation of *unification* of two terms, captured syntactically by special terms $(a = b) \in \mathcal{T}^*$.

$$\llbracket a = b \rrbracket(V:\sigma) = \begin{cases} \infty & \text{if } U(a, b, \sigma) \not\rightarrow \\ \emptyset & \text{if } U(a, b, \sigma) \rightarrow \perp \\ (V:\sigma').\emptyset & \text{if } U(a, b, \sigma) \rightarrow \sigma' \in \Sigma \end{cases}$$

The unification $U(a, b, \sigma)$ of a and b under σ may succeed with an equal or stronger substitution $\sigma' \vdash \sigma$, yield failure (\perp) or diverge (when unifying infinite terms). A successful unification $U(a, b, \sigma) \rightarrow \sigma'$ yields the *least* substitution σ' , in the partial order of entailment, that makes a and b identical, $a[\sigma'] = b[\sigma']$. We can formalise unification by an inductive definition, adapting to our more ideal framework the classical definition introduced by Robinson for the predicate calculus but without the occurs-check, in the spirit of Colmerauer's suggestion and in accordance with most Prolog implementations, although not mandated by its standard (Deransart et al. 1991). This being quite well known we omit the details.

Although abstraction terms are included in our syntax and implicit in clauses, they never actually appear inside unification, if no explicit concrete syntax for them is available. Otherwise unification has to handle also α -conversion equivalence.

5.7 Atomic terms

In our abstract syntax the only atomic terms are the variables $v \in \mathcal{V}$ and the null \square . Since pairs were given the semantics of conjunction, the natural extension to lists

is to treat the null as special ($\llbracket \] \in \mathcal{T}^*$), with the idle successful outcome:

$$\llbracket \](s) = s. \emptyset$$

We equate **true** in the concrete syntax to the abstract null $\llbracket \]$.

Variables are also special ($\mathcal{V} \subset \mathcal{T}^*$), being interpreted under the setting. A resulting free variable has no procedural meaning, yielding a finite failure outcome.

$$v \in \mathcal{V} \quad \Rightarrow \quad \llbracket v \rrbracket(V:\sigma) = \begin{cases} \emptyset & \text{if } \sigma(v) \in \mathcal{V} \\ \llbracket \sigma(v) \rrbracket(V:\sigma) & \text{otherwise} \end{cases}$$

This simple definition captures the quite useful higher-order feature of Prolog-like languages exemplified by `(build_task(Data,Task), Task, process(Data))`, where the **Task** variable is first bound to a term (sharing variables with a **Data** pattern) that gets to be executed as a task (instantiating the **Data** to be processed).

5.8 Procedure call

The denotational semantics of a non-special term $(p \triangleleft a) \in \mathcal{T} \setminus \mathcal{T}^*$, interpreted as a procedure call, is parametric on a given program P , as follows.

$$\llbracket p \triangleleft a \rrbracket_P = \llbracket P(p) \bullet a \rrbracket.$$

We have seen that $P(p)$, the closed definition for p in the program P , is always an abstraction term whose outermost abstraction is on the argument list. Taking the example of **int** in section 4.2, a call `int(s(a))` \equiv `int \triangleleft [s(a)]` results in applying the **int** abstraction to the call's argument list `[s(a)]`, resulting in the task

$$([s(a)] = [0]; \lambda X. ([s(a)] = [s(X)], \text{int}(X)))$$

whose behaviour, since that of the first disjunct yields \emptyset (unification failure), is the behaviour of the inner abstraction $\lambda X. ([s(a)] = [s(X)], \text{int}(X))$. This term has an implicit existential reading of **X** as a clause variable, and its launch as a task starts by replacing the abstracted variable in the inner term, before executing it, with a fresh new variable for the current setting—the analogue of using clause variants in resolution—as defined next.

The denotational equation given above states the correctness of unfolding a procedure call with its procedure definition. This is what makes e.g. partial evaluation much easier for this structured language than for Prolog, where cuts in procedure definitions make such unfolding unsound due to the scope extrusion of the cuts.

5.9 Abstraction

Abstraction terms $(\lambda x. t) \in \mathcal{T}^*$ come from clause variable scoping in procedure definitions. Invoked as tasks they give rise, as mentioned, to the creation of new variables for the solutions of the clause case. Formally,

$$\llbracket \lambda v. t \rrbracket(V:\sigma) = \llbracket (\lambda v. t) \bullet \check{V} \rrbracket(V \cup \{\check{V}\}:\sigma)$$

with \check{V} being the function, implicitly defined by the countable order on \mathcal{V} , that returns the least variable greater than those in V . This justifies the need for the scope in settings, formalising how the stack grows in the WAM implementation.

5.10 Fixpoint semantics

If a term's structure is composed solely of special terms, the corresponding recursive equations uniquely define the term's denotational semantics. This is no longer the case for procedure calls, because of the circularity introduced by recursive definitions. The standard solution in logic programming is to define a mapping from programs to continuous operators on the possible interpretations and give the semantics of a program as the least fixpoint of its operator (van Emden and Kowalski 1976). We will proceed likewise, but for our different semantic domain.

The *interpretations* \mathcal{I} are the functions $I : \mathcal{T} \rightarrow \mathcal{B}$ that map each term into a behaviour for its free variables, $I(t) \in \mathcal{B}_{\hat{\tau}}$, and satisfy the equations given for *special* terms when I is taken for $\llbracket \cdot \rrbracket$. Interpretations differ, then, in the behaviours of the non-special terms, i.e. the procedure calls.

We define a partial order on interpretations based on that of outcomes,

$$I \sqsubseteq I' \iff \forall t \in \mathcal{T} \forall s \in \mathcal{S}_{\hat{\tau}} I(t)(s) \sqsubseteq I'(t)(s),$$

the partial order on outcomes being the greatest relation that satisfies

$$o \sqsubseteq o' \iff (o = \infty) \vee (o = o' = \emptyset) \vee (o = s.u, o' = s.u', u \sqsubseteq u').$$

Notice that having $o \sqsubseteq o'$ with $o \neq o'$ is possible only when o ends in ∞ after a common (finite) prefix with o' . Intuitively this can be understood as o and o' being partial outcomes for the same task but with fewer computation steps available to produce o , reflected in the divergence “termination”.

The *call step transformer* $S_P \in \mathcal{I}^{\mathcal{I}}$ for a given program P maps an interpretation I into an interpretation $S_P(I)$ such that, for any non-special term $(p \triangleleft a) \in \mathcal{T} \setminus \mathcal{T}^*$,

$$S_P(I)(p \triangleleft a) = I(P(p) \bullet a).$$

S_P is continuous (we omit the proof) and has a least fixpoint which is the semantics (the model) of the program P , satisfying the semantic equation for procedure calls.

5.11 Abduction

An interesting semantic insight is to interpret the behaviour of unification tasks as performing abduction (Kakas et al. 1993). A setting $V : \sigma$ can be thought of as a theory $\Theta(V : \sigma) = \{v=t \mid (v,t) \in \sigma\}$ in a variable-free logic language where V are considered Skolem constants interpreted in the realm of ideal terms, and the single predicate symbol ‘=’ is interpreted under the standard equality axioms \mathcal{E} . Whenever $\llbracket a=b \rrbracket(s) = s'.\emptyset$ we can see that $\Theta(s')$ is a minimal consistent extension of $\Theta(s)$ such that $\Theta(s') \cup \mathcal{E} \models a=b$, and if no such extension exists then $\llbracket a=b \rrbracket(s) = \emptyset$. We spot here the hallmarks of abduction, and indeed the outcome solutions may be seen as the abductive extensions that make true the equality statements implicit in the unifications along the way.

The pruning semantics, interestingly, can establish another way of relating tasks to abduction. Calls to the procedure (`possible X <> not not X`), and to (`var X <> possible X=A, possible X=b`), are actually statements about abducibility in the *current* setting, rather than requirements for abductive extension. The given

`var` definition reads directly as “it is currently possible to abduce equality of X to both a and b ”. This semantic dependency on the current setting, rather than a final solution, clearly explains why conjunction is not commutative, e.g. $(\text{var}(X), X=a)$.

6 Language design: alternatives and extensions

We presented `until` as the basic semantic ingredient for achieving the power of Prolog’s cut, but in practice several derived constructs are available to the programmer, such as `not`, `once` or `if-then-else`. The latter is pervasive, being the implicit underpinning of exclusive clauses, the vast majority in real programs. The meaning we took for `if-then-else` is expressed in its definition in section 5.5—only the first solution of the *if* condition, if it exists, is retained as initial setting for the *then* part. Alternatively, the designers of e.g. NU-Prolog (Naish 1986) and Mercury (Somogyi et al. 1996) have chosen, on the grounds of it being more declarative and logically sound, to use *all* solutions of the condition. This is just as easy to define, using $\llbracket \textit{if} \rrbracket(s) \otimes \llbracket \textit{then} \rrbracket$ instead of $\llbracket \textit{then} \rrbracket(s')$ in the third case of the definition. Our choice was pragmatic, being aligned with Prolog and validated by usefulness in applications. We never encountered a real need for the supposedly more declarative reading of `if-then-else`, even though consciously on the lookout for it. We did provide in `Cube` a related *otherwise* construct yielding all solutions of its *if* part, but also found it wanting of applicability. Interestingly, we point out that while our reading of `if-then-else` can be implemented with unification, conjunction, disjunction and `until`, this is not the case for the alternative reading. It must be either provided as another primitive, or implemented with side-effects (to remember that *if* had solutions). So, what does “declarative” mean? One might argue that “declarative” is really about having a compositional semantics that is simple to understand, whether this is based in predicate logic and set-oriented or based in outcomes and sequence-oriented. Simple compositionality is what eases the task of reasoning about programs, by both human programmers and meta-level software tools.

The constructions presented in this paper are just the essential core for a language with real-world applicability, that must include several semantic extensions. A paramount example is arithmetic. Following Prolog’s way we must consider a (partial) denotation $\mathcal{A}\llbracket \cdot \rrbracket : \mathcal{T} \times \mathcal{S} \rightarrow \mathcal{T}$ that interprets a term under a setting as an arithmetic expression yielding a constant term representing a number, and define

$$\llbracket x \text{ is } e \rrbracket(s) = \begin{cases} \llbracket x=y \rrbracket(s) & \text{if } \mathcal{A}\llbracket e \rrbracket(s) = y \\ \emptyset & \text{otherwise} \end{cases}$$

Another practical requirement is the ability to generate and handle exceptions. For example, an exception is better than failure for the semantics of a free variable task. The formalisation requires the introduction of a third type of final outcome, the *exception* $\{t\} \in \mathcal{O}^f$ with a term $t \in \mathcal{T}$ conveying contextual information. The semantic equations handling exception in sum, product and pruning are similar to those for divergence. A special task must be introduced to throw an exception,

$$\llbracket \text{throw}(t) \rrbracket(V:\sigma) = \{\sigma(t)\}$$

and another one for catching it,

$$\begin{aligned} \llbracket \text{catch}(task, exc, handler) \rrbracket(s) &= \llbracket task \rrbracket(s) \langle exc, handler \rangle s \\ \emptyset \langle x, h \rangle s &= \emptyset & (s.o) \langle x, h \rangle s &= s.(o \langle x, h \rangle s) \\ \infty \langle x, h \rangle s &= \infty & \{t\} \langle x, h \rangle s &= \begin{cases} \llbracket h \rrbracket(s') & \text{if } \llbracket x=t \rrbracket(s) = s'.\emptyset \\ \{t\} & \text{otherwise} \end{cases} \end{aligned}$$

Yet another unavoidable extension, in practice, is to have internal side-effects. The required change of the semantic domain is relatively simple, adding a persistent state alongside the setting. Lacking space here, this has to be reported elsewhere.

7 Conclusions and further work

We have shown that the expressive power of Prolog can be captured with three structural ingredients—disjunction, conjunction and *until*—plus unification, with a simple compositional denotational semantics handling the deterministic sequential nature of multiple solutions—equating variables to rational trees—and final outcomes of finite failure and divergence. For the first time the equivalent of Prolog’s cut has been given compositional semantics based solely on the state of variable bindings. The semantics are quite naturally extended to deal with exceptions and even side-effects, not presented here due to space limitations. It would be interesting to cast the semantics in a co-algebraic account. We have also defined, but not yet reported, a more concrete operational (step) semantics in terms of graph rewriting that nicely formalises the so-called 4-port model introduced for Prolog debugging.

Procedures are composed from clauses with a redesigned syntax, corresponding to abstractions of the alternative branch of either a disjunction or an *if-then-else*, the latter being an ubiquitous programming construct that is implementable with *until* (but not vice-versa). We may, therefore, express *if-then-else* chains across clauses, not just within one. The formalisation of clause composition uses an extended term syntax with variable abstraction as in the λ -calculus. This paves the way for a more ambitious endeavour to adapt the modularity style of contextual logic programming (Monteiro and Porto 1989) to naturally and properly handle defaults and exceptions and higher-order procedures, a great help for building complex applications. An issue worth exploring is the possible combination of sequence-based semantics with program parts having set-based semantics that can profit from computation techniques such as tabling. Another is a classification of procedures and call patterns according to their behaviour, and its impact on compilation.

The ideas in the paper have been turned into a practical alternative to Prolog, easy to program and debug, and more readily amenable to partial evaluation, important for compile-time optimisation of clean high-level declarative code. The language—*Cube*—has been implemented on top of a Prolog system and heavily used to good effect in building a sophisticated large real-world application (Porto 2003). It incorporates several other features such as structural abstraction and application (Porto 2002) for higher order and functional notation. We currently work on its contextual modularity, for which we plan to write a modular partial evaluator.

References

- AIT-KACI, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press.
- ANDREWS, J. H. 2003. The witness properties and the semantics of the prolog cut. *Theory Pract. Log. Program.* 3, 1, 1–59.
- BOSSI, A., GABRIELLI, M., LEVI, G., AND MARTELLI, M. 1994. The s-semantics approach: Theory and applications. *The Journal of Logic Programming* 19/20, 149–197.
- CLARK, K. L. 1978. Negation as failure. In *Logic and Databases* (New York), H. Gallaire and J. Minker, Eds. Plenum Press, 293–322.
- COLMERAUER, A. 1993. Prolog and infinite trees. In *Logic Programming*, K. Clark and S.-A. Tärnlund, Eds. A.P.I.C. Studies in Data Processing, vol. 16. Academic Press, 231–251.
- DE BRUIN, A. AND DE VINK, E. 1989. Continuation semantics for PROLOG with cut. In *TAPSOFT '89; Proceedings of the International Joint Conference on Theory and Practice of Software Development*, J. Díaz and F. Orejas, Eds. Springer, 178–192.
- DEBRAY, S. K. AND MISHRA, P. 1988. Denotational and operational semantics for Prolog. *Journal of Logic Programming* 5, 1, 81–91.
- DERANSART, P., ED-DBALI, A., AND CERVONI, L. 1991. *Prolog: The Standard; reference manual*. Spinger.
- KAKAS, A., KOWALSKI, R., AND TONI, F. 1993. Abductive logic programming. *Journal of Logic and Computation* 2, 6, 719–770.
- LI, B. Z. 1994. A pi-calculus specification of Prolog. In *European Symposium on Programming*. 379–393.
- MONTEIRO, L. AND PORTO, A. 1989. Contextual logic programming. In *Logic Programming, Proceedings of the Sixth International Conference*, G. Levi and M. Martelli, Eds. MIT Press, 284–299.
- MONTEIRO, L. AND PORTO, A. 1998. Entailment-based actions for coordination. *Theoretical Computer Science* 192, 259–286.
- NAISH, L. July 1986. Negation and quantifiers in NU-Prolog. *Proceedings of the Third International Conference on Logic Programming*, 624–634.
- PORTO, A. 2002. Structural abstraction and application in logic programming. In *Functional and Logic Programming, 6th International Symposium, FLOPS 2002, Proceedings*, Z. Hu and M. Rodríguez-Artalejo, Eds. Lecture Notes in Computer Science, vol. 2441. Springer, 275–289.
- PORTO, A. 2003. An integrated information system powered by Prolog. In *Practical Aspects of Declarative Languages, 5th International Symposium, Proceedings*, V. Dahl and P. Wadler, Eds. Lecture Notes in Computer Science, vol. 2562. Springer, 92–109.
- SERES, S., SPIVEY, M., AND HOARE, T. 1999. Algebra of logic programming. In *Proceedings of the 1999 international conference on Logic programming*. Massachusetts Institute of Technology, Cambridge, MA, USA, 184–199.
- SOMOGYI, Z., HENDERSON, F., AND CONWAY, T. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming* 29, 1–3, 17–64.
- SPOTO, F. 2000. Operational and goal-independent denotational semantics for prolog with cut. *Journal of Logic Programming* 42, 1, 1–46.
- VAN EMDEN, M. H. AND KOWALSKI, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM* 23(4), 733–742.