Specifying Prolog Trace Models with a Continuation Semantics*

Erwan Jahier[1], Mireille Ducassé[1], and Olivier Ridoux[2]

[1] IRISA/INSA, jahier@irisa.fr, ducasse@irisa.fr
[2] IRISA/IFSIC, ridoux@irisa.fr
Campus Universitaire de Beaulieu, F-35042 RENNES Cedex - France

Abstract Byrd's box model is a fine grained Prolog execution model that can be the basis of high level debugging tools. In this article, we provide a formal specification of Byrd's box model, based on an already existing operational and denotational continuation semantics for Prolog with cut. We show how this specification can be extended to specify richer Prolog trace models. To be able to experiment with various Prolog trace models, we translate these specifications into λProlog. This translation leads to a Prolog interpreter that performs execution traces. We have hence a formal framework to specify, prototype, and validate trace models for Prolog.

1 Introduction

Byrd's box model [9] is a very fine grained Prolog execution model that provides a precise but verbose image of the execution. It is sometimes stated that it should not be used in debuggers. Whether Byrd's trace is a proper output format for an end-user may indeed be discussed. A trace, however, can be the basis of higher level tools as we have shown for debugging [13] or monitoring [16]. In general, automated dynamic analysis needs an execution model to be based upon, and Byrd's box model is a good candidate for logic programming languages.

As detailed by Tobermann and Beckstein [31], Prolog debugging systems have quite different interpretations of what Byrd's box model is. It is not always clear whether theses differences come from a misinterpretation of the original model or from a will to improve it. In this article we propose a formal specification of Byrd's box model. One can conjecture that a formal specification would avoid misinterpretations. Indeed, people would have the formal framework to prove that their model conforms to the specification. Furthermore, people enhancing the model would then be able to state precisely what the enhancements are. There is a large number of slightly different execution models for Prolog [15, 21,

* Extended Abstract of LOPSTR 2000, Tenth International Workshop on Logic-based Program Synthesis and Transformation, 24-28 July 2000, London, UK. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1. http://www.cs.man.ac.uk/cstechrep/titles00.html

23, 29, 31]. This means that there is no best model. It is therefore important to be able to easily specify model variants. The validation of the formal specification becomes then an essential feature if one wants to reason about the generated trace. We show how we can extend a model with our formalism and how we can validate it experimentally by animating the specification. We show how this can be done in this article. We also show how various execution counters can be added with that formalism.

An operational semantics formally specifies the way a program is executed. Since a trace reflects the program execution, relying on an existing formal operational semantics to specify a trace is natural. Our formal specification of Byrd's box model is based on the Prolog operational semantics given by Nicholson and Foo [25]. De Bruin and De Vink proposed a very similar semantics [11] at the same time. This semantics is simple, concise, and it captures very elegantly the cut predicate ('!') behavior. Moreover, this semantics is compositional; this is an important property since it allows simpler program properties proofs to be carried; proofs are local and can be carried by case on the syntactic structure of the semantics. We instrument this semantics in such a way that it specifies how to compute query answers as well as Byrd's execution traces associated to them. Further advantage of this semantics is that it can be directly translated into a λProlog program which leads to a Prolog interpreter. Whereas this interpreter is not an efficient implementation of a trace generator, it is a very convenient way to validate the formal specification in a pragmatic basis. One can test the specification on simple Prolog programs to convince oneself that the generated trace is indeed what one would expect.

Ducassé and Noyé describe [14] how to implement an extended Byrd box model by source-to-source transformation. They also show that this method, in terms of efficiency, is comparable with object code instrumentation, which is the way tracers are usually implemented. On one side we have a formal specification of a trace model which truly reflects the Prolog operational semantics, since it is built on top of it. On the other side, we have an efficient way to implement the specified trace model by source transformation on any Prolog system. As further work, we plan to describe this Prolog program transformation in the same formalism as the one used in this article, and then to prove that the implementation is correct with respect to the specification.

The contribution of this article is to give a formal specification of Byrd's box model, to show how this model can be extended, and thus to give a formal framework to specify trace models. We also show how to translate the specification into λProlog, which lets one easily prototype trace models. In Section 2 we present the operational semantics for Prolog that we use in Section 3 to formally specify Byrd's box model. We also show how it can be translated into a λProlog program. We define several extensions of this model in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 An operational continuation semantics for Prolog

Nicholson and Foo [25] define a continuation-based operational semantics for Prolog which captures very elegantly the cut behavior. In order to define an operational semantics for λProlog, Brisset and Ridoux [7] reformulate it by abstracting away the data flow and by handling the substitutions implicitly. This makes the semantics more concise. Since Byrd's box model is related to the control flow, (the Prolog subset of) this reformulation is a very good candidate to be the basis of our Prolog model specification. In the following, we briefly describe this semantics; more detailed descriptions can be found in [7] and [25].

Pettorossi and Proietti [26, 28] also proposed an operational semantics for Prolog, but it does not handle the cut. The direct style semantics proposed by Jones and Mycroft [17] as well as the one of Debray and Mishra [12] do handle the cut. But we believe that the semantics of Nicholson and Foo is more elegant because the search stack and the resolvent are not handled by explicit data-structures. This leaves open all kinds of actual representations of the search-stack and the resolvent. As a matter of fact, it is rather natural that continuations appeared to be well-suited to model the resolvent, the search stack and the cut since continuations were precisely designed to model stacks.

2.1 Description of the semantics

Figure 1 defines the Nicholson and Foo operational semantics for Prolog. We suppose that all the predicates are of arity 1, which is not a restriction as Prolog constructs can be used to deal with greater arities. We note *pred(P)* the set of predicates defined in the program P and *clauses(p, P)* the set of clauses defining the predicate p. Abstractions are written $\lambda x_1 ... x_n(E)$ where x_1, ..., x_n are λ-variables and E is a λ-expression. The application of the λ-term t_1 to the λ-term t_2 is written $(t_1\ t_2)$. Nested applications $((t_1\ t_2)\ t_3)$ are written $(t_1\ t_2\ t_3)$. The semantic function $\lambda x(\mathcal{S}[\![x]\!])$ takes a Prolog program as input, and translates it into an abstract machine language with only one instruction: *unify*. By evaluating all the *unify* instructions of a translated program query, we can simulate a Prolog interpretation. The semantic functions $\lambda x(\mathcal{S}_{pred}[\![x]\!])$, $\lambda x(\mathcal{S}_p[\![x]\!])$ and $\lambda x(\mathcal{S}_g[\![x]\!])$ inductively translate a predicate, a sequence of clauses, and a goal, respectively; $\lambda x([\![x]\!])$ translates Prolog terms into terms of the abstract machine language. Continuations denote code that remains to be executed. The success continuation, κ, is the code to be executed after a success; it controls the formation of proofs and can be thought of as the resolvent. The failure continuation, ζ, is the code to be executed after a failure and it can be thought of as the search stack. The cut failure continuation, ξ, is the code to be executed after a cut. Actually, the cut failure continuation is simply a copy of the failure continuation (see the two ζ's in Equation 1.4[1]). This copy is made when entering a predicate to be able to substitute it to the current failure continuation whenever a cut is encountered in that predicate (Equation 1.10).

[1] The convention we use is that Equation n.m refers to the mth equation of Figure n.

Combinator *unify* (Equation 1.1) takes two terms and two continuations. If the terms are unifiable, it outputs the success continuation. The failure continuation is also applied to it because the solutions contained in the failure continuation may be needed if other solutions are requested on backtracking. If the terms are not unifiable, *unify* simply outputs the failure continuation. Equation 1.2 tells how to formulate a top level query (e.g., `?- p(X).`) in terms of the semantics defined by Equations 1.3 to 1.10. In these equations, the type of continuations is very general; the only constraints are that cut failure and failure continuations should have the same type (see Equation 1.4); success continuations should be functions from failure continuations to failure continuations (see the definition of *unify*). Equation 1.2 imposes on failure and cut failure continuations to be lists of `yes` and `no`. For example, a query '$\mathcal{S}_{query}[\![P]\!]\ p\ X$' that leads to two solutions will produce the list `yes.yes.no.nil`. Therefore the types of failure, cut failure, and success continuations are:

`failureT` $\equiv$ `cutT` $\equiv$ `(list solutionT)`, where `solutionT = yes | no`
`successT` $\equiv$ `failureT` $\rightarrow$ `failureT`

Equation 1.3 translates a Prolog program P; $\mathcal{S}[\![P]\!]$ takes a predicate name p as input, and outputs the code of that predicate. If the predicate p is not part of the program P, then it is translated into the predicate $false$ ($\equiv \lambda\kappa\xi\zeta(\zeta)$). Equation 1.4 translates a Prolog predicate; it translates its defining clauses using Equations 1.5, 1.6, and 1.7. The cut failure continuation is initialized with the failure continuation for the current predicate; it will replace the failure continuation when a cut is encountered, removing in this way all the choice points that might have been created in Equation 1.5. Equation 1.5 inductively translates a conjunction of clauses $C_1.C_2$. The translation of the first part of the conjunction of clauses, C_1, takes as arguments the predicate argument ε, the success continuation κ, and the cut failure continuation ξ of the whole conjunction. Its failure continuation, on the other hand, is the translation of the second part of the conjunction of clauses C_2. Indeed, if a solution is still requested after the resolution of the first conjunction, the second one must be tried. Equation 1.6 translates a clause with an empty body. The argument of the clause head, $[\![X]\!]$, is unified with the predicate argument ε. It outputs the success continuation if this unification succeeds, the failure continuation otherwise. Equation 1.7 translates a clause with a non empty body. The argument of the clause head, $[\![X]\!]$, is unified with the predicate argument ε. If the unification fails, the failure continuation ζ is executed. If this unification succeeds, the code to be executed is the translation of the body of the clause: $\lambda\zeta'(\mathcal{S}_g[\![P : G]\!]\ \kappa\ \xi\ \zeta')$. Equation 1.8 inductively translates a conjunction of goals. The first conjunction of goals takes as arguments the failure ζ continuation and the cut failure continuation ξ of the whole conjunction respectively as second and third argument. Its first argument (i.e., its success continuation) is the evaluation of the second conjunction of goals. Indeed, once the first conjunction has succeeded, the second one needs to be tried. The use of the λ-variable ζ' is necessary here because the value of the failure continuation is only known after the first conjunction of goals is exe-

$$unify \stackrel{1}{\equiv} \lambda t_1 t_2 \kappa \zeta \begin{cases} (\kappa\ \zeta) & \text{if } t_1 \text{ unifies with } t_2 \\ (\zeta) & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{query}[\![P]\!] \stackrel{2}{\equiv} \lambda p X (\mathcal{S}_g[\![P : p(X)]\!]\ \lambda\zeta(\texttt{yes.}\zeta)\ \ \texttt{no.nil no.nil})$$

$$\mathcal{S}[\![P]\!] \stackrel{3}{\equiv} \lambda p \begin{cases} (\mathcal{S}_{pred}[\![P : p]\!]) & \text{if } p \in \text{pred(P)} \\ (\lambda\kappa\xi\zeta(\zeta)) & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{pred}[\![P : p]\!] \stackrel{4}{\equiv} \lambda\varepsilon\kappa\zeta(\mathcal{S}_p[\![P : clauses(p, P)]\!]\ \varepsilon\ \kappa\ \zeta\ \zeta)$$

$$\begin{aligned}
\mathcal{S}_p[\![P : C_1.C_2]\!] &\stackrel{5}{\equiv} \lambda\varepsilon\kappa\xi\zeta(\mathcal{S}_p[\![P : C_1]\!]\ \varepsilon\ \kappa\ \xi\ (\mathcal{S}_p[\![P : C_2]\!]\ \varepsilon\ \kappa\ \xi\ \zeta)) \\
\mathcal{S}_p[\![P : p(X)]\!] &\stackrel{6}{\equiv} \lambda\varepsilon\kappa\xi\zeta(unify\ [\![X]\!]\ \varepsilon\ \kappa\ \zeta) \\
\mathcal{S}_p[\![P : p(X)\ \text{:-}\ G]\!] &\stackrel{7}{\equiv} \lambda\varepsilon\kappa\xi\zeta(unify\ [\![X]\!]\ \varepsilon\ \lambda\zeta'(\mathcal{S}_g[\![P : G]\!]\ \kappa\ \xi\ \zeta')\ \zeta)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}_g[\![P : G_1, G_2]\!] &\stackrel{8}{\equiv} \lambda\kappa\xi\zeta(\mathcal{S}_g[\![P : G_1]\!]\ \lambda\zeta'(\mathcal{S}_g[\![P : G_2]\!]\ \kappa\ \xi\ \zeta')\ \xi\ \zeta) \\
\mathcal{S}_g[\![P : q(X)]\!] &\stackrel{9}{\equiv} \lambda\kappa\xi\zeta(\mathcal{S}[\![P]\!]\ q\ [\![X]\!]\ \kappa\ \zeta) \\
\mathcal{S}_g[\![P : !]\!] &\stackrel{10}{\equiv} \lambda\kappa\xi\zeta(\kappa\ \xi)
\end{aligned}$$

Fig. 1. An Operational Continuation Semantics for Prolog

cuted. Equation 1.9 translates an atomic goal. The argument of the clause head, $[\![X]\!]$, the success and failure continuations, are applied to the code of predicate q produced by Equation 1.3. Note that ξ is not used here, since the scope of a cut is local to a predicate. Equation 1.10 translates the cut goal. When a cut is encountered, the failure continuation is replaced by the cut failure continuation; this is done by applying the cut success continuation to the failure one.

Contrary to usage, we write λ-terms in their η-expanded form, for example, $\lambda\zeta'(\mathcal{S}_g[\![P : G]\!]\ \kappa\ \xi\ \zeta')$ instead of $(\mathcal{S}_g[\![P : G]\!]\ \kappa\ \xi)$. The reasons are multiple: using η-reduced form worries beginners who could believe that an argument is missing; the feasibility of η-reduction depends too much on a smart choice of parameter ordering; the different variants of the operational semantics that we propose are not equally η-reducible; and finally, adopting an η-expanded form makes the presentation more homogeneous.

2.2 Examples

Using this formalism, we can define different predicates such as `true/0`, `false/0` or `=/2` which unifies (a list of) two elements.

$$\begin{aligned}
&\mathcal{S}_{pred}[\![P : true]\!] \equiv \lambda\varepsilon\kappa\zeta(\kappa\ \zeta) \\
&\mathcal{S}_{pred}[\![P : false]\!] \equiv \lambda\varepsilon\kappa\zeta(\zeta) \\
&\mathcal{S}_{pred}[\![P : \text{`='}]\!] \equiv \lambda\varepsilon\kappa\zeta(unify\ [\![\text{left } \varepsilon]\!]\ [\![\text{right } \varepsilon]\!]\ \kappa\ \zeta)
\end{aligned}$$

Where 'left [l, r] ≡ l' and 'right [l, r] ≡ r'. These definitions directly result from the definition of *unify*. The predicate `p/1` defined by the clause "`p(f(X,Y)) :-`

X = Y.", which we need to represent as "p(f([X,Y])) :- =([X, Y])." because our semantics only deals with predicates of arity 1, is translated as follows:

$$\mathcal{S}_{pred}[\![P : p]\!] \equiv \lambda\varepsilon\kappa\zeta(unify\ [\![f([X,Y])]\!]\ \varepsilon\ (unify\ [\![X]\!]\ [\![Y]\!]\ \kappa\ \zeta)\ \zeta)$$

3 A formal specification of Byrd's box model

The idea of Byrd's box model is to model the control flow of Prolog programs by arrows between boxes. The boxes are the program procedures, where a procedure is a set of clauses defining the same predicate. The arrows relate the different boxes through ports. The `call` port corresponds to the initial invocation of the procedure. The `exit` port corresponds to a successful return of the procedure. The `redo` port indicates that a subsequent goal has failed and that the system is backtracking to search for other solutions to the procedure. The `fail` port corresponds to an unsuccessful return of the procedure. *Events* are program points that correspond to those ports. In the remaining of this section, we give a formal specification of this model. Then, we show how it is possible to animate the specification by a direct translation into λProlog.

3.1 A formal specification of Byrd's box model

$$unify \stackrel{1}{\equiv} \lambda t_1 t_2 \kappa\zeta \begin{cases} (\kappa\ \zeta) & \text{if } t_1 \text{ unifies with } t_2 \\ (\zeta) & \text{otherwise} \end{cases}$$

$$\mathcal{B}yrd_{query}[\![P]\!] \stackrel{2}{\equiv} \lambda pX(\mathcal{B}yrd_g^O[\![P : p(X)]\!]\ \lambda\zeta(\texttt{yes.}\zeta)\ \texttt{no.nil}\ \texttt{no.nil})$$

$$\mathcal{B}yrd^O[\![P]\!] \stackrel{3}{\equiv} \lambda p \begin{cases} (\mathcal{B}yrd_{pred}^O[\![P : p]\!]) & \text{if p} \in \text{pred(P)} \\ (\lambda\kappa\xi\zeta(\zeta)) & \text{otherwise} \end{cases}$$

$$\mathcal{B}yrd_{pred}^O[\![P : p]\!] \stackrel{4}{\equiv} \lambda\varepsilon\kappa\zeta(\mathcal{B}yrd_p^O[\![P : clauses(p,P)]\!]\ \varepsilon\ \kappa\ \zeta\ \zeta)$$

$$\begin{aligned}
\mathcal{B}yrd_p^O[\![P : C_1.C_2]\!] &\stackrel{5}{\equiv} \lambda\varepsilon\kappa\xi\zeta(\mathcal{B}yrd_p^O[\![P : C_1]\!]\ \varepsilon\ \kappa\ \xi\ (\mathcal{B}yrd_p^O[\![P : C_2]\!]\ \varepsilon\ \kappa\ \xi\ \zeta)) \\
\mathcal{B}yrd_p^O[\![P : p(X)]\!] &\stackrel{6}{\equiv} \lambda\varepsilon\kappa\xi\zeta(unify\ [\![X]\!]\ \varepsilon\ \kappa\ \zeta) \\
\mathcal{B}yrd_p^O[\![P : p(X)\ \text{:-}\ G]\!] &\stackrel{7}{\equiv} \lambda\varepsilon\kappa\xi\zeta(unify\ [\![X]\!]\ \varepsilon\ \lambda\zeta'(\mathcal{B}yrd_g^O[\![P : G]\!]\ \kappa\ \xi\ \zeta')\ \zeta)
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}yrd_g^O[\![P : G_1, G_2]\!] &\stackrel{8}{\equiv} \lambda\kappa\xi\zeta(\mathcal{B}yrd_g^O[\![P : G_1]\!]\ \lambda\zeta'(\mathcal{B}yrd_g^O[\![P : G_2]\!]\ \kappa\ \xi\ \zeta')\ \xi\ \zeta) \\
\mathcal{B}yrd_g^O[\![P : q(X)]\!] &\stackrel{9}{\equiv} \lambda\kappa\xi\zeta(\texttt{<call,q,X>}.(\mathcal{B}yrd^O[\![P]\!]\ q\ [\![X]\!] \\
&\qquad \lambda\zeta'(\texttt{<exit,q,X>}.(\kappa\ \texttt{<redo,q,X>}.\zeta'))\ \texttt{<fail,q,X>}.\zeta)) \\
\mathcal{B}yrd_g^O[\![P : !]\!] &\stackrel{10}{\equiv} \lambda\kappa\xi\zeta(\texttt{<call,!>}.\texttt{<exit,!>}.(\kappa\ \texttt{<redo,!>}.\texttt{<fail,!>}.\xi))
\end{aligned}$$

Fig. 2. A continuation semantics that specifies the Byrd trace (Equations 9 and 10 have been instrumented)

The equations presented in the previous section specify the operational semantics for any query. In particular, they specify, in terms of an abstract machine language, how the control flows from one predicate to another. In order to specify Byrd's box model, we extend this semantics in such a way that it does not only compute answers to queries, but also the Byrd's trace corresponding to their resolutions. We extend Figure 1 into Figure 2 and instrument Equations 1.9 and 1.10 with execution *events*. Events are triple of event attributes represented with the term constructor `<_,_,_>`; the first element of the triple is the port, the second one is the predicate name, and the third one is the list of the predicate arguments. Once the *unify* instructions of the abstract machine produced by the semantics have been fully evaluated (i.e., β-reduced), there only remains a sequence of events and query answers (`yes` and `no`).

Equations 1.9 and 1.10 translate atomic goals. They are the only equations that are changed. We insert a `call` event before the code that implements its resolution, i.e., before each goal invocation. If the goal succeeds, its success continuation will be executed; therefore we insert an `exit` event just before it. If another solution for that goal is requested on backtracking, the execution will search for another solution in the failure continuation and thus we insert a `redo` event just before it. If the goal fails, its failure continuation will be executed; hence we insert a `fail` event just before it. Equation 1.10 specifies how to translate the cut. Since the way the cut should be traced is not described in Byrd's original article [9], we handle it the same way as any other atomic goal, and we instrument Equation 1.10 in the same manner as Equation 1.9.

The result of a call to $\mathcal{B}yrd_{query}[\![P]\!]$ is a list of events, where an event is either an atom `yes` or `no`, or a tuple. Here, the tuples are triples containing a port, a predicate name, and a predicate argument, but events can (and will, in the next sections) have more attributes.

3.2 Skipped and traced predicates

When tracing an execution, there are predicates that do not need to be traced; there are even predicates that cannot be traced such as built-in and library predicates. We call them *skipped predicates*. To handle them, we only need to replace in Equation 2.9 the function call $\mathcal{B}yrd^{O}[\![P]\!]$ by $\mathcal{S}_p[\![P]\!]$; then, events related to the detail of the resolution of such predicates do not appear in the trace.

3.3 Animating the specification by translating it into λProlog

One of the advantages of the semantics of Figure 2 is that the language used to write it can be easily coded into any language that manipulates λ-terms. Hence, we can animate this formal specification and check that it corresponds to the informal specification of Byrd [9]. Furthermore, we can use it to validate its extensions, as we do in Section 4.

We translate the specification into λProlog [3, 24]. λProlog is able to manipulate λ-terms and is well suited for meta-programming, as demonstrated by Miller and Nadathur [22]. In the following, we show how this semantics directly

```
p :- q, r.          q :- s.          s.
r :- a, b.          q :- t.          a.
```

Fig. 3. A Prolog program

```
[(call p), (call q), (call s), (exit s), (exit q), (call r), (call a),
 (exit a), (call b), (fail b), (redo a), (fail a), (fail r), (redo q),
 (redo s), (fail s), (call t), (fail t), (fail q), (fail p), no]
```

Fig. 4. Trace produced by the program above and the query "?- p."

derives into a λProlog program which is a Prolog interpreter that produces the specified trace. The translation from Figure 1 and 2 to λProlog is purely syntactic. In order to transform functions into relations, we pass the result of each semantic function as an argument of its corresponding λProlog predicate. Equation 2.2 "$\mathcal{B}yrd_{query}[\![P]\!] \equiv \lambda pX(\mathcal{B}yrd_g^O[\![P]\!]\ p(X)\ \lambda\zeta(\mathtt{yes}.\zeta)\ \ \mathtt{no.nil}\ \ \mathtt{no.nil})$" is translated into:

```
byrd_query Program Predicate X Result :-
   semtrace_goal Program (app Predicate X) SemGoal,
   Result = (SemGoal (Z\ [yes|Z]) [no] [no]).
```

The semantic function $\mathcal{B}yrd_{query}[\![\ ]\!]$ is translated into the λProlog predicate `byrd_query/4`. The principal syntactic difference between Prolog and λProlog is in the way terms and predicates are represented; the Prolog equivalent of "`app Predicate X`" is "`app(Predicate, X)`", where `app` is term constructor, and `Predicate` as well as `X` are logical variables. The abstraction λx is translated by `x\`. Predicate `byrd_query/4` takes a program name, a predicate name, and a term (the Prolog predicate argument) respectively as first, second and third argument. It outputs the result of the function in its fourth argument; this result is obtained by applying a success continuation `(Z\ [yes|Z])`, a cut failure `[no]`, and a failure continuation `[no]` to the result of `semtrace_goal/4`, the λProlog version of $\mathcal{B}yrd_g^O[\![\ ]\!]$. Equation 1.9 "$\mathcal{S}_g[\![P : q(X)]\!] \equiv \lambda\kappa\xi\zeta(\mathcal{S}[\![P]\!]\ \mathrm{q}\ [\![X]\!]\ \kappa\ \zeta)$", which translate an atomic goal different from cut, is translated as follows:

```
semprolog_goal Program (app Q X) Result :-
    \+(Q = cut),
    semprolog_term X SemX,
    semprolog Program Q SemQ,
    Result = K\XI\Z\ (SemQ SemX K Z).
```

The predicates `semprolog/3`, `semprolog_goal/3`, and `semprolog_term/2` implements respectively the functions $\lambda x.(\mathcal{S}[\![x]\!])$, $\lambda x.(\mathcal{S}_g[\![x]\!])$, and $\lambda x.([\![x]\!])$. The term `(app Q X)` is a representation of the Prolog goal `Q(X)`. The predicate `cut/0` is the cut of Prolog. In order to instrument the semantics more easily, we make use of the macro facilities of Prolog/MALI [6] to define various combinators:

```
#define CALL (x\y\  [(port_call x)|y])
```

This "`#define`" directive tells the Prolog/MALI pre-processor to replace in the program all the occurrences of "`CALL`" by the λ-term $\lambda xy([(\texttt{port\_call}\ x)|y])$. In the same way, we define combinators that ease the instrumentation of `exit`, `fail`, and `redo` ports. Then, we translate Equation 2.9 as follows:

```
semtrace_goal
    Program (app Q X) Result :-
    \+(Q = cut), semtrace_term X SemX,
    semtrace Program Q SemQ,
    Result = K\XI\Z\ (CALL Q (SemQ SemX (Z2\ (EXIT Q (K (REDO Q Z2))))
        (FAIL Q Z))).
```

Figure 3 shows a Prolog program taken from [9]. This program and the query '$\mathcal{B}yrd_{query}[\![P]\!]\ p$', once evaluated with the interpreter that results from the λProlog translation of Figure 2, produce the trace of Figure 4. We can observe that it is the same trace as in the original article of Byrd [9].

4 Extending Byrd's box model

In this section, we describe a few possible extensions of Byrd's box model. They are useful to understand the control and data flow of Prolog program executions, and they show the flexibility of the specification technique. For didactic purposes, each extension is described as an extension of the basic model of Figure 2. The extensions are then all combined in Figure 5.

4.1 Special treatment for deterministic predicates

Byrd's original article is not explicit about the way the cut predicate should be traced. In Equation 2.10, we conservatively chose to treat the cut as any other predicate. However, since the cut always succeeds exactly once, the event sequence "`<redo,!>.<fail,!>`" is redundant; we can therefore remove it from the trace by replacing Equation 2.10 by:

$$\mathcal{B}yrd_g^O[\![P :!]\!] \stackrel{10}{\equiv} \lambda\kappa\xi\zeta(\texttt{<call,!>.<exit,!>}.(\kappa\ \xi))$$

Hence, the trace produced by the evaluation of a cut only consists of the event sequence: "`<call,!>.<exit,!>`". We could do the same for all deterministic predicates, i.e., predicates that have exactly one solution.

4.2 A `unify` port

The `unify` port corresponds to events generated when the execution finds a clause whose head unifies with the current goal. This port gives the result of the unification of the current goal with the matching clause head. This port is very useful to understand the control and data flow of Prolog executions. For example, when the execution backtracks from a very deep branch of the SLD-tree, what one sees is a long sequence of `redo` events. A `unify` event means that a choice point leads to a successful unification between the current goal and a clause head. In order to generate `unify` events, we insert a triple before the success continuation of Equations 2.6 and 2.7 which explicitly handle unification:

$$\mathcal{B}yrd^O_p[\![P : p(X)]\!] \stackrel{6}{\equiv} \lambda\varepsilon\kappa\xi\zeta(\mathit{unify}\ [\![X]\!]\ \varepsilon\ \lambda\zeta'(\texttt{<unify,p,X>}.(\kappa\ \zeta'))\ \zeta)$$
$$\mathcal{B}yrd^O_p[\![P : p(X)\text{:-}G]\!] \stackrel{7}{\equiv} \lambda\varepsilon\kappa\xi\zeta(\mathit{unify}\ [\![X]\!]\ \varepsilon\ \lambda\zeta'(\texttt{<unify,p,X>}.(\mathcal{B}yrd^O_g[\![P : G]\!]\ \kappa\ \xi\ \zeta')\ \zeta))$$

4.3 Clause number in `unify` events

We call *clause number of a predicate p* the rank of a clause in the order of appearance in the definition of p. This information is useful at `unify` ports because it tells which clause successfully unifies with the current goal. This is particularly important to combine static and dynamic analyses.

To implement the clause number, we add a counter ι that we pass as argument of the failure continuation of $\mathcal{B}yrd^O_p[\![\ ]\!]$ in Equations 2.5, 2.6 and 2.7. We pass it as argument of the failure continuation because it is through this continuation that the clause number is incremented. Let us write "`eventT`" the type of events. The type of failure continuation, which was "`(list eventT)`", now becomes "`int` $\rightarrow$ `(list eventT)`". The counter ι is initialized to 1 when $\mathcal{B}yrd^O_p[\![\ ]\!]$ is called, i.e., in Equation 2.4. We need to increment this counter every time another clause is tried; therefore, we increment it in the failure continuations of Equations 2.6 and 2.7. The new Equations 2.4, 2.5, 2.6 and 2.7 below. We have underlined what has changed in these equations; the other ones have not changed.

$$\mathcal{B}yrd^O_{pred}[\![P : p]\!] \stackrel{4}{\equiv}\ \lambda\varepsilon\kappa\zeta(\mathcal{B}yrd^O_p[\![P : \mathit{clauses}(p,P)]\!]\ \varepsilon\ \kappa\ \zeta\ \underline{(\lambda\iota}\ \zeta\underline{)\ 1})$$
$$\mathcal{B}yrd^O_p[\![P : C_1.C_2]\!] \stackrel{5}{\equiv}\ \lambda\varepsilon\kappa\xi\zeta\underline{\iota}(\mathcal{B}yrd^O_p[\![P : C_1]\!]\ \varepsilon\ \kappa\ \xi\ \underline{(\lambda\iota'}.(\mathcal{B}yrd^O_p[\![P : C_2]\!]\ \varepsilon\ \kappa\ \xi\ \zeta\ \underline{\iota'))}\ \underline{\iota})$$
$$\mathcal{B}yrd^O_p[\![P : p(X)]\!] \stackrel{6}{\equiv}\ \lambda\varepsilon\kappa\xi\zeta\underline{\iota}(\mathit{unify}\ [\![X]\!]\ \varepsilon\ \lambda\zeta'(\texttt{<unify,p,X,}\underline{\iota}\texttt{>}.(\kappa\ \zeta'))\ \underline{(}\zeta\ \underline{\iota+1)})$$
$$\mathcal{B}yrd^O_p[\![P : p(X)\text{:-}G]\!] \stackrel{7}{\equiv}\ \lambda\varepsilon\kappa\xi\zeta\underline{\iota}(\mathit{unify}\ [\![X]\!]\ \varepsilon$$
$$\lambda\zeta'(\texttt{<unify,p,X,}\underline{\iota}\texttt{>}.(\mathcal{B}yrd^O_g[\![P : G]\!]\ \kappa\ \xi\ \zeta'))\ \ \underline{(}\zeta\ \underline{\iota+1)})$$

For `unify` events, the clause number is added to the tuple of event attributes. Note that here, we add a counter as argument of failure continuations. Adding counters in continuations might be quite delicate; sometimes the counters we manipulate are instantiated, sometimes their values will only be known in the future. The general method to deal with unknown values is to create abstractions $(\lambda\iota'...\iota'...)$ that will be instantiated when available by applying them to some value. For example, in Equation 2.5, the value of the clause number ι cannot be known while splitting a conjunction of clauses because nothing says where the partition is made. Hence, we create an abstraction $(\lambda\iota')$ that will be instantiated in Equations 2.6 and 2.7 and which will be equal to the number of clauses in C_1.

4.4 Execution depth

The *execution depth* of a goal is its number of ancestors plus 1. In order to compute it within our semantics, we add a counter, δ, as argument of all the failure continuations. As for the previous extension, the type of failure continuations becomes "`failureT` $\equiv$ `int` $\rightarrow$ `(list eventT)`"[2]. As a matter of fact, the equations of Figure 2 are almost unchanged; the counter δ is simply added as argument of Equations 2.3 to 2.10. For example, Equation 2.5 becomes:

$$\mathcal{B}yrd_p^O[\![P : C_1.C_2]\!] \stackrel{5}{\equiv} \lambda\varepsilon\kappa\xi\zeta\underline{\delta}.(\mathcal{B}yrd_p^O[\![P : C_1]\!]\ \varepsilon\ \kappa\ \xi\ (\mathcal{B}yrd_p^O[\![P : C_2]\!]\ \varepsilon\ \kappa\ \xi\ \zeta\ \underline{\delta})\ \underline{\delta})$$

The depth counter is initialized in the top level function, namely, the function defined by Equation 2.2:

$$\mathcal{B}yrd_{query}[\![P]\!] \stackrel{2}{\equiv} \lambda pX(\mathcal{B}yrd_g^O[\![P]\!]\ p(X)\ \lambda\zeta(\texttt{yes}.\zeta)\ \ \texttt{no.nil}\ \texttt{no.nil}\ \underline{1})$$

Then, it needs to be incremented each time a call is made; we can do that either in Equation 2.3, 2.4 or 2.9. For example, let us do it in Equation 2.9:

$$\begin{array}{l}\mathcal{B}yrd_g^O[\![P : q(X)]\!] \stackrel{9}{\equiv} \lambda\kappa\xi\zeta\underline{\delta}.(\texttt{<call,q,X,}\delta\texttt{>}(\mathcal{B}yrd^O[\![P]\!]\ q\ [\![X]\!] \\ \quad \lambda\zeta'(\texttt{<exit,q,X,}\delta\texttt{>}.(\kappa\ \ \texttt{<redo,q,X,}\delta\texttt{>}.\zeta'))\ \ \texttt{<fail,q,X,}\delta\texttt{>}.\zeta\ \underline{(\delta+1)}))\end{array}$$

We also need to modify the definition of `unify` by applying the failure continuation to the depth as follows:

$$unify \stackrel{1}{\equiv} \lambda t_1 t_2 \kappa\zeta\delta. \begin{cases} (\kappa\ \zeta\ \underline{\delta}) & \text{if } t_1 \text{ unifies with } t_2 \\ \underline{(}\zeta\ \underline{\delta)} & \text{otherwise} \end{cases}$$

4.5 Goal invocation number

We call *goal invocation number* or *call number* a counter that is incremented every time a goal is invoked. To implement it, we do the same sort of transformation as in the previous section to compute the execution depth; we add a counter (ν) as argument of Equations 2.3 to 2.10. This counter is initialized in Equation 2.2 and incremented in Equations 2.9 and 2.10, where each atomic goal translation is made. The only difference with the depth is in Equations 2.5 and 2.8 because the value of the call number of the second conjunction of clauses and of goals respectively is not known at that point. Therefore, as in Section 4.3, we create an abstraction $\lambda\nu'$; the value of the call number will be applied to this abstraction when it is known (in the *unify*). Hence, Equation 2.5 becomes:

$$\mathcal{B}yrd_p^O[\![P : C_1.C_2]\!] \stackrel{5}{\equiv} \lambda\varepsilon\kappa\xi\zeta\underline{\nu}(\mathcal{B}yrd_p^O[\![P : C_1]\!]\ \varepsilon\ \kappa\ \xi\ \underline{\lambda\nu'.}(\mathcal{B}yrd_p^O[\![P : C_2]\!]\ \varepsilon\ \kappa\ \xi\ \zeta\ \underline{\nu')}\ \underline{\nu})$$

Equation 2.8 is transformed in the same way.

[2] The type of success continuations remains "`successT` $\equiv$ `failureT` $\rightarrow$ `failureT`"

4.6 Chronological event number

We call *chronological event number* the rank of an event in the trace. It is different from the call number; the chronological event number is incremented for each event, and a goal generates several events. To implement the chronological event number, again we add an integer (γ) as argument of the failure continuation and we increment it whenever an event is generated. The general method to modify the value of an argument of a λ-term is to explicit the argument by creating an abstraction and then to apply the modified argument to the abstraction. For example, $\lambda\gamma(\zeta\ \gamma+1)$ increments the first argument of failure continuation ζ of type "`int` $\rightarrow$ `(list eventT)`". Thus, to increment γ each time an event is generated, we only need to transform an event such as $\texttt{<fail,}p,X,\gamma\texttt{>.}\zeta$ into $\lambda\gamma'(\texttt{<fail,}p,X,\gamma'\texttt{>.}(\zeta\ \gamma'+1))$. Equation 2.9 thus becomes:

$$\mathcal{B}yrd_g^O[\![P:q(X)]\!] \stackrel{9}{\equiv} \lambda\kappa\xi\zeta\underline{\gamma}(\texttt{CALL}\ \mathrm{q}\ \mathrm{X}\ \underline{\gamma}\ (\mathcal{B}yrd^O[\![P]\!]\ q\ [\![X]\!]$$
$$\lambda\zeta'(\texttt{EXIT}\ \mathrm{q}\ \mathrm{X}\ \underline{\gamma}\ (\kappa\ (\texttt{REDO}\ \mathrm{q}\ \mathrm{X}\ \underline{\gamma}\ \zeta')))\ \ (\texttt{FAIL}\ \mathrm{q}\ \mathrm{X}\ \underline{\gamma}\ \zeta)\ \underline{\gamma+1}))$$

$$\texttt{CALL} \equiv \lambda pX\gamma\zeta(\texttt{<call,}p,X,\gamma'\texttt{>.}\zeta)$$
$$\texttt{PORT} \equiv \lambda pX\gamma\zeta\gamma'(\texttt{<port,}p,X,\gamma'\texttt{>.}(\zeta\ \gamma'+1))$$

where `PORT` is in {`EXIT`, `FAIL`, `REDO`}, and where `port` is in {`exit`, `fail`, `redo`}.

The combinators `EXIT`, `FAIL` and `REDO` have been introduced to instrument the semantics in order to simplify the presentation. Combinator `CALL` has been added for the sake of homogeneousness.

4.7 Merging all the extensions

To merge all the previously described extensions, each extension has to be applied to the previously extended semantics, instead of extending each time the basic model. The result of the merging of all the extensions is given in Figure 5. integer is the chronological event number, the second one is the call number, and the last one is the execution depth (ν: γ `[`δ`] port predicate`). The only difficulty is to get the call number at `unify` ports. Indeed, ν that is passed down in argument of $\mathcal{B}yrd_p^O[\![\ ]\!]$ in Equation 5.9 will change before we want to use it to generate the `unify` event in Equations 5.6 and 5.7. In order to solve this problem, we make a copy, μ, of ν, and add it as an argument of $\mathcal{B}yrd_p^O[\![\ ]\!]$ (see Equations 5.5, 5.6, 5.7). This new counter is initialized with the current value of ν in Equation 5.9.

Adding a new counter is delicate because constructing Equations 3 to 10 is like programming in a language that is not a programming language. However, there exists a general method: (1) locate equations where "something happens" (e.g., Equation 9 for a depth-counter); (2) modify the equation for handling what happens (e.g., the depth-counter is incremented); this changes the type of what is defined; (3) locate all equations that use the modified definition, consider them as equations where "something happens", and go back to (2).

Figure 6 gives an example of what this specification outputs; it displays on two columns the output of this specification with the program of Figure 3

$$unify \overset{1}{\equiv} \lambda t_1 t_2 \kappa\zeta\nu\delta\gamma \begin{cases} (\kappa\ \zeta\ \nu\ \delta\ \gamma) & \text{if } t_1 \text{ unifies with } t_2 \\ (\zeta\ \nu\ \delta\ \gamma) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathcal{B}yrd_{query}[\![P]\!] &\overset{2}{\equiv} \lambda pX(\mathcal{B}yrd_g^O[\![P:p(X)]\!]\ \lambda\zeta\nu\delta\gamma(\texttt{yes.}(\zeta\ \nu\ \delta\ \gamma)) \\
&\qquad \lambda\nu\delta\gamma(\texttt{no.nil})\ \lambda\nu\delta\gamma(\texttt{no.nil})\ 1\ 1\ 1)
\end{aligned}$$

$$\mathcal{B}yrd^O[\![P]\!] \overset{3}{\equiv} \lambda p \begin{cases} (\mathcal{B}yrd^O_{pred}[\![P:p]\!]) & \text{if p} \in \text{pred(P)} \\ (\lambda\varepsilon\kappa\zeta\nu\mu\delta\gamma(\zeta\ \nu\ \delta\ \gamma)) & \text{otherwise} \end{cases}$$

$$\mathcal{B}yrd^O_{pred}[\![P:p]\!] \overset{4}{\equiv} \lambda\varepsilon\kappa\zeta\nu\mu\delta\gamma(\mathcal{B}yrd_p^O[\![P:clauses(p,P)]\!]\ \varepsilon\ \kappa\ \zeta\ (\lambda\iota\ \zeta)\ 1\ \nu\ \mu\ \delta\ \gamma)$$

$$\begin{aligned}
\mathcal{B}yrd_p^O[\![P:C_1.C_2]\!] &\overset{5}{\equiv} \lambda\varepsilon\kappa\xi\zeta\iota\nu\mu\delta\gamma(\mathcal{B}yrd_p^O[\![P:C_1]\!]\ \varepsilon\ \kappa\ \xi \\
&\qquad \lambda\iota'\nu'\delta'\gamma'(\mathcal{B}yrd_p^O[\![P:C_2]\!]\ \varepsilon\ \kappa\ \xi\ \zeta\ \iota'\ \nu'\ \mu\ \delta\ \gamma') \\
&\qquad \iota\ \nu\ \mu\ \delta\ \gamma) \\
\mathcal{B}yrd_p^O[\![P:p(X)]\!] &\overset{6}{\equiv} \lambda\varepsilon\kappa\xi\zeta\iota\nu\mu\delta\gamma(unify\ [\![X]\!]\ \varepsilon \\
&\qquad \lambda\zeta'(\texttt{UNIFY}\ \text{p X}\ \mu\ \delta\ \gamma\ \iota\ (\kappa\ \zeta'))) \\
&\qquad (\zeta\ \iota+1)\ \nu\ \delta\ \gamma) \\
\mathcal{B}yrd_p^O[\![P:p(X)\ \text{:-}\ G]\!] &\overset{7}{\equiv} \lambda\varepsilon\kappa\xi\zeta\iota\nu\mu\delta\gamma(unify\ [\![X]\!]\ \varepsilon\ \lambda\zeta'\nu'\delta'\gamma'(\texttt{UNIFY}\ \text{p X}\ \mu\ \delta\ \gamma\ \iota \\
&\qquad (\mathcal{B}yrd_g^O[\![P:G]\!]\ \kappa\ \xi\ \zeta')\ \nu'\ \delta'\ \gamma')\ \ (\zeta\ \iota+1)\ \nu\ \delta\ \gamma)
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}yrd_g^O[\![P:G_1,G_2]\!] &\overset{8}{\equiv} \lambda\kappa\xi\zeta\nu\delta\gamma(\mathcal{B}yrd_g^O[\![P:G_1]\!] \\
&\qquad \lambda\zeta'\nu'\delta'\gamma'(\mathcal{B}yrd_g^O[\![P:G_2]\!]\ \kappa\ \xi\ \zeta'\ \nu'\ \delta\ \gamma') \\
&\qquad \xi\ \zeta\ \nu\ \delta\ \gamma) \\
\mathcal{B}yrd_g^O[\![P:q(X)]\!] &\overset{9}{\equiv} \lambda\kappa\xi\zeta\nu\delta\gamma(\texttt{CALL}\ \text{q X}\ \gamma\ (\mathcal{B}yrd^O[\![P]\!]\ q\ [\![X]\!] \\
&\qquad \lambda\zeta'(\texttt{EXIT}\ \text{q X}\ \nu\ \delta\ \gamma\ (\kappa\ (\texttt{REDO}\ \text{q X}\ \nu\ \delta\ \gamma\ \zeta'))) \\
&\qquad (\texttt{FAIL}\ \text{q X}\ \nu\ \delta\ \gamma\ \zeta)\ (\nu+1)\ \nu\ (\delta+1)\ (\gamma+1))) \\
\mathcal{B}yrd_g^O[\![P:!]\!] &\overset{10}{\equiv} \lambda\kappa\xi\zeta\nu\delta\gamma(\texttt{CALL}\ !\ \nu\ \delta\ \gamma\ ((\texttt{EXIT}\ !\ \nu\ \delta\ \gamma\ (\kappa\ \xi))\ (\nu+1)\ \delta\ (\gamma+1)))
\end{aligned}$$

$\texttt{CALL} \equiv\ \lambda pX\nu\delta\gamma\zeta(\texttt{<call}, p, X, \nu, \delta, \gamma'\texttt{>.}\zeta)$
$\texttt{UNIFY} \equiv\ \lambda pX\nu\delta\gamma\iota\zeta\nu'\delta'\gamma'(\texttt{<unify}, p, X, \nu, \delta-1, \gamma', \iota\texttt{>.}(\zeta\ \nu'\ \delta'\ (\gamma'+1)))$
$\texttt{PORT} \equiv\ \lambda pX\nu\delta\gamma\zeta\nu'\delta'\gamma'(\texttt{<port}, p, X, \nu, \delta, \gamma'\texttt{>.}(\zeta\ \nu'\ \delta'\ (\gamma'+1)))$,
where $\texttt{PORT} = \texttt{EXIT}, \texttt{FAIL}, \texttt{REDO}$, $\texttt{port} = \texttt{exit}, \texttt{fail}, \texttt{redo}$.

Fig. 5. A continuation semantics that specifies an extended Byrd trace. κ = success continuation, $\{\zeta,\ \xi\}$ = failure continuations, ι = clause number, $\{\nu,\ \mu\}$ = goal invocation number, δ = execution depth, γ = chronological event number.

```
1:  1 [1] call  p                       14: 6 [3] call  b
2:  1 [1] unify p (clause 1)            15: 6 [3] fail  b
3:  2 [2] call  q                       16: 5 [3] redo  a
4:  2 [2] unify q (clause 1)            17: 5 [3] fail  a
5:  3 [3] call  s                       18: 4 [2] fail  r
6:  3 [3] unify s (clause 1)            19: 2 [2] redo  q
7:  3 [3] exit  s                       20: 3 [3] redo  s
8:  2 [2] exit  q                       21: 3 [3] fail  s
9:  4 [2] call  r                       22: 2 [2] unify q (clause 2)
10: 4 [2] unify r (clause 1)            23: 7 [3] call  t
11: 5 [3] call  a                       24: 7 [3] fail  t
12: 5 [3] unify a (clause 1)            25: 2 [2] fail  q
13: 5 [3] exit  a                       26: 1 [1] fail  p
                                          no (more) solution.
```

Fig. 6. Output of the specification of Figure 5 with the program of Figure 3 with the query "?- p.".

with the query "?- p.". The λProlog is very important to check the formal specification.

5 Related work

5.1 Using natural semantics and structured operational semantics

Structured operational semantics (SOS) [27] describe with a transition system the way expressions are evaluated. Bernstein and Stark [4] propose to base the definition of debuggers on such semantics. They define a simple functional programming language with a SOS. The transition rules are extended to specify the behavior of the debugger. These new rules are in fact copies of the semantics rules extended with a debugging context. This debugging context contain the information that let the user see or not the detail of the evaluation for every sub-expression of an expression.

Berry [5] and Da Silva [30] have a similar approach using a natural semantics [18]. Natural semantics are similar to SOS, except that they abstract away from some operational details that SOS describe. With natural semantics, each program statement is described in one step, whereas it can be defined by several step with a SOS.

It is difficult to say, in the general case, which one between SOS, natural, or denotational semantics, is better suited to formally define dynamic analysis tools. For a given dynamic analysis tool, one just need to make sure that the level of granularity of the semantics is sufficiently fine. However, compositional denotational semantics are maybe more easy to use to make proofs.

5.2 Other systems based on denotational semantics

Kishon and al. [19, 20] use a denotational and operational continuation semantics to formally define monitors for functional programming languages. The kind of monitors they define are profilers, debuggers, and statistic collectors. From the operational semantics, a formal description of the monitor, and a program, they derive an instrumented executable file that performs the specified monitoring activity. The semantics of the original programs is preserved. Then, they use partial evaluation to make their monitors reasonably efficient. Their approach is very similar to the one of Consel and Khoo [10], that use the same kind of semantics together with partial evaluation to derive Prolog compilers.

Although this use of operational semantics is comparable to what we do, the objectives are not the same. We simply propose a way to formally specify trace models and to experiment these specifications. Then, the next step is to prove that existing implementations of trace systems are correct with respect to a model specified in this kind of settings. We plan to do that with the Prolog tracer implementation described in [14]. Once the tracer is validated, we can safely base dynamic analyses on it, as we have done for debugging [13] or monitoring [16]. Kishon and al., on the other hand, aim at directly deriving the monitoring tools from the semantics. One advantage of our approach is that it can be used on top on existing systems whereas Kishon and al. rebuild the whole monitoring tools from scratch.

5.3 Deriving programming environments

PSG [2] is a tool that aims at automating the development language-dependent programming environments. Bahlke and al. [1] show how to derive debuggers with PSG. This automation is based on a denotational semantics that describes the way programs are executed. PSG also uses a grammar for the language that allows PSG to manipulate the language expressions as syntax trees. The nodes of the syntax tree are the potential event sites of the debugger. The denotational semantics is used to navigate in the tree to simulate the program execution.

This system produce very nice syntactic and semantic based source code highlighting. However, as far as dynamic analysis is concerned, we believe that having two distinct systems to compile and to debug is not desirable. As noted by Brooks and al. [8], firstly, some errors only occur in presence of optimisations, and vice versa; some programs can only be executed in their optimized form because of time and memory constraints; when searching for "hot spots", it is better to do it with the optimized program as lots of things can be optimized away; and finally, sometimes, the error comes from the optimisations themselves.

6 Conclusion

In this article, we have given a formal specification of Byrd's box model based on an existing denotational and operational semantics for Prolog with cut. We have shown how this specification can be extended to describe richer trace models. We have also shown how these specifications can be executed by a direct translation into λProlog, leading to a Prolog interpreter that generates execution traces. This interpreter can be used both to experiment with various trace models and to validate the different event specifications. Hence we have a formal framework to specify and prototype trace models. We also have the framework to validate tracers implementation with respect to a trace model specification.

References

1. R. Bahlke, B. Moritz, and G. Snelting. A generator for language-specific debugging systems. *ACM SIGPLAN Notices*, 22(7):92–101, July 1987.
2. R. Bahlke and G. Snelting. The PSG system: From formal language definition to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.
3. C. Belleannée, P. Brisset, and O. Ridoux. A pragmatic reconstruction of λProlog. *Journal of Logic Programming*, 41(1), 1999.
4. K.L. Bernstein and E.W. Stark. Operational semantics of a focusing debugger. In ENCS, editor, *Proc. of the 11th Conf. on the Mathematical Foundations of Programming Semantics*, 1995.
5. D. Berry. *Generating program animators from programming language semantics.* PhD thesis, University of Edinburgh, 1991.
6. P. Brisset and O. Ridoux. The compilation of λProlog and its execution with MALI. PI 687, IRISA, 1992.
7. P. Brisset and O. Ridoux. Continuations in λProlog. In David S. Warren, editor, *Proc. of the Tenth Int. Conf. of Logic Programming.* The MIT Press, 1993.
8. G. Brooks, G.J. Hansen, and S. Simmons. A new approach to debugging optimized code. In *SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 1–11, 1992.
9. L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tärnlund, editor, *Logic Programming Workshop*, 1980.
10. C. Consel and S.C. Khoo. Semantics-directed generation of a prolog compiler. In *Proc. of the 3rd Int. Symp. on Programming Language Implementation and Logic Programming*, volume 528 of *LNCS*. Springer-Verlag, 1991.
11. A. de Bruin and E.P. de Vink. Continuation semantics for Prolog with cut. In *TAPSOFT'89: Proc. of the Int. Joint Conf. on Theory and Practice of Software Development*, LNCS. Springer-Verlag, 1989.
12. S. K. Debray and P. Mishra. Denotational and operational semantics for Prolog. *Journal of Logic Programming*, 5, 1988.
13. M. Ducassé. Opium: An extendable trace analyser for Prolog. *Journal of Logic programming*, 39, 1999. Special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds).
14. M. Ducassé and J. Noyé. Tracing Prolog programs by source instrumentation is efficient enough. *Journal of Logic Programming*, 43(2), May 2000. Tech. Notes.
15. M. Eisenstadt. A powerful Prolog trace package. In *Proc. of the 6th European Conf. on Artificial Intelligence*. North-Holland, 1984.
16. E. Jahier and M. Ducassé. A generic approach to monitor program executions. In D. De Schreye, editor, *Proc. Int. Conf. on Logic Programming*, pages 139–153. MIT Press, 1999.
17. N. D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for Prolog. In *Proc. of the Int. Symp. on Logic Programming.* IEEE, Computer Society Press, 1984.

18. G. Kahn. Natural semantics. In *Proc. of the Symp. on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, Passau, Germany, February 1987. Springer-Verlag. The paper is also available as INRIA Report 601, February, 1987.
19. A. Kishon and P. Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4), 1995.
20. A. Kishon, P. Hudak, and C. Consel. Monitoring semantics: a formal framework for specifying, implementing and reasoning about execution monitors. *ACM Sigplan Notices*, 26(6), 1991.
21. A.B. Matos. A matrix model for the flow of control in Prolog programs with applications to profiling. *Software Practice and Experience*, 24(8), 1994.
22. D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *Proc. of the Fifth Int. Conf. and Symp. of Logic Programming*. IEEE, Computer Society Press, 1987.
23. S. Moroshita and M. Numao. Prolog computation model BPM and its debugger PROEDIT2. In *Proc. of the 5th Int. Conf. of Logic Programming*. Springer-Verlag, 1986.
24. G. Nadathur and D.A. Miller. An overview of λProlog. In R.A. Kowalski and K.A. Bowen, editors, *Proc. of the Joint Int. Conf. and Symp. of Logic Programming*. MIT Press, 1988.
25. T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.
26. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19 & 20, 1994.
27. G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.
28. M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *Proc. of the Symp. on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, 1991. Sigplan Notices 26:9.
29. A. Schleiermacher and J.F.H. Winkler. The implementation of ProTest, a Prolog Debugger for a refined box model. *Software Practice and Experience*, 20, 1990.
30. F.Q.B. Da Silva. *Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics*. PhD thesis, Univ. of Edinburgh, 1991.
31. G. Tobermann and C. Beckstein. What's in a trace: The box model revisited. In Peter Fritzon, editor, *AADEBUG*, volume 749 of *LNCS*. Springer-Verlag, 1993.