Science of Computer Programming 21 (1993) 263–291
Elsevier

Semantics-directed generation of a Prolog compiler [*]

Charles Consel and Siau Cheng Khoo [†]

Yale University, Department of Computer Science, New Haven, CT, USA

Communicated by D. Bjørner
Received April 1992
Revised June 1993

Abstract

Consel, C. and S.C. Khoo, Semantics-directed generation of a Prolog compiler, Science of Computer Programming 21 (1993) 263–291.

In a recent paper [27], the denotational semantics for the language Prolog was presented. The aim was to define precisely the language.

This paper goes further in this direction by describing how the denotational semantics of Prolog can be used to *interpret* and to *compile* Prolog programs, as well as to *automatically generate a compiler* for the language. Our approach is based on *partial evaluation.* Compilation is achieved by specializing the Prolog definition. The compiler is obtained by self-application of the partial evaluator. It is well-structured and the speed of the compiled code has been found to be about six times faster than interpretation.

Our approach improves on previous work [14,22,32] in that: (i) it enables compiler generation and consequently speeds up the compilation process, and (ii) it goes beyond the usual mapping from syntax to denotations by processing the static semantics of the language definition.

1. Introduction

The denotational semantics for the core of Prolog was recently presented in [27]. As in [13,19], the aim was to formalize Prolog. Beyond the the-

Correspondence to: C. Consel, Pacific Software Research Center, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 19600 N.W. Von Neumann Drive, Beaverton, OR 97006-1999, USA. E-mail: consel@cse.ogi.edu.

*This research was supported in part by NSF and DARPA grants CCR-8809919 and N00014-91-J-4043, respectively. The second author was also supported by a National University of Singapore Overseas Graduate Scholarship.

†Current address: Department of Information Systems and Computer Science, National University of Singapore, 10 Kent Ridge Crescent, Singapore 0511. E-mail: khoosc@iscs.nus.sg.

oretical interest, it has long been argued that denotational definitions can be used to derive interpreters or compilers (e.g., [18,25,26,29,32]). This approach is attractive because it closely relates the formal specification and its implementation. This paper describes how the denotational semantics of the core of Prolog can be used to *interpret* and to *compile* Prolog programs. Furthermore, it is shown how a *compiler* for the language can be *automatically generated.*

Our approach can be decomposed as follows. The semantic equations of Prolog are coded in a functional programming language: a side-effect-free dialect of Scheme [28]. The result can be viewed as an interpreter and consequently be executed by a Scheme processor.

Then, compilation is achieved, using partial evaluation [1], by specializing the interpreter with respect to a program. Because partial evaluation is semantic-preserving [21], the target code has the same behavior as the interpretation of the original program. Moreover, since a partial evaluator is a static semantic processor [29], compile-time actions are executed, and the result solely represents dynamic operations as shown in [22]. A source program and the corresponding compiled code are displayed in Appendix A.

Although the compiled code has been found to run about six times faster than the interpreted code, the compilation phase might be slow [22]. However, our experiment is based on Schism [6,7,10], a self-applicable partial evaluator for a side-effect-free dialect of Scheme. As such, Schism can generate compilers; this is done by specializing the partial evaluator with respect to an interpreter [21]. As a result, the compilation phase is about twelve times faster than specialization of the interpreter.

An important component of our system is a preliminary phase called *binding time analysis.* This phase automatically splits the definition of a language into two parts: the static semantics (the usual compile-time actions) and the dynamic semantics. This considerably facilitates the partial evaluation phase and is crucial for self-application [21].

Mix [20] was the first partial evaluator able to generate compilers as well as a compiler generator. It partially evaluates first-order recursive equations. Schism handles both higher-order functions and data structures [8,9]. As a result, it extends the class of applications that can be tackled by a self-applicable partial evaluator. In particular, continuation semantics, which is essential in defining the language Prolog, can be handled.

The paper is organized as follows. Section 2 introduces partial evaluation, presents Schism, and lists the related work. Section 3 discusses the language Prolog by first briefly presenting its denotational definition, and then by describing its representation in Schism. Section 4 investigates the partial evaluation aspects of the specification. Section 5 gives an assessment of the work. Finally, Section 6 proposes a method to improve the compiled programs.

2. Partial evaluation

2.1. Background

Partial evaluation aims at specializing a program with respect to some of its input. The partial evaluation phase can be seen as a staging of the computations of the program: expressions that only operate on available data are executed during this phase; for the others, a residual expression is generated. This staging improves the execution time of the specialized program compared to the original program.

Using binding time analysis, these early computations can be identified independently of the actual values of the input. In essence, this phase determines when the value of a variable is available: if the value is known at partial evaluation time, it is said to be *static*; if it is not known until run-time it is *dynamic*. This information characterizes the computations that may be performed at partial evaluation-time—*static expressions*—or at run-time—*dynamic expressions*. Semantically speaking, binding time analysis determines the static and the dynamic semantics of a program for a given description of its inputs (i.e., known/unknown). This division greatly simplifies the partial evaluation process. Indeed, to process the static semantics, the partial evaluator simply follows the binding time information to reduce the static expressions of the program. This simplification of the partial evaluation process is crucial to self-application [21].

Self-application is achieved by specializing the partial evaluator with respect to an interpreter and yields a compiler. A compiler generator can be obtained by specializing the partial evaluator with respect to itself. Beyond the unusual aspects of these applications, they are of practical interest: compilation and generation of compilers are improved. Indeed, compilation using a generated compiler is about twelve times faster than compilation by specialization of an interpreter with respect to a program. A comparable speed-up is obtained for the generation of a compiler by applying the compiler generator to an interpreter rather than by specializing the partial evaluator with respect to an interpreter.

2.2. Schism

Schism [6,7,10] is a partial evaluator for a side-effect-free dialect of Scheme. The source programs are written in pure Scheme: a weakly typed, applicative order implementation of lambda calculus. Schism handles *higher-order functions* as well as the *data structures* manipulated by the source programs, even when they are only *partially known*. Schism is written in pure Scheme and is *self-applicable*. As such, it can generate a compiler out of the interpretive specification of a programming language.

Before describing the structure of Schism, let us first examine how function calls are partially evaluated. The treatment of function calls is one of the key aspects in processing successfully the Prolog definition.

2.2.1. Treatment of function calls

Partial evaluation of functional programs relies on the treatment of function calls. A function call can either be unfolded or suspended (that is, the function is specialized). Each transformation has a major pitfall which may cause non-termination of the partial evaluation process. Those pitfalls are infinite unfolding and infinite specialization.

This section first presents the treatment of function calls. Then, we describe the strategy used in Schism.

Call unfolding

Unfolding a function call consists in replacing the call by the result of partially evaluating the function body, in an environment binding the parameters to the arguments.

As an example, consider the following function,[1] which appends `n+1` numbers (from `m` to `m+n`) to a list (`l`).

```
(define (appendn n m l)
  (if (<? n 0)
      l
      (cons m (appendn (- n 1) (+ m 1) l))))
```

If the function call `(appendn 2 4 v)`, where variable `v` is dynamic, is unfolded, then the resulting expression is

```
(cons 4 (cons 5 (cons 6 v)))
```

In this case, unfolding is safe because the induction variable (`n`) is static. Although safe, unfolding may cause computations to be duplicated. This happens when a parameter occurs more than once and its corresponding argument is dynamic. However, this can easily be avoided by a preliminary analysis as described in [4].

Not all function calls can be unfolded. Consider the call `(appendn u v '(3 4))`, where `u` and `v` are dynamic. In this context, the recursion of function `appendn` is under dynamic control. Therefore, systematic unfolding of calls to this function will cause non-termination of partial evaluation.

When unfolding cannot be performed, the function call has to be suspended.

[1]Although contrived, this example illustrates many aspects of the treatment of function calls.

Call suspension

When a function call is suspended, a specialized version of the function is created with respect to the static arguments. A new function call consisting of the name of the specialized function and the dynamic arguments is substituted for the original function call. Consider the function call

```
(appendn u v '(3 4))
```

where u and v are dynamic, and assume it is suspended. The resulting specialized function is

```
(define (appendn-1 n m)
  (if (<? n 0)
      '(3 4)
      (cons m (appendn-1 (- n 1) (+ m 1)))))
```

and the new function call is (appendn-1 u v). Since a function is specialized with respect to static arguments, calls with different static arguments produce different specialized functions. However, whenever a function call is suspended, it is first determined whether a specialized version of the function called already exists for the same static arguments. If so, call suspension does not yield a new specialized function; it only consists in replacing the original call by the suspended call, as before. This is illustrated by the above specialized function appendn-1, where the recursive call also refers to appendn-1 since the suspended call contained the same static arguments.

It is not always safe to specialize a function with respect to all the static arguments of the call, some arguments may cause infinite specialization [7]. Consider the expression (appendn u 2 '(3 4)) where variable u is dynamic and assume that calls to function appendn are systematically suspended. The treatment of the initial call to function appendn causes a specialized version of function appendn to be created with respect to the values 2 and '(3 4). Since the induction variable is dynamic, both branches of the conditional in function appendn are partially evaluated. Therefore, the recursive call to function appendn causes a new specialized version of function appendn to be created. Indeed, the argument (+ m 1) causes the pattern of static arguments to be different from the previous one: previously it was 2 '(3 4), now it is 3 '(3 4). Since the condition to end the recursion is dynamic, each recursive call to function appendn will yield a new specialized function and this process will not terminate. To prevent infinite specialization, some static arguments should not be propagated.

In general, the problems caused by the treatment of function calls are solved by user annotations. In Schism, the construct filter makes it possible to specify how a call should be transformed and how static arguments should be propagated in the case of specialization.

2.2.2. *Filters*

In Schism, each function (or abstraction) contains a filter that controls the partial evaluation process. A filter specifies how to transform a function call (unfolding/suspension) and how to specialize a function, when suspension occurs. Typically, a function is defined as follows

```
(define (f p1 ... pn) (filter e1 e2) e3)
```

The first part of the filter (e_1) specifies whether a call to this function should be unfolded or suspended. It can be seen as

$$(\lambda(p_1 \ldots\ p_n).e_1) : B^n \to \{\text{UNFOLD, SPECIALIZE}\}.$$

It is invoked with the binding time values of the arguments of a call (domain B). If it returns UNFOLD, the call is unfolded. Otherwise, it returns SPECIALIZE; the call is then suspended and the function is specialized.

The way a function is specialized is specified by the second part of the filter (e_2). It can be viewed as

$$(\lambda(p_1 \ldots\ p_n).e_2) : B^n \to B^n.$$

It receives the binding time value of each argument of the call and returns a list of binding time values; each of which specifies if the corresponding argument should be propagated. As an example, consider again the function appendn

```
(define (appendn n m l)
  (filter (if (stat? n) UNFOLD SPECIALIZE)
          (list n DYN l))
  (if (<? n 0)
      l
      (cons m (appendn (- n 1) (+ m 1) l))))
```

where predicate stat? returns true if its argument is static. The above filter specifies that whenever the first argument of a call to function appendn is static, the function call should be unfolded. Otherwise, the call should be suspended. In specializing the function, the second argument should be considered as dynamic (DYN), and thus not propagated.

When no filter is provided in the definition of a function, a default filter is used by the system. For this experiment, the default filter specifies unconditional unfolding.

Keeping the annotation local to each function is crucial to control the partial evaluation process with respect to the context of a call. Consider the filter of function appendn

```
(filter (if (stat? n) UNFOLD SPECIALIZE)
        (list n DYN l))
```

It expresses different behaviors: if function appendn is called with the first argument (variable n) static, then unfold the call; otherwise, suspend the call and specialize the function without using the value of variable m. This contrasts with the existing strategies which are unconditional [17,20]: an annotation denotes an unconditional directive to the partial evaluator. Further discussion can be found in [6,7,10].

Note that one could introduce an automatic phase to annotate a program as to what to do for each function call. However, these annotations may lower the quality of the residual programs and can sometimes cause non-termination [30].

2.2.3. Structure of Schism

Schism is structured in three parts: one determining the static semantics of the program [8] (the binding time analysis); one specifying how to specialize the program [11]; and one specializing the program. Considering the scope of this paper we only discuss the binding time analysis.

Binding time analysis

As discussed earlier, binding time analysis determines the static and the dynamic expression of a program with respect to a description of its inputs. Schism handles both *partially static data* (i.e., data structure which contains both static and dynamic elements) and higher-order functions. For clarity, we simplify the set of binding time values presented in this paper. The symbols *St*, *Dy*, *Cl*, and *Ps* are used to denote respectively the binding time value "static", "dynamic", "closures", and "partially static data".

The binding time properties of a function are represented by a *binding time signature*: it specifies the binding time value of each parameter of a function and the binding time value of the result. For example, a possible binding time signature for function appendn would be

$$Appendn : Dy \times Dy \times Ps \rightarrow Dy.$$

This binding time signature specifies that the first and the second parameter of function appendn are dynamic; the third parameter is a partially static list; and the result is dynamic.

2.3. Related work

Partial evaluation of Prolog was taken up in [23]. Since then, several partial evaluators of Prolog have been developed, but mostly written in Prolog (e.g., [15,16,31]).

In [24], Komorowski presents a partial evaluator for Prolog programs. An *Abstract Prolog Machine* is defined in Lisp to describe the operational semantics of Prolog. Then, formally correct program transformations based on partial evaluation are introduced by enhancing this abstract machine. Self-application is not investigated.

Kahn and Carlsson describe in [22] a partial evaluator that specializes a Prolog interpreter, written in Lisp, with respect to a Prolog program, yielding an equivalent Lisp program. This program is then compiled into machine language using an existing Lisp compiler. The resulting programs are said to be efficient, however, the compilation phase appeared to be slow. They suggested that self-application could solve the problem but did not explore this issue.

Felleisen presents in [14] an implementation of Prolog in Scheme based on macro-expansion. Prolog entities are transliterated into corresponding Scheme constructs on a one-to-one basis. The efficiency of the code produced depends highly on how each Prolog entity is being transliterated. The approach does not address compile-time processing (no static reductions).

Bondorf and Danvy describe a first-order, binding-time-based partial evaluator for a subset of Scheme [4]. Its higher-order extension is presented in [2,3]; it relies on a monovariant binding time analysis.

3. Specification of Prolog

This section first gives an overview of the denotational semantics of Prolog as described in [27]. Then, its representation for Schism is described.

3.1. Denotational semantics

The denotational definition of the language consists of three parts: the abstract syntax, the semantics domains, and the valuation functions.

$I \in Ide$	Identifiers	
$B \in Con$	Constants symbols	
$F \in Fun$	Function/predicate symbols	$S ::= D, Q$
$G \in Goals$	Goal lists	$Q ::= \text{:-}\ G$
$P \in Pred$	Predicates and terms	$D ::= C, D_1$
$A \in Arg$	Argument lists	$C ::= P \mid P \text{ :- } G$
$C \in Clause$	Clause	$G ::= P, G_1$
$D \in Database$	Databases	$P ::= I \mid B \mid F(A)$
$S \in Prog$	Sentences (or programs)	$A ::= P, A_1 \mid P$
$Q \in Input$	Queries	

Fig. 1. Syntactic domains and syntactic rules.

Compound Domains

$\tau \in Tv \;\; = Var + Con + [Fun \times Av]$	Terms
$\pi \in Av \;\; = Tv^*$	Argument lists
$\rho \in Env = Ide \rightarrow Var$	Identifier environments
$\theta \in Subs = Var \rightarrow [Tv + uninstantiated]$	Substitutions

Continuation Functions

$\psi \in Qc = Subs \rightarrow State$	Substitution continuation
$\zeta \in Ec \;\; = Env \rightarrow Qc$	Continuation with environment
$\upsilon \in Tc \;\; = Tv \rightarrow Env \rightarrow Qc$	Terms
$\omega \in Ac = Av \rightarrow Env \rightarrow Qc$	Argument lists
$\kappa \in Kc \;\; = Res \rightarrow Qc$	Continuation with failure
$\gamma \in Gc \;\; = Env \rightarrow Res \rightarrow Qc$	Goal list
$\delta \in Db = Tv \rightarrow Kc \rightarrow Kc$	Database

Fig. 2. Semantic domains.

3.1.1. Abstract syntax

The abstract syntax of Prolog, described in Fig. 1, is due to Clocksin and Mellish [5]. Note that, as in [27], constructs which modify programs are not considered. The primitive syntactic domains are the domains of identifier symbols, constant symbols, and function or predicate symbols. These are called predicates and terms, or simply atoms, and are used to build goal sets, clauses, and complete programs. A clause can either be a *fact* or a *rule*. The latter consists of two components, a *conclusion* and a set of atoms called *premises*.

3.1.2. Semantic domains

This section briefly presents the semantic domains used in the denotational semantics of the language. The semantic domains are displayed in Fig. 2.

There are three types of terms that may appear in Prolog: the variables, the constants, and the functions. A function is made up of a function symbol and a list of terms that form the arguments to the function.

The domain *Env* is a finite mapping from identifiers (*Ide*) to variables (*Var*). The function

$$newvar = Subs \rightarrow Var$$

generates unique variables. Those variables can be seen as locations. The domain *Subs* maps variables to terms. The domain of final answers is $\phi \in Res$.

The crucial aspect of the semantics of Prolog is the control. Traditionally, it can be modeled by a semantic argument called *continuation*. As described in

[29], a backtracking facility can be integrated into a language by using a *failure continuation*. The continuation representing the usual evaluation sequence is called the *success continuation*. These two continuations capture the main aspects of the control of the denotational definition. The continuation functions are displayed in Fig. 2.

3.1.3. *Valuation functions*

The meaning of a program is given by the functions defined in Fig. 3. Functions $\mathcal{C}$ and $\mathcal{D}$ are responsible for the declaration of facts and rules, and for setting up the database. Function $\mathcal{G}$ processes queries and premises. Function $\mathcal{S}$ specifies the semantics of a program. Functions $\mathcal{P}$ and $\mathcal{A}$ assign new locations for identifiers in the current clause. Unify defines the unification process. We defer the description of function unify to Section 4.2; its implementation is displayed in Appendix B. Figure 4 gives a general idea of the call graph of the valuation functions. Note that for convenience, functions $\mathcal{P}$ and $\mathcal{A}$ appear twice in this figure. Indeed, they first manipulate goals (upper part of the diagram) and then clauses (lower part).

$$
\begin{array}{l}
\mathcal{S} : Prog \to Input \to Res \\
\quad \mathcal{S}[\![D]\!][\![{:}\!-\ G.]\!] \\
\qquad = \mathcal{G}[\![G]\!](\mathcal{D}[\![D]\!])\ printall\ (\lambda\iota.unbound)\ \langle\,\rangle\ (\lambda\upsilon.unused) \\
\mathcal{D} : Clause \to Tv \to Kc \to Kc \\
\quad \mathcal{D}[\![D]\!] = fixedpoint(\mathcal{C}[\![D]\!]) \\
\mathcal{C} : Clause \to Db \to Tv \to Kc \to Res \to Subs \to Res \\
\quad \mathcal{C}[\![C_1, C_2]\!]\delta\tau\kappa\phi\theta = \mathcal{C}[\![C_1]\!]\delta\tau\kappa(\mathcal{C}[\![C_2]\!]\delta\tau\kappa\phi\theta)\theta \\
\quad \mathcal{C}[\![P.]\!]\delta\tau\kappa\phi = \mathcal{P}[\![P]\!](\lambda\tau_1\rho.unify\ \tau\tau_1\kappa\phi)(\lambda\iota.unbound) \\
\quad \mathcal{C}[\![P\ {:}\!-\ G.]\!]\delta\tau\kappa\phi \\
\qquad = \mathcal{P}[\![P]\!](\lambda\tau_1\rho_1.unify\ \tau\tau_1(\mathcal{G}[\![G]\!]\delta(\lambda\rho_2.\kappa)\rho_1)\phi)(\lambda\iota.unbound) \\
\mathcal{G} : Goals \to Db \to Gc \to Env \to Res \to Qc \\
\quad \mathcal{G}[\![P]\!]\delta\gamma\rho\phi = \mathcal{P}[\![P]\!](\lambda\tau\rho_1.\delta\tau(\gamma\rho_1)\phi)\rho \\
\quad \mathcal{G}[\![G_1, G_2]\!]\delta\gamma = \mathcal{G}[\![G_1]\!]\delta(\mathcal{G}[\![G_2]\!]\delta\gamma) \\
\mathcal{P} : Pred \to Tc \to Ec \\
\quad \mathcal{P}[\![I]\!]\upsilon\rho = \rho I = unbound \to newvar(\lambda\tau.\upsilon\tau\rho[\tau/I]), \upsilon(\rho I)\rho \\
\quad \mathcal{P}[\![F(A)]\!]\upsilon = \mathcal{A}[\![A]\!]\lambda\zeta.\upsilon\langle F, \zeta\rangle \\
\mathcal{A} : Arg \to Ac \to Env \to Qc \\
\quad \mathcal{A}[\![P]\!]\omega = \mathcal{P}[\![P]\!]\lambda\tau.\omega\langle\tau\rangle \\
\quad \mathcal{A}[\![P, A]\!]\omega = \mathcal{P}[\![P]\!]\lambda\tau.\mathcal{A}[\![A]\!]\lambda\zeta.\omega(\tau.\zeta) \\
Unify : Tv \to Tv \to Kc \to Res \to Subs \to Res
\end{array}
$$

Fig. 3. Valuation functions.

$$\begin{tikzcd}
S \arrow[r] & G \arrow[d] & \\
A \arrow[r, leftrightarrow] & P \arrow[r] & D \arrow[d] \\
& & C \arrow[d] \\
& A \arrow[r, leftrightarrow] & P \arrow[d] \\
& & \text{Unify} \arrow[uuuul, bend right] \arrow[uuu, bend right] \arrow[loop below]
\end{tikzcd}$$

Fig. 4. The call graph of the valuation functions.

3.2. *Interpreter*

This section presents the Schism representation for the language as described above. To ease the coding of the denotational definition, our system provides the user with constructs that define and manipulate types; they are simplified versions of ML constructs. The construct `defineType` defines a product or a sum depending on whether it contains one clause or more. The constructs `let` and `let*` create new bindings, as in Scheme, but in addition may perform destructuring operations on elements of products. The construct `caseType` is a conditional on the injection tag of the element of a sum and allows the destructuring of this element.

Except for a few technical details, this interpreter is a direct coding in Scheme of the valuation functions presented above. As a result, the implementation is precise and easy to reason about.

3.2.1. *Abstract syntax*

The abstract syntax of the language is defined by declaring the appropriate data types. As shown in Fig. 5, these declarations are direct coding of the abstract syntax presented in the denotational semantics.

3.2.2. *Data structures of the interpreter*

The data type corresponding to the domain of denotable values is defined in Fig. 6.

There is a data structure called *state*, which captures the dynamic aspects of the specification. This consists of the accumulation of results obtained from

```
(defineType Atom
   (Constant           value)
   (Identifier         name)
   (Predicate function arguments))

(defineType Clause
   (Fact            atom)
   (Rule conclusion premises))
```

Fig. 5. Abstract syntax.

```
(defineType Term
   (Const         value)
   (Var           name ref)
   (Pred function arguments))
```

Fig. 6. Denotable values.

executing a Prolog program and some mechanism that provides a new variable when a local identifier is defined.

3.2.3. Interpretation functions

The interpretation functions, corresponding to the valuation functions, are shown in Figs. 7 and 8. These interpretation functions differ from the semantic valuation functions in two ways.

First, interpretation functions are uncurried. This transformation makes it possible to use conventional techniques for function specialization: these techniques are limited to uncurried functions. Note that one could introduce a phase that would uncurry functions automatically as is usually done in compilers for functional programs.

Second, filters are used in some functions to specify how calls to these functions should be treated. This issue is addressed in Section 4.3.

4. The interpreter from a partial evaluation point of view

The Prolog interpreter (with main function S) receives two inputs: the first is the Prolog program, called the database; the second input is the query. The first input is static, and the second is dynamic.

$$S : Prog \times Input \rightarrow Res.$$

```
(define (S database queries)
  (State-result
    (G queries database
       (lambda (env s1 subs)
               (update-state-result s1 (Q:inst env subs)))
       (lambda (s1) s1)
       (init-env) (init-state) (init-subs))))

(define (G goals database gc fc env stt subs)
  (filter SPECIALIZE
          (list goals database DYN fc env stt subs))
  (cond
    ((null? goals) (gc env stt subs))
    (else
      (P (car goals)
         (lambda (goal e st1)
           (D goal database database
              (lambda (env1 st2 subs1)
                (G (cdr goals) database gc fc e st2 subs1))
              fc env st1 subs))
         env stt))))

(define (P t tc env stt)
  (filter (if (static? t) UNFOLD SPECIALIZE)
          (list DYN DYN DYN DYN))
  (caseType t
    ([Constant number] (tc (Const number) env stt))
    ([Identifier name]
     (let ([varlist (associate name env)])
       (if (null? varlist)
           (let* ([(list v0 st1) (New-var name stt)]
                  [v (Var name v0)])
                 (tc v (cons (cons name v) env) st1))
           (tc (cdr varlist) env stt))))
    ([Predicate fn args]
     (A args (lambda (arglis e st1)
                (tc (Pred fn arglis) e st1))
           env stt))))
```

Fig. 7. Interpretation functions.

```
(define (D t clauses database gc fc env stt subs)
  (filter SPECIALIZE (list t clauses database gc fc env stt subs))
  (if (null? clauses) (fc stt)
      (C (car clauses) database t
         (lambda (goals e st1 subs1)
           (D t (cdr clauses) database gc fc env
              (G goals database gc fc e st1 subs1) subs))
         (lambda (env1 st1 subs1)
           (D t (cdr clauses) database gc fc env
              (gc env1 st1 subs1) subs))
         (lambda (st1)
           (D t (cdr clauses) database gc fc env st1 subs))
         env stt subs)))

(define (C clause db t pc gc fc env stt subs)
  (caseType clause
    ([Fact term]
     (P term (lambda (tv e st1)
               (D-unify tv t (lambda (r) (gc e st1 r))
                        (lambda (r) (fc st1)) subs))
               (init-env) stt))
    ([Rule conclusion premises]
     (P conclusion
        (lambda (tv e st1)
          (D-unify tv t (lambda (r) (pc premises e st1 r))
                   (lambda (r) (fc st1)) subs))
        (init-env) stt))))

(define (A args w env stt)
  (filter (if (static? args) UNFOLD SPECIALIZE)
          (list DYN DYN DYN DYN))
  (if (null? args)
      (w '() env stt)
      (P (car args)
         (lambda (t e1 st1)
           (A (cdr args)
              (lambda (arglis e st2)
                      (w (cons t arglis) e st2)) e1 st1))
         env stt)))
```

Fig. 8. Interpretation functions (*continued*).

During partial evaluation, the interpreter is specialized with respect to a Prolog program. The resulting residual program (with specialized main function S_{Prog}) accepts an input query and yields a result.

$$S_{Prog} : Input \rightarrow Res.$$

4.1. Multiple binding time signatures

This section discusses the binding time properties of the Prolog interpreter. It is illustrated with binding time signatures of the interpretation functions. For simplicity, we assume that the binding time value of the result of a function is dynamic.

Initially, the main function S calls function G with the program, which is static, and the input query, which is *dynamic*. Then, to satisfy subgoals function G is called *recursively*, but with *static* goals, i.e., the premises (see function D in Fig. 8). Therefore, function G and the inner functions are called firstly in a context of a dynamic query, and then, in a context of a static query.

This is illustrated in Fig. 9 where the binding time signatures of the interpretation functions are displayed. The functions have two binding time signatures to reflect the fact that they are called in two different binding time contexts.

Note, that partially static data arise because of terms containing both static and dynamic data. These dynamic data are retrieved from the environment which binds static identifiers to dynamic variables.

```
(G goals database gc fc env state subs)
```
$$G : Dy \times St \times Cl \times Cl \times Dy \times Dy \times Dy \rightarrow Dy$$
$$St \times St \times Cl \times Cl \times Ps \times Dy \times Dy \rightarrow Dy$$
```
(D t clauses database gc fc env state subs)
```
$$D : Dy \times St \times St \times Cl \times Cl \times Dy \times Dy \times Dy \rightarrow Dy$$
$$Ps \times St \times St \times Cl \times Cl \times Ps \times Dy \times Dy \rightarrow Dy$$
```
(C clause db t pc gc fc env state subs)
```
$$C : St \times St \times Dy \times Cl \times Cl \times Cl \times Dy \times Dy \times Dy \rightarrow Dy$$
$$St \times St \times Ps \times Cl \times Cl \times Cl \times Ps \times Dy \times Dy \rightarrow Dy$$
```
(P t tc env state)
```
$$P : Dy \times Cl \times Dy \times Dy \rightarrow Dy$$
$$St \times Cl \times Ps \times Dy \rightarrow Dy$$
```
(A args w env state)
```
$$A : Dy \times Cl \times Dy \times Dy \rightarrow Dy$$
$$St \times Cl \times Ps \times Dy \rightarrow Dy$$

Fig. 9. Binding time signatures for the functions.

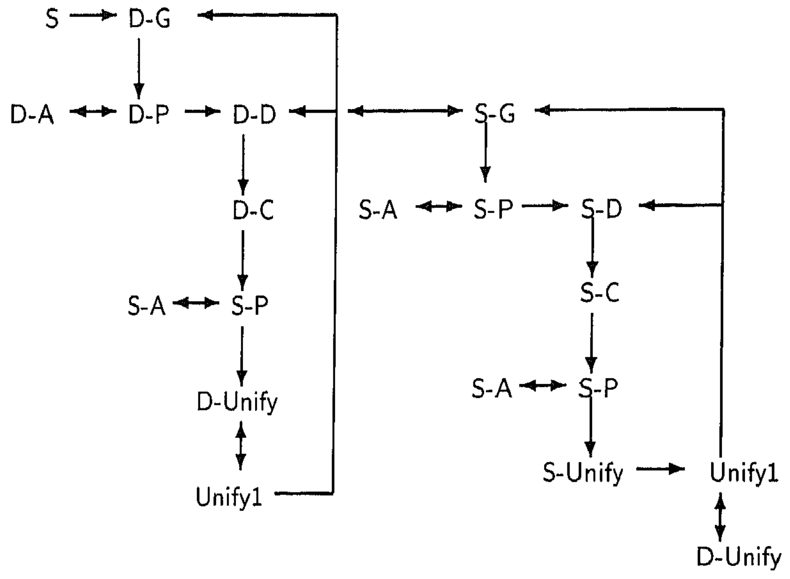

Fig. 10. The call graph of the interpretation functions.

Since each function has two different binding time signatures, if a binding time analysis maps each function to only one binding time signature, it will have to *fold* these two binding time signatures into one. As a result, the binding time information will be less precise and consequently there will be less static processing during partial evaluation.

To avoid this situation, we duplicate the original set of interpretation functions: one set of functions deals with the initial query (dynamic), the other manipulates the premises (static). To distinguish these functions, we prefix a function by "D-" if it belongs to the first set, and by "S-" if it belongs to the second set. Figure 10 shows the call graph of the resulting program. Because each function of this program is now called with only one pattern of binding time values, the binding time analysis will not do any "folding" and consequently more static computations will be detected.

Another way to handle multiple binding time signatures is to enhance the binding time analysis so that it determines multiple binding time signatures for each function; such binding time analysis is called *polyvariant*. A polyvariant binding time analysis for a first-order functional language is presented in [7] and its extension to higher-order functions is discussed in [9].

4.2. *Unify*

Unification of two terms involves comparing these two terms and performing the instantiation of variables to terms when needed. The result of the instantiation of a variable to a term is stored in the substitution list. The unification process is defined by two functions (`unify` and `unify1`). This allows the partial evaluator to use the static parts of the unified terms when the interpreter manipulates the premises. Appendix B displays the code for unification.

4.3. Termination of partial evaluation

In Section 2.2.1, we examined the treatment of function calls and its pitfalls. In this section, we give some insights as to how filters are specified for the Prolog interpreter.

Infinite call unfolding may occur when recursive calls to a function under dynamic control are systematically unfolded. This situation may happen in the recursive calls to functions `G`, `D`, and `unify1`, (see Fig. 10). Therefore, we instruct the partial evaluator to create specialized versions of these functions. For instance, the interpretation function for `G` in Fig. 7 has a filter whose first part instructs specialization.

Infinite function specialization may arise when recursive calls are systematically suspended and the static arguments are always different. This situation may occur in the interpretation function `G`. Notice that the success continuation passed to function `G` (i.e., variable `gc`) is a closure. Schism is unable to compare extensionally two closures; that is, it cannot compare the values they capture. Since function `G` is called (from function `D`) with a new closure every time, this can cause infinite function specialization. Therefore, the second part of the filter of function `G` prevents parameter gc from being propagated.

5. Assessment

In this section we discuss what has been achieved by partial evaluation, and evaluate its performance.

5.1. What has actually been processed by the partial evaluation process?

The failure continuation has been eliminated

Compiled programs have an interesting property: *the failure continuation has been completely eliminated.* Therefore, the backtracking has been determined statically. This is because the database is static.

Consider the Prolog program in Fig. 11 and its corresponding residual program displayed in Appendix A. The compilation of the backtracking continuation can be illustrated by comparing the traversal of the database that the interpreter would performed (Fig. 12(a)) with the traversal of the database represented by the compiled program (Fig. 12(b)). These diagrams also include the accumulation of a result which is denoted by the symbol *res*. The

```
c1    Male (Adam),
c2    Female (Eve),
c3    Person (x) :- Male (x),
c4    Person (x) :- Female (x).
```

Fig. 11. A source program.

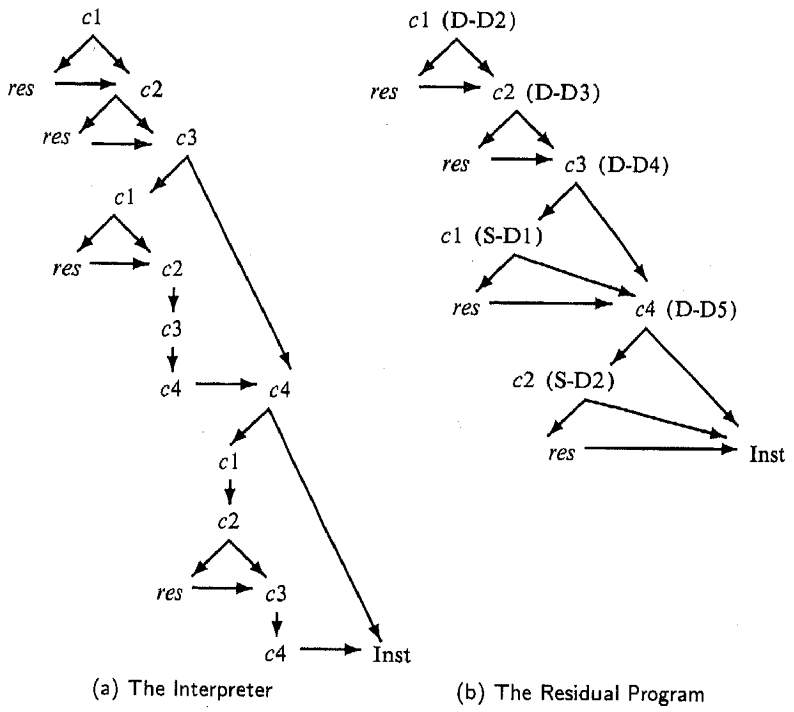

Fig. 12. The traversal of the database.

labels $c1, \ldots, c4$ correspond to the clauses of the source program. Each clause in Fig. 12(b) has attached its corresponding specialized function.

Figure 12 clearly shows that the intermediate backtracking has been eliminated in the compiled code.

Lookup operations in the environment are compiled

Because the environment is partially static (static identifiers and dynamic variables), the access to a given identifier/variable pair has been compiled. The resulting expression is a sequence of operations to access the variable in this pair.

Table 1
Static versus dynamic functions

Static functions	Dynamic functions
S-C	D-C
S-P, S-A	D-P, D-A
S-Unify	D-Unify, Unify1
S-lookup-env	D-lookup-env
	S, S-G, D-G
	lookup-subs, update-subs
	New-var
	Q:inst

Part of the unification process is compiled

When the interpreter manipulates premises, part of the unification process can be performed. Indeed, in this context, the type of terms to unify is static and thus processing depending on this information can be performed.

5.2. *What are the dynamic operations?*

Some operations cannot be performed during the partial evaluation process because some data are not available. The printing of the result is deferred to run-time (function `Q:inst`). Part of the unification process is frozen (function `unify1`). Since the variables cannot be generated until run-time, the substitution list is dynamic. Therefore, operations that manipulate the substitution list are frozen.

Table 1 summarizes the above explanations by classifying the interpretation functions as static if they are eliminated during partial evaluation and dynamic otherwise.

5.3. *Performance of partial evaluation*

Programs compiled by partial evaluation have been found to be about six times faster than interpretation. Note that this speed-up is more important with other languages. For Algol-like programs, we reported in [12] that compiled code has been found to be more than a hundred times faster than the interpreted source program. This difference is due to the fact that the static semantics of Prolog is not as important as other languages. It is especially difficult to see how unification could be further processed statically without introducing special purpose program transformations. As we have seen in the example of the compiled code, the unification is the major component of the dynamic semantics. It is interesting to notice that partial evaluators for Prolog written in Prolog do not deal with unification either. Indeed, unification is part of the target language.

It is generally difficult to fairly evaluate the performance improvement obtained by partial evaluation and the size of the resulting programs since

Table 2
Speed-up with compilation

Prolog program		Speed-up
Number of facts	Number of rules	
4	4	4
6	2	5
8	8	7

they strongly depend on the specificity of the Prolog program. In particular, we notice that the speed-up with compilation is related to the number of possible unsuccessful unifications contained in the source program. Indeed, the intermediate backtracking have been removed by partial evaluation.

Some run-time results are displayed in Table 2; they are obtained with both the interpreter and the residual programs compiled into machine language using a Scheme compiler.

The size and the structure of compiled code are not surprising: the residual program represents the traversal of the database. A specialized function is generated for each unification clause. This is illustrated in Appendix A.

Further work needs to be done to extend the Prolog interpreter for a larger subset of the language. To improve the size of target code, we want to investigate combinator-based semantics [29]. This approach could capture more compactly the dynamic semantics and be more abstract with respect to its implementation. Consequently, different strategies for implementing the dynamic semantics could be explored to improve the run-time of target code.

5.4. Incorporating Cut operation

The Cut operation is used to control backtracking. It can be modeled semantically by another continuation function. This requires few modifications to the Prolog semantics as described in [27]. We have applied similar modifications to our Prolog interpreter to include this operation. When specializing the resulting interpreter with respect to a program including the Cut operation, the failure continuation is eliminated. This is not surprising since the Cut operation is part the program it can be treated statically.

6. Specializing further compiled programs

So far, we consider the input query as being dynamic and the database static. As a result, the residual programs are general: they can handle any query. One might want to obtain more specific residual programs, i.e., programs dedicated to a query restricted to some predicates. This could be achieved by extending the Prolog interpreter to deal with incompletely specified queries. They are called *partial queries.*

For example, consider the Prolog program given in Appendix A. A partial query could be

```
Male (□)
```

where □ stands for the missing part of the query. Note that, a partial query only amounts to abstracting away the atomic arguments of a predicate (constants or identifiers). It is not desirable to abstract away predicates because we would have to keep the residual programs as general as before. Indeed, the missing part of the query could then be a predicate; consequently, the residual program should handle any query.

Let us examine what can be determined from the partial query `Male (□)`. We know that only clause *c*1 in the Prolog program can match with this partial query. This means that the partial evaluator will use the partial query to eliminate irrelevant clauses and yield a residual program dedicated to this partial query. As a result, the size of the residual program will be drastically reduced. This technique of using partial input has been investigated in [24] for source-to-source program transformations; the resulting program is called a *pruned version.*

6.1. *Handling partial query*

In essence a partial query is partially static; it contains some dynamic (unknown) parts. However, Schism can only receive completely static or completely dynamic inputs. To circumvent this limitation, we choose to split the partially static input into a static and a dynamic input. Since Schism handles partially static data, the partial query can then be reconstructed from the static and dynamic inputs.

Handling partial query requires few changes to the Prolog interpreter. Only the initial function (`S`) and the syntax of a query have to be modified. A query may now contains *meta-variables,* i.e., variables that abstract away the missing parts. These missing parts can only be identifiers or constants. Also, we introduce another input to the Prolog interpreter, called the *meta-environment,* to represent the missing parts of the query. It is a mapping from meta-variables into their values (identifiers or constants). The changes are summarized in Fig. 13. This new syntax allows to split a query into a static part—the partial query—and a dynamic part—the meta-environment.

6.2. *Modifying the interpreter*

As mentioned above, only function `S` has to be modified for the Prolog interpreter to handle partial queries. The new function `S` is displayed in Fig. 14. It now has three parameters: the database, the partial queries and

$$\vdots$$

$V \in \mathit{MetaVar}$	Meta-variables
$M \in \mathit{MetaEnv}$	Meta-environment

$$\begin{aligned}
S &::= D, Q, M\\
Q &::= \text{:-}\ G'\\
D &::= C, D_1\\
C &::= P \mid P \text{ :- } G\\
G' &::= F(V), G_1' \mid P, G_2'\\
G &::= P, G_1\\
P &::= I \mid B \mid F(A)\\
A &::= P, A_1 \mid P\\
M &::= (V, I), M_1 \mid (V, B), M_2
\end{aligned}$$

Fig. 13. Modified syntactic domains and syntactic rules.

```
(define (S database queries menv)
  (Let ([queries1 (complete-query query menv)])
    (State-result
      (G queries1 database
         (lambda (env s1 subs)
                (update-state-result s1 (Q:inst env subs)))
         (lambda (s1) s1)
         (init-env) (init-state) (init-subs)))))
```

Fig. 14. New function S handling partial queries.

the meta-environment. Parameters `queries` and `menv` respectively are static and dynamic; they represent the complete query. Function `complete-query` is invoked with these two components to construct the complete query. The rest of the interpreter is unchanged. Appendix D displays function `complete-query`.

Appendix A. Prolog program and its residual

A.1. A source program

```
c1 Male (Adam),
c2 Female (Eve),
c3 Person (x) :- Male (x),
c4 Person (x) :- Female (x).
```

A.2. The residual program (sugared)

```
(define (S0 queries)
  (state-result
    (D-G1 queries '() (init-state) (init-subs))))

; Comparing query with each clauses in database

(define (D-G1 goals env stt subs)
  (cond
    ((null? goals) (update-state-result stt (Q:INST env subs)))
    (else
      (D-P (car goals)
           (lambda (goal2 e3 st1)
             (D-D2 goal2
               (lambda (e2 subs1 st2)
                 (D-G1 (list-tail goals 1) subs1 st2 e3))
               env st1 subs))
           env stt))))

(define (D-D5 t gc env stt subs)
  (let* ([(list v0 st1) (new-var 'x stt)]
         [v1 (Var 'x v0)])
    (D-Unify (Pred 'person (list v1)) t
             (lambda (r1) (S-G1 gc (list (cons 'x v1)) st1 r1))
             (lambda (r1) st1)
             subs)))

(define (D-D4 t gc env stt subs)
  (let* ([(list v0 st1) (new-var 'x stt)]
         [v1 (Var 'x v0)])
    (D-Unify (Pred 'person (list v1)) t
             (lambda (r1)
               (D-D5 t gc env (S-G2 gc (list (cons 'x v1)) st1 r1)
                     subs))
             (lambda (r1) (D-D5 t gc env st1 subs st1))
             subs)))

(define (D-D3 t gc env stt subs)
  (D-Unify (Pred 'female '((Const eve))) t
           (lambda (r1) (D-D4 t gc env (gc '() stt r1) subs))
           (lambda (r1) (D-D4 t gc env stt subs))
           subs))
```

```
(define (D-D2 t gc env stt subs)
  (D-Unify (Pred 'male '((Const adam))) t
           (lambda (r1) (D-D3 t gc env (gc '() stt r1) subs))
           (lambda (r1) (D-D3 t gc env stt subs)) subs))

; Handling premises begins

(define (S-G1 gc env stt subs)
  (S-D2 (Pred 'female (list (list-tail (car env) 1)))
        (lambda (e st2 su1) (gc env st2 su1)) env stt subs))

(define (S-G2 gc env stt subs)
  (S-D1 (Pred 'male (list (list-tail (car env) 1)))
        (lambda (e st2 su1) (gc env st2 su1)) env stt subs))

; Comparing premises with clauses in database

(define (S-D2 t gc env stt subs)
  (unifyargs
   '((Const eve)) (list-ref t 2)
    (lambda (r1)
      (list-ref
        (new-var 'x (list-ref (new-var 'x (gc '() stt r1)) 1)) 1))
    (lambda (r1)
      (list-ref (new-var 'x (list-ref (new-var 'x stt) 1)) 1))
    subs))

(define (S-D1 t gc env stt subs)
  (unifyargs
   '((Const adam)) (list-ref t 2)
    (lambda (r1)
      (list-ref
        (new-var 'x (list-ref (new-var 'x (gc '() stt r1)) 1)) 1))
    (lambda (r1)
      (list-ref (new-var 'x (list-ref (new-var 'x stt) 1)) 1))
    subs))
```

Appendix B. Unification

```
(define (unify t1 t2 gc res subs)
  (filter (if (dynamic? t2) SPECIALIZE UNFOLD)
          (list DYN DYN DYN DYN DYN))
  (caseType t1
    ([Var - n1]
     (caseType t2
       ([Var - n2]
        (if (equal? n1 n2)
            (gc subs)
            (unify1 t1 t2 gc res subs)))
       (else (unify1 t1 t2 gc res subs))))
    ([Const n1]
     (caseType t2
       ([Const n2]
        (if (equal? n1 n2) (gc subs) (res '())))
       ([Var - -] (unify1 t1 t2 gc res subs))
       (else (res '()))))
    ([Pred f1 args1]
     (caseType t2
       ([Var - -] (unify1 t1 t2 gc res subs))
       ([Const -] (res '()))
       ([Pred f2 args2]
        (if (equal? f1 f2)
            (unifyargs args1 args2 gc res subs)
            (res '()))))))))

(define (unify1 t1 t2 gc res subs)
  (filter SPECIALIZE
          (list DYN DYN DYN DYN DYN))
  (caseType t1
    ([Var - n1]
     (caseType t2
       ([Var - n2]
        (cond ((equal? (lookup-subs subs t1) 'uninstantiated)
               (if (equal? (lookup-subs subs t2) 'uninstantiated)
                   (gc (update-subs subs t1 t2))
                   (unify t1 (lookup-subs subs t2) gc res subs)))
              (else
                (unify (lookup-subs subs t1) t2 gc res subs))))
       (else (if (equal? (lookup-subs subs t1) 'uninstantiated)
                 (gc (update-subs subs t1 t2))
                 (unify (lookup-subs subs t1) t2 gc res subs)))))
    (else
      (if (equal? (lookup-subs subs t2) 'uninstantiated)
          (gc (update-subs subs t2 t1))
          (unify t1 (lookup-subs subs t2) gc res subs)))))
```

```
(define (unifyargs l1 l2 gc res subs)
  (filter SPECIALIZE
          (list DYN DYN DYN DYN DYN))
  (cond
    ((and (null? l1) (null? l2)) (gc subs))
    ((or (null? l1) (null? l2)) (res '()))
    (else (unify (car l1) (car l2)
                 (lambda (x) (unifyargs (cdr l1) (cdr l2) gc res x))
                 (lambda (x) (res x))
                 subs))))
```

Appendix C. The rest of the Prolog interpreter

```
;; Auxiliary functions

(define (associate id env)
  (filter (if (static? id) UNFOLD SPECIALIZE)
          (list DYN DYN))
  (if (null? env)
      '()
      (if (equal? id (car (car env)))
          (car env)
          (associate id (cdr env)))))

;; Manipulating the substitution

(define (lookup-subs subs v)
  (let* ([(Var - num) v])
    (lookup-subs/1 subs num)))

(define (lookup-subs/1 subs num)
  (filter SPECIALIZE (list subs num))
  (let ((t (assq num subs)))
    (if (null? t) 'uninstantiated (cdr t))))

(define (update-subs subs v t)
  (filter SPECIALIZE (list DYN DYN))
  (let* ([(var - num) v])
    (cons (cons num t) subs)))

;; This section treats the data after unification. It
;; instantiates and generates the final result.
```

```
(define (Q:inst env subs)
  (filter SPECIALIZE (list DYN DYN))
  (if (null? env)
      '()
      (let ([name (car (car env))] [v (cdr (car env))])
        (cons (cons name (Q:inst/1 v subs (lambda (v f) f)))
              (Q:inst (cdr env) subs)))))

(define (Q:inst/1 query subs unbound-var-handler)
  (caseType query
    ([Const num] (Constant num))
    ([Var id -]
     (let ([term (lookup-subs subs query)])
       (if (equal? term 'uninstantiated)
           (unbound-var-handler (Identifier id) subs)
           (Q:inst/1 term subs unbound-var-handler))))
    ([Pred fn args]
     (let ([targs (Q:inst args subs)]) (Predicate fn targs)))))

(define (update-state-result stt res)
  (let ([(State idx result) stt])
    (State idx (cons res result))))

(define (state-result stt)
  (let ([(State idx result) stt]) result))
```

Appendix D. Constructing complete queries

```
(define (complete-query queries menv)
  (map (lambda (a) (complete-a-query a menv)) queries))

(define (complete-a-query query menv)
  (caseType query
    ([Constant number] query)
    ([Identifier name] query)
    ([Predicate fn args]
     (Predicate fn
       (map (lambda (a) (complete-meta a menv)) args)))))

(define (complete-meta arg menv)
  (caseType arg
    ([Constant number] arg)
([Identifier name] arg)
([MetaVar    name] (cdr (associate name menv)))
([Predicate fn args]
 (Predicate fn
   (map (lambda (a) (complete-meta a menv)) args)))))
```

Acknowledgments

This work has benefited from David Schmidt's interest and insightful comments. Thanks are also due to Olivier Danvy and Paul Hudak for their thoughtful comments.

References

[1] D. Bjørner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988).

[2] A. Bondorf, Automatic autoprojection of higher-order recursive equations, *Sci. Comput. Programming* **17** (1991) 3–34.

[3] A. Bondorf, Improving binding times without explicit CPS-conversion, in: *Proceedings ACM Conference on Lisp and Functional Programming* (1992) 1–10.

[4] A. Bondorf and O. Danvy, Automatic autoprojection of recursive equations with global variables and abstract data types, *Sci. Comput. Programming* **16** (1991) 151–195.

[5] W.F. Clocksin and C.S. Mellish, *Programming in Prolog* (Springer, Berlin, 1981).

[6] C. Consel, New insights into partial evaluation: the Schism experiment, in: H. Ganzinger, ed., *ESOP'88, 2nd European Symposium on Programming*, Lecture Notes in Computer Science **300** (Springer, Berlin, 1988) 236–246.

[7] C. Consel, Analyse de programmes, evaluation partielle et génération de compilateurs, Ph.D. Thesis, Université de Paris VI, Paris, France (1989).

[8] C. Consel, Binding time analysis for higher order untyped functional languages, in: *Proceedings ACM Conference on Lisp and Functional Programming* (1990) 264–272.

[9] C. Consel, Polyvariant binding-time analysis for higher-order, applicative languages, in: *Proceedings ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1993) 145–154.

[10] C. Consel, A tour of Schism: a partial evaluation system for higher-order applicative languages, in: *Proceedings ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1993) 66–77.

[11] C. Consel and O. Danvy, From interpreting to compiling binding times, in: N.D. Jones, ed., *ESOP'90, 3rd European Symposium on Programming*, Lecture Notes in Computer Science **432** (Springer, Berlin, 1990) 88–105.

[12] C. Consel and O. Danvy, Static and dynamic semantics processing, in: *Proceedings ACM Symposium on Principles of Programming Languages* (1991) 14–23.

[13] S.K. Debray and P. Mishra, Denotational and operational semantic for Prolog, *J. Logic Programming* **5** (1988) 61–91.

[14] M. Felleisen, Transliterating Prolog into Scheme, Tech. Report 182, Indiana University, Bloomington, IN (1985).

[15] H. Fujita and K. Furukawa, A self-applicable partial evaluator and its use in incremental compiler, in: Y. Futamura, ed., *New Generation Computing* **6** (Ohmsha, Tokyo/Springer, Berlin, 1988).

[16] D.A. Fuller and S. Abramsky, Mixed computation of Prolog, in: D. Bjørner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988).

[17] A. Haraldsson, A program manipulation system based on partial evaluation, Ph.D. Thesis, Linköping Studies in Science and Technology Dissertations **14**, Linköping University, Sweden (1977).

[18] N.D. Jones, ed., *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science **94** (Springer, Berlin, 1980).

[19] N.D. Jones and A. Mycroft, Stepwise development of operational and denotational semantics for Prolog, in: *Proceedings IEEE International Symposium on Logic Programming* (1984) 289–298.

[20] N.D. Jones, P. Sestoft and H. Søndergaard, An experiment in partial evaluation: the generation of a compiler generator, in: J.-P. Jouannaud, ed., *Rewriting Techniques and Applications, Dijon, France*, Lecture Notes in Computer Science **202** (Springer, Berlin, 1985) 124–140.

[21] N.D. Jones, P. Sestoft and H. Søndergaard, Mix: a self-applicable partial evaluator for experiments in compiler generation, *Lisp and Symbolic Computation* **2** (1) (1989) 9–50.

[22] K.M. Kahn and M. Carlsson, The compilation of Prolog programs without the use of Prolog compiler, in: *Proceedings International Conference on Fifth Generation Computer Systems*, Tokyo, Japan (1984) 348–355.

[23] H.J. Komorowski, A specification of an abstract Prolog machine and its application to partial evaluation, Linkoping Studies in Science and Technology Dissertations **69**, Linkoping University, Sweden (1981).

[24] H.J. Komorowski, Partial evaluation as a means for inferencing data structures in an applicative language: a theory and implementation in the case of Prolog, in: *Proceedings ACM Symposium on Principles of Programming Languages* (1982).

[25] P. Lee and U.F. Pleban, On the use of Lisp in implementing denotational semantics, in: *Proceedings ACM Conference on Lisp and Functional Programming* (1986) 233–248.

[26] P.A. Mosses, Compiler generation using denotational semantics, in: *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **45** (Springer, Berlin, 1976) 436–441.

[27] T. Nicholson and N. Foo, A denotational semantics for Prolog, *ACM Trans. Programming Languages Syst.* **11** (4) (1989).

[28] J. Rees and W. Clinger, Revised^3 report on the algorithmic language Scheme, *SIGPLAN Notices* **21** (12) (1986) 37–79.

[29] D.A. Schmidt, *Denotational Semantics: A Methodology for Language Development* (Allyn and Bacon, Newton, MA, 1986).

[30] P. Sestoft, Automatic call unfolding in a partial evaluator, in: D. Bjørner, A.P. Ershov and N.D. Jones, eds., *Partial Evaluation and Mixed Computation* (North-Holland, Amsterdam, 1988).

[31] R. Venken, A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimisation, in: T. O'Shea, ed., *ECAI'84* (North-Holland, Amsterdam, 1984).

[32] M. Wand, A semantic prototyping system, *SIGPLAN Notices* (*Proceedings ACM Symposium on Compiler Construction*) **19** (6) (1984) 213–221.