Semantics-Directed Generation of a Prolog Compiler

Charles Consel and Siau Cheng Khoo
Research Report YALEU/DCS/RR-781
May 1990

This work is supported by the Darpa grant N00014-88-K-0573.

Semantics-Directed Generation of a Prolog Compiler

Charles Consel Siau Cheng Khoo

Yale University
Department of Computer Science

May 9, 1990

Abstract

In a recent paper [17], the denotational semantics for the language Prolog was presented. The aim was to define precisely the language which had only been described informally.

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, 9, 22] 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 language Prolog was recently presented in [17]. The aim was to formalize the core of Prolog. Beyond the theoretical interest, it has long been argued that denotational definitions can be used to derive interpreters or compilers [19, 22, 16]. This approach is attractive because it closely relates the formal specification and its implementation.

This paper describes how the denotational semantics 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 [18]. 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 [13], the target code has the same behaviour as the interpretation of the original program. Moreover, since a partial evaluator is a static semantic processor [19], compile-time actions are executed, and the result solely represents dynamic operations as shown in [14]. 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 [14]. However, our experiment is based on Schism [3, 4], 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 [13]. 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 [13].

Mix [12] 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 [5]. As a result, it extends the class of applications that can be tackled by 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 discusses the partial evaluation aspects of the specification. The paper concludes with an assessment in section 5.

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. Semantically speaking, binding time analysis determines the static and the dynamic semantics of a given program. 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 sub-expressions of the program. This simplification of the partial evaluation process is crucial to self-application [13]. 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 is a partial evaluator for a side-effect free dialect of Scheme [3, 4]. 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.

The system is structured in three parts: one determining the static semantics of the program [5] (the binding time analysis); one specifying how to specialize the program [8]; one specializing the program.

It is the second phase which determines for each expression which partial evaluation *action* (*i.e.*, program transformation) will be performed during the specialization. As discussed in [8], actions can be inferred from the binding time information. These actions are the usual program transformations[1] [4] extended to handle higher order functions and operations on partially static data (*i.e.*, structured data consisting in static and dynamic parts).

2.3 Related Work

Partial evaluation of Prolog was taken up in [15]. Since then, several partial evaluators of Prolog have been developed, but mostly written in Prolog (*e.g.*, [10, 11, 21]). In [11] and [10] self-application of Prolog is addressed. The former discusses the problem of self-application and proposes some solutions. The latter describes an implementation which is said to be small

[1] We will include an overview of the partial evaluation actions of Schism in the final version of this paper.

and minimal in functionality; the results obtained are limited. None of them address the binding time analysis of Prolog.

Kahn and Carlsson describe in [14] a partial evaluator that partially evaluates 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 [9] 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).

3 Specification of Prolog

This section first gives an overview of the denotational semantics of Prolog as described in [17]. 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.

3.1.1 Abstract Syntax

The abstract syntax of Prolog, described in figure 1, is due to Clocksin and Mellish [2]. Note that, as in [17], to respect the denotational approach, 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 *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 figure 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.

I	$\in$	*Ide*	Identifiers
B	$\in$	*Con*	Constants symbols
F	$\in$	*Fun*	Function/Predicate symbols
G	$\in$	*Goals*	Goal lists
P	$\in$	*Pred*	Predicates and terms
A	$\in$	*Arg*	Argument lists
C	$\in$	*Clause*	Clause
D	$\in$	*Database*	Databases
S	$\in$	*Prog*	Sentences (or Programs)
Q	$\in$	*Input*	Queries

S	$::=$	Q, D
Q	$::=$	$:-G$
D	$::=$	C, D_1
C	$::=$	$P \mid P :- G$
G	$::=$	P, G_1
P	$::=$	$I \mid B \mid F(A)$
A	$::=$	$P, A_1 \mid P$

Figure 1: Syntactic Domains and Syntactic Rules

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 [19], 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 figure 2.

3.1.3 Valuation Functions

The meaning of a program is given by the functions defined in figure 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.

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

Figure 2: Semantic Domains

$\mathcal{S} : Prog \rightarrow Input \rightarrow Res$

$\mathcal{S}[\![D]\!][\![:-Q.]\!] = \mathcal{G}[\![Q]\!](\mathcal{D}[\![D]\!])\ printall\ (\lambda\iota.unbound) <> (\lambda v.unused)$

$\mathcal{D} : Clause \rightarrow Tv \rightarrow Kc \rightarrow Kc$

$\mathcal{D}[\![D]\!] = fixedpoint(\mathcal{C}[\![D]\!])$

$\mathcal{C} : Clause \rightarrow Db \rightarrow Tv \rightarrow Kc \rightarrow Res \rightarrow Subs \rightarrow Res$

$\mathcal{C}[\![C_1, C_2]\!]\delta\tau\kappa\phi\theta = \mathcal{C}[\![C_1]\!]\delta\tau\kappa(\mathcal{C}[\![C_2]\!]\delta\tau\kappa\phi\theta)\theta$

$\mathcal{C}[\![P.]\!] = \mathcal{P}[\![P]\!](\lambda\tau_1\rho.\ unify\ \tau\tau_1(\kappa\phi)\phi)(\lambda\iota.unbound)$

$\mathcal{C}[\![P :- G]\!] = \mathcal{P}[\![P]\!](\lambda\tau_1\rho_1.\ unify\ \tau\tau_1(\mathcal{G}[\![G]\!]\delta(\lambda\rho_2.\kappa)\rho_1\phi)\phi)(\lambda\iota.unbound)$

$\mathcal{G} : Goals \rightarrow Db \rightarrow Gc \rightarrow Env \rightarrow Res \rightarrow Qc$

$\mathcal{G}[\![P]\!]\delta\gamma\rho\phi = \mathcal{P}[\![P]\!](\lambda\tau\rho_1.\delta\tau(\gamma\rho_1)\phi)\rho$

$\mathcal{G}[\![G_1, G_2]\!]\delta\tau = \mathcal{G}[\![G_1]\!]\delta(\mathcal{G}[\![G_2]\!]\delta\gamma)$

$\mathcal{P} : Pred \rightarrow Tc \rightarrow Ec$

$\mathcal{P}[\![I]\!]\upsilon\rho = \rho I\ =\ unbound \rightarrow newvar(\lambda\tau.\upsilon\tau\rho[\tau/I]), \upsilon(\rho I)\rho$

$\mathcal{P}[\![F(A)]\!]\upsilon = \mathcal{A}[\![A]\!]\lambda\zeta.\upsilon < F, \zeta >$

$\mathcal{A} : Arg \rightarrow Ac \rightarrow Env \rightarrow Qc$

$\mathcal{A}[\![P]\!]\omega = \mathcal{P}[\![P]\!]\lambda\tau.\omega < \tau >$

$\mathcal{A}[\![P, A]\!]\omega = \mathcal{P}[\![P]\!]\lambda\tau.\mathcal{A}[\![A]\!]\lambda\zeta.\omega(\tau.\zeta)$

$Unify : Tv \rightarrow Tv \rightarrow Kc \rightarrow Res \rightarrow Subs \rightarrow Res$

Figure 3: Valuation Functions

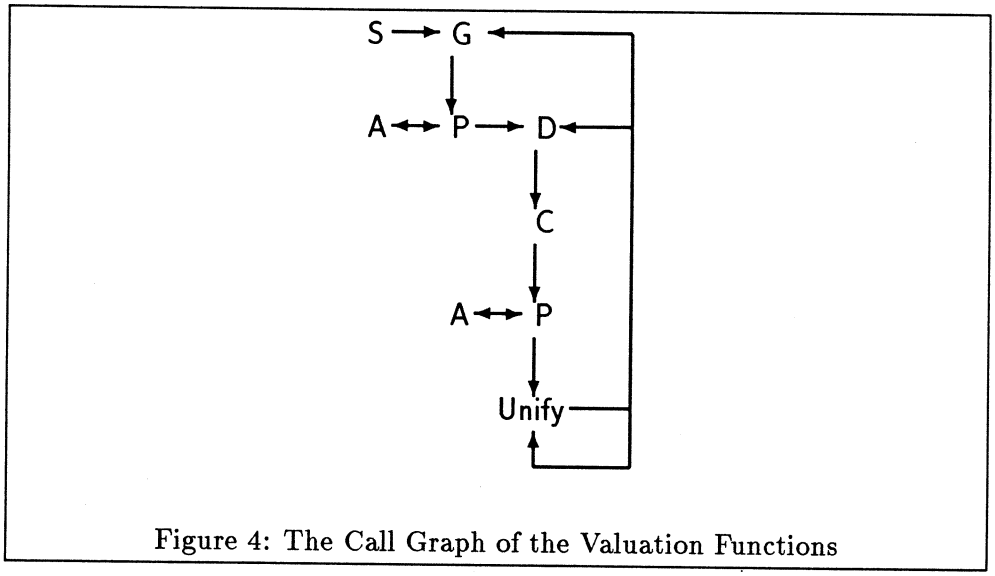

Figure 4: The Call Graph of the Valuation Functions

We defer the description of function *unify* to section 4.2. 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).

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 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 figure 5, these declarations are direct coding of the abstract syntax presented in the denotational semantics.

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


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

Figure 5: Abstract Syntax

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

Figure 6: Denotable Values

3.2.2 Data Structures of the Interpreter

The data type corresponding to the domain of denotable values is defined in figure 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 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 figures 7 and 8.

The construct `filter`, contained in some definitions, is a directive to the partial evaluator. It specifies how to treat a call to the function, *i.e.*, unfolding (`UNFOLD`) or specialization (`SPECIALIZE`). This construct addresses the issue of the termination of the partial evaluation process; it is discussed in section 4.3.

4 The Interpreter from a Partial Evaluation Point of View

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

```
(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 gc 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 (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)) ))
```

Figure 7: Interpretation Functions

```
(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))))

(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)))
```

Figure 8: Interpretation Functions (*continued*)

first input is known at partial evaluation-time, *i.e., static*, and the second is not known until run-time, *i.e., dynamic.*

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

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

To explain how the static and the dynamic semantics of the interpreter are determined, we first need to introduce the notion of *binding time signature.* The binding time analysis determines a binding time signature for each function of a program. The binding time signature specifies the binding time value of each parameter of a function and the binding time value of a result. For clarity, we simplify the binding time signatures presented in this paper. The symbols used to represent a binding time signature are *St*, *Dy*, *Cl* and *Ps*; they denote respectively the binding time value static, dynamic, closures

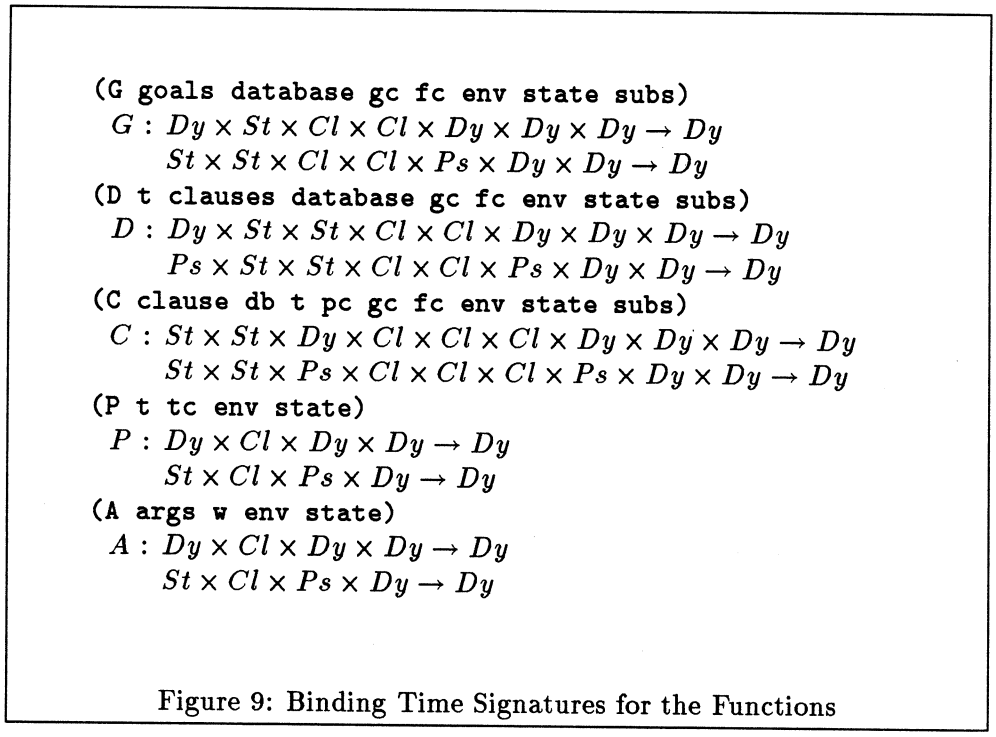

Figure 9: Binding Time Signatures for the Functions

and partially static data. The binding time value of a result of a function is assumed to be 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 the definition of function D in figure 7). Therefore, function G and the inner functions are first in a context of a dynamic query, and then, in a context of a static query.

This is illustrated in figure 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 arises because of terms consisting of static and dynamic parts in the environment which, in some context, binds static identifiers to dynamic variables.

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

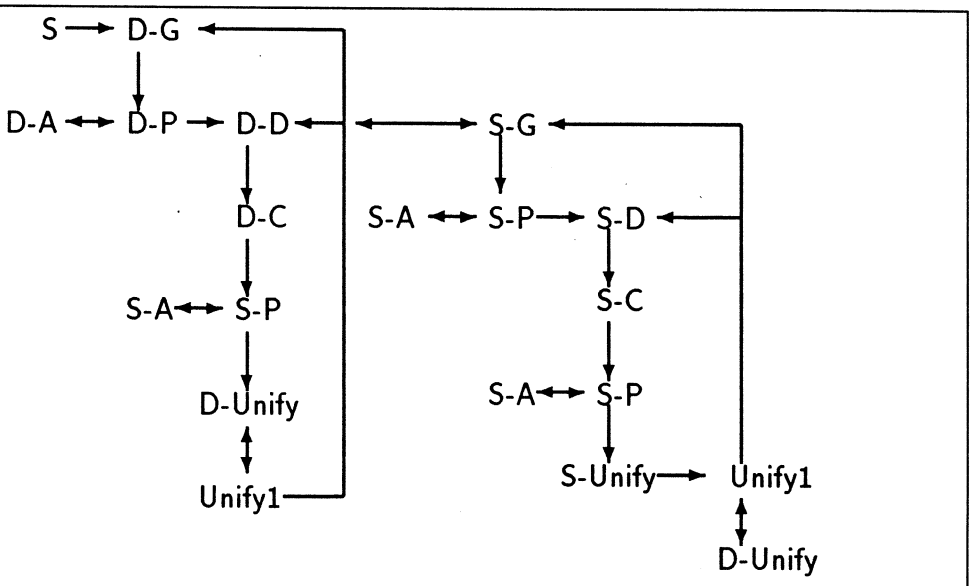

Figure 10: The Call Graph of the Interpretation Functions

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 [4] and its extension to higher order functions is discussed in [6].

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

Partial evaluating function calls has two common pitfalls: infinite call unfolding and infinite function specialization [13, 4]. Although one can introduce an automatic phase to annotate a program as to what to do for each function call, these annotations may lower the quality of the residual programs and can sometimes cause non-termination [20].

In this experiment, functions have been annotated manually as follows. As shown by figure 10, infinite call unfolding may occur for calls to recursive functions `G`, `D` and `unify1`. Therefore we instruct the partial evaluator to create specialized versions of these functions. Furthermore, infinite specialization can occur when the success continuation for `G` (`gc`) is propagated into the function body. This is avoided by preventing `gc` from being propagated.

5 Assessment

In this section we discuss what has been achieved by partial evaluation and we evaluate performances.

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 appendix A and its corresponding residual program. The compilation of the backtracking continuation can be illustrated by comparing the traversal of the database that the interpreter would performed (figure 11-a) with the traversal of the database represented by the compiled program (figure 11-b). These diagrams also include the accumulation of a result which is denoted by the symbol *res*. The labels $c1, \ldots, c4$ correspond to the clauses of the source program. Each clause in figure 11-b has attached its corresponding specialized function.

Figure 11 clearly shows that the intermediate backtracking has been eliminated in the compiled code.

Lookup Operations in the Environment are Compiled

Because the environment is a partially static data (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.

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

Table 1: Static vs. Dynamic Functions

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 has 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 [7] that compiled code is about thirteen times faster than interpretation. This difference is due to the fact that the static semantics of Prolog is not as much 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.

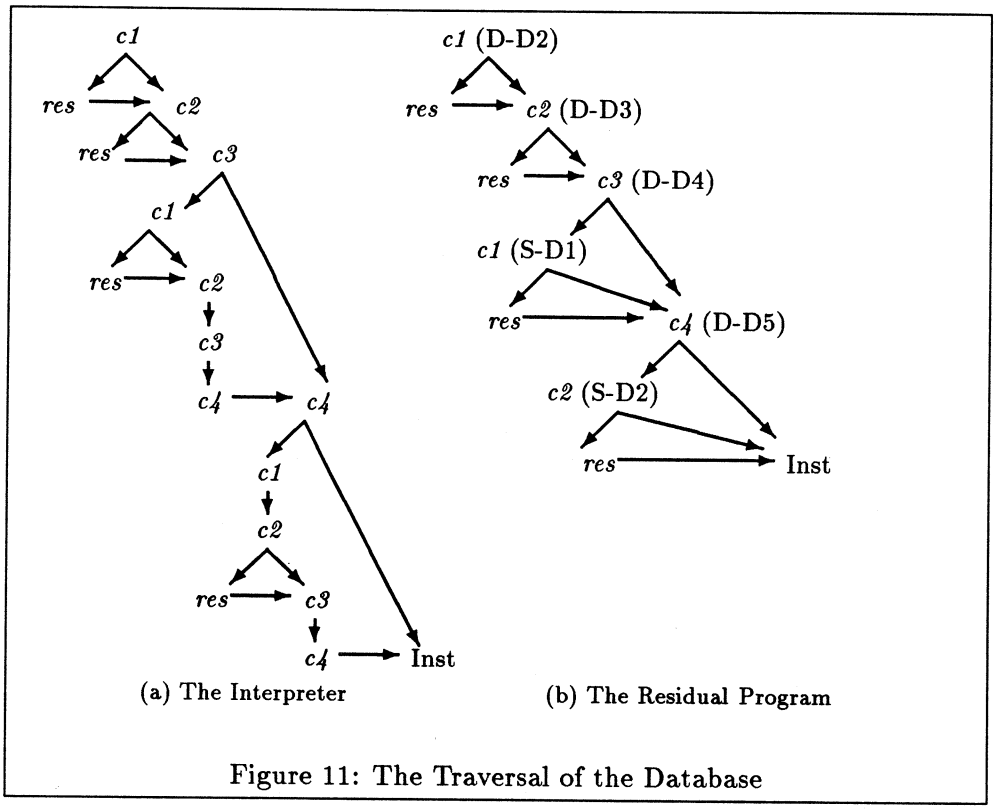

Figure 11: The Traversal of the Database

Prolog Program		*Speed-up*
Number of Facts	Number of Rules	
4	4	4
6	2	5
8	8	7

Table 2: Speed-up with Compilation

It is generally difficult to fairly evaluate the performance improvement obtained by partial evaluation and the size of the resulting programs since 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 [19]. 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.

6 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, editors. *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[2] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[3] C. Consel. New insights into partial evaluation: the Schism experiment. In H. Ganzinger, editor, *ESOP'88, 2nd European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 236–246. Springer-Verlag, 1988.

[4] C. Consel. *Analyse de Programmes, Evaluation Partielle et Generation de Compilateurs.* PhD thesis, Université de Paris VI, Paris, France, 1989.

[5] C. Consel. Binding time analysis for higher order untyped functional languages. Technical report, Yale University, 1989. To appear in the proceedings of the 1990 ACM Lisp and Functional Programming.

[6] C. Consel and Khoo S. C. Higher order polyvariant analysis, 1990. Paper in preparation.

[7] C. Consel and O. Danvy. Static and dynamic semantics processing. Research Report 761, Yale University, New Haven, Connecticut, USA, 1989.

[8] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[9] M. Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University, Bloomington, Indiana, 1985.

[10] H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compiler. In Y. Futamura, editor, *New Generation Computing*, volume 6 of *2,3*. OHMSHA. LTD. and Springer-Verlag, 1988.

[11] D. A. Fuller and S. Abramsky. Mixed computation of Prolog. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[12] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

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

[14] K. M. Kahn and M. Carlsson. The compilation of Prolog programs without the use of Prolog compiler. In *International Conference on Fifth Generation Computer Systems*, pages 348–355, 1984.

[15] H. J. Komorowski. A specification of an abstract Prolog machine and its application to partial evaluation. Linkoping studies in science and technology dissertations n° 69, Linkoping University, Linkoping, Sweden, 1981.

[16] P. Lee and U. F. Pleban. On the use of Lisp in implementing denotational semantics. In *ACM Conference on Lisp and Functional Programming, Cambridge, Massachusetts*, pages 233–248, 1986.

[17] T. Nicholson and N. Foo. A denotational semantics for Prolog. *ACM Transaction on Programming Languages and Systems*, 11(4), 1989.

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

[19] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[20] P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[21] R. Venken. A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query-optimisation. In T. O'Shea, editor, *ECAI'84*. North-Holland, 1988.

[22] M. Wand. A semantic prototyping system. *SIGPLAN Notices, ACM Symposium on Compiler Construction*, 19(6), 1984.

A Prolog Program and Its Residual

A Source Program :

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

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))
```

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))))
```

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))
```