

Static and Dynamic Semantics Processing *

Charles Consel

Department of Computer Science
Yale University †
(consel@cs.yale.edu)

Olivier Danvy

Department of Computing and Information Sciences
Kansas State University ‡
(danvy@cis.ksu.edu)

Abstract

This paper presents a step forward in the use of partial evaluation for interpreting and compiling programs, as well as for automatically generating a compiler from denotational definitions of programming languages.

We determine the static and dynamic semantics of a programming language, reduce the expressions representing the static semantics, and generate object code by instantiating the expressions representing the dynamic semantics. By processing the static semantics of the language, programs get compiled. By processing the static semantics of the partial evaluator, compilers are generated. The correctness of a compiler is guaranteed by the correctness of both the executable specification and our partial evaluator.

The results reported in this paper improve on previous work in the domain of compiler generation [16, 30], and solves several open problems in the domain of partial evaluation [15]. In essence:

- Our compilation goes beyond a mere syntax-to-semantics mapping since the static semantics gets processed at compile time by partial evaluation. It is really a syntax-to-dynamic-semantics mapping.
- Because our partial evaluator is self-applicable, a compiler is actually generated.
- Our partial evaluator handles non-flat binding time domains. Our source programs are enriched with an open-ended set of algebraic operators.

This experiment parallels the one reported by Montenyohl and Wand in [24]: starting with the same denotational semantics of an Algol subset, we obtain the same good results, but completely automatically and using the original semantics only, instead of writing several others, which requires proving their congruence. We are able to compile strongly typed, Algol-like programs and to process their static semantics at compile time (scope resolution, storage calculation, and type checking), as well as to generate the corresponding compiler completely automatically.

* To appear in the proceedings of POPL'91.

† P.O. Box 2158, New Haven, CT 06520, USA. This research was supported by Darpa under grant N00014-88-k-0573.

‡ Manhattan, KS 66506, USA

The object code is reasonably efficient. It has been found to be more than a hundred times faster than the interpreted source program, whereas it runs about two times slower than the output of a regular, production-quality compiler. The compiler is well-structured and efficient. The static semantics still gets processed at compile time. Self-application pays off: compiling using the compiler is correspondingly much faster than compiling by partially evaluating the interpreter with respect to the source program. Our compiler is still slower than a production-quality one, but we derived it automatically.

Keywords

Compiler generation, semantics, partial evaluation, self-application, binding times, Algol, Scheme

Introduction

Existing semantics-directed compiler generators essentially amount to a syntax-to-semantics mapping [16, 35, 30]. They map the representation of programs as abstract syntax trees into the representation of their meaning as lambda-expressions. Lacking a static/dynamic distinction in the semantic specification, it is not clear how to simplify the resulting lambda-expressions [26]. For this reason compile time actions are likely to be performed at run time, thereby impeding the performances of the whole system. This specific problem gets solved by distinguishing static and dynamic semantics soundly in semantic specifications [28].

As a static semantics processor, a self-applicable partial evaluator is an obvious choice. It ensures the static semantics of a program to be processed at compile time. Viewing the set of valuation functions as a definitional interpreter [32], compiling a program is achieved by specializing its interpreter. Further, generating a compiler is achieved by specializing the partial evaluator with respect to the interpreter [1].

Partial evaluation provides a great deal of flexibility. Executable specifications can be experimented with. Compiling can be experimented with by specializing the same executable specification. A stand-alone compiler can be generated by self-application with respect to the same executable specification. However, the partial evaluator needs to be sufficiently powerful.

We have solved a series of open problems in partial evaluation [15]. Our partial evaluator specializes higher order Scheme programs [31] with an open-ended set of algebraic operators and non-flat binding time domains (*i.e.*, partially static structures). Also, it can specialize itself and thus generate compilers automatically. Its source language has the

```

block                               (letrec ([evProgram1 (lambda (s)
{ n int 5; r int 1;}                (loop2 (intUpdate 1 5 (intUpdate 0 1 s))))])
{ while n > 0                        [loop2 (lambda (s)
do                                  (if (gtInt (fetchInt 1 s) 0)
r := n * r;                         (let ([s1 (intUpdate 0 (mulInt (fetchInt 1 s) (fetchInt 0 s)) s)])
n := n - 1;                          (loop2 (intUpdate 1 (subInt (fetchInt 1 s1) 1) s1)))
od; }                                (initCcont s))))])
end                                  evProgram1)

```

Figure 1: Source and object code of the factorial program.

same expressive power as the meta-language of denotational semantics. This makes it possible to match the requirements that have been found necessary in semantics-directed compiler generation [30].

The present paper illustrates this step forward with the compilation of Algol-like programs and the automatic derivation of a stand-alone compiler from an executable specification of this language.

Our experiment parallels the one reported by Montenyohl and Wand in [24], where a compiler is derived by hand, which necessitates (1) to introduce three denotational specifications and prove their congruence to make it possible to process the static semantics and (2) to introduce combinators and use the compiling algorithm of [38] to generate object code.

In contrast, self-applicable partial evaluation offers a unified framework for semantics-directed compiler generation. The static and dynamic semantics are determined by analyzing the binding times [18] of the executable specification. Compile time and run time combinators are extracted automatically, based on the binding time information [9]. The compiling algorithm is provided by the partial evaluator.

Two reasons motivate Montenyohl and Wand's subset of Algol. It is small enough for its complete description to fit in a paper, and yet significant enough to highlight the effectiveness of partial evaluation. It is precisely the same as in [24] and thus our treatment can be compared directly. We have processed a superset of this language that includes procedures, but though the results are comparable they fall out of the scope of this paper.

Compiling includes reducing the expressions representing the static semantics and therefore goes beyond a mere syntax-semantics mapping. Our compiler is stand-alone and therefore optimizes the compilation process. The whole static semantics of Algol is processed at compile time: syntax analysis, scope resolution, storage calculation, and type checking.

We compile Algol programs into low level, tail-recursive Scheme code, with explicit store and properly typed operators. Figure 1 displays the source and compiled code of the factorial program. The Algol source program is written with a while loop and an accumulator. The target program is a specialized version of the interpreter with respect to the source program. It is written in Scheme because the interpreter is written in Scheme. It computes the factorial of 5 because the source program computes the factorial of 5. The main procedure is passed a store and updates it during the computation. The while loop has been mapped to a tail-recursive procedure iterating on the store. All the continuations of the original continuation semantics except the initial one have disappeared: the target program is tail-

recursive and in direct style. All the locations have been computed at compile time (variables *r* and *n* at locations 0 and 1, respectively). There is no type-checking at run time: all the injection tags have disappeared and all the operators are properly typed.

Essentially we obtain a front-end compiler that maps syntax to a lambda-expression representing the dynamic semantics. Representing this lambda-expression with assembly language instructions is out of the scope of this paper, but is naturally achieved by program transformation [21].

Let us evaluate our approach by comparing interpreted and compiled code, and then compiled code with object programs generated by a production quality compiler:

- Running the compiled code can be more than a hundred times faster than interpreting the source code. Compiling an Algol program using the stand-alone compiler is correspondingly faster than compiling by specializing the definitional interpreter of Algol with respect to the Algol program.
- Running a compiled program (using the T system [22]) is only two times slower than running the output of a Pascal or a C compiler.

The rest of this paper is organized as follows. Section 1 describes the background of this work and compares its results with related works. Section 2 presents an overview of the source language specification. Section 3 describes how a specification is analyzed and its static and dynamic semantics are separated. Section 4 describes how the binding time information is processed. Section 5 describes the generation of a compiler. Finally this work is put into perspective.

1 Background and Related Works

Section 1.1 addresses the implementation of formal semantics. Section 1.2 covers partial evaluation. Section 1.3 draws a synthesis.

1.1 Implementation of formal semantics

This section addresses semantics-directed interpretation, compilation, and compiler generation.

Semantics-directed interpretation

Stoy asserted it firmly: denotational semantics specifications may have the format of a program, but programs they are not — they are mathematical objects [37, pages 179-181]. This issue is getting blurred progressively as programming

languages are getting closer to the meta-languages of formal semantics. Today, transliterating formal specifications into executable programs is a matter of routine. For example, denotational semantics definitions can be mapped into executable specifications by transliterating their valuation functions into functional programs that act as definitional interpreters [32] and can be experimented with.

Next step aims at getting rid of their interpretive overhead. This is achieved by considering them as data objects [14].

Semantics-directed compilation

A definitional interpreter serves two distinct purposes but they are interleaved. On the one hand programs need to be syntactically analyzed. On the other hand the operations they specify need to be performed. Compiling a program based on its formal semantics amounts to analyzing its syntax and to producing a representation of the operations the program specifies. In other terms, compiling a program amounts to dividing the representation of its formal meaning into compile time and run time forms, and to reducing the former while emitting a representation of the latter as object code [26].

It is only natural to compile programs with a program [16]. Next section describes how to derive such a compiler from the semantics of a programming language.

Semantics-directed compiler generation

A naive semantics-based compiler is organized in several phases: one mapping source programs to a representation of their meaning; another simplifying this representation; and a last one mapping this representation to object code [30]. This strategy requires to separate compile time and run time forms within the meaning of each source program.

This separation can be foreseen in the semantics of the programming language instead of in the semantics of each source program. Then programs can be mapped directly to their run time meaning. This contrasts with simplifying expressions that need to be generated first. The separation is achieved by analyzing the binding times of expressions in the representation of the semantic specification [18, 27].

Intuitively, this separation makes sense. For example, the first compiler derived from an interpreter using Peter Landin's Applicative Expressions was based on the results of an implicit binding time analysis [25]. Today the TML approach is based on an explicit binding time analysis [28].

State of the art

From the point of view of partial evaluation, which in essence processes static semantics [35], existing semantics-directed compiler generators suffer from the same problem of not processing the static semantics for fear of looping [26]. The call-by-need strategy of Paulson's compiler generator [29] often delays compile time computations until run time, which clearly is unsatisfactory. The toolbox approach illustrated by SPS [39] stresses the problem of mere compositions of tools: they make compiler compilers without actual speedups. What is expected from a compiler is that it processes the static semantics of a language. Yet as sound as it is and despite its binding time analysis, PSI [28] does not unfold static fixpoints and has no partially static data. In contrast, the micro and macro semantics of MESS [23]

characterize the static and the dynamic semantics of a language and ensure the static semantics to be processed at compile time, even though distinguishing between micro and macro semantics relies on the initiative of the user, who also needs to separate static and dynamic representations into completely static and completely dynamic structures. More recently, Wand and Wang introduce a log recording static information about earlier static reductions [40]. The corresponding change in the semantics (type information located in the log instead of the run time store) can be propagated directly using partial evaluation.

Semantics-directed compiler generation systems share the same goal, and ultimately use the same methods. The trends, as analyzed in Uwe Pleban's POPL'87 tutorial [30], are to use semantic algebras instead of λ -terms; to improve the software engineering of systems; to manage a tradeoff between generality and efficiency; and to choose existing functional languages as specification languages. Since this tutorial, the open problem of including specifications of flow analyses and optimization transformations has been solved [28].

On the other hand, the very format of denotational semantics may be criticized [33]. This is not our point here. We want to illustrate progress in partial evaluation with the classical example of compiling and compiler generation.

1.2 Partial evaluation

A thorough overview of partial evaluation can be found in [1, 12]. This section addresses its extensional aspects, including self-application, and compiling and compiler generation by partial evaluation.

Extensional aspects

Let P be a dyadic program computing a binary function p and let PE be a partial evaluator computing Kleene's specialization function S_1^1 . A version of P specialized with respect to its first argument can be derived as follows:

$$S_1^1 (P, \langle v, _ \rangle) \quad (1)$$

The first argument of S_1^1 is the source program P . The second argument of S_1^1 represents the arguments of P . The first argument of P is known to be v . The second argument of P is unknown.

Let P_v be the specialized version of P with respect to v and let p_v be the function computed by P_v . By definition of partial evaluation, applying p to the two input values v and w yields the same result as applying p_v to w :

$$p (v, w) \equiv p_v (w) \quad (2)$$

Partial evaluation differs from currying because it is a program transformation and we cannot confuse a program with the function this program computes. Using a processor and representing it as the function 'run', (1) and (2) become

$$\text{run } PE \langle P, \langle v, _ \rangle \rangle \quad (3)$$

and

$$\text{run } P \langle v, w \rangle \equiv \text{run } P_v \langle w \rangle \quad (4)$$

$$\text{run } INT \langle P, I \rangle \equiv \text{run } INT_P \langle I \rangle$$

The essential motivation for using PE is to optimize the run time of a program if part of its input is known beforehand or somehow is invariant. Partial evaluation can be optimized as follows.

Suppose program P needs to be specialized repeatedly with respect to several values v . In this case, program PE will be run repeatedly with an invariant first argument (P) and a variable second argument ($\langle v, _ \rangle$). Kleene's S_n^m theorem tells us that we can build a version PE_P of PE specialized with respect to P and run PE_P instead.

$$\text{run } PE \langle PE, \langle P, _ \rangle \rangle = PE_P \quad (5)$$

based on the identity

$$\text{run } PE \langle P, \langle v, _ \rangle \rangle \equiv \text{run } PE_P \langle v, _ \rangle \quad (6)$$

PE_P is a specializer dedicated to P . Whereas PE can specialize any program with respect to part of its input, PE_P can only specialize P with respect to part of its input.

PE_P has been obtained by self-application.

Similarly, because, in (5), P is the variable and PE is invariant, we can derive a generator of dedicated specializers by specializing the partial evaluator with respect to itself.

State of the art

Before Mix Due to its generality, partial evaluation has been largely investigated. Unfortunately partial evaluators very quickly were running into complexity barriers and were loop-prone [1, 12].

Mix The MIX project aimed at realizing a self-applicable partial evaluator based on simplicity and reasonable automatism [19, 20]. MIX handles first-order Lisp-type recursive equations with a fixed set of symbolic operators. As one of the assets of the MIX project, binding time analysis has been discovered to be an essential component for realistic self-application [4].

Since Mix On the side of functional programming, the activity has been concentrated on strengthening self-applicable specializers. Building on top of them [5, 6, 3], the barrier of higher-orderness has been torn down in 1989 [17, 2, 7].

Further progress has led us to the results reported here. [7] presents a unified treatment of non-flat binding time domains, *i.e.*, of partially static structures in function spaces, products and co-products. [9] enhances the actual treatment of the static and dynamic semantics.

Compiling by partial evaluation

Let INT be an interpreter for the language L . INT is written in B and PE is a partial evaluator for B programs. Interpreting a L -program P on some input I is achieved by running INT as follows

$$\text{run } INT \langle P, I \rangle$$

Because P may be run on several input values, we can use partial evaluation to build a specialized version of the interpreter tailored to P :

$$\text{run } PE \langle INT, \langle P, _ \rangle \rangle = INT_P$$

under the definitional condition

As can be observed, P is a source program written in L and INT_P is a residual program written in B . In other terms, P has been compiled from L to B .

Specializing an interpreter with respect to a program is a particular application of partial evaluation. It is known as the first Futamura projection [13].

Compiler generation by partial evaluation

Correspondingly, we can build a specializer dedicated to INT :

$$\text{run } PE \langle PE, \langle INT, _ \rangle \rangle = PE_{INT}$$

under the definitional condition

$$\text{run } PE \langle INT, \langle P, _ \rangle \rangle \equiv \text{run } PE_{INT} \langle P, \langle P, _ \rangle \rangle$$

PE_{INT} has the functionality of a compiler since it maps a source program into object code. Finally, the generator of dedicated specializers PE_{PE} has the functionality of a compiler generator, in the particular case of interpreters.

These two applications of partial evaluation are known as the second and third Futamura projections [13].

1.3 Synthesis

Higher order programming constructs match the expressive power needed for in semantics-directed compiler generation (intensional reason). Partial evaluation captures semantics-directed compiler generation from interpretive specifications (extensional reason). Therefore it makes sense to use the new generation of partial evaluators to solve problems like the turning of non-trivial interpreters into realistic compilers. To this end, we use the self-applicable partial evaluator Schism [5, 6, 7].

2 Representation of Formal Definitions

As for semantics-directed compiler generation systems, the expressiveness of the input language in Schism mainly determines the class of languages that can be described. This section presents an overview of this specification language and how closely it matches the meta-language of denotational semantics. We illustrate it with excerpts of the specification of Montenyohl and Wand's Algol subset [24].

2.1 The specification language

Schism specializes programs written in a pure dialect of Scheme [31]. This specification language has the same expressive power as the usual meta-language of denotational semantics [37, 35]. In particular, it is higher order, which makes it possible to handle *e.g.*, continuation semantics. Actually since Montenyohl and Wand's is a continuation semantics, our transliteration is evaluation-order independent [32] and thus is not tailored to run only in Scheme.

Further, we can specify data abstractions by defining data types algebraically, based on the following rationale.

A source program presumably includes primitive operations that are defined in the underlying machine (here, the T system). In a language specification, primitive operations are used to abstract certain low level semantic details, *e.g.*,

<pre> (Program) ::= (Block) (Block) ::= block { (DeclList) } { (StmtList) } end (DeclList) ::= empty (Declaration) ; (DeclList) (Declaration) ::= (Ident) (Expr) (StmtList) ::= empty (Stmt) ; (StmtList) (Stmt) ::= (Block) (Ident) := (Expr) while (Expr) do (StmtList) od if (Expr) then (Stmt) else (Stmt) (Expr) ::= (Constant) (Ident) (Expr) (AritBinop) (Expr) (Expr) (RelBinop) (Expr) (Constant) ::= int (Int) real (Real) bool (Bool) (AritBinop) ::= + - / * (RelBinop) ::= < > = </pre>	<pre> (defineType Program block) (defineType Declaration ident expr) (defineType Statement (Block declList stmtList) (While expr stmtList) (If expr stmt1 stmt2) (Assign ident expr)) (defineType Expr (Int value) (Real value) (Bool value) (Identifier ident) (AritBinop op expr1 expr2) (RelBinop op expr1 expr2)) </pre>
--	---

Figure 2: The abstract syntax and its concrete specification (sample).

the actual representation of the store. As a result, different implementation strategies can be explored within the same semantic framework. Furthermore, properties of the language definition can allow efficient implementations of primitives in the underlying machine. For an outstanding example, primitive operations modeling a store algebra can be implemented imperatively given a denotational specification which is single-threaded in its store argument [34].

2.2 Specification of the Algol subset

This section presents fragments of the Algol definition. It is essentially a direct transliteration of Montenyohl and Wand’s continuation semantics. The rest of the definition is displayed in appendix A.

We first illustrate the data type facility by defining the abstract syntax. Together with the concrete syntax, this definition is displayed in figure 2.

One could include the definition of the store operations in the specification. Every compiled program would then be dedicated to a fixed implementation. This implementation would be functional since our specification language is purely functional. Alternatively, one can abstract the implementation of the store from the language definition by declaring the store operations to be primitive. Like most other imperative sequential languages, this definition of Algol is single-threaded in its store argument. Therefore, as outlined in section 2.1, we can implement the store imperatively and thus improve the run time of compiled programs.

As another illustration of our metalanguage, the function evaluating statements is displayed in figure 3. Its type is presented in figure 4.

Notice that the valuation functions are uncurried, for the sake of readability in Scheme.

3 Separating the Static and the Dynamic Semantics

Traditionally, to derive a compiler from a semantic definition, one first has to determine its static semantics. Then, the soundness of the static semantics is proved manually. This process is generally agreed to be very difficult and error prone [20, 30].

This section first presents an analysis that determines the static and the dynamic semantics of a language specification automatically. Then, we illustrate the approach by

giving examples of information yielded by this analysis when applied to the Algol definition.

3.1 Binding time analysis

Our approach consists in automatically analyzing the semantic definition to split it into two parts: the static semantics (the usual compile time actions) and the dynamic semantics. This phase is a *binding time analysis* [4, 7, 18, 27].

The binding time analysis of Schism processes higher order functional programs. For each function, this phase yields a *binding time signature* that describes the binding time properties of function parameters and the binding time properties of results. The analysis determines the binding time descriptions of structured data made of both static and dynamic parts (or recursively of partially static data). As a consequence, the usual restriction in denotational semantics for separating static and dynamic elements of a specification can be relaxed without loss of staticness. This capability is illustrated below with a disjoint sum defining the domain of expressible values. In the representation of the elements of this domain, and as determined by the binding time analysis, the injection tag is static and the value is dynamic.

3.2 Static and dynamic semantics of the Algol subset

This section presents examples of binding time descriptions yielded by the binding time analysis for the definition of the Algol subset. Let us annotate the domains and the type constructors as follows: they are overlined or underlined depending on how they account for static or dynamic computations, respectively. Further, constructors are accented with a tilde to denote partially static data. For simplicity, we have not annotated continuations, though they are a good example of partially static, higher order values.

Figure 4 displays semantics domains of our language definition and their binding time properties. By analogy with the traditional approach, let us explain these results by considering the binding time analysis as a theorem prover: for each syntactic construct, a set of inference rules is defined to infer the static properties; as starting axioms, the program is static (available at compile time) and the store is dynamic (not available until run time).

Property 1 *Every valuation function is static in its program argument.*

```

(define (evStmt stmt r k s)
  (caseType stmt
    [(Block declList stmtList)
     (evBlock declList stmtList r k s)]
    [(While expr stmtList)
     ((fix (lambda (loop)
            (lambda (s)
              (evExpr expr r (lambda (v)
                (caseType v
                  [(Int -) (terminate 'error7 k s)]
                  [(Real -) (terminate 'error8 k s)]
                  [(Bool b) (if b (evStmtList stmtList r loop s) (k s)))))) s))]
      [(If expr stmt1 stmt2)
       (evExpr expr r (lambda (v)
         (caseType v
           [(Bool b) (if b (evStmt stmt1 r k s) (evStmt stmt2 r k s))]
           [else (terminate 'error1 k s)])) s))]
    [(Assign ident expr)
     (evExpr expr r (lambda (v)
       (caseType (locIdent ident r)
         [(IntLoc l) (caseType v
           [(Int n) (let ([s1 (intUpdate l n s)]) (k s1))]
           [(Real -) (terminate 'error2 k s)]
           [(Bool -) (terminate 'error3 k s)]))]
         [(RealLoc l) (caseType v
           [(Int n) (let ([s1 (realUpdate l (coerceToReal n) s)]) (k s1))]
           [(Real n) (let ([s1 (realUpdate l n s)]) (k s1))]
           [(Bool -) (terminate 'error4 k s)]))]
         [(BoolLoc l) (caseType v
           [(Int -) (terminate 'error5 k s)]
           [(Real -) (terminate 'error6 k s)]
           [(Bool b) (let ([s1 (boolUpdate l b s)]) (k s1))])))) s))))

(define (evBlock declList stmtList r k s)
  (let ([([list r s] (makeDecl declList r s))]
        (evStmtList stmtList r k s)))

(define (evStmtList stmtList r k s)
  (if (empty? stmtList)
      (k s)
      (evStmt (head stmtList) r (lambda (s)
        (evStmtList (tail stmtList) r k s)) s)))

```

Figure 3: Valuation functions for the statements.

Initially the program is static. Because the semantic definition respects the denotational assumption [35], *i.e.*, it is compositional, the meaning of a sentence is solely defined in terms of its proper subparts. The original motivation for the denotational assumption was to enable structural induction over abstract syntax trees. Presently, compositionality implies staticness: no abstract syntax tree is ever built. Furthermore, because the arguments of the valuation functions are used consistently, *i.e.*, a variable is uniquely bound to elements of the same domain, no dynamic arguments can interfere with a program argument. Therefore, the staticness of a program argument is guaranteed in every valuation function. \square

Property 1 is at the basis of compiling by partial evaluation. It has numerous consequences.

Property 2 *Storage calculation and location types are completely static.*

All the identifiers of the program are static since they are part of the program (by property 1). Because storage calculation only depends on identifiers, this operation is completely static (*cf.* function `locIdent` in figure 4), and therefore will be performed at compile time. \square

Property 3 *In the representation of an expressible value, the injection tag is static.*

Let us consider the domain *Evalue*, defined as a disjoint sum of basic values. As such, this sum is used to perform type checking. Operationally, this domain is implemented as a cartesian product. Its elements hold the injection tag and the actual value. The injection tag is static because it is induced by the program text (static by property 1) or the location type (static by property 2). As a remarkable consequence, all the operations depending on the injection tag, *i.e.*, type checking, will be performed at compile time. \square

$$\begin{aligned}
v \in \widetilde{Evaluate} &= \overline{\underline{Int} \mp \underline{Real} \mp \underline{Bool}} \\
l \in \overline{Location} &= \overline{\underline{IntLoc} \mp \underline{RealLoc} \mp \underline{BoolLoc}} \\
r \in \overline{Environment} &= \overline{\underline{FreeLoc} \times \underline{Mappings}} \\
k \in \overline{Ccont} &= \overline{\underline{Store} \rightrightarrows \underline{Store}} \\
e \in \overline{Econt} &= \overline{\underline{Evaluate} \rightrightarrows \underline{Store}}
\end{aligned}$$

$$\begin{aligned}
\text{evProgram} &: \overline{\underline{Program} \times \underline{Store} \rightrightarrows \underline{Store}} \\
\text{evBlock} &: \overline{\underline{Block} \times \underline{Env} \times \underline{Ccont} \times \underline{Store} \rightrightarrows \underline{Store}} \\
\text{makeDecl} &: \overline{\underline{DeclList} \times \underline{Env} \times \underline{Store} \rightrightarrows \underline{Env} \times \underline{Store}} \\
\text{evStmtList} &: \overline{\underline{StmtList} \times \underline{Env} \times \underline{Ccont} \times \underline{Store} \rightrightarrows \underline{Store}} \\
\text{evStmt} &: \overline{\underline{Stmt} \times \underline{Env} \times \underline{Ccont} \times \underline{Store} \rightrightarrows \underline{Store}} \\
\text{evExpr} &: \overline{\underline{Expr} \times \underline{Env} \times \underline{Econt} \times \underline{Store} \rightrightarrows \underline{Store}} \\
\text{locIdent} &: \overline{\underline{Ident} \times \underline{Env} \rightarrow \underline{Location}}
\end{aligned}$$

For simplicity, we have left out lifting domains, *etc.* that account for errors.

Figure 4: Annotated semantic domains.

Once static properties of language definitions have been determined automatically, we can perform static and dynamic processing. In addition to the present application, static properties are useful both from a language design and from an implementation point of view: they give precise and safe bases to reason about this language.

4 Processing the Static and Dynamic Information

The static and dynamic properties of a language definition determine what to process at compile time and at run time. This section focuses on how the information provided by the binding time analysis is processed to determine compile time actions. Section 4.2 presents the treatment of binding time information in Schism. Section 4.3 describes the result of processing the binding time information about Algol.

4.1 Principles

Classical, MIX-type partial evaluators specialize source programs based on interpreting the binding time information. In Schism, this information is compiled into elementary specialization actions (*i.e.*, basic program transformations) to perform for each source expression during specialization. This simplifies and improves the specialization phase considerably, as introduced and discussed in [9]. Indeed, the binding time information of a given expression does not have to be interpreted repeatedly to determine which elementary action to perform.

4.2 Processing the binding time information in Schism

The main partial evaluation actions denote the following treatments.

- Reduction — *Red*: the outermost syntactic construct can be reduced. For example, a conditional expression whose test yields a value statically reduces to its consequent or its alternative.

- Standard evaluation — *Ev*: because the expression only manipulates available data it can be reduced to a value statically. *Ev* refines *Red*.
- Rebuilding — *Reb*: the outermost syntactic construct has to be rebuilt, but sub-components need to be partially evaluated. For example, a primitive operation that does not yield a partially static data has to be reconstructed if one of its arguments does not yield a value statically.
- Reproduction of the expression verbatim — *Id*: no data are available. For example, a primitive operation that does not yield a partially static data and whose arguments are reproduced verbatim can be reproduced verbatim. *Id* refines *Reb*.

These actions are defined for each syntactic construct of the input language. Those described above capture the usual program transformations for partial evaluation of first order functional programs as described in [6, 36]. This set of actions is extended in [8] to handle higher order functions and structured data.

Actions can be exploited further for extracting two sets of combinators representing the *purely static* and *purely dynamic* semantics of a language definition [9]. An *Ev*-combinator is extracted from an expression solely annotated with *Ev*, and an *Id*-combinator is extracted from an expression solely annotated with *Id*. These combinators actually establish an instruction set to specialize a source program — with the *Ev*-combinators; and an instruction set to execute a specialized program — with the *Id*-combinators.

At this stage, we have reduced specialization to executing the partial evaluation actions. The specializer is implemented as a simple processor for these actions.

4.3 Processing the static and dynamic information about Algol

In the case of Algol, processing the specialization actions amounts to compiling Algol programs since static, compile time operations get executed and a representation of the dynamic, run time meaning of programs gets constructed. The *Ev*-combinators define an instruction set for compiling and the *Id*-combinators define the instruction set of a run time machine for compiled programs.

In the present specification, an *Ev*-combinator will perform compile time storage calculation and some *Id*-combinators will perform run time storage management. For example, looking up in the environment ends up in the instruction set of the compiler because it has been established to be a static operation.

In the core compiler, the case dispatch on an abstract syntax node has been established to be static and therefore reduces to one of its consequent branches statically. Symmetrically, accessing the store has been established to be a dynamic operation, but its offset can be determined statically.

4.4 Synthesis

Separating the static and dynamic semantics of partial evaluation appears to be particularly convenient for compiling programs. It has also been found to be crucial for generating compilers by self-application, not only for generating a compiler and but also for running a generated compiler [20].

With respect to simplicity, orthogonality, and efficiency, our treatment as given in this section improves earlier results. For example, extracting combinators usually reduces the size of source programs sharply, yielding smaller programs that are faster to specialize and, correspondingly, smaller and faster compilers [9]. This observation has been confirmed again in the present Algol experiment.

The object code obtained by compiling an Algol program represents the same dynamic semantics as the one in [24] since we have identified and processed the same static semantics. Its correctness depends only on the correctness of the denotational specification of the language and the correctness of the partial evaluation technique.

5 Generating a Compiler

Generating a compiler is achieved by specializing the partial evaluator with respect to the executable specification:

$$\text{run } PE \langle PE, \langle INT, _ \rangle \rangle = PE_{INT}$$

This is realized straightforwardly because our partial evaluator is self-applicable. What we specialize is the processor for partial evaluation actions presented in section 4. We specialize it with respect to the preprocessed executable specification, encoded with partial evaluation actions. As listed below, deriving a compiler follows the standard steps of specialization.

Representing the semantics of partial evaluation

This step is trivial, since we already have the program PE . As in section 2, we also need to represent the semantics of the programming language as a definitional interpreter.

Separating the static and dynamic semantics

Next step consists in determining the static and dynamic semantics of partial evaluation. This is achieved by analyzing the binding times of the partial evaluator, knowing it to be specialized with respect to its program argument. As in section 3, we also need to analyze the binding times of the definitional interpreter.

Processing the static and dynamic information

Next step amounts to processing the information accumulated by the binding time analysis to determine the static actions, *i.e.*, the actions to be executed at compiler generation time. As in section 4, we also need to process the binding time information about the definitional interpreter.

Given the actions to execute in the partial evaluator and the actions to execute in the interpreter, executing the former on the latter generates a specializer dedicated to the interpreter. In essence, all the static semantics of the specializer is processed at compiler generation time, yielding a residual program dedicated to processing the static semantics of the language and emitting a representation of its dynamic semantics — in other terms: a stand-alone compiler whose instruction set is defined by the *Ev*-combinators of the language definition.

Generating a compiler generator

Based on the equation

$$\text{run } PE \langle PE, \langle PE, _ \rangle \rangle = PE_{PE}$$

and following the steps above based the knowledge that the inner specializers will be passed their program argument [4], a generator of dedicated specializers can be built. Its instruction set is defined by the *Ev*-combinators of the partial evaluator. Using this compiler generator requires one to represent and analyze the binding times, *etc.* of a programming language specification as described above.

Synthesis

Beyond its theoretical interest and its conceptual elegance, self-application pays off: compiling using the compiler is considerably faster than compiling by specializing the executable specification. On the other hand, it is still much slower than a conventional, production quality compiler, which is not produced automatically out of an interpreter. This was expectable because our syntax-to-dynamic-semantics mapping has to be composed with a Scheme compiler.

The object code obtained with our Algol compiler represents the same dynamic semantics as the one in [24]. Its correctness depends only on the correctness of the denotational specification and the correctness of the partial evaluator.

Conclusion and Issues

The general mechanism of compilation and compiler generation — reducing expressions representing the static semantics and emitting residual expressions representing the dynamic semantics — was captured extensionally in the definition of a partial evaluator: partially evaluating an interpreter with respect to a program amounts to compiling this program. Previous experiences in semantics-directed compiler generation established the need for modularity, for algebras, for handling higher order constructs, *etc.* in source specifications.

This paper illustrates the intensional use of partial evaluation where source programs are specified modularly and algebraically with non-flat binding time domains. We consider an executable specification of a block-structured, strongly typed language. We automatically derive a compiler where all the static semantics — scope resolution, storage calculation, type checking, *etc.* — is reduced at compile time. The compiler inherits the structure of the partial evaluator, and the object programs inherit the structure of the interpreter. All our systems run in Scheme: partial evaluators, interpreters, compilers, and object programs (though they are parameterized by the semantic algebras of the specializer and of the interpreter).

This work was possible due to a series of breakthroughs starting with separating the static and dynamic semantics of partial evaluation and including: binding time analysis for non-flat binding time domains [7] and combinator extraction [9].

Present applications address tackling pattern matching [11] and Prolog [10]: environment-wise, all the potential substitutions are computed at compile time; control-wise, failure continuations are discovered to be bound statically, enabling backtracking to be determined entirely at compile

time. We are currently working on the problem of restructuring source programs automatically to ensure them to be specialized better. Future works will include developing a better programming environment; better extensional criteria for the quality of a source specification and their implementation; parameterizing post-optimizers; and the automatic generation of congruence relations and correctness proofs, both from the binding time analysis and from the actual specialization.

Acknowledgements

To Neil D. Jones for his scientific insight as to how to tame self-application. To Mitchell Wand and Margaret Montenyohl for providing the Algol example and for their encouragements. Thanks are also due to Karoline Malmkjær, David Schmidt, Austin Melton, François Bodin, Pierre Jouvelot, Siau Cheng Khoo, Paul Hudak, and Andrzej Filinski for their thoughtful comments. Finally we would like to thank the referees.

References

- [1] D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.
- [2] A. Bondorf. Automatic autoprojection of higher order recursive equations. In N. D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 70–87. Springer-Verlag, 1990.
- [3] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. Diku Research Report 90/04, University of Copenhagen, Copenhagen, Denmark, 1990. To appear in *Science of Computer Programming*.
- [4] A. Bondorf, N. D. Jones, T. Mogensen, and P. Sestoft. Binding time analysis and the taming of self-application. Diku report, University of Copenhagen, Copenhagen, Denmark, 1988.
- [5] 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.
- [6] C. Consel. *Analyse de Programmes, Evaluation Partielle et Generation de Compilateurs*. PhD thesis, Université de Paris VI, Paris, France, 1989.
- [7] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 264–272, 1990.
- [8] C. Consel. *The Schism Manual*. Yale University, New Haven, Connecticut, USA, 1990. Version 1.0.
- [9] 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*, pages 88–105. Springer-Verlag, 1990.
- [10] C. Consel and S. C. Khoo. Semantics-directed generation of a Prolog compiler. Research Report 781, Yale University, New Haven, Connecticut, USA, 1990.
- [11] O. Danvy. Semantics-directed compilation of non-linear patterns. Technical Report 303, Indiana University, Bloomington, Indiana, USA, 1990.
- [12] A. P. Ershov, D. Bjørner, Y. Futamura, K. Furukawa, A. Haraldsson, and W. L. Scherlis, editors. *Selected Papers from the Workshop on Partial Evaluation and Mixed Computation*, volume 6 of 2,3. OHMSHA. LTD. and Springer-Verlag, 1988.
- [13] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls* 2, 5, pages 45–50, 1971.
- [14] H. Ganzinger and N. D. Jones, editors. *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.
- [15] N. D. Jones. Challenging problems in partial evaluation and mixed computation. In [12], pages 1–14.
- [16] N. D. Jones, editor. *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [17] N. D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, and T. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *IEEE International Conference on Computer Languages*, pages 49–58, 1990.
- [18] N. D. Jones and S. S. Muchnick. Some thoughts towards the design of an ideal language. In *ACM Conference on Principles of Programming Languages*, pages 77–94, 1976.
- [19] 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.
- [20] 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):9–50, 1989.
- [21] R. Kesley and P. Hudak. Realistic compilation by program transformation. In *ACM Symposium on Principles of Programming Languages*, pages 281–292, 1989.
- [22] D. A. Kranz, R. Kesley, J. A. Rees, P. Hudak, J. Philbin, and N. I. Adams. Orbit: an optimizing compiler for Scheme. *SIGPLAN Notices, ACM Symposium on Compiler Construction*, 21(7):219–233, 1986.
- [23] P. Lee. *Realistic Compiler Generation*. MIT Press, 1989.
- [24] M. Montenyohl and M. Wand. Correct flow analysis in continuation semantics. In *ACM Symposium on Principles of Programming Languages*, pages 204–218, 1988.
- [25] F. L. Morris. The next 700 formal language descriptions. Unpublished paper, 1970.

- [26] P. Mosses. *SIS – Semantics Implementation System, reference manual and user guide*. University of Aarhus, Aarhus, Denmark, 1979. Version 1.0.
- [27] H. R. Nielson and F. Nielson. Automatic binding time analysis for a typed λ -calculus. In *ACM Symposium on Principles of Programming Languages*, pages 98–106, 1988.
- [28] H. R. Nielson and F. Nielson. The TML-approach to compiler-compilers. Technical Report 88-47, Technical University of Denmark, Lyngby, Denmark, 1988.
- [29] L. Paulson. A semantics-directed compiler generator. In *ACM Symposium on Principles of Programming Languages*, pages 224–233, 1982.
- [30] U. Pleban. Semantics-directed compiler generation. In *ACM Symposium on Principles of Programming Languages*, 1987. Tutorial.
- [31] J. Rees and W. Clinger. Revised³ report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.
- [32] J. Reynolds. Definitional interpreters for higher order programming languages. In *ACM National Conference*, pages 717–740, 1971.
- [33] J. Reynolds. The essence of Algol. In Van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, 1981.
- [34] D. A. Schmidt. Detecting global variables in denotational specification. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, 1985.
- [35] D. A. Schmidt. *Denotational Semantics: a Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
- [36] P. Sestoft. The structure of a self-applicable partial evaluator. In [14], pages 236–256.
- [37] J. Stoy. *Denotational Semantics: the Scott-Strachey approach to programming languages theory*. MIT Press, 1977.
- [38] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.
- [39] M. Wand. A semantic prototyping system. *SIGPLAN Notices, ACM Symposium on Compiler Construction*, 19(6):213–221, 1984.
- [40] M. Wand and Z. Wang. Conditional lambda-theories and the verification of static properties of programs. In *IEEE Symposium on Logic in Computer Science*, pages 321–332, 1990.

A The Algol Definition (continued)

Application of operators

```
(define (applyAritBinop op type)
  (case type
    [(Int)
     (case op
       [(+) addInt]
       [(-) subInt]
       [(*) mulInt]
       [(/) divInt])]
    [(Real)
     (case op
       [(+) addReal]
       [(-) subReal]
       [(*) mulReal]
       [(/) divReal])]))

(define (applyRelBinop op type)
  (case type
    [(Int)
     (case op
       [(=) eqInt]
       [(>) gtInt]
       [(<) ltInt])]
    [(Real)
     (case op
       [(=) eqReal]
       [(>) gtReal]
       [(<) ltReal])]
    [(Bool) (case op [(=) eqBool])]))
```

Scope resolution

```
(define (locIdent ident r)
  (let ([[Environment - mappings) r])
    (fetchLoc ident mappings)))
```

Valuation function for a program

```
(define (evProgram program s)
  (let ([[Program block) program])
    (caseType block
      [(Block declList stmtList)
       (evBlock declList
                 stmtList
                 (Environment 0 initMappings)
                 initCcont
                 s)])]))
```

Valuation function for the expressions

```

(define (evExpr expr r c s)
  (caseType expr
    [(Int value) (c (Int value))]
    [(Real value) (c (Real value))]
    [(Bool value) (c (Bool value))]
    [(Identifier ident) (caseType (locIdent ident r)
      [(IntLoc l) (c (Int (fetchInt l s)))]
      [(RealLoc l) (c (Real (fetchReal l s)))]
      [(BoolLoc l) (c (Bool (fetchBool l s)))]))
    [(AritBinop op expr1 expr2)
      (evExpr expr1 r (lambda (v1)
        (evExpr expr2 r (lambda (v2)
          (caseType v1
            [(Int value1)
              (caseType v2
                [(Int value2) (c (Int ((applyAritBinop op 'Int) value1 value2)))]
                [(Real value2)
                  (c (Real ((applyAritBinop op 'Real) (coerceToReal value1) value2)))]
                [(Bool value2) (terminate 'error9 c s)))]
            [(Real value1)
              (caseType v2
                [(Int value2) (c (Int ((applyAritBinop op 'Int) value1 value2)))]
                [(Real value2)
                  (c (Real ((applyAritBinop op 'Real) (coerceToReal value1) value2)))]
                [(Bool -) (terminate 'error10 c s)]))
            [(Bool -) (terminate 'error11 c s)])) s)) s))]
    [(RelBinop op expr1 expr2)
      (evExpr expr1 r (lambda (v1)
        (evExpr expr2 r (lambda (v2)
          (caseType v1
            [(Int value1)
              (caseType v2
                [(Int value2) (c (Bool ((applyRelBinop op 'Int) value1 value2)))]
                [(Real value2)
                  (c (Bool ((applyRelBinop op 'Real) (coerceToReal value1) value2)))]
                [(Bool value2) (terminate 'error12 c s)]))
            [(Real value1)
              (caseType v2
                [(Int value2) (c (Bool ((applyRelBinop op 'Int) value1 value2)))]
                [(Real value2)
                  (c (Bool ((applyRelBinop op 'Real) (coerceToReal value1) value2)))]
                [(Bool -) (terminate 'error13 c s)]))
            [(Bool -) (terminate 'error14 c s)])) s)) s))]
  )

```

Location and Store algebras

Domain $m \in \overline{\text{Mappings}}$	Domain $s \in \underline{\text{Store}}$
Operations	Operations
$\text{initMappings} : \overline{\text{Mappings}}$	$\text{initStore} : \underline{\text{Store}}$
$\text{addMapping} : \overline{\text{Ident}} \times \overline{\text{Location}} \times \overline{\text{Mappings}} \rightarrow \overline{\text{Mappings}}$	$\text{intUpdate} : \overline{\text{IntLoc}} \times \overline{\text{Int}} \times \underline{\text{Store}} \rightarrow \underline{\text{Store}}$
$\text{fetchLoc} : \overline{\text{Ident}} \times \overline{\text{Mappings}} \rightarrow \overline{\text{Location}}$	$\text{realUpdate} : \overline{\text{RealLoc}} \times \overline{\text{Real}} \times \underline{\text{Store}} \rightarrow \underline{\text{Store}}$
	$\text{BoolUpdate} : \overline{\text{BoolLoc}} \times \overline{\text{Bool}} \times \underline{\text{Store}} \rightarrow \underline{\text{Store}}$
	$\text{fetchInt} : \overline{\text{IntLoc}} \times \underline{\text{Store}} \rightarrow \underline{\text{Int}}$
	$\text{fetchReal} : \overline{\text{RealLoc}} \times \underline{\text{Store}} \rightarrow \underline{\text{Real}}$
	$\text{fetchBool} : \overline{\text{BoolLoc}} \times \underline{\text{Store}} \rightarrow \underline{\text{Bool}}$