Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation

Olivier Danvy and René Vestergaard

BRICS ⋆
Computer Science Department
Aarhus University ⋆⋆

Abstract. We illustrate a simple and effective solution to semantics-based compiling. Our solution is based on "type-directed partial evaluation", and

- our compiler generator is expressed in a few lines, and is efficient;
- its input is a well-typed, purely functional definitional interpreter in the style of denotational semantics;
- the output of the generated compiler is effectively three-address code, in the fashion and efficiency of the Dragon Book;
- the generated compiler processes several hundred lines of source code per second.

The source language considered in this case study is imperative, block-structured, higher-order, call-by-value, allows subtyping, and obeys stack discipline. It is bigger than what is usually reported in the literature on semantics-based compiling and partial evaluation.

Our compiling technique uses the first Futamura projection, *i.e.*, we compile programs by specializing a definitional interpreter with respect to the program. Specialization is carried out using type-directed partial evaluation, which is a mild version of partial evaluation akin to λ-calculus normalization.

Our definitional interpreter follows the format of denotational semantics, with a clear separation between semantic algebras, and valuation functions. It is thus a completely straightforward stack-based interpreter in direct style, which requires no clever staging technique (currying, continuations, binding-time improvements, *etc.*), and does not rely on any other framework (attribute grammars, annotations, *etc.*) than the typed λ-calculus. In particular, it uses no other program analysis than traditional type inference.

The overall simplicity and effectiveness of the approach has encouraged us to write this paper, to illustrate this genuine solution to denotational semantics-directed compilation, in the spirit of Scott and Strachey. Our conclusion is that λ-calculus normalization suffices for compiling by specializing an interpreter.

⋆ Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

⋆⋆ Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: {danvy, jrvest}@brics.dk

1 Introduction

1.1 Denotational semantics and semantics-implementation systems

Twenty years ago, when denotational semantics was developed [23, 37], there were high hopes for it to be used to specify most, if not all programming languages. When Mosses developed his Semantics Implementation System [25], it was with the explicit goal of generating compilers from denotational specifications.

Time passed, and these hopes did not materialize as concretely as was wished. Other semantic frameworks are used today, and other associated semantics-implementation systems as well. Two explanations could be that (1) domains proved to be an interesting area of research *per se*, and they are studied today quite independently of programming-language design and specification; and (2) the λ-notation of denotational semantics was deemed untamable — indeed writing a denotational specification can be compared to writing a program in a module-less, lazy functional language without automatic type-checker.

As for semantics-implementation systems, there have been many [1, 13, 16, 22, 25, 28, 29, 31, 34, 35, 38, 39, 40], and they were all quite complicated. (Note: this list of references is by no means exhaustive. It is merely meant to be indicative.)

1.2 Partial evaluation

For a while, partial evaluation has held some promise for compiling and compiler generation, through the Futamura projections [17, 18]. The first Futamura projection states that specializing a definitional interpreter with respect to a source program "compiles" the source program from the defined language to the defining language [32]. This idea has been applied to a variety of interpreters and programming-language paradigms, as reported in the literature [7, 17].

One of the biggest and most successful applications is Jørgensen's BAWL, which produces code that is competitive with commercially available systems, given a good Scheme implementation [19]. The problem with partial evaluation, however, is the same as for most semantics-implementation system: it requires an expert to use it successfully.

1.3 Type-directed partial evaluation

Recently, the first author has developed an alternative approach to partial evaluation which is better adapted to specializing interpreters and strikingly simpler [8]. The approach is type-directed and amounts to normalizing a closed, well-typed program, given its type (see Figure 1). The approach has been illustrated on a toy programming language, by transliterating a denotational specification into a definitional interpreter, making this term closed by abstracting all the (run-time) semantics operators, applying it to the source program, and normalizing the result.

Type-directed partial evaluation thus merely requires the user to write a purely functional, well-typed definitional interpreter, to close it by abstracting its semantic operators, and to provide its type. The type is obtained for free, using ML or Haskell. No annotations or binding-time improvements are needed.

The point of the experiment with a toy programming language [8] is that it can be carried out at all. The point of this paper is to show that the approach scales up to a non-trivial language.

$$t \in \text{Type} ::= b \mid t_1 \times t_2 \mid t_1 \to t_2$$

$$\begin{aligned}
\text{reify} &= \lambda t.\lambda v.{\downarrow^t}\, v \\
{\downarrow^b}\, v &= v \\
{\downarrow^{t_1 \times t_2}}\, v &= \underline{\text{pair}}({\downarrow^{t_1}}\, \overline{\text{fst}}\, v, {\downarrow^{t_2}}\, \overline{\text{snd}}\, v) \\
{\downarrow^{t_1 \to t_2}}\, v &= \underline{\lambda} x_1.{\downarrow^{t_2}}\, (v\, \overline{@}\, ({\uparrow_{t_1}}\, x_1)) \\
&\quad \text{where } x_1 \text{ is fresh.}
\end{aligned}$$

$$\begin{aligned}
\text{reflect} &= \lambda t.\lambda e.{\uparrow_t}\, e \\
{\uparrow_b}\, e &= e \\
{\uparrow_{t_1 \times t_2}}\, e &= \overline{\text{pair}}({\uparrow_{t_1}}\, \underline{\text{fst}}\, e, {\uparrow_{t_2}}\, \underline{\text{snd}}\, e) \\
{\uparrow_{t_1 \to t_2}}\, e &= \overline{\lambda} v_1.{\uparrow_{t_2}}\, (e\, \underline{@}\, ({\downarrow^{t_1}}\, v_1))
\end{aligned}$$

$$\text{residualize} = \text{statically-reduce} \circ \text{reify}$$

The down arrow is read *reify*: it maps a static value and its type into a two-level λ-term [27] that statically reduces to the dynamic counterpart of this static value. Conversely, the up arrow is read *reflect*: it maps a dynamic expression into a two-level λ-term representing the static counterpart of this dynamic expression.

In residualize, reify (resp. reflect) is applied to types occurring positively (resp. negatively) in the source type.

N.B. In practice, residual let expressions need to be inserted to maintain the order of execution, *à la* Similix [5]. This feature makes it possible to specialize direct-style programs with the same advantages one gets when specializing continuation-passing programs, and is described elsewhere [9].

Fig. 1. Type-directed partial evaluation (a.k.a. residualization)

1.4 Disclaimer

We are not suggesting that well-typed, compositional and purely definitional interpreters written in the fashion of denotational semantics are the way to go always. This method of definition faces the same problems as denotational semantics. For example, it cannot scale up easily to truly large languages, which has motivated *e.g.*, the development of Action Semantics [26].

Our point is that given such a definitional interpreter, type-directed partial evaluation provides a remarkably simple and effective solution to semantics-based compiling.

1.5 This paper

We consider an imperative language with block structure, higher-order and mutually recursive procedures, call-by-value, and that allows subtyping. We write a direct-style definitional interpreter that is stack-based, reflecting the traditional call/return strategy of Algol 60 [30], but is otherwise completely straightforward.[3]

[3] Being stack-based is of course not a requirement. An unstructured store would also need to be threaded throughout, but its implementation would require a garbage collector [23].

We specialize it and obtain three-address code that is comparable to what can be expected from a compiler implemented *e.g.*, according to the Dragon Book [1].

This experiment is significant for the following reasons:

Semantics-implementation systems: Our source language is realistic. Our compiler generator is extremely simple (it is displayed in Figure 1). Our language specification is naturally in direct style and requires no staging transformations [20]. Our compiler is efficient (several hundred lines per second). Our target code is reasonable.

We are not aware of other semantics-implementation systems that yield similar target code with a similar efficiency. In any case, our primary goal is not efficiency. Our point here is that the problem of semantics-based compiling can be stated in such a way that a solution exists that is extremely simple and yields reasonable results.

Partial evaluation: Similar results can be obtained using traditional partial evaluation, but not as simply and not as efficiently.

An online partial evaluator incurs a significant interpretive overhead. An offline partial evaluator necessitates a binding-time analysis, and then either incurs interpretive overhead by direct specialization, or requires its user to generate a generating extension. Furthermore, an offline partial evaluator usually requires binding-time improvements, which are an art in themselves [17, Chapter 12].

It is our experience that in a way, partial evaluators today are too powerful: our present experiment shows that λ-calculus normalization is enough for compiling by interpreter specialization.

1.6 Overview

Section 2 presents the essence of semantics-based compiling by type-directed partial evaluation. Section 3 briefly outlines our source programming language. Based on this description, it is simple to transcribe its denotational specification into a definitional interpreter [32]. We describe one such interpreter in Section 4, and display the outline of it in Figure 6. It is well-typed. Given its type and a source program such as the one in Figure 3, we can residualize the application of the interpreter to the source program and obtain a residual program such as the one in Figure 4. Each residual program is a specialized version of the definitional interpreter and appears as three-address code. We use the syntax of Scheme to represent this three-address code, but it is trivial to map it into assembly code.

2 The essence of semantics-based compiling by type-directed partial evaluation

Our point here is that to compile programs by interpreter specialization, λ-calculus normalization is enough.

Interpreter specialization: the first Futamura projection states that specializing an interpreter with respect to a program (*i.e.*, performing all the interpreter operations that only depend on the program, but not on its input) amounts

to compiling this program into the defining language of the interpreter [17]. Along with the second and third Futamura projections, this property has been instrumental to the renaissance of partial evaluation in the 80's [18].

Normalization is enough: the technique of type-directed partial evaluation is otherwise used to normalize simply typed programs extracted from proofs. There, it is referred to as "normalization by evaluation" [3, 4]. Type-directed partial evaluation slightly differs in that to make it fit call-by-value, we insert let expressions naming residual expressions to avoid code and computation duplication.

Throughout, we consider Schmidt-style denotational specifications [34], *i.e.*, specifications with a clear separation between semantic algebras, and valuation functions. We normalize the result of applying the main valuation function to a program, given the type of its denotation. The result is a combination of semantic-algebra operations, sequentialized with let expressions, and thus corresponding to three-address code [1].

Let us illustrate this point with one extremely simple example. We consider a microscopic imperative language and its definitional interpreter (Figure 2). Applying this interpreter to a source program yields a functional value. Normalizing it yields a textual representation of the dynamic semantics of the source program, *i.e.*, its compiled version: a sequence of semantic-algebra operations.

2.1 Syntax

The following BNF specifies our microscopic imperative language. A program consists of a sequence of zeros and ones terminated by a stop.

$$\begin{aligned} p \in \text{Program} &::= c \\ c \in \text{Command} &::= \text{stop} \mid i\ ;\ c \\ i \in \text{Instruction} &::= \text{zero} \mid \text{one} \end{aligned}$$

2.2 Semantic algebra

The valuation functions uses a semantic algebra consisting of a store and two store-transforming operations f and g.

2.3 Valuation functions

We respectively interpret zero and one with f and g.

$$\begin{array}{rcl} \mathcal{C} : \text{Command} & \to & \text{Store} \to \text{Store} \\ \mathcal{C}[\![\text{stop}]\!] & = & \lambda\sigma.\sigma \\ \mathcal{C}[\![i\ ;\ c]\!] & = & \lambda\sigma.\mathcal{C}[\![c]\!](\mathcal{I}[\![i]\!]\sigma) \end{array} \qquad \begin{array}{rcl} \mathcal{I} : \text{Instruction} & \to & \text{Store} \to \text{Store} \\ \mathcal{I}[\![\text{zero}]\!] & = & f \\ \mathcal{I}[\![\text{one}]\!] & = & g \end{array}$$

2.4 Compiling by normalization

Figure 2 displays a direct-style definitional interpreter, expressed in Scheme. This interpreter is curried: when applied to a source program, it returns a higher-order procedure expecting f and g and returning a store transformer. We have parameterized it with f and g merely for convenience (precisely: to make it a closed term, *i.e.*, a term without free variables that can be characterized with its type).

```
(define-record (Stop))
(define-record (Sequence i c))
(define-record (Zero))
(define-record (One))

(define meaning
  (lambda (c)
    (lambda (f g)
      (letrec ([meaning-command
                (lambda (c)
                  (case-record c
                    [(Stop) (lambda (s) s)]
                    [(Sequence i c) (lambda (s)
                                      ((meaning-command c)
                                       ((meaning-instruction i) s)))]))]
               [meaning-instruction
                (lambda (i)
                  (case-record i
                    [(Zero) f]
                    [(One) g]))])
        (meaning-command c)))))

(define-base-type sto "s")
(define-compound-type one (sto -!> sto) "f" alias)
(define-compound-type two (sto -!> sto) "g" alias)
(define-compound-type denotation-type ((one two) => sto -!> sto))
```

Fig. 2. Microscopic definitional interpreter

The figure also shows the definition of the abstract syntax as records, and the definition of the types of the store, of f, and g. The function type of f and g is annotated with "!" to indicate that they operate on a single-threaded value, and thus that their application should be named with a let expression [9].

Let `p` denote the source program "zero; one; zero; one; stop". In the following Scheme session, we residualize the application of the definitional interpreter with respect to the type of the denotation. The result is the text of the corresponding normal form, *i.e.*, of the dynamic semantics of `p`.

```
> (load "tdpe.scm")
> (load "mic-def-int.scm")
> (residualize (meaning p) 'denotation-type)
(lambda (f g)
  (lambda (s0)
    (let* ([s1 (f s0)]
           [s2 (g s1)]
           [s3 (f s2)])
      (g s3))))
```

This residual Scheme program is the specialized version of the interpreter of Figure

2 with respect to the source program `p`. It performs the run-time actions of `p`, sequentially. The interpretive overhead is gone.

2.5 Assessment

This microscopic example is significant for at least three reasons.

This is semantics-based compilation: we have compiled a program based on the semantics of a programming language.

This is partial evaluation: we have specialized a definitional interpreter with respect to a source program.

This is normalization: all we have used is a λ-calculus normalizer.

Furthermore, the target language of the normalizer (a flat sequence of let expressions) has the same structure as three-address code as described in the Dragon book [1]. Translating residual programs into assembly language therefore does not amount to writing a compiler for the λ-calculus, but more simply to writing the back-end of a standard compiler. The method is thus not a step sideways, but an actual step forward in the process of compiling a source program. In essence, it makes it possible to derive the front-end of a compiler from a definitional interpreter.

Methodologically, our type-directed partial evaluator is simpler than a traditional semantics-implementation system or a standard partial evaluator for three more reasons:

1. Normalization is simpler than partial evaluation.
2. Type-directed partial evaluation does not require one to re-implement the λ-calculus: it relies on an existing implementation (presently Scheme, but ML or Haskell would do just as well).
3. There is no need to massage (a.k.a. stage [20]) the definitional interpreter to derive [the front-end of] the compiler. It is enough to follow the format of denotational semantics: semantic algebras, and valuation functions.

We come back to these issues in Section 6.

The example language of this section is deliberately microscopic, but it captures the essence of our case study. Examples of semantics-based compiling by type-directed partial evaluation have been published for Paulson's Tiny language [8, 9], which is an imperative while-language traditional in the semantics community [16]. In this paper, we explore semantics-based compiling with a more substantial programming language, in an effort to explore the applicability of type-directed partial evaluation.

3 A block-structured procedural language with subtyping

The following programming language is deliberately reminiscent of Reynolds's idealized Algol [33], although it uses call-by-value. We briefly present it here. A full description is available in a companion technical report [10] and in the second author's MS thesis (forthcoming).

3.1 Abstract Syntax

The language is imperative: its basic syntactic units are commands. It is block-structured: any command can have local declarations. It is procedural: commands can be abstracted and parameterized. It is higher-order: procedures can be passed as arguments (though not returned, to enable stack discipline for activation records) to other procedures. Finally it is typed and supports subtyping in that an integer value can be assigned to a real variable, a procedure expecting a real can be passed instead of a procedure expecting an integer, *etc.*

$$
\begin{array}{lll}
p, \langle pgm \rangle & \in Pgm & \text{—domain of programs} \\
c, \langle cmd \rangle & \in Cmd & \text{—domain of commands} \\
e, \langle exp \rangle & \in Exp & \text{—domain of expressions} \\
i, \langle ide \rangle & \in Ide & \text{—domain of identifiers} \\
d, \langle decl \rangle & \in Decl & \text{—domain of declarations} \\
t, \langle type \rangle & \in Type & \text{—domain of types} \\
o, \langle btype \rangle & \in BType & \text{—domain of base types}
\end{array}
$$

$$
\begin{array}{rcl}
\langle pgm \rangle & ::= & \langle cmd \rangle \\
\langle decl \rangle & ::= & \textbf{Var}\ \langle ide \rangle : \langle btype \rangle\ =\ \langle exp \rangle \\
& | & \textbf{Proc}\ \langle ide \rangle\ (\langle ide \rangle{:}\langle type \rangle, \ldots, \langle ide \rangle{:}\langle type \rangle)\ =\ \langle cmd \rangle \\
\langle cmd \rangle & ::= & \textbf{skip} \mid \textbf{write}\ \langle exp \rangle \mid \textbf{read}\ \langle ide \rangle \mid \langle cmd \rangle\ ;\ \langle cmd \rangle \\
& | & \langle ide \rangle := \langle exp \rangle \mid \textbf{if}\ \langle exp \rangle\ \textbf{then}\ \langle cmd \rangle\ \textbf{else}\ \langle cmd \rangle \\
& | & \textbf{while}\ \langle exp \rangle\ \textbf{do}\ \langle cmd \rangle \mid \textbf{call}\ \langle ide \rangle\ (\langle exp \rangle, \ldots, \langle exp \rangle) \\
& | & \textbf{block}\ (\langle decl \rangle, \ldots, \langle decl \rangle)\ \textbf{in}\ \langle cmd \rangle \\
\langle exp \rangle & ::= & \langle lit \rangle \mid \langle ide \rangle \mid \langle exp \rangle\ \langle op \rangle\ \langle exp \rangle \\
\langle lit \rangle & ::= & \langle bool \rangle \mid \langle int \rangle \mid \langle real \rangle \\
\langle op \rangle & ::= & + \mid \times \mid - \mid < \mid = \mid \textbf{and} \mid \textbf{or} \\
\langle btype \rangle & ::= & \textbf{Bool} \mid \textbf{Int} \mid \textbf{Real} \\
\langle type \rangle & ::= & \langle btype \rangle \mid \textbf{Proc}\ (\langle type \rangle, \ldots, \langle type \rangle)
\end{array}
$$

3.2 Semantics

The language is block structured, procedural, and higher-order. Since it is also typed, we define the corresponding domain of values inductively, following its typing structure. Our implementation is stack-based, and so we want to pass parameters on top of the stack. We thus define the denotation of procedures as a store transformer, whose elements are functions accepting a stack whose top frame contains parameters of an appropriate type. We furthermore index the denotation of a procedure with its free variables.

The store can hold untagged booleans, integers, reals, and procedures. It is accessed with typed operators. We define expressible values (the result of evaluating an expression) to be type-annotated storable values. A denotable value (held in the environment) is a reference to the stack, paired with the type of the corresponding stored value.

The stack holds a sequence of activation records which are statically (for the environment) and dynamically (for the continuation) linked via base pointers, as is traditional in Algol-like languages [1, 30]. An activation record is pushed at each procedure call, and popped at each procedure return. A block (of bindings) is

pushed whenever we enter the scope of a declaration, and popped upon exit of this scope. Each block extends the current activation record. Procedures are call-by-value: they are passed the (storable) values of their arguments on the stack [1, 30]. The global store pairs a push-down stack and i/o channels. It is addressed through type-indexed operators.

The language is statically typed: any type mismatch yields an error. In addition, subtyping is allowed: any context expecting a value of type t (*i.e.*, assignments and parameter passing) accepts an expressible value whose type is a subtype of t. Our coercions at higher type simulate intermediate procedures that coerce their arguments as required by their types [33]. Along the same lines, the type-directed partial evaluator of Figure 1 is simply a coercion from static to dynamic [8].

We want the language to obey a stack discipline. Stack discipline can only be broken when values may outlive the scope of their free variables. This can only occur when an identifier i of higher type is updated with the value of an identifier j which was declared in the scope of i. We thus specify that at higher type, i must be local to j.

4 A definitional interpreter and its specialization

We have transcribed the denotational specification of Section 3 into a definitional interpreter which is compositional and purely functional [32, 36]. We make the interpreter a closed term by abstracting all its free variables at the outset (*i.e.*, the semantic operators fix, test, *etc.*). Figure 6 displays the skeleton of the interpreter.

As in Section 2, we can then residualize the application of the definitional interpreter to any source program, with respect to the codomain of the definitional interpreter, *i.e.*, with respect to the type of the meaning of a source program. The result is a textual representation of the dynamic semantics of the source program, *i.e.*, of its step-by-step execution without interpretive overhead.

Figure 3 displays such a source program. Figures 4 and 5 display the resulting target program, unedited (except for the comments).

```
block Var x : Int = 100
      Proc print (y : Real) = write y
      Proc p (q : Proc(Int), y : Int) = call q (x + y)
in call p (print, 4)
```

Fig. 3. Sample source program

```
(lambda (fix test int-to-real conj disj eq-bool read-int read-real
         read-bool write-int write-real write-bool add-int mul-int
         sub-int less-int eq-int add-real mul-real sub-real less-real
         eq-real push-int push-real push-bool push-proc push-func
         push-al push-base-pointer pop-block update-base-pointer
         pop-frame lookup-int lookup-real lookup-bool lookup-proc
         lookup-func update-int update-real update-bool update-proc
         update-func current-al lookup-al update-al) ...
```

Fig. 4. Sample target program (specialized version of Fig. 6 with respect to Fig. 3)

```
...
(lambda (s)
  (let* ([s   (push-int 100 s)]  ;;; decl. of x
         [a0  (current-al s)]
         [s   (push-proc
                (lambda (s)      ;;; decl. of print (code pointer)
                  (let ([r1 (lookup-real 0 0 s)])
                    (write-real s r1)))
                  s)]
         [s   (push-al a0 s)]    ;;; decl. of print (access link)
         [a2  (current-al s)]
         [s   (push-proc
                (lambda (s)   ;;; decl. of p (code pointer)
                  (let* ([p3 (lookup-proc 0 0 s)]
                         [a4 (lookup-al 0 1 s)]
                         [i5 (lookup-int 1 0 s)]
                         [i6 (lookup-int 0 2 s)]
                         [i7 (add-int i5 i6)]
                         [s  (push-al a4 s)]
                         [s  (push-base-pointer s)]
                         [s  (push-int i7 s)]
                         [s  (update-base-pointer s 1)]
                         [s  (p3 s)])
                    (pop-frame s)))
                s)]
         [s   (push-al a2 s)] ;;; decl. of p (access link)
         [p8  (lookup-proc 0 3 s)]
         [a9  (lookup-al 0 4 s)]
         [p10 (lookup-proc 0 1 s)]
         [a11 (lookup-al 0 2 s)]
         [s   (push-al a9 s)]
         [s   (push-base-pointer s)]
         [s   (push-proc
                (lambda (s)   ;;; coercion of print (code pointer)
                  (let* ([i12 (lookup-int 0 0 s)]
                         [a13 (current-al s)]
                         [s   (push-al a13 s)]
                         [s   (push-base-pointer s)]
                         [s   (push-real (int-to-real i12) s)]
                         [s   (update-base-pointer s 1)]
                         [s   (p10 s)])
                    (pop-frame s)))
                s)
         [s   (push-al a11 s)] ;;; coercion of print (access link)
         [s   (push-int 4 s)]
         [s   (update-base-pointer s 3)]
         [s   (p8 s)]          ;;; actual call to p
         [s   (pop-frame s)])
    (pop-block s 5))))
```

Fig. 5. Fig. 4 (continued and ended)

```
(define meaning
  (lambda (p)
    (lambda (fix test ...)
      (lambda (s)
        (letrec ([meaning-program (lambda (p s) ...)]
                 [meaning-command
                  (lambda (c r s)
                    (case-record c
                      ...
                      [(Assign i1 e)
                       (case-record e
                         [(Ide i2)
                          (let ([(Pair x1 t1) ((cdr r) i1)]
                                [(Pair x2 t2) ((cdr r) i2)])
                            (if (or (is-local? x1 x2)
                                    (and (is-base-type? t1)
                                         (is-base-type? t2)))
                                (update t1 x1
                                        (coerce t2 t1 (lookup t2 x2 s))
                                        s)
                                (wrong "not stackable")))]
                         [else
                          (let ([(Pair x1 t1) ((cdr r) i1)]
                                [(ExpVal t2 v2)
                                 (meaning-expression e r s)])
                            (update t1 x1 (coerce t2 t1 v2) s))])]
                      ...))]
                 [meaning-expression (lambda (e r s) ...)]
                 [meaning-operator (lambda (op v1 v2) ...)]
                 [meaning-declarations (lambda (ds r s) ...)]
                 [meaning-declaration (lambda (d r s) ...)])
          (meaning-program p s))))))
```

Fig. 6. Skeleton of the medium definitional interpreter

The source program of Figure 3 illustrates block structure, non-local variables, higher-order procedures, and subtyping. The body of Procedure p refers to the two parameters of p and also to the global integer-typed variable x. Procedure p expects an integer-expecting procedure and an integer. It is passed the real-expecting procedure *print*, which is legal in the present subtyping discipline. Procedure *print* needs to be coerced into an integer-expecting procedure, which is then passed to Procedure p.

The residual program of Figure 4 is a specialized version of the definitional interpreter of Figure 6 with respect to the source program of Figure 3. It is a flat Scheme program threading the store throughout and reflecting the step-by-step execution of the source program. The static semantics of the source program, *i.e.*, all the interpretive steps that only depend on the text of the source program, has been processed at partial-evaluation time: all the location offsets are solved and all primitive operations are properly typed. The coercion, in particular, has been

residualized as a call to an intermediate procedure coercing its integer argument to a real and calling Procedure *print*.

The target language of this partial evaluator is reminiscent of continuation-passing style without continuations, otherwise known as *nqCPS*, *A-normal forms* [11], or *monadic normal forms* [15], *i.e.*, for all practical purposes, three-address code, as in the Dragon book [1]. It can be indifferently considered as a Scheme program or be translated into assembly language.

5 Assessment

5.1 The definitional interpreter

Our definitional interpreter has roughly the same size as the denotational specification of Section 3: 530 lines of Scheme code and less than 16 Kb. This however also includes the treatment of functions, which we have elided here.

The semantic operations occupy 120 lines of Scheme code and about 3.5 Kb.

5.2 Compiler generation

Generating a compiler out of an interpreter, using type-directed partial evaluation, amounts to specializing the type-directed partial evaluator with respect to the type of the interpreter (applied to a source program) [8, Section 2.4]. The improvement obtained by eliminating the type interpretive overhead is negligible in practice.

5.3 Compiling efficiency

We have constructed a number of source programs of varying size (up to 18,000 lines), and have observed that on the average, compiling takes place at about 400 lines per second on a SPARC station 20 with two 75 Mhz processors running Solaris 2.4, using R. Kent Dybvig's Chez Scheme Version 4.1u. On a smaller machine, a SPARC Station ELC with one 33 Mhz processor running SunOS 4.1.1, about 100 lines are compiled per second, again using Chez Scheme.

5.4 Efficiency of the compiled code

The compiled code is of standard, Dragon-book quality [1]. It accounts for the dynamic interpretation steps of the source program. As this paper is going to press, we do not have actual figures yet about the efficiency of assembly code (only of intermediate code).

5.5 Interpreted vs. compiled code

Compiled intermediate code runs four times faster than interpreted code, on the average, in Scheme. This is consistent with traditional results in partial evaluation [7, 17].

6 Related work

6.1 Semantics-implementation systems

Our use of type-directed partial evaluation to specialize a definitional interpreter very precisely matches the goal of Mosses's Semantics Implementation System [25], as witnessed by the following recent quote [26]:

> "SIS [...] took denotational descriptions as input. It transformed a denotational description into a λ-expression which, when applied to the abstract syntax of a program, reduced to a λ-expression that represented the semantics of the program in the form of an input-output function. This expression could be regarded as the 'object code' of the program for the λ-reduction machine that SIS provided. By applying this code to some input, and reducing again, one could get the output of the program according to the semantics."

When SIS was developed, functional languages and their programming environment did not exist. Today's definitional interpreters can be (1) type-checked automatically and (2) interactively tested. Correspondingly, today's λ-reduction machines are simply Scheme, ML, or Haskell systems. Alternatively, though, we can translate our target three-address code directly into assembly language.

SIS was the first semantics-implementation system, but as mentioned in Section 1, it was followed by a considerable number of other systems. All of these systems are non-trivial. Some of them are definitely on the sophisticated side. None of them are so simple that they can, in a type-directed fashion and in a few lines, as in Figure 1,

1. take a well-typed, runnable, unannotated definitional interpreter in direct style, as in Figure 6, together with a source program, as in Figure 3; and
2. produce a textual representation of its dynamic semantics, as in Figure 4.

6.2 Compiler derivation

Deriving compilers from interpreters is a well-documented exercise by now [12, 24, 40]. These derivations are significantly more involved than the present work, and require significant more handcraft and ingenuity. For example, the source interpreter usually has to be expressed in continuation-passing style.

6.3 Partial evaluation

The renaissance of partial evaluation we see in the 90s originates in Jones's Mix project [18], which aimed at compiling by interpreter specialization and at generating compilers by self-application. This seminal work has paved the way for further work on partial evaluation, namely with offline strategies and binding-time improvements [7, 17]. Let us simply mention two such works.

Definitional interpreter for an imperative language: In the proceedings of POPL'91 [6], Consel and Danvy report the successful compilation and compiler generation for an imperative language that is much simpler than the present one (procedureless and stackless). The quality of the residual code is comparable, but a full-fledged partial evaluator is used, including source annotations, and so are several non-trivial binding-time improvements, most notably continuations.

Definitional interpreter for a lazy language: In the proceedings of POPL'92 [19], Jørgensen reports the successful compilation and compiler generation for a lazy language of a commercial scale. The target language is the high-level programming language Scheme. The quality of the result is competitive with contemporary

implementations, given an efficient Scheme implementation. Again, a full-fledged partial evaluator is used, including source annotations, and so is a veritable arsenal of binding-time improvements, including continuations.

This work: In our work, we use λ-calculus normalization, no annotations, no binding-time analysis, no binding-time improvements, and no continuations. Our results are of Dragon-book quality.

7 Conclusion

We hope to have shown that type-directed partial evaluation of a definitional interpreter does scale up to a realistic example with non-trivial programming features. We would also like to point out that our work offers evidence that Scott and Strachey were right, each in their own way, when they developed Denotational Semantics: Strachey in that the λ-calculus provides a proper medium for encoding at least traditional programming languages, as illustrated by Landin [21]; and Scott for organizing this encoding with types and domain theory. The resulting format of denotational semantics proves ideal to apply a six-lines λ-calculus normalizer (using a higher-order functional language) and obtain the front-end of a semantics-based compiler towards three-address code — a strikingly concise and elegant solution to the old problem of semantics-based compiling.

8 Limitations

Modern programming languages (such as ML) have more type structure than Algol-like languages, in that whereas the type of the denotation of an Algol program is always the same, the type of the denotation of an ML program depends on the type of this program. Such languages are beyond the reach of the current state-of-the-art of type-directed partial evaluation, which is simply typed. Higher type systems are necessary — a topic for future work.

Acknowledgements

Grateful thanks to John Hatcliff, Nevin Heintze, Julia L. Lawall, Peter Lee, Karoline Malmkjær, Peter D. Mosses, and Peter O'Hearn for discussions and comments; and to the referees, for their lucid reviews.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
2. France E. Allen, editor. *Proceedings of the 1982 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 17, No 6, Boston, Massachusetts, June 1982. ACM Press.
3. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt and David E. Rydeheard, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, 1995.

4. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
5. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.
6. Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.
7. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
8. Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.
9. Olivier Danvy. Pragmatics of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996. To appear.
10. Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. Technical report BRICS-RS-96-13, Computer Science Department, Aarhus University, Aarhus, Denmark, May 1996.
11. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Prg. Lng. Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.
12. Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.
13. Harald Ganzinger, Robert Giegerich, Ulrich Mönke, and Reinhard Wilhem. A truly generative semantics-directed compiler generator. In Allen [2], pages 172–184.
14. Susan L. Graham, editor. *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, Montréal, Canada, June 1984. ACM Press.
15. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Prg. Lng.*, pages 458–471, Portland, Oregon, January 1994. ACM Press.
16. Neil D. Jones, editor. *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, Aarhus, Denmark, 1980.
17. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.
18. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
19. Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 258–268, Albuquerque, New Mexico, January 1992. ACM Press.
20. Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In Mark Scott Johnson and Ravi Sethi, editors, *Proceedings of the Thirteenth Annual*

ACM Symposium on Principles of Programming Languages, pages 86–96, St. Petersburg, Florida, January 1986.
21. Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.
22. Peter Lee. *Realistic Compiler Generation*. MIT Press, 1989.
23. Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and John Wiley, New York, 1976.
24. Lockwood Morris. The next 700 formal language descriptions. In Carolyn L. Talcott, editor, *Special issue on continuations (Part I)*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4, pages 249–258. Kluwer Academic Publishers, December 1993.
25. Peter D. Mosses. SIS — semantics implementation system, reference manual and user guide. Technical Report MD-30, DAIMI, Computer Science Department, Aarhus University, Aarhus, Denmark, 1979.
26. Peter D. Mosses. Theory and practice of Action Semantics. In *Proceedings of the 1996 Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, 1996. To appear.
27. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
28. Larry Paulson. Compiler generation from denotational semantics. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.
29. Uwe Pleban. Compiler prototyping using formal semantics. In Graham [14], pages 94–105.
30. B. Randell and L. J. Russell. *Algol 60 Implementation*. Academic Press, New York, 1964.
31. Martin R. Raskovsky. Denotational semantics as a specification of code generators. In Allen [2], pages 230–244.
32. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.
33. John C. Reynolds. The essence of Algol. In van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, The Netherlands, 1982. North-Holland.
34. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
35. Ravi Sethi. Control flow aspects of semantics-directed compiling. In Allen [2], pages 245–260.
36. Joseph Stoy. Some mathematical aspects of functional programming. In John Darlington, Peter Henderson, and David A. Turner, editors, *Functional Programming and its Applications*. Cambridge University Press, 1982.
37. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
38. Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.
39. Mitchell Wand. A semantic prototyping system. In Graham [14], pages 213–221.
40. Mitchell Wand. From interpreter to compiler: a representational derivation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, pages 306–324, Copenhagen, Denmark, October 1985.