BRICS RS-96-13 Danvy & Vestergaard: A Case Study in Type-Directed Partial Evaluation

BRICS

Basic Research in Computer Science

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

Olivier Danvy
René Vestergaard

BRICS Report Series RS-96-13

ISSN 0909-0878 May 1996

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

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

Olivier Danvy and René Vestergaard
Aarhus University *
{danvy,jrvest}@brics.dk

May 1996

Abstract

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

- our compiler generator is expressed in a few lines, and is efficient;
- its input is a well-typed, purely functional definitional interpreter in the manner of denotational semantics;
- the output of the generated compiler is 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 this program.

Our definitional interpreter is completely straightforward, stack-based, and in direct style. In particular, it requires no clever staging technique (currying, continuations, binding-time improvements, *etc.*), nor does it 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.

---

*Computer Science Department, Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark. Home pages: http://www.brics.dk/~{danvy,jrvest}. Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55. This work is supported by BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).

1 Introduction

1.1 Denotational semantics and semantics-implementation systems

Twenty years ago, when denotational semantics was developed [27, 45], there were high hopes for it to be used to specify most, if not all programming languages. When Mosses developed his Semantics Implementation System [29], 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. Briefly stated, (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, 14, 20, 26, 29, 31, 34, 35, 37, 41, 43, 46, 47, 48], 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 [13, 22]. The Futamura projections state that specializing a definitional interpreter with respect to a source programs "compiles" this source program from the defined language to the defining language [38]. This idea has been applied to a variety of interpreters and programming-language paradigms, as reported in the literature [7, 21].

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 system [23]. The problem with partial evaluation, however, is the same as for most other 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 [9]. 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 [9] is that it could be carried out at all. The point of this paper is to show that the approach scales up to a non-trivial language.

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. 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 procedures, call-by-value, and that allows subtyping. We write a direct-style definitional interpreter that is stack-based, reflecting the tradi-

$$
\begin{array}{rcl}
t \in \mathrm{Type} & ::= & b \mid t_1 \times t_2 \mid t_1 \rightarrow t_2 \\[1em]
\mathrm{reify} & = & \lambda t.\lambda v.{\downarrow^{t}}\, v \\
{\downarrow^{b}}\, v & = & v \\
{\downarrow^{t_1 \times t_2}}\, v & = & \underline{\mathrm{pair}}({\downarrow^{t_1}}\, \overline{\mathrm{fst}}\; v, {\downarrow^{t_2}}\, \overline{\mathrm{snd}}\; v) \\
{\downarrow^{t_1 \rightarrow t_2}}\, v & = & \underline{\lambda} x_1.{\downarrow^{t_2}}\, (v \,\overline{@}\, ({\uparrow_{t_1}}\, x_1)) \\
 & & \text{where } x_1 \text{ is fresh.} \\[1em]
\mathrm{reflect} & = & \lambda t.\lambda e.{\uparrow_{t}}\, e \\
{\uparrow_{b}}\, e & = & e \\
{\uparrow_{t_1 \times t_2}}\, e & = & \overline{\mathrm{pair}}({\uparrow_{t_1}}\, \underline{\mathrm{fst}}\, e, {\uparrow_{t_2}}\, \underline{\mathrm{snd}}\, e) \\
{\uparrow_{t_1 \rightarrow t_2}}\, e & = & \overline{\lambda} v_1.{\uparrow_{t_2}}\, (e \,\underline{@}\, ({\downarrow^{t_1}}\, v_1)) \\[1em]
\mathrm{residualize} & = & \text{statically-reduce} \circ \mathrm{reify}
\end{array}
$$

The down arrow is read *reify*: it maps a static value and its type into a two-level λ-term [32] 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 [4]. 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 [8].

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

tional call/return strategy of Algol 60 [36], and otherwise is completely straightforward.[1] 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 [24]. 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, we are not focused on cycles. 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: No other partial evaluator today could take our direct-style definition and specialize it as effectively and as efficiently.

An online partial evaluator would incur a significant interpretive overhead. In most likelihood, it would require the definitional interpreter to be CPS-transformed prior to specialization.

An offline partial evaluator would require binding-time improvements [21, Chapter 12]. For example, the interpreter should be expressed in continuation-passing style (CPS). It would necessitate a binding-time analysis, and then either incur interpretive overhead by direct specialization, or would require to generate a generating extension, which would be in CPS and thus less efficient than in direct style. Finally, the target code would be expressed in CPS.

In summary, similar results could be obtained using traditional partial evaluation, but not as simply and not as efficiently.

[1] 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 [27].

1.6 Overview

Section 2 describes our source programming language. Based on this description, it is simple to transcribe it into a definitional interpreter for this language [38]. We briefly describe this interpreter in Section 3, and display parts of it in Figure 3. It is well-typed. Given its type and a source program such as the one in Figure 4, we can residualize the application of the interpreter to this program and obtain a residual program such as the one in Figure 5. Each residual program is a specialized version of the definitional interpreter. We use the syntax of Scheme [5] to represent this three-address code, but it is trivial to map it into assembly code.

2 A Block-Structured Procedural Language with Subtyping

The following programming language is deliberately reminiscent of Reynolds's idealized Algol [39], although it uses call-by-value. Our presentation follows Schmidt's format for denotational specifications [41]. In particular, we use strict, destructuring let expressions.

2.1 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 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.*

The full language also features functions, but for the sake of conciseness, we have omitted them here.

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

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

For simplicity, we make the syntactic domain of booleans, integers, and reals coincide with the corresponding semantic domains, *i.e.*, with $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{R}$, respectively.

2.2 Semantic values

The language is block structured, procedural, and higher-order. Since it is also typed, we define the corresponding domain of values D inductively, following its typing structure.

At base type, $D_{\mathbf{Bool}} = \mathbb{B}$, $D_{\mathbf{Int}} = \mathbb{N}$, and $D_{\mathbf{Real}} = \mathbb{R}$. At higher type, we can try to define a procedure as a function mapping a tuple of values into a store transformer (whose co-domain is lifted, to account for divergence or errors).

$$
D_{\mathbf{Proc}(t_1,\ ...,\ t_n)} \quad = \quad (D_{t_1} \times ... \times D_{t_n}) \to \mathit{Sto} \to \mathit{Sto}_\perp
$$

Our implementation, however, is stack-based, and so we want to pass parameters on top of the stack. This suggests to define $D_{\mathbf{Proc}(t_1,\ ...,\ t_n)}$ as a store transformer, whose elements are functions accepting a stack whose top frame contains parameters of an appropriate type.

The language, however, is also block-structured and lexically scoped, and so we also need to index the denotation of procedures with their free variables (what Reynolds calls their "store shape"). Several such constructions exist, *e.g.*, by Oles [33], and by Even and Schmidt [10], but they are quite complicated.

Fortunately, constructive type theory tells us that an inductively defined domain, such as D, is isomorphic to the product of (1) a well-ordering accounting for the induction and (2) a possibly larger other domain [19].[2] Accordingly, we choose our well-ordering to be isomorphic to the type structure of our language (as defined by its BNF), and we choose the larger domain to be a store transformer.

$$D_{\mathbf{Proc}(t_1,\, ...,\, t_n)} \quad = \quad \{\mathbf{Proc}(t_1,\, ...,\, t_n)\} \times \mathit{Access} \times (\mathit{Sto} \rightarrow \mathit{Sto}_\perp)$$

The first element is the type tag of the procedure; the second element accounts for the store shape of the procedure: it is the well-known static link (access link) of Algol implementations; and the third component is the store transformer.

We use this domain construction to define our storable values, expressible values, and denotable values.

2.2.1 Storable values

The store can hold untagged booleans, integers, reals, and procedures. It is accessed with typed operators.

$$\begin{array}{rcl}
\mathit{StoVal} & = & \mathit{Val}_{\mathbf{Bool}} + \mathit{Val}_{\mathbf{Int}} + \mathit{Val}_{\mathbf{Real}} + \mathit{Val}_{\mathbf{Proc}(t_1,\, ...,\, t_n)} \\
\mathit{Val}_{\mathbf{Bool}} & = & \mathbb{B} \\
\mathit{Val}_{\mathbf{Int}} & = & \mathbb{N} \\
\mathit{Val}_{\mathbf{Real}} & = & \mathbb{R} \\
\mathit{Val}_{\mathbf{Proc}(t_1,\, ...,\, t_n)} & = & \mathit{Access} \times (\mathit{Sto} \rightarrow \mathit{Sto}_\perp)
\end{array}$$

It is the duty of the interpreter to set up a proper access link and to stack up proper actual parameters before calling procedures.

[2] The case in point is Nordström's construction of "multilevel functions" [18].

2.2.2 Expressible values

We define expressible values (the result of evaluating an expression) to be storable values.

2.2.3 Denotable values

A denotable value (held in the environment) is a reference to the stack, paired with the type of the corresponding stored value.

$$\mathit{DenVal} \quad = \quad \mathit{Index} \times \mathit{Type}$$

2.3 Stack architecture

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, 36]. 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. This block extends the current activation record.

Procedures are call-by-value: they are passed the (expressible) values of their arguments on the stack [1, 36]. Calling a procedure is thus achieved as follows:

1. the static link of the procedure, the current base pointer (*i.e.*, the dynamic link of the callee) and the values of all actual parameters are pushed;

2. a new activation record is created by updating the current base pointer to the location immediately above the pushed dynamic link; and

3. the procedure is called.

Upon return, the activation record is popped. This restores the base pointer and thus the static and dynamic links.

Call-by-name would be similarly implemented.

The storable values held on the stack are addressed through the double index of denotable values: the first index specifies the activation record in which the corresponding variable was declared, starting from the top of the stack; and the second index specifies the entry in the record.

2.4 Semantic algebras

Store: *Sto*. The global store pairs a push-down stack and i/o channels. The stack was described above. The i/o channels enable the program to read and write values of base type.

The store is addressed through the following type-indexed operators.

$$
\begin{array}{rcl}
\mathrm{Lookup}_t & : & (\mathit{Index} \times \mathit{Sto}) \rightarrow \mathit{Val}_t \\
\mathrm{Update}_t & : & (\mathit{Index} \times \mathit{Val}_t \times \mathit{Sto}) \rightarrow \mathit{Sto} \\
\mathrm{Push}_t & : & \mathit{Val}_t \times \mathit{Sto} \rightarrow \mathit{Sto} \\
\mathrm{PopBlock} & : & \mathit{Sto} \times \mathbb{N} \rightarrow \mathit{Sto}
\end{array}
$$

where Lookup_t, Push_t, and Update_t construct and deconstruct *StoVal*-values and the $\mathbb{N}$-argument of PopBlock is the size of the block to pop off the stack.

Environment: The environment maps identifiers to denotable values.

Truth values: $\mathbb{B}$.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathrm{Conj} & : & \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} & \mathrm{Read}_{\mathbf{Bool}} & : & \mathit{Sto} \rightarrow \mathit{Sto} \times \mathbb{B} \\
\mathrm{Disj} & : & \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} & \mathrm{Write}_{\mathbf{Bool}} & : & \mathit{Sto} \times \mathbb{B} \rightarrow \mathit{Sto} \\
\mathrm{Eq}_{\mathbf{Bool}} & : & \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} & & &
\end{array}
$$

Integers: $\mathbb{N}$.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathrm{Add}_{\mathbf{Int}} & : & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} & \mathrm{IntToReal} & : & \mathbb{N} \rightarrow \mathbb{R} \\
\mathrm{Sub}_{\mathbf{Int}} & : & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} & \mathrm{Read}_{\mathbf{Int}} & : & \mathit{Sto} \rightarrow \mathit{Sto} \times \mathbb{N} \\
\mathrm{Mul}_{\mathbf{Int}} & : & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} & \mathrm{Write}_{\mathbf{Int}} & : & \mathit{Sto} \times \mathbb{N} \rightarrow \mathit{Sto} \\
\mathrm{Less}_{\mathbf{Int}} & : & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B} & \mathrm{Eq}_{\mathbf{Int}} & : & \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}
\end{array}
$$

Reals: $\mathbb{R}$.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathrm{Add}_{\mathbf{Real}} & : & \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} & \mathrm{Read}_{\mathbf{Real}} & : & \mathit{Sto} \rightarrow \mathit{Sto} \times \mathbb{R} \\
\mathrm{Sub}_{\mathbf{Real}} & : & \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} & \mathrm{Write}_{\mathbf{Real}} & : & \mathit{Sto} \times \mathbb{R} \rightarrow \mathit{Sto} \\
\mathrm{Mul}_{\mathbf{Real}} & : & \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} & \mathrm{Eq}_{\mathbf{Real}} & : & \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{B} \\
\mathrm{Less}_{\mathbf{Real}} & : & \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{B} & & &
\end{array}
$$

Procedures: $\mathbb{P}$.

$$\begin{array}{rcl}
\text{UpdateBasePointer} & : & \mathit{Sto} \times \mathbb{N} \rightarrow \mathit{Sto} \\
\text{PushBasePointer} & : & \mathit{Sto} \rightarrow \mathit{Sto} \\
\text{CurrentAccessLink} & : & \mathit{Sto} \rightarrow \mathit{Access} \\
\text{PushAccessLink} & : & \mathit{Access} \times \mathit{Sto} \rightarrow \mathit{Sto} \\
\text{PopFrame} & : & \mathit{Sto} \rightarrow \mathit{Sto}
\end{array}$$

where UpdateBasePointer updates the current base pointer to the location with offset n $\in \mathbb{N}$ from the stack top. PopFrame behaves as PopBlock but restores the old base pointer at the same time.

Misc. (for the denotation of while loops and conditional expressions)

$$\begin{array}{lcl}
\text{Fix} & : & ((\mathit{Sto} \rightarrow \mathit{Sto}) \rightarrow \mathit{Sto} \rightarrow \mathit{Sto}) \rightarrow \mathit{Sto} \rightarrow \mathit{Sto} \\
\text{Test} & : & \mathbb{B} \times \mathit{Sto} \times \mathit{Sto} \ \rightarrow \mathit{Sto}
\end{array}$$

Size of types: Following the definition of Val_t we define $|\cdot|$ to be the size of values of type t. The size of a value of base type is 1. The size of procedures is 2: one location for the code pointer, and another for the access link.

$$\begin{array}{rcl}
|o| & = & 1 \\
|\mathbf{Proc}(t_1, ..., t_n)| & = & 2
\end{array}$$

2.5 Typing, subtyping, and coercions

The language is 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. The subtyping relation $\leq$ is defined to be the least reflexive relation such that:

$$\mathbf{Int} \leq \mathbf{Real} \qquad\qquad \frac{t'_1 \leq t_1 \quad \ldots \quad t'_n \leq t_n}{\mathbf{Proc}(t_1, ..., t_n) \leq \mathbf{Proc}(t'_1, ..., t'_n)}$$

The typing rules of the language are the obvious ones. (We omit them for conciseness.) The coercion functions are defined in appendix.

$$\text{Coerce}^{t}_{t'} : Val_{t'} \rightarrow Val_t \qquad \forall t', t \ . \ t' \leq t$$

2.6 Stackability

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. To this end, we define the binary relation $\leq_{\text{Index}}$ (read "is local to") as follows.

$$\langle i_1, i_2 \rangle \leq_{\text{Index}} \langle j_1, j_2 \rangle \quad \Leftrightarrow^{def} \quad i_1 < j_1 \vee (i_1 = j_1 \wedge i_2 \geq j_2)$$

This relation could be refined by considering relative lexical positions in the same block, and otherwise is traditional [3].

2.7 Valuation functions

The main valuation functions are displayed in Figure 2. The other ones are shown in appendix.

3 A Definitional Interpreter and its Residualization

We have transcribed the denotational specification of Section 2 into a definitional interpreter which is compositional and purely functional [38, 44]. We make this interpreter a closed term by abstracting all its free variables at the outset (*i.e.*, the semantic operators fix, test, *etc.*). Figure 3 displays the skeleton of the interpreter.

We can then residualize the application of this 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 4 displays such a source program. Figure 5 displays the resulting target program, unedited (except for the comments).

The source program of Figure 4 illustrates block structure, non-local variables, higher-order procedures, and subtyping. The body of Procedure

$$
\begin{array}{rl}
\mathcal{C}: & \mathit{Cmd} \to \mathit{Env} \to \mathit{Sto} \to \mathit{Sto}_\perp \\
\mathcal{C}[\![\mathbf{skip}]\!]\,\rho\,\sigma = & \sigma \\
\mathcal{C}[\![\mathbf{write}\ e]\!]\,\rho\,\sigma = & \text{let inj}_o(v) \ = \ \mathcal{E}[\![e]\!]\ \rho\ \sigma \text{ in Write}_o\langle\sigma,\ v\rangle \\
\mathcal{C}[\![\mathbf{read}\ i]\!]\,\rho\,\sigma = & \text{let } \langle\iota, o\rangle \ = \ \rho\ i \\
 & \quad \langle\sigma', v\rangle \ = \ \text{Read}_o\ \sigma \\
 & \text{in Update}_o\langle\iota,\ v,\ \sigma'\rangle \\
\mathcal{C}[\![c_1\ ;\ c_2]\!]\,\rho\,\sigma = & \text{let } \sigma' \ = \ \mathcal{C}[\![c_1]\!]\,\rho\,\sigma \text{ in } \mathcal{C}[\![c_2]\!]\ \rho\ \ \sigma' \\
\mathcal{C}[\![i_1 := i_2]\!]\,\langle\rho_1, \rho_2\rangle\,\sigma = & \text{let } \langle\iota_1, t_1\rangle \ = \ \rho_2\ i_1 \\
 & \quad \langle\iota_2, t_2\rangle \ = \ \rho_2\ i_2 \\
 & \text{in if } \iota_1 \leq_{\text{Index}} \iota_2 \vee \ |\,t_1\,| = |\,t_2\,| = 1 \\
 & \quad \text{then let } v \ = \ \text{Lookup}_{t_2}\langle\iota_2,\ \sigma\rangle \\
 & \qquad\quad \text{in Update}_{t_1}\langle\iota_1,\ \text{Coerce}^{t_1}_{t_2} v,\ \sigma\rangle \\
 & \quad \text{else wrong ``}\textit{not stackable}\text{''} \\
\mathcal{C}[\![i := e]\!]\,\langle\rho_1, \rho_2\rangle\,\sigma = & \text{let } \langle\iota, t\rangle \ = \ \rho_2\ i \\
 & \quad \text{inj}_{t'}(v') \ = \ \mathcal{E}[\![e]\!]\ \langle\rho_1, \rho_2\rangle\ \sigma \\
 & \text{in Update}_t\langle\iota,\ \text{Coerce}^{t}_{t'} v',\ \sigma\rangle \\
\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]\,\rho\,\sigma = & \text{let inj}_{\mathbf{Bool}}(v) = \mathcal{E}[\![b]\!]\ \rho\ \sigma \\
 & \text{in Test } \langle v,\ \mathcal{C}[\![c_1]\!]\,\rho\,\sigma,\ \mathcal{C}[\![c_2]\!]\,\rho\,\sigma\rangle \\
\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!]\,\rho\,\sigma = & ((\text{Fix } \lambda f.\lambda\sigma.\text{let inj}_b(v) \ = \ \mathcal{E}[\![b]\!]\ \rho\ \sigma \\
 & \qquad\qquad\qquad \text{in Test } \langle v,\ f\ (\mathcal{C}[\![c]\!]\,\rho\,\sigma),\ \sigma\rangle) \\
 & \ \sigma) \\
\mathcal{C}[\![\mathbf{call}\ i\ (e_1, ..., e_n)]\!]\,\rho\,\sigma = & \text{let } \langle\iota, \mathbf{Proc}(t_1, ..., t_n)\rangle \ = \ \rho\ i \\
 & \quad \langle a, p\rangle \ = \ \text{Lookup}_{\mathbf{Proc}(t_1, ..., t_n)}\langle\iota,\ \sigma\rangle \\
 & \quad \sigma_0 \ = \ \text{PushAccessLink } \langle a,\ \sigma\rangle \\
 & \quad \sigma_1 \ = \ \text{PushBasePointer } \sigma_0 \\
 & \quad \text{inj}_{t'_1}(v'_1) \ = \ \mathcal{E}[\![e_1]\!]\ \rho\ \sigma_1 \\
 & \quad \sigma_2 \ = \ \text{Push}_{t_1}\langle\text{Coerce}^{t_1}_{t'_1}\ v'_1,\ \sigma_1\rangle \\
 & \quad \vdots \\
 & \quad \text{inj}_{t'_n}(v'_n) \ = \ \mathcal{E}[\![e_n]\!]\ \rho\ \sigma_n \\
 & \quad \sigma_{n+1} \ = \ \text{Push}_{t_n}\langle\text{Coerce}^{t_n}_{t'_n}\ v'_n,\ \sigma_n\rangle \\
 & \quad \sigma' \ = \ \text{UpdateBasePointer } \langle\sigma_{n+1},\ n\rangle \\
 & \quad \sigma'' \ = \ p\ \sigma' \\
 & \text{in PopFrame } \sigma'' \\
\mathcal{C}[\![\mathbf{block}\ (d_1, ..., d_n)\ \mathbf{in}\ c]\!]\,\rho\,\sigma = & \text{let } \langle\rho', \sigma'\rangle \ = \ \mathcal{D}[\![(d_1, ..., d_n)]\!]\ \rho\ \sigma \\
 & \quad \sigma'' \ = \ \mathcal{C}[\![c]\!]\ \rho'\ \sigma' \\
 & \text{in PopBlock}\langle\sigma'',\ n\rangle
\end{array}
$$

Figure 2: Valuation functions for commands

```
(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))))))
```

Figure 3: Skeleton of the definitional interpreter

$$
\begin{array}{l}
\textbf{block Var } x : \textbf{Int} \;=\; 100 \\
\qquad \textbf{Proc } \mathit{print}\ (y : \textbf{Real}) \;=\; \textbf{write } y \\
\qquad \textbf{Proc } p\ (q : \textbf{Proc(Int)},\ y : \textbf{Int}) \;=\; \textbf{call } q\ (x + y) \\
\textbf{in call } p\ (\mathit{print},\ 4)
\end{array}
$$

Figure 4: 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)
  (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)
           ...
```

Figure 5: Sample target program (specialized version of Figure 3 with respect to Figure 4)

```
     ...
     [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))))
```

Figure 6: Figure 5 (continued and ended)

p refers to the two parameters of p and also to the global int-typed variable x. Procedure p expects an int-expecting procedure and an int. It is passed the real-expecting procedure *print*, which is legal in the present subtyping discipline. Procedure *print* needs to be coerced into an int-expecting procedure, which is then passed to Procedure p.

The residual program of Figure 5 is a specialized version of the definitional interpreter of Figure 3 with respect to the source program of Figure 4. 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 int 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* [17], *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.

4 Assessment

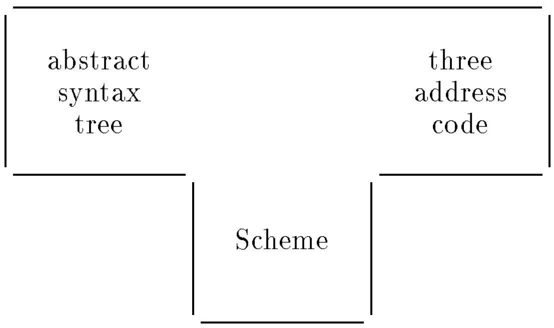
4.1 The definitional interpreter

Our definitional interpreter has roughly the same size as the denotational specification of Section 2: 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.

4.2 Compiler generation

Generating a compiler out of an interpreter, in 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) [9, Section 2.4]. The improvement this yields is negligible in practice.



4.3 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.

4.4 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. An optimizing interpreter would yield optimized code.

4.5 Interpreted vs. compiled code

Compiled code runs consistently four times faster than interpreted code, on the average, in Scheme. This is consistent with traditional results in partial evaluation [7, 21].

5 Related work

5.1 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 [29], as witnessed by the following recent quote [30]:

> "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 3, together with a source program in Figure 4; and

2. produce a textual representation of its dynamic semantics, as in Figure 5.

5.2 Compiler derivation

Deriving compilers from interpreters is a well-documented exercise by now [12, 28, 48]. 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.

5.3 Partial evaluation

The renaissance of partial evaluation we see in the 90s originates in Jones's Mix project [22], which aimed to compile by interpreter specialization and to generate 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, 21]. Let us simply mention two such works.

Definitional interpreter for an imperative language: In the proceedings of POPL'91 [6], Consel and Danvy reported 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 was comparable, but a full-fledged partial evaluator was used, including source annotations, and so were several non-trivial binding-time improvements, most notably continuations.

Definitional interpreter for a lazy language: In the proceedings of POPL'92 [23], Jørgensen reported the successful compilation and compiler generation for a lazy language of the scale of a commercialized product. The target language was Scheme. The quality of the result was competitive, given a good Scheme system. Again, a full-fledged partial evaluator was used, including source annotations, and so was a veritable arsenal of binding-time improvements, including continuations.

This work: In our work, we use a very simple partial evaluator (displayed in Figure 1), no annotations, no binding-time analysis, no binding-time improvements, and no continuations. Our results are of Dragon-book quality.

6 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 [25]; 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.

7 Limitations

Modern programming languages (such as ML) have more type structure than Algol-like languages. They 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 Nevin Heintze, Julia L. Lawall, Karoline Malmkjær, Peter D. Mosses, and Peter O'Hearn for discussions and comments. Many years ago, David Schmidt pointed out to the first author the relevance of constructive type theory for inductively defined domains.

The diagram of Section 4 was drawn with Kristoffer Rose's XY-pic package.

A Valuation functions

$$\begin{array}{rcl} \mathcal{P} & : & \mathit{Pgm} \rightarrow \mathit{Sto} \rightarrow \mathit{Sto}_\perp \\ \mathcal{P}[\![c]\!]\ \sigma & = & \mathcal{C}[\![c]\!]\ \langle 0, \perp \rangle\ \sigma \end{array}$$

$$
\begin{aligned}
&\mathcal{D} : \mathit{Dcls} \to \mathit{Env} \to \mathit{Sto} \to (\mathit{Env} \times \mathit{Sto})_\perp \\
&\mathcal{D}[\![(d_1,\, ...,\, d_n)]\!]\ \rho\ \sigma \\
&\ = \text{let } \langle \rho_1,\ \sigma_1 \rangle\ =\ \mathcal{D}'[\![d_1]\!]\ \rho\ \sigma \\
&\qquad\quad \vdots \\
&\qquad\quad \langle \rho_n,\ \sigma_n \rangle\ =\ \mathcal{D}'[\![d_n]\!]\ \rho_{n-1}\ \sigma_{n-1} \\
&\quad\ \text{in } \langle \rho_n,\ \sigma_n \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}' : \mathit{Decl} \to \mathit{Env} \to \mathit{Sto} \to (\mathit{Env} \times \mathit{Sto})_\perp \\
&\mathcal{D}'[\![\mathbf{Var}\ i{:}\mathbf{Bool}\ =\ e]\!]\ \langle \rho_1,\ \rho_2 \rangle\ \sigma \\
&= \text{let inj}_{\mathbf{Bool}}(v)\ =\ \mathcal{E}[\![e]\!]\ \langle \rho_1, \rho_2 \rangle\ \sigma \\
&\quad \text{in } \langle \langle \rho_1 + 1,\ [i \mapsto \langle \langle 0,\ \rho_1 \rangle,\ \mathbf{Bool} \rangle]\rho_2 \rangle,\ \text{Push}_{\mathbf{Bool}} \langle v,\ \sigma \rangle \rangle \\
&\mathcal{D}'[\![\mathbf{Var}\ i{:}\mathbf{Int}\ =\ e]\!]\ \langle \rho_1,\ \rho_2 \rangle\ \sigma \\
&= \text{let inj}_{\mathbf{Int}}(v)\ =\ \mathcal{E}[\![e]\!]\ \langle \rho_1, \rho_2 \rangle\ \sigma \\
&\quad \text{in } \langle \langle \rho_1 + 1,\ [i \mapsto \langle \langle 0,\ \rho_1 \rangle,\ \mathbf{Int} \rangle]\rho_2 \rangle,\ \text{Push}_{\mathbf{Int}} \langle v,\ \sigma \rangle \rangle \\
&\mathcal{D}'[\![\mathbf{Var}\ i{:}\mathbf{Real}\ =\ e]\!]\ \langle \rho_1,\ \rho_2 \rangle\ \sigma \\
&= \text{let } v = \text{case } \mathcal{E}[\![e]\!]\ \langle \rho_1, \rho_2 \rangle\ \sigma \text{ of} \\
&\qquad\qquad\quad \text{inj}_{\mathbf{Int}}(v) \Rightarrow \text{IntToReal } v \\
&\qquad\qquad\ [\!]\ \ \text{inj}_{\mathbf{Real}}(v) \Rightarrow v \\
&\qquad\qquad\ \text{end} \\
&\quad \text{in } \langle \langle \rho_1 + 1,\ [i \mapsto \langle \langle 0,\ \rho_1 \rangle,\ \mathbf{Real} \rangle]\rho_2 \rangle,\ \text{Push}_{\mathbf{Real}} \langle v,\ \sigma \rangle \rangle \\
&\mathcal{D}'[\![\mathbf{Proc}\ i\ (i_1{:}t_1,\, ...,\, i_n{:}t_n)\ =\ c]\!]\ \langle \rho_1,\ \rho_2 \rangle\ \sigma \\
&= \text{let } \rho' = \langle \Sigma_{i=1}^{n-1} |\, t_i \,|,\ [i_n \mapsto \langle \langle 0, \Sigma_{i=1}^{n-1} |\, t_i \,| \rangle, t_n \rangle]...[i_1 \mapsto \langle \langle 0, 0 \rangle, t_1 \rangle]\rho_2^{+} \rangle \\
&\qquad\ v\ = \langle \text{CurrentAccessLink } \sigma,\ \lambda\ \sigma\ .\ \mathcal{C}[\![c]\!]\ \ \rho'\ \sigma \rangle \\
&\qquad\ \sigma' = \ \text{Push}_v \langle v,\ \sigma \rangle \\
&\quad \text{in } \langle \langle \rho_1 + 2,\ [i \mapsto \langle \langle 0,\ \rho_1 \rangle,\ \mathbf{Proc}(t_1,\, ...,\, t_n) \rangle]\rho_2 \rangle,\ \sigma' \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E} :\ \ & \mathit{Exp} \to \mathit{Env} \to \mathit{Sto} \to \mathit{ExpVal}_\perp \\
\mathcal{E}[\![n]\!]\ \rho\ \sigma &= \text{inj}_{\mathbf{Int}}(n) \\
\mathcal{E}[\![r]\!]\ \rho\ \sigma &= \text{inj}_{\mathbf{Real}}(r) \\
\mathcal{E}[\![b]\!]\ \rho\ \sigma &= \text{inj}_{\mathbf{Bool}}(b) \\
\mathcal{E}[\![i]\!]\ \rho\ \sigma &= \text{let } \langle \iota, t \rangle\ =\ \rho\ i \text{ in inj}_t(\text{Lookup}_t \langle \iota,\ \sigma \rangle) \\
\mathcal{E}[\![e_1\ \mathit{op}\ e_2]\!]\ \rho\ \sigma &= \mathcal{O}[\![\mathit{op}]\!]\ \langle \mathcal{E}[\![e_1]\!]\ \rho\ \sigma,\ \mathcal{E}[\![e_2]\!]\ \rho\ \sigma \rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{O} &: Op \to (ExpVal \times ExpVal) \to ExpVal_\perp \\
\mathcal{O}[\![+]\!]\ \langle \mathrm{inj}_{\mathbf{Int}}(v_1),\ \mathrm{inj}_{\mathbf{Int}}(v_2)\rangle &= \mathrm{inj}_{\mathbf{Int}}(\mathrm{Add}_{\mathbf{Int}}\langle v_1,\ v_2\rangle) \\
\mathcal{O}[\![+]\!]\ \langle \mathrm{inj}_{\mathbf{Real}}(v_1),\ \mathrm{inj}_{\mathbf{Real}}(v_2)\rangle &= \mathrm{inj}_{\mathbf{Real}}(\mathrm{Add}_{\mathbf{Real}}\langle v_1,\ v_2\rangle) \\
\mathcal{O}[\![+]\!]\ \langle \mathrm{inj}_{\mathbf{Int}}(v_1),\ \mathrm{inj}_{\mathbf{Real}}(v_2)\rangle &= \mathrm{inj}_{\mathbf{Real}}(\mathrm{Add}_{\mathbf{Real}}\langle \mathrm{IntToReal}\ v_1,\ v_2\rangle) \\
\mathcal{O}[\![+]\!]\ \langle \mathrm{inj}_{\mathbf{Real}}(v_1),\ \mathrm{inj}_{\mathbf{Int}}(v_2)\rangle &= \mathrm{inj}_{\mathbf{Real}}(\mathrm{Add}_{\mathbf{Real}}\langle v_1,\ \mathrm{IntToReal}\ v_2\rangle) \\
&\ \vdots
\end{aligned}
$$

B Coercions

The coercion functions

$$\mathrm{Coerce}^{t}_{t'} : Val_{t'} \to Val_t \qquad \forall t', t\ .\ t' \leq t$$

are the Store-based $\leq$-extension of IntToReal defined by

$$\mathrm{Coerce}^{t}_{t} v \ = \ v$$

$$\mathrm{Coerce}^{\mathbf{Real}}_{\mathbf{Int}} v \ = \ \mathrm{IntToReal}\ v$$

$$
\begin{aligned}
&\mathrm{Coerce}^{\mathbf{Proc}(t_1,\ldots,t_n)}_{\mathbf{Proc}(t'_1,\ldots,t'_n)}\langle a, p\rangle \\
&= \langle a, \lambda\, \sigma\ .\ \mathrm{let}\ \sigma_0 = \mathrm{PushAccessLink}\ \langle a,\ \sigma\rangle \\
&\qquad\qquad \sigma_1 = \mathrm{PushBasePointer}\ \sigma_0 \\
&\qquad\qquad v'_1 = \mathrm{Coerce}^{t'_1}_{t_1}\ (\mathrm{Lookup}_{t_1}\langle\langle 0, 0\rangle,\ \sigma_1\rangle) \\
&\qquad\qquad \sigma_2 = \mathrm{Push}_{t'_1}\langle v'_1,\ \sigma_1\rangle \\
&\qquad\qquad \vdots \\
&\qquad\qquad v'_n = \mathrm{Coerce}^{t'_n}_{t_n}\ (\mathrm{Lookup}_{t_n}\langle\langle 0, \Sigma_{i=1}^{n-1} \,|\, t_i \,|\rangle,\ \sigma_n\rangle) \\
&\qquad\qquad \sigma_{n+1} = \mathrm{Push}_{t'_n}\langle v'_n,\ \sigma_n\rangle \\
&\qquad\qquad \sigma' = \mathrm{UpdateBasePointer}\ \langle \sigma_{n+1},\ n\rangle \\
&\qquad\qquad \sigma'' = p\ \sigma' \\
&\qquad\quad \mathrm{in}\ \ \mathrm{PopFrame}\ \sigma''\rangle
\end{aligned}
$$

In essence, a coercion at higher type simulates an intermediate procedure or function that coerces its arguments or result as required by the types [40, 42]. In that sense, the type-directed partial evaluator of Figure 1 is simply a coercion from static to dynamic [9].

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] Daniel M. Berry. Block structure: Retention or deletion? (extended abstract). In *Conference Record of the Third Annual ACM Symposium on Theory of Computing*, pages 86–100, Shaker Heights, Ohio, May 1971.

[4] 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.

[5] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 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. Pragmatics of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996. To appear.

[9] 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.

[10] Susan Even and David A. Schmidt. Category-sorted algebra-based action semantics. *Theoretical Computer Science*, 77:73–96, 1990.

[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 Programming Languages 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] Yoshihito Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2, 5*, pages 45–50, 1971.

[14] Harald Ganzinger, Robert Giegerich, Ulrich Mönke, and Reinhard Wilhem. A truly generative semantics-directed compiler generator. In Allen [2], pages 172–184.

[15] Harald Ganzinger and Neil D. Jones, editors. *Programs as Data Objects*, number 217 in Lecture Notes in Computer Science, Copenhagen, Denmark, October 1985.

[16] Susan L. Graham, editor. *Proceedings of the 1984 Symposium on Compiler Construction*, SIGPLAN Notices, Vol. 19, No 6, Montréal, Canada, June 1984. ACM Press.

[17] 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 Programming Languages*, pages 458–471, Portland, Oregon, January 1994. ACM Press.

[18] Bengt Nordström. Multilevel functions in martin-Löf's type theory. In Ganzinger and Jones [15], pages 206–221.

[19] Bengt Nordström, Kent Petersson, and Jan Smith. *Programming and Martin-Löf's Type Theory.* International Series on Monographs on Computer Science No. 7. Oxford University Press, 1990.

[20] Neil D. Jones, editor. *Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science, Aarhus, Denmark, 1980.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation. *Communications of the ACM*, 8:89–101 and 158–165, 1965.

[26] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[27] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, and John Wiley, New York, 1976.

[28] 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.

[29] 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.

[30] 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.

[31] Steve S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[32] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[33] Frank J. Oles. Type algebras, functor categories, and block structure. In *Algebraic Methods in Semantics*, pages 543–573. Cambridge University Press, 1985.

[34] Larry Paulson. Compiler generation from denotational semantics. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.

[35] Uwe Pleban. Compiler prototyping using formal semantics. In Graham [16], pages 94–105.

[36] B. Randell and L. J. Russell. *Algol 60 Implementation*. Academic Press, New York, 1964.

[37] Martin R. Raskovsky. Denotational semantics as a specification of code generators. In Allen [2], pages 230–244.

[38] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.

[39] John C. Reynolds. The essence of Algol. In van Vliet, editor, *International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, 1982. North-Holland.

[40] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1988.

[41] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[42] David A. Schmidt. *The Structure of Typed Programming Languages*. The MIT Press, 1994.

[43] Ravi Sethi. Control flow aspects of semantics-directed compiling. In Allen [2], pages 245–260.

[44] 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.

[45] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[46] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[47] Mitchell Wand. A semantic prototyping system. In Graham [16], pages 213–221.

[48] Mitchell Wand. From interpreter to compiler: a representational derivation. In Ganzinger and Jones [15], pages 306–324.

Recent Publications in the BRICS Report Series

RS-96-13 **Olivier Danvy and René Vestergaard.** ***Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation*****. May 1996. 28 pp.**

RS-96-12 **Lars Arge, Darren E. Vengroff, and Jeffrey S. Vitter.** ***External-Memory Algorithms for Processing Line Segments in Geographic Information Systems*****. May 1996. 34 pp. An shorter version of this paper was presented at the** ***Third Annual European Symposium on Algorithms*****, ESA '95.**

RS-96-11 **Devdatt Dubhashi, David A. Grable, and Alessandro Panconesi.** ***Near-Optimal, Distributed Edge Colouring via the Nibble Method*****. May 1996. 17 pp. Invited to be published in in a special issue of** ***Theoretical Computer Science*** **devoted to the proceedings of ESA '95.**

RS-96-10 **Torben Braüner and Valeria de Paiva.** ***Cut-Elimination for Full Intuitionistic Linear Logic*****. April 1996. 27 pp. Also available as Technical Report 395, Computer Laboratory, University of Cambridge.**

RS-96-9 **Thore Husfeldt, Theis Rauhe, and Søren Skyum.** ***Lower Bounds for Dynamic Transitive Closure, Planar Point Location, and Parentheses Matching*****. April 1996. 11 pp. To appear in** ***Algorithm Theory: 5th Scandinavian Workshop*****, SWAT '96 Proceedings, LNCS, 1996.**

RS-96-8 **Martin Hansen, Hans Hüttel, and Josva Kleist.** ***Bisimulations for Asynchronous Mobile Processes*****. April 1996. 18 pp. Appears in** ***Tbilisi Symposium on Language, Logic, and Computation*****, 1995.**

RS-96-7 **Ivan Damgård and Ronald Cramer.** ***Linear Zero-Knowledegde - A Note on Efficient Zero-Knowledge Proofs and Arguments*****. April 1996. 17 pp.**

RS-96-6 **Mayer Goldberg.** ***An Adequate Left-Associated Binary Numeral System in the*** **λ*****-Calculus (Revised Version)*****. March 1996. 19 pp. Accepted for** ***Information Processing Letters*****. This report is a revision of the BRICS Report RS-95-38.**