BRICS

Basic Research in Computer Science

A Unifying Approach to Goal-Directed Evaluation

Olivier Danvy
Bernd Grobauer
Morten Rhiger

BRICS Report Series RS-01-29

ISSN 0909-0878 July 2001

A Unifying Approach to Goal-Directed Evaluation [*]

Olivier Danvy, Bernd Grobauer, and Morten Rhiger
BRICS [†]
Department of Computer Science
University of Aarhus [‡]

July 13, 2001

Abstract

Goal-directed evaluation, as embodied in Icon and Snobol, is built on the notions of backtracking and of generating successive results, and therefore it has always been something of a challenge to specify and implement. In this article, we address this challenge using computational monads and partial evaluation.

We consider a subset of Icon and we specify it with a monadic semantics and a list monad. We then consider a spectrum of monads that also fit the bill, and we relate them to each other. For example, we derive a continuation monad as a Church encoding of the list monad. The resulting semantics coincides with Gudeman's continuation semantics of Icon.

We then compile Icon programs by specializing their interpreter (i.e., by using the first Futamura projection), using type-directed partial evaluation. Through various back ends, including a run-time code generator, we generate ML code, C code, and OCaml byte code. Binding-time analysis and partial evaluation of the continuation-based interpreter automatically give rise to C programs that coincide with the result of Proebsting's optimized compiler.

[*]To appear in New Generation Computing, Vol. 20 No. 1, Nov. 2001. A preliminary version is available in the proceedings of SAIG 2001.

[†]Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

[‡]Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: {`danvy,grobauer,mrhiger`}`@brics.dk`
Home pages: `http://www.brics.dk/~`{`danvy,grobauer,mrhiger`}

Contents

List of Figures

1 Introduction

Goal-directed languages combine expressions that can yield multiple results through backtracking. Results are generated one at a time: an expression can either succeed and generate a result, or fail. If an expression fails, control is passed to a previous expression to generate the next result, if any. If so, control is passed back to the original expression in order to try whether it can succeed this time. Goal-directed programming specifies the order in which subexpressions are retried, thus providing the programmer with a succint and powerful control-flow mechanism. A well-known goal-directed language is Icon [11].

Backtracking as a language feature complicates both semantics and implementation. Gudeman [13] gives a continuation semantics of a goal-directed language; continuations have also been used in implementations of languages with control structures similar to those of goal-directed evaluation, such as Prolog [3, 15, 30]. Proebsting and Townsend, the implementors of an Icon compiler in Java, observe that continuations can be compiled into efficient code [1, 14], but nevertheless dismiss them because "[they] are notoriously difficult to understand, and few target languages directly support them" [23, p.38]. Instead, their compiler is based on a translation scheme proposed by Proebsting [22], which is based on the four-port model used for describing control flow in Prolog [2]. Icon expressions are translated to a flow-chart language with conditional, direct and indirect jumps using templates; a subsequent optimization which, amongst other things, reorders code and performs branch chaining, is necessary to produce compact code. The reference implemention of Icon [12] compiles Icon into byte code; this byte code is then executed by an interpreter that controls the control flow by keeping a stack of expression frames.

In this article, we present a unified approach to goal-directed evaluation:

1. We consider a spectrum of semantics for a small goal-directed language. We relate them to each other by deriving semantics such as Gudeman's [13] as instantiations of one generic semantics based on computational monads [21]. This unified approach enables us to show the equivalence of different semantics simply and systematically. Furthermore, we are able to show strong conceptual links between different semantics: Continuation semantics can be derived from semantics based on lists or on streams of results by Church-encoding the lists or the streams, respectively.

2. We link semantics and implementation through semantics-directed compilation using partial evaluation [5, 17]. In particular, binding-time analysis guides us to extract templates from the specialized interpreters. These templates are similar to Proebsting's, and through partial evaluation, they give rise to similar flow-chart programs, demonstrating that templates are not just a good idea—they are intrinsic to the semantics of Icon and can be provably derived.

The rest of the paper is structured as follows: In Section 2 we first describe syntax and monadic semantics of a small subset of Icon; we then instantiate the semantics with various monads, relate the resulting semantics to each other, and present an equivalence proof for two of them. In Section 3 we describe semantics-directed compilation for a goal-directed language. Section 4 concludes.

2 Semantics of a Subset of Icon

An intuitive explanation of goal-directed evaluation can be given in terms of lists and list-manipulating functions. Consequently, after introducing the subset of Icon treated in this paper, we define a monadic semantics in terms of the list monad. We then show that also a stream monad and two different continuation monads can be used, and we give an example of how to prove equivalence of the resulting monads using a monad morphism.

2.1 A subset of the Icon programming language

We consider the following subset of Icon:

$$E ::= i \mid E_1 \texttt{ + } E_2 \mid E_1 \texttt{ to } E_2 \mid E_1 \texttt{ <= } E_2 \mid \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3$$

Intuitively, an Icon term either fails or succeeds with a value. If it succeeds, then subsequently it can be resumed, in which case it will again either succeed or fail. This process ends when the expression fails. Informally, i succeeds with the value i; E_1 `+` E_2 succeeds with the sum of the sub-expressions; E_1 `to` E_2 (called a *generator*) succeeds with the value of E_1 and each subsequent resumption yields the rest of the integers up to the value of E_2, at which point it fails; E_1 `<=` E_2 succeeds with the value of E_2 if it is larger than the value E_1, otherwise it fails; `if` E_1 `then` E_2 `else` E_3 produces the results of E_2 if E_1 succeeds, otherwise it produces the results of E_3.

Generators can be nested. For example, the Icon term `4 to (5 to 7)` generates the result of the expressions `4 to 5`, `4 to 6`, and `4 to 7` and concatenates the results.

In a functional language such as Scheme, ML or Haskell, we can achieve the effect of Icon terms using the functions `map` and `concat`. For example, if we define

```
fun to i j = if i<=j then i::(to (i+1) j) else nil
```

in ML, then evaluating `concat (map (to 4) (to 5 7))` yields `[4, 5, 4, 5, 6, 4, 5, 6, 7]` which is the list of the integers produced by the Icon term `4 to (5 to 7)`.

2.2 Monads and semantics

Computational monads were introduced to structure denotational semantics [21]. The basic idea is to parameterize a semantics over a monad; many language extensions, such as adding a store or exceptions, can then be carried out by simply instantiating the semantics with a suitable monad. Further, correspondence

$$\begin{array}{lcl} unit_{\mathsf{M}} & : & \alpha \to \alpha\,\mathsf{M} \\ map_{\mathsf{M}} & : & (\alpha \to \beta) \to \alpha\,\mathsf{M} \to \beta\,\mathsf{M} \\ join_{\mathsf{M}} & : & (\alpha\,\mathsf{M})\,\mathsf{M} \to \alpha\,\mathsf{M} \end{array}$$

Figure 1: Monad operators and their types

Standard monad operations:

$$\begin{array}{lcl} unit_{\mathsf{L}}\ x & = & [x] \\ map_{\mathsf{L}}\ f\ [\,] & = & [\,] \\ map_{\mathsf{L}}\ f\ (x :: xs) & = & (f\ x) :: (map_{\mathsf{L}}\ f\ xs) \\ join_{\mathsf{L}}\ [\,] & = & [\,] \\ join_{\mathsf{L}}\ (l :: ls) & = & l\ @\ (join_{\mathsf{L}}\ ls) \end{array}$$

Special operations for sequences:

$$\begin{array}{lcl} empty_{\mathsf{L}} & = & [\,] \\ if\_empty_{\mathsf{L}}\ [\,]\ ys\ zs & = & ys \\ if\_empty_{\mathsf{L}}\ (x :: xs)\ ys\ zs & = & zs \\ append_{\mathsf{L}}\ xs\ ys & = & xs\ @\ ys \end{array}$$

Figure 2: The list monad

proofs between semantics arising from instantiation with different monads can be conducted in a modular way, using the concept of a monad morphism [28].

Monads can also be used to structure functional programs [29]. In terms of programming languages, a monad M is described by a unary type constructor M and three operations $unit_{\mathsf{M}}$, map_{M} and $join_{\mathsf{M}}$ with types as displayed in Figure 1. For these operations, the so-called monad laws have to hold.

In Section 2.4 we give a denotational semantics of the goal-directed language described in Section 2.1. Anticipating semantics-directed compilation by partial evaluation, we describe the semantics in terms of ML, in effect defining an interpreter. The semantics $[\![\cdot]\!]_{\mathsf{M}} : Exp \to int\,\mathsf{M}$ is parameterized over a monad M, where $\alpha\,\mathsf{M}$ represents a sequence of values of type α.

$$
\begin{array}{rcl}
[\![\cdot]\!]_{\mathsf{M}} & : & \mathit{Exp} \to \mathit{int}\,\mathsf{M} \\[1em]
[\![i]\!]_{\mathsf{M}} & = & \mathit{unit}_{\mathsf{M}}\ i \\
[\![E_1\,\texttt{to}\,E_2]\!]_{\mathsf{M}} & = & \mathit{bind2}_{\mathsf{M}}\ (\lambda xy.\mathit{to}_{\mathsf{M}}\ x\ y)\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![E_1\,\texttt{+}\,E_2]\!]_{\mathsf{M}} & = & \mathit{bind2}_{\mathsf{M}}\ (\lambda xy.\mathit{unit}_{\mathsf{M}}\ (x+y))\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![E_1\,\texttt{<=}\,E_2]\!]_{\mathsf{M}} & = & \mathit{bind2}_{\mathsf{M}}\ (\lambda xy.\mathit{leq}_{\mathsf{M}}\ x\ y)\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![\texttt{if}\,E_0\,\texttt{then}\,E_1 & & \\
\texttt{else}\,E_2]\!]_{\mathsf{M}} & = & \mathit{if\_empty}_{\mathsf{M}}\ [\![E_0]\!]_{\mathsf{M}}\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}}
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathit{bind2}_{\mathsf{M}}\ f\ xs\ ys & = & \mathit{join}_{\mathsf{M}}\ (\mathit{map}_{\mathsf{M}}\ (\lambda x.\mathit{join}_{\mathsf{M}}\ (\mathit{map}_{\mathsf{M}}\ (f\ x)\ ys))\ xs) \\
\mathit{leq}_{\mathsf{M}}\ i\ j & = & \textbf{if}\ i \le j\ \textbf{then}\ \mathit{unit}_{\mathsf{M}}\ j\ \textbf{else}\ \mathit{empty}_{\mathsf{M}} \\
\mathit{to}_{\mathsf{M}}\ i\ j & = & \textbf{if}\ i > j\ \textbf{then}\ \mathit{empty}_{\mathsf{M}} \\
 & & \qquad\qquad \textbf{else}\ \mathit{append}_{\mathsf{M}}\ (\mathit{unit}_{\mathsf{M}}\ i)\ (\mathit{to}_{\mathsf{M}}\ (i+1)\ j)
\end{array}
$$

Figure 3: Monadic semantics for a subset of Icon

2.3 A monad of sequences

In order to handle sequences, some structure is needed in addition to the three generic monad operations displayed in Figure 1. We add three operations:

$$
\begin{array}{rcl}
\mathit{empty}_{\mathsf{M}} & : & \alpha\,\mathsf{M} \\
\mathit{if\_empty}_{\mathsf{M}} & : & \alpha\,\mathsf{M} \to \beta\,\mathsf{M} \to \beta\,\mathsf{M} \to \beta\,\mathsf{M} \\
\mathit{append}_{\mathsf{M}} & : & \alpha\,\mathsf{M} \to \alpha\,\mathsf{M} \to \alpha\,\mathsf{M}
\end{array}
$$

Here, $\mathit{empty}_{\mathsf{M}}$ stands for the empty sequence; $\mathit{if\_empty}_{\mathsf{M}}$ is a discriminator function that, given a sequence and two additional inputs, returns the first input if the sequence is empty, and returns the second input otherwise; $\mathit{append}_{\mathsf{M}}$ appends two sequences.

A straightforward instance of a monad of sequences is the list monad L, which is displayed in Figure 2; for lists, “join” is sometimes also called “flatten” or, in ML, “concat”.

2.4 A monadic semantics

A monadic semantics of the goal-directed language described in Section 2.1. is given in Figure 3. We explain the semantics in terms of the list monad. A literal i is interpreted as an expression that yields exactly one result; consequently, i is mapped into the singleton list $[i]$ using *unit*. The semantics of `to`, `+` and `<=` are given in terms of *bind2* and a function of type $\mathit{int} \to \mathit{int} \to \mathit{int}\,\mathsf{list}$. The type of function $\mathit{bind2}_{\mathsf{L}}$ is

$$(\alpha \to \beta \to \gamma\,\mathsf{list}) \to \alpha\,\mathsf{list} \to \beta\,\mathsf{list} \to \gamma\,\mathsf{list},$$

i.e., it takes two lists containing values of type α and β, and a function mapping $\alpha \times \beta$ into a list of values of type γ. The effect of the definition of $\mathit{bind2}_{\mathsf{L}}\ f\ \mathit{xs}\ \mathit{ys}$ is (1) to map $f\,x$ over ys for each x in xs and (2) to flatten the resulting list of lists. Both steps can be found in the example at the end of Section 2.1 of how the effect of goal-directed evaluation can be achieved in ML using lists.

2.5 A spectrum of semantics

In the following, we describe four possible instantiations of the semantics given in Figure 3. Because a semantics corresponds directly to an interpreter, we thus create four different interpreters.

2.5.1 A list-based interpreter

Instantiating the semantics with the list monad from Figure 2 yields a list-based interpreter. In an eager language such as ML, a list-based interpreter always computes all results. Such behavior may not be desirable in a situation where only the first result is of interest (or, for that matter, whether there exists a result): Consider for example the conditional, which examines whether a given expression yields at least one result or fails. An alternative is to use laziness.

2.5.2 A stream-based interpreter

Implementing the list monad from Figure 2 in a lazy language results in a monad of (finite) lazy lists; the corresponding interpreter generates one result at a time. In an eager language, this effect can be achieved by explicitly implementing a data type of streams, i.e., finite lists built lazily: a thunk is used to delay computation.

$$\alpha\,\mathsf{stream} \;\equiv\; \mathsf{End} \mid \mathsf{More\ of}\ (\alpha \times (\mathbf{1} \to \alpha\,\mathsf{stream}))$$

The definition of the corresponding monad operations is straightforward.

2.5.3 A continuation-based interpreter

Gudeman [13] gives a continuation-based semantics of a goal-directed language. We can derive this semantics by instantiating our monadic semantics with the continuation monad C as defined in Figure 4. The type-constructor $\alpha\,\mathsf{C}$ of the continuation monad is defined as $(\alpha \to R) \to R$, where R is called the *answer type* of the continuation.

A conceptual link between the list monad and the continuation monad with answer type $\beta\,\mathsf{list} \to \beta\,\mathsf{list}$ can be made through a Church encoding [4] of the higher-order representation of lists proposed by Hughes [16]. Hughes observed that when constructing the partially applied concatenation function $\lambda \mathit{ys}.\mathit{xs} \mathbin{@} \mathit{ys}$ rather than the list xs, lists can be appended in constant time. In the resulting representation, the empty list corresponds to the function that appends no elements, i.e., the identity, whereas the function that appends a single element is

Standard monad operations:

$$\begin{array}{lcl} \mathit{unit}_{\mathsf{C}}\ x & = & \lambda k.k\,x \\ \mathit{map}_{\mathsf{C}}\ f\ \mathit{xs} & = & \lambda k.\mathit{xs}\,(\lambda x.k\,(f\,x)) \\ \mathit{join}_{\mathsf{C}}\ \mathit{ls} & = & \lambda k.\mathit{ls}\,(\lambda x.x\,k) \end{array}$$

Special operations for sequences:

$$\begin{array}{lcl} \mathit{empty}_{\mathsf{C}} & = & \lambda k.\lambda l.l \\ \mathit{if\_empty}_{\mathsf{C}}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} & = & \lambda k.\lambda l.\mathit{xs}\,(\lambda\_.\lambda\_.\mathit{ys}\,k\,l)\,(\mathit{zs}\,k\,l) \\ \mathit{append}_{\mathsf{C}}\ \mathit{xs}\ \mathit{ys} & = & \lambda k.(\mathit{xs}\,k) \circ (\mathit{ys}\,k) \end{array}$$

Figure 4: The continuation monad

represented by a partially applied cons function:

$$\begin{array}{rcl} \mathit{nil} & = & \lambda \mathit{ys}.\mathit{ys} \\ \mathit{cons}\,x & = & \lambda \mathit{ys}.x :: \mathit{ys} \end{array}$$

Church-encoding a data types means abstracting over selector functions, in this case " :: ":

$$\begin{array}{rcl} \mathit{nil} & = & \lambda s_c.\lambda \mathit{ys}.\mathit{ys} \\ \mathit{cons}\,x & = & \lambda s_c.\lambda \mathit{ys}.s_c\,x\,\mathit{ys} \end{array}$$

The resulting representation of lists can be typed as

$$(\alpha \to \beta \to \beta) \to \beta \to \beta,$$

which indeed corresponds to $\alpha\,\mathsf{C}$ with answer type $\beta \to \beta$. Notice that *nil* and *cons* for this list representation yield $\mathit{empty}_{\mathsf{C}}$ and $\mathit{unit}_{\mathsf{C}}$, respectively. Similarly, the remaining monad operations correspond to the usual list operations.

Figure 5 displays the definition of $[\![\cdot]\!]_{\mathsf{C}}$ where all monad operations have been inlined and the resulting expressions β-reduced.

2.5.4 An interpreter with explicit success and failure continuations

A tail-recursive implementation of a continuation-based interpreter for Icon uses explicit success and failure continuations. The result of interpreting an Icon expression then has type

$$(\mathit{int} \to (\mathbf{1} \to \alpha) \to \alpha) \to (\mathbf{1} \to \alpha) \to \alpha,$$

where the first argument is the success continuation and the second argument the failure continuation. Note that the success continuation takes a failure continuation as a second argument. This failure continuation determines the resumption behavior of the Icon term: the success continuation may later on apply

$$\begin{array}{rcl}
[\![\cdot]\!]_{\mathsf{C}} & : & \mathit{Exp} \to (\mathit{int} \to \beta \to \beta) \to \beta \to \beta \\
\\
[\![i]\!]_{\mathsf{C}} & = & \lambda k.k\,i \\
[\![E_1\,\mathtt{to}\,E_2]\!]_{\mathsf{C}} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}}\,(\lambda j.\mathit{to}_{\mathsf{C}}\ i\ j\,k)) \\
[\![E_1\,\mathtt{+}\,E_2]\!]_{\mathsf{C}} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}}\,(\lambda j.k\,(i+j))) \\
[\![E_1\,\mathtt{<=}\,E_2]\!]_{\mathsf{C}} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}}\,(\lambda j.\mathit{leq}_{\mathsf{C}}\ i\ j\,k)) \\
[\![\mathtt{if}\,E_0\,\mathtt{then}\,E_1 & & \\
\mathtt{else}\,E_2]\!]_{\mathsf{C}_2} & = & \lambda k.\lambda l.[\![E_0]\!]_{\mathsf{C}_2}\,(\lambda\_.\lambda\_.[\![E_1]\!]_{\mathsf{C}_2}\,k\,l)\,([\![E_2]\!]_{\mathsf{C}_2}\,k\,l)
\end{array}$$

where

$$\begin{array}{rcl}
\mathit{leq}_{\mathsf{C}}\ i\ j & = & \lambda k.\textbf{if}\ i \le j\ \textbf{then}\ (k\ j)\ \textbf{else}\ (\lambda l.l) \\
\mathit{to}_{\mathsf{C}}\ i\ j & = & \lambda k.\textbf{if}\ i > j\ \textbf{then}\ (\lambda l.l) \\
 & & \qquad\qquad \textbf{else}\ (k\ i) \circ (\mathit{to}_{\mathsf{C}}\ (i+1)\ j\ k)
\end{array}$$

Figure 5: A continuation semantics

its failure continuation to generate more results. The corresponding continuation monad C_2 has the same standard monad operations as the continuation monad displayed in Figure 4, and the sequence operations

$$\begin{array}{lcl}
\mathit{empty}_{\mathsf{C}_2} & = & \lambda k.\lambda f.f\,() \\
\mathit{if\_empty}_{\mathsf{C}_2}\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} & = & \lambda k.\lambda f.\mathit{xs}\,(\lambda\_.\lambda\_.\mathit{zs}\,k\,f)\,(\lambda().\mathit{ys}\,k\,f) \\
\mathit{append}_{\mathsf{C}_2}\ \mathit{xs}\ \mathit{ys} & = & \lambda k.\lambda f.(\mathit{xs}\,k)(\lambda().\mathit{ys}\,k\,f)
\end{array}$$

Just as the continuation monad from Figure 4 can be conceptually linked to the list monad, the present continuation monad can be linked to the stream monad by a Church encoding of the data type of streams:

$$\begin{array}{lcl}
\mathit{end} & = & \lambda s_m.\lambda s_e.s_e() \\
\mathit{more}\,x\,\mathit{xs} & = & \lambda s_m.\lambda s_e.s_m\,x\,\mathit{xs}
\end{array}$$

The fact that the second component in a stream is a thunk suggests one to give the selector function s_m the type $\mathit{int} \to (\mathbf{1} \to \alpha) \to \beta$; the resulting type for end and $\mathit{more}\,x\,\mathit{xs}$ is then

$$(\mathit{int} \to (\mathbf{1} \to \alpha) \to \beta) \to (\mathbf{1} \to \beta) \to \beta.$$

Choosing α as the result type of the selector functions yields the type of a continuation monad with answer type $(\mathbf{1} \to \alpha) \to \alpha$.

The interpreter defined by the semantics $[\![\cdot]\!]_{\mathsf{C}_2}$ is the starting point of the semantics-directed compilation described in Section 3. Figure 6 displays the definition of $[\![\cdot]\!]_{\mathsf{C}_2}$ where all monad operations have been inlined and the resulting expressions β-reduced. Because the basic monad operations of C_2 are the same as those of C, the semantics based on C_2 and C only differ in the definitions of leq, to, and in how `if` is handled.

$$
\begin{array}{rcl}
[\![\cdot]\!]_{\mathsf{C}_2} & : & \mathit{Exp} \to (\mathit{int} \to (\mathbf{1} \to \alpha) \to \alpha) \to (\mathbf{1} \to \alpha) \to \alpha \\
\\
[\![i]\!]_{\mathsf{C}_2} & = & \lambda k.k\,i \\
[\![E_1\,\texttt{to}\,E_2]\!]_{\mathsf{C}_2} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}_2}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}_2}\,(\lambda j.\mathit{to}_{\mathsf{C}_2}\ i\ j\,k)) \\
[\![E_1\,\texttt{+}\,E_2]\!]_{\mathsf{C}_2} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}_2}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}_2}\,(\lambda j.k\,(i+j))) \\
[\![E_1\,\texttt{<=}\,E_2]\!]_{\mathsf{C}_2} & = & \lambda k.[\![E_1]\!]_{\mathsf{C}_2}\,(\lambda i.[\![E_2]\!]_{\mathsf{C}_2}\,(\lambda j.\mathit{leq}_{\mathsf{C}_2}\ i\ j\,k)) \\
[\![\texttt{if}\,E_0\,\texttt{then}\,E_1 & & \\
\texttt{else}\,E_2]\!]_{\mathsf{C}_2} & = & \lambda k.\lambda f.[\![E_0]\!]_{\mathsf{C}_2}\,(\lambda\_.\lambda\_.[\![E_1]\!]_{\mathsf{C}_2}\,k\,f)\,(\lambda().[\![E_2]\!]_{\mathsf{C}_2}\,k\,f)
\end{array}
$$

where

$$
\begin{array}{rcl}
\mathit{leq}_{\mathsf{C}_2}\ i\ j & = & \lambda k.\lambda f.\mathbf{if}\ i \leq j\ \mathbf{then}\ k\ j\ f\ \mathbf{else}\ f\,() \\
\mathit{to}_{\mathsf{C}_2}\ i\ j & = & \lambda k.\lambda f.\mathbf{if}\ i > j\ \mathbf{then}\ f\,() \\
 & & \qquad\qquad\quad \mathbf{else}\ (k\,i)\,(\lambda().\mathit{to}_{\mathsf{C}_2}\ (i+1)\ j\,k\,f)
\end{array}
$$

Figure 6: A semantics with success and failure continuations

2.6 Correctness

So far, we have related the various semantics presented in Section 2.5 only conceptually. Because the four different interpreters presented in Section 2.5 were created by instantiating one parameterized semantics with different monads, a *formal* correspondence proof can be conducted in a modular way building on the concept of a monad morphism [28].

Definition 1 (Monad morphism) *If* M *and* N *are two monads, then* $h : \alpha\,\mathsf{M} \to \alpha\,\mathsf{N}$ *is a* monad morphism *if it preserves the monad operations*[1]*, i.e.,*

$$
\begin{array}{rcl}
h \circ \mathit{unit}_{\mathsf{M}} & = & \mathit{unit}_{\mathsf{N}} \\
h \circ \mathit{map}_{\mathsf{M}}\,f & = & \mathit{map}_{\mathsf{N}}\,f \circ h \\
h \circ \mathit{join}_{\mathsf{M}} & = & \mathit{join}_{\mathsf{N}} \circ h \circ \mathit{map}_{\mathsf{M}}\,h \\
h\ \mathit{empty}_{\mathsf{M}} & = & \mathit{empty}_{\mathsf{N}} \\
h \circ \mathit{if\_empty}_{\mathsf{M}} & = & \lambda xs.\lambda ys.\lambda zs.\mathit{if\_empty}_{\mathsf{N}}(h\,xs)(h\,ys)(h\,zs) \\
h \circ \mathit{append}_{\mathsf{M}} & = & \lambda xs.\lambda ys.\mathit{append}_{\mathsf{N}}(h\,xs)(h\,ys)
\end{array}
$$

The following lemma shows that the semantics resulting from two different monad instantiations can be related by defining a monad morphism between the two sequence monads in question.

Lemma 2 *Let* M *and* N *be monads of sequences as specified in Section 2.3. If* h *is a monad morphism from* M *to* N*, then* $(h\,[\![E]\!]_{\mathsf{M}}) = [\![E]\!]_{\mathsf{N}}$ *for every Icon expression* E*.*

[1]We strengthen the definition of a monad morphism somewhat by considering a *sequence-preserving* monomorphism that also preserves the monad operations specific to the monad of sequences.

Proof: By induction over the structure of E. A lemma to the effect that $h\ (to_{\mathsf{M}}\ i\ j) = to_{\mathsf{N}}\, i\, j$ is shown by induction over $i - j$ for $i \geq j$. □

We use Lemma 2 to show that the list-based interpreter from Section 2.5.1 and the continuation-based interpreter from Section 2.5.3 always yield comparable results:

Proposition 3 *Let show* $: \alpha\,\mathsf{C} \to \alpha\,\mathsf{L}$ *be defined as*

$$\mathit{show}\ f \ = \ f\ (\lambda x.\lambda xs.\mathit{append}_{\mathsf{L}}\ (\mathit{unit}_{\mathsf{L}}\ x)\ xs)\ \mathit{empty}_{\mathsf{L}}.$$

Then $(\mathit{show}\ [\![E]\!]_{\mathsf{C}}) = [\![E]\!]_{\mathsf{L}}$ *for all Icon expressions* E.

Proof: We show that (1) $h\ :\ \alpha\,\mathsf{L} \to \alpha\,\mathsf{C}$, which is defined as

$$\begin{array}{rcl} h\ [\,] & = & \mathit{empty}_{\mathsf{C}} \\ h\ (x :: xs) & = & \mathit{append}_{\mathsf{C}}\ (\mathit{unit}_{\mathsf{C}}\ x)\ (h\ xs) \end{array}$$

is a monad morphism from L to C, and (2) the function $(\mathit{show} \circ h)$ is the identity function on lists. The proposition then follows immediately with Lemma 2. □

2.7 Conclusion

Taking an intuitive list-based semantics for a subset of Icon as our starting point, we have defined a stream-based semantics and two continuation semantics. Because our inital semantics is defined as the instantiation of a monadic semantics with a list monad, the other semantics can be defined through a stream monad and two different continuation monads, respectively. The modularity of the monadic semantics allows us to relate the semantics to each other by relating the corresponding monads, both conceptually and formally. To the best of our knowledge, the conceptual link between list-based monads and continuation monads via Church encoding has not been observed before.

It is known that continuations can be compiled into efficient code relatively easily [1, 14]; in the following section we show that partial evaluation is sufficient to generate efficient code from the the continuation semantics derived in Section 2.5.4.

3 Semantics-Directed Compilation

The goal of partial evaluation is to specialize a source program $p : S \times D \to R$ of two arguments to a fixed "static" argument $s : S$. The result is a residual program $p_s : D \to R$ that must yield the same result when applied to a "dynamic" argument d as the original program applied to both the static and the dynamic arguments, i.e., $[\![p_s(d)]\!] = [\![p(s, d)]\!]$.

Our interest in partial evaluation is due to its use in semantics-directed compilation: when the source program p is an interpreter and the static argument s

is a term in the domain of p then p_s is a compiled version of s represented in the implementation language of p. It is often possible to implement an interpreter in a functional language based on the denotational semantics.

Our starting point is a functional interpreter implementing the denotational semantics in Figure 6. The source language of the interpreter is shown in Figure 7. In Section 3.1 we present the Icon interpreter written in ML. In Section 3.1, 3.2, and 3.3 we use type-directed partial evaluation to specialize this interpreter to Icon terms yielding ML code, C code, and OCaml byte code as output. Other partial-evaluation techniques could be applied to yield essentially the same results.

```
structure Icon = struct
  datatype icon = LIT  of int
                | TO   of icon * icon
                | PLUS of icon * icon
                | LEQ  of icon * icon
                | IF   of icon * icon * icon
end
```

Figure 7: The abstract syntax of Icon terms

3.1 Type-directed partial evaluation

We have used type-directed partial evaluation to compile Icon programs into ML. This is a standard exercise in semantics-directed compilation using type-directed partial evaluation [9].

Type-directed partial evaluation is an approach to off-line specialization of higher-order programs [8]. It uses a normalization function to map the (value of the) trivially specialized program $\lambda d.p(s, d)$ into the (text of the) target program p_s.

The input to type-directed partial evaluation is a binding-time separated program in which static and dynamic primitives are separated. When implemented in ML, the source program is conveniently wrapped in a functor parameterized over a structure of dynamic primitives. The functor can be instantiated with evaluating primitives (for running the source program) and with residualizing primitives (for specializing the source program).

3.1.1 Specializing Icon terms using type-directed partial evaluation

In our case the dynamic primitives operations are addition (`add`), integer comparison (`leq`), a fixed-point operator (`fix`), a conditional functional (`cond`), and a quoting function (`qint`) lifting static integers into the dynamic domain. The signature of primitives is shown in Figure 8. For the residualizing primitives we let the partial evaluator produce functions that generate ML programs with meaningful variable names [8].

The parameterized interpreter is shown in Figure 9. The main function `eval` takes an Icon term and two continuations, `k` : `tint` $\rightarrow$ (`tunit` $\rightarrow$ `res`) $\rightarrow$ `res` and `f` : `tunit` $\rightarrow$ `res`, and yields a result of type `res`. We intend to specialize the interpreter to a static Icon term and keeping the continuation parameters `k` and `f` dynamic. Consequently, residual programs are parameterized over two continuations. (If the continuations were also considered static then the residual programs would simply be the list of the generated integers.)

```
signature PRIMITIVES = sig
  type tunit
  type tint
  type tbool
  type res

  val qint : int -> tint
  val add  : tint * tint -> tint
  val leq  : tint * tint -> tbool
  val cond : tbool * (tunit -> res) * (tunit -> res) -> res
  val fix  : ((tint -> res) -> tint -> res) -> tint -> res
end
```

Figure 8: Signature of primitive operations

The output of type-directed partial evaluation is the text of the residual program. The residual program is in long beta-eta normal form, that is, it does not contain any beta redexes and it is fully eta-expanded with respect to its type.

Example 4 *The following is the result of specializing the interpreter with respect to the Icon term* `10 + (4 to 7)`*.*

```
fn k => fn f =>
   fix (fn loop0 =>
           fn i0 =>
              cond (leq (i0, qint 7),
                    fn () => k (add (qint 10, i0))
                               (fn () => loop0 (add (i0, qint 1))),
                    fn () => f ()))
       (qint 4)
```

```
functor MakeInterp(P : PRIMITIVES) = struct
  fun loop (i, j) k f =
      P.fix
        (fn walk =>
            fn i =>
               P.cond (P.leq (i, j),
                       fn _ =>
                          k i (fn _ =>
                                  walk (P.add (i, P.qint 1))),
                       f))
        i

  fun select (i, j) k f =
      P.cond (P.leq (i, j), fn _ => k j f, f)

  fun sum (i, j) k = k (P.add (i, j))

  fun eval (LIT i)          k = k (P.qint i)
    | eval (TO(e1, e2))     k =
      eval e1 (fn i => eval e2 (fn j => loop (i, j) k))
    | eval (PLUS(e1, e2))   k =
      eval e1 (fn i => eval e2 (fn j => sum (i, j) k))
    | eval (LEQ(e1, e2))    k =
      eval e1 (fn i => eval e2 (fn j => select (i, j) k))
    | eval (IF(e1, e2, e3)) k =
      fn f =>
         eval e1
              (fn _ => fn _ => eval e2 k f)
              (fn _ => eval e3 k f)
end
```

Figure 9: Parameterized interpreter

3.1.2 Avoiding code duplication

The result of specializing the interpreter in Figure 9 may be exponentially large. This is due to the continuation parameter `k` being duplicated in the clause for `IF`. For example, specializing the interpreter to the Icon term `100 + (if 1 < 2 then 3 else 4)` yields the following residual program in which the context `add(100, ·)` occurs twice.

```
fn k => fn f =>
   cond (leq (qint 1, qint 2),
         fn () => k (add (qint 100, qint 3)) (fn () => f ()),
         fn () => k (add (qint 100, qint 4)) (fn () => f ()))
```

Code duplication is a well-known problem in partial evaluation [17]. The equally well-known solution is to bind the continuation in the residual program, just before it is used. We introduce a new primitive `save` of two arguments, `k` and `g`, which applies `g` to two "copies" of the continuation `k`.

```
signature PRIMITIVES = sig
  ...
  type succ = tint -> (tunit -> res) -> res
  val save : succ -> (succ * succ -> res) -> res
end
```

The final clause of the interpreter is modified to save the continuation parameter before it proceeds, as follows.

```
fun eval (LIT i)          k = k (P.qint i)
    ...
  | eval (IF(e1, e2, e3)) k =
    fn f =>
       save k
        (fn (k0, k1) => eval e1
                             (fn _ => fn _ => eval e2 k0 f)
                             (fn _ => eval e3 k1 f))
```

Specializing this new interpreter to the Icon term from above yields the following residual program in which the context `add(100, ·)` occurs only once.

```
fn k => fn f =>
   save (fn v0 =>
            fn resume0 =>
               k (add (qint 100, v0)) (fn () => resume0 ()))
        (fn (k0_0, k1_0) =>
            cond (leq (qint 1, qint 2),
                  fn () => k0_0 (qint 3) (fn () => f ()),
                  fn () => k1_0 (qint 4) (fn () => f ())))
```

Two copies of continuation parameter `k` are bound to `k0_0` and `k1_0` before the continuation is used (twice, in the body of the second lambda). In order just to prevent code duplication, passing one "copy" of the continuation parameter is actually enough. But the translation into C introduced in Section 3.2 uses the two differently named variables, in this case `k0_0` and `k1_0`, to determine the `IF`-branch inside which a continuation is applied.

3.2 Generating C programs

Residual programs are not only in long beta-eta normal form. Their type

$$(\texttt{tint} \to (\texttt{tunit} \to \texttt{res}) \to \texttt{res}) \to (\texttt{tunit} \to \texttt{res}) \to \texttt{res}$$

imposes further restrictions: A residual program must take two arguments, a success continuation `k` : `tint` $\rightarrow$ (`tunit` $\rightarrow$ `res`) $\rightarrow$ `res` and a failure continuation `f` : `tunit` $\rightarrow$ `res`, and it must produce a value of type `res`. When we also consider the types of the primitives that may occur in residual programs we see that values of type `res` can only be a result of

- applying the success continuation `k` to an integer n and function of type `tunit` $\rightarrow$ `res`;
- applying the failure continuation `f`;
- applying the primitive `cond` to a boolean and two functions of type `tunit` $\rightarrow$ `res`;
- applying the primitive `fix` to a function of two arguments, $\texttt{loop}_n$: `tint` $\rightarrow$ `res` and $\texttt{i}_n$: `tint`, and an integer;
- (inside a function passed to `fix`) applying the function $\texttt{loop}_n$ to an integer;
- applying the primitive `save` to two arguments, the first being a function of two arguments, $\texttt{v}_n$: `tint` and $\texttt{resume}_n$: `tunit` $\rightarrow$ `res`, and the second being a function of a pair of arguments, $\texttt{k}_n^0$ and $\texttt{k}_n^1$, each of type `tint` $\rightarrow$ (`tunit` $\rightarrow$ `res`) $\rightarrow$ `res`;
- (inside the first function passed to `save`) applying the function $\texttt{resume}_n$; or
- (inside the second function passed to `save`) applying one of the functions $\texttt{k}_n^0$ or $\texttt{k}_n^1$ to an integer and a function of type `tunit` $\rightarrow$ `res`.

A similar analysis applies to values of type `tint`: they can only arise from evaluating an integer n, a variable $\texttt{i}_n$, or a variable $\texttt{v}_n$ or from applying `add` to two argument of type `tint`. As a result, we observe that the residual programs of specializing the Icon interpreter using type-directed partial evaluation are restricted to the grammar in Figure 10. (The restriction that the variables $\texttt{loop}_n$, $\texttt{i}_n$, $\texttt{v}_n$, and $\texttt{resume}_n$ each must occur inside a function that binds them cannot be expressed using a context-free grammar. This is not a problem for our development.) We have expressed the grammar as an ML datatype and used this datatype to represent the output from type-directed partial evaluation. Thus, we have essentially used the type system of ML as a theorem prover to show the following lemma.

Lemma 5 *The residual program generated from applying type-directed partial evaluation to the interpreter in Figure 9 can be generated by the grammar in Figure 10.*

The idea of generating grammars for residual programs has been studied by, e.g., Malmkjær [20] and is used in the run-time specializer Tempo to generate code templates [6].

$$
\begin{array}{rcl}
I & ::= & \texttt{fn k => fn f => } S \\
S & ::= & \texttt{k } E \texttt{ (fn () => } S\texttt{)} \\
 & | & \texttt{f ()} \\
 & | & \texttt{cond (}E\texttt{, fn () => }S\texttt{, fn () => }S\texttt{)} \\
 & | & \texttt{fix (fn loop}_n\texttt{ => fn i}_n\texttt{ => }S\texttt{) }E \\
 & | & \texttt{loop}_n\ E \\
 & | & \texttt{save (fn v}_n\texttt{ => fn resume}_n\texttt{ => }S\texttt{) (fn (k}_n^0\texttt{, k}_n^1\texttt{) => }S\texttt{)} \\
 & | & \texttt{resume}_n\texttt{ ()} \\
 & | & \texttt{k}_n^i\ E\texttt{ (fn () => }S\texttt{)},\quad \text{where } i \in \{0,1\} \\
E & ::= & \texttt{qint } n \mid \texttt{i}_n \mid \texttt{v}_n \mid \texttt{add (}E\texttt{, }E\texttt{)} \mid \texttt{leq (}E\texttt{, }E\texttt{)}
\end{array}
$$

Figure 10: Grammar of residual programs

The simple structure of output programs allows them to be viewed as programs of a flow-chart language. We choose C as a concrete example of such a language. Figure 11 and 12 show the translation from residual programs to C programs.

The translation replaces function calls with jumps. Except for the call to $\texttt{resume}_n$ (which only occurs as the result of compiling if-statements), the name of a function uniquely determines the corresponding label to jump to. Jumps to $\texttt{resume}_n$ can end up in two different places corresponding to the two copies of the continuation. We use a boolean variable $\texttt{gate}_n$ to distinguish between the two possible destinations. Calls to $\texttt{loop}_n$ and $\texttt{k}_n$ pass arguments. The names of the formal parameters are known ($\texttt{i}_n$ and $\texttt{v}_n$, respectively) and therefore arguments are passed by assigning the variable before the jump.

In each translation of a conditional a new label l must be generated. The entire translated term must be wrapped in a context that defines the labels `succ` and `fail` (corresponding to the initial continuations). The statements following the label `succ` are allowed to jump to `resume`. The translation in Figure 11 and 12 generates a C program that successively prints the produced integers one by one. A lemma to the effect that the translation from residual ML programs into C is semantics preserving would require giving semantics to C and to the subset of ML presented in Figure 10 and then showing equivalence.

Example 6 *Consider again the Icon term* `10 + (4 to 7)` *from Example 4. It is translated into the following C program.*

```
        i0 = 4;
loop0:  if (i0 <= 7) goto L0;
        goto fail;

L0:     value = 10 + i0;
        goto succ;
```

$$|\texttt{fn k => fn f => }S|_{\mathrm{I}} = \begin{cases} & |S|_{\mathrm{S}} \\ \texttt{succ:} & \texttt{printf("\%d ", value);} \\ & \texttt{goto resume;} \\ \texttt{fail:} & \texttt{printf("\textbackslash n");} \\ & \texttt{exit(0);} \end{cases}$$

$$|\texttt{k }E\texttt{ (fn () => }S\texttt{)}|_{\mathrm{S}} = \begin{cases} & \texttt{value = }|E|_{\mathrm{E}}\texttt{;} \\ & \texttt{goto succ;} \\ \texttt{resume:} & |S|_{\mathrm{S}} \end{cases}$$

$$|\texttt{f ()}|_{\mathrm{S}} = \begin{cases} \texttt{goto fail;} \end{cases}$$

$$|\texttt{cond (}E\texttt{, fn () => }S\texttt{, fn () => }S'\texttt{)}|_{\mathrm{S}} = \begin{cases} & \texttt{if (}|E|_{\mathrm{E}}\texttt{) goto }l\texttt{;} \\ & |S'|_{\mathrm{S}} \\ l\texttt{:} & |S|_{\mathrm{S}} \end{cases}$$

$$|\texttt{fix (fn loop}_n\texttt{ => fn i}_n\texttt{ => }S\texttt{) }E|_{\mathrm{S}} = \begin{cases} & \texttt{i}_n\texttt{ = }|E|_{\mathrm{E}}\texttt{;} \\ \texttt{loop}_n\texttt{:} & |S|_{\mathrm{S}} \end{cases}$$

$$|\texttt{loop}_n\ E|_{\mathrm{S}} = \begin{cases} \texttt{i}_n\texttt{ = }|E|_{\mathrm{E}}\texttt{;} \\ \texttt{goto loop}_n\texttt{;} \end{cases}$$

$$\left|\begin{array}{r}\texttt{save (fn v}_n\texttt{ => fn resume}_n\texttt{ => }S\texttt{)} \\ \texttt{(fn (k}^0_n\texttt{, k}^1_n\texttt{) => }S'\texttt{)}\end{array}\right|_{\mathrm{S}} = \begin{cases} & |S'|_{\mathrm{S}} \\ \texttt{succ}_n\texttt{:} & |S|_{\mathrm{S}} \end{cases}$$

$$|\texttt{resume}_n\texttt{ ()}|_{\mathrm{S}} = \begin{cases} \texttt{if (gate}_n\texttt{) goto resume}^1_n\texttt{;} \\ \texttt{goto resume}^0_n\texttt{;} \end{cases}$$

$$|\texttt{k}^i_n\ E\texttt{ (fn () => }S\texttt{)}|_{\mathrm{S}} = \begin{cases} & \texttt{gate}_n\texttt{ = }i\texttt{;} \\ & \texttt{v}_n\texttt{ = }|E|_{\mathrm{E}}\texttt{;} \\ & \texttt{goto succ}_n\texttt{;} \\ \texttt{resume}^i_n\texttt{:} & |S|_{\mathrm{S}} \end{cases}$$

Figure 11: Translating residual programs into C (Statements)

$$
\begin{array}{rcl}
|\texttt{qint}\ n|_{\mathrm{E}} & = & n \\
|\texttt{i}_n|_{\mathrm{E}} & = & \texttt{i}_n \\
|\texttt{v}_n|_{\mathrm{E}} & = & \texttt{v}_n \\
|\texttt{add (}E\texttt{, }E'\texttt{)}|_{\mathrm{E}} & = & |E|_{\mathrm{E}}\ \texttt{+}\ |E'|_{\mathrm{E}} \\
|\texttt{leq (}E\texttt{, }E'\texttt{)}|_{\mathrm{E}} & = & |E|_{\mathrm{E}}\ \texttt{<=}\ |E'|_{\mathrm{E}}
\end{array}
$$

Figure 12: Translating residual programs into C (Expressions)

```
resume: i0 = i0 + 1;
        goto loop0;

succ:   printf("%d ", value);
        goto resume;

fail:   printf("\n");
        exit(0);
```

The C target programs corresponds to the target programs of Proebsting's optimized template-based compiler [22]. In effect, we are automatically generating flow-chart programs from the denotation of an Icon term.

3.3 Generating byte code

In the previous two sections we have developed two compilers for Icon terms, one that generates ML programs and one that generates flow-chart programs. In this section we unify the two by composing the first compiler with the third author's automatic run-time code generation system for OCaml [25] and by composing the second compiler with a hand-written compiler from flow charts into OCaml byte code.

3.3.1 Run-time code generation in OCaml

Run-time code generation for OCaml works by a deforested composition of traditional type-directed partial evaluation with a compiler into OCaml byte code. Deforestation is a standard improvement in run-time code generation [6, 19, 26]. As such, it removes the need to manipulate the text of residual programs at specialization time. As a result, instead of generating ML terms, run-time code generation allows type-directed partial evaluation to directly generate executable OCaml byte code.

Specializing the Icon interpreter from Figure 9 to the Icon term `10 + (4 to 7)` using run-time code generation yields a residual program of about 110 byte-code instructions in which functions are implemented as closures and calls are implemented as tail-calls. (Compiling the residual ML program using the OCaml compiler yields about 90 byte-code instructions.)

3.3.2 Compiling flow charts into OCaml byte code

We have modified the translation in Figure 11 and 12 to produce OCaml byte-code instructions instead of C programs. The result is an embedding of Icon into OCaml.

Using this compiler, `10 + (4 to 7)` yields 36 byte-code instructions in which functions are implemented as labelled blocks and calls are implemented as an assignment (if an argument is passed) followed by a jump. This style of target code was promoted by Steele in the first compiler for Scheme [27].

3.4 Conclusion

Translating the continuation-based denotational semantics into an interpreter written in ML and using type-directed partial evaluation enables a standard semantics-directed compilation from Icon terms into ML. A further compilation of residual programs into C yields flow-chart programs corresponding to those produced by Proebsting's Icon compiler [22].

4 Conclusions and Issues

Observing that the list monad provides the kind of backtracking embodied in Icon, we have specified a semantics of Icon that is parameterized by this monad. We have then considered alternative monads and proven that they also provide a fitting semantics for Icon. Inlining the continuation monad, in particular, yields Gudeman's continuation semantics [13].

Using partial evaluation, we have then specialized these interpreters with respect to Icon programs, thereby compiling these programs using the first Futamura projection. We used a combination of type-directed partial evaluation and code generation, either to ML, to C, or to OCaml byte code. Generating code for C, in particular, yields results similar to Proebsting's compiler [22].

Gudeman [13] shows that a continuation semantics can also deal with additional control structures and state; we do not expect any difficulties with scaling up the code-generation accordingly. The monad of lists, on the other hand, does not offer enough structure to deal, e.g., with state. It should be possible, however, to create a rich enough monad by combining the list monad with other monads such as the state monad [10, 18].

It is our observation that the traditional (in partial evaluation) generalization of the success continuation avoids the code duplication that Proebsting presents as problematic in his own compiler. We are also studying the results of defunctionalizing the continuations, à la Reynolds [24], to obtain stack-based specifications and the corresponding run-time architectures.

Acknowledgments: Thanks are due to the anonymous referees for comments and to Andrzej Filinski for discussions. This work is supported by the ESPRIT Working Group APPSEM (`http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`).

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[2] Lawrence Byrd. Understanding the control of Prolog programs. Technical Report 151, University of Edinburgh, 1980.

[3] Mats Carlsson. On implementing Prolog in functional programming. *New Generation Computing*, 2(4):347–359, 1984.

[4] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[5] 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.

[6] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In Guy L. Steele, editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, January 1996. ACM Press.

[7] Ron K. Cytron, editor. *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, Las Vegas, Nevada, June 1997. ACM Press.

[8] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[9] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Springer-Verlag. Extended version available as the technical report BRICS-RS-96-13.

[10] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[11] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Prentice Hall, Inc., 1983.

[12] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language.* Princeton University Press, 1986.

[13] David A. Gudeman. Denotational semantics of a goal-directed language. *ACM Transactions on Programming Languages and Systems*, 1992.

[14] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[15] Ralf Hinze. Prological features in a functional setting—axioms and implementations. In Masahiko Sato and Yoshihito Toyama, editors, *Third Fuji International Symposium on Functional and Logic Programming (FLOPS'98)*, pages 98–122, Kyoto, Japan, April 1998. World Scientific.

[16] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[17] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[18] David J. King and Philip Wadler. Combining Monads. In John Launchbury and Patrick M. Sansom, editors, *Glasgow Workshop on Functional Programming*, Workshops in Computing, Ayr, Scotland, 1992. Springer, Berlin.

[19] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 137–148. ACM Press, May 1996.

[20] Karoline Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms.* PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, March 1993.

[21] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[22] Todd A. Proebsting. Simple translation of goal-directed evaluation. In Cytron [7], pages 1–6.

[23] Todd A. Proebsting and Gregg M. Townsend. A new implementation of the Icon language. Technical Report 99-13, University of Arizona, Department of Computer Science, 1999.

[24] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[25] Morten Rhiger. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, 2001. Forthcoming.

[26] Michael Sperber and Peter Thiemann. Two for the price of one: composing partial evaluation and compilation. In Cytron [7], pages 215–225.

[27] Guy L. Steele Jr. Compiler optimization based on viewing LAMBDA as RENAME + GOTO. In Patrick Henry Winston and Richard Henry Brown, editors, *Artificial Intelligence: An MIT Perspective*, volume 2. The MIT Press, 1979.

[28] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992.

[29] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 24–52. Springer-Verlag, 1995.

[30] Richard S. Wallace. An easy implementation of pil (PROLOG in LISP). *Association for Computing Machinery Special Interest Group on Artificial Intelligence. SIGART NEWSL.*, (85):29–32, July 1983.