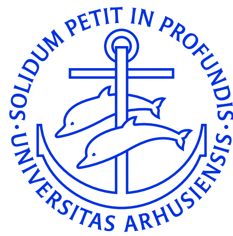
Run-Time Code Generation for Type-Directed Partial Evaluation

Morten Rhiger

Progress Report



BRICS Ph.D. School
Department of Computer Science
University of Aarhus
Denmark

December 1999 Supervisor: Olivier Danvy

Abstract

This Ph.D. progress report documents a combination of partial evaluation and compilation that enables “just-in-time program specialization”. To this end, we have composed a type-directed partial evaluator for OCaml programs with a run-time code generator for the OCaml virtual machine. The composition is deforested, i.e., residual programs are directly expressed as byte code, which is then dynamically loaded. The implementation seamlessly extends OCaml with two new keywords for generating code dynamically. The system has been applied to traditional examples of partial evaluation and run-time code generation, such as the first Futamura projection for Action Semantics and the BSD Packet Filter.

Contents

1 Introduction **1**
1.1 Partial evaluation . . . 1
1.1.1 What . . . 1
1.1.2 How . . . 2
1.1.3 Type-directed partial evaluation . . . 3
1.2 Run-time code generation . . . 3
1.2.1 What . . . 3
1.2.2 How . . . 4
1.2.3 Naive run-time code generation in Scheme . . . 4
1.3 Run-time program specialization . . . 5
1.4 This work . . . 6
1.5 This report . . . 6

2 Type-directed partial evaluation **7**
2.1 Binding-time coercions . . . 7
2.2 Normalization by evaluation . . . 8
2.3 Type-directed partial evaluation . . . 9
2.4 Example: The power function (1) . . . 11
2.5 Conclusions . . . 13

3 Run-time code generation for type-directed partial evaluation **14**
3.1 Design: New expressions . . . 14
3.2 Types of the new expressions . . . 14
3.3 Implementation . . . 15
3.3.1 Overview of the OCaml byte-code compiler . . . 16
3.3.2 The generating extension of type-directed partial evaluation . . . 17
3.3.3 Extensions to the byte-code generator . . . 18
3.4 Example: The power function (2) . . . 19
3.5 Conclusions . . . 20

4 Example: The BSD packet filter **21**
4.1 Network packet filters . . . 21
4.2 Compiling packet filters . . . 21
4.3 Benchmarks . . . 22
4.4 Conclusions . . . 23

5 Conclusion and future work **24**

Chapter 1

Introduction

In this report we present the design and implementation of a run-time code generator for type-directed partial evaluation in OCaml, a statically typed, modular, and higher-order language [29]. This current chapter gives an introduction to standard concepts of partial evaluation and run-time code generation and provides an overview of related work. Chapter 2 presents type-directed partial evaluation. In general, this is background knowledge but we emphasize on the parts that differ from Chapter 3. Chapter 3 describes the development of the run-time code generator which is the new material contributed by this report. In Chapter 4 we consider *just-in-time* compilation of BSD packet filters using the run-time code generator. Chapter 5 concludes and gives directions for future investigations.

Given a language L we let $\tau\ \mathsf{prog}_L$ denote the type of the representation of L-programs of type τ. The *semantic valuation* $[\![\cdot]\!]_L : \tau\ \mathsf{prog}_L \to \tau$ is the (partial, mathematical) function that gives the semantics of an L-program. (We omit the subscript from prog and $[\![\cdot]\!]$ when there are no ambiguities.)

1.1 Partial evaluation

1.1.1 What

Partial evaluation is a technique for specializing programs [24]. Given a partial evaluator

$$\mathsf{PE} : ((S \times D \to R)\ \mathsf{prog} \times S \to (D \to R)\ \mathsf{prog})\ \mathsf{prog}$$

applying $[\![\mathsf{PE}]\!]$ to a program $\mathsf{p} : (S \times D \to R)\ \mathsf{prog}$ of two inputs and a value $s : S$ gives a *specialized* program (also called the *residual* program) $\mathsf{p}_s : (D \to R)\ \mathsf{prog}$ that behaves as p with fixed input s, provided that partial evaluation terminates:

$$\begin{aligned} [\![\mathsf{PE}]\!](\mathsf{p}, s) &= \mathsf{p}_s \\ [\![\mathsf{p}_s]\!](d) &= [\![\mathsf{p}]\!](s, d) \end{aligned}$$

Kleene's S^m_n theorem establishes the existence of computable partial evaluators but not their advantage: The residual program p_s is often smaller than the original source program p, and running p_s on input d is often more efficient than running p on input (s, d). Nor does the theorem state how to implement a partial evaluator that exploits these advantages. This has been the source of much research work during the last two decades. Implementations of partial evaluators have successfully been applied in such areas as pattern matching, ray tracing, and operating systems [7]. However, the most fascinating applications of partial evaluation are perhaps those of the Futamura projections: Let

intp be an interpreter for a language S written in a language T

$$\mathsf{intp} : ((I \to R)\ \mathsf{prog}_S \times I \to R)\ \mathsf{prog}_T$$

and let src be a program written in S. Specializing intp with respect to src amounts to *compiling* src from S to T. (This is called the *first Futamura projection*.) Furthermore, if the partial evaluator is *self-applicable* then specializing the partial evaluator itself with respect to the interpreter intp amounts to *generating a compiler* from S to T. (This is called the *second Futamura projection*.) Finally, still assuming a self-applicable partial evaluator, specializing the partial evaluator with respect to itself amounts to *generating a compiler generator*. (This is called the *third Futamura projection*.) That is

$[\![\mathsf{PE}]\!](\mathsf{intp}, \mathsf{src})$	$=$	target	Generate target program	(First Futamura projection)
$[\![\mathsf{PE}]\!](\mathsf{PE}, \mathsf{intp})$	$=$	comp	Generate compiler	(Second Futamura projection)
$[\![\mathsf{PE}]\!](\mathsf{PE}, \mathsf{PE})$	$=$	cogen	Generate compiler generator	(Third Futamura projection)

The correctness criteria is

$$\begin{aligned} [\![\mathsf{target}]\!]_T(i) &= [\![\mathsf{src}]\!]_S(i) = [\![\mathsf{intp}]\!]_T(\mathsf{src}, i) \\ [\![\mathsf{comp}]\!](\mathsf{src}) &= \mathsf{target} \\ [\![\mathsf{cogen}]\!](\mathsf{intp}) &= \mathsf{comp} \end{aligned}$$

(Yoshihiko Futamura's insightful original manuscript on this subject has recently been reprinted in Higher-Order and Symbolic Computation [21].)

1.1.2 How

As its name might suggest, partial evaluation of a source program p with respect to input s works by evaluating those parts of p that only depend on s (these are called the *static* parts) and re-creating those parts that may depend on on the remaining input d (these are called the *dynamic* parts). An *offline* partial evaluator first approximates the binding times of the source program, that is, which parts only depend on s and which parts might depend on d. The binding-time information then directs the specialization phase, which builds the residual program. Specialization amounts to *reduce* the static program parts and *residualize* the dynamic parts. In contrast, an *online* partial evaluator determines on the fly whether to reduce or residualize. The online approach is more precise than the offline approach because binding-time information is necessarily approximative. On the other hand, the online approach imposes an interpretive overhead at specialization time. It is generally believed that offline partial evaluation is necessary to obtain good results from self application. (In fact, offline partial evaluation was developed for self application [25].)

A partial evaluator for a language with functions is said to perform *polyvariant specialization* if it can produce several versions of each source function. In contrast, *monovariant specialization* produces at most one version of each source function. A *polyvariant binding-time analysis* handles several versions of each source function, one for each binding-time signature, that is, assignment of binding-time to the function arguments. In contrast, *monovariant binding-time analysis* chooses the conservative "dynamic" when merging two functions with different binding-time signature and is therefore less flexible.

The *generating extension* of a program $\mathsf{p} : (S \times D \to R)\ \mathsf{prog}$ of two inputs is a program $\mathsf{pgen} : (S \to (D \to R)\ \mathsf{prog})\ \mathsf{prog}$ that when applied to one argument generates a program that

behaves as the original program with the first argument fixed.

$$\begin{aligned} [\![\mathsf{pgen}]\!](s) &= \mathsf{p}_s \\ [\![\mathsf{p}_s]\!](d) &= [\![\mathsf{p}]\!](s,d) \end{aligned}$$

The generating extension of a program p can be obtained by self application of a partial evaluator by using the second Futamura projection

$$\mathsf{pgen} = [\![\mathsf{PE}]\!](\mathsf{PE}, \mathsf{p})$$

Sometimes it is easier to code the generating extension by hand. This is particularly easy when the source program is binding-time annotated. Then static expressions are left unchanged while dynamic expressions are changed into program-generating expressions.

1.1.3 Type-directed partial evaluation

Type-directed partial evaluation is an approach to offline, monovariant program specialization for higher-order programs [9, 11]. When the source program is closed, the binding-time information that directs specialization is given by the type of the source program. Thus, binding-time analysis amounts to type analysis. When the source program contains free dynamic variables, type-directed partial evaluation is more involved as we will see in Chapter 2.

In type-directed partial evaluation, the source program is represented by (possibly higher-order) values and the result is usually represented by a datatype which can be pretty-printed into the text of the residual program. This contrasts with traditional, syntax-directed partial evaluators like Similix [3], Schism [6], Tempo [8], and Mix [25], which all represent both source and residual programs as a datatype.

Type-directed partial evaluation uses a normalization function nbe to achieve specialization. (The name of the function stands for “normalization by evaluation”.) Representing a closed source program as a function $\mathsf{p} : S \times D \to R$ and given a static value $s : S$, the (text of the) residual program is obtained by $\mathsf{nbe}(\lambda d.\, \mathsf{p}(s,d))$. Again, when the source program contains free dynamic variables, the situation is more involved.

Type-directed partial evaluation has been stated and formalized in a call-by-name setting [20] and it has also been adopted to a call-by-value setting. Type-directed partial evaluation is also extended with online features, such as primitive operations that probe their arguments for static reduction opportunities at specialization time [10]. Type-directed partial evaluation has been implemented in Scheme [9], ML [42], and Haskell [37, 38].

1.2 Run-time code generation

1.2.1 What

Run-time code generation is a term that covers a diversity of techniques enabling programs to generate and execute code dynamically. The primary reason for using run-time code generation is that of *efficiency*: A traditional compiler exploits information that is invariant at compile time whereas a run-time code generator can also exploit information which becomes invariant at run time.

1.2.2 How

A run-time code generator is able to create and manipulate representations of program fragments, called *code templates*, at run time. Eventually, code templates are combined into a complete program, compiled and linked, and finally executed. A code template contains a list of instructions (or a way of computing such a list) corresponding to the program fragment. Code templates for higher-level languages also contain information about how to combine two code templates. For example, a code template may contain information about its free variables such that one code template can bind variables that are used in another code template.

The simplest representation of code templates is as syntax trees. (See Section 1.2.3 below for an example in Scheme.) A syntax tree contains all possible information about a program fragment. This representation has the advantage that when the code template is compiled and linked, standard optimization techniques can be applied. However, it requires a complete compiler to be available at run time. A more efficient representation of code templates is as a sequence of instructions with "holes" for the values of the free variables plus some relocation information. Compiling and linking such a code template amounts to filling the holes, writing the result into memory, and adjusting the result according to the relocation information.

The compiler decides where and when to do run-time code generation. This decision can be taken automatically or it can be controlled by the programmer. In any case, the result is a binding-time separation of the program. This tells the compiler when code should be generated at run time and how the code will depend on run-time values.

1.2.3 Naive run-time code generation in Scheme

As an example of a naive implementation of run-time code generation we consider a Scheme system compliant with the R^5RS [26] (the latest report on Scheme), here Chez Scheme [18]. Such systems provide a procedure `eval` that compiles and runs a program represented by an S-expression.

```
> (* 10 (eval '(+ 1 2)))
30
>
```

This allows for run-time code generation using syntax trees as a representation of code templates, as outlined above. As an example, Figure 1.1 presents two procedures in Scheme, `pow`, which computes the power function, and `powgen`, which computes the generating extension of `pow` using *quote* and *unquote* [2]. The generating extension returns a specialized version of `pow` given as a syntax tree. One can view `powgen` as a binding-time annotated version of `pow` with respect to a binding-time signature where `n` is static and `x` is dynamic. Applying `powgen` to a positive integer yields the syntax tree of a specialized version of the power function which can be compiled using `eval`.

```
> ((pow 5) 2)
32
> (define pow5-exp (powgen 5))
> pow5-exp
(lambda (g1404)
  (* g1404 (* g1404 (* g1404 (* g1404 (* g1404 1))))))
> (define pow5 (eval pow5-exp))
> pow5
#<procedure>
> (pow5 2)
```

```
(define (pow n)
  (lambda (x)
    (let loop ([n n])
      (if (zero? n)
          1
          (* x (loop (- n 1)))))))

(define (powgen n)
  (let ([x (gensym)])
    `(lambda (,x)
       ,(let loop ([n n])
          (if (zero? n)
              `1
              `(* ,x ,(loop (- n 1))))))))
```

Figure 1.1: Naive run-time code generation in Scheme

```
32
>
```

Running the specialized procedure `pow5` is about twice as fast as running the general power function for small numbers. The *amortization factor* is the minimum number of times the specialized program should be run in order for specialization to pay for itself. In this example the amortization factor is about 26.

1.3 Run-time program specialization

As we have seen above, run-time code generation amounts to specializing a program with respect to values that become available at run time. When using an offline technique, a binding-time annotation of the program that should be specialized is needed. In the example above, the binding-time annotation was given by the programmer in the form of the generating extension using quote an unquote. A similar idea is implemented in 'C [19], an extension of ANSI C that supports run-time code generation.

Alternatively, standard partial evaluation techniques can be used to obtain the generating extension automatically from a source procedure. For example, Mark Leone and Peter Lee's Fabius system is a compiler for a subset of ML [27, 28] that produces generating extensions of curried functions. Applying a curried function to one argument triggers run-time code generation. The result is a function of the rest of the arguments which is specialized with respect to the given first argument. Fabius generates native code.

In other systems the use of partial evaluation is explicit. For example, Vincent Balat and Olivier Danvy have combined type-directed partial evaluation with run-time code generation in OCaml, requiring the user to specialize top-level bindings. Type-directed partial evaluation produces the source code of the specialized value, which is then compiled into OCaml byte code by a fast special-purpose compiler [1].

Another approach is to have executable code generated directly by partial evaluation. Michael Sperber and Peter Thiemann combine a syntax-directed partial evaluator for Scheme with a Scheme-compiler that generates byte-code for a virtual machine [39]. They use standard partial evaluation

techniques to *deforest* the intermediate representation of programs as text. The result enables run-time code generation of Scheme into byte code.

Tempo [8] is an offline partial evaluator for C programs that is able of doing both compile-time and run-time specialization. The binding-time analysis includes alias, side-effect, and dependency analyses. From the binding-time analysis, a dedicated run-time specializer is generated along with code templates. At run-time, the holes in the code templates are filled with actual values and the code templates are linked and executed.

1.4 This work

We present the design and implementation of a run-time code generator for type-directed partial evaluation in OCaml. Our solution is a deforested composition of type-directed partial evaluation and byte-code compilation. We extend OCaml with one directive for building primitive functions in source programs and one directive for specializing source programs at run time. Both directives can be placed anywhere in a program, not only at the top level. Because OCaml is statically typed and infers types of expressions automatically and because binding-time information coincides with type information there is (1) no need to hand-code generating extensions and (2) no need for a complicated binding-time analysis.

Let us briefly compare our work to previous work, pointwise:

‘C and Tempo: We consider OCaml, not C, and thus have the safety of a statically typed language. Type-directed partial evaluation is monovariant, whereas Tempo is polyvariant. Like Tempo, type-directed partial evaluation gives both the ability to do specialization at compile time and our system also gives the ability to do specialization at runtime.

Fabius: The source language of Fabius is a 1st-order subset of pure ML; the target language is actual assembly code. We consider all of ML as the source language and byte code as the target language.

Balat and Danvy: We consider a deforested composition of type-directed partial evaluation and code generation. Our normalization directive can be placed anywhere, not just at the top level.

Sperber and Thiemann: We consider type-directed partial evaluation for OCaml, which is statically typed. Sperber and Thiemann considers a syntax-directed partial evaluator for Scheme, which is dynamically typed.

1.5 This report

The rest of this report is structured as follows. Chapter 2 presents an implementation of “conventional” type-directed partial evaluation in OCaml and an application to the power function. Chapter 3 presents the design and implementation of a run-time code generator for type-directed partial evaluation in OCaml. For comparison, this chapter also contains an application of run-time code generation to the power function. Chapter 4 presents an application of run-time code generation to the BSD packet filter. Chapter 5 concludes and gives directions for future work.

Chapter 2

Type-directed partial evaluation

In this chapter we give an overview of type-directed partial evaluation and its origin. We first consider a way of improving the result of partial evaluation and the we see how this leads to a normalization function and type-directed partial evaluation. We present an implementation of type-directed partial evaluation in OCaml and point out some differences between this implementation and the usual Standard ML implementation. Finally, we apply type-directed partial evaluation in OCaml to the canonical example of the power function. A more detailed introduction to type-directed partial evaluation can be found in Danvy's DIKU lecture notes [11].

2.1 Binding-time coercions

Binding-time improvements are often necessary to obtain good results from partial evaluation. Consider the program

$$\lambda d.\, \lambda h.\, \mathsf{let}\ f = \lambda g.\, g\ (g\, d)\ \mathsf{in}\ (f\ (\lambda x.\, x), f\ h) \tag{2.1}$$

where d and h are dynamic. Since f is applied to both a static and a dynamic argument a monovariant binding-time analysis will assign the most conservative binding-time to f, namely "dynamic". Specialization will therefore residualize the first application of f instead of reducing it. For example, using Similix to specialize the program gives

$$\lambda d.\, \lambda h.\, \mathsf{let}\ g = \lambda x.\, x\ \mathsf{in}\ (g\ (g\, 1), h\ (h\, d))$$

which contains two redexes. We can obtain a better result by *eta-expanding* the occurrence of h in the source program:

$$\lambda d.\, \lambda h.\, \mathsf{let}\ f = \lambda g.\, g\ (g\, d)\ \mathsf{in}\ (f\ (\lambda x.\, x), f\ (\lambda x.\, h\, x)) \tag{2.2}$$

Now, the argument in the second application of f is static and a binding-time analysis can assign the binding-time "static" to f. Specializing the improved program using, e.g. Similix gives

$$\lambda d.\, \lambda h.\, (d, h\ (h\, d))$$

which contains no redexes. What happens is that the argument to the second application of f is *coerced* from dynamic to static. Such binding-time coercions play a crucial role in making partial evaluation effective. For example, binding-time improvements were instrumental for Jens Palsberg and Anders Bondorf to obtain good results from specializing an interpreter for Action Notation using Similix [4, 5, 14].

$$
\begin{array}{llcll}
\text{(Types)} & \tau & = & \bullet \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 & \\[1em]
\text{(Reify)} & \downarrow^{\bullet} e & = & e & \\
 & \downarrow^{\tau_1 \times \tau_2} e & = & \underline{\mathsf{pair}}(\downarrow^{\tau_1} (\overline{\mathsf{fst}}\, e), \downarrow^{\tau_2} (\overline{\mathsf{snd}}\, e)) & \\
 & \downarrow^{\tau_1 \to \tau_2} e & = & \underline{\lambda} x.\ \downarrow^{\tau_2} (e \,\overline{@}\, (\uparrow_{\tau_1} x)), & \text{where } x \text{ is fresh} \\[1em]
\text{(Reflect)} & \uparrow_{\bullet} e & = & e & \\
 & \uparrow_{\tau_1 \times \tau_2} e & = & \overline{\mathsf{pair}}(\uparrow_{\tau_1} (\underline{\mathsf{fst}}\, e), \uparrow_{\tau_2} (\underline{\mathsf{snd}}\, e)) & \\
 & \uparrow_{\tau_1 \to \tau_2} e & = & \overline{\lambda} x.\ \uparrow_{\tau_2} (e \,\underline{@}\, (\downarrow^{\tau_1} x)), & \text{where } x \text{ is fresh} \\[1em]
\text{(Coercions)} & \mathsf{coerce}\ \tau\ e & = & \downarrow^{\tau} e & \\
 & \mathsf{coerce}'\ \tau\ e & = & \uparrow_{\tau} e &
\end{array}
$$

Figure 2.1: Type-directed binding-time coercions

2.2 Normalization by evaluation

Offline type-directed partial evaluation gives rise to a notion of *two-level programs*: The static parts of a program are executed at specialization time while the dynamic parts are re-created in the specialized program and executed at run time. Two-level languages have been studied in the context of partial evaluation and in other areas as well [34]. The common idea is to explicitly present binding times in the syntax of programs. In the following grammar, overlined terms are “static” and underlined terms are “dynamic”.

$$
\begin{array}{rl}
e = x & \mid \overline{\lambda} x.\, e \mid e_1 \,\overline{@}\, e_2 \mid \overline{\mathsf{pair}}(e_1, e_2) \mid \overline{\mathsf{fst}}\, e \mid \overline{\mathsf{snd}}\, e \mid \overline{\mathsf{let}}\ x = e_1\ \overline{\mathsf{in}}\ e_2 \\
 & \mid \underline{\lambda} x.\, e \mid e_1 \,\underline{@}\, e_2 \mid \underline{\mathsf{pair}}(e_1, e_2) \mid \underline{\mathsf{fst}}\, e \mid \underline{\mathsf{snd}}\, e \mid \underline{\mathsf{let}}\ x = e_1\ \underline{\mathsf{in}}\ e_2
\end{array}
$$

Function application is denoted by the infix @ and the construction of pairs is explicit. It is required that a two-level term is well-annotated in the sense that static reduction “does not go wrong” and yields a completely dynamic term. It is exactly the purpose of a binding-time analysis to produce a well-annotated two-level term from a source program. The results of Similix’s binding-time analysis of the two example programs (2.1) and (2.2) are

$$
\underline{\lambda} d.\, \underline{\lambda} h.\, \overline{\mathsf{let}}\ f = \overline{\lambda} g.\, g \,\underline{@}\, (g \,\underline{@}\, d)\ \overline{\mathsf{in}}\ \underline{\mathsf{pair}}(f \,\overline{@}\, (\underline{\lambda} x.\, x), f \,\overline{@}\, h) \tag{2.1$'$}
$$

$$
\underline{\lambda} d.\, \underline{\lambda} h.\, \overline{\mathsf{let}}\ f = \overline{\lambda} g.\, g \,\overline{@}\, (g \,\overline{@}\, d)\ \overline{\mathsf{in}}\ \underline{\mathsf{pair}}(f \,\overline{@}\, (\underline{\lambda} x.\, x), f \,\overline{@}\, (\overline{\lambda} x.\, h \,\underline{@}\, x)) \tag{2.2$'$}
$$

These two-level programs are well-annotated. Statically reducing the first term and removing the underlines gives the sub-optimal residual programs shown above. Doing the same for the second gives the optimal residual program. This is called *normalization by evaluation* because static reduction (evaluation) gives the normal form of the original term. In the last equation the eta-expanded term $\overline{\lambda} x.\, h \,\underline{@}\, x$ serves the purpose of coercing the dynamic h into a static function. Hence, the applications of g in the body of f are static.

Danvy has observed that completely binding-time coercing a well-typed static lambda term in a type-directed fashion followed by static reduction yields a completely dynamic term which is the

normal form of the original source term [9, 11]. The binding-time coercion is given by the pair of mutually recursive operators $\downarrow$ (reify) and $\uparrow$ (reflect) in Figure 2.1. Given a term e of type τ, $\downarrow^\tau e$ followed by static reduction yields the normal form of e, if it exists. For example, the terms in equations (2.1) and (2.2) both have type $\bullet \to (\bullet \to \bullet) \to \bullet \times \bullet$, where $\bullet$ stands for any base type. Reifying either of the terms at this type and then statically reducing the result gives

$$\underline{\lambda} d.\, \underline{\lambda} h.\, \underline{\mathsf{pair}}(d, h \mathbin{\underline{@}} (h \mathbin{\underline{@}} d))$$

which contains no redexes. Removing the underlines gives the same result as applying Similix to the binding-time improved program in equation (2.2). Figure 2.1 is the basis of *type-directed partial evaluation*.

2.3 Type-directed partial evaluation

When type-directed partial evaluation is implemented in a higher-order language one usually represents static terms as implementation-language terms and dynamic terms as a datatype of expressions. This has the consequence that static reduction coincides with implementation-language evaluation. This is generally believed to be one of the key reason why type-directed partial evaluation is fast. Figure 2.2 presents an implementation of type-directed partial evaluation in OCaml. To obtain this implementation from the binding-time coercions in Figure 2.1 the following issues must be addressed.

- Representing static terms as implementation-language terms makes the binding-time coercions type-dependent. Hence, Figure 2.1 cannot be statically typed in a Hindley/Milner type system. Andrzej Filinski and then Zhe Yang [42] and the present author [37] have shown how an encoding of types as pairs makes the binding-time coercions typeable in a Hindley/Milner type system. In the OCaml program in Figure 2.2, types are represented by the datatype τ typ. (This datatype is often called τ rr because it encapsulates both reification and reflection at type τ.)

  A value $\ulcorner\tau\urcorner$ representing a type τ is constructed from `b`, `int`, `pair`, and `arrow` using the obvious type representations

$$\begin{array}{rcl} \ulcorner\bullet\urcorner &=& \texttt{b} \\ \ulcorner\mathsf{int}\urcorner &=& \texttt{int} \\ \ulcorner\tau_1 \times \tau_2\urcorner &=& \texttt{pair}(\ulcorner\tau_1\urcorner, \ulcorner\tau_2\urcorner) \\ \ulcorner\tau_1 \to \tau_2\urcorner &=& \texttt{arrow}(\ulcorner\tau_1\urcorner, \ulcorner\tau_2\urcorner) \end{array}$$

  By construction, type-directed partial evaluation does not work on types in which integers (or other base types) occur in *contravariant* position.

- The binding-time coercion in Figure 2.1 assumes a call-by-name setting. Reifying the term

$$\lambda f.\, \lambda x.\, (\lambda y.\, x)\, (f\, x) \tag{2.3}$$

  at its type $(\bullet \to \bullet) \to \bullet \to \bullet$, followed by static evaluation yields $\lambda f.\, \lambda x.\, x$. The source term and residual term are only equivalent in a call-by-name setting. To work also in a call-by-value setting the binding-time coercions are modified to do dynamic let-insertions. The let-bindings

```
exception Error

type exp = VAR of int | INT of int | LAM of int * exp | APP of exp * exp
         | PAIR of exp * exp | FST of exp | SND of exp
         | LET of (int * exp) list * exp | GLB of string

let gensym = let  c = ref 0  in  fun () -> c := (!c) + 1; !c

type 'a typ = RR of ('a -> exp) * (exp -> 'a)

let b =   RR((fun e -> e),     (fun e -> e))
let int = RR((fun i -> INT i), (fun i -> raise Error))

let pair(RR(dw1, up1), RR(dw2, up2)) =
  RR((fun(e1, e2) -> PAIR(dw1 e1, dw2 e2)),
     (fun e       -> (up1(FST e), up2(SND e))))

let hook = ref []
let makelet = function
    ([],             m)                 -> m
  | ([(x, e)],       VAR y) when y = x -> e
  | ((x, e) :: bind, VAR y) when y = x -> LET(List.rev bind, e)
  | (bind,           m)                 -> LET(List.rev bind, m)

let arrow(RR(dw1, up1), RR(dw2, up2)) =
  RR((fun e ->
       let x = gensym() in
       let previous_hook = !hook
       in hook := [];
          let body = dw2(e(up1(VAR x))) in
          let header = !hook
          in hook := previous_hook;
             LAM(x, makelet(header, body))),
     (fun e x ->
       let r = gensym()
       in hook := (r, APP(e, dw1 x)) :: !hook;
          up2(VAR r)))

let nbe (RR(dw, up)) e = dw e
let nbe'(RR(dw, up)) e = up e
```

Figure 2.2: Type-directed partial evaluation in OCaml

are accumulated in a global hook, a solution suggested by Eijiro Sumii [41]. Using the modified version to reify the term in equation (2.3) yields residual term $\lambda f.\,\lambda x.\,\mathsf{let}\ y = f\ x\ \mathsf{in}\ x$.

Functional programs use recursive *tail-calls* to implement iteration. Compilers for functional languages therefore reuse the topmost stack-frame in tail-calls such that the stack does not grow unboundedly. It is therefore essential that type-directed partial evaluation does not reduce opportunities for tail calls. All that it needs to do is not generate any trivial let expressions such

as $\mathsf{let}\ x = e\ \mathsf{in}\ x$ but generate e instead.

The function `makelet` takes a list of bindings and an expression and produces a let expression. (A binding is a pair of an identifier and an expression.) It preserves tail calls in the sense that instead of generating $\mathsf{let}\ x = e\ \mathsf{in}\ x$ (in which e does not occur in tail position) it returns just e. The following examples use the second, third, and fourth branch of the case expression in `makelet`, respectively. Additionally, they all use the first branch.

$$\begin{array}{lcl}
\mathsf{nbe}\ \ulcorner(\bullet \to \bullet) \to (\bullet \to \bullet)\urcorner\ (\lambda f.\, \lambda x.\, f\ x) & = & \lambda f.\, \lambda x.\, f\ x \\
\mathsf{nbe}\ \ulcorner(\bullet \to \bullet) \to (\bullet \to \bullet)\urcorner\ (\lambda f.\, \lambda x.\, f\ (f\ x)) & = & \lambda f.\, \lambda x.\, \mathsf{let}\ y = f\ x\ \mathsf{in}\ f\ y \\
\mathsf{nbe}\ \ulcorner(\bullet \to \bullet) \to (\bullet \to \bullet)\urcorner\ (\lambda f.\, \lambda x.\, (f\ x;\ x)) & = & \lambda f.\, \lambda x.\, \mathsf{let}\ y = f\ x\ \mathsf{in}\ x
\end{array}$$

For conventional type-directed partial evaluation, which is equipped with a pretty printer, either type-directed partial evaluation or the pretty printer can make sure that the residual code is properly tail-recursive. However, since there is no pretty-printing involved in run-time code generation it is essential that type-directed partial evaluation produces properly tail-recursive code straight away.

- Section 2.4 below contains an example of how type-directed partial evaluation is applied and in particular how dynamic primitives are handled in practice. (Here a dynamic primitive is a function that occurs unbound in the source program and that should also occur in the residual program.) A primitive of type τ is generated by reflecting the name of a global variable at the type of the primitive (using nbe' from Figure 2.2). The source program is given as a module abstracted over a module of primitives. This allows the same program to be applied directly (by instantiating the module with the standard interpretation of primitives) and to be specialized (by instantiating the module with the non-standard interpretation of primitives).

2.4 Example: The power function (1)

As an example, we apply type-directed partial evaluation to the power function. Figure 2.3 shows the source program of a signature of primitives, a standard and a non-standard interpretation of the signature, and a definition of the power function parameterized by a module of primitives.

Standard ML has both a *transparent* signature matching operator (`:`) and an *opaque* one (`:>`). OCaml has just one (`:`) which is opaque. In conventional type-directed partial evaluation the module of primitives is usually matched transparently. This is not compatible with run-time code generation as shown in the next chapter (and since OCaml lacks transparent matching it would not be possible anyway). Using opaque signature matching means that a type bound by the module becomes different from all other types (*abstract*). For example, the type PrimEv.tint is a different from the type of integers so values of type PrimEv.tint cannot be treated as integers. For example

```
# PrimEv.qint 100;;
- : PrimEv.tint = <abstr>
#
```

(# is the interactive prompt of OCaml and `;;` terminates an input expression.) We equip the standard interpretation `PrimEv` with an unquote function `uint` that returns the underlying integer of a PrimEv.tint.

```
module type PRIM = sig
  type tint
  val qint : int  -> tint
  val uint : tint -> int
  val mul  : tint * tint -> tint
end

module PrimEv : PRIM = struct
  type tint     = int
  let qint i    = i
  let uint i    = i
  let mul(x, y) = x * y
end

module PrimSp : PRIM with type tint = exp = struct
  exception Error
  type tint  = exp
  let qint i = INT i
  let uint i = raise Error
  let mul    = nbe' (arrow(pair(b, b), b)) (GLB "mul")
end

module Power(P : PRIM) = struct
  let rec pow n x =
    if  n = 0  then  P.qint 1  else  P.mul(x, pow (n-1) x)
end
```

Figure 2.3: The power function in OCaml(1)

```
# PrimEv.uint(PrimEv.qint 100);;
- : int = 100
#
```

We need not provide an unquote function for the non-standard interpretation `PrimSp` because the unquote function is only used when printing the result of the power function. The specialized power function uses the same unquote function when applied. The non-standard interpretation needs to export the equality $\mathsf{tint} = \mathsf{exp}$ in order to apply $\mathsf{nbe}\ulcorner\bullet \to \bullet\urcorner$ (which has type $(\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$) to the result of `PowSp.pow 5` (which has type $\mathsf{PrimSp.tint} \to \mathsf{PrimSp.tint}$.) Another possibility is to let `PrimEv` export the equality $\mathsf{tint} = \mathsf{int}$. Then OCaml can print values of type $\mathsf{PrimEv.tint}$ as integers. However, this does not work when it comes to run-time code generation we will see in Chapter 3.

We can now build a module for evaluating the power function using the standard interpretation of primitives. Note that, because of opaque matching in the functor instantiation we need to quote integers.

```
# module PowEv = Power(PrimEv);;
module PowEv : sig val pow : int -> PrimEv.tint -> PrimEv.tint end
# PrimEv.uint(PowEv.pow 5 (PrimEv.qint 3));;
- : int = 243
#
```

```
module Power5(P : PRIM) = struct
  open P
  let power5 =
    fun x1 ->
      let x2 = mul (x1, qint 1) in
      let x3 = mul (x1, x2) in
      let x4 = mul (x1, x3) in
      let x5 = mul (x1, x4) in mul (x1, x5)
end
```

Figure 2.4: The power function specialized to $n = 5$

We can also build a module for specializing the power function using the non-standard interpretation of primitives. Below we specialize the power function to a static exponent $n = 5$. The residual program is an element of the datatype `exp`.

```
# module PowSp = Power(PrimSp);;
module PowSp : sig val pow : int -> PrimSp.tint -> PrimSp.tint end
# nbe (arrow(b, b)) (PowSp.pow 5);;
- : exp =
  LAM
   (1,
    LET
     ([2, APP (GLB "mul", PAIR (VAR 1, INT 1));
       3, APP (GLB "mul", PAIR (VAR 1, VAR 2));
       4, APP (GLB "mul", PAIR (VAR 1, VAR 3));
       5, APP (GLB "mul", PAIR (VAR 1, VAR 4))],
      APP (GLB "mul", PAIR (VAR 1, VAR 5))))
#
```

Pretty-printing this residual program and wrapping the result in an appropriate functor gives the program in Figure 2.4. The functor can be instantiated for evaluation or for specialization purposes. The purpose of run-time code generation, as described in the next chapter, is to short-cut the intermediate representation of the residual program as text in such a way that specialization directly yields a value that can be used.

2.5 Conclusions

This chapter has presented background knowledge of type-directed partial evaluation. We have seen how a normalization function for the simply-typed lambda calculus can be used to achieve partial evaluation. We presented an implementation of this idea in OCaml and showed how the power function is specialized. Furthermore, we outlined some differences between this implementation and the usual Standard ML implementation caused by differences in the module languages of OCaml and Standard ML.

Chapter 3

Run-time code generation for type-directed partial evaluation

In this chapter we present the design of a run-time code generator for type-directed partial evaluation and an implementation in OCaml. For the purpose of comparison, this chapter also repeats the example of the power function from the previous chapter but this time using run-time code generation.

3.1 Design: New expressions

The purpose of run-time code generation is to deforest the intermediate representation of residual programs between type-directed partial evaluation and evaluation. The implementation provides a seamless interface to run-time specialization in OCaml. We extend OCaml with two new expressions normalize e and primitive x, where e is an expression and x is a globally bound variable. Informally, normalize e mimics eval (nbe $\ulcorner\lceil\tau_e\rceil\urcorner$ e) and primitive x mimics nbe$'$ $\ulcorner\tau_x\urcorner$ (GLB "x") for appropriate types τ_e and τ_x. (Below we discuss what the appropriate types are.) The result of normalize e is a (possibly higher-order) value corresponding to the normal-form of e. This contrast with conventional type-directed partial evaluation where the result is a textual representation of the normal-form of e. The result of primitive x is a value that contains a dynamic reference to the global variable x. In contrast with conventional type-directed partial evaluation the programmer needs not specify the type of the primitive.

The types at which nbe and nbe$'$ are applied are automatically computed from the types of e and x, respectively. This contrasts with conventional type-directed partial evaluation where the user needs to specify the types. Since OCaml is statically typed, and since OCaml automatically infers the types of expressions, the extension of OCaml must also compute the correct type of the new expressions.

3.2 Types of the new expressions

What is a reasonable choice for τ in nbe $\ulcorner\tau\urcorner$ e if e has type τ'? And what is the type of the result of nbe $\ulcorner\tau\urcorner$ e? These questions must be dealt with in the extension of OCaml since (1) the extension automatically decides at which type nbe is applied and (2) OCaml is required to infer the type of expressions automatically. In conventional type-directed partial evaluation, these questions are handled by (1) the user who delivers the type at which nbe is applied and (2) the compiler that type-checks the pretty-printed result. (Similar questions can be asked about the application of nbe$'$.)

Let us first consider the case of nbe. If the source program is closed then the type of the source program, the type at which nbe is applied, and the type of the residual program coincide. This is a consequence of the following lemma.

Lemma 1 *If e is a closed program of type τ then $(\downarrow^\tau\ e)$ is a well-annotated two-level term. Statically reducing $(\downarrow^\tau\ e)$ gives a term of type τ.*

When the source program contains an unbound reference to a dynamic primitive then the type of the primitive may influence the type of the residual program. Below we define a function of type $\mathsf{int} \to \mathsf{int}$ and a dynamic primitive of that function using conventional type-directed partial evaluation.

```
# let double x = 2 * x;;
val double : int -> int = <fun>
# let double_prim = nbe' (arrow(b, b)) (GLB "double");;
val double_prim : exp -> exp = <fun>
#
```

If the dynamic primitive `double_prim` occurs in a program which is subsequently normalized, then the residual program will contain a reference to the original function `double`. A simple example is

```
# pp (nbe (arrow(b, b)) double_prim);;
fun x7 -> double x7
- : unit = ()
# fun x7 -> double x7;;
- : int -> int = <fun>
#
```

The system should deduce the type $\mathsf{int} \to \mathsf{int}$ of the residual program from the type $\mathsf{exp} \to \mathsf{exp}$ of the source program. This is clearly a problem because the exp does not carry enough information to "turn it into" an int. Therefore, we replace the type exp by a new type, τ exp, that carries this information. It is the intention that `double_prim` should have type $\mathsf{int\ exp} \to \mathsf{int\ exp}$. From this type it is easy to deduce the type of the result of nbe by erasing the exp's from the base types. Similarly, the type $\mathsf{int\ exp} \to \mathsf{int\ exp}$ can easily be deduced from the type of the original function, $\mathsf{int} \to \mathsf{int}$, by annotating the base types with exp's. These considerations have motivated the type conversions both of which are shown in Figure 3.1. Using the conversion we define (informally)

$$\begin{aligned} \mathsf{normalize}\ e &\approx \mathsf{eval}\ (\mathsf{nbe}\ulcorner\lceil\tau_e\rceil\urcorner e) \\ \mathsf{primitive}\ x &\approx \mathsf{nbe}'\ulcorner\tau_x\urcorner(\mathtt{GLB}\ \texttt{"}x\texttt{"}) \end{aligned}$$

where $e : \tau_e$ and $x : \tau_x$ is a globally bound variable. In the current implementation the types of the new expressions are

$$\begin{aligned} \mathsf{normalize}\ e &: \lceil\tau_e\rceil \\ \mathsf{primitive}\ x &: \lfloor\tau_x\rfloor \end{aligned}$$

(In Chapter 5 we show a limitation of this type system.)

3.3 Implementation

We have implemented the new type and expressions described above in the higher-order, statically typed, modular language OCaml, a dialect of ML [29, 32]. We concentrate on version 2.02 of the OCaml implementation.[1] The system consists of a byte-code compiler, a native-code compiler, and

[1] Version 2.03 was released during the writing of this report. We plan to upgrade our implementation to this version in the near future.

$$
\begin{aligned}
\lfloor \bullet \rfloor &= \bullet\ \mathsf{exp} & \lceil \bullet \rceil &= \bullet \\
\lfloor \mathsf{int} \rfloor &= \mathsf{int} & \lceil \mathsf{int} \rceil &= \mathsf{int} \\
\lfloor \bullet\ \mathsf{exp} \rfloor &= \text{error} & \lceil \bullet\ \mathsf{exp} \rceil &= \bullet \\
\lfloor \mathsf{int\ exp} \rfloor &= \text{error} & \lceil \mathsf{int\ exp} \rceil &= \mathsf{int} \\
\lfloor \tau_1 \to \tau_2 \rfloor &= \lfloor \tau_1 \rfloor \to \lfloor \tau_2 \rfloor & \lceil \tau_1 \to \tau_2 \rceil &= \lceil \tau_1 \rceil \to \lceil \tau_2 \rceil
\end{aligned}
$$

Figure 3.1: Type conversions

---

a runtime system with an virtual machine for running byte-code executables. Both compilers are implemented in OCaml and they share a common front end. The run-time system, consisting of a byte-code interpreter, a garbage collector, and a set of predefined library procedures, is implemented in C. We have extended the byte-code compiler to handle run-time code generation.

3.3.1 Overview of the OCaml byte-code compiler

OCaml's compiler consists of modules each implementing a phase. The initial input is a stream of characters, either read from a file (in batch mode) or from standard input (in interactive mode).

Lexical analysis and parsing: Together, lexical analysis and parsing read a sequence of characters and produces a sequence tokens. We have added actions that parse the new expressions $\mathsf{normalize}\ e$ and $\mathsf{primitive}\ x$.

Type analysis: This phase type-checks the source program. It produces a type-annotated abstract syntax tree. We have extended the type checker to account for the design presented above: An expression $\mathsf{normalize}\ e$, where $e : \tau$, is annotated with its type $\lceil \tau \rceil$. At the same time the type $\lceil \tau \rceil$ is remembered as part of the expression for use in the code-generation phase. Similar extensions are carried out to account for $\mathsf{primitive}\ x$.

Semantics-preserving translations: This phase translates OCaml expressions into lambda terms and performs some optimizations on lambda terms. The major difference between an OCaml expression and a lambda term is that module expressions are represented by lambda terms. We have extended these phases to pass normalize- and primitive-expressions unmodified through.

Code generation: This phase produces a list of symbolic byte-codes from a lambda term. The extensions to this phase are described in detail in Section 3.3.3.

Byte-code emitting: This phase writes a list of symbolic byte-codes to a file (in batch mode) or into memory (in interactive mode).

The amount of dynamically generated code can grow arbitrarily. Generated code, however, may eventually become inaccessible, just like normal closures in a higher-order programming language such as OCaml. It is thus essential for the applicability of the run-time code generator that such inaccessible code is garbage collected. However, this is not a new problem: OCaml faces the same one due to its interactive top-level loop. OCaml solves this problem by storing executable byte code in the garbage collected heap. We use the same solution for dynamically generated code.

$$
\begin{array}{llcll}
\text{(Reify)} & \Downarrow^{\bullet} e & = & e & \\
& \Downarrow^{\tau_1 \times \tau_2} e & = & \underline{\underline{\mathsf{pair}}}(\Downarrow^{\tau_1} (\underline{\mathsf{fst}}\, e), \Downarrow^{\tau_2} (\underline{\mathsf{snd}}\, e)) & \\
& \Downarrow^{\tau_1 \to \tau_2} e & = & \underline{\underline{\lambda}} x.\ \Downarrow^{\tau_2} (e \,\underline{@}\, (\Uparrow_{\tau_1} x)) & \text{where } x \text{ is fresh} \\
& & & & \\
\text{(Reflect)} & \Uparrow_{\bullet} e & = & e & \\
& \Uparrow_{\tau_1 \times \tau_2} e & = & \underline{\mathsf{pair}}(\Uparrow_{\tau_1} (\underline{\underline{\mathsf{fst}}}\, e), \Uparrow_{\tau_2} (\underline{\underline{\mathsf{snd}}}\, e)) & \\
& \Uparrow_{\tau_1 \to \tau_2} e & = & \underline{\lambda} x.\ \Uparrow_{\tau_2} (e \,\underline{\underline{@}}\, (\Downarrow^{\tau_1} x)) & \text{where } x \text{ is fresh} \\
& & & & \\
\text{(Coercions)} & \mathsf{coercegen}\ \tau & = & \underline{\lambda} x.\ \Downarrow^{\tau} x, & \text{where } x \text{ is fresh} \\
& \mathsf{coercegen}'\ \tau & = & \underline{\lambda} x.\ \Uparrow_{\tau} x, & \text{where } x \text{ is fresh}
\end{array}
$$

Figure 3.2: Generating extension of Figure 2.1

3.3.2 The generating extension of type-directed partial evaluation

Consider an OCaml program where $\mathsf{normalize}\ e$ occurs in a position where it is evaluated several times. Because OCaml is statically typed the type τ that the underlying nbe is applied at is fixed at compile time. Instead of evaluating

$$\mathsf{eval}\ (\mathsf{nbe}\ulcorner\lceil\tau\rceil\urcorner\, e)$$

several times it is more efficient to specialize nbe with respect to the type τ at compile time and to use this specialized version at run time. The reason is that most computations in nbe depend only on the type. We use the generating extension of nbe to obtain the specialized version. The generating extension of nbe is a function nbegen that, when applied to a representation of a type $\ulcorner\tau\urcorner$, returns the representation of a program nbe_τ. Applying nbe_τ to a value e of type τ yields the representation of the normal form of e, which is also the result of $\mathsf{nbe}\ \ulcorner\tau\urcorner\ e$.

$$
\begin{array}{rcl}
[\![\mathsf{nbegen}]\!]\ \ulcorner\tau\urcorner & = & \mathsf{nbe}_\tau \\
[\![\mathsf{nbe}_\tau]\!]\ e & = & [\![\mathsf{nbe}]\!]\ \ulcorner\tau\urcorner\ e
\end{array}
$$

Figure 3.2 shows the generating extension of the type-directed binding-time coercions in Figure 2.1. The generating extension of the binding-time coercions arises from the same sequence of steps as the generating extension of the power function in Scheme in Chapter 1.2.3. Namely, by binding-time analyzing the source program according to the binding-time signature of the input, here a type (static) and a value (dynamic), and then implementing the static parts so they reduce and the dynamic parts so they generate code. In Scheme, we used quote and unquote to accomplish this. Here we use a two-level language with overlines and underlines. Because the original program already contained overlined and underlined terms, the generating extension now contain double underlined terms. (Double underlined terms represent program parts that generate programs that generate programs.)

As an example

$$\mathsf{coercegen}\ (\bullet \to \bullet) \to (\bullet \to \bullet) \quad = \quad \underline{\lambda} x_1 .\, \underline{\underline{\lambda}} x_2 .\, \underline{\underline{\lambda}} x_3 .\, x_1 \,\underline{@}\, (\underline{\lambda} x_4 .\, x_2 \,\underline{\underline{@}}\, x_4) \,\underline{@}\, x_3$$

`app`	: code × code → code	Make an application
`lam`	: var × code → code	Make a lambda abstraction
`var`	: var → code	Make a lambda-bound variable
`pair`	: code × code → code	Make a pair
`fst`	: code → code	Make a left-projection
`snd`	: code → code	Make a right-projection
`glb`	: string → code	Make a globally bound variable

Figure 3.3: Code template combinators (subset)

This term is completely dynamic. It is a representation of the following term (which is obtained by removing one level of underlines).

$$\lambda x_1.\, \underline{\lambda} x_2.\, \underline{\lambda} x_3.\, x_1\ (\lambda x_4.\, x_2 \,\underline{@}\, x_4)\ x_3$$

Applying this term to $\lambda f.\, \lambda x.\, f\ x$ (which indeed has type $(\bullet \to \bullet) \to (\bullet \to \bullet)$) yields the completely dynamic

$$\underline{\lambda} x_2.\, \underline{\lambda} x_3.\, x_2 \,\underline{@}\, x_3$$

In practice, because of let insertions, the generating extension of nbe is slightly more involved than Figure 3.2 but conceptually they operate similarly.

3.3.3 Extensions to the byte-code generator

The two main addition to the OCaml compiler are (1) an implementation of code templates and (2) an implementation of the generating extension of type-directed partial evaluation that produces code templates. Code templates directly represent expressions with free variables. An expression e is represented as a triple (f, c, v) of type code where

- f of type var list is a list of the free variables of e;
- c of type env × instruction list → instruction list is a function that yields the compiled list of instructions of the expression, given an environment that binds the free variables and a list of instructions that represents the continuation of the expression;
- v of type var option is Some(x) if e is a variable x and None if e is not a variable. This field is used to test whether a code template represents a variable. This is needed when inserting let expressions such that tail calls are preserved.

The code templates are constructed using *code template combinators* that resemble the constructors of abstract syntax. The types of a subset of these are shown in Figure 3.3. They correspond exactly to the (single) underlined operators used in the generating extension in Figure 3.2. The double underlined operators are represented by code templates that references the global variable corresponding to the single underlined constructor. For example, the term $\underline{\underline{\mathsf{pair}}}(e_1, e_2)$ is generated by `app(glb "pair", pair(`e_1`,`e_2`))`.

Given a code template (f, c, x) the list of instructions that it encapsulates is obtained by applying c to the empty environment and a representation of the empty continuation. To compile the expression normalize e where $e : \tau$ the compiler applies coercegen τ to obtain the code template for the specialized instance of nbe at type τ. The instruction list of this code template is combined with the instruction list of the expression e in such a way that the specialized instance of nbe is applied to the value of e at run time. The result is that, at run time, normalize e yields a value which is the normal form of e. Something similar happens for primitive x.

The original OCaml distribution has more than 30,000 lines of code implementing the (byte-code) compiler and around 12,000 lines of code implementing the run-time system. We have added about 230 lines of OCaml code to account for 15 code template combinators handling integers, booleans, strings, tuples, and let expressions. We have added less than 100 lines of code to account for the generating extension of the type-directed partial evaluator. The generating extension handles products, functions, and base types such as integers and booleans. We have added about 150 lines to the type checking phase and about 50 lines of additional code to the lexical analyser, the parser, and the semantic-preserving simplification phase.

3.4 Example: The power function (2)

As an application of run-time code generation we consider the example of the power function again. As in Figure 2.3 on page 12 the source program is structured as three modules, a standard interpretation of primitives, a non-standard interpretation of primitives, and the power function parameterized over primitives. The source program is shown in Figure 3.4. It is only the non-standard interpretation of primitives `PrimSp` that differs from the program presented in Figure 2.3. It builds primitives using the new expression primitive x. The type constructor `prog` implements the types τ exp mentioned above.

Figure 3.4 provides an explanation why the modules must be opaquely matched against the signature `PRIM` and why the standard interpretation `PrimEv` does not export the equality tint = int. (See page 12.) The type of `PrimEv.qint` directs the construction of the primitive in the module `PrimEv`. If tint = int then `PrimEv.qint` would have type int $\rightarrow$ int. An error would occur from building a primitive of this type since reflect ($\uparrow$) is undefined on int. This would happen if modules are matched opaquely or if modules are not matched at all, or if the type equality were exported from the module `PrimEv`.

Using the run-time code generator we can specialize the power function as follows.

```
# module PowSp = Power(PrimSp);;
module PowSp : sig val pow : int -> PrimSp.tint -> PrimSp.tint end
# let power5 = normalize(PowSp.pow 5);;
val power5 : PrimEv.tint -> PrimEv.tint = <fun>
# PrimEv.uint(power5 (PrimEv.qint 3));;
- : int = 243
#
```

Unlike using conventional type-directed partial evaluation the residual program obtained using run-time code generation is not wrapped inside a functor. We come back to this point in Chapter 5, in the paragraph about *incremental specialization*.

```
module type PRIM = sig              (*** Same as conventional tdpe ***)
  type tint
  val qint : int  -> tint
  val uint : tint -> int
  val mul  : tint * tint -> tint
end

module PrimEv : PRIM = struct       (*** Same as conventional tdpe ***)
  type tint     = int
  let qint i    = i
  let uint i    = i
  let mul(x, y) = x * y
end

module PrimSp : PRIM with type tint = PrimEv.tint prog = struct
  exception Error
  type tint  = PrimEv.tint  prog
  let qint   = primitive PrimEv.qint
  let uint i = raise Error
  let mul    = primitive PrimEv.mul
end

module Power(P : PRIM) = struct     (*** Same as conventional tdpe ***)
  let rec pow n x =
    if  n = 0  then  P.qint 1  else  P.mul(x, pow (n-1) x)
end
```

Figure 3.4: The power function in OCaml(2)

3.5 Conclusions

In this chapter we have extented OCaml with run-time code generation for type-directed partial evaluation. The result is a seamless extension of OCaml with a “normalize” directive and a “primitive” directive. The core of the implementation is the generating extension for type-directed partial evaluation and a set of code-template combinators. In total, we have increased the size of the original OCaml implementation with less than 2%.

For pedagogical purpose, this chapter also contained an application of run-time code generation to the power function. The specialized version is about 36% faster than the original, for small numbers. The next chapter presents a larger application of run-time code generation.

Chapter 4

Example: The BSD packet filter

In this chapter we present an application of run-time code generation to *network packet filtering* similar to the example of Leone and Lee [28].

4.1 Network packet filters

When a user-level process receives network packets, these are copied from the kernel space to the user space. To minimize copying and context switching, the user-level program can tell the kernel to filter out packets satisfying certain criteria. Because recognizing an incoming packet can be quite complicated, filtering mechanisms have quite general filter languages. BSD packet filters [31] are expressed as the abstract syntax of a general, RISC-like instruction set. A virtual machine run as a kernel process applies a filter on each incoming packet and depending on the outcome the packet is copied to the user-level process.

4.2 Compiling packet filters

Packet filtering is an obvious application area for partial evaluation. A user-level process specifies at most one filter which will be applied to a large number of incoming packets. The overhead from having the kernel interpreting filters can obviously be reduced by specializing the packet filter interpreter with respect to the filter. However, conventional partial evaluation is not sufficient because (1) exactly how to recognize packets may depend on information which is not present until the user-level program is executed and (2) the filter must be verifiably safe. Thus, it is not possible to specialize the kernel code when the user-level program is compiled. Something like "just-in-time" specialization is needed. This is just what run-time code generation for type-directed partial evaluation gives.

Leone and Lee [28] have already applied run-time code generation to packet filtering using the Fabius system. They conclude that using Fabius to compile a filter and then using the compiled filter is faster than running the C implementation of the packet-filter interpreter after about 250 packets.

We have performed some initial experiments to test whether the same idea applies using run-time code generation for type-directed partial evaluation in OCaml. We have concentrated on measuring the speedup of running compiled filters compared with interpreted filters. The packet-filter interpreter is implemented in OCaml.[1] It operates on abstract representations of filters and not on sequences of bytes as in the C implementation.

[1]Thanks to Lasse Reichstein Nielsen for providing a packet filter interpreter for conventional type-directed partial evaluation in Standard ML.

Filter	Accepts
1	no packets
2	IP packets
3	packets not from host 128.3.112.x or host 128.3.254.x
4	TCP packets to port N
5	reverse ARP requests
6	IP packets between host 128.3.112.15 and 128.3.112.35

Figure 4.1: Example packet filters

4.3 Benchmarks

The packet-filter interpreter is applied to the six filters described in Figure 4.1. They are taken from the BPF manual page [30] and McCanne and Jacobson's original article on the BSD packet filter [31]. The examples are all measured on a IBM ThinkPad 600 equipped with a Pentium II running at 266MHz and with 96 Mb of RAM running RedHat Linux.

In all cases, compiling a filter "pays for itself" after less than 2000 packets have been received and in all cases this can happen after less than 0.03 seconds when enough packets are available. For comparison, in the connection between the Department of Computer Science in Aarhus and the outside world, more than 50 million packets transit every day. Typical packet filters handle packet counted in millions, e.g., for a TCP/IP monitor. In that context, an amortization factor of a few thousands is thus eminently reasonable.

The benchmark results are shown in the graphs in Figure 4.2. Each graph present (1) the accumulated time for filtering 2000 random packets using the interpreter and (2) the accumulated time for filtering the same packets using the compiled filter and also accounting for the time spent compiling the filter. The compiled filters handles around 50-100% more packets per time-unit than the interpreted filters. The numbers are summarized in the following table. The second column shows the number of packets required before compilation pays for itself and the third column shows how early that may happen. The fourth column shows the increase in the number of packets the compiled filter handles per time-unit.

Filter	Amortization factor		Speedup
1	125 packets	0.00061 sec.	118%.
2	275 packets	0.00477 sec.	65%.
3	1016 packets	0.01243 sec.	48%.
4	1798 packets	0.02414 sec.	58%.
5	783 packets	0.00957 sec.	49%.
6	1870 packets	0.02223 sec.	49%.

Using random packets probably gives a higher amortization factor than using "real" packets. Many filters reject random packets early because the packet header does not match the very first tests. For example, a filter that checks whether packets come from a specific IP address would first check whether packets are IP packets at all and only then check the address. Compilation time is independent of the packets so if packet processing increases for both interpreted filters and compiled filters then the amortization factor decreases.

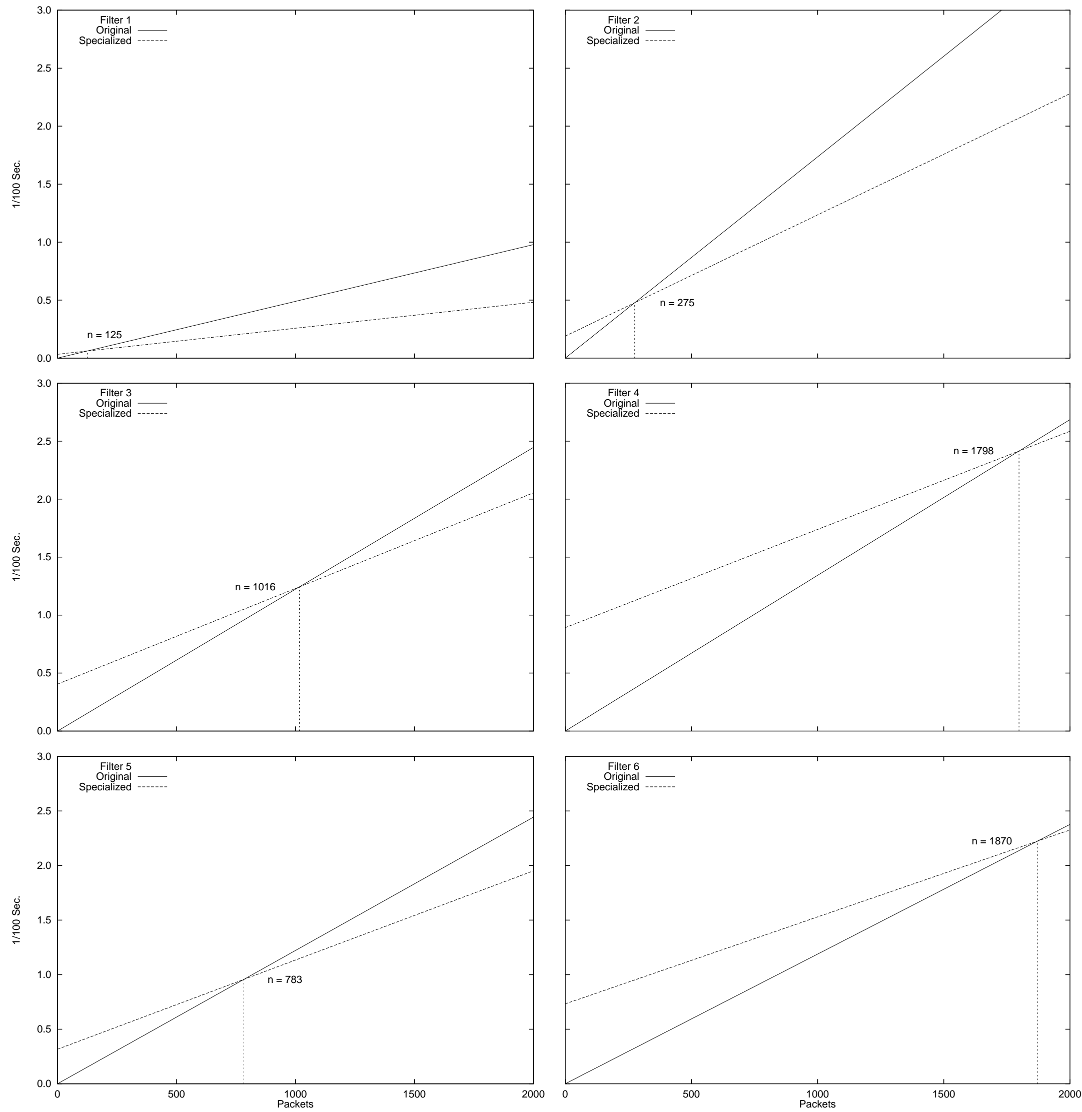


Figure 4.2: Relative effect of compiling packet filters

4.4 Conclusions

Compiling network packet filters is an obvious application for run-time code generation. The BSD packet filter has previously been compiled in Fabius with an amortization factor of around 250 compared to *the packet-filter interpreter implemented in C*.

We have compiled filters in OCaml and compared the result to *interpreting filters in OCaml*. The amortization factor is between 125 and 1870 for example filters. It remains to compare our compiled filters with interpreting filters using the native C implementation. Our run-time code generator produces byte codes which are interpreted by a virtual machine. This contrasts with C and Fabius which generate native code. We intend to alleviate the time penalty of the virtual machine by using OCaml's capabilities of interfacing with C.

Chapter 5

Conclusion and future work

In this report we have presented the design motivations of a run-time code generator for type-directed partial evaluation together with an implementation in OCaml. The implementation is both fast and efficient: As mentioned in Chapter 4, the amortization factor (i.e., the least number of times the specialized program should be run in order for specialization to pay for itself) is negligible for BSD packet filters. Furthermore, an experiment with Action Semantics show an amortization factor less than 2 [15]. We hope to report further experiments in our Ph.D. thesis.

Type correctness: The current implementation has a limitation in that `normalize` e does not restrict the type of e sufficiently. This may cause run-time type errors, which is of course contrary to the spirit of ML. We are currently investigating how to restrict the type of the expressions that are normalized. Following that, a "type correctness" lemma should be stated and proved. Informally, such a statement should say that if a program passes the type checker then running the program will not yield a run-time type error.

Multi-level type-directed partial evaluation: The two-level language discussed in Section 2.2 can be generalized to n levels for fixed n or even to an arbitrary number of levels (using for example the style of Glück and Hatcliff [23]). This gives rise to the type constructor τ `prog` (presented in the beginning of this report) that corresponds to the *next* operator from the modal lambda calculus [16, 17]. The generating extension in Figure 3.2 can be generalized to also handle types on form τ `prog` and produce n-level terms. In its generalized form, the generating extension does *incremental specialization*. It also provides a connection with the second Futamura projection for type-directed partial evaluation, a problem that has been tackled differently by Grobauer and Yang [22].

The multi-level generating extension may also provide a link to the conventional way of using functors to specify incremental type-directed partial evaluation [11]. The result of conventional type-directed partial evaluation is given as a functor that abstracts the primitives from the residual program. This functor can be instantiated with a standard interpretation of primitives for direct evaluation or it can be instantiated with nonstandard primitives for further specialization. The implementation presented in this report does not have these capabilities. Implementing this scheme in a statically typed language has proven to be a challenge. The problem is the following. Prior to (the first round of) specialization, the user instantiates the functor of the source program with the nonstandard interpretation of primitives. The result is a module containing a version of the source program fixed to a certain interpretation of types. After normalization the residual program is still fixed to a certain interpretation of types. To wrap the residual program into a functor would require these types to be captured by the

argument-module of the functor. We are currently investigating this issue more closely, to find out to which extent incremental specialization can be combined with run-time code generation. One viable option is to revert from functors to polymorphic functions. In the context of type-directed partial evaluation functor and polymorphic functions can serve the same purpose, namely to abstract the source program with its primitives. Polymorphic functions achieve this in such a way that the specialized program is polymorphic and can be instantiated again. The cost of using functions is that modules cannot be used to structure the source program.

Typed abstract syntax: ML programs that manipulate terms of a higher-order *object language* usually represent these terms as a datatype. An example is the datatype `exp` in Figure 2.2. OCaml is statically typed but representing the object language as a datatype does not provide static typing for the object language terms. In the following session the type system of OCaml reports an error when applying an integer to another integer at the implementation-language level. No error is reported when doing the same at the object-language level.

```
# 1 2;;
Characters 0-1:
This expression is not a function, it cannot be applied
# APP(INT 1, INT 2);;
- : exp = APP (INT 1, INT 2)
#
```

That the object-language term is accepted not a restriction of OCaml since we could be manipulating untyped terms. However, if terms are known to be typed, it might be useful to have them type checked. We are not aware of any earlier typed representation of object terms, though since then, Zhe Yang and Peter Thiemann each have independently found another alternative representation. Here, we obtain a typed object-language by combining *higher-order abstract syntax* [36] with the module system of ML. An implementation in OCaml is shown in Figure 5.1. (It can be implemented in Standard ML in a similar way.) The module `Exp` implements functions that construct terms. The signature `EXP` restricts the way terms can be combined by assigning more type information to the syntax constructors. For example, in the previous approach in Figure 2.2

$$\begin{aligned} \texttt{APP} &: \mathsf{exp} \times \mathsf{exp} \to \mathsf{exp} \\ \texttt{INT} &: \mathsf{exp} \to \mathsf{exp} \end{aligned}$$

These constructors do not prohibit the construction `APP(INT 1, INT 2)`. In Figure 5.1 the corresponding types are

$$\begin{aligned} \texttt{app} &: (\alpha \to \beta)\ \mathsf{exp} \times \alpha\ \mathsf{exp} \to \beta\ \mathsf{exp} \\ \texttt{qint} &: \mathsf{int} \to \mathsf{int\ exp} \end{aligned}$$

These constructors do prohibit applying an integer because the first argument to `app` must have type $(\alpha \to \beta)$ exp but the type of an (object-language) integer is $\mathsf{int\ exp}$.

```
# app(qint 1, qint 2);;
Characters 4-18:
This expression has type int exp * int exp but is here
used with type
  ('a -> 'b) exp * 'a exp
#
```

```
module type EXP = sig
  type 'a exp
  val qint : int -> int exp
  val app  : ('a -> 'b) exp * 'a exp -> 'b exp
  val lam  : ('a exp -> 'b exp) -> ('a -> 'b) exp
  val pair : 'a exp * 'b exp -> ('a * 'b) exp
  val fst  : ('a * 'b) exp -> 'a exp
  val snd  : ('a * 'b) exp -> 'b exp
end

module Exp : EXP = struct
  type exp0 = INT  of int
            | VAR  of int
            | APP  of exp0 * exp0
            | LAM  of int * exp0
            | PAIR of exp0 * exp0
            | FST  of exp0
            | SND  of exp0
  type 'a exp = exp0

  let gensym = let  c = ref 0  in  fun () -> c := (!c) + 1; !c

  let qint i = INT i
  let app(a, b) = APP(a, b)
  let lam f = let x = gensym() in LAM(x, f(VAR x))
  let pair(a, b) = PAIR(a, b)
  let fst(a) = FST(a)
  let snd(a) = SND(a)
end
```

Figure 5.1: Typed abstract syntax

Lambda abstractions in the object language are built from implementation-language functions. This representation enables OCaml to mimic the implementation-language type at the object level, using the same type checker.

```
# app(lam(fun x -> x), qint 1);;
- : int exp = <abstr>
#
```

One application of the typed representation is in type-directed partial evaluation. If we replace the old, untyped representation with the new, typed representation in Figure 2.2, then OCaml's type system will deduce the type of the residual programs. For example

$$\mathsf{nbe}\ulcorner\bullet \to \mathsf{int}\urcorner \quad : \quad (\alpha\,\mathsf{exp} \to \mathsf{int}) \to (\alpha \to \mathsf{int})\,\mathsf{exp}$$

Exactly how the type τ exp is connected with (1) the type τ prog presented in Chapter 1, (2) the type τ exp from Chapter 3, and (3) the multi-level type-directed partial evaluation outlined in the paragraph on page 24 remains to be investigated. It also remains to be investigated which application benefit from the typed abstract syntax presented here.

Scaling up: The run-time code generator still needs to be applied to larger examples to show under which circumstances run-time code generation is an advantage. We are currently experimenting with semantics-directed compilation using run-time code generation and an interpreter for Peter Mosses's Action Semantics [15, 33]. A preliminary example show that the cost of run-time code generation is suppressed when the source action is run more than 1.7 times.

Perspectives: Traditional syntax-directed partial evaluation is a *source-to-source* translation, type-directed partial evaluation is a *compiled-to-source* translation, and compilation is a *source-to-compiled* translation. By composing type-directed partial evaluation with compilation we have obtained a *compiled-to-compiled* translation. By short-cutting the intermediate representation of source programs, we have reduced the compilation cycle of this translation enabling fast just-in-time code specialization.

Bibliography

[1] Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998.

[2] Alan Bawden. Quasiquotation in Lisp. In Danvy [12], pages 4–12.

[3] Anders Bondorf. Similix 5.0 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1993. Included in the Similix 5.0 distribution.

[4] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In Arvind, editor, *Proceedings of the Sixth ACM Conference on Functional Programming and Computer Architecture*, pages 308–317, Copenhagen, Denmark, June 1993. ACM Press.

[5] Anders Bondorf and Jens Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.

[6] Charles Consel. New insights into partial evaluation: the Schism experiment. In Harald Ganzinger, editor, *Proceedings of the Second European Symposium on Programming*, number 300 in Lecture Notes in Computer Science, pages 236–246, Nancy, France, March 1988.

[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] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volanshi. A uniform approach for compile-time and run-time specialization. In Danvy et al. [13], pages 54–72.

[9] Olivier Danvy. Type-directed partial evaluation. In Steele [40], pages 242–257.

[10] Olivier Danvy. Online type-directed partial evaluation. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 271–295, Kyoto, Japan, April 1998. World Scientific. Extended version available as the technical report BRICS RS-97-53.

[11] 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. Extended version available as the lecture note BRICS LN-98-3.

[12] Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, San Antonio, Texas, January 1999.

[13] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996. Springer-Verlag.

[14] Olivier Danvy and Morten Rhiger. Compiling actions by partial evaluation, revisited. Technical Report BRICS RS-98-13, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1998.

[15] Olivier Danvy and Morten Rhiger. Action-semantics-directed compiling by partial evaluation. Submitted for publication, 1999.

[16] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[17] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Steele [40], pages 258–283.

[18] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, second edition, 1996.

[19] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In Steele [40], pages 131–144.

[20] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag.

[21] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4), 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[22] Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. In PEPM'00 [35]. To appear.

[23] John Hatcliff and Robert Glück. An operational theory of self-applicable on-line program specialization. In Danvy et al. [13], pages 161–182.

[24] 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.

[25] 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.

[26] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[27] Mark Leone and Peter Lee. Lightweight run-time code generation. In Peter Sestoft and Harald Søndergaard, editors, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report 94/9, University of Melbourne, Australia, pages 97–106, Orlando, Florida, June 1994.

[28] 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.

[29] Xavier Leroy. *The Objective Caml system, release 2.02*. INRIA, Rocquencourt, France, March 1999.

[30] Steven McCanne. *The BPF Manual Page*. Lawrence Berkeley Laboratory, Berkeley, California, May 1991.

[31] Steven McCanne and Van Jacobson. The BSD Packet Filter: A new architecture for user-level packer capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, California, January 1993.

[32] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[33] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

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

[35] *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Boston, Massachusetts, January 2000. ACM Press. To appear.

[36] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[37] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In Danvy [12], pages 25–29.

[38] Kristoffer Rose. Type-directed partial evaluation using type classes. In Olivier Danvy and Peter Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Chalmers, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998.

[39] Michael Sperber and Peter Thiemann. Two for the price of one: composing partial evaluation and compilation. In Ron K. Cytron, editor, *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, pages 215–225, Las Vegas, Nevada, June 1997. ACM Press.

[40] Guy L. Steele, editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM Press.

[41] Eijiro Sumii and Naoki Kobayashi. Online-and-offline partial evaluation: A mixed approach. In PEPM'00 [35]. To appear.

[42] Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, Baltimore, Maryland, September 1998. ACM Press. Extended version available as the technical report BRICS RS-98-9.