Higher-Order Program Generation

Morten Rhiger

BRICS Dissertation Series DS-01-4

ISSN 1396-7002 August 2001

Higher-Order Program Generation

Morten Rhiger

Ph.D. Dissertation

BRICS

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

July 2001 Supervisor: Olivier Danvy

Higher-Order Program Generation

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
Ph.D. Degree.

Morten Rhiger
July 30, 2001

Abstract

This dissertation addresses the challenges of embedding programming languages, specializing generic programs to specific parameters, and generating specialized instances of programs directly as executable code. Our main tools are higher-order programming techniques and automatic program generation. It is our thesis that they synergize well in the development of customizable software.

Recent research on domain-specific languages propose to embed them into existing general-purpose languages. Typed higher-order languages have proven especially useful as meta languages because they provide a rich infrastructure of higher-order functions, types, and modules. Furthermore, it has been observed that embedded programs can be restricted to those having simple types using a technique called “phantom types.” We prove, using an idealized higher-order language, that such an embedding is sound (i.e., when all object-language terms that can be embedded into the meta language are simply typed) and that it is complete (i.e., when all simply typed object-language terms can be embedded into the meta language). The soundness proof is shown by induction over meta-language terms using a Kripke logical relation. The completeness proof is shown by induction over object-language terms. Furthermore, we address the use of Haskell and Standard ML as meta-languages.

Normalization functions, as embodied in type-directed partial evaluation, map a simply-typed higher-order value into a representation of its long beta-eta normal form. However, being dynamically typed, the original Scheme implementation of type-directed partial evaluation does not restrict input values to be typed at all. Furthermore, neither the original Scheme implementation nor the original Haskell implementation of type-directed partial evaluation guarantee that type-directed partial evaluation preserves types. We present three implementations of type-directed partial evaluation in Haskell culminating with a version that restricts the input to typed values and for which the proofs of type-preservation and normalization are automated.

Partial evaluation provides a solution to the disproportion between general programs that can be executed in several contexts and their specialized counterparts that can be executed efficiently. However, stand-alone partial evaluation is usually too costly when a program must be specialized at run time. We introduce a collection of byte-code combinators for OCaml, a dialect of ML, that provides run-time code generation for OCaml programs. We apply these byte-code combinators in semantics-directed compilation for an imperative language and in run-time specialization using type-directed partial evaluation.

Finally, we present an approach to compiling goal-directed programs, i.e., programs that backtrack and generate successive results: We first specify the semantics of a goal-directed language using a monadic semantics and a spectrum of monads. We then compile goal-directed 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.

Acknowledgements

The work that led this dissertation was carried out at the BRICS Ph.D. School at the University of Aarhus from the fall of 1998 to the summer of 2001.

I am grateful to Helmut Schwichtenberg, Peter Sestoft, and Mogens Nielsen for serving on my Ph.D. committee.

Foremost, however, I sincerely thank my supervisor Olivier Danvy for sharing with me his visions of computer science.

Andrzej Filinski's insights have greatly helped my understanding of the theory on programming languages. I am grateful for the numerous enlightening comments he provided on several parts of this dissertation.

Thanks are due to Jason Hickey who kindly hosted me for a pleasant half a year at the Department of Computer Science, Caltech, Pasadena. Also thanks to Alexey Nogin for unraveling the mysteries of MetaPRL.

I would like to thank my colleagues, Daniel Damian, Bernd Grobauer, Lasse R. Nielsen, and Zhe Yang, for many fruitful discussions at our weekly programming language meetings at BRICS. Thanks are also due to the occasional other participants to these meetings. Julia Lawall and John Reynolds, in particular, have been inspiring guests.

Finally, I wish to thank my wife Margrethe for her patience, understanding, and support.

Revised version.

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

Author's E-mail: `mrhiger@brics.dk`
Homepage: `http://www.brics.dk/~mrhiger`

Contents

List of Figures

Chapter 1

Introduction and motivations

Customizable software adapts to its context of application. To this end, its components are parameterized over the characteristics of the context and are instantiated to specific characteristics when they are applied. This notion of parameterization and instantiation is precisely captured by the notion of abstraction from the λ-calculus. The λ-notation is inherently "higher-order", i.e., abstractions may occur as arguments to and as results of other abstractions. Therefore, we use this term instead of, e.g., "object oriented."

Higher-order programming languages make it possible to respond to the challenge of writing customizable software because they support parameterization at their core. Reusable libraries of solutions to commonly occurring problems can typically be expressed as higher-order programs. Adaptive software components can be implemented in a higher-order programming language as a collection of common operations that can be instantiated in different environments. Higher-order functions, polymorphic types, and parameterized implementation modules, in particular, support the implementation of generic programs. Generic programs have a flaw, however: They are often less efficient than their specialized counterparts.

Automatic program generation techniques can alleviate the penalty of generic programming by constructing specialized instances of general software components. To also address the cost of generating specialized instances, specialized programs can be generated directly as executable code.

The need for adjustment and customization is not only reserved for programs. It also arises for programming languages. General-purpose languages will almost certainly fulfill the requirements of a certain application but it is sometimes desirable to use a specifically tailored language. Compilers or interpreters for such domain-specific languages can be implemented by standard means. Because these languages occur frequently, however, it has been suggested that they should be realized as extensions of existing general-purpose languages by an embedding of their domain-specific operations.

1.1 Themes

Several concepts recur throughout this dissertation.

1.1.1 The λ-calculus

In the late 1930's and early 1940's, Church invented the λ-calculus as a system for investigating the decision problem for the predicate calculus — the problem of finding effective methods to determine whether or not an expression is provable in the predicate calculus [20]. Here, an effective method is one which consists of a finite number of instructions that can be carried out without any ingenious knowledge and which ends with the desired result if carried out without error. Church showed that λ-definable functions cannot solve the decision problem for the predicate calculus. Since then, the λ-calculus has been an object of independent study in the areas of logics, semantics, and programming languages.

The three fundamental kinds of λ-term are variables, function abstraction, consisting of a variable and a term, and function application, consisting of two terms. For the pure λ-calculus, the only means of computation is that of a β-reduction: An application of an abstraction to an argument term can be replaced by the body of the abstraction instantiated to the argument term:

$$(\lambda x.\, e)e' \longrightarrow_\beta e[e'/x]$$

The left-hand side is called the redex and the right-hand side the contractum. When the λ-calculus is extended with constants, there are other reduction rules as well. Church's thesis states that this apparently simple notion of computation captures what it means for a function to be *effective*. The even simpler *combinatory* logic dispenses with bound variables but is compatible with the notion of effectiveness.

The λ-calculus can be viewed as either *typed* or *untyped*. In the untyped world, all λ-terms are valid, but repeatedly reducing a term may end in a *stuck* configuration instead of yielding a element of a designated set of *values*. In the typed world, a type system distinguishes *well-typed* λ-term from *ill-typed* ones. Repeatedly reducing a well-typed term is guaranteed not to end in a stuck configuration. Different type systems classify different sets of λ-terms as invalid. Type systems that classify many λ-terms as ill-typed may be inconvenient to work with. On the other hand, type systems that classify many λ-terms as well-typed may be difficult to implement or even undecidable.

- **Simple types**

  In the simply-typed λ-calculus, types τ consist of base types b and function types $\tau_1 \rightarrow \tau_2$. An abstraction $\lambda x.\, e$ has function type $\tau_1 \rightarrow \tau_2$ if the variable x has type τ_1 and the term e has type τ_2. An application $e_1\, e_2$ has type τ if the function term e_1 has type $\tau_2 \rightarrow \tau$ and the argument term e_2 has type τ_2.

  The type system sketched here is decidable. It is the core of most type systems for the λ-calculus.

- **Polymorphic types**

  Polymorphic type systems allows some terms to have more than one type. In the area of programming languages, one of the most frequently used polymorphic type systems extends the set of λ-terms with a polymorphic let-expression, $\mathbf{let}\ x = e\ \mathbf{in}\ e'$, and allows the type of the variable x to vary in ways consistent with the type of e'. More precisely, variables are assigned *type schemes* of form $\sigma = \forall\alpha_1, \ldots, \alpha_n.\tau$ and a type τ may be a polymorphic type identifier α. Each occurrence of variable may then be given a separate instance of the type scheme, $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$.

- **Dependent types**

  In dependent type systems, the result type of a function may depend on the value of the function argument. Dependent type systems are often undecidable.

Type systems describe methods by which it can be decided whether a term has a certain type. *Type checking* is the process of deciding whether a type-annotated term satisfy its annotations. The goal of *type reconstruction*, on the other hand, is to determine the type of an un-annotated term.

1.1.2 Higher-order programming languages

Several higher-order languages are intimately connected with the λ-calculus and are then often called *functional* languages. Functional languages can be classified according to several properties. The *evaluation order* determines which of possible several redexes in a program should be reduced. In particular, a *call-by-name* strategy selects the left-most outer-most redex and a *call-by-value* strategy selects the left-most inner-most redex. Given an expression, if both these strategies terminate with a value, then they produce the same results.

Dynamically typed languages correspond to the untyped λ-calculus. They do not require before-hand that evaluation a program always will yield a result instead of terminating with an error. In contrast, statically typed languages correspond to a typed λ-calculus. They only accept well-typed programs whose evaluations never terminate with an error. Statically typed languages are typically extensions of the polymorphically typed λ-calculus.

We shall mainly consider the following functional languages in this dissertation.

- **Scheme**

  Scheme is a dynamically typed functional language with a call-by-value evaluation strategy [89]. The core of Scheme consists of a small number of expressions which can be extended using hygienic macros.

  The base types of Scheme include booleans, numbers, strings, and symbols. Scheme programs can construct compound objects by grouping two values into a pair, also called a cons cell. Cons cells are used to represent compound data structures such as, e.g., lists and trees.

  The Scheme library provides primitive functions on numbers, strings, cons cells, etc. It also provides operations for primitive I/O and assignment operations.

  A unique feature of Scheme is that the text of a program is also the text of a piece of data representing the abstract syntax tree of the program. Such syntax trees are called S-expressions and are constructed using cons cells. S-expression are not limited to representing Scheme program.

- **Standard ML**

  Standard ML is a statically typed functional language with a call-by-value evaluation strategy [102]. Standard ML provides polymorphic let-expressions. There is no macro system for Standard ML.

  The base types of ML includes booleans, numbers, and strings. There are two ways a Standard ML program can construct compound objects. Tuples, or finite products, group together several values of possible different types. An element in a tuple can be accessed by providing its index. (The index must be fixed: It is not possible to iterate over a tuple.) Other compound values are given as elements of data types. A data type is a (possibly recursively defined) disjoint union of types. Data types are used to represent complex data structures such as, e.g., lists and trees. Functions and pattern matching are used to take values of data types apart. These functions are recursively defined when the data type is recursively defined.

  The Standard ML library provides primitive functions on numbers, strings, lists, etc. It also provides operations for primitive I/O and assignment operations.

  Standard ML does not support representing programs using S-expressions, such as Scheme. Instead, programs can be represented by a data type of abstract syntax trees. Such data types have the advantage over S-expressions that only syntactically correct abstract syntax trees can be represented. They do not, however, guarantee that only well-typed programs can be represented.

- **Haskell**

  Haskell is also a statically typed functional language [59]. The syntax and type system of Haskell is akin to the syntax and type system of Standard ML. But Haskell uses a call-by-name evaluation strategy rather than a call-by-value evaluation strategy. In addition, Haskell provides overloaded function symbols via type classes.

Higher-order programming languages embody the principles of abstraction and parameterization, i.e., that any syntactic category of the language can be named an parameterized [124]. Higher-order programs re-use a general pattern in many different places by instantiating its formal parameter to an actual entity. Scheme, Standard ML, and Haskell permit abstracting values over values; such an abstraction is a *function* that can be applied to produce a result. In higher-order languages, functions are ordinary values that can be abstracted over. Standard ML and Haskell permit abstracting types over types; such an abstraction is a *type constructor* that can be instantiated to an actual type. Haskell even permits a limited way of abstracting over type constructors. Standard ML permits abstracting over implementation modules; such an abstraction is a *functor* that can be applied to a module to produce another module.

Functional languages support several programming styles. Let us illustrate a few in Standard ML using a function that computes the total sum of a list of integers. All functions recursively traverse the argument list from left to right.

- **Direct style**

  This version of the sum function immediately returns the sum of the elements of its argument. The result of a recursive call over the remaining list is added to the current element and returned.

  ```
  (* sum : int list → int *)
  fun sum []       = 0
    | sum (x :: xs) = x + sum xs
  ```

- **Accumulator style**

  This version uses an auxiliary function which is passed a list and an *accumulator*. The accumulator holds the sum of the numbers seen so far.

  ```
  (* sum’ : int list → int → int *)
  fun sum’ []       ac = ac
    | sum’ (x :: xs) ac = sum xs (x + ac)

  fun sum xs = sum’ xs 0
  ```

- **Continuation-passing style**

  This version uses an auxiliary function which is passed a list and a *continuation*. The continuation represents the rest of the computation; in this case, adding a number to the sum of the elements of the rest of the list and then passing it to the continuation.

  ```
  (* sum’ : int list → (int → ’a) → a *)
  fun sum’ []       k = k 0
    | sum’ (x :: xs) k = sum xs (fn v ⇒ k (x + v))
  ```

  The continuation parameter has a polymorphic type. We can apply several different initial continuations, each with its own type of final result. The following examples return the computed sum (like in the direct-style and accumulator-style functions), print the sum, and check whether the sum is greater than 10, resp. On the right, we have displayed the type of the continuations.

  ```
  fun sum xs = sum’ xs (fn v ⇒ v)       (* int → int  *)
  fun sum xs = sum’ xs (fn v ⇒ print v) (* int → unit *)
  fun sum xs = sum’ xs (fn v ⇒ v > 10)  (* int → bool *)
  ```

Direct-style functions usually correspond to mathematical definitions by structural induction. Likewise, functions defined using an accumulator correspond to definitions by well-founded induction. Accumulator-style functions are often advantageous since they do not require the intermediate values to be remembered for each recursive call. Instead, they pass the hitherto computed value as an argument. Hence, accumulator-style functions can be executed without the need for a stack. Functions in continuation-passing style make evaluation order explicit. A continuation-passing style program has the same termination behavior when evaluated under call-by-name and call-by-value.

1.1.3 Interpretation and compilation

Programs are effective methods for solving specific problems. The fields of algorithmics and complexity concerns how algorithms, i.e., idealized representations of programs, solves specific problems and how efficient they are with respect to a given model of computation. The field of semantics is concerned with what programs compute. Yet, the majority of programs are written so that they can be executed on a physical machine.

Physical machines, or computers, are designed to run one kind of programs written in a machine language. Machine-language programs can be executed very efficiently. However, they cannot easily be ported to run on other machines, they are often cumbersome to write, and they only directly provide one paradigm of computation. Therefore, machine languages are seldom used directly as programming languages. Instead, programs are implemented in machine-independent languages that support the needed paradigm. There are then several approaches to running such a source program on a computer. (The following list is not exhaustive.)

- **Compilation**

  The program is first translated into machine language by a *compiler*. The machine-language program can be executed directly on a physical machine.

- **Interpretation**

  The source program is passed to an *interpreter* that already runs on a physical machine. When the interpreter runs, it simulates the execution of the source program.

- **Byte-code generation**

  In this hybrid approach, the program is first translated into another language, sometimes called a *byte-code language*, by a compiler. The result is passed to an interpreter, sometimes called a *virtual machine*, that simulates the execution of the source program.

Running an interpreter is usually slower than running a compiled program because the interpreter takes additional actions to parse and traverse the source program. On the other hand, the compilation approach exercises a cost of first translating the source program into machine language. The hybrid approach may seem as a step backwards: It requires the source program to be compiled but the result is still executed by an interpreter. There are advantages of the hybrid approach also, however. Most notably, both the byte-code compiler and the virtual machine are machine independent. In addition, byte-code programs may be considerably smaller than the corresponding machine-language programs [7]. Furthermore, programs in most languages cannot easily be simulated without some preprocessing phases performing, e.g, macro expansion and type checking.

1.1.4 Program transformations

A compiler is a semantics-preserving translator from one language to another. The purpose of compiling a source program is to obtain an efficient program that can be executed on a physical machine. Transformations for the sake of efficiency need not produce machine-language programs. For example, a reasonable pass in a compiler for a higher-order language may translate direct-style source functions into source functions using an accumulator. Most source-to-source transformations are motivated by speed or space but there are other reasons to transform one program into another.

- **Partial evaluation**

  A general, parameterized program is likely to be less efficient than a specialized program. Of course, the specialized program is not applicable in as many situation as the general program so it is desirable to write general programs while having efficient implementations. The goal of *partial evaluation* [83] is precisely to map a general, parameterized program and some fixed parameters into a specialized program.

  Partial evaluation is often able to produce specialized programs that are faster and require less memory to run than their general counterparts. Partial evaluation is particularly successful when most of the flow of control in the source program only depend on the fixed parameters.

  Ideally, partial evaluation should be an automatic program transformation. In practice, however, an expert user must often provide the partial evaluator with hints to ensure that partial evaluation terminates with an efficient specialized program.

  Partial evaluation has been applied in many areas. The *Futamura projections* describe how partial evaluation can be used in compiling and compiler generation given just an interpreter [65, 66]. To this end, an interpreter is viewed as a general program-execution mechanism capable of running any source program. Given an interpreter and a source program, partial evaluation thus yields a program-execution mechanism specialized to the source program — in effect, a compiled version of the source program. In a similar fashion, specializing *the partial evaluator* with respect to an interpreter yields a compiler from the interpreted language to the implementation language of the interpreter.

  Type-directed partial evaluation is an approach to partial evaluation for higher-order programs. A distinct feature of type-directed partial evaluation is its use of normalization functions to extract, from a higher-order *value*, the syntax of its normal form. Thus, the input to type-directed partial evaluation is not the text of a source program but its compiled, higher-order value.

- **Deforestation**

  Consider the following Standard ML function that generates a list of successive integers.

```
fun to i j = if i ≤ j then i :: to (i + 1) j else []
```

Combined with one of the the sum functions presented above, we can calculate the sum of a range of integers, $i + (i+1) + \cdots + j$, by evaluating `sum (to i j)`. (We shall not consider the obvious solution that computes $i(j-i+1)+(j-i)(j-i+1)/2$ directly.) But evaluating this expression will generate an intermediate list that is traversed once and then discarded. Such a solution incurs an overhead from constructing the list and from discarding it.

We can eliminate the intermediate list by combining the functions `sum` and `to` into one function `sumto` by a few steps involving the definitions of the original functions, here the direct-style sum function. Functional languages permit reasoning equationally on the syntax of program, as follows.

```
sumto ::  0 i j  =  sum (to i j)
                 =  sum (if i ≤ j then i :: to (i + 1) j else [])
                 =  if i ≤ j then sum (i :: to (i + 1) j) else sum []
                 =  if i ≤ j then i + sum (to (i + 1) j) else 0
                 =  if i ≤ j then i + sumto (i + 1) j else 0
```

We can therefore define a function to sum a list of successive integers as follows.

```
fun sumto i j = if i  ≤  j then i + sumto (i + 1) j else 0
```

This function does not construct an intermediate list and is therefore more efficient than combining the two general functions. This transformation is called *loop fusion* [16] or more generally *deforestation* [136].

- **Church encoding**

  Most functional languages provide data types for representations of compound values such as lists and trees. In contrast, the only "data types" of the pure λ-calculus are higher-order functions. Yet, through Church encodings it is possible to map the functionality of data types into the pure λ-calculus. The common idea is to represent data in a form where the constructors of the data type are abstracted away using λ's.

  The data type of lists have two constructors, `::` and `[]`. Traditional lists can thus be constructed using the following two values.

```
fun cons x xs = x  ::  xs
val nil       = []
```

  In contrast, a Church-representation of lists abstracts away the constructors, as follows.

```
fun cons x xs = fn c ⇒ fn n ⇒ c (x, xs c n)
val nil       = fn c ⇒ fn n ⇒ n
```

A list of elements of type τ is then represented as a polymorphic function of type $\forall\alpha, \beta. (\tau \times \alpha \to \beta) \to \alpha \to \beta$. Using the Church-encoded representation, we can sum a list of integers $i, i+1, \ldots, j$ by constructing a list and supplying an addition function and its identity element.

```
let val l = cons i (cons (i+1) ···(cons j nil)···)
in l (fn (x, y) ⇒ x + y) 0
end
```

Note that the list is not deforested. Instead, it is represented intermediately as a set of higher-order functions.

Church-encoded data types may expose properties of the values they represent. For example, the Church-encoded lists presented here directly encode the fold-function for lists into the alternative representation.

1.2 Dissertation outline

The rest of the dissertation is organized into five chapters.

- **Chapter 2. Embedded languages**

  During the 1990's, compilers and interpreters for domain-specific languages (i.e., languages dedicated to a particular application domain) have been implemented by adding the domain-specific operations to existing general-purpose languages. In these implementations, the general-purpose *meta* language provides the linguistic structure and the domain-specific *object* language provides objects and operations on them.

  Functional languages are particularly successful meta languages since they provide a rich infrastructure of higher-order functions, types, and modules. The primary goal of embedding an object language into a meta language is to provide the object-language *functionality* to end users. In addition, it has been suggested that also a part of the object-language infrastructure can be provided. In particular, type systems for simply-typed object languages have been embedded into polymorphically typed languages, such as Haskell and Standard ML, by parameterizing the meta-language type associated with embedded object-terms over their object-language types. This technique is called "phantom types."

  In Chapter 2 we investigate under which conditions such an embedding is *sound* (i.e., when all object-language terms that can be embedded into the meta language are simply typed) and *complete* (i.e., when all simply typed object-language terms can be embedded into the meta language). The chapter presents the syntax of a small functional meta language into which an object-language has been embedded. The denotational semantics of the meta language is given along with a representation of object terms. Soundness is then established using a Kripke logical relation and completeness is shown by structural induction. The chapter ends with a discussion of when Haskell and Standard ML can be used as meta-languages.

This chapter is based on an article submitted for publication [120].

- **Chapter 3. Type-directed partial evaluation**

  Type-directed partial evaluation extracts normal forms from values given their simple type. The original Scheme implementation, however, can not insist on the input value to have the given type since Scheme is dynamically typed. In fact, Scheme does not even insist on a well-typed input value at all.

  The first part of Chapter 3 presents an implementation of type-directed partial evaluation in Haskell which does insist on input values of the correct type. The challenge of the implementation is to implement the seemingly dependently typed normalization functions of type-directed partial evaluation in a language with Hindley-Milner polymorphism. The first solution, as shown in the first part of the chapter, is to use a Church-encoded representation of types instead of a data-type encoded representation.

  This part is based on work presented at PEPM 1999 [119].

  The second and third parts of Chapter 3 are concerned with the output from type-directed partial evaluation. Since it yields the normal form of its input, type-directed partial evaluation preserves types. This result follows as a corollary of the general correctness results for type-directed partial evaluation [61, 62]. In this second part, we present instead two implementations of type-directed partial evaluation in statically typed languages that provably preserve types. The first solution uses a Church-encoded representation of types. The second solution uses Haskell's overloading to avoid dependent types. In addition, both solutions also show that type-directed partial evaluation yields normal forms.

  This part is based on joint work with Olivier Danvy presented at FLOPS 2001 [42] and joint work with Olivier Danvy and Kristoffer H. Rose to appear in the Journal of Functional Programming [43].

- **Chapter 4. Run-time code generation**

  Partial evaluation addresses the trade-off between *generic* program components that can be adapted to the context of application and *specialized* program components that are efficient to use. But traditional partial evaluation generates specialized components that must be compiled before use. Adaptive systems address this discrepancy by using efficient run-time compilation techniques instead of stand-alone compilers.

  In Chapter 4 we present a collection of byte-code combinators that provides support for run-time code generation in OCaml, a dialect of ML. The byte-code combinators provide Lisp-like S-expressions in OCaml, but without the interpretive overhead of Lisp-like `eval` for running them. We apply the byte-code combinators in semantics-directed compilation and run-time specialization.

- **Chapter 5. Goal-directed evaluation**

  Goal-directed evaluation is built on the notions of backtracking and of generating successive results. In goal-directed languages, such as Icon and Snobol, evaluating an expression either *succeeds* with a value or it *fails*. A successful evaluation may be *resumed* to produce more results. When the evaluation of an expression fails, previously successful expressions are resumed.

  Goal-directed evaluation has always been a challenge to specify and implement. Icon has previously been specified using a continuation-based denotational semantics [71] and implemented by template-based compilers [113]. In Chapter 5 we specify a subset of Icon with a monadic semantics and using a spectrum of related monads. For example, we derive a continuation monad as a Church encoding of the list monad. The resulting semantics coincides with the existing continuation-based semantics of Icon.

  We then implement the continuation-based semantics as a continuation-passing style interpreter in Standard ML. This interpreter can directly be specialized to yield Standard ML programs. However, we also show how to compose partial evaluation with a translation from Standard ML to C program. This composition coincides with the existing template-based compilers for Icon. As a final back-end we also consider generating OCaml byte code as the output of partial evaluation.

  This chapter is based on joint work with Olivier Danvy and Bernd Grobauer presented at SAIG 2001 [37] and to appear in New Generation Computing [38].

- **Chapter 6. Conclusions and perspectives**

  Chapter 6 concludes and also discusses future work.

Chapter 2

Embedded languages

Recent work on embedding object languages into Haskell use "phantom types" (i.e., parameterized types whose parameter does not occur on the right-hand side of the type definition) to ensure that the embedded object-language terms are simply typed. But is it a safe assumption that *only* simply-typed terms can be represented in Haskell using phantom types? And conversely, can *all* simply-typed terms be represented in Haskell under the restrictions imposed by phantom types? In this chapter we investigate the conditions under which these assumptions are true: We show that these questions can be answered affirmatively for an idealized Haskell-like language and discuss to which extent Haskell can be used as a meta-language.

Note. This chapter is based on an article which is submitted for publication [120].

Thanks are due to Olivier Danvy and Andrzej Filinski for providing insightful comments and to Mikkel N. Hansen and Lasse R. Nielsen for kindly proof-reading earlier versions of this work.

2.1 Introduction

A program is typically written in terms of library routines. Once stabilized, it may itself become a library routine and be used in other programs. This bottom-up style of programming makes program development and maintenance easier and more efficient since the programmer can rely on well-understood routines. One of the goals of module systems, macro systems, and separate compilation is precisely to ease the definition of new routines and the use of existing routines in new programs.

Similarly, new programming languages typically arise from extending (or, as Steele says, "growing" [130]) existing languages with new features. Programming languages built from existing pieces are easier to design, implement, and understand since, in principle, they comprise well-understood components. For example, many realistic programming languages embody the λ-calculus as a core component. Yet, it was not until Scott that the λ-calculus itself was given a meaning. He warned that the hitherto "formalistic play with symbols" was useless and that, eventually, symbols must be given an interpretation [125]. Scott developed domain theory to provide a foundation for the λ-calculus. The result is the denotational semantics that we know it today in which the λ-calculus is used as a meta-language to define the semantics of programming languages [131].

A programming language can also be embedded into another, instead of directly giving its semantics in terms of, e.g., domains. The result combines the domain-specific operations (such as domain-specific values, their types, and operations on them) of the object language with the domain-independent linguistic features (such as evaluation strategy and type system) of the meta-language. (There is an unfortunate clash of terminology between formal *domains* as complete, partially ordered sets and informal *domains* as specific areas of applications.) Already in the 1960's, this style was envisioned by Landin [92] who observed that the design of programming languages splits into "the choice of written appearances of programs" and "the choice of abstract entities that can be referred to in the language."

Statically typed higher-order languages provide powerful domain-independent linguistic features. In particular, the module system, type classes, and polymorphic types of Haskell [59] make it a natural candidate to host domain-specific components [78]. Examples of domain-specific languages embedded into Haskell include languages for geometric region analysis [18], interactive 3D animation [53], image synthesis and manipulation [54], interfacing with Microsoft's Component Object Model [63, 85], music description and composition [79], and accessing SQL databases [93].

The domain-specific operations may be provided by an external system such as a graphical display showing images, a sound device playing music, or a database processing SQL queries; or they may be provided by an interpreter implemented in the meta-language. The domain-specific language may also provide a notion of well-typed domain-specific objects and restrict the range of the domain-specific operations to these objects. It is then crucial that only well-typed domain-specific objects can be expressed in the meta-language. When the object-language type system is sufficiently simple it can be expressed directly by the meta-language types of the domain-specific operations.

"Phantom types" were introduced to embed stronger object-language type into Haskell [63, 85, 93]. (We call a formal type parameter of a parameterized type "phantom" if all instances of the parameterized type are independent of the actual type parameters.) For example, phantom types are instrumental for embedding COM objects and for embedding SQL queries into Haskell. They also provide a key for using Haskell's type inferencer as a theorem prover to show that normalization functions as embodied in type-directed partial evaluation preserve types and yield normal forms [42, 43]. These applications of phantom types show that embedded type systems provide a powerful tool for both designing embedded programming languages and reasoning about program correctness. All existing embeddings using phantom types, however, take the following key properties for granted.

- **Soundness**

  No well-typed meta-language expression can represent an ill-typed domain-specific object; and

- **Completeness**

  Any well-typed domain-specific object can be represented in the meta-language.

In this chapter we formally prove that these properties hold when embedding a monomorphic, higher-order language into an idealized Haskell-like meta-language. This paper is organized as follows. Section 2.2 gives an implementation of an embedding of the terms and types of the simply typed λ-calculus into Haskell. In Section 2.3 we present a semantics of an idealized Haskell-like meta-language and then prove the two properties mentioned above. In Section 2.4 we investigate the use of Haskell as a meta-language, in partcular, which precautions to take when Haskell is used as a meta-language. This section also discusses extensions to the object and meta-languages. Section 2.5 concludes.

2.2 An embedded higher-order language

Let us consider a domain-specific object language for manipulating integers and higher-order functions, i.e., an extension of the simply typed λ-calculus. We include addition of integers but, depending on the domain, one can add other base types, finite products, lists, etc. and primitive operations on these types. In Haskell, the object language can be represented by the following data type.

```
type Ide = String
data Raw = INT Int | ADD Raw Raw
         | VAR Ide | LAM Ide Raw | APP Raw Raw

instance Show Raw where
  showsPrec _ t = ...
```

Here we have equipped the data type with a function showing terms as strings. An alternative to using a data type is to directly encode terms as strings (if, for example, terms are passed to an external system as concrete syntax).

2.2.1 Higher-order abstract syntax

The key to embedding a typed higher-order object language is to use higher-order abstract syntax [110] as the interface to constructing terms. In higher-order abstract syntax variables and bindings of the object language are modeled by variables and bindings of the meta-language, in this case Haskell. It is precisely this connection that enables meta-language types to model object-language types: Short of a typeable representation of type contexts for free variables, this appears an imposible task for first-order syntax. The higher-order implementation hides the (first-order) constructors `VAR` and `LAM`; instead a (higher-order) constructor `lam` groups together the construction of a symbolic variable and the lambda that binds it. Hence, only closed object-language terms can be constructed.

To facilitate automatic generation of variable names, terms are abstracted over a list of fresh variable names. The type of terms are `[Ide] → Raw`. The same list of variable names is

```
module Lambda(Exp, int, add, lam, app) where
  ...
  type Exp = [Ide] → Raw

  int i    = λns → INT i
  add a b  = λns → ADD (a ns) (b ns)
  lam f    = λ(n:ns) → LAM n (f (λz→VAR n) ns)
  app a b  = λns → APP (a ns) (b ns)
```

Figure 2.1: Higher-order abstract syntax.

passed to all subterms, except in the case of lambdas where the first name is used to construct a symbolic variable and the rest of the names are passed to the body of the lambda. Variables will therefore be named by their de Bruijn level [48]. Figure 2.1 shows how the higher-order abstract syntax is added on top of the raw first-order data type above.

The higher-order abstract syntax does not impose any type constraints on the embedded language. For example, Haskell accepts the expression

```
app (int 1) (lam (λx → x))
```

even though it represents the ill-typed object term `1 (λx → x)` (here presented in Haskell syntax).

2.2.2 An embedded type discipline

We restrict the higher-order constructors to only yield representations of well-typed terms. To this end, we make the following observations about constructing representations of well-typed object terms:

- `int` produces an object-language term of type `Int`.
- When `add` is applied to two object-language terms of type `Int` it produces an object-language term of type `Int`.
- When `app` is applied to two object-language terms of types `a → b` and `a` it produces a term of object-language type `b`.
- When `lam` is applied to a function mapping an object-language term of type `a` into an object-language term of type `b` it produces an object-language term of type `a → b`.

These dependencies are just verbal formulations of the standard type rules for the simply-typed λ-calculus. They suggest that the (polymorphic) types of the constructors actually could reflect the object-language types. We thus parameterize the type `Exp` over the type of the represented object-language term and we restrict the types of the constructors

```
module Lambda(Exp, int, add, lam, app) where

  type Ide = String
  data Raw = INT Int | ADD Raw Raw
           | VAR Ide | LAM Ide Raw | APP Raw Raw

  newtype Exp t = E ([Ide] → Raw)
  make (E a) ns = a ns

  int   :: Int → Exp Int
  add   :: Exp Int → Exp Int → Exp Int
  lam   :: (Exp a → Exp b) → Exp (a → b)
  app   :: Exp (a → b) → Exp a → Exp b

  int i     = E (λns → INT i)
  add a b   = E (λns → ADD (make a ns) (make b ns))
  lam f     = E (λ(n:ns) → LAM n (make (f (E (λz→VAR n))) ns))
  app a b   = E (λns → APP (make a ns) (make b ns))

  instance Show (Exp t) where
    showsPrec i (E a) = showsPrec i r where
      r = a [c:i | i←("":map show [1..]), c←['a'..'z']]
```

Figure 2.2: A typed higher-order language embedded into Haskell.

according to the observations above. The type parameter of Exp is a phantom type: it is not used in the right-hand side of the definition of Exp.

The complete embedding of the simply-typed λ-calculus into Haskell is shown in Figure 2.2. Terms are given by an abstract data type: a `newtype` declaration hides their representations as functions of type `[Ide] → Raw` and the module only exports the typed constructors.

As an example, Haskell rejects the expression `app (int 1) (lam (λx → x))`, since this expression is an attempt to represent the ill-typed object-language term `1 (λx → x)`. On the other hand, the object-language term `λf → 2 + (f 1)` has the type `(Int → Int) → Int`. It can thus be represented in Haskell by `lam (λf → add (int 2) (app f (int 1)))` of type `Exp ((Int → Int) → Int)`. It is, however, not clear just from the implementation in Figure 2.2 and the inferred types that *all* ill-typed terms are ruled out. It is also not clear that every well-typed term *can* be represented. In the next section we present an idealized Haskell-like meta-language in which these properties do hold.

2.3 Soundness and completeness

We consider an idealized meta-language without let-polymorphism but with a set of predefined polymorphic constant symbols. We first give the syntax, type system, and denotational semantics of the meta-language. We then introduce the syntax and type system of the simply typed λ-calculus (the object language) with constants of base type. Soundness follows from a proof using a Kripke logical relation and completeness fothe llows by structural induction. Section 2.4 discussed the differences between the idealized meta-language and Haskell.

2.3.1 A small functional meta-language

Let β range over a set $\mathbf{B}$ of base types. The types of the meta-language consist of base types, function types, and types of representations of terms.

$$\tau \quad ::= \quad \beta \mid \tau_1 \to \tau_2 \mid \mathsf{Exp}\,\tau$$

In the type $\mathsf{Exp}\,\tau$, the intention is that τ is the type of the represented object term (as also suggested by the examples at the end of Section 2.2). As we shall see, soundness implies that this τ cannot itself contain occurrences of Exp. This also means that the object language cannot itself express encoded terms, such as used in a multi-stage framework.

Let x range over an infinite set $\mathbf{V}$ of variable names and $c_{\tau_1\cdots\tau_n}$ over a set $\mathbf{C}$ of constant symbols. The syntax of our small language is then given as follows.

$$e \quad ::= \quad x \mid \lambda x.\, e \mid e_1\, e_2 \mid c_{\tau_1\cdots\tau_n} \mid \mathsf{rec}\, x.e$$

A signature $\Sigma = (\mathbf{B}, \mathbf{C})$ lists base types and constant symbols and additionally assigns types $\Sigma(c_{\tau_1\cdots\tau_n})$ to constants $c_{\tau_1\cdots\tau_n}$. (In other words, we treat $c_{\tau_1\cdots\tau_n}$ as an instance of a polymorphic constant symbol c.) A type context Γ is a finite mapping from variables to types. Given a signature Σ, the type rules for the language are as follows.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma[x : \tau_1] \vdash_\Sigma e : \tau_2}{\Gamma \vdash_\Sigma \lambda x.\, e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \tau_2 \to \tau \quad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma e_1\, e_2 : \tau}$$

$$\frac{\Sigma(c_{\tau_1\cdots\tau_n}) = \tau}{\Gamma \vdash_\Sigma c_{\tau_1\cdots\tau_n} : \tau} \qquad \frac{\Gamma[x : \tau] \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathsf{rec}\, x.e : \tau}$$

We define a base signature $\Sigma_0 = (\mathbf{B}_0, \mathbf{C}_0)$ for an addition function and for higher-order constructors,

$$\begin{array}{rcl} \mathbf{B}_0 & = & \{\mathsf{Int}\} \\ \mathbf{C}_0 & = & \mathbf{Z} \cup \{+, \mathsf{int}, \mathsf{add}\} \cup \{\mathsf{lam}_{\tau_1,\tau_2}, \mathsf{app}_{\tau_1,\tau_2}\}_{\tau_1,\tau_2} \end{array}$$

where $\mathbf{Z}$ is the set of integers. The associated assignment of types to constant symbols is given as follows. (This is where the types of the higher-order constructors are restricted.)

$$\begin{array}{rcl@{\qquad}rcl} \Sigma_0(i) & = & \mathsf{Int},\ \text{for each } i \in \mathbf{Z} & \Sigma_0(\mathsf{int}) & = & \mathsf{Int} \to \mathsf{Exp}\,\mathsf{Int} \\ \Sigma_0(+) & = & \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int} & \Sigma_0(\mathsf{add}) & = & \mathsf{Exp}\,\mathsf{Int} \to \mathsf{Exp}\,\mathsf{Int} \to \mathsf{Exp}\,\mathsf{Int} \end{array}$$

$$\begin{array}{rcl} \Sigma_0(\mathsf{lam}_{\tau_1,\tau_2}) & = & (\mathsf{Exp}\,\tau_1 \to \mathsf{Exp}\,\tau_2) \to \mathsf{Exp}\,(\tau_1 \to \tau_2) \\ \Sigma_0(\mathsf{app}_{\tau_1,\tau_2}) & = & \mathsf{Exp}\,(\tau_1 \to \tau_2) \to \mathsf{Exp}\,\tau_1 \to \mathsf{Exp}\,\tau_2 \end{array}$$

It is straightforward to extend the meta-language with other base types and constant symbols. The formal requirements that constant symbols must satisfy are discussed in Section 2.3.3.

2.3.2 Denotational semantics

To model the construction of object-language terms, we assume the existence of a discretely ordered set $\mathbf{L}$ that is capable of representing terms and we assume the existence of the following injective functions with mutually disjoint ranges,

$$\begin{array}{rclrcl} \mathit{INT} & \in & \mathbf{N} \to \mathbf{L} & \mathit{ADD} & \in & \mathbf{L} \times \mathbf{L} \to \mathbf{L} \\ \mathit{VAR} & \in & \mathbf{V} \to \mathbf{L} & \mathit{LAM} & \in & \mathbf{V} \times \mathbf{L} \to \mathbf{L} \\ \mathit{APP} & \in & \mathbf{L} \times \mathbf{L} \to \mathbf{L} & & & \end{array}$$

where $\mathbf{N}$ is the set of natural numbers. As an example one might take $\mathbf{L}$ to be, e.g., the set of finite strings of symbols. We need not require the constructor functions to be surjective. That is, we do not rule out “syntactically invalid” elements in $\mathbf{L}$.

We draw fresh object-language variables from the same source as meta-language variables, namely the infinite set $\mathbf{V}$. In particular, we assume that there is an injective function $\mathit{fresh} \in \mathbf{N} \to \mathbf{V}$.

In the semantical development that follows we shall use standard domain-theoretic notation: For CPOs A and B, we write $A_\perp$ for the lifted domain, $\perp_A$ for the bottom element of this domain, $\mathbf{up} \in A \to A_\perp$ for the lifting injection, $f^* \in A_\perp \to B_\perp$ for the strict extension of $f \in A \to B_\perp$, $f_\perp \in A_\perp \to B_\perp$ for the strict version of $f \in A \to B$. We shall write $(\cdot,\cdot)_\perp \in A_\perp \times B_\perp \to (A \times B)_\perp$ for the strict pairing function, e.g., the function for which $(\mathbf{up}(a), \mathbf{up}(b))_\perp = \mathbf{up}(a,b)$ and where $(a,b)_\perp = \perp_{A\times B}$ if either $a = \perp_A$ or $b = \perp_B$. We write $A \to_{\mathrm{c}} B$ for the continuous function space between A and B. The partial order on a domain A is $\sqsubset_A$.

An interpretation $\mathcal{I}$ of a signature Σ assigns predomains (i.e., bottomless CPOs) to base types and values to constant symbols. For a given interpretation $\mathcal{I}$, the meaning of types is defined as follows.

$$\begin{array}{rcl} [\![\beta]\!]_{\mathcal{I}} & = & \mathcal{I}(\beta)_\perp \\ [\![\tau_1 \to \tau_2]\!]_{\mathcal{I}} & = & [\![\tau_1]\!]_{\mathcal{I}} \to_{\mathrm{c}} [\![\tau_2]\!]_{\mathcal{I}} \\ [\![\mathsf{Exp}\,\tau]\!]_{\mathcal{I}} & = & \mathbf{N}_\perp \to_{\mathrm{c}} \mathbf{L}_\perp \end{array}$$

The interpretation furthermore assigns values to constant symbols such that if $\Sigma(c_{\tau_1\cdots\tau_n}) = \tau$ then $\mathcal{I}(c_{\tau_1\cdots\tau_n}) \in [\![\tau]\!]_{\mathcal{I}}$.

The meaning of a type context Γ is the labelled product

$$[\![\Gamma]\!]_{\mathcal{I}} = \prod_{x \in \mathrm{dom}(\Gamma)} [\![\Gamma(x)]\!]_{\mathcal{I}}$$

Finally, to any well typed meta-language expression $\Gamma \vdash_\Sigma e : \tau$ we assign a continuous function $[\![e]\!]_{\mathcal{I}} \in [\![\Gamma]\!]_{\mathcal{I}} \to_{\mathrm{c}} [\![\tau]\!]_{\mathcal{I}}$. The following is a standard call-by-name semantics for functional languages.

$$\begin{array}{rcl} [\![x]\!]_{\mathcal{I}}\,\rho & = & \rho(x) \\ [\![\lambda x.\,e]\!]_{\mathcal{I}}\,\rho & = & \lambda a.\,[\![e]\!]_{\mathcal{I}}\,\rho[x \mapsto a] \\ [\![e_1\,e_2]\!]_{\mathcal{I}}\,\rho & = & [\![e_1]\!]_{\mathcal{I}}\,\rho\,([\![e_2]\!]_{\mathcal{I}}\,\rho) \\ [\![c_{\tau_1\cdots\tau_n}]\!]_{\mathcal{I}}\,\rho & = & \mathcal{I}(c_{\tau_1\cdots\tau_n}) \\ [\![\mathsf{rec}\,x.e]\!]_{\mathcal{I}}\,\rho & = & \bigsqcup_{i\in\omega} \phi^i(\perp_{[\![\tau]\!]_{\mathcal{I}}}), \quad \text{where } \phi(a) = [\![e]\!]_{\mathcal{I}}\,\rho[x \mapsto a] \end{array}$$

The initial signature Σ_0 is given the interpretation $\mathcal{I}_0$: First, the type Int is interpreted by the integers, i.e., $\mathcal{I}_0(\mathsf{Int}) = \mathbf{Z}$. For the constant symbols, the interpretation is defined as follows.

$$\begin{array}{rcl}
\mathcal{I}_0(i) & = & \mathbf{up}(i) \\
\mathcal{I}_0(+) & = & \lambda x.\, \lambda y.\, x +_\perp y \\
\mathcal{I}_0(\mathsf{int}) & = & \lambda i.\, \lambda n.\, \mathit{INT}_\perp(i) \\
\mathcal{I}_0(\mathsf{add}) & = & \lambda v_1.\, \lambda v_2.\, \lambda n.\, \mathit{ADD}_\perp(v_1(n), v_2(n))_\perp \\
\mathcal{I}_0(\mathsf{lam}_{\tau_1,\tau_2}) & = & \lambda f.\, \lambda n.\, \mathit{LAM}_\perp(\mathit{fresh}_\perp(n), \\
 & & \qquad f\,(\lambda z.\ \mathit{VAR}_\perp(\mathit{fresh}_\perp(n)))\,(n +_\perp \mathbf{up}(1)))_\perp \\
\mathcal{I}_0(\mathsf{app}_{\tau_1,\tau_2}) & = & \lambda v_1.\, \lambda v_2.\, \lambda n.\, \mathit{APP}_\perp(v_1(n), v_2(n))_\perp
\end{array}$$

It follows by construction that the functions involved in defining the semantics of terms and in defining the initial interpretation of the constant symbols are all continuous.

2.3.3 Soundness

To reason about object-language types and terms we use the following concrete representations.

$$\begin{array}{rcl}
\sigma & ::= & \mathsf{Int} \mid \sigma_1 \to \sigma_2 \\
u & ::= & i \mid u_1 + u_2 \mid x \mid \lambda x.\, u \mid u_1\, u_2
\end{array}$$

We let Δ range over finite mappings from variables to object language types. The following type rules assigns types to object-language terms.

$$\frac{}{\Delta \vdash i : \mathsf{Int}} \qquad \frac{\Delta \vdash u_1 : \mathsf{Int} \quad \Delta \vdash u_2 : \mathsf{Int}}{\Delta \vdash u_1 + u_2 : \mathsf{Int}}$$

$$\frac{\Delta(x) = \sigma}{\Delta \vdash x : \sigma} \qquad \frac{\Delta[x : \sigma_1] \vdash u : \sigma_2}{\Delta \vdash \lambda x.\, u : \sigma_1 \to \sigma_2} \qquad \frac{\Delta \vdash u_1 : \sigma_1 \to \sigma_2 \quad \Delta \vdash u_2 : \sigma_1}{\Delta \vdash u_1\, u_2 : \sigma_2}$$

These rules are standard and standard results apply to them. We shall only need the following weakening lemma.

Lemma 1 (Weakening) *If for all $x \in \mathrm{dom}(\Delta)$, $\Delta'(x) = \Delta(x)$ and $\Delta \vdash u : \sigma$ then $\Delta' \vdash u : \sigma$*

Proof. By induction on the derivation of $\Delta \vdash u : \sigma$. □

We define the obvious injective representation functions mapping object-language terms into $\mathbf{L}$ as follows.

$$\begin{array}{rclcrcl}
\lceil i \rceil & = & \mathit{INT}(i) & \qquad & \lceil u_1 + u_2 \rceil & = & \mathit{ADD}(\lceil u_1 \rceil, \lceil u_2 \rceil) \\
\lceil x \rceil & = & \mathit{VAR}(x) & & \lceil \lambda x.\, u \rceil & = & \mathit{LAM}(x, \lceil u \rceil) \\
\lceil u_1\, u_2 \rceil & = & \mathit{APP}(\lceil u_1 \rceil, \lceil u_2 \rceil) & & & &
\end{array}$$

This representation function need not be surjective: Some elements in $\mathbf{L}$ may not correspond to any object-language term. It is our goal to show, however, that any element of $\mathbf{L}$ that is denoted by an expression in the meta-language corresponds to an object-language term and, furthermore, that these terms are well typed.

Definition 1 *For a type context Δ and a type σ define a subset of $\mathbf{L}_\bot$ as follows.*

$$\mathcal{T}_\sigma^\Delta = \{\bot_\mathbf{L}\} \cup \{\mathbf{up}(l) \in \mathbf{L}_\bot \mid \exists u.\lceil u \rceil = l \wedge \Delta \vdash u : \sigma\}$$

The set $\mathcal{T}_\sigma^\Delta$ contains exactly the elements in $\mathbf{L}_\bot$ that are undefined or that correspond to object-language terms of type σ in type context Δ.

Lemma 2 (Admissibility of $\mathcal{T}$) *For any type τ and type context Δ the relation $\mathcal{T}_\tau^\Delta$ is admissible. That is, it is pointed (i.e., $\bot_\mathbf{L} \in \mathcal{T}_\tau^\Delta$) and it is chain complete (i.e., if $(l_i)_{i\in\omega}$ is a chain in $\mathbf{L}_\bot$ such that for all $i \in \omega$, $l_i \in \mathcal{T}_\tau^\Delta$ then $\bigsqcup_{i\in\omega} l_i \in \mathcal{T}_\tau^\Delta$).*

Proof. Pointedness follows trivially by the definition. Chain completeness follows from the fact that $\mathbf{L}_\bot$ is discretely ordered and that any chain in $\mathbf{L}_\bot$ therefore eventually becomes constant. □

Definition 2 *A* world *is a type context Δ. Worlds are ordered as follows.*

$$\Delta' \succeq \Delta \iff \forall x \in \mathrm{dom}(\Delta).x \in \mathrm{dom}(\Delta') \wedge \Delta'(x) = \Delta(x)$$

We let $\#\Delta = \max(\{0\} \cup \{i \mid \mathit{fresh}(i) \in \mathrm{dom}(\Delta)\})$.

It is easy to show that $\succeq$ is reflexive and transitive, that $\Delta' \succeq \Delta$ implies $\#\Delta' \geq \#\Delta$, and that $n > \#\Delta$ implies $n + 1 > \#\Delta[g : \sigma]$ where $g = \mathit{fresh}(n)$.

Terms that are well typed in one world are also well typed in any larger world, a result due to weakening.

Lemma 3 (Monotonicity of $\mathcal{T}$) *For type σ, if two type contexts satisfy $\Delta' \succeq \Delta$ then $\mathcal{T}_\tau^{\Delta'} \supseteq \mathcal{T}_\tau^\Delta$.*

Proof. A consequence of Lemma 1. □

Definition 3 (Logical relation $\mathcal{R}$) *For any type τ and type context Δ we define a subset of $[\![\tau]\!]_\mathcal{I}$ as follows.*

(1) $\mathcal{R}_\beta^\Delta = [\![\beta]\!]_\mathcal{I}$

(2) $\mathcal{R}_{\tau_1\to\tau_2}^\Delta = \{f \in [\![\tau_1]\!]_\mathcal{I} \to_\mathrm{c} [\![\tau_2]\!]_\mathcal{I} \mid \forall\Delta' \succeq \Delta.\forall a \in \mathcal{R}_{\tau_1}^{\Delta'}.f(a) \in \mathcal{R}_{\tau_2}^{\Delta'}\}$

(3) $\mathcal{R}_{\mathsf{Exp}\,\sigma}^\Delta = \{f \in \mathbf{N}_\bot \to_\mathrm{c} \mathbf{L}_\bot \mid \forall n > \#\Delta.f(\mathbf{up}(n)) \in \mathcal{T}_\sigma^\Delta\}$

The logical relation defines a notion of well-behaved values. Informally, it states that all values of base type are well-behaved, that a function is well-behaved if it maps well-behaved values to well-behaved result, and that a representation of an object term is well-behaved if, given a fresh variable index, it has the correct type in the given type context.

Lemma 4 (Admissibility of $\mathcal{R}_\tau$) *For any type τ and type context Δ the relation $\mathcal{R}_\tau^\Delta$ is admissible.*

Proof. Using admissibility of $\mathcal{T}_\tau^\Delta$ (Lemma 2). Chain completeness follows by induction on τ. For the case $\tau = \beta$ we use the fact $\mathcal{R}_\beta^\Delta$ is the constantly true predicate. For the case $\tau = \mathsf{Exp}\,\sigma$ we use chain completeness of $\mathcal{T}_\tau^\Delta$ and the fact that for any chain of continuous functions $(f_n)_{n\in\omega}$, $\left(\bigsqcup_{n\in\omega} f_n\right)(x) = \bigsqcup_{n\in\omega} f_n(x)$. Pointedness follows by induction on τ using pointedness of $\mathcal{T}_\tau^\Delta$ and $[\![\beta]\!]_\mathcal{I}$. $\square$

Together with Definition 3, the following lemma shows that $\mathcal{R}_\tau$ is a Kripke logical relation [103].

Lemma 5 (Monotonicity of $\mathcal{R}_\tau$) *For any type τ, if two type contexts satisfy $\Delta' \succeq \Delta$ then $\mathcal{R}_\tau^{\Delta'} \supseteq \mathcal{R}_\tau^\Delta$.*

Proof. By induction on τ.

Case $\tau = \beta$. Holds trivially since $\mathcal{R}_\beta^{\Delta'} = \mathcal{R}_\beta^\Delta$.

Case $\tau = \tau_1 \to \tau_2$. Follows from the transitivity of $\succeq$.

Case $\tau = \mathsf{Exp}\,\sigma$. Follows from the monotonicity of $\mathcal{T}$ (Lemma 3) and using the fact that $\#\Delta' \geq \#\Delta$. $\square$

We extend the logical relation of values and types to a relation of environments and type contexts. This gives a notion of well-behaved environments.

Definition 4 *For any type contexts Δ and Γ we define a subset of $[\![\Gamma]\!]_\mathcal{I}$ as follows.*

$$\mathcal{R}_\Gamma^\Delta = \{\rho \in [\![\Gamma]\!]_\mathcal{I} \mid \forall x \in \mathrm{dom}(\Gamma).\rho(x) \in \mathcal{R}_{\Gamma(x)}^\Delta\}$$

This extension preserves monotonicity.

Lemma 6 (Monotonicity of $\mathcal{R}_\Gamma$) *For all type contexts Γ, Δ, and Δ', if $\Delta' \succeq \Delta$ then $\mathcal{R}_\Gamma^{\Delta'} \supseteq \mathcal{R}_\Gamma^\Delta$.*

Proof. Follows from Lemma 5. $\square$

Adding a well-behaved value to an already well-behaved environment results in another well-behaved environment.

Lemma 7 *If $d \in \mathcal{R}_\tau^\Delta$ and $\rho \in \mathcal{R}_\Gamma^\Delta$ then also $\rho[x \mapsto d] \in \mathcal{R}_{\Gamma[x:\tau]}^\Delta$.*

Proof. Follows from Definition 4. $\square$

Using the results established so far, soundness amounts to showing that the semantics of a well-typed expression is well-behaved. The following lemma shows that this result holds for the constant symbols defined by the initial interpretation.

Lemma 8 *For any constant symbol $c_{\tau_1\cdots\tau_n} \in \mathbf{C}_0$ with $\Sigma_0(c_{\tau_1\cdots\tau_n}) = \tau$ and for any type context Δ we have $\mathcal{I}_0(c_{\tau_1\cdots\tau_n}) \in \mathcal{R}_\tau^\Delta$.*

Proof. By analysis of the individual constant symbols.

Case $c_{\tau_1 \cdots \tau_n} = i$. Holds trivially since all values of base type are well behaved.

Case $c_{\tau_1 \cdots \tau_n} = +$. Holds trivially since $+$ produces a value of base type which is always well behaved.

Case $c_{\tau_1 \cdots \tau_n} = \mathsf{int}$. Since $\Delta \vdash i : \mathsf{Int}$ we have $\mathbf{up}(INT(i)) \in \mathcal{T}^{\Delta}_{\mathsf{Int}}$ for any $i \in \mathbf{Z}$ and type context Δ. Therefore $\lambda i.\, \lambda n.\, INT_\bot(\mathbf{up}(i)) = \lambda i.\, \lambda n.\, \mathbf{up}(INT(i)) \in \mathcal{R}^{\Delta}_{\mathsf{Int} \to \mathsf{Exp\,Int}}$ as required.

Case $c_{\tau_1 \cdots \tau_n} = \mathsf{add}$. Given $\Delta' \succeq \Delta$, $\Delta'' \succeq \Delta'$, $v_1 \in \mathcal{R}^{\Delta'}_{\mathsf{Exp\,Int}}$, $v_2 \in \mathcal{R}^{\Delta''}_{\mathsf{Exp\,Int}}$, and $n > \#\Delta''$ we must show that $ADD_\bot(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\bot$ is either $\bot_\mathbf{L}$ or equal to $\mathbf{up}(\lceil u \rceil)$ for some u with $\Delta'' \vdash u : \mathsf{Int}$.

Since $n > \#\Delta'' \geq \#\Delta' \geq \#\Delta$ we immediately have $v_2(\mathbf{up}(n)) \in \mathcal{T}^{\Delta''}_{\mathsf{Int}}$. Using Lemma 3 we also get $v_1(\mathbf{up}(n)) \in \mathcal{T}^{\Delta'}_{\mathsf{Int}} \subseteq \mathcal{T}^{\Delta''}_{\mathsf{Int}}$. Therefore, either $v_1(\mathbf{up}(n)) = \bot_\mathbf{L}$ or $v_2(\mathbf{up}(n)) = \bot_\mathbf{L}$, in which case we are done since $ADD_\bot(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\bot = \bot_\mathbf{L}$, or we have terms u_1 and u_2 with $v_1(\mathbf{up}(n)) = \mathbf{up}(\lceil u_1 \rceil)$, $v_2(\mathbf{up}(n)) = \mathbf{up}(\lceil u_2 \rceil)$, $\Delta'' \vdash u_1 : \mathsf{Int}$, and $\Delta'' \vdash u_2 : \mathsf{Int}$. We set $u = u_1 + u_2$ and have

$$ADD_\bot(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\bot = \mathbf{up}\lceil u_1 + u_2 \rceil$$

and

$$\frac{\Delta'' \vdash u_1 : \mathsf{Int} \quad \Delta'' \vdash u_2 : \mathsf{Int}}{\Delta'' \vdash u_1 + u_2 : \mathsf{Int}}$$

as required.

Case $c_{\tau_1 \cdots \tau_n} = \mathsf{lam}_{\tau_1,\tau_2}$. Given $\Delta' \succeq \Delta$, $n > \#\Delta'$, $g = \mathit{fresh}(n)$, and $f \in \mathcal{R}^{\Delta'}_{\mathsf{Exp}\,\tau_1 \to \mathsf{Exp}\,\tau_1}$ we must show that

$$LAM_\bot(\mathbf{up}(g), f\,(\lambda z.\, VAR_\bot(\mathbf{up}(g)))\,(\mathbf{up}(n+1)))_\bot$$

is either $\bot_\mathbf{L}$ or equal to $\mathbf{up}(\lceil u \rceil)$ for some u with $\Delta' \vdash u : \tau_1 \to \tau_2$.

We have $\lambda z.\, VAR_\bot(\mathbf{up}(g)) = \lambda z.\, \mathbf{up}(VAR(g)) \in \mathcal{R}^{\Delta'[g:\tau_1]}_{\mathsf{Exp}\,\tau_1}$ since $\lceil g \rceil = VAR(g)$ and $\Delta'[g : \tau_1] \vdash g : \tau_1$. Since also $n + 1 > \#(\Delta'[g : \tau_1])$ we have

$$f\,(\lambda z.\, VAR_\bot(\mathbf{up}(g)))\,(\mathbf{up}(n+1)) \in \mathcal{T}^{\Delta'[g:\tau_1]}_{\tau_2}$$

This value must therefore either be $\bot_\mathbf{L}$, in which case we are done, or be equal to $\mathbf{up}(\lceil u' \rceil)$ for some term u' with $\Delta'[g : \tau_1] \vdash u' : \tau_2$. We set $u = \lambda g.\, u'$ and have

$$LAM_\bot(\mathbf{up}(g), f\,(\lambda z.\, VAR_\bot(\mathbf{up}(g)))\,(\mathbf{up}(n+1)))_\bot = \mathbf{up}(\lceil \lambda g.\, u' \rceil)$$

and

$$\frac{\Delta'[g : \tau_1] \vdash u' : \tau_2}{\Delta' \vdash \lambda g.\, u' : \tau_1 \to \tau_2}$$

as required.

Case $c_{\tau_1\cdots\tau_n} = \mathsf{app}_{\tau_1,\tau_2}$. (Follows the same structure as the case for addition.) Given $\Delta' \succeq \Delta$, $\Delta'' \succeq \Delta'$, $v_1 \in \mathcal{R}^{\Delta'}_{\mathsf{Exp}\,(\tau_1\to\tau_2)}$, $v_2 \in \mathcal{R}^{\Delta''}_{\mathsf{Exp}\,\tau_1}$, and $n > \#\Delta''$ we must show that

$$APP_\perp(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\perp$$

is either $\perp_{\mathbf{L}}$ or equal to $\mathbf{up}(\ulcorner u \urcorner)$ for some u with $\Delta'' \vdash u : \tau_2$

Since $n > \#\Delta'' \geq \#\Delta' \geq \#\Delta$ we immediately have $v_2(\mathbf{up}(n)) \in \mathcal{T}^{\Delta''}_{\tau_1}$. Using Lemma 3 we also get $v_1(\mathbf{up}(n)) \in \mathcal{T}^{\Delta'}_{\tau_1\to\tau_2} \subseteq \mathcal{T}^{\Delta''}_{\tau_1\to\tau_2}$. Thus, either $v_1(\mathbf{up}(n)) = \perp_{\mathbf{L}}$ or $v_2(\mathbf{up}(n)) = \perp_{\mathbf{L}}$, in which case we are done since $APP_\perp(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\perp = \perp_{\mathbf{L}}$, or we have terms u_1 and u_2 with $v_1(\mathbf{up}(n)) = \mathbf{up}(\ulcorner u_1 \urcorner)$, $v_2(\mathbf{up}(n)) = \mathbf{up}(\ulcorner u_2 \urcorner)$, $\Delta'' \vdash u_1 : \tau_1 \to \tau_2$, and $\Delta'' \vdash u_2 : \tau_1$. We set $u = u_1\, u_2$ and have

$$APP_\perp(v_1(\mathbf{up}(n)), v_2(\mathbf{up}(n)))_\perp = \mathbf{up}\ulcorner u_1\, u_2 \urcorner$$

and

$$\frac{\Delta'' \vdash u_1 : \tau_1 \to \tau_2 \quad \Delta'' \vdash u_2 : \tau_1}{\Delta'' \vdash u_1\, u_2 : \tau_2}$$

as required. □

Other constant symbols can be added to the meta-language if they satisfy Lemma 8. The main result states that evaluating a well-typed expression in a well-behaved environment yields a well-behaved value.

Theorem 1 (Soundness) *In the initial interpretation $\mathcal{I}_0$ of the initial signature Σ_0, if $\Gamma \vdash_{\Sigma_0} e : \tau$ and $\rho \in \mathcal{R}^{\Delta}_{\Gamma}$ then $[\![e]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^{\Delta}_{\tau}$.*

Proof. By structural induction on e.

Case $e = x$. Then $\Gamma(x) = \tau$ and $[\![x]\!]_{\mathcal{I}_0}\, \rho = \rho(x)$ so $[\![x]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^{\Delta}_{\tau}$ follows from Definition 4.

Case $e = \lambda x.\, e'$. Then $\tau = \tau_1 \to \tau_2$ where $\Gamma[x : \tau_1] \vdash_{\Sigma_0} e' : \tau_2$ and

$$[\![\lambda x.\, e']\!]_{\mathcal{I}_0} = \lambda a.\, [\![e']\!]_{\mathcal{I}_0}\, \rho[x \mapsto a]$$

We must show $[\![\lambda x.\, e']\!]_{\mathcal{I}_0} \in \mathcal{R}^{\Delta}_{\tau_1\to\tau_2}$. So given $\Delta' \succeq \Delta$ and $d \in \mathcal{R}^{\Delta'}_{\tau_1}$ we must show that $[\![e']\!]_{\mathcal{I}_0}\, \rho[x \mapsto d] \in \mathcal{R}^{\Delta'}_{\tau_2}$. From Lemma 6 it follows that $\rho \in \mathcal{R}^{\Delta'}_{\Gamma}$ and then from Lemma 7 we have that $\rho[x \mapsto d] \in \mathcal{R}^{\Delta'}_{\Gamma[x:\tau_1]}$. Thus, $[\![e']\!]_{\mathcal{I}_0}\, \rho[x \mapsto d] \in \mathcal{R}^{\Delta'}_{\tau_2}$ follows from the induction hypothesis.

Case $e = e_1\, e_2$. Then $\Gamma \vdash_{\Sigma_0} e_1 : \tau_2 \to \tau$, $\Gamma \vdash_{\Sigma_0} e_2 : \tau_2$, and $[\![e_1\, e_2]\!]_{\mathcal{I}_0}\, \rho = [\![e_1]\!]_{\mathcal{I}_0}\, \rho\, ([\![e_2]\!]_{\mathcal{I}_0}\, \rho)$. Using the induction hypothesis twice we get $[\![e_1]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^{\Delta}_{\tau_2\to\tau}$ and $[\![e_2]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^{\Delta}_{\tau_2}$. Using reflexivity of $\succeq$, Definition 3(2) gives $[\![e_1\, e_2]\!]_{\mathcal{I}_0}\, \rho \in \mathcal{R}^{\Delta}_{\tau}$ as required.

Case $e = c_{\tau_1\cdots\tau_n}$. Follows from Lemma 8.

Case $\mathsf{rec}\, x.e'$. Using pointedness for the base case and Lemma 7 for the induction step an induction on i shows that $\phi^i(\perp) \in \mathcal{R}^{\Delta}_{\tau}$ for all i, where $\phi(a) = [\![e']\!]_{\mathcal{I}_0}\, \rho[x \mapsto a]$. The result then follows from admissibility (Lemma 4). □

The following corollary states that an expression of type $\mathsf{Exp}\,\tau$ evaluates to a function that, when passed an initial variable index, either diverges or yields a representation of an object-language term of type τ.

Corollary 1 *For any type σ and closed expression e with $\emptyset \vdash_{\Sigma_0} e : \mathsf{Exp}\,\sigma$, if $[\![e]\!]_{\mathcal{I}_0}\,[\,]\,0 = l$ then either $l = \bot_{\mathbf{L}}$ or $l = \mathbf{up}(\lceil u \rceil)$ for some term u with $\emptyset \vdash u : \sigma$.*

Proof. Follows from Theorem 1 since $[\,] \in \mathcal{R}^{\emptyset}_{\emptyset}$. □

2.3.4 Completeness

We show that for any well-typed object-language term there exists a meta-language expression that evaluates to a representation of the term. The translation straightforwardly uses int, add, $\mathsf{lam}_{\tau_1,\tau_2}$, and $\mathsf{app}_{\tau_1,\tau_2}$ to build integers, additions, lambdas, and applications. We present the translation as an extended type system so that the polymorphic constant symbols are indexed by the correct types. (In languages such as Haskell, where constant symbols are not annotated by the polymorphic type, the translation can be defined by a simple structural induction over the object-language term.) In addition, the type rules carry symbolic variable indices so that these can be related to the corresponding meta-language variables in the proofs below. (This extra information is also not needed to just perform the translation.)

We let a translation context Ξ range over finite mappings from variables to pairs of types and integers. The translation rules are given as follows.

$$\frac{}{n;\Xi \vdash i : \mathsf{Int} \,/\, \mathsf{int}\; i} \qquad \frac{n;\Xi \vdash u_1 : \mathsf{Int} \,/\, e_1 \quad n;\Xi \vdash u_2 : \mathsf{Int} \,/\, e_2}{n;\Xi \vdash u_1 + u_2 : \mathsf{Int} \,/\, \mathsf{add}\; e_1\; e_2}$$

$$\frac{\Xi(x) = (\sigma, m)}{n;\Xi \vdash x : \sigma \,/\, x} \qquad \frac{n+1;\Xi[x : (\sigma_1, n)] \vdash u : \sigma_2 \,/\, e}{n;\Xi \vdash \lambda x.\, u : \sigma_1 \to \sigma_2 \,/\, \mathsf{lam}_{\sigma_1,\sigma_2}\,(\lambda x.\, e)}$$

$$\frac{n;\Xi \vdash u_1 : \sigma_1 \to \sigma_2 \,/\, e_1 \quad n;\Xi \vdash u_2 : \sigma_1 \,/\, e_2}{n;\Xi \vdash u_1\, u_2 : \sigma_2 \,/\, \mathsf{app}_{\sigma_1,\sigma_2}\, e_1\, e_2}$$

We first relate the type of the object-language term and the type of the meta-language expression.

Lemma 9 *For any translation context Ξ, type context Γ, $n \in \mathbf{Z}$, and type σ, if $n;\Xi \vdash u : \sigma \,/\, e$ and if $\Gamma(x) = \mathsf{Exp}\,\sigma$ when $\Xi(x) = (\sigma, n)$ then $\Gamma \vdash_{\Sigma_0} e : \mathsf{Exp}\,\sigma$*

Proof. By induction on the derivation of $n;\Xi \vdash u : \sigma \,/\, e$. □

Not all well-typed object-language terms can be encoded *syntactically* in the meta-language since fresh variable names drawn from the predetermined list might differ from the intended variable names. We will show, however, that a term and its encoding as given above are equal up to renaming of bound variables.

Definition 5 (Substitutions and α-convertibility) *A* substitution *s is a finite mapping of variables to variables. We let $(s\backslash x)$ be the restricted substitution satisfying $(s\backslash x)\,(x) = x$ and $(s\backslash x)\,(y) = s(y)$ for $x \neq y$. Substitutions extend to terms in such a way that $s(\lambda x.\, u) = \lambda x.\, u'$ where $u' = (s\backslash x)\,(u)$.*

*We say that a term u can be α-*converted *to a term u' under a substitution s, written $s \vdash u \longrightarrow u'$, if renaming bound variables in u according to s yields u'. The relation is defined by the following rules.*

$$\frac{}{s \vdash i \longrightarrow i} \qquad \frac{s(x) = y}{s \vdash x \longrightarrow y} \qquad \frac{y \notin (s\backslash x)\,(u) \quad s[x \mapsto y] \vdash u \longrightarrow u'}{s \vdash \lambda x.\, u \longrightarrow \lambda y.\, u'}$$

$$\frac{s \vdash u_1 \longrightarrow u_1' \quad s \vdash u_2 \longrightarrow u_2'}{s \vdash u_1 + u_2 \longrightarrow u_1' + u_2'} \qquad \frac{s \vdash u_1 \longrightarrow u_1' \quad s \vdash u_2 \longrightarrow u_2'}{s \vdash u_1\, u_2 \longrightarrow u_1'\, u_2'}$$

where $y \notin u$ indicates that the variable y does not occur (free or bound) in the expression u.

Two closed term u_1 and u_2 are α-congruent in the traditional sense [6, p. 26] if they are related by $[\,] \vdash u_1 \longrightarrow u_2$.

Definition 6 *Given a translation context Ξ we define a substitution $\Xi^{\mathcal{S}}$ and an environment $\Xi^{\mathcal{E}}$ such that if $fresh(n) = g$ then*

(1) $(\Xi[x : (\sigma, n)])^{\mathcal{S}} = \Xi^{\mathcal{S}}[x \mapsto g]$

(2) $(\Xi[x : (\sigma, n)])^{\mathcal{E}} = \Xi^{\mathcal{E}}[x \mapsto \lambda z.\, \mathbf{up}(\mathit{VAR}(g))]$

Lemma 10 *For any integer n, term u, expression e, type σ, and translation context Ξ with* $\mathrm{range}(\Xi) = \{(\sigma, i) \mid i < n\}$, *if $n; \Xi \vdash u : \sigma \,/\, e$ then there exists a term u' such that $\Xi^{\mathcal{S}} \vdash u \longrightarrow u'$ and $[\![e]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}(\lceil u' \rceil)$.*

Proof. By induction on the derivation of $n; \Xi \vdash u : \sigma \,/\, e$.

Case $n; \Xi \vdash i : \mathsf{Int} \,/\, \mathsf{int}\, i$. Then $[\![\mathsf{int}\, i]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}(\mathit{INT}(i)) = \mathbf{up}(\lceil i \rceil)$ and indeed $\Xi^{\mathcal{S}} \vdash i \longrightarrow i$.

Case $n; \Xi \vdash u_1 + u_2 : \mathsf{Int} \,/\, \mathsf{add}\, e_1\, e_2$ where $n; \Xi \vdash u_1 : \mathsf{Int} \,/\, e_1$ and $n; \Xi \vdash u_2 : \mathsf{Int} \,/\, e_2$. Then by two applications of the induction hypothesis there exist u_1' and u_2' such that $\Xi^{\mathcal{S}} \vdash u_1 \longrightarrow u_1'$, $\Xi^{\mathcal{S}} \vdash u_2 \longrightarrow u_2'$, $[\![e_1]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}(\lceil u_1' \rceil)$, and $[\![e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}(\lceil u_2' \rceil)$. We therefore have

$$\begin{array}{rcl} [\![\mathsf{add}\, e_1\, e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) & = & \mathit{ADD}_\perp\big([\![e_1]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)), [\![e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n))\big)_\perp \\ & = & \mathit{ADD}_\perp(\mathbf{up}(\lceil u_1' \rceil), \mathbf{up}(\lceil u_2' \rceil))_\perp \\ & = & \mathbf{up}(\mathit{ADD}(\lceil u_1' \rceil, \lceil u_2' \rceil)) \\ & = & \mathbf{up}(\lceil u_1' + u_2' \rceil) \end{array}$$

and indeed $\Xi^{\mathcal{S}} \vdash u_1 + u_2 \longrightarrow u_1' + u_2'$.

Case $n; \Xi \vdash x : \sigma \,/\, x$ where $\Xi(x) = (\sigma, m)$. Then with $g = \mathit{fresh}(m)$,

$$[\![x]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \Xi^{\mathcal{E}}(x)\, (\mathbf{up}(n)) = \mathbf{up}(\mathit{VAR}(g)) = \mathbf{up}(\ulcorner g \urcorner)$$

and indeed $\Xi^{\mathcal{S}} \vdash x \longrightarrow g$.

Case $n; \Xi \vdash \lambda x.\, u : \sigma_1 \to \sigma_2 \,/\, \mathsf{lam}_{\tau_1,\tau_2}\, (\lambda x.\, e)$ where $n+1; \Xi[x : (\sigma_1, n)] \vdash u : \sigma_2 \,/\, e$. Then with $g = \mathit{fresh}(n)$, we have from the induction hypothesis that there exists u' such that $\Xi[x : (\sigma_1, n)]^{\mathcal{S}} \vdash u \longrightarrow u'$. Definition 6(1) then gives $\Xi^{\mathcal{S}}[x \mapsto g] \vdash u \longrightarrow u'$. Together with Definition 6(2) the induction hypothesis also gives

$$\begin{array}{rcl} [\![\lambda x.\, e]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) & = & \mathit{LAM}_{\bot}\big(\mathbf{up}(g), [\![e]\!]_{\mathcal{I}_0}\, \Xi[x : (\sigma_1, n)]^{\mathcal{E}}\, (\mathbf{up}(n+1))\big)_{\bot} \\ & = & \mathit{LAM}_{\bot}\big(\mathbf{up}(g), \mathbf{up}(\ulcorner u' \urcorner)\big)_{\bot} \\ & = & \mathbf{up}(\mathit{LAM}(g, \ulcorner u' \urcorner)) \\ & = & \mathbf{up}(\ulcorner \lambda g.\, u' \urcorner) \end{array}$$

Since $\mathrm{range}(\Xi) \subseteq \{(\sigma, i) \mid i < n\}$ we have that for all z, $\Xi^{\mathcal{S}}(z) \neq g$. Hence indeed $\Xi^{\mathcal{S}} \vdash \lambda x.\, u \longrightarrow \lambda g.\, u'$.

Case $n; \Xi \vdash u_1\, u_2 : \sigma_2 \,/\, \mathsf{app}_{\tau_1,\tau_2}\, e_1\, e_2$ where $n; \Xi \vdash u_1 : \sigma_1 \,/\, e_1$ and $n; \Xi \vdash u_2 : \sigma_1 \to \sigma_2 \,/\, e_2$. (Follows the same structure as the case for addition.) Then by induction hypothesis there exist u_1' and u_2' such that $\Xi^{\mathcal{S}} \vdash u_1 \longrightarrow u_1'$, $\Xi^{\mathcal{S}} \vdash u_2 \longrightarrow u_2'$, $[\![e_1]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}(\ulcorner u_1' \urcorner)$, and $[\![e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) = \mathbf{up}(\ulcorner u_2' \urcorner)$. We therefore have

$$\begin{array}{rcl} [\![\mathsf{app}_{\tau_1,\tau_2}\, e_1\, e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)) & = & \mathit{APP}_{\bot}\big([\![e_1]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n)), [\![e_2]\!]_{\mathcal{I}_0}\, \Xi^{\mathcal{E}}\, (\mathbf{up}(n))\big)_{\bot} \\ & = & \mathit{APP}_{\bot}(\mathbf{up}(\ulcorner u_1' \urcorner), \mathbf{up}(\ulcorner u_2' \urcorner))_{\bot} \\ & = & \mathbf{up}(\mathit{APP}(\ulcorner u_1' \urcorner, \ulcorner u_2' \urcorner)) \\ & = & \mathbf{up}(\ulcorner u_1'\, u_2' \urcorner) \end{array}$$

and indeed $\Xi^{\mathcal{S}} \vdash u_1\, u_2 \longrightarrow u_1'\, u_2'$. □

Theorem 2 (Completeness) *For any closed term $\emptyset \vdash u : \sigma$ there exists an expression $\emptyset \vdash_{\Sigma_0} e :$ $\mathsf{Exp}\,\sigma$ and a closed term u' such that $[\,] \vdash u \longrightarrow u'$ and $[\![e]\!]_{\mathcal{I}_0}\, [\,]\, (\mathbf{up}(0)) = \mathbf{up}(\ulcorner u' \urcorner)$.*

Proof. The type part follows from Lemma 9 and the evaluation part from Lemma 10. □

2.4 Haskell as a meta-language

There are a number of issues that must be addressed to use Haskell as a meta-language for embedded languages. Differences between Haskell and the idealized meta-language presented in the previous section influence the conditions under which Haskell can safely be used as a meta-language.

There are no built-in types $\mathsf{Exp}\,\tau$ or constant symbols $\mathsf{lam}_{\tau_1,\tau_2}$ and $\mathsf{app}_{\tau_1,\tau_2}$ in Haskell. Instead, we must define global variables as in Figure 2.2 and argue that their semantics agree with the semantics of $\mathsf{lam}_{\tau_1,\tau_2}$ and $\mathsf{app}_{\tau_1,\tau_2}$. We have designed the semantics of these constant symbols to match the Haskell implementation. Furthermore, in Haskell, the types of these symbols are restricted outside the module of their implementation. In contrast, in the formal development they are given as constant symbols of the restricted types.

The formal treatment in this work uses strict data type constructors. In contrast, Haskell's constructors are non-strict. For the soundness result, the strict constructors guarantee that we only observe the types of "finite" object-language terms. In Haskell, it is possible to observe the shape of "infinite" terms using the top-level read-eval-print loop. For example, the following well-typed Haskell expression uses recursion to build an infinite object-language term.

```
let f = lam (λx → app f x) in f
```

Its meaning is the representation of the limit of the chain of incomplete terms

$$\begin{array}{ll} & \bot \\ \sqsubseteq & \texttt{LAM "x1" (APP } \bot \texttt{ (VAR "x1"))} \\ \sqsubseteq & \texttt{LAM "x1" (APP (LAM "x2" (APP } \bot \texttt{ (VAR "x2"))) (VAR "x1"))} \\ \sqsubseteq & \cdots \end{array}$$

where $\bot$ denotes the bottom element of the CPO of (non-strict) data types. In a formal account for the soundness property using non-strict data types it may be desirable not to give types to such infinitely expanding terms. Instead, the soundness property might state that if a "finite" object-language term is denoted by an expression of type $\mathsf{Exp}\,\tau$ then it has type τ.

Haskell's function space is lifted, e.g. $[\![\tau_1 \rightarrow \tau_2]\!]_{\mathcal{I}} = ([\![\tau_1]\!]_{\mathcal{I}} \rightarrow_{\mathrm{c}} [\![\tau_2]\!]_{\mathcal{I}})_{\bot}$, allowing observing termination of expressions of higher types using the top-level read-eval-print loop. In contrast, in the idealized meta-language an expression of higher type is always applied so its termination behavior is never observed independently. Changing the semantics to account for Haskell-like lifted function spaces does not appear to introduce any difficulties in the soundness proof.

Finally, the approach to embedding languages into higher-order host languages that we have presented inherits the known problems of higher-order abstract syntax [76]. Most importantly, a higher-order abstract syntax does not admit a notion of case analysis. In order to make any use of constructed object-language terms, the embedding we have presented directly represents terms using a first-order data type which can be printed as text. However, the standard function spaces of functional languages are generally too large for adequate higher-order abstract syntax. For example, the implementation in Figure 2.2 actually allows representations of "exotic terms", such as the following expression of type `Exp (Int → Int)`.

```
lam (λx → if show x = "a" then int 1 else int 2)
```

Although closed, this term do not correspond to any object-language term. Depending on the context, it may behave as either `λa → 1` or `λx → 2`. Research on higher-order abstract syntax alleviates these problems by restricting the function spaces of the meta-language using, e.g., modal logics [49]. In Haskell-like languages, however, such a requirement cannot be enforced by the type system. In the presence of `show`-like functions, there are no ways around

```
-- Finite products
pair :: Exp a → Exp b → Exp (a, b)
pfst :: Exp (a, b) → Exp a
psnd :: Exp (a, b) → Exp b

-- Lists
nil  :: Exp [a]
cons :: Exp a → Exp [a] → Exp [a]
hd   :: Exp [a] → Exp a
tl   :: Exp [a] → Exp [a]

-- Booleans
tt   :: Exp Bool
ff   :: Exp Bool
ift  :: Exp Bool → Exp a → Exp a → Exp a

iszero :: Exp Int → Exp Bool
isnull :: Exp [a] → Exp Bool
```

Figure 2.3: Extended object language.

the problems of higher-order abstract syntax other than to informally require the programmer to only observe the behavior of closed object-language terms. This requirement appears to match the typical use of embedded languages in existing applications.

Below, we briefly discuss the embedded type discipline in the context of extended object languages and in the context of extended meta-languages.

2.4.1 Extended object language

As already mentioned, other domain-specific types and operations are easily added to the object language. Figure 2.3 presents the types of operations on finite products, lists, and booleans. (The actual definition of these primitives are similar to those presented in Figure 2.2 and are left out for conciseness.) It is straightforward to extend the proof of soundness and completeness to also handle the extensions outlined here.

It is also possible to extend the object language with types that do not exist in the meta-language. For example,

```
type Ref a = ()

ref :: Exp a → Exp (Ref a)
get :: Exp (Ref a) → Exp a
set :: Exp (Ref a) → Exp a → Exp ()
```

declares the types of pointer-manipulating object-language functions in Haskell. (Appropriate values can be defined as in Figure 2.2.) Values of type `Ref` will never be constructed and therefore we can choose a minimal representation, here as the unit type `()`. A similar technique is used to integrate COM objects into Haskell [63].

The domain-specific operations on base types considered here (i.e., `int`, `add`, and the operations of the extended language in Figure 2.3) are *meta functions*: they do not represent first-class functions in the object language. For example, there is no object-language equivalent of a first-class zero predicate. Indeed, Haskell rejects the expression `cons iszero nil` (an attempt to represent a singleton list of a zero predicate). A first-class zero predicate can be embedded by

```
data Raw = ... | ISZERO'

iszero' :: Exp (Int → Bool)
iszero' =  E (λns → ISZERO')
```

and used as in `cons iszero' nil` of type `Exp [Int → Bool]`. Since the object language is higher order we can also apply object-language β and η conversions at the meta level. The two function compositions

```
lam . app :: Exp (a → b) → Exp (a → b)
app . lam :: (Exp a → Exp b) → Exp a → Exp b
```

are the extensional identity function on object-language functions, and the extensional identity function on meta-language functions. In fact, in the sense of binding-times and two-level coercions, the first composition coerces a dynamic function to static and back to dynamic again whereas the second composition coerces a static function to dynamic and back to static again, using two-level η-expansions [40].

Given, for example, a Haskell-like object language where the meta-language expression `iszero E` represents `(e ≡ 0)` (where `E` represents `e`) and the meta-language expression `iszero'` represents `(≡ 0)`. (In Haskell, `≡` denotes structural equality.) In the meta-language we then have the (extensional) identities

```
iszero  = app iszero'     -- of type Exp Int → Exp Bool
iszero' = lam iszero      -- of type Exp (Int → Bool)
```

since (for the first equality) for any meta-language expression `E` of type `Exp Int` representing `e`, `app iszero' E` represents `(≡ 0) e` and (for the second equality) `lam iszero'` represents `λx → x ≡ 0` and indeed `e ≡ 0` and `(≡ 0) e` are semantically identical, as are `(≡ 0)` and `λx → x ≡ 0`.

As opposed to Haskell, the object-language does not provide polymorphism and recursion. Polymorphic or recursive meta-language expressions that are accepted by Haskell's type system are unfolded in the object-language. For example, it is possible to define polymorphic representations of object-language values in the meta-language as illustrated above for finite products and lists. Even though we cannot represent object term such as

```
let f = λx → x in (f 0, f True)
```

we can inline let-bound polymorphic values in the object term. For example, the following expression has type `Exp (Int, Bool)`.

```
let f = lam(λx → x) in pair (app f (int 0)) (app f tt)
```

It evaluates to the object term `((λa → a) 0, (λa → a) True)`.

2.4.2 Extended meta-language

It is possible to extend the meta-language with other base types, finite products, and lists à la Haskell, and also to extend the proofs of soundness and completeness.

It is also possible to change the evaluation strategy of the meta-language to call-by-value. Adding other effects than non-termination, however, might give an unsound embedding. This is the case for ML [102]. Using assignments, for example, some well-typed ML expressions do not represent well-typed object language terms. In fact, as the following example shows, some expressions do not even yield representations of closed terms.

```
let val v = ref (int 0)
in  lam (fn x ⇒ (v := x; x));
    v
end
```

This expression returns whatever symbolic variable was generated at the time of constructing the object-language lambda. This problem is due to the use of higher-order abstract syntax, not the typed embedding.

2.5 Related work and conclusions

Phantom types make it possible to embed not only the (abstract) syntax but also the type system of a monomorphic object language into a statically typed meta-language such as Haskell. They have been used to design embedded languages [63, 85, 93] and to prove properties of programs [42, 43]. In this chapter, we have formally proved that phantom types yield sound and complete embeddings into an idealized meta-language and we have discussed the shortcomings of Haskell as such a meta-language.

Our soundness proof uses a Kripke logical relation. The development is akin to Filinski's proof that type-directed partial evaluation implements a normalization function [61]. A corollary of Filinski's work is that type-directed partial evaluation is type preserving, a result we have also established using the typed embedded language described in this chapter [42, 43].

Completeness could also be stated by giving a semantics of the object language and then show that a term and its encoding are semantically equivalent. Our proof is stronger in that it shows that a term and its encoding are syntactically equal modulo renaming of bound variables. It also avoids to deal with the semantics of the object language.

Phantom types provide a simple form of dependent types for *constructing* representations of simply-typed terms in Haskell. The embedding is limited, however, in that it does not allow a type-preserving *deconstruction* of simply-typed terms. This is partly due to the lack of higher-order matching in Haskell. A dependently typed language, such as Martin-Löf's type theory [108], can directly express both type-safe construction and deconstruction of object terms. Previously, Yang has shown how to encode another class of dependently typed expressions in ML [145].

In the mid-1980's, Wand has defined the meaning of an object language (with primitive I/O and non-local jumps) in terms of the λ-calculus, viewing the latter as a semantic meta-language [142]. His translation is sound and complete in the sense that exactly the well-typed object-language terms have a well-typed meta-language counterpart. Thus, it can be seen as providing both a static and a dynamic semantics for the object language. (In fact, since the translated terms are in continuation-passing style, Wand also shows that the CPS transformation preserves well-typedness [101].) Wand's translation does not yield first-order data as result such as the embedding we consider here. Instead, it maps higher-order functions to higher-order functions. Furthermore, Wand's object-language type system is not presented in terms of his meta-language. In contrast, the embedded type discipline we consider here expresses the type system of the object language in terms of the type system of the meta-language (using phantom types).

Davies's $\lambda^{\bigcirc}$-calculus is an extension of the simply-typed λ-calculus for expressing staged evaluation [45]. A term of type $\bigcirc\tau$ evaluates in one stage to a value representing a term of type τ. Terms of type $\bigcirc\tau$ are thus first-class representations of terms, much as the expressions of type $\mathsf{Exp}\,\tau$ in our work are representations of object-language terms. There is a syntactic difference, however. In $\lambda^{\bigcirc}$ there is one uniform construct for building terms at the next stage. The meta-language in our work provides several constructors, one for each syntactic category of the object language. (Analogy: $\lambda^{\bigcirc}$ provides a type system for Lisp-like quasiquotation [8] whereas phantom types provide a type system for ML-like data types.) Another difference is that $\lambda^{\bigcirc}$-terms build other $\lambda^{\bigcirc}$-terms for a later stage. In contrast, the object-language terms in our work are always simply typed λ-terms. Consequently, we cannot embed object languages with an arbitrary number of stages. There is some hope, though, that type encodings as described by Yang [145] could achieve such a nested embedding.

Chapter 3

Type-directed partial evaluation

In this chapter we present three implementations of type-directed partial evaluation in statically typed languages.

3.1 Deriving a statically typed type-directed partial evaluator

Type-directed partial evaluation was originally implemented in Scheme, a dynamically typed language. It has also been implemented in ML, a statically Hindley-Milner typed language. This section shows how the latter implementation can be derived from the former through a functional representation of inductively defined types.

Note. This section is based on work presented at PEPM 1999 [119].

Thanks are due to Olivier Danvy and the anonymous referees for commenting earlier versions of this paper. Also many thanks to Kristoffer Rose for shepherding this paper.

3.1.1 Introduction

Type-directed partial evaluation is an approach to specializing a term written in a higher-order language. Such a higher-order term is specialized by normalizing it with respect to its type. Normalization is done by η-expanding the term in a two-level lambda-calculus and statically β-reducing the expanded term.

The two stages — η-expansion and β-reduction — share an intermediate result: a two-level term. We consider two versions of the type-directed partial evaluation algorithm:

(1) If the two-level term is represented as a value of an inductively defined data type then the algorithm can be implemented in any (Turing complete) language. In this case both η-expansion and β-reduction are implemented in this language.

(2) If the algorithm is implemented in a higher-order language an alternative is to represent the two-level term as a mixture of object-language terms (dynamic parts) and implementation-language terms (static parts). In this case η-expansion is implemented in the implementation language. It generates a two-level term whose implementation-language parts are β-reduced by the native β-reducer of the implementation language.

The problem

Both of the versions of the algorithm are directed by the type of the term to be normalized. They are applied to a term and a representation of the type of the term. In (2), if the type is given as an element of an inductively defined type then it is impossible to statically type-check the algorithm, and indeed this also has been implemented in a dynamically typed language, Scheme [11, 30].

This section shows how to derive a statically typed analogue of (2) from (1). It is obtained by Church-encoding types as higher-order values. Andrzej Filinski was the first to use a Church-encoding of types for type-directed partial evaluation.

Related work

The first statically typed version of type-directed partial evaluation is due to Andrzej Filinski in the spring of 1995. This unpublished work showed that type-directed partial evaluation could be implemented at all using a Hindley-Milner type system. The second one is due to Zhe Yang in the spring of 1996 [145]. Yang provides general methods for encoding type-indexed values in a Hindley-Milner typed language and applies them to type-directed partial evaluation. The present work was carried out in the fall of 1997 and, according to Olivier Danvy, it is the third independent implementation of type-directed partial evaluation in a Hindley-Milner typed language [118].

Kristoffer Rose implemented type-directed partial evaluation in Haskell in the spring of 1998 using type classes [121]. Haskell's type classes permit overloaded functions, i.e., functions that have several definitions, one for each type of argument. Type-directed partial evaluation fits exactly into this pattern.

In her M. Sc. thesis, Belmina Dzafic implements type-directed partial evaluation in Elf, a statically typed, constraint logical language (summer 1998) [52]. Furthermore, she proves the equivalence of the dynamically typed and the statically typed versions of type-directed partial evaluation. Earlier on, Catarina Coquand stated and proved the correctness of the type-directed partial evaluation algorithm using the proof editor Alf [26].

The derivation

Type-directed partial evaluation is given by $\downarrow$ (reify) and $\uparrow$ (reflect) in Figure 3.1. Based on this algorithm we derive a statically typed, type-directed partial evaluator analogous to (2) in the following steps:

$$
\begin{array}{llcl}
\text{(types)} & t & ::= & \mathrm{b} \mid t_1 \to t_2 \\
\\
\text{(reify)} & \downarrow^{\mathrm{b}} v & = & v \\
 & \downarrow^{t_1 \to t_2} v & = & \underline{\lambda} x.\ \downarrow^{t_2} (v @ (\uparrow_{t_1}\ x)) \\
 & & & \text{where } x \text{ is fresh} \\
\\
\text{(reflect)} & \uparrow_{\mathrm{b}} e & = & e \\
 & \uparrow_{t_1 \to t_2} e & = & \lambda x.\ \uparrow_{t_2} (e \underline{@} (\downarrow^{t_1}\ x)) \\
\\
 & \mathrm{tdpe}\, t\, v & = & \downarrow^{t} v
\end{array}
$$

Figure 3.1: Type-directed partial evaluation

- Starting from (1), we represent types using a datatype Type and representing both static and dynamic terms as a datatype `Term` we translate the $\downarrow$ and $\uparrow$ of Figure 3.1 into into `reify` and `reflect`. This solution is interpretive in that it uses an explicit static β-reducer.

- We change the representation of static terms from elements of the datatype `Term` into higher-order values and replace the explicit β reducer by the native β reducer of the implementation language. This solution is not interpretive but it is also not statically typeable.

- We observe an invariant property: `reify` is applied only to types occurring covariantly in the source type, and `reflect` only to types appearing contravariantly in the source type. We encode this distinction in the datatype Type. This solution is not statically typeable but by changing the representation of types from the datatype Type to higher-order values we obtain a statically typed solution which is still not interpretive: it uses the native (implicit) β-reducer to statically reduce the two-level term.

Using the implicit static β-reducer or an explicit one is independent of the representation of types. However, using the implicit static β-reducer together with a datatype representation of types does not yield a solution that is statically typeable in a Hindley-Milner typing system.

Overview

We consider four successive implementations of type-directed partial evaluation in this section.

In Section 3.1.2, object-language types are represented by an inductively defined type. Terms are represented either by values of an inductively defined type or by a mixture of higher-order values and terms.

In Section 3.1.3, types are represented by higher-order values. The main result is an implementation of terms as higher-order values. For completeness we also take a step backwards and represent terms by values of an inductively defined data type.

In Section 3.1.4 we briefly consider the efficencies of the programs appearing in this section. In Section 3.1.5 we present two extensions over the statically typed algorithm. Section 3.1.6 concludes.

All programs in this section are written in a functional notation à la Haskell [59]. We defer the problem of generating fresh identifiers [118].

3.1.2 Inductively defined representation of types

First we consider two implementations of type-directed partial evaluation that use a representation of types as elements of the following inductively defined type.

```
data Type =  Base | Func Type Type
```

The two implementations differ in the way the static parts of two-level terms are represented.

Inductively defined representation of terms

In this approach, of which only an outline is given here, both the static and the dynamic parts of terms are represented by an inductively defined type.

```
type Id   =  [Char]
data Term =  Num Int | Str String
          |  SVar Id | SLam Id Term | SApp Term Term
          |  DVar Id | DLam Id Term | DApp Term Term
```

The static syntax constructors are prefixed with an "S" and the dynamic syntax constructors are prefixed with a "D". Constants are both static and dynamic.

The algorithm proceeds in two stages: First, given a completely static term and its type, a fully η-expanded two-level term is constructed using `reify` and `reflect` below. Second, the static parts of the term are β-reduced (the β-reducer is omitted here).

```
reify, reflect          :: Type → Term → Term

reify   Base         v =  v
reify   (Func t1 t2) v =
    DLam x (reify t2 (SApp v (reflect t1 (DVar x))))
    where x = fresh "x"

reflect Base         v =  v
reflect (Func t1 t2) v =
    SLam x (reflect t2 (DApp v (reify t1 (SVar x))))
    where x = fresh "x"

etaExpand t v      =  reify t v
```

```
staticBetaReduce v =  ...

tdpe              ::  Type → Term → Term
tdpe t v           =  staticBetaReduce (etaExpand t v)
```

This program is *statically typeable* in a Hindley-Milner typing system and can be implemented in any language with inductively defined types, regardless of the typing discipline. The implementation-language type of the output of `reify` and `reflect` does not depend on the representation of the object-language type (a value of type `Type`).

The explicit use of a β-reducer is undesirable since it embeds the lambda calculus into the implementing language via an interpreter. This is inefficient and the question arises whether we could not implement the algorithm in a higher-order language using the underlying β-reduction mechanism of this implementation language.

Higher-order representation of terms

The following approach uses the β-reduction mechanism of the higher-order implementation language.

Types are represented by the same type as above. Two-level terms are represented by a mixture of implementation-language terms (values) and object-language terms.

```
reify   Base          v =  v
reify   (Func t1 t2)  v =
    DLam x (reify t2 (v (reflect t1 (DVar x))))
    where x = fresh "x"
reflect Base          v =  v
reflect (Func t1 t2)  v =
    λx → reflect t2 (DApp v (reify t1 x))

etaExpand  t v  = reify t v

tdpe t v        = etaExpand t v
```

This program is *not statically typeable* in a Hindley-Milner typing system: The implementation-language type of the output of `reify` and `reflect` depends on the representation of the object-language type (a value of type `Type`).

Short of dependent types, at compile time, there is not enough information available so that the type-checker can accept the program. The above program corresponds to the original implementation of type-directed partial evaluation in Scheme [30].

3.1.3 Higher-order representation of types

Observe that `reify` is always applied to types that occur *covariantly* in the source type (a value of type `Type`) and that `reflect` is always applied to types that occur *contravariantly* in the source type. We make this explicit by distinguishing between covariant occurrences (postfixed by "`P`" for positive) and contravariant occurrences (postfixed by "`N`" for negative):

```
data TypeP =  BaseP | FuncP TypeN TypeP
data TypeN =  BaseN | FuncN TypeP TypeN
type Type  =  TypeP
```

Higher-order representation of terms

Now `reify` and `reflect` are

```
reify BaseP          v =  v
reify (FuncP t1 t2)  v =
    DLam x (reify t2 (v (reflect t1 (DVar x))))
    where x = fresh "x"
reflect BaseN        v =  v
reflect (FuncN t1 t2) v =
    λx → reflect t2 (DApp v (reify t1 x))
```

This program is *not statically typeable* in a Hindley-Milner typing system. Again, the implementation-language type of the output of `reify` and `reflect` depends on the representation of the object-language type (values of types `TypeP` and `TypeN`).

In order to obtain a statically typeable program we apply the following change: Instead of representing a positively occuring type t as a `TypeP` we represent it as a value equal to (`reify` t) and instead of representing a negatively occuring type t as a `TypeN` we represent it as a value equal to (`reflect` t).

```
baseP, baseN      :: a → a
funcP ::
    (Term → a) → (b → Term) → (a → b) → Term
funcN ::
    (a → Term) → (Term → b) → Term → (a → b)

baseP       v    =  v
funcP t1 t2 v    =  DLam x (t2 (v (t1 (DVar x))))
                    where x = fresh "x"
baseN       v    =  v
funcN t1 t2 v    =  λx → t2 (DApp v (t1 x))

etaExpand t v    =  t v

tdpe             :: (a → b) → a → b
tdpe t v         =  etaExpand t v
```

This program is *statically typeable* in a Hindley-Milner typing system. The implementation-language type of `tdpe` does not depend on the representation of the object-language type, but on the type of the object-language type.

Thus, even without dependent types, the type-checker has enough information to instantiate the polymorphic type of `tdpe`. This is the main result of this section.

For example, the type b (a base type) is represented by `Base` of type `Type` in the first part of Section 3.1.2. In the current section it is represented by `baseP` (i.e., the identity function) of type a → a. Filinski and Yang's representation of this type is the pair of functions ($\downarrow^{b}$, $\uparrow_{b}$), i.e., a pair of identity functions. (A discussion of this representation follows below.)

The type $b \rightarrow b$ is represented by `Func Base Base` of type `Type` in the first part of Section 3.1.2. In the current section it is represented by `funcP baseN baseP` of type

```
(Term → Term) → Term
```

Filinski and Yang's representation of this type is the pair of functions ($\downarrow^{b\rightarrow b}$, $\uparrow_{b\rightarrow b}$).

As an example, let's specialize some terms that contain static redeces using the result of this section:

```
> :t tdpe baseP
tdpe baseP :: a → a
> :type tdpe (funcP baseN baseP)
tdpe (funcP baseN baseP) :: (Term → Term) → Term
> tdpe baseP ((λx → x) (Num 42))
Num 42
> tdpe (funcP baseN baseP) ((λx → λy → x) (Num 42))
DLam "x0" (Num 42)
>
```

Inductively defined representation of terms

For the record, let us repeat the solution above using a "traditional" inductively defined representation of terms. To this end we use again the type of terms.

```
type Id   =  [Char]
data Term =  Num Int | Str String
          |  SVar Id | SLam Id Term | SApp Term Term
          |  DVar Id | DLam Id Term | DApp Term Term

baseP       v =  v
funcP t1 t2 v =  DLam x (t2 (SApp v (t1 (DVar x))))
                 where x = fresh "x"
baseN       v =  v
funcN t1 t2 v =  SApp x (t2 (DApp v (t1 (SVar x))))
                 where x = fresh "x"

etaExpand t v        =  t v
staticBetaReduce v   =  ...

tdpe     :: Type → Term → Term
tdpe t v =  staticBetaReduce (etaExpand t v)
```

This program is *statically typeable*. It also requires explicit static β-reduction (which is omitted here).

3.1.4 Pragmatics

We have compared the performance of the three statically typed solution in ML. We used a simple-minded, hand-coded static β-reducer for terms represented as a data type and the native β-reducer of ML for the higher-order representation of terms. The hand-coded reducer uses the same reduction strategy as the native reducer of ML.

Specializing the power function with respect to a static exponent of value 12 is about 9 times faster using the higher-order representation of term than using a data-type representation. Specializing $(S\,K)\,K$ at type $\mathrm{b} \rightarrow \mathrm{b}$ is about 4 times faster using the higher-order version than using the data-type solutions versions. These results confirm Berger, Eberl, and Schwichtenberg's empirical observations [10].

There do not appear to be any perceptible difference between the two solutions that use the hand-coded static reducer.

3.1.5 Extending the statically typed algorithm

Consider again

```
tdpe baseP               :: a → a
tdpe (funcP baseN baseP) :: (Term → Term) → Term
```

This indicates that constants of base type must be coerced to dynamic values in the source programs. For example, to obtain something of type `Term` we must coerce the integer 42 into a `Term` in

```
> tdpe baseP (Num 42)
Num 42
> tdpe (funcP baseN baseP) (λx → (Num 42))
DLam "x0" (Num 42)
>
```

Furthermore, we must explicitly indicate the variance of types (by the postfix “`P`” or “`N`”). Both shortcomings are alleviated below.

The need for coercing base values

At covariant base types the explicit coercion of values of base type can be removed by distinguishing the base types. We introduce a more specific version of `baseP` for each base type.

```
numP    v =  Num v
strP    v =  Str v

tdpe numP               :: Int → Term
tdpe strP               :: String → Term
tdpe (funcP baseN numP) :: (Term → Int) → Term
```

These new functions will reify a static value into its dynamic counterpart. Static values of base type, such as integers and strings, are represented uniquely and these values are used directly when constructing the dynamic term. A similar solution does not work at contravariant base type since dynamic values of base type can be any dynamic term.

```
> tdpe (funcP baseN numP) (λx → 42)
DLam "x0" (Num 42)
> tdpe (funcP baseN strP) (λx → "fortytwo")
DLam "x0" (Str "fortytwo")
>
```

The need for specifying the variance

The other shortcoming of the implementation — the explicit distinction between covariant and contravariant types — can also be alleviated. Instead of representing a type in two parts (i.e., a covariant part and a contravariant part as above) we can merge the two parts into a pair that represents the type, obtaining Filinski and Yang's solution [145].

```
base       :: (a → a, b → b)
func       :: (a → Term, Term → b) →
                (c → Term, Term → d) →
                  ((b → c) → Term, Term → (a → d))

base       = (reify, reflect)
    where reify   v = v
          reflect v = v
func t1 t2 = (reify, reflect)
    where reify   v =
              DLam x (fst t2 (v (snd t1 (DVar x))))
              where x = fresh "x"
          reflect v =
              λx → (snd t2 (DApp v (fst t1 x))))

etaExpand t v  =  (fst t) v

tdpe           :: (a → b,c) → a → b
tdpe t v       =  etaExpand t v
```

Using this implementation we can specialise terms without specifying the covariance and contravariance of the type involved.

```
> tdpe (func (func base base) base) (λf → f (Num 8))
DLam "x0" (DApp (DVar "x0") (Num 8))
>
```

3.1.6 Conclusion and issues

Being directed by the *type* of a term, the type-directed partial evaluation algorithm requires a way of representing types. In dynamically typed languages an inductively defined sum over the different kind of types (base types, product types, function types, etc.) suffices. In statically typed languages with a Hindley-Milner typing system this does not work: the *type* of the algorithm depends on the *value* of the representation of the type.

The solution is to represent types as higher-order polymorphic functions. This works since the *type* of the algorithm thus depends on the implementation-language *type* of the representation of the object-language type.

Our work suggests to view the higher-order encoding as a functional representation of types, specialised to the purpose of being deconstructed by reify and reflect.

3.2 A simple take on typed abstract syntax in Haskell-like languages

We present a simple way to program typed abstract syntax in a language following a Hindley-Milner typing discipline, such as Haskell and ML, and we apply it to automate two proofs about normalization functions as embodied in type-directed partial evaluation for the simply typed lambda calculus: normalization functions (1) preserve types and (2) yield long beta-eta normal forms.

Note. This section is based on joint work with Olivier Danvy presented at FLOPS 2001 [42] .

Part of it was carried out while visiting Jason Hickey at Caltech, in the summer and fall of 2000. We are grateful to the anonymous reviewers and to Julia Lawall for perceptive comments.

3.2.1 Introduction

Programs (implemented in a *meta language*) that manipulate programs (implemented in an *object language*) need a representation of the manipulated programs. Examples of such programs include interpreters, compilers, partial evaluators, and logical frameworks.

When the meta language is a functional language with a Hindley-Milner type system, such as Haskell [59] or ML [102], a data type is usually chosen to represent object programs. In functional languages, data types are instrumental in representing sum types and inductive types, both of which are needed to represent even the simplest programs such as arithmetic expressions.

However, the *object-language types* of object-language terms represented by data types cannot be inferred from the representation if the meta language does not provide dependent types. Hence, regardless of any typing discipline in the object language, when the meta language follows a Hindley-Milner type discipline, it cannot prevent the construction of object-language terms that are untyped, and correspondingly, it cannot report the types of object-language terms that are well-typed. This typeless situation is familiar to anyone who has represented λ-terms using a data type in an Haskell-like language.

In this section we consider a simple way of representing monomorphically typed λ-terms in an Haskell-like language. We describe a typeful representation of terms that prevents one from constructing untyped object-language terms in the meta language and that makes the type system of the meta language report the types of well-typed object-language terms.

We apply this typeful representation to *type-directed partial evaluation* [34, 61], using Haskell [119]. In Haskell, the object language of type-directed partial evaluation is a subset of the meta language, namely the monomorphically typed λ-calculus. Type-directed partial evaluation is an implementation of *normalization functions*. As such, it maps a meta-language value that is simply typed into a (textual) representation of its *long beta-eta normal form*.

All previous implementations of type-directed partial evaluation in Haskell-like languages have the type $t \to \mathsf{Term}$, for some t and where Term denotes the typeless representation of object programs. This type does not express that the output of type-directed partial evaluation is a representation of an object of the same type as the input. In contrast, our implementation has the more expressive type $t \to \mathsf{Exp}(t)$, where Exp denotes our typeful representation of object programs. This type proves that type-directed partial evaluation preserves types. Furthermore, using the same technique, we also prove that the output of type-directed partial evaluation is indeed in long beta-eta normal form.

The rest of this section is organized as follows. In Section 3.2.2 we review a traditional, typeless data-type representation of λ-terms in Haskell. In Section 3.2.3, we review higher-order abstract syntax, which is a stepping stone towards our typeful representation. Section 3.2.4 presents our main result, namely an extension of higher-order abstract syntax that only allows well-typed object-language terms to be constructed. In Section 3.2.5, we review type-directed partial evaluation, which is our chosen domain of application. Section 3.2.6 presents our first application, namely an implementation of type-directed partial evaluation preserving types. Section 3.2.7 presents our second application, namely another implementation of type-directed partial evaluation demonstrating that it produces long beta-eta normal forms. Section 3.2.8 concludes.

3.2.2 Typeless first-order abstract syntax

We consider the simply typed λ-calculus with integer constants, variables, applications, and function abstractions:

$$\begin{array}{lccl} \text{(Types)} & t & ::= & \alpha \mid \mathsf{int} \mid t_1 \to t_2 \\ \text{(Terms)} & e & ::= & i \mid x \mid e_0\, e_1 \mid \lambda x.e \end{array}$$

Other base types (booleans, reals, etc.) and other type constructors (products, sums, lists, etc.) are easy to add. So our object language is the λ-calculus.

Our meta language is Haskell. We use the following data type to represent λ-terms. Its constructors are: integers (`INT`), variables (`VAR`), applications (`APP`), and functional abstractions (`LAM`).

```
data Term = INT Int
          | VAR String
          | APP Term Term
```

```
module TypelessExp(int, app, lam, Exp) where

  data Term = INT Int | VAR String | APP Term Term | LAM String Term

  type Exp = Int → Term

  int i     j = INT i
  app e0 e1 j = APP (e0 j) (e1 j)
  lam f     j = LAM v (f (λ_  → VAR v) (j + 1))
                where v = "x" ++ show j
```

Figure 3.2: Typeless higher-order abstract syntax in Haskell

```
                  | LAM String Term
```

Object-language terms are constructed in Haskell using the translation below. Note that the type of $\lceil e \rceil_0$ is Term regardless of the type of e in the λ-calculus.

$$\begin{array}{rcl}
\lceil i \rceil_0 & = & \texttt{INT}\ i \\
\lceil x \rceil_0 & = & \texttt{VAR "}x\texttt{"} \\
\lceil e_0\, e_1 \rceil_0 & = & \texttt{APP}\ \lceil e_0 \rceil_0\ \lceil e_1 \rceil_0 \\
\lceil \lambda x.e \rceil_0 & = & \texttt{LAM "}x\texttt{"}\ \lceil e \rceil_0
\end{array}$$

The constructors of the data type are typed in Haskell: The term `INT 9` is valid whereas `INT "a"` is not. However, Haskell knows nothing of the λ-terms we wish to represent. In other words, the translation $\lceil\cdot\rceil_0$ is not surjective: Some well-typed *encodings* of object-language terms do not correspond to any legal object-language term. For example, the term `APP (INT 1) (LAM "x" (VAR "x"))` has type Term in Haskell, even though it represents the term $1(\lambda x.x)$ which has no type in the λ-calculus.

The fact that we can represent untyped λ-terms is not a shortcoming of the meta language. One might want to represent programs of an untyped object language like Scheme [89] or even structures for which no notion of type exists.

3.2.3 Typeless higher-order abstract syntax

To the data type Term we add an interface using *higher-order abstract syntax* [110]. In higher-order abstract syntax, object-language variables and bindings are represented by meta-language variables and bindings. The interface to the data type Term is shown in Figure 3.2.

The interface consists of syntax constructors for integers, applications, and abstractions. There is no constructor for variables. Instead, fresh variable names are generated and passed to the higher-order representation of abstractions. A λ-expression is represented by a function accepting the next available fresh-variable name, using de Bruijn levels.

Object-language terms are constructed in the meta language using the following translation. Note again that the type of $\lceil e \rceil_1$ is Term regardless of the type of e in the λ-calculus.

$$\begin{array}{rcl} \lceil i \rceil_1 & = & \texttt{int}\ i \\ \lceil x \rceil_1 & = & x \\ \lceil e_0\, e_1 \rceil_1 & = & \texttt{app}\ \lceil e_0 \rceil_1\ \lceil e_1 \rceil_1 \\ \lceil \lambda x.e \rceil_1 & = & \texttt{lam (\textbackslash}x\ \texttt{->}\ \lceil e \rceil_1\texttt{)} \end{array}$$

This translation is also not surjective in the sense outlined in Section 3.2.2. Indeed, the types of the three higher-order constructors in Haskell still allow untypable λ-terms to be constructed. These three constructors are typed as follows.

$$\begin{array}{lcl} \texttt{int} & :: & \mathsf{Int} \to \mathsf{Exp} \\ \texttt{app} & :: & \mathsf{Exp} \to (\mathsf{Exp} \to \mathsf{Exp}) \\ \texttt{lam} & :: & (\mathsf{Exp} \to \mathsf{Exp}) \to \mathsf{Exp} \end{array}$$

Therefore, the term `app (int 1) (lam (\x -> x))` still has a type in Haskell, namely Exp.

3.2.4 Typeful higher-order abstract syntax

Let us restrict the three higher-order constructors above to only yield well-typed terms. To this end, we make the following observations about constructing well-typed terms.

- The constructor `int` produces a term of object-language type Int.
- The first argument to app is a term of object-language type $\alpha \to \beta$, the second argument is a term of object-language type α, and app produces a term of object-language type β.
- The argument to `lam` must be a function mapping a term of object-language type α into a term of object-language type β, and `lam` produces a term of object-language type $\alpha \to \beta$.

These observations suggest that the (polymorphic) types of the three constructors actually could reflect the object-language types. We thus parameterize the type Exp with the object-language type and we restrict the types of the constructors according to these observations. In Haskell we implement the new type constructor as a data type, not just as an alias for Int $\to$ Term as in Figure 3.2. In this way the internal representation is hidden. The result is shown in Figure 3.3. The three constructors are typed as follows.

$$\begin{array}{lcl} \texttt{int} & :: & \mathsf{Int} \to \mathsf{Exp}(\mathsf{Int}) \\ \texttt{app} & :: & \mathsf{Exp}(\alpha \to \beta) \to (\mathsf{Exp}(\alpha) \to \mathsf{Exp}(\beta)) \\ \texttt{lam} & :: & (\mathsf{Exp}(\alpha) \to \mathsf{Exp}(\beta)) \to \mathsf{Exp}(\alpha \to \beta) \end{array}$$

The translation from object-language terms to meta-language terms is the same as the one for the typeless higher-order abstract syntax. However, unlike for the typeless version, if e is an (object-language) term of type t then the (meta-language) type of $\lceil e \rceil_1$ is $\mathsf{Exp}(t)$.

```
module TypefulExp (int, app, lam, Exp) where

  data Term = INT Int | VAR String | APP Term Term | LAM String Term

  data Exp t = EXP (Int → Term)

  int :: Int → Exp Int
  app :: Exp (a → b) → Exp a → Exp b
  lam :: (Exp a → Exp b) → Exp (a → b)

  int i                 = EXP (λx → INT i)
  app (EXP e0) (EXP e1) = EXP (λx → APP (e0 x) (e1 x))
  lam f                 = EXP (λx → let v     = "x" ++ show x
                                        EXP b = f (EXP (λ_ → VAR v))
                                    in  LAM v (b (x + 1)))
```

Figure 3.3: Typeful higher-order abstract syntax in Haskell

As an example, consider the λ-term $\lambda f.f(1)$ of type $(\mathsf{Int} \to \alpha) \to \alpha$. It is encoded in Haskell by $\lceil \lambda f.f(1) \rceil_1 =$ `lam (\f -> app f (int 1))` of type $\mathsf{Exp}((\mathsf{Int} \to \alpha) \to \alpha)$. Now consider the λ-term $1(\lambda x.x)$ which is not well-typed in the λ-calculus. It is encoded by $\lceil 1(\lambda x.x) \rceil_1 =$ `app 1 (lam (\x -> x))` which is rejected by Haskell.

In the remaining sections, we apply typeful abstract syntax to type-directed partial evaluation.

3.2.5 Type-directed partial evaluation

The goal of *partial evaluation* [83] is to specialize a program p of type $t_1 \to t_2 \to t_3$ to a fixed first argument v of type t_1. The result is a *residual* program p_v that satisfies $p_v(w) = p(v)(w)$ for all w of type t_2, if both expressions terminate. The motivation for partial evaluation is that running $p_v(w)$ is more efficient than running $p(v)(w)$.

In *type-directed partial evaluation* [34, 61, 119, 145], specialization is achieved by normalization. For simply typed λ-terms, the partial application $p(v)$ is residualized into (the text of) a program p_v in *long beta-eta normal form*. That is, the residual program contains no beta-redexes and it is fully eta-expanded with respect to its type.

Type-directed partial evaluation in Haskell

Figure 3.4 displays a typeless implementation of type-directed partial evaluation for the simply typed λ-calculus in Haskell. To normalize a polymorphic value v of type t, one applies the main function `normalize` to the value, v, and a *representation* of the type, $|t|$, defined as follows.

$$\begin{aligned} |\alpha| &= \texttt{rra} \\ |t_1 \to t_2| &= \texttt{rrf}(|t_1|,\ |t_2|) \end{aligned}$$

```
module TypelessTdpe where

  import TypelessExp  -- from Figure 3.2

  data Reify_Reflect(a) =
    RR { reify   :: a → Exp,
         reflect :: Exp → a }

  rra =                -- atomic types
    RR { reify   = λx → x,
         reflect = λx → x }

  rrf (t1, t2) =       -- function types
    RR { reify   = λv → lam (λx → reify t2 (v (reflect t1 x))),
         reflect = λe → λx → reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 3.4: A typeless implementation of type-directed partial evaluation

To analyze the type of the representations of types, we first define the *instance* of a type as follows.

$$\begin{array}{rcl} [\alpha]_0 & = & \mathsf{Exp} \\ [t_1 \to t_2]_0 & = & [t_1]_0 \to [t_2]_0 \end{array}$$

Then, for any type t, the type of $|t|$ is $\mathsf{Reify\_Reflect}([t]_0)$. Haskell infers the following type for the main function.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Exp}$$

This type shows that `normalize` maps an α-typed input value into an Exp-typed output value, i.e., a term. This type, however, does not show that the input (meta-language) value and the output (object-language) term have the same type. In Section 3.2.6, we show that type-directed partial evaluation is type-preserving, and in Section 3.2.7, we show that the output term is in normal form.

Example: Church numerals, typelessly

As an example, we apply type-directed partial evaluation to specialize the addition of two Church numerals with respect to one argument. The Church numeral zero, the successor function, and addition are defined as follows.

```
zero     :: (a → a) → a → a
zero     = λs → λz → z
suc n    = λs → λz → s (n s z)
add m n = λs → λz → m s (n s z)
```

Specializing `add` **with respect to 0:** We specialize the addition function with respect to the Church numeral 0 by normalizing the partial application `add zero`. This expression has the following type.

$$t_{\mathsf{add}} = ((\alpha \to \alpha) \to \beta \to \alpha) \to (\alpha \to \alpha) \to \beta \to \alpha$$

This type is represented in Haskell as follows.

$$|t_{\mathsf{add}}| = \texttt{rrf(rrf(rrf(rra, rra), rrf(rra, rra)),}$$
$$\texttt{rrf(rrf(rra, rra), rrf(rra, rra)))}$$

Thus, evaluating the Haskell expression

$$\texttt{normalize}\ |t_{\mathsf{add}}|\ \texttt{(add zero) 37}$$

(taking 37, for example, as the first de Bruijn level) yields a representation of the following residual term.

$$\lambda x_{37}.\lambda x_{38}.\lambda x_{39}.x_{37}(\lambda x_{40}.x_{38}\ x_{40})x_{39}$$

For readability, let us rename this residual term:

$$\lambda n.\lambda s.\lambda z.n(\lambda n'.s\ n')z$$

This term is the (η-expanded) identity function over Church numerals, reflecting that 0 is neutral for addition.

Haskell infers the following type of the expression `normalize` $|t_{\mathsf{add}}|$.

$$(((t' \to t') \to t' \to t') \to (t' \to t') \to t' \to t') \to t', \quad \text{where } t' = \mathsf{Int} \to \mathsf{Term}$$

This type does not express any relationship between the type of the input term and the type of the residual term.

Specializing `add` **with respect to 5:** We specialize the addition function with respect to the Church numeral 5 by normalizing the partial application `add five`, where `five` is defined as follows.

```
five    = suc (suc (suc (suc (suc zero))))
```

The expression `add five` also has the type t_{add}. Thus, evaluating the Haskell expression

$$\texttt{normalize}\ |t_{\mathsf{add}}|\ \texttt{(add five) 57}$$

(taking 57 this time as the first de Bruijn level) yields a representation of the following residual term.

$$\lambda x_{57}.\lambda x_{58}.\lambda x_{59}.x_{58}(x_{58}(x_{58}(x_{58}(x_{58}(x_{57}(\lambda x_{60}.x_{58}\ x_{60})x_{59})))))$$

For readability, let us rename this residual term:

$$\lambda n.\lambda s.\lambda z.s(s(s(s(s(n(\lambda n'.s\ n')z)))))$$

In this term, the successor function is applied five times, reflecting that the addition function has been specialized with respect to five.

```
module TypefulExpCoerce (int, app, lam, coerce, uncoerce, Exp) where

  [...]

  coerce   :: Exp a → Exp (Exp a)
  uncoerce :: Exp (Exp a) → Exp a

  coerce   (EXP f) = EXP f
  uncoerce (EXP f) = EXP f
```

Figure 3.5: Typeful higher-order abstract syntax with coercions for atomic types

3.2.6 Application 1: Type-directed partial evaluation preserves types

In this section, we use the type inferencer of Haskell as a theorem prover to show that type-directed partial evaluation preserves types. To this end, we implement type-directed partial evaluation using typed abstract syntax.

Typeful type-directed partial evaluation (first variant)

We want the type of `normalize` to be $\mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Exp}(\alpha)$. As a first step to achieve this more expressive type, we shift to the typeful representation of terms from Figure 3.3. The parameterized type constructor $\mathsf{Exp}(\alpha)$ replaces the type Exp. Thus, we change the data type $\mathsf{Reify\_Reflect}(\alpha)$ from Figure 3.4 to the following.

```
data Reify_Reflect a =
  RR { reify   :: a → Exp a,
       reflect :: Exp a → a }
```

This change, however, makes the standard definition of `rra` untypable: The identity function does not have type $\alpha \to \mathsf{Exp}(\alpha)$ (or $\mathsf{Exp}(\alpha) \to \alpha$ for that matter). We solve this problem by introducing two identity functions in the module of typed terms.

$$\begin{array}{rcl} \texttt{coerce} & :: & \mathsf{Exp}(\alpha) \to \mathsf{Exp}(\mathsf{Exp}(\alpha)) \\ \texttt{uncoerce} & :: & \mathsf{Exp}(\mathsf{Exp}(\alpha)) \to \mathsf{Exp}(\alpha) \end{array}$$

At first it might seem that a function of type $\mathsf{Exp}(\alpha) \to \mathsf{Exp}(\mathsf{Exp}(\alpha))$ cannot be the identity. However, internally $\mathsf{Exp}(t)$ is an alias for $\mathsf{Int} \to \mathsf{Term}$, thus discarding t, so in effect we are looking at two identity functions of type $(\mathsf{Int} \to \mathsf{Term}) \to (\mathsf{Int} \to \mathsf{Term})$. Figure 3.5 shows the required changes to the typeful representation of Figure 3.3.

We can now define `rra` using `coerce` and `uncoerce`. The complete implementation is shown in Figure 3.6. Types are represented as in Section 3.2.5, but the types of the represented types differ. We define the instance as follows.

$$\begin{array}{rcl} [\alpha]_1 & = & \mathsf{Exp}(\alpha) \\ [t_0 \to t_1]_1 & = & [t_0]_1 \to [t_1]_1 \end{array}$$

```
module TypefulTdpe where

  import TypefulExpCoerce  -- from Figure 3.5

  data Reify_Reflect(a) =
    RR { reify   :: a → Exp a,
         reflect :: Exp a → a }

  rra =                     -- atomic types
    RR { reify   = λx → coerce x,
         reflect = λx → uncoerce x }

  rrf (t1, t2) =            -- function types
    RR { reify   = λv → lam (λx → reify t2 (v (reflect t1 x))),
         reflect = λe → λx → reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 3.6: A typeful implementation of type-directed partial evaluation

Then the type of $|t|$ is $\mathsf{Reify\_Reflect}([t]_1)$. Haskell infers the following type for the main function.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Exp}(\alpha)$$

This type proves that type-directed partial evaluation preserves types.

N.B. The typeless implementation in Figure 3.4 and the typeful implementation in Figure 3.6 are equally efficient. Indeed, they differ only in the two occurrences of `coerce` and `uncoerce` in `rra` in Figure 3.6, which are defined as the identity function.

Typeful type-directed partial evaluation (second variant)

The two auxiliary functions `coerce` and `uncoerce` are only necessary to obtain an automatic proof of the type-preservation property of type-directed partial evaluation: They are artefacts of the typeful encoding. But could one do without them? In this section, we present an alternative proof of the type preservation of type-directed partial evaluation without using these coercions. Instead, we show that when type-directed partial evaluation is applied to a correct representation of the type of the input value, the residual term has the same type as the input value.

To this end, we implement `rra` as a pair of identity functions, as in Figure 3.4, and we modify the data type Reify_Reflect by weakening the connection between the domains and the codomains of the reify / reflect pairs.

```
module TypefulTdpe where
  import TypefulExp  -- from Figure 3.3

  data Reify_Reflect a b =
    RR { reify   :: a → Exp b,
         reflect :: Exp b → a }
```

```
        [...]
```

These changes make all of `rra`, `rrf`, and `normalize` well-typed in Haskell. Their types read as follows.

$$
\begin{array}{rcl}
\texttt{rra} & :: & \mathsf{Reify\_Reflect}(\mathsf{Exp}(\alpha))(\alpha) \\
\texttt{rrf} & :: & (\mathsf{Reify\_Reflect}(\alpha)(\gamma), \mathsf{Reify\_Reflect}(\beta)(\delta)) \rightarrow \\
 & & \qquad\qquad \mathsf{Reify\_Reflect}(\alpha \rightarrow \beta)(\gamma \rightarrow \delta) \\
\texttt{normalize} & :: & \mathsf{Reify\_Reflect}(\alpha)(\beta) \rightarrow \alpha \rightarrow \mathsf{Exp}(\beta)
\end{array}
$$

The type of `normalize` no longer proves that it preserves types. However, we can fill in the details by hand using the inferred types of `rra` and `rrf`: We prove by induction on the type t that the type of $|t|$ is $\mathsf{Reify\_Reflect}([t]_1)(t)$. For $t = \alpha$, we have $|t| = \texttt{rra}$ which has type $\mathsf{Reify\_Reflect}(\mathsf{Exp}(\alpha))(\alpha)$ as required. For $t = t_1 \rightarrow t_2$, we have $|t| = \texttt{rrf}(|t_1|,\ |t_2|)$. By hypothesis, $|t_i|$ has type $\mathsf{Reify\_Reflect}([t_i]_1)(t_i)$ for $i \in \{1, 2\}$. Hence, by the inferred type for `rrf` we have that $\texttt{rrf}(|t_1|,\ |t_2|)$ has type $\mathsf{Reify\_Reflect}([t_1]_1 \rightarrow [t_2]_1)(t_1 \rightarrow t_2)$ as required. As a corollary we obtain that for all types t,

$$\texttt{normalize}\ |t| :: [t]_1 \rightarrow \mathsf{Exp}(t)$$

This proof gives a hint about how to prove (by hand) that typeless type-directed partial evaluation preserves types.

Example: Church numerals, typefully

Let us revisit the example of Section 3.2.5. We specialize the addition function with respect to a fixed argument using the two typeful variants of type-directed partial evaluation. In both cases the residual terms are the same as in Section 3.2.5. The Haskell expression `normalize` $|t_{\text{add}}|$ has type $[t_{\text{add}}]_1 \rightarrow \mathsf{Exp}([t_{\text{add}}]_1)$ using the first variant and it has type $[t_{\text{add}}]_1 \rightarrow \mathsf{Exp}(t_{\text{add}})$ using the second variant.

3.2.7 Application 2: Type-directed partial evaluation yields normal forms

In this section, we use the type inferencer of Haskell as a theorem prover to show that type-directed partial evaluation yields long beta-eta normal forms. We first specify long beta-eta normal forms, both typelessly and typefully. Then we revisit type-directed partial evaluation, both typelessly and typefully.

Long beta-eta normal forms

We consider explicitly typed λ-terms:

$$
\begin{array}{llcl}
\text{(Types)} & t & ::= & \mathsf{a} \mid t_1 \rightarrow t_2 \\
\text{(Terms)} & e & ::= & x \mid e_0\, e_1 \mid \lambda x :: t.\, e
\end{array}
$$

```
module TypelessNf where

  data Nf_ = AT_ At_
           | LAM String Nf_
  data At_ = VAR String
           | APP At_ Nf_

  type Nf = Int → Nf_
  type At = Int → At_

  app e1 e2  x = APP (e1 x) (e2 x)
  lam f      x = LAM v (f (λ_ → VAR v) (x + 1))
                 where v = "x" ++ show x
  at2nf e x = AT_ (e x)
```

Figure 3.7: Typeless representation of normal forms

```
module TypefulNf where

  data Nf_ = AT_ At_
           | LAM String Nf_
  data At_ = VAR String
           | APP At_ Nf_

  data Nf t = NF (Int → Nf_)
  data At t = AT (Int → At_)

  app      :: At (a → b) → Nf a → At b
  lam      :: (At a → Nf b) → Nf (a → b)

  coerce   :: Nf a → Nf (Nf a)
  uncoerce :: At (Nf a) → Nf a

  at2nf    :: At a → Nf a

  app (AT e1) (NF e2) = AT (λx → APP (e1 x) (e2 x))
  lam f               = NF (λx → let v    = "x" ++ show x
                                     NF b = f (AT (λ_ → VAR v))
                                 in  LAM v (b (x + 1)))

  coerce   (NF f) = NF f
  uncoerce (AT f) = NF (λx → AT_ (f x))

  at2nf    (AT f) = NF (λx → AT_ (f x))
```

Figure 3.8: Typeful representation of normal forms

```
module TypelessTdpeNf where

  import TypelessNf  -- from Figure 3.7

  data Reify_Reflect a =
    RR { reify   :: a → Nf,
         reflect :: At → a }

  rra =             -- atomic types
    RR { reify   = λx → x,
         reflect = λx → at2nf x }

  rrf (t1, t2) =    -- function types
    RR { reify   = λv → lam (λx → reify t2 (v (reflect t1 x))),
         reflect = λe → λx → reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 3.9: Typeless implementation of type-directed partial evaluation with normal forms

Definition 7 (long beta-eta normal forms [61, 80]) *A closed term e of type t is in* long beta-eta normal form *if and only if it satisfies $\cdot \vdash_{\mathrm{nf}} e :: t$ where "$\cdot$" denotes the empty environment and where terms in normal form and atomic form are defined by the following rules:*

$$\frac{\Delta, x :: t_1 \vdash_{\mathrm{nf}} e :: t_2}{\Delta \vdash_{\mathrm{nf}} \lambda x{::}t_1.\, e :: t_1 \to t_2}[\mathsf{lam}] \qquad \frac{\Delta \vdash_{\mathrm{at}} e :: \mathsf{a}}{\Delta \vdash_{\mathrm{nf}} e :: \mathsf{a}}[\mathsf{coerce}]$$

$$\frac{\Delta \vdash_{\mathrm{at}} e_0 :: t_1 \to t_2 \quad \Delta \vdash_{\mathrm{nf}} e_1 :: t_1}{\Delta \vdash_{\mathrm{at}} e_0\, e_1 :: t_2}[\mathsf{app}] \qquad \frac{\Delta(x) = t}{\Delta \vdash_{\mathrm{at}} x :: t}[\mathsf{var}]$$

No term containing β-redexes can be derived by these rules, and the coerce rule ensures that the derived terms are fully η-expanded.

Figure 3.7 displays a typeless representation of normal forms in Haskell. Figure 3.8 displays a typeful representation of normal forms in Haskell.

Typeless type-directed partial evaluation and normal forms

We now reexpress type-directed partial evaluation as specified in Figure 3.9 to yield typeless terms, as also done by Filinski [61]. The type of `normalize` reads as follows.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Nf}$$

This type proves that type-directed partial evaluation yields residual terms in beta normal form since the representation of Figure 3.7 does not allow beta redexes. These residual terms are also in eta normal form because `at2nf` is only applied at base type: residual terms are thus fully eta expanded.

```
module TypefulTdpeNf1 where

  import TypefulNf  -- from Figure 3.8

  data Reify_Reflect a =
    RR { reify   :: a → Nf a,
         reflect :: At a → a }

  rra =             -- atomic types
    RR { reify   = λx → coerce x,
         reflect = λx → uncoerce x }

  rrf (t1, t2) =    -- function types
    RR { reify   = λv → lam (λx → reify t2 (v (reflect t1 x))),
         reflect = λe → λx → reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 3.10: Typeful implementation of type-directed partial evaluation with normal forms (first variant)

Typeful type-directed partial evaluation and normal forms (first variant)

We now reexpress type-directed partial evaluation to yield typeful terms as specified in Figure 3.10. The type of `normalize` reads as follows.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Nf}(\alpha)$$

This type proves that type-directed partial evaluation (1) preserves types and (2) yields terms in normal form.

Typeful type-directed partial evaluation and normal forms (second variant)

On the same ground as Section 3.2.6, i.e., to bypass the artefactual coercions of the typeful encoding of abstract syntax, we now reexpress type-directed partial evaluation to yield typeful terms as specified in Figure 3.11. The type of `normalize` reads as follows.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha)(\beta) \to \alpha \to \mathsf{Nf}(\beta)$$

This type only proves that type-directed partial evaluation yields terms in normal form. As in Section 3.2.6, we can prove type preservation by hand, i.e., that

$$\texttt{normalize}\ |t| :: [t]_2 \to \mathsf{Nf}(t)$$

where the instance of a type is defined by

$$\begin{array}{rcl} [\alpha]_2 & = & \mathsf{Nf}(\alpha) \\ [t_1 \to t_2]_2 & = & [t_1]_2 \to [t_2]_2 \end{array}$$

```
module TypefulTdpeNf2 where

  import TypefulNf  -- from Figure 3.8

  data Reify_Reflect a b =
    RR { reify   :: a → Nf b,
         reflect :: At b → a }

  rra =             -- atomic types
    RR { reify   = λx → x,
         reflect = λx → at2nf x }

  rrf (t1, t2) =    -- function types
    RR { reify   = λv → lam (λx → reify t2 (v (reflect t1 x))),
         reflect = λe → λx → reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 3.11: Typeful implementation of type-directed partial evaluation with normal forms (second variant)

3.2.8 Conclusions and issues

We have presented a simple way to express typed abstract syntax in a Haskell-like language, and we have used this typed abstract syntax to demonstrate that type-directed partial evaluation preserves types and yields residual programs in normal form. The encoding is limited because it does not lend itself to programs taking typed abstract syntax as input—as, e.g., a typeful transformation into continuation-passing style. Nevertheless, the encoding is sufficient to establish two key properties of type-directed partial evaluation automatically.

These two properties could be illustrated more directly in a language with dependent types such as Martin-Löf type theory. In such a language, one can directly represent typed abstract syntax and program type-directed partial evaluation typefully.

3.3 Normalization by evaluation with typed abstract syntax

Note. This section is based on joint work with Olivier Danvy and Kristoffer H. Rose to appear in the Journal of Functional Programming [43].

A preliminary and longer version of this article is available in the proceedings of FLOPS 2001 [42]. We would like to thank Simon Peyton Jones for identifying phantom types in it. The present version has benefited from Richard Bird's editorial advice and from Ralf Hinze's comments.

3.3.1 A write-only typed abstract syntax

In higher-order abstract syntax, the variables and bindings of an object language are represented by variables and bindings of a meta-language. Let us consider the simply typed λ-

calculus as object language and Haskell as meta-language. For concreteness, we also throw in integers and addition, but only in this section.

```
data Term = INT Int | ADD Term Term
          | APP Term Term | LAM (Term → Term)
```

The constructors are typed as follows.

```
INT :: Int → Term                    ADD :: Term → Term → Term
APP :: Term → (Term → Term)          LAM :: (Term → Term) → Term
```

They do not prevent us from forming ill-typed terms. For example, in the scope of these constructors, evaluating `LAM(λx→APP x x)` yields a value of type `Term`.

We can, however, provide a typed interface to these constructors preventing us from forming ill-typed terms.

```
newtype Exp t = EXP Term

int :: Int → Exp Int        add :: Exp Int → Exp Int → Exp Int
int i = EXP (INT i)         add (EXP e1) (EXP e2) = EXP (ADD e1 e2)

app :: Exp (a → b) → (Exp a → Exp b)
app (EXP e1) (EXP e2) = EXP (APP e1 e2)

lam :: (Exp a → Exp b) → Exp (a → b)
lam f = EXP (LAM (λx → let EXP b = f (EXP x) in b))
```

The type `Exp` is parameterized over a type `t` but does not use it: `t` is a *phantom type*.

These typeful constructors prevent us from forming ill-typed terms. For example, in the scope of these constructors, evaluating `lam(λx→app x x)` yields a type error. Conversely, if a term has the simple type `t` then its typed abstract-syntax representation has type `Exp t`, which can be illustrated as follows.

```
λx → x + 5                :: Int → Int
lam (λx → add x (int 5))  :: Exp (Int → Int)
```

We intend to use this typed abstract syntax to show that normalization by evaluation preserves types (Section 3.3.2) and yields normal forms (Section 3.3.3) for the pure and simply typed λ-calculus. Therefore, we are only interested in constructing abstract syntax. (To convert a constructed term into first-order abstract syntax where variables are represented as strings, one needs to add another constructor to `Term` for free variables.) Furthermore, such a write-only typed abstract syntax does not solve the basic problem of programming higher-order abstract syntax in Haskell, which is that the function space in the `LAM` summand is “too big” in the sense that it allows both non-strict and non-total functions. But again, this representation is sufficient for our purpose here. In the remainder of this section, `Term` and `Exp` are restricted to the pure λ-calculus.

3.3.2 Normalization by evaluation preserves types

Normalization by evaluation is an extensional, reduction-free technique for strongly normalizing closed λ-terms. Source terms are represented as meta-language values and a *normalization function* maps these values into a syntactic representation of their normal form.

The technique is extensional instead of intensional because the source terms are (higher-order) values, not (first-order) symbolic representations. It is reduction-free because all the β-reductions needed to yield a normal form are carried out implicitly by the underlying implementation of the meta-language. For this reason, it runs at native speed and thus is more efficient than traditional, symbolic normalization.

Normalization by evaluation uses two type-indexed and mutually recursive functions. One, *reify*, traditionally noted $\downarrow$, maps a value into its representation and the other, *reflect*, traditionally noted $\uparrow$, maps a representation into a value. These two functions are canonically defined as follows, for the simply typed λ-calculus.

$$
\begin{array}{rcl}
t & ::= & \alpha \mid t_1 \rightarrow t_2 \\
\downarrow^{\alpha} & = & \overline{\lambda} v.\, v \\
\downarrow^{t_1 \rightarrow t_2} & = & \overline{\lambda} v.\, \underline{\lambda} x.\, \downarrow^{t_2}\ \overline{@}\, (v\, \overline{@}\, (\uparrow_{t_1}\ \overline{@}\, x)) \\
\uparrow_{\alpha} & = & \overline{\lambda} e.\, e \\
\uparrow_{t_1 \rightarrow t_2} & = & \overline{\lambda} e.\, \overline{\lambda} x.\, \uparrow_{t_2}\ \overline{@}\, (e\, \underline{@}\, (\downarrow^{t_1}\ \overline{@}\, x))
\end{array}
$$

where overlined λ and @ denote meta-level abstractions and applications, respectively, and underlined λ and @ denote object-level abstractions and applications.

A simply typed term is normalized by reifying its value. For example, let us consider Church numbers.

$$
\begin{array}{rclcrcl}
\mathit{zero} & = & \overline{\lambda} s.\, \overline{\lambda} z.\, z & \qquad & \mathit{succ} & = & \overline{\lambda} n.\, \overline{\lambda} s.\, \overline{\lambda} z.\, s\, \overline{@}\, (n\, \overline{@}\, s\, \overline{@}\, z) \\
\mathit{three} & = & \mathit{succ}\, \overline{@}\, (\mathit{succ}\, \overline{@}\, (\mathit{succ}\, \overline{@}\, \mathit{zero})) & \qquad & \mathit{add} & = & \overline{\lambda} m.\, \overline{\lambda} n.\, \overline{\lambda} s.\, \overline{\lambda} z.\, m\, \overline{@}\, s\, \overline{@}\, (n\, \overline{@}\, s\, \overline{@}\, z)
\end{array}
$$

Reifying three yields $\underline{\lambda} s.\, \underline{\lambda} z.\, s\, \underline{@}\, (s\, \underline{@}\, (s\, \underline{@}\, z))$, i.e., the representation in normal form of 3. Similarly, reifying $\mathit{add}\, \overline{@}\, \mathit{zero}$ yields

$$\underline{\lambda} n.\, \underline{\lambda} s.\, \underline{\lambda} z.\, n\, \underline{@}\, (\underline{\lambda} n'.\, s\, \underline{@}\, n')\, \underline{@}\, z$$

i.e., the representation in long $\beta\eta$-normal form of the identity function over Church numbers, reflecting that zero is identity for addition. And finally, reifying $\mathit{add}\, \overline{@}\, \mathit{three}$ yields the representation in normal form of a function iterating the successor function three times, i.e., $\underline{\lambda} n.\, \underline{\lambda} s.\, \underline{\lambda} z.\, s\, \underline{@}\, (s\, \underline{@}\, (s\, \underline{@}\, (n\, \underline{@}\, (\underline{\lambda} n'.\, s\, \underline{@}\, n')\, \underline{@}\, z)))$. The source terms are values (i.e., with overlined λ and @) and, using $\downarrow$, we have reified them into a syntactic representation of their normal form (i.e., with underlined λ and @).

The type of a Church number is `(a→a) → a → a`. The type of its normal form is `Term`, or, perhaps more vividly, `(Exp a → Exp a) → Exp a → Exp a`.

Normalization by evaluation is defined by induction on the structure of types, which makes it a natural candidate to be expressed with type classes. We thus define a type class

`Nbe` hosting two type-indexed functions, `reify` and `reflect`. Representing object terms with the type `Term` of Section 3.3.1 would give us the usual uninformative type `t→Term` for `reify` and `Term→t` for `reflect`. Instead, let us use the parameterized type `Exp` of Section 3.3.1.

```
    class Nbe a
      where  reify :: a → Exp a    reflect :: Exp a → a
```

The challenge now is to populate this type class with values of function type and of base type implementing normalization by evaluation. If we can do that, the type inferencer of Haskell will act as a theorem prover and will demonstrate that this implementation of normalization by evaluation preserves types.

The canonical definition above dictates how to instantiate `Nbe` at function type.

```
    instance  (Nbe a, Nbe b) ⇒ Nbe (a→b)
      where  reify   v = lam (λx→reify (v (reflect x)))
             reflect e = λx→reflect (app e (reify x))
```

For base types, `reify` and `reflect` are two identity functions. To be type correct, however, `reify` must produce a term and `reflect` must consume a term. We can ensure that `reify` produces a term when its argument is a term. Similarly, we can ensure that `reflect` consumes a term when its result is a term. Taking advantage of the fact that the type parameter of `Exp` is a phantom type, we thus introduce the following two 'phantom' identity functions for the base case.

```
coerce :: Exp (Exp a) → Exp a      uncoerce :: Exp a → Exp (Exp a)
coerce (EXP v) = EXP v             uncoerce (EXP e) = EXP e

    instance  Nbe (Exp a)
      where  reify = uncoerce    reflect = coerce
```

A value `v` is normalized by applying `reify` to it. In usual implementations of normalization by evaluation, (a representation of) the type of `v` must be supplied on par with `v`, as an input data. Here, because we use type classes, this type is supplied as a cast, to resolve overloading. It is obtained by instantiating type variables `a` with `Exp a`, in the original type. So for example, `id . id` has the type `a→a`. Reifying it at type `Exp a → Exp a` yields `λx→x`, and reifying it at type `(Exp a → Exp a) → (Exp a → Exp a)` yields `λx→λx'→x x'`.

3.3.3 Normalization by evaluation yields normal forms

In the simply typed λ-calculus, long $\beta\eta$-normal forms are closed terms without β-redexes that are fully η-expanded with respect to their type. A closed term e of type t and in normal form satisfies $\vdash_{\text{nf}} e :: t$, where terms in normal form (and atomic form) are defined by the following rules.

$$\frac{\Delta, x :: t_1 \vdash_{\text{nf}} e :: t_2}{\Delta \vdash_{\text{nf}} (\lambda x{::}t_1.\, e) :: t_1 \to t_2}(\text{Lam}) \qquad \frac{\Delta \vdash_{\text{at}} e :: \alpha}{\Delta \vdash_{\text{nf}} e :: \alpha}(\text{Coerce})$$

$$\frac{\Delta \vdash_{\text{at}} e_0 :: t_1 \to t_2 \quad \Delta \vdash_{\text{nf}} e_1 :: t_1}{\Delta \vdash_{\text{at}} e_0\, e_1 :: t_2}(\text{App}) \qquad \frac{\Delta(x) = t}{\Delta \vdash_{\text{at}} x :: t}(\text{Var})$$

No term containing β-redexes can be derived by these rules, and restricting the Coerce rule to base types ensures that the derived terms are fully η-expanded.

As in Section 3.3.1, we provide a typed interface to the constructors of terms in normal form, preventing us from forming ill-typed terms.

```
data NfTerm = COERCE AtTerm | LAM (AtTerm → NfTerm)
data AtTerm = APP AtTerm NfTerm

newtype NfExp a = NF NfTerm
newtype AtExp a = AT AtTerm

app’ :: AtExp (a → b) → (NfExp a → AtExp b)
app’ (AT e1) (NF e2) = AT (APP e1 e2)

lam’ :: (AtExp a → NfExp b) → NfExp (a → b)
lam’ f = NF (LAM (λx → let NF t = f (AT x) in t))

coerce’ :: AtExp (NfExp a) → NfExp a
coerce’ (AT v) = NF (COERCE v)

uncoerce’ :: NfExp a → NfExp (NfExp a)
uncoerce’ (NF e) = NF e
```

These declarations specialize the representation from Section 3.3.2 to reflect that the represented terms are in normal form. As in Section 3.3.2, we provide two phantom identity functions, `coerce’` and `uncoerce’`, where `coerce’` constructs terms that arise from using the above Coerce rule.

Thus equipped, we can re-express normalization by evaluation in an implementation that yields a representation of λ-terms in normal form.

```
class  Nbe’ a
  where  reify :: a → NfExp a    reflect :: AtExp a → a
```

Again, the challenge is to populate this type class with values of function type and of base type implementing normalization by evaluation. If we can do that, the type inferencer of Haskell will act as a theorem prover and will demonstrate that this implementation of normalization by evaluation preserves types and yields normal forms.

The instances use the constructors for terms in normal forms but are otherwise defined as in Section 3.3.2.

```
instance  (Nbe’ a, Nbe’ b) ⇒ Nbe’ (a → b)
  where  reify   v = lam’ (λx → reify (v (reflect x)))
         reflect e = λx → reflect (app’ e (reify x))

instance  Nbe’ (NfExp a)
  where  reify = uncoerce’    reflect = coerce’
```

As in Section 3.3.2, reifying `id . id` at type `NfExp a → NfExp a` yields `λx→x`, and reifying it at type `(NfExp a → NfExp a) → (NfExp a → NfExp a)` yields `λx→λx'→x x'`.

For a last example, here are the Haskell definitions of Church numbers mentioned in Section 3.3.2.

```
type Number a = (a → a) → a → a
zero  = λs z→z
succ  = λn s z→s (n s z)
three = succ (succ (succ zero))
add   = λm n s z→m s (n s z)
```

Reifying `three`, `add zero`, and `add three` at type `Number (Exp a) → Number (Exp a)` gives the text of their normal form.

3.3.4 Conclusions and issues

We have presented a simple encoding of typed abstract syntax in Haskell, and we have used this typed abstract syntax to demonstrate that normalization by evaluation preserves simple types and yields residual programs in $\beta\eta$-normal form. The encoding is write-only because it does not lend itself to programs taking typed abstract syntax as input—as, e.g., a typed transformation into continuation-passing style. Nevertheless, it is sufficient to establish two key properties of normalization by evaluation automatically, using the Haskell type inferencer as a theorem prover.

These two properties could be illustrated more directly in a language with dependent types such as Martin-Löf's type theory. In such a language, one can directly embed simply typed λ-terms (in normal form or not), express normalization by evaluation, and prove that it preserves types and yields normal forms.

Normalization by evaluation takes its roots in type theory [27, 99], proof theory [9, 10, 11], logic [3], category theory [2, 28, 117], and partial evaluation [34, 61, 119, 121]. Long $\beta\eta$-normal forms were specified, e.g., in Huet's thesis [80]. The particular characterization we use originates in Pfenning's work on Logical Frameworks, and so does higher-order abstract syntax [110]. We use it further to pair normalization by evaluation and run-time code generation [5]. Our typed abstract syntax is akin to Leijen and Meijer's embedding of SQL into Haskell, which introduced phantom types [93]. Phantom types provide a typing discipline for otherwise untyped values such as pointers in a foreign language interface [63].

Chapter 4

Run-time code generation

Automated program-optimization techniques, such as partial evaluation, can improve the efficiency of programs by several orders of a magnitude. However, they are often expressed as source-to-source transformations. A separate compilation phase is required to run the optimized programs.

In this chapter we present a library of byte-code combinators for OCaml, a dialect of ML. They allow direct generation of optimized programs as OCaml byte code. We present two applications of byte-code combinators in program optimizations. One is in semantics-directed compilation: From a definitional interpreter for an imperative language we obtain a compiler producing byte code. We have thus achieved a source-to-target transformation. The other application is in run-time specialization: We implement a type-directed partial evaluator producing residual programs as byte code. The input to type-directed partial evaluation is already compiled code. We have thus also achieved a target-to-target transformation.

4.1 Introduction

Lisp dialects, such as Scheme [89], support a distinctive style of programming where quasi-quotation is used to construct syntactic representations of programs (S-expressions) and where the procedure `eval` is used to execute such S-expressions. Quasi-quotation, S-expressions, and the `eval` procedure probably account for a good part of the popularity of Lisp-like languages [8]. One particular application is in run-time code generation. Consider, for example, the problem of computing the nth power of several arguments, where n is unknown. The following traditional solution maps a general procedure for computing x^n onto a list of elements.

```
(define (power n x)
  (if (zero? n) 1 (* x (power (- n 1) x))))
```

```
(define (main1 n xs)
  (map (lambda (x) (power n x)) xs))
```

The problem of specializing the power function to a known value for n can be solved by standard partial evaluation. However, in the case we consider here, n is unknown and cannot be inlined to yield a specialized version of `power`. Instead, we can construct (the text of) a specialized version when n becomes known and then use `eval` to produce an executable function. This idea is implemented using the generating extension of the power function, as follows.

```
(define (power-gen n)
  (if (zero? n) '1 `(* x ,(power-gen (- n 1)))))

(define (main2 n xs)
  (map (eval `(lambda (x) ,(power-gen n))) xs))
```

Given a fixed exponent n, the auxiliary procedure `power-gen` constructs a specialized multiplication which is wrapped inside a λ-abstraction as follows. (Since the specialized program contains only one variable we do not bother giving it a fresh name.)

$$\texttt{(lambda (x) } \underbrace{\texttt{(* x (* x} \cdots \texttt{(* x}}_{n} \texttt{ 1))))}$$

The result of evaluating this expression is a specialized version of the power function which is faster to execute than the unspecialized power function. For example, computing the fifth power of 10000 small integers using the improved solution is about 2.5 times faster than using the original solution (measured on a 266MHz Pentium system running Petite Chez Scheme [109]).

The style of run-time code generation that we have sketched here can easily be combined with traditional partial evaluation [83] to yield run-time specialization. In run-time code generation, however, it is crucial that the compilation or interpretation of the generated programs exercise as little overhead as possible. Most run-time code generators therefore make an effort to pre-compile as much of the generated program fragments as possible before the main program is executed. These pre-compiled code templates are then combined at run-time to produce the final executable program.

Code templates cannot generally be compiled completely into executable code. For example, the position of free variables in the pre-compiled program fragments may be unknown when these positions (in, e.g., a register or the stack) depend on where the variable is bound. The pre-compiled program fragments therefore have “holes” for the free variables which are adjusted when the position of variables become known. In this chapter we define a set of combinators for generating OCaml byte code at run time. These byte-code combinators can be combined freely to produce complete programs which can then be executed directly without first applying a stand-alone compiler. We thus provide support for run-time code generation using generating extensions as sketched above. We also use the byte-code combinators as code-generating primitives in an implementation of type-directed partial evaluation. The result is a run-time specializer for higher-order OCaml programs.

This chapter is structured as follows. In Section 4.2 we present a deforestation technique for inductively defined data types. In Section 4.3 we give an overview of the OCaml byte-code compiler and run-time system and we apply deforestation to an abstract syntax tree of OCaml expressions. The result is a set of byte-code combinators implemented in OCaml. In Section 4.4 we apply these byte-code combinators to semantics-directed compilation and in Section 4.5 we apply them to run-time specialization. In Section 4.6 we discuss related work. The implementation of the byte-code combinators is given in Section 4.7. Section 4.8 concludes.

4.2 Deforested data types

In statically typed, higher-order languages, such as Haskell [59] and ML [102], a data type is an inductively defined disjoint sum T with n injective constructors $C_i : t_i[T] \to T$ for $1 \leq i \leq n$. (We write $t[T]$ for an expression that may contain T as a free variable.) Such a data type can be modeled by an (infinite) union of disjoint sets,

$$\bigcup_{n \in \omega} D_n, \quad \text{where} \begin{cases} D_0 & = \emptyset \\ D_{k+1} & = \bigcup_{1 \leq i \leq n} \{(i, x) \mid x \in t_i[D_k/T]\} \end{cases}$$

The only meaningful operation on a data type is the corresponding fold-function (or catamorphism)

$$\mathit{fold}^T_\alpha : T \to (t_1[\alpha/T] \to \alpha) \times \cdots \times (t_n[\alpha/T] \to \alpha) \to \alpha$$

where the result type α may be a function type $\beta_1 \to \cdots \to \beta_k$. A data type is uniquely defined by its constructors (up to isomorphism).

Example 1 Lists and binary trees are traditional examples of inductively defined data types given by the following injective constructors.

$$\begin{array}{rcl@{\qquad}rcl} \mathit{nil}_\alpha & : & \mathbf{1} \to \mathsf{list}_\alpha & \mathit{leaf}_\alpha & : & \alpha \to \mathsf{tree}_\alpha \\ \mathit{cons}_\alpha & : & \alpha \times \mathsf{list}_\alpha \to \mathsf{list}_\alpha & \mathit{node}_\alpha & : & \mathsf{tree}_\alpha \times \mathsf{tree}_\alpha \to \mathsf{tree}_\alpha \end{array}$$

Other (degenerated) examples of data types include Booleans, the type `option` of ML, and the type `Either` of Haskell. The fold function for Booleans is an if-expression.

Consider a particular application of the fold function,

$$f_\alpha(v) = \mathit{fold}^T_\alpha\ v\ (g_1, \ldots, g_n)$$

When f is applied to an element $v : T$, a case dispatch on v essentially replaces each of the constructors $C_i : t_i[T] \to T$ with the corresponding $g_i : t_i[\alpha/T] \to \alpha$. The result is an element of type α. A natural alternative to applying the fold function is to directly replace the C_i's by g_i's in the construction of the element v.

Example 2 Instead of evaluating

$$\mathit{fold}_{\alpha}^{\mathsf{list}_{\mathsf{int}}}\ (\mathit{cons}_{\mathsf{int}}(41, \mathit{cons}_{\mathsf{int}}(42, \mathit{nil}_{\mathsf{int}}())))\ (g_1, g_2)$$

using the auxiliary functions $g_1 : \mathbf{1} \to \alpha$ and $g_2 : \mathsf{int} \times \alpha \to \alpha$ we can directly evaluate

$$g_2(41, g_2(42, g_1()))$$

Evaluating the latter is more efficient than evaluating the former since the case dispatch is removed.

Computations inside the g_i's may be performed at the time of constructing an element instead of at the time of deconstructing the element using its associated fold function. As a consequence, if elements are constructed at an inexpensive early stage and deconstructed at an expensive late stage then the alternative representation may be more efficient than the traditional representation.

Consider, for example, the following function which sums a list of integers using an accumulator.

$$\begin{array}{rcl} \mathit{sum}\ (\mathit{nil}()) & = & \lambda a.\, a \\ \mathit{sum}\ (\mathit{cons}(0, \mathit{xs})) & = & \lambda a.\, \mathit{sum}\ \mathit{xs}\ a \\ \mathit{sum}\ (\mathit{cons}(x, \mathit{xs})) & = & \lambda a.\, \mathit{sum}\ \mathit{xs}\ (x + a) \end{array}$$

The function sum tests whether it has encountered the identity element 0 in the list and short-cuts the addition if this is the case. However, when applied to a list and an initial value for the accumulator, it must traverse the whole list and make the test for each element. It is straightforward to implement this function using the fold-function for lists. The corresponding g_i's are defined as follows.

$$\begin{array}{rcl} g_1\ () & = & \lambda a.\, a \\ g_2\ (0, \mathit{xs}) & = & \lambda a.\, \mathit{xs}\ a \\ g_2\ (x, \mathit{xs}) & = & \lambda a.\, \mathit{xs}\ (x + a) \end{array}$$

In the context of the summation function, these g_i's provide a representation of lists which is more efficient than the traditional one. Although summing a list still traverses the whole list, it does not test for occurrences of 0.

The alternative to an inductively defined data type corresponds to a Church encoding [20]. Such an encoding is not only a more efficient representation than traditional disjoint sums. The transformation can also expose properties of the data type that are otherwise hidden. For example, Church-encoded data types have been used to encode dependently typed functions in the Hindley-Milner type disciplines of ML [32, 64, 119, 145]. They can also provide a link between different styles of implementations, such as relating a direct-style implementation using an accumulator with a continuation-passing style implementation [37, 38].

Our interest in Church-encoded data types is due to their use in deforestation of abstract syntax trees in compilers. Most compilers can be expressed using a fold function over abstract syntax trees [1]. By Church-encoding the abstract syntax tree, computations in the compiler that only depend on the syntax can be carried out when an abstract syntax tree is constructed. In stand-alone compilers, where both the construction and the deconstruction of abstract syntax is done at compile time, such an improvement is negligible. In staged evaluation, such as in run-time code generation, where abstract syntax is generated at an early stage and compiled to executable code at a later stage, using a Church-encoded abstract syntax tree may be more efficient than using a traditional representation.

4.3 Run-time code generation for OCaml

We describe a representation of byte-code combinators for OCaml byte code. These correspond to a Church-encoded abstract syntax of OCaml expressions.

4.3.1 Overview of OCaml

OCaml is a dialect of ML [102]: It is a strict, higher-order, statically typed, language with a module system [96]. The OCaml implementation consists of a byte-code compiler, a native-code compiler, and a run-time system with a virtual machine for running byte-code executables. Both compilers are implemented in OCaml and they share a common front end. The run-time system consists of a byte-code interpreter, a garbage collector, and a set of predefined library procedures. It is implemented in C.

The 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 produce an abstract syntax tree.

- **Type analysis**

  This phase type-checks the source program. It produces a type-annotated abstract syntax tree.

- **Semantics-preserving translations**

  This phase translates OCaml expressions into an extended λ-calculus. The major difference between an OCaml expression and a λ-term is that modules and functors are represented as tuples and higher-order functions in the λ-terms.

- **Code generation**

  This phase produces a list of symbolic byte-code instructions from a λ-term.

- **Byte-code emission**

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

The byte-code run-time system

OCaml's run-time system provides memory management, a virtual machine, and primitive operations. The memory is split into a stack and a heap. Function arguments, return addresses, let-bound variables, and temporary values are stored on the stack. So are instruction operands that do not fit into registers. For example, there is one instruction that allocates tuples and vectors by copying the top portion of the stack the heap.

Heap-allocated blocks are tagged by the memory management system to differentiate values during garbage collection. These tags are not accessible to any of the instructions. Small constants, such as integers, booleans, and characters, are represented as unboxed integers. All other values are represented by pointers to heap-allocated data. Function values are represented by closures that group a code pointer together with values for the free variables of the function. Blocks of byte-code instructions are heap-allocated and are subject to garbage collection.

The virtual machine executes OCaml byte-code instructions. It uses the following registers. The code pointer, `pc`, points to the current instruction to execute. Jumps and returns may set this register explicitly but otherwise it is just incremented to point to the next instruction when a new cycle is started. The stack pointer, `sp`, points to the top of the stack. The accumulator, `accu`, contains the most recently computed value. It also points to the closure in an application of a function. The current environment, `env`, points to a heap-allocated block of values for free variables in the current closure. In fact, `env` points directly to the current closure itself. There are a few other registers that we do not deal with.

The OCaml implementation encourages the use of curried functions and applications by compiling these into efficient byte code [95]. Traditional approaches to evaluating strict functional languages allocate one closure per λ-abstraction. For example, using a naive right-to-left evaluation strategy to apply a curried n-argument closure $f = \lambda x_1. \ldots \lambda x_n. e$ to k arguments $a_1 \cdots a_k$ amounts to push the values of the a_i on the stack, load the accumulator with the closure for f, and then repeatedly call the closure in the accumulator until all arguments are processed. Each call processes one argument and returns a new closure in the accumulator. In contrast, applying an uncurried n-argument closure $f = \lambda(x_1 \ldots x_n). e$ to n arguments $f\,(a_1, \ldots, a_k)$ amounts to allocate a stack frame containing all the values of the a_i, evaluate f to a closure, and call it. The call processes all arguments at once. OCaml's run-time system supports both of these function call mechanism. But in addition, OCaml defines instructions for applying a curried function to k arguments using one instruction without introducing k intermediate closures.

4.3.2 A library of byte-code combinators for OCaml

Byte-code combinators encapsulate enough information that they can be reassembled into a sequence of byte-code instructions.

(1) A byte-code combinator carries a list of the variables that occur free in the expression it represents. Such a list is used to construct the byte-code instructions for creating closures.

(2) If a byte-code combinator represents a variable, then it carries the name of that variable. This information is used in generating byte-code combinators for let-expression that preserves tail calls, as discussed below.

(3) A byte-code combinator carries a function that generates the actual byte-code instructions. It takes two arguments, an environment mapping variable names to stack or environment positions and a list of byte-code instructions for the continuation. It adds code for the current byte-code combinator to the front of the continuation.

Byte-code combinators efficiently support two key operations, namely concatenation of byte-code instructions and instantiation of free variables. When the code-generating function of a complete byte-code combinator is applied to an environment and a continuation, byte-code instructions are generated in a backwards manner using the continuation and the positions of variables are resolved using the environment. There is no copying of the generated byte-code instructions and they are not traversed once they are created. The type of byte-code combinators is `exp` and is defined as follows.

```
type code = instruction list
type exp  = ide list * ide option * (env * code → code)
```

Here `instruction` is a data type of symbolic byte-code instructions defined in the OCaml compiler. The code-generating function of a byte-code combinator corresponds to a Church encoding of an abstract syntax tree of expressions in the context of a compiler of type

```
comp : ast → (env * code) → code
```

For byte-code combinators involving variables and bindings (such as variables, λ-abstractions, and let-expressions) we provide both a low-level first-order interface and a higher-order interface similar to a higher-order abstract syntax [110]. The low-level interface allows direct generation and manipulation of variables, λ-abstractions, and let-expressions. The higher-order interface groups common patterns involving bindings into convenient functions.

OCaml byte-code combinators

Below follows a description of a library of byte-code combinators. Each byte-code combinator corresponds naturally to an OCaml expression. The byte-code combinators are implemented in the OCaml compiler as part of the interactive environment.

- `mkunit : exp`
- `mkbool : bool → exp`
- `mkint : int → exp`
- `mkstr : string → exp`

  Construct values of base types.

- `mkglob : string → exp`
- `mkqref : string list → exp`

  Construct global variables. The first function generates a reference to a global variable from a string of its name. The second function generates a reference to a field of a global module. At run time, global data is stored in a table. The search for the index of global variables in the table is done when the byte-code combinators are applied.

- `mktup : exp list → exp`

  Constructs a finite product of more than 1 element. An auxiliary function generates code that evaluates a sequence of expressions and pushes their results on the stack. The first element must be stored in the accumulator.

- `mkprj : int → exp → exp`

  Constructs a projection of a tuple of values. The index is known at the time of generating the byte-code combinator. It is assumed that the index is within the range of the heap-allocated block representing the tuple.

- `mkift : exp → exp → exp → exp`

  Constructs an if-expression.

- `mkvar : ide → exp`

  This function is used in generating fresh variables independently of their binding λ-abstractions or let-expression.

- `mklam1 : ide → exp → exp`
- `mklam : (exp → exp) → exp`
- `mkclam : int → (exp list → exp) → exp`

  Construct λ-abstractions. The function `mklam1` provides a low-level first-order interface while `mklam` and `mkclam` provide higher-order interfaces. The first two functions construct unary lambda expressions. The third constructs a curried n-ary lambda expressions given an integer n. It generates optimized code similar to what the OCaml byte code compiler produces for curried λ-abstractions.

- `mkapp : exp → exp → exp`
- `mkcapp : exp → exp list → exp`

  Construct function applications. The first function constructs an application of a function to one argument. The second function constructs a curried application of a function to n arguments given a list of length n. It generates optimized code similar to what the OCaml byte-code compiler produces for curried applications.

- `mklet1 : ide → exp → exp → exp`
- `mklet  : exp → (exp → exp) → exp`

  Construct let-expressions. The first function provides a low-level first-order interface. The second function provides a higher-order interface. (The second function is implemented in terms of the first.)

  These constructors preserve tail-calls: For example, instead of generating byte-code instructions corresponding to an expression `let x =` E `in x` they generate simply the byte-code instructions corresponding to E. Such an optimization is not performed by the OCaml compiler. It is sometimes needed when let-expressions are generated automatically, as done, e.g., by partial evaluation.

- `mkseq : exp → exp → exp`

  Constructs a sequencing of two expressions. This byte-code combinator does not generate any instructions itself. Instead, it concatenates the instructions from the two subcombinators.

- `mkref : exp → exp`
- `mkget : exp → exp`
- `mkset : exp → exp → exp`

  Construct byte-code combinators for allocating a mutable cell, for accessing the contents of a mutable cell, and for overwriting the contents of a mutable cell.

- `mkadd : exp → exp → exp`
- `mksub : exp → exp → exp`
- `mkmul : exp → exp → exp`
- `mkeqint : exp → exp → exp`
- `mklss : exp → exp → exp`
- `mkleq : exp → exp → exp`
- `mkgre : exp → exp → exp`
- `mkgeq : exp → exp → exp`

  Construct byte-code combinators corresponding binary addition, subtraction, multiplication, and comparisons.

The implementation of the byte-code combinators and a description of the byte-code instructions they generate are shown in Section 4.7.

An `eval` for OCaml byte code

To run a byte-code combinator, its code-generating function is first applied to an empty environment and a continuation consisting only of a symbolic return instruction. The result is a list of symbolic instructions. Using functions provided by OCaml's interactive run-time system, the list of symbolic instructions is then written to the memory in the form of executable byte-code instructions. We use the following two auxiliary function as a front-end to the internals of the run-time system.

- `run_code : instruction list → 'a`

  This function writes a list of symbolic byte-code instructions to the memory, relocates global pointers in the allocated block, and passes it to the virtual machine for execution.

- `run_exp  : exp  → 'a`

  This function instantiates a byte-code combinator and executes the resulting list of instructions.

```
let run_exp (_, _, f) = run_code (f ([], [Kreturn 1]))
```

If a complete byte-code combinator contains free variables, then instantiation will stop when the code generator for variables (`mkvar`) is applied. Together, byte-code combinators and `run_exp` supports run-time code generation in the same way as Lisp-like S-expressions and `eval`.

Example 3 Consider again the generating extension of the power function from the introduction. It can be implemented in OCaml as follows.

```
let rec power_gen n x =
  if n = 0 then mkint 1 else mkmul x (power_gen (n - 1) x)
let main n (xs : int list) : int list =
  List.map (run_code (mklam (fun x → power_gen n x))) xs
```

Specializing the power function to the exponent 3 yields the following byte-code instructions.

```
        closure L1, 0     |        acc 1
        return 1          |        mulint
L1:     const 1           |        push
        push              |        acc 1
        acc 1             |        mulint
        mulint            |        return 1
        push              |
```

Type safety

The byte-code combinators provide an untyped interface to OCaml byte code. OCaml is a statically typed language, so no type-checks occur in the run-time system. There are, however, no compile-time type checks to ensure that only legal byte-code combinators are generated. The programmer must manually cast the results of using `run_code` and `run_exp` to the correct type of the values they return.

Static type-checking in the context of run-time code generation is somewhat of an open problem. Davies's $\lambda^{\bigcirc}$-calculus [45], motivated by linear-time temporal logic, provides a type $\bigcirc\tau$ of programs of type τ. A complete program of type $\bigcirc\tau$ evaluates to a value $\mathbf{next}\ M$ where M is a program of type τ that can be evaluated subsequently. The type-system of $\lambda^{\bigcirc}$ is able to describe standard partial evaluation where one program (the partial evaluator) generates another complete program (the residual program). For example, a partial evaluator that specializes programs of type $S \times D \to R$ has type $\bigcirc(S \times D \to R) \to S \to \bigcirc(D \to R)$. However, in $\lambda^{\bigcirc}$, there is no way to express immediate evaluation of sub-terms because terms of type $\bigcirc\tau$ may contain free variables. The same problem actually exists for Lisp-like `eval` procedures: The source S-expression may contain free variables. In Scheme, this problem is solved by passing an environment of bindings to `eval` along with the S-expression to evaluate [89, 115]. This approach cannot easily be adapted to $\lambda^{\bigcirc}$, however, since such environments cannot be given a meaningful type.

Davies and Pfenning's λ -calculus [46], motivated by modal logic, provides a type $\Box\tau$ of closed programs of type τ. Therefore, since it contains no free variables, a sub-term of type $\Box\tau$ can be directly evaluated to a value of type τ. However, since only closed programs can be constructed these often contain administrative redexes [45, 47]. At any rate, neither $\lambda^{\bigcirc}$ nor λ have been designed for a strict language with mutable state such as OCaml. There are other approaches to languages that support both generation of staged programs and for evaluating them but these seem to be operationally rather than logically motivated [56, 133, 134].

Lisp-like languages, being dynamically typed, leave the generation of correctly staged programs to the programmer. The challenge is not to mix computed values with (textual) program parts. The following two expressions illustrate how a value (the addition function) may end up in the text of a program and how the text of an identifier (+) may be evaluated, here in Scheme.

```
`(apply ,+ '(4 5))  ⟶  (apply #<procedure +> '(4 5))
`(apply '+ '(4 5))  ⟶  (apply '+ '(4 5))
```

Taken as two-stage programs, both expressions are incorrect. We notice, however, that the result of the first expression is a valid argument to Chez Scheme's `eval` [51]:

```
(eval `(apply ,+ '(4 5)))  ⟶  9
```

The Chez Scheme compiler directly inlines the value of the addition function as a constant in the generated target code. We also notice that the value of the second "invalid" expression above is a valid argument to `eval` in Emacs Lisp:

```
(eval `(apply '+ '(4 5)))  ⟶  9
```

The Lisp function `apply` interprets the function argument as an expression.

For statically typed languages, implementing S-expressions as a first-order data type prevents these problems. In addition, under certain conditions, "phantom types" can provide a typing discipline for data types by restricting generated terms to the simply typed

λ-calculus. (Phantom types are discussed in Chapter 2.) The idea is to parameterize the type of term-representations over the type of the represented terms. In a similar fashion, it is possible to give types to byte-code combinators. However, phantom types only apply to certain representations of terms. Variables and bindings must be implemented using a higher-order abstract syntax, so phantom types does not give meaningful types to the byte-code combinators `mkglob`, `mkvar`, `mklam1`, and `mklet1`. Furthermore, variable-length argument list for the byte-code combinators `mktup`, `mkprj`, `mkclam`, and `mkcapp` cannot be typed either. We do not use phantom types in the rest of this chapter.

4.4 Semantics-directed compilation

The pipeline of programming-language development involves analyzing and reasoning about existing languages, inventing and designing new languages, implementing compilers and interpreters, and using the language to solve programming problems. These stages are not always followed chronologically as listed. For example, using an inappropriate language to solve a specific problem may suggest to design a new and better language for the task. Similarly, implementing a compiler for a language may suggest a different design to facilitate efficient compilation. In fact most realistic languages pass through several rounds of analysis, design, and implementation, and are even used meanwhile by the end-users.

The reference points for researchers involved in analyzing, designing, and implementing a programming language and for programmers who use the language to solve specific problems are formal or informal explanations of what programs of the language look like and how they behave. The "look" of a programming language, commonly called its syntax, is most succinctly described by a (context-free) grammar. A grammar is a formal description that lists the rules allowed in constructing programs of the language. The syntax of the vast majority of programming languages and other formal languages is given by grammars.

The behavior of deterministic programs can only meaningfully be given by unambiguous means. An unambiguous explanation of the behavior of a program is the only contract that avoids conflicts between, e.g., the intended behavior of program and its actual behavior when executed on a physical machine. A property of a programming language may intentionally be unspecified but even that must be communicated to the users of the language. Whether an English explanation of the language serves as an unambiguous description is a question that can be debated. Numerous examples have shown, however, that programming languages are most succinctly described by formal semantics. A formal semantics provides unambiguous means for communicating the behavior of programs among the programming languages researcher and end-users. In fact, the formal semantics itself may also provide guide-lines at each stage of the programming-language development, such as, e.g., by suggesting to design more general features or to implement more efficient compilers.

The purpose of an implementation of a programming language is to model its look and behavior on a physical machine. The task of writing a compiler or an interpreter can be seen as translating the syntax and semantics of the language into a program that processes programs of the language. The problem of translating a grammar into a parser is well-supported by most general-purpose languages. They typically offer a tool that generates lexical analyzers and parsers from textual description of the syntax of tokens and compound program parts [1, 82, 97].

The semantics of a programming language formally specifies how programs are translated into semantic entities. For example, a denotational semantics specifies the meaning of programs by mapping program phrases to mathematical entities, usually domains (i.e., function spaces with certain properties) and functions and (other) relations on domains [123, 131]. An equivalent view states that a *concrete* denotational semantics maps program phrases to terms in a mathematical meta language and that these terms then denote the semantic entities [140, 141, 142]. Due to the foundational work by Scott, the λ-calculus has proven a sound meta language [125].

The λ-calculus is also embodied in higher-order programming languages, such as Haskell [59], Scheme [89], and ML [102]. These languages seemingly provide a connection between the λ-calculus as a mathematical meta language and as a programming language. A direct translation from the meta language into a higher-order programming language is, however, not always possible. It is tempting, but often incorrect, to equate the function spaces denoted by higher-order functions with the function spaces of the mathematical meta language. Yet, when these problems are carefully addressed, a denotational semantics can be translated into a *definitional interpreter*, thus providing a direct implementation for a programming language [116]. This approach is particularly attractive since partial evaluation enables compilation of source programs given their definitional interpreters [65]. This style of *semantics-directed compilation* has been used to generate compilers for a wide variety of functional [13, 86, 87], logical [22, 37, 38, 88], and imperative languages [12, 23, 44, 67] as well as for semantics descriptions of programming languages [14, 15, 41].

Traditional partial evaluation yields target programs in the implementation language of the definitional interpreter, in our case a higher-order language. Such a target program must be subsequently compiled to be executed on a physical machine. But what we want is a directly executable program. In this section we propose to solve this discrepancy by using byte-code combinators in the course of translating the semantics of a programming language into a definitional interpreter. We support our proposal by an application to the development of a small imperative language.

4.4.1 An imperative language

In an imperative program, the programmer describes in a step-by-step manner how he wants the computer to behave. An imperative program is thus very similar to a traditional cooking recipe or to assembling instructions, which may be one reason for the popularity of imperative languages. Another reason could be that the core syntactic category of imperative languages, the statement, can be executed without the need for a stack. This obviously makes it easier (for humans) to trace the execution of programs.

$$\begin{array}{rcl}
\langle\text{pgm}\rangle & ::= & \texttt{input(}x\texttt{);}\ \langle\text{block}\rangle \\
\langle\text{block}\rangle & ::= & \texttt{var}\ x\ \texttt{=}\ \langle\text{exp}\rangle\texttt{;}\ \langle\text{block}\rangle \\
 & | & \langle\text{stm}\rangle\texttt{;}\ \texttt{output(}\langle\text{exp}\rangle\texttt{);} \\
\langle\text{stm}\rangle & ::= & \texttt{skip} \\
 & | & x\ \texttt{:=}\ \langle\text{exp}\rangle \\
 & | & \langle\text{stm}\rangle\texttt{;}\ \langle\text{stm}\rangle \\
 & | & \texttt{if}\ \langle\text{exp}\rangle\ \texttt{then}\ \langle\text{stm}\rangle\ \texttt{else}\ \langle\text{stm}\rangle \\
 & | & \texttt{while}\ \langle\text{exp}\rangle\ \texttt{do}\ \langle\text{stm}\rangle \\
\langle\text{exp}\rangle & ::= & i \mid x \mid \langle\text{exp}\rangle\ \texttt{-}\ \langle\text{exp}\rangle \mid \langle\text{exp}\rangle\ \texttt{*}\ \langle\text{exp}\rangle \mid \langle\text{exp}\rangle\ \texttt{<}\ \langle\text{exp}\rangle
\end{array}$$

Figure 4.1: Syntax of an imperative language

Syntax

We consider a small imperative language for computing integer results from integer inputs. The syntax of the language is given by the grammar in Figure 4.1. There are four syntactic categories: Expressions, statements, blocks, and complete programs. An expression is either a constant, a variable, or a primitive operation applied to some sub-expressions. A statement is either empty, an assignment, a sequencing of two expressions, an if-statement, or a while loop. A block, which only occurs as the outermost component of a program, lists bindings for global variables. Its body consists of a statement and an expression. Finally, a program consists of a formal input parameter and a block. A valid program is one where the variables that occur in the body are declared in the surrounding block.

Example 4 The factorial program **Fac** looks as follows.

```
input(n);
var x = 1;

while (0 < n) do (
    x := x * n;
    n := n - 1
);
output(x);
```

Semantics

The intended behavior of expressions and statements is straightforward. A program reads an integer and assigns it to the input parameter. It then assign the values of the top-level bound expressions to the variables and executes the body. The value of the final output statement is also the value of the entire program. An expression in a top-level binding may

$$
\begin{aligned}
\mathcal{P}[\![\texttt{input(}x\texttt{); }b]\!] &= \lambda v.\, \mathcal{B}[\![b]\!][x \mapsto v] \\[1ex]
\mathcal{B}[\![\texttt{var }x\texttt{ = }e\texttt{; }b]\!] &= \lambda\sigma.\, \mathcal{B}[\![b]\!]\sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma] \\
\mathcal{B}[\![S\texttt{; output(}e\texttt{);}]\!] &= \lambda\sigma.\, \textbf{let}\,\sigma' = \mathcal{S}[\![s]\!]\sigma\,\textbf{in}\,\textbf{up}(\mathcal{E}[\![e]\!]\sigma') \\[1ex]
\mathcal{S}[\![\texttt{skip}]\!] &= \lambda\sigma.\, \textbf{up}(\sigma) \\
\mathcal{S}[\![x\texttt{ := }e]\!] &= \lambda\sigma.\, \textbf{up}(\sigma[x \mapsto \mathcal{E}[\![e]\!]\sigma]) \\
\mathcal{S}[\![s_1\texttt{; }s_2]\!] &= \lambda\sigma.\, \textbf{let}\,\sigma' = \mathcal{S}[\![s_1]\!]\sigma\,\textbf{in}\,\mathcal{S}[\![s_2]\!]\sigma' \\
\mathcal{S}[\![\texttt{if }e\texttt{ then }s_1\texttt{ else }s_2]\!] &= \lambda\sigma.\, \begin{cases} \mathcal{S}[\![s_1]\!]\sigma, & \text{if } \mathcal{E}[\![e]\!]\sigma = 1 \\ \mathcal{S}[\![s_2]\!]\sigma, & \text{otherwise} \end{cases} \\
\mathcal{S}[\![\texttt{while }e\texttt{ do }s]\!] &= \textstyle\bigsqcup_{n\in\omega} \Phi^n(\bot) \\
&\ \text{where } \Phi = \lambda f\sigma.\, \begin{cases} \textbf{up}(\sigma), & \text{if } \mathcal{E}[\![e]\!]\sigma = 0 \\ f^*(\mathcal{S}[\![s]\!]\sigma), & \text{otherwise} \end{cases} \\[1ex]
\mathcal{E}[\![i]\!] &= \lambda\sigma.\, i \\
\mathcal{E}[\![x]\!] &= \lambda\sigma.\, \sigma(x) \\
\mathcal{E}[\![e_1\texttt{ - }e_2]\!] &= \lambda\sigma.\, \mathcal{E}[\![e_1]\!]\sigma - \mathcal{E}[\![e_2]\!]\sigma \\
\mathcal{E}[\![e_1\texttt{ * }e_2]\!] &= \lambda\sigma.\, \mathcal{E}[\![e_1]\!]\sigma \times \mathcal{E}[\![e_2]\!]\sigma \\
\mathcal{E}[\![e_1\texttt{ < }e_2]\!] &= \lambda\sigma.\, \begin{cases} 1, & \text{if } \mathcal{E}[\![e_1]\!]\sigma < \mathcal{E}[\![e_2]\!]\sigma \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.2: Valuation functions for an imperative language

use the bound variables from earlier bindings. Both the statements in the body of the program and the final output statement may use all the bound variables. The purpose of the semantics is to formalize these verbal explanations of the behavior of programs.

Both the syntax of our mathematical meta language as well as the semantics that we present are similar to what one might find in standard text-books on denotational semantics [106, 123, 144]. A state, $\mathbf{\Sigma} \in \mathbf{Var} \to \mathbf{Z}$, is a mapping from variables to integers. Expressions, whose evaluation does not diverge or have any other (side) effects, are simply modeled by functions from states to the integers. Statements, whose evaluation may diverge and may have (side) effect, are traditionally modeled by partial functions between states. We shall instead model statements as continuous functions whose range is a lifted domain $A_\bot = \{\bot\} \cup \{\textbf{up}(a) \mid a \in A\}$. We write $f^* \in A_\bot \to B_\bot$ for the strict extension of $f \in A \to B_\bot$, i.e., $f(\bot) = \bot$ and $f(\textbf{up}(a)) = f(a)$, and abbreviate $(\lambda x.\, e)^*(v)$ as $\textbf{let}\, x = v\, \textbf{in}\, e$. Blocks are modeled by partial functions from states to integers. Finally, programs are partial functions between integers. To summarize, for any expression e, statement s, block b, and program p,

$$\mathcal{E}[\![e]\!] \in \mathbf{\Sigma} \to \mathbf{Z}, \quad \mathcal{S}[\![s]\!] \in \mathbf{\Sigma} \to \mathbf{\Sigma}_\bot, \quad \mathcal{B}[\![b]\!] \in \mathbf{\Sigma} \to \mathbf{Z}_\bot, \quad \mathcal{P}[\![p]\!] \in \mathbf{Z} \to \mathbf{Z}_\bot$$

```
type ide   = string

type exp   = INT    of int
           | VAR    of ide
           | SUB    of exp * exp
           | MUL    of exp * exp
           | LESS   of exp * exp

type stm   = SKIP
           | SEQ    of stm * stm
           | ASSIGN of ide * exp
           | IF     of exp * stm * stm
           | WHILE  of exp * stm

type block = BIND   of ide * exp * block
           | BODY   of stm * exp

type pgm   = INPUT  of ide * block
```

Figure 4.3: Abstract syntax of an imperative language

The valuation functions for the imperative language are displayed in Figure 4.2.

Example 5 The semantics of the factorial program is as follows.

$$\mathcal{P}[\![\textbf{Fac}]\!] = \lambda v.\,\textbf{let}\,\sigma = \Big(\bigsqcup_{n\in\omega} \Phi^n(\bot)\Big)[x \mapsto 1, n \mapsto v]\;\textbf{in}\;\textbf{up}(\sigma(x))$$

where

$$\Phi = \lambda f\sigma.\begin{cases}\textbf{up}(\sigma), & \text{if } \sigma(n) \le 0\\ f(\sigma[n \mapsto \sigma(n)-1, x \mapsto \sigma(x)\times\sigma(n)]), & \text{otherwise}\end{cases}$$

For example, $\mathcal{P}[\![\textbf{Fac}]\!]4 = \textbf{up}(24)$.

4.4.2 A definitional interpreter for imperative programs

The implementation of the syntax and semantics of the imperative language into OCaml provides an abstract syntax of programs and a definitional interpreter for evaluating program given by their abstract syntax. (We shall not deal with the problem of parsing the concrete syntax of programs.) The abstract syntax of programs, blocks, statements, and expressions is given in Figure 4.3.

Example 6 The factorial program is represented as the following piece of abstract syntax.

```
let fact =
  INPUT ("n",
    BIND ("x", INT 1,
let rec fix f x = f (fix f) x

let rec e_exp e st =
  match e with
    INT(i)        → i
  | VAR(x)        → lookup st x
  | SUB(e1, e2)   → e_exp e1 st  -  e_exp e2 st
  | MUL(e1, e2)   → e_exp e1 st  *  e_exp e2 st
  | LESS(e1, e2)  →
      if e_exp e1 st  <  e_exp e2 st then 1 else 0

let rec e_stm s =
  match s with
    SKIP            → fun st → st
  | ASSIGN(x, e)    → fun st → update st x (e_exp e st)
  | SEQ(s1, s2)     → fun st → e_stm s2 (e_stm s1 st)
  | IF(e, s1, s2)   → fun st →
      if e_exp e st  =  1 then e_stm s1 st else e_stm s2 st
  | WHILE(e, s)     →
      fix (fun f st →
             if e_exp e st  =  0 then st else f (e_stm s st))

let rec e_block b st =
  match b with
    BIND(x, e, b) → e_block b (update st x (e_exp e st))
  | BODY(s, e)    → e_exp e (e_stm s st)

let start (INPUT(x, b)) v = e_block b (init x v)
```

Figure 4.4: Direct-style interpreter for an imperative language

```
      BODY
        (WHILE (LESS (INT 0, VAR "n"),
           SEQ (ASSIGN ("x", MUL (VAR "x", VAR "n")),
                ASSIGN ("n", SUB (VAR "n", INT 1)))),
         VAR "x")))
```

The definitional interpreter, shown in Figure 4.4, is an almost direct transcription of the denotational semantics into OCaml. We have taken precautions in mapping lifted domains into OCaml types. Fortunately, we have arranged the semantic domains to match OCaml's types. Note, however, that the domain associated with expressions guarantees that the evaluation of expressions terminate. The type of the definitional interpreter cannot make such guarantee. Instead, such a result can be shown manually by observing, e.g., that the evaluation function for expressions can be defined by induction instead of recursion.

Example 7 Evaluating `start fact 4` in OCaml yields the integer 24.

4.4.3 A definitional compiler for imperative programs

The first Futamura projection states that specializing an interpreter with respect to a source program yields, if specialization terminates, an equivalent program in the implementation language of the interpreter [65]. Even if the implementation language is a higher-level language, as opposed to, e.g., machine code, it is often the case that the translation produces a more efficient version of the source program. In particular, an efficient partial evaluator will remove the interpretive overhead involved in processing the abstract syntax of the source program. (We come back to partial evaluation in Section 4.5.)

The second Futamura projection states that specializing a partial evaluator with respect to an interpreter yields, if specialization terminates, a compiler from the interpreted language to the implementation language of the interpreter (which must coincide with the source language of the partial evaluator). In the general case of specializing a partial evaluator with respect to a program, the result is also called a "generating extension" of the program. Ershov, who coined the term "generating extension", used them derive a compiler from an interpretational semantics for an imperative language [58]. Ershov's generating extension was not obtained by specializing a self-applicable partial evaluator. Instead, he defined a function mapping a program into its generating extension. By the third Futamura projection, such a compiler generator can also be achieved by partial evaluation.

We shall implement a range of compilers for the imperative language. We start out with a handwritten generating extension for the definitional interpreter. This first definitional compiler is obtained, as usual in handwriting generating extensions, by first binding-time annotating the interpreter (in this case with respect to a static source program) and then translating the result into a two-level program [24, 83]. Program parts annotated as "static" are evaluated during compile-time while program parts annotated as "dynamic" are re-built in the residual target program. Because we use byte-code combinators, the residual program parts are re-built (almost) directly as OCaml byte code. We shall use the following code-generating versions of the environment manipulating functions and fixed-point operator.

- `mklookup : exp → string → exp`

  Constructs the byte-code combinator for a call `lookup` $st\ x$ given a byte-code combinator for st and a string x.

  ```
  let mklookup st x =
    mkcapp (mkglob "lookup") [st; mkstr x]
  ```

- `mkupdate : exp → string → exp → exp`

  Constructs the byte-code combinator for a call `update` $st\ x\ v$ given byte-code combinators for st and v and a string x.

```
let rec c_exp e =
  mklam (fun st →
    match e with
      INT(i)        → mkint i
    | VAR(x)        → mklookup st x
    | SUB(e1, e2)   →
      mksub (mkapp (c_exp e1) st) (mkapp (c_exp e2) st)
    | MUL(e1, e2)   →
      mkmul (mkapp (c_exp e1) st) (mkapp (c_exp e2) st)
    | LESS(e1, e2)  →
      mkif (mklss (mkapp (c_exp e1) st) (mkapp (c_exp e2) st))
        (mkint 1)
        (mkint 0))

let rec c_stm s =
  match s with
    SKIP            → mklam (fun st → st)
  | ASSIGN(x, e)    → mklam (fun st →
      mkupdate st x (mkapp (c_exp e) st))
  | SEQ(s1, s2)     → mklam (fun st →
      mkapp (c_stm s2) (mkapp (c_stm s1) st))
  | IF(e, s1, s2)   → mklam (fun st →
      mkif (mkeqint (mkapp (c_exp e) st) (mkint 1))
        (mkapp (c_stm s1) st)
        (mkapp (c_stm s2) st))
  | WHILE(e, s)     →
      mkfix (mkclam 2 (fun [f; st] →
        mkif (mkeqint (mkapp (c_exp e) st) (mkint 0))
          st
          (mkapp f (mkapp (c_stm s) st))))

let rec c_block b =
  mklam (fun st →
    match b with
      BIND(x, e, b)  →
        mkapp (c_block b) (mkupdate st x (mkapp (c_exp e) st))
    | BODY(s, e)      → mkapp (c_exp e) (mkapp (c_stm s) st))

let start (INPUT(x, b)) =
  mklam (fun v → mkapp (c_block b) (mkinit x v))
```

Figure 4.5: Direct-style compiler for an imperative language

```
let mkupdate st x v =
  mkcapp (mkglob "update") [st; mkstr x; v]
```

- `mkinit : string → exp → exp`

  Constructs the byte-code combinator for a call `init` x v given a byte-code combinator for v and a string x.

  ```
  let mkinit x v =
     mkcapp (mkglob "init") [mkstr x; v]
  ```

- `mkfix : exp → unit → exp`

  Constructs the byte-code combinator for a call `fix` f given a byte-code combinator for f.

  ```
  let mkfix f =
    mkapp (mkglob "fix") f
  ```

Residual programs map states to states. The operations that are re-built in the residual target program are those involving integer operations (i.e., subtraction, multiplication, and comparison), operations on states (i.e., lookups, updates, and initialization), and recursion (i.e., the fixed-point operator). Thus, the first compiler we consider, shown in Figure 4.5, corresponds to a binding-time separation where all state-transforming functions are annotated as dynamic. Each call to one of the semantic valuations will generate a residual call and each statement will generate a residual λ-abstraction. As a result, residual programs contain many “administrative redexes” that do not correspond to redexes in the source program.

Example 8 Compiling the program

```
input(n); skip; output(2 * n);
```

using the compiler in Figure 4.5 yields the byte-code instructions corresponding to the following OCaml expression.

```
fun v →
  (fun st0 →
      (fun st1 →
         (((fun st4 → 2) st1) *
          ((fun st3 → lookup  st3 "n") st1)))
      ((fun st2 → st2) st0))
  (init "n" v)
```

This expression contains five administrative β-redexes.

In the following two sections we consider two compilers that do not generate administrative redexes.

```
let rec c_exp e st = ...

let generalize k_sta f =
  mklet (mklam k_sta)
    (fun k_dyn → f (mkapp k_dyn))

let rec c_stm s k st =
  match s with
    SKIP           → k st
  | ASSIGN(x, e)   → mklet (mkupdate st x (c_exp e st)) k
  | SEQ(s1, s2)    → c_stm s1 (c_stm s2 k) st
  | IF(e, s1, s2)  →
      generalize k
        (fun k →
          mkif (mkeqint (c_exp e st) (mkint 1))
            (c_stm s1 k st)
            (c_stm s2 k st))
  | WHILE(e, s)    →
      mkapp (mkfix (mkclam 2 (fun [f; st] →
        mkif (mkeqint (c_exp e st) (mkint 0))
          (k st)
          (c_stm s (mkapp f) st))))
        st

let rec c_block b st =
  match b with
    BIND(x, e, b) →
      mklet (mkupdate st x (c_exp e st)) (c_block b)
  | BODY(s, e)    → c_stm s (fun st → c_exp e st) st

let start (INPUT(x, b)) =
  mklam (fun v →
    mklet (mkinit x v) (c_block b))
```

Figure 4.6: Continuation-passing style compiler for an imperative language

4.4.4 An optimized compiler for imperative programs

The second compiler uses continuations to translate statements into residual byte-code instructions. Continuations are (static) functions from (dynamic) states to (dynamic) integers. They correspond to static state-transformers between dynamic states. (The direct-style compilers produced dynamic state-transformers between dynamic states.) The use of continuations enables the generation of residual programs where intermediate states are bound in let expressions and where let-expressions are flattened. The translation of expressions is similar to the direct-style compiler. The compiler is shown in Figure 4.6.

Example 9 Using the continuation-passing style compiler to compile the program

```
input(n); skip; output(2 * n);
```

from Example 8 yields the byte-code instructions

```
        closure L1, 0          push
        return 1               acc 1
L1:     acc 0                  push
        push                   getglobal lookup/1860g
        const "n"              apply 2
        push                   push
        getglobal init/1868g   const 2
        apply 2                mulint
        push                   return 2
        const "n"
```

which corresponds to the following OCaml expression.

```
fun v → let st = init "n" v in 2 * (lookup st "n")
```

The residual program does not contain any administrative redexes.

The program in Figure 4.6 contains a “generalization” in the case for if-statements. To avoid code duplication, the continuation, which is passed to both branches of the if-statement, must be inlined in the residual program just once. The compiler first binds the continuation in a residual let-expression and then passes the bound variable to the two branches of the if-statement. Without this coercion of binding-times, the continuation may be applied in both branches of the if-statement thus inlining the code for the continuation twice.

Example 10 The following program performs one multiplication after having made a test.

```
input(n);
if n then n := 1 else n := 2;
output(n * 10);
```

The continuation-passing style compiler in Figure 4.6 translates it into byte-code instructions that correspond to the following expression. The residual program contains one multiplication.

```
fun v →
  let st0 = init "n" v in
  let k = fun st3 → lookup st3 "n" * 10 in
  if lookup st0 "n" = 1 then
    let st2 = update st0 "n" 1 in k st2
  else
    let st1 = update st0 "n" 2 in k st1
```

Compiling this program without the generalization yields the following program, which contains two multiplications.

```
fun v →
  let st0 = init "n" v  in
  if lookup st0 "n" = 1 then
    let st2 = update st0 "n" 1 in lookup st2 "n" * 10
  else
    let st1 = update st0 "n" 2 in lookup st1 "n" * 10
```

The direct-style compiler in Figure 4.5 does not duplicate code.

4.4.5 A native compiler for imperative programs

The third compiler is based on the observation that the small imperative language that we consider can be mapped directly into OCaml byte code. The most prominent features of the imperative language are its store and its sequential evaluation of statements. Both are easily modeled by OCaml's mutable cells and strict evaluation. The third compiler, shown in Figure 4.7, is therefore in direct style and uses the native OCaml store to model the store of while-programs. Instead of passing a store, this compiler passes an environment mapping variables to their run-time location. Locations are conveniently represented by residual temporary variables stored on the stack. Consequently, there are no occurrences of variable names in the residual program.

Example 11 Using the native compiler to compile the program

```
input(n); skip; output(2 * n);
```

from Example 8 yields the byte-code instructions

```
        closure L1, 0      |      acc 0
        return 1           |      getfield 0
L1:     acc 0              |      push
        makeblock 1, 0     |      const 2
        push               |      mulint
        const 0a           |      return 2
```

which correspond to the following OCaml expression.

```
fun v → let n = ref v in (); 2 * !n
```

4.4.6 Benchmarks

To measure the effect of using the three compilers we use the factorial program, `fact`, from Example 6 and four programs, $\texttt{mat}_n$ for $n = 1 \dots 4$, that multiplies two $n \times n$ matrices.

```
let mkfix f () =
  mkcapp (mkglob "fix") [f; mkunit]

let rec c_exp e env =
  match e with
    INT(i)       → mkint i
  | VAR(x)       → mkget (lookup env x)
  | SUB(e1, e2)  → mksub (c_exp e1 env) (c_exp e2 env)
  | MUL(e1, e2)  → mkmul (c_exp e1 env) (c_exp e2 env)
  | LESS(e1, e2) →
      mkif (mklss (c_exp e1 env) (c_exp e2 env))
        (mkint 1)
        (mkint 0)

let rec c_stm s env =
  match s with
    SKIP           → mkunit
  | ASSIGN(x, e)   → mkset (lookup env x) (c_exp e env)
  | SEQ(s1, s2)    → mkseq (c_stm s1 env) (c_stm s2 env)
  | IF(e, s1, s2)  →
      mkif (mkeqint (c_exp e env) (mkint 1))
        (c_stm s1 env)
        (c_stm s2 env)
  | WHILE(e, s)    →
      mkfix (mkclam 2 (fun [f; _] →
        mkif (mkeqint (c_exp e env) (mkint 0))
          mkunit
          (mkseq
             (c_stm s env)
             (mkapp f mkunit))))
        ()

let rec c_block b env =
  match b with
    BIND(x, e, b) →
      mklet (mkref (c_exp e env))
        (fun c → c_block b ((x, c) :: env))
  | BODY(s, e)    → mkseq (c_stm s env) (c_exp e env)

let start (INPUT(x, b)) =
  mklam (fun v →
    mklet (mkref v)
      (fun c → c_block b [(x, c)]))
```

Figure 4.7: Native compiler for an imperative language

The matrix-multiplication programs statically represents the indices of three matrices and contains an unfolded version of naive matrix-multiplication. The sizes of these program are proportional to n^3. The following measures were performed on an IBM ThinkPad 600 equipped with a 266MHz Pentium II and with 96 Mb of RAM running RedHat Linux 2.2.1. We have not measured the space usage.

We use one interpreter and three compilers.

INTP: The definitional interpreter from Figure 4.4.

DS-COMP: The first definitional compiler from Figure 4.5 corresponding to a naive binding-time analysis of the interpreter INTP.

CPS-COMP: The second compiler from Figure 4.6 corresponding to an improved binding-time analysis of the interpreter INTP.

NATIVE-COMP: The third compiler from Figure 4.7 that directly maps imperative programs into OCaml.

The times spent compiling the example imperative programs using the three compilers are shown in Figure 4.8. The times spent running the residual program and the time spent applying the interpreter are shown in Figure 4.9. We make the following observations about compiling imperative programs into byte code.

- On the average, CPS-COMP is 5.09 times faster than DS-COMP.
- On the average, NATIVE-COMP is 10.09 times faster than the DS-COMP and 2.42 times faster than the CPS-COMP.
- On the average, for DS-COMP, writing byte-code combinators to memory accounts for 86.14% of the total time spent compiling.
- On the average, for CPS-COMP, writing byte-code combinators to memory accounts for 80.11% of the total time spent compiling.
- On the average, for NATIVE-COMP, writing byte-code combinators to memory accounts for 57.13% of the total time spent compiling.

We make the following observations about running the residual program.

- On the average, the residual programs produced by DS-COMP are 1.11 times faster than running the interpreter INTP.
- On the average, the residual programs produced by CPS-COMP are 1.29 times faster than running the interpreter INTP.
- On the average, the residual programs produced by NATIVE-COMP are 17.93 times faster than running the interpreter INTP.

Compile time (ms.)		Source program				
		`fact`	`mat1`	`mat2`	`mat3`	`mat4`
DS-COMP	Generate	2.61	0.31	1.44	4.53	11.36
	Write	4.64	1.96	13.05	69.67	309.30
	Total	7.25	2.26	14.49	74.20	320.67
CPS-COMP	Generate	0.32	0.23	0.98	2.64	5.83
	Write	1.30	1.02	3.46	9.95	26.08
	Total	1.63	1.24	4.45	12.58	31.92
NATIVE-COMP	Generate	0.16	0.11	0.64	3.51	14.78
	Write	0.71	0.22	0.84	2.88	8.73
	Total	0.87	0.32	1.48	6.39	23.51

Times are in milli-seconds (1/1000 of a second) and are averaged over 100 iterations.

The table shows both the times for generating byte-code combinators (Generate), the time for writing them to memory (Write), and the total time spent compiling (Total).

Figure 4.8: Semantics-directed compilation of imperative programs

Run time (ms.)	Source program				
	`fact`	`mat1`	`mat2`	`mat3`	`mat4`
INTP	0.1747	0.0192	0.2090	1.3026	5.1944
DS-COMP	0.1625	0.0165	0.1899	1.1827	4.6515
CPS-COMP	0.1267	0.0125	0.1704	1.1220	4.5024
NATIVE-COMP	0.0239	0.0019	0.0051	0.1225	0.2519

Times are in milli-seconds (1/1000 of a second) and are averaged over 1000 iterations.

The input to the program `fact` was 10.

The input to the programs $\texttt{mat}_n$ was 0. These programs do not use their input.

Figure 4.9: Running compiled imperative programs

We conclude that the cost of avoiding generating many byte-code combinators pays off for the continuation-passing style compiler. The native compiler processes environments at compile time. This processing exercises a cost at compile time but the residual program are correspondingly faster.

4.5 Run-time specialization

In the previous section, we have seen examples of compilers generating OCaml byte-code from imperative programs. Each compiler was a handwritten generating extension of an interpreter for imperative program. In this section we shall illustrate how partial evaluation, a technique for specializing a program with respect to parts of its input, can be applied to achieve the effect of compilation.

4.5.1 Partial evaluation: What

Partial evaluation is an approach to program specialization. Given a partial evaluator PE (implemented in a language L_1) for a source language L_2, a two-argument L_2-program p, and an static value s, applying PE to p and s yields a specialized version of p with respect to s (if partial evaluation terminates). Applying the specialized program p_s to a dynamic value d gives the same result as applying the original program p to both s and d. If we let $[\![\cdot]\!]_L$ denote the partial valuation function for L-programs then the correctness criterion for PE is given by the following *mix equation* [83].

$$\begin{array}{rcll} [\![\mathsf{PE}]\!]_{L_1}(p,s) & = & p_s & \text{where} \\ [\![p_s]\!]_{L_2}(d) & = & [\![p]\!]_{L_2}(s,d) & \end{array}$$

The motivation for partial evaluation is efficiency: Running p_s on input d is likely to be faster and require less memory than running p on inputs (s,d) since operations in p that depend only on s may have been removed from p_s.

The most fascinating applications of partial evaluation are perhaps found in the area of semantics-directed compilation and compiler generation: In the early 1970's, Yoshihiko Futamura observed the following applications of partial evaluation, which became known as the Futamura projections [65, 66].

(1) Using a partial evaluator PE, an interpreter intp, and a source program src written in the interpreted language one can compile src into the implementation language L_2 of intp by specializing the interpreter with respect to the source program.

$$\begin{array}{rcll} [\![\mathsf{PE}]\!]_{L_1}(\mathsf{intp},\mathsf{src}) & = & \mathsf{trg} & \text{where} \\ [\![\mathsf{trg}]\!]_{L_2}(\mathsf{inp}) & = & [\![\mathsf{intp}]\!]_{L_2}(\mathsf{src},\mathsf{inp}) & \end{array}$$

(2) If the partial evaluator is self-applicable, that is, if its implementation language L_1 is a subset of its source language L_2, one can generate a compiler from the interpreted

language to the implementation language of the interpreter by specializing the partial evaluator with respect to the interpreter.

$$\begin{array}{rcll} [\![\mathsf{PE}]\!]_{L_1}(\mathsf{PE}, \mathsf{intp}) & = & \mathsf{comp} & \text{where} \\ [\![\mathsf{comp}]\!]_{L_1}(\mathsf{src}) & = & [\![\mathsf{PE}]\!]_{L_1}(\mathsf{intp}, \mathsf{src}) & \end{array}$$

(3) Finally, one can generate a compiler generator by specializing the partial evaluator with respect to itself.

$$\begin{array}{rcll} [\![\mathsf{PE}]\!]_{L_1}(\mathsf{PE}, \mathsf{PE}) & = & \mathsf{cogen} & \text{where} \\ [\![\mathsf{cogen}]\!]_{L_1}(\mathsf{intp}) & = & [\![\mathsf{PE}]\!]_{L_1}(\mathsf{PE}, \mathsf{intp}) & \end{array}$$

In the previous section we already derived a compiler from an interpreter, although it was handwritten instead of obtained by self-applying a partial evaluator. Such a compiler is a generating extension of an interpreter, or equivalently, a dedicated partial evaluator for the interpreter: It takes the static input src and produces the specialized version of intp with respect to src, namely trg. As illustrated above, a generating extension for a program p can be obtained by specializing the partial evaluator with respect to p.

4.5.2 Partial evaluation: How

A partial evaluator is a non-standard interpreter that combines evaluation with code generation. Program parts that only depend on the static input are reduced by the partial evaluator while program parts that depend on the dynamic input are rebuilt in the residual program.

On-line partial evaluation

An *on-line* partial evaluator decides whether to reduce or rebuild a program part as it processes the source program and the static input. This decision is *precise* since it is based on the actual static input and therefore, on-line partial evaluation most often yields efficient residual programs [122]. Typically, however, on-line partial evaluation does not yield good results when self-applied. A compiler generated from self-application of on-line partial evaluation, $\mathsf{comp} = [\![\mathsf{PE}]\!]_{L_1}(\mathsf{PE}, \mathsf{intp})$, is too general to be efficient. The reason is that the generated compiler can not only be applied to a static source program and a dynamic input but also to a dynamic source program and a static input. The price for generality is a compiler in which few static reductions are carried out.

Off-line partial evaluation

An *off-line* partial evaluator decides whether to reduce or rebuild a program part independently of the actual static input. To this end, off-line partial evaluation is split into (1) a *binding-time analysis* stage which annotates source program parts as either static or dynamic and (2) a *specialization* stage which reads the static input and reduces or rebuilds program parts depending on whether they are static or dynamic. The specialization stage is fast since the reduce/rebuild decisions have been made beforehand. However, due to the necessarily approximate binding-time information, residual programs are usually less efficient than those obtained by on-line partial evaluation. Off-line partial evaluation was introduced by Jones's group in the mid 1980's to make self-applicable partial evaluation feasible in practice [84].

Off-line partial evaluation gives rise to a notion of *two-stage programs*: The static parts of a program are executed at specialization time while the dynamic parts are rebuilt in the specialized program and executed at run time. *Staged languages* provide frameworks where binding times are explicitly represented in the syntax of programs. In the following grammar for a higher-order language with integers, underlined terms are "dynamic". (Here, function application is denoted by the infix @. A program is a multi-argument λ-abstraction.)

$$\begin{aligned} p &::= \lambda(x_1,\ldots,x_n).\, e \\ e &::= x \mid i \mid \lambda x.\, e \mid e_1 \mathbin{@} e_2 \mid \mathbf{let}\; x = e_1 \;\mathbf{in}\; e_2 \mid e_1 + e_2 \\ &\quad\;\; \mid \underline{i} \mid \underline{\lambda} x.\, e \mid e_1 \mathbin{\underline{@}} e_2 \mid \underline{\mathbf{let}}\; x = e_1 \;\underline{\mathbf{in}}\; e_2 \mid e_1 \mathbin{\underline{+}} e_2 \end{aligned}$$

Staged languages provide an interpretation of binding-annotations. Specializing a binding-time annotated terms amounts to *reducing* the overlined construct of a term while *rebuilding* the underlined terms. It is required that binding-time annotated terms are well-annotated in the sense that static reduction "does not go wrong" and yields a completely dynamic term. This requirement can be expressed by extended type systems such as, e.g., the $\lambda^{\bigcirc}$-calculus [45] or two-level calculi [105]. A binding-time analysis must produce a well-annotated two-stage term from a source program.

Example 12 Consider a program fragment

$$\lambda(d,h).\, \mathbf{let}\; f = \lambda g.\, g\,(g\,d)\; \mathbf{in}\; (f\,(\lambda x.\, x)) + (f\,h)$$

where d and h are dynamic. Since f is applied to both a static and a dynamic argument a binding-time analysis must assign the most conservative binding-time to f, namely "dynamic". Specialization will therefore rebuild the first application of f instead of reducing it.

$$\lambda(d,h).\, \overline{\mathbf{let}}\; f = \overline{\lambda} g.\, g \mathbin{\underline{@}} (g \mathbin{\underline{@}} d)\; \overline{\mathbf{in}}\; f \mathbin{\overline{@}} (\underline{\lambda} x.\, x) \mathbin{\underline{+}} f \mathbin{\overline{@}} h$$

Specializing this binding-time annotated program gives the residual program

$$\lambda(d,h).\, \mathbf{let}\; g = \lambda x.\, x\; \mathbf{in}\; (g\,(g\,d)) + (h\,(h\,d))$$

which contains two β-redexes.

As illustrated by this example, *binding-time improvements* are often necessary to obtain good results from partial evaluation. In particular, inserting *η-expansion* in higher-order programs can coerce static functions in dynamic contexts and vice versa [39, 40]. Such binding-time coercions play a crucial role in making partial evaluation effective. For example, binding-time improvements were instrumental for Palsberg and Bondorf to obtain good results from specializing an interpreter for Action Notation using Similix [14, 15, 41].

Example 13 We η-expand the occurrence of h in the source program:

$$\lambda(d,h).\,\mathbf{let}\ f = \lambda g.\, g\,(g\,d)\ \mathbf{in}\ (f\,(\lambda x.\, x)) + (f\,(\lambda x.\, h\,x))$$

Now, the argument in the second application of f is static and a binding-time analysis can assign the binding-time “static” to f.

$$\lambda(d,h).\,\overline{\mathbf{let}}\ f = \overline{\lambda} g.\, g\,\overline{@}\,(g\,\overline{@}\,d)\ \overline{\mathbf{in}}\ (f\,\overline{@}\,(\underline{\lambda} x.\, x))\,\underline{+}\,(f\,\overline{@}\,(\lambda x.\, h\,\underline{@}\,x))$$

Statically reducing this improved binding-time annotated program gives the residual program

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

which contains no redexes.

The two stages of off-line partial evaluation operate as follows.

$$\begin{aligned} p_{\text{bta}} &= [\![bta]\!]_{L_1}\,(p) \\ p_s &= [\![spec]\!]_{L_1}\,(p_{\text{bta}}, s) \end{aligned}$$

where bta is the binding-time analyser, $spec$ is the specializer, and p_{bta} is the binding-time annotated version of p. Languages of binding-time annotated terms directly give a semantics to annotated terms. The specializer is then, in fact, an (definitional) interpreter for a language of binding-time annotated terms. The binding-time annotated program is a generating extension of the source program implemented in the language of binding times:

$$[\![p_{\text{bta}}]\!]_{\text{bta}}\,(s) = [\![spec]\!]_{L_1}\,(p_{\text{bta}}, s) = p_s$$

where $[\![\cdot]\!]_{\text{bta}}$ is the semantic valuation for the language of binding times. As illustrated by the second Futamura projection, we can also obtain a generating extension for p in the implementation language L_1 of the partial evaluator as follows.

$$\begin{aligned} p_{\text{gen}} &= [\![\mathsf{PE}]\!]_{L_1}\,(spec, p) \\ p_s &= [\![p_{\text{gen}}]\!]_{L_1}\,(s) \end{aligned}$$

Hybrid approaches to partial evaluation combine the static decisions from off-line partial evaluation with the dynamic choices from on-line partial evaluation [126, 132, 143]. Such approaches typically consists of a binding-time analysis stage, which annotates program parts as either static, dynamic, or unknown, and a specialization stage, which reduces static program parts, rebuilds dynamic parts, and dynamically decides the action to take upon unknown program parts.

Monovariance and polyvariance

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.

4.5.3 Type-directed partial evaluation

Type-directed partial evaluation is an approach to off-line, monovariant specialization for higher-order programs [30, 34]. For closed source programs, binding-time information is given by the type of the program. If a source program contains free variables, these must be separated into static and dynamic occurrences beforehand. Type-directed partial evaluation is extended with online features, such as primitive operations that probe their arguments for static reduction opportunities at specialization time [33].

Type-directed partial evaluation is stated and formalized in both a call-by-name setting [61] and a call-by-value setting [62]. Due to the unsoundness of the general β-reduction rule for call-by-value, “serious” expressions, such as applications and certain primitive operations, that may have effects must be let-bound in the residual programs. For pure, closed λ-terms, type-directed partial evaluation coincides with Berger and Schwichtenberg’s normalization by evaluation [10, 11].

Type-directed partial evaluation is based on the idea of extracting normal forms from the semantics (or compiled value) of a program: In higher-order languages with a base type exp of syntactic representations of terms, there exists, for each type τ built out of exp and function space alone, a term $\downarrow^{\tau}\colon \tau \rightarrow \mathsf{exp}$ such that for any pure closed term $e : \tau$, evaluating $\downarrow^{\tau}\ e$ either diverges or yields the text of the long $\beta\eta$-normal form of e. Such a normal form does not contain β-redexes and it is fully η-expanded with respect to its type (i.e., terms of higher type are either abstractions or they occur in function position in an application). In the call-by-value setting all applications are let-bound.

Given a closed program $p : \mathsf{s} \times \mathsf{d} \rightarrow \mathsf{r}$ and a static input $s : \mathsf{s}$, type-directed partial evaluation works as follows. The trivially specialized program $\lambda d.\, p(s, d)$ is a solution to the mix equation, albeit a sub-optimal one: For any value v, evaluating $(\lambda d.\, p(s, d))d$ yields the same result as evaluating $p(s, d)$. The result of evaluating $\downarrow^{\mathsf{d} \rightarrow \mathsf{r}}\ (\lambda d.\, p(s, d))$, on the other hand, is in normal form and is thus an efficient specialized version of p with respect to s. For programs with free variables, such as arithmetic primitives, conditionals, and fixed point operators, the input to type-directed partial evaluation is a binding-time separated program in which statically occurring variables are given a standard *evaluating interpretation* and where dynamically occurring variables are given a non-standard *residualizing interpretation*. In languages with a module system, the source program can be given by a functor parameterized over an interpretation. The same program can then be subject to both evaluation and specialization.

Type-directed partial evaluation for pure λ-terms can be characterized as the two type-indexed and inductively defined functions in Figure 4.10. It is here expressed in a staged λ-calculus where the underlined constructs are code generating primitives and where @ denotes infix application.

$$\begin{array}{llcl}
\text{(Types)} & \tau & = & \mathsf{exp} \mid \tau_1 \to \tau_2 \\
\\
\text{(Reify)} & \downarrow^{\mathsf{exp}} & = & \lambda v.\, v \\
 & \downarrow^{\tau_1 \to \tau_2} & = & \lambda v.\, \underline{\lambda} x.\, \downarrow^{\tau_2} \,@\, (v \,@\, (\uparrow_{\tau_1} \,@\, x)) \\
\\
\text{(Reflect)} & \uparrow_{\mathsf{exp}} & = & \lambda v.\, v \\
 & \uparrow_{\tau_1 \to \tau_2} & = & \lambda v.\, \lambda x.\, \uparrow_{\tau_2} \,@\, (v \,\underline{@}\, (\downarrow^{\tau_1} \,@\, x)) \\
\\
\text{(Tdpe)} & \mathsf{tdpe}_{\tau_1 \to \tau_2 \to \tau_3} & = & \lambda p : \tau_1 \to \tau_2 \to \tau_3.\, \lambda s : \tau_1.\, \downarrow^{\tau_2 \to \tau_3} \,@\, (\lambda d.\, p\, s\, d)
\end{array}$$

Figure 4.10: Type-directed partial evaluation for pure λ-terms

Example 14 The closed term $M = \lambda x.\, (\lambda y.\, y)x$ of type $\mathsf{exp} \to \mathsf{exp}$ contains a β-redex. Statically reducing $\downarrow^{\mathsf{exp} \to \mathsf{exp}} M$ yields the normal form of M, $\lambda x.\, x$.

Example 15 (Running example.) Consider function composition as defined by $C = \lambda f.\, \lambda g.\, \lambda x.\, f(g\, x)$. We can specialize the term $M = \lambda g.\, \lambda f.\, g\, f\, f$ with respect to $g = C$ by statically reducing $\downarrow^{(\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp} \to \mathsf{exp}} (M\, C)$. The result is the specialized term $\lambda f.\, \lambda x.\, f\,(f\, x)$.

In practice, type-directed partial evaluation is usually not given by a class of terms $(\downarrow^{\tau} \colon \tau \to \mathsf{exp})_{\tau}$ but rather as a function *reify* mapping a representation $\ulcorner \tau \urcorner : \mathsf{type}$ of a type τ into a function of type $\tau \to \mathsf{exp}$. Such a dependently typed function can be implemented in a Hindley-Milner type system by a Church-encoded representation of types [119, 145].

Partial evaluation was originally designed to specialize source programs to their static inputs at compile time. The result was the text of the residual program which was compiled by a stand-alone compiler to yield an executable program. During the 1990's, it was observed that a number of programs can benefit from partial evaluation but that the static input to these systems are not available at compile time. Instead the static data is computed by one part of the system and passed to the program part that would benefit from specialization. In most situations, it is immediately clear that just applying a standard partial evaluator to the program part under consideration and then applying a standard compiler to produce the binary executable is infeasible in practice. A number of run-time specializers were introduced that supported efficient specialization and subsequent compilation at compile time. (See Section 4.6 for a discussion of related work.)

All existing implementation of type-directed partial evaluation model the dynamic constructs by operations that generate text. Instead, we shall use code templates to directly generate byte code. We parameterize the implementation of type-directed partial evaluation over a module of constructors of dynamic λ-terms satisfying the signature in Figure 4.11. The core of the implementation is shown in Figure 4.12. It defines two functions for constructing representations of types. Such a representation consists of two functions that inductively

```
module type RESIDUAL = sig
  val lam : (exp → exp) → exp
  val app : exp → exp → exp
end
```

Figure 4.11: Two-level code-generating primitives

```
module Tdpe(R : RESIDUAL) = struct
  type 'a rr = RR of ('a → exp) * (exp → 'a)

  let base =
    RR ((fun e → e),
        (fun e → e))

  let func(RR(reify1, reflect1), RR(reify2, reflect2)) =
    RR ((fun v → R.lam (fun x → reify2 (v (reflect1 x)))),
        (fun e → fun x → reflect2 (R.app e (reify1 x))))

  let tdpe (RR (reify, reflect)) e = reify e
  let tdpe'(RR (reify, reflect)) e = reflect e
end
```

Figure 4.12: Type-directed partial evaluation in OCaml

descends the represented type. See Chapter 3 for a discussion of this technique.

In the following two sections we instantiate type-directed partial evaluation to a call-by-name setting and to a call-by-value setting by providing two implementations of residual terms.

Call-by-name type-directed partial evaluation

By directly plugging the module of byte-code combinators into type-directed partial evaluation we obtain an implementation of type-directed partial evaluation for pure λ-terms in OCaml. The module is shown in Figure 4.13.

Example 16 (Example continued.) In OCaml we can specialize the pure term $\lambda g.\, \lambda f.\, g\, f\, f$ with respect to $\lambda f.\, \lambda g.\, \lambda x.\, f(g\, x)$ by running

```
tdpe (func(func(base,base),func(base,base)))
     ((fun g f → g f f) (fun f g x → f(g x)))
```

The result is the following byte-code instructions.

```
module CBN = struct
  let lam = mklam
  let app = mkapp
end

module TdpeCBN = Tdpe(CBN)
```

Figure 4.13: Two-level code-generating primitives for call-by-name

```
        closure L1, 0          push
        return 1               envacc 1
L1:     acc 0                  apply 1
        closure L2, 1          push
        return 1               envacc 1
L2:     acc 0                  appterm 1, 2
```

These instructions correspond to the term

```
fun f → fun x → f (f x)
```

of type `(exp → exp) → exp → exp`.

Call-by-value type-directed partial evaluation

Type-directed partial evaluation as presented above is unsound in call-by-value settings, such as OCaml, when functions that may diverge or that have (other) side effects are involved. For example, using call-by-name type-directed partial evaluation to normalize the term

$$\lambda f.\, \lambda x.\, (\lambda y.\, x)(f\, x)$$

yields $\lambda f.\, \lambda x.\, x$. This residual term is generally not equivalent to the original term in a call-by-name setting. (Consider, for example, a situation where f denotes a function that diverges.) To remedy this, we can insert a residual let expression to bind the possible effect-full expression $f\, x$ as in $\lambda f.\, \lambda x.\, \textbf{let}\, y = f\, x\, \textbf{in}\, x$.

This kind of let-insertion can be implemented in type-directed partial evaluation if the host language supports first-class continuations or mutable state [34, 62]. Both solutions amounts to recording a list of residual applications for each residual λ-abstraction. A stack of such lists is maintained in order to correctly gather let-bindings for nested residual λ-abstractions. The solution using first-class continuations also provides the means to specialize dynamic disjoint sums. The standard distribution of OCaml does not support first-class continuations so we shall use the state-based solution.

Residual terms, given in Figure 4.14, adds let-expressions to the terms used in the call-by-name setting. Residual bindings are accumulated in the global variable `hook` in reverse order of occurrence. The function `reset` overwrites the current list of accumulated bindings

```
module CBV = struct

  let hook = ref []

  let reset c = let temp = !hook in hook := c; temp

  let rec wrap = function
      ([],         e)  → e
    | ((x, v) :: bs, e)  → wrap(bs, mklet1 x v e)

  let bind e =
    let r = gensym () in
    hook := (r, e)  ::  !hook;
    mkvar r

  let lam f =
    let x = gensym () in
    let c = reset [] in
    let e = f (mkvar x) in
    let b = reset c in
    mklam1 x (wrap(b, e))

  let app a b = bind (mkapp a b)
end

module TdpeCBV = Tdpe(CBV)
```

Figure 4.14: Two-level code-generating primitives for call-by-value

and returns the previous value. The stack is not explicitly represented. Instead, lists of bindings are saved in temporary variables in the function `lam`. Thus, the stacking of lists is provided by the execution stack. Bindings are added to the current list by the function `bind`. Let expressions are generated by `wrap`, which unfolds a list of bindings to a chain of nested let-expressions. We rely on code templates to preserve tail-calls in let-expressions.

Example 17 (Example continued.) Specializing $M = \lambda g.\lambda f.\, g\, f\, f$ with respect to $C = \lambda f.\lambda g.\lambda x.\, f(g\, x)$ under call-by-value yields the following sequence of byte-code instructions.

```
        closure L1, 0     |       envacc 1
        return 1          |       apply 1
L1:     acc 0             |       push
        closure L2, 1     |       acc 0
        return 1          |       push
L2:     acc 0             |       envacc 1
        push              |       appterm 1, 3
```

These instructions correspond to the following term.

```
module type INTERPRETATION = sig
  type tint
  val qint : int  → tint
  val mul  : tint → tint → tint
end

module Eval : INTERPRETATION with type tint = int = struct
  type tint   = int
  let qint i  = i
  let mul x y = x * y
end

module Resi : INTERPRETATION with type tint = exp = struct
  type tint  = exp
  let qint i = mkint i
  let mul x y =
    tdpe’ (func(base, func(base, base)))
      (mkqref ["Eval"; "mul"]) x y
end

module Power(I : INTERPRETATION) = struct
  let rec power n x =
    if  n = 0  then  I.qint 1  else  I.mul x (power (n-1) x)
end

module PowerEval = Power(Eval)
module PowerResi = Power(Resi)
```

Figure 4.15: The power function in OCaml

```
fun f  →  fun x  →  let y = f x in f y
```

Note that the final call occurs at tail-position.

4.5.4 Applications

We present examples and applications of run-time code generation for type-directed partial evaluation, starting with some benchmark results.

The power function: Example of specialization

As an application of run-time code generation for type-directed partial evaluation we consider the example of the power function. The source program, shown in Figure 4.15 is structured as three modules, a standard evaluating interpretation of primitives, a non-standard residualizing interpretation of primitives, and the power function parameterized over these primitives. The residualizing interpretation builds byte-code combinators.

We can run the general power function by using the evaluating interpretation. For example, running `PowerEval.power 3 5` directly yields 125. We can also generate an intermediate specialized function first. To this end, we apply type-directed partial evaluation to generate the byte-code combinator of the specialized power function. Instantiating the combinator and running the result yields an executable function. (It must be manually cast to its type, `int → int`.) Incidentally, it is safe to specialize the power function using both type-directed partial evaluation for pure λ-terms and type-directed partial evaluation that inserts residual let-expressions.

Example 18 Specializing the power function to the static exponent 3 using call-by-name type-directed partial evaluation yields the following byte-code instructions.

```
        closure L1, 0
        return 1
L1:     const 1
        push
        acc 1
        push
        getglobal Eval/945g
        getfield 1
        apply 1
        apply 1
        push
        acc 1
        push
        getglobal Eval/945g
        getfield 1
        apply 1
        apply 1
        push
        acc 1
        push
        getglobal Eval/945g
        getfield 1
        appterm 2, 3
```

which corresponds to the expression

```
fun x → Eval.mul x (Eval.mul x (Eval.mul x 1))
```

The multiplication by 1 is not reduced such as can be done by an on-line definition of multiplication. On-line primitives for standard type-directed partial evaluation can generate specialized code when their input are known constants [33]. We cannot take this approach here since the inputs to primitives are byte-code combinators that cannot be compared. It would be possible, however, to add extra information to byte-code combinators indicating whether they represent known constants or not.

Example 19 Specializing the power function to the static exponent 3 using call-by-value type-directed partial evaluation yields the following byte-code instructions.

```
        closure L1, 0
        return 1
L1:     acc 0
        push
        getglobal Eval/1130g
        getfield 1
        apply 1
        push
        const 1
        push
        acc 1
        apply 1
        push
        acc 2
        push
        getglobal Eval/1130g
        getfield 1
        apply 1
        push
        acc 1
        push
        acc 1
        apply 1
        push
        acc 4
        push
        getglobal Eval/1130g
        getfield 1
        apply 1
        push
        acc 1
        push
        acc 1
        appterm 1, 7
```

which corresponds to the expression

```
fun x →
  let f1 = Eval.mul x in
  let v1 = f1 1 in
  let f2 = Eval.mul x in
  let v2 = f2 v1 in
  let f3 = Eval.mul x in
  f3 v2
```

All applications are let-bound except the last which occurs in tail position. Even partial applications of curried function are bound. Using the byte-code combinator for curried applications, this can be avoided by adding a special n-argument function type to type-directed partial evaluation. The technique is similar to generating function calls with several arguments in type-directed partial evaluation for Scheme [31].

The previous two examples follow the standard pattern of type-directed partial evaluation by defining the residualizing interpretation in terms of the evaluation interpretation. This means that the residual programs contain references to the fields in the module for the evaluating interpretation. A more efficient solution is to use residualizing primitives that directly generate the correct instructions in the residual program. Figure 4.16 implements such a direct residualizing interpretation.

Example 20 Specializing the power function with respect to 3 using the direct residualizing primitive (using either call-by-name or call-by-value type-directed partial evaluation) yields the following byte-code instructions.

```
module Direct : INTERPRETATION with type tint = exp = struct
  type tint  = exp
  let qint i = mkint i
  let mul x y = mkmul x y
end

module PowerDirect = Power(Direct)
```

Figure 4.16: A direct interpretation of primitives

```
        closure L1, 0        acc 1
        return 1             mulint
L1:     const 1              push
        push                 acc 1
        acc 1                mulint
        mulint               return 1
        push
```

which corresponds to the expression

```
fun x → x * (x * (x * 1))
```

The generating extension for the power function in Example 3 yields the same residual program.

The power function: Assessment

We have compared the efficiency of the different versions of the power function. The following measures were performed on an IBM ThinkPad 600 equipped with a 266MHz Pentium II and with 96 Mb of RAM running RedHat Linux 2.2.1. We consider the following programs.

POWER: The parameterized power function using the evaluating interpretation, `Eval`. This function exercises a cost of accessing the multiplication function and the lifting operation in the module of primitives and it incurs an interpretive overhead.

DIRECT-POWER: A power function where the multiplication has been inlined and which does not lift integers. This function incurs an interpretive overhead.

CBN-TDPE: The parameterized power function using the residualizing interpretation `Resi` and which is specialized using call-by-name type-directed partial evaluation.

SHORT-CBN-TDPE: The same as for CBN-TDPE except that curried applications are implemented as one efficient call instead of several. The OCaml compiler automatically improves both the interpreters in a similar way.

CBV-TDPE: The parameterized power function using the residualizing interpretation `Resi` and which is specialized using call-by-value type-directed partial evaluation. Curried applications are not optimized.

DIRECT-TDPE: The parameterized power function using the direct interpretation `Direct`.

We have measured these functions over the static values 0, 10, 100, 1000, and 2000 for the exponent. In order not to overflow for large exponents, we have used a dynamic base value of 1. The times for compiling the power function are shown in Figure 4.17. The sizes of the residual programs as functions of the exponent n are shown in Figure 4.18. The average code-generation speed is shown in Figure 4.19. The times for running the residual programs and for running the interpreters are shown in Figure 4.20. We make the following observations about specializing the power function.

- On the average, SHORT-CBN-TDPE is 1.06 times faster than CBN-TDPE.
- On the average, CBN-TDPE is 13.17 times faster than CBV-TDPE.
- On the average, DIRECT-TDPE is 3.90 times faster than CBN-TDPE, 3.65 times faster than SHORT-CBN-TDPE, and 66.37 times faster than CBV-TDPE.
- On the average, for CBN-TDPE, writing byte-code combinators to memory accounts for 61.81% of the total time spent specializing.
- On the average, for SHORT-CBN-TDPE, writing byte-code combinators to memory accounts for 66.89% of the total time spent specializing.
- On the average, for CBV-TDPE, writing byte-code combinators to memory accounts for 48.60% of the total time spent specializing.
- On the average, for DIRECT-TDPE, writing byte-code combinators to memory accounts for 81.46% of the total time spent specializing.

We also observe that the specialization times for CBV-TDPE are quadratic in the value of the exponent.[1] The reason is that looking up variables in the representation of environments in byte-code combinators is linear in the number of variables in the environment. Furthermore, the residual programs comprise a sequence of nested let expressions that all refer to the input parameter `x`. It is the repeated lookup of this variable that leads to quadratic time usage.

We make the following observations on the speed at which byte-code instructions are generated.

- On the average, SHORT-CBN-TDPE generates 1.03 times more instructions per second than CBN-TDPE.

[1]This quadratic behavior was spotted by Peter Sestoft.

Compile time (ms.)		Exponent 0	10	100	1000	2000
CBN-TDPE	Generate	0.01	0.36	3.95	50.33	108.38
	Write	0.08	0.52	4.81	57.89	120.74
	Total	0.09	0.89	8.76	108.21	229.17
SHORT-CBN-TDPE	Generate	0.01	0.30	3.00	41.61	86.51
	Write	0.08	0.58	4.91	60.24	121.59
	Total	0.09	0.88	7.91	101.85	208.11
CBV-TDPE	Generate	0.01	0.78	35.78	3462.92	—
	Write	0.08	0.88	19.31	1237.45	—
	Total	0.10	1.66	55.08	4700.04	—
DIRECT-TDPE	Generate	0.01	0.05	0.33	4.29	10.70
	Write	0.08	0.21	1.47	16.38	35.27
	Total	0.09	0.26	1.80	20.67	45.96

Times are in milli-seconds (1/1000 of a second) and are averaged over 100 iterations.

The table shows both the times for generating byte-code combinators (Generate), the time for writing them to memory (Write), and the total time spent compiling (Total).

Specializing the power function to a static exponent 2000 did not terminate within reasonable time.

Figure 4.17: Specializing the power function

Size (instructions)	Exponent 0	n
CBN	5	$7\times n+3$
SHORT-CBN-TDPE	5	$6\times n+4$
CBV-TDPE	5	$11\times n+2$
DIRECT-TDPE	5	$3\times n+5$

Number of instructions in the residual programs measured as a function of the exponent n.

Figure 4.18: Sizes of residual programs

Code-generation (instruction / ms.)		Exponent 0	10	100	1000	2000
CBN	Generate	294	205	178	139	129
	Write	40	137	146	121	116
	Total	35	82	80	65	61
SHORT-CBN-TDPE	Generate	393	214	201	144	139
	Write	48	111	123	100	99
	Total	43	73	76	59	58
CBV-TDPE	Generate	143	143	31	3	—
	Write	23	128	57	9	—
	Total	20	68	20	2	—
DIRECT-TDPE	Generate	295	713	927	701	562
	Write	65	163	208	183	170
	Total	53	133	170	145	131

Instructions generated per milli-second (1 / 1000 of a second).

The table shows both the speed at which byte-code combinators are generated (Generate), the speed at which they are written to memory (Write), and the overall speed of compiling (Total).

Figure 4.19: Code-generation speed

Run time (ms.)	Exponent 0	10	100	1000	2000
POWER	0.001	0.007	0.054	0.541	1.098
DIRECT-POWER	0.001	0.005	0.040	0.404	0.927
CBN	0.001	0.009	0.081	0.802	1.592
SHORT-CBN-TDPE	0.001	0.004	0.031	0.329	0.680
CBV-TDPE	0.001	0.009	0.088	1.031	—
DIRECT-TDPE	0.001	0.002	0.008	0.073	0.147

Times are in milli-seconds (1 / 1000 of a second) and are averaged over 10000 iterations.

Figure 4.20: Running the specialized power function

- On the average, CBN-TDPE generates 9.86 times more instructions per second than CBV-TDPE.
- On the average, DIRECT-TDPE generates 1.93 times more instructions per second than CBN-TDPE, 2.00 times more instructions per second than SHORT-CBN-TDPE, and 17.12 times more instructions per second than CBV-TDPE.

We make the following observations about running the residual program.

- On the average, the residual programs produced by CBN-TDPE are 1.34 times *slower* than running the original POWER and 1.71 times slower than running the original DIRECT-POWER
- On the average, the residual programs produced by SHORT-CBN-TDPE are 1.55 times *faster* than running the original POWER and 1.23 times slower than running the original DIRECT-POWER
- On the average, the residual programs produced by CBV-TDPE are 1.46 times *slower* than running the original POWER and 1.89 times slower than running the original DIRECT-POWER
- On the average, the residual programs produced by DIRECT-TDPE are 5.23 times *faster* than running the original POWER and 4.07 times slower than running the original DIRECT-POWER

The reason why running the residual programs produced by CBN-TDPE is slower than directly running POWER is that the OCaml compiler generates optimized curried applications for the original power function. In the programs produced by CBV-TDPE the inserted let-expressions add an extra penalty. Inserting let-expressions also exercise a high cost at specialization time. These observations confirm the results of the following two preliminary experiments with run-time code generation for type-directed partial evaluation.

A BSD packet-filter interpreter

Packet filtering is an obvious application area for run-time specialization [25, 94]. 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 ask 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 [100] 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.

A filter specified by a user-level process 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

Filter	Amortization factor		Speedup
1	125 packets	0.61 ms.	118%.
2	275 packets	4.77 ms.	65%.
3	1016 packets	12.43 ms.	48%.
4	1798 packets	24.14 ms.	58%.
5	783 packets	9.57 ms.	49%.
6	1870 packets	22.23 ms.	49%.

Times are in milli-seconds (1/1000 of a second).

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.

Figure 4.21: Relative effect of compile packet filters

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 have already applied run-time code generation to packet filtering using the Fabius system [94]. 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. It operates on the abstract syntax of filters and not on sequences of bytes as in the C implementation.

The packet-filter interpreter is applied to the six filters. 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. The compiled filters handles around 50-100% more packets per time-unit than the interpreted filters. The numbers are summarized in Figure 4.21.

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.

An amortization factor ranging between 125 and 1870 may be eminently reasonable for packet filtering. Typical packet filters handle packets counted in millions, e.g., for a TCP / IP monitor, and it is not uncommon for a host to have more than 50 million packets transit every day. However, our target programs are interpreted by OCaml's virtual machine and are therefore less efficient than running the original packet-filter interpreter written in C.

The MetaPRL term rewriter

Another application area for run-time code generation is in the proof-search engines of automated theorem provers. Higher-order logical frameworks provide an expressive foundation for reasoning about formal systems. They permit concise problem descriptions and re-use of logical models. MetaPRL is a logical programming environment that embeds OCaml in a higher-order logical framework [72]. The MetaPRL tactic prover consists of a proof editor, logic definitions, and a refiner. The logic definitions describes the language of a logic. It also contains *primitive inference rules* that define axioms and theorems of the logic, *rewrites* that define computational equivalences, and *theorems* which provide proofs for derived inference rules and axioms. Inference rules are compiled to a byte-code language. The refiner interprets the byte-code instructions during rule application and term rewriting.

Since reasoning about programs is expensive, it is crucial that proof searching is efficient. To this end, MetaPRL provides specialized implementations for the parts of a logic, such as term representations and proof-search strategies [73]. Furthermore, the set of inference rules is fixed for a particular proof search so the term rewriter can be specialized to the set of inference rules at the time a proof search is initiated. We have applied run-time code generation for type-directed partial evaluation to specialize the term rewriter of MetaPRL to a set of inference rules, directly yielding compiled byte-code programs for inference rules.

Our implementation consists of a binding-time separated version of the term rewriter, a module of evaluating primitives, and a module of residualizing primitives. The original module of the term rewriter consists of a dozen recursive functions of a total of around 500 lines. It is parameterized over specialized domain-specific representations of terms and over primitive operations on terms. The refiner modules of MetaPRL consist of roughly 5000 lines of OCaml code. Since byte-code combinators do not support generating references to parameterized modules, the binding-time separated rewriting module has been instantiated to a fixed representation of terms. The binding-time separated module is around 800 lines of OCaml code. In total, the evaluating and residualizing primitives consist of around 700 lines of code. As usual with type-directed partial evaluation [34], the binding times of the primitive operations of the term rewriter have been determined manually. (The original rewriter makes one recursive call feeding a dynamic term into a static argument. To avoid collapsing binding times, it was necessary to split the rewriter into a static part for specialization and a dynamic part for running dynamic terms. The lifting of terms into dynamic values at specialization time is performed by OCaml's marshaling library.)

The first preliminary experiments with run-time code generation for the MetaPRL term rewriter show high amortization factors. Compiling small terms does not pay for itself until after several hundreds of iterations. Furthermore, the gain in speed of using compiled terms over interpreted terms is small. There are two reasons for the little gain in speed. First, dynamic primitive operations require an extra level of indirection in the residual programs since they refer to the module of evaluating primitives instead of directly to the module where they are defined. Second, type-directed partial evaluation produces residual programs where all function calls are let-bound. Such programs are larger and less efficient than equivalent programs where only a minimal amount of calls are let-bound.

A more promising approach to specializing MetaPRL's term rewriter is to directly implement its generating extension by hand, using byte-code combinators. In the generating extension, the programmer can control the amount of let-insertions. Constructing the generating extension of a program also amounts to producing a binding-time separation of the primitive operations. We have not implemented this idea.

We draw two conclusions from the experiments with MetaPRL. First, manually determining binding times is an error-prone and tedious task. In particular, binding-time separating large programs that were not written as source programs to type-directed partial evaluation, may require several iterations until a correctly separated program is obtained. In practice, applying type-directed partial evaluation to large existing system would benefit from an automated binding-time analysis. Second, let-expressions exercise a cost for both type-directed partial evaluation and the residual programs it produces.

4.6 Related work

4.6.1 Run-time byte-code generation

Sperber and Thiemann present a run-time specializer for Scheme that generates byte code [127]. Conceptually, they have composed an existing partial evaluator for Scheme [135] with an existing Scheme compiler producing byte code [90]. For efficiency, however, their run-time specializer does not construct intermediate residual programs as text. Instead, they have obtained a set of byte-code combinators directly from the Scheme compiler by deforesting the data type of abstract syntax trees. The result is a set of byte-code combinators similar to those presented in this chapter.

Balat and Danvy have designed a run-time specializer for OCaml using type-directed partial evaluation [5]. They have composed standard type-directed partial evaluation with a compiler from normal forms into OCaml byte code. They have exploited the fact that normal forms form a subset of full OCaml to implement a fast dedicated compiler. They implement continuation-based type-directed partial evaluation using an experimental version of `call/cc` for OCaml.

We have designed a set of byte-code combinators for directly representing OCaml byte code. We have illustrated the use of byte-code combinators to stage a program directly using its generating extension. This style of programming amounts to implementing generating extensions by hand and occurs in macro-systems and partial evaluation. The generating extensions we write are fast and they generate efficient byte code. We have also integrated the byte-code combinators with type-directed partial evaluation to facilitate specialization with byte code as output. This style of programming amounts to implementing general programs that can be specialized at run time. The representation of residual programs as text or as byte-code combinators is orthogonal to type-directed partial evaluation. Similarly, the byte-code combinators could easily be integrated with other partial evaluators.

Sperber and Thiemann's run-time specialization generates byte-code instructions for an untyped language. Balat and Danvy's dedicated compiler generates byte-code instructions for a strongly typed language. They only produce normal forms of closed terms. These are always simply typed. In contrast, we produce residual programs that are not guaranteed to be typed. We note, however, that run-time specializers can predict the type of a residual program from the type of the source program.

4.6.2 Run-time code generation for C

There exist a number of run-time code generators for C. Both DCG [57] and VCODE [55] extend C with low-level code-generation primitives. DCG implements a set of library routines for constructing binary code while VCODE provides a RISC-like low-level interface to generating binary code. ‘C is an extension of the C language with Lisp-like quasiquotation for specifying dynamically generated code [56]. ‘C programs are translated into either ANSI C programs employing either DCG [56] or VCODE [112] as a run-time code generation back-end. DCG, VCODE, and ‘C support writing generating extensions by hand but do not automatically specialize source program.

Tempo [25, 107], DyC [68], and Cyclone [77] are run-time specialization systems for C. Based on an initial binding-time signature, these systems split the source program into static and dynamic parts. While the dynamic parts do contain unknowns, they can be compiled into binary code by an almost standard C compiler. From the binding-time annotations the static compiler also generates a template that specifies how the pre-compiled blocks should be re-assembled at run time. Tempo, DyC, and Cyclone do not support writing generating extensions by hand.

Template-based run-time code generation often generates efficient code at run time. In particular, when the source program is split into few large blocks, they may provide enough flow information that an optimizing C compiler can generate efficient code for them. In contrast, run-time code generators that supports direct manipulation of compiled code, such as DCG, VCODE, ‘C, and the system that we have presented, typically does not expose optimization opportunities. Furthermore, byte-code programs are difficult to optimize since the set of instructions is often highly specialized. Instead, byte-code run-time systems can often provide a fast virtual machine since each byte-code instruction corresponds to several machine instructions. Thus, the virtual machine imposes a minimal interpretive overhead and can be compiled into efficient binary code.

4.7 Implementation of byte-code combinators

In this section we present the details of the implementation of byte-code combinators.

4.7.1 Constants

```
let mkunit =
  ([], None,
   fun(rho, k) →
     Kconst(Lambda.Const_pointer 0) :: k)

let mkbool b =
  ([], None,
   fun(rho, k) →
     Kconst(Lambda.Const_pointer (if b then 1 else 0)) :: k)

let mkint i =
  ([], None,
   fun(rho, k) →
     Kconst(Lambda.Const_base(Asttypes.Const_int i)) :: k)

let mkstr s =
  ([], None,
   fun(rho, k) →
     Kconst(Lambda.Const_base(Asttypes.Const_string s)) :: k)
```

The instruction `Kconst` p loads the accumulator with the operand p.

4.7.2 Global variables

We illustrate `mkglob` here. The function `mkqref` is similar, but searches through modules.

```
let mkglob g =
  match lookup_global g with
    None       → raise ERROR
  | Some ident →
      ([], None,
       fun (env, k) → Kgetglobal ident :: k)
```

The instruction `Kgetglobal` i accesses the global table at run time.

4.7.3 Finite products

```
let rec comp l es rho k =
  match es with
    [(_, _, f)]     → f(rho, Kmakeblock (l, 0) :: k)
  | (_, _, f) :: es → f(rho, Kpush :: comp l es (installTmp rho) k)

let mktup (es : term list) =
  let l   = List.length es in
  let es' = List.rev es in
  (List.concat (List.map (fun (vs, _, _) → vs) es), None,
   fun(rho, k) → comp l es' rho k)
```

The instruction Kmakeblock(l, t) allocates a block of l elements, stores the value of the accumulator in the first position, and copies the top $l - 1$ elements from the stack to the rest of the block. The top $l - 1$ elements are removed from the stack. The value t becomes the tag for the block.

The instruction push pushes the value of the accumulator on top of the stack.

```
let mkprj (vs, _, f) i =
  (vs, None,
   fun(rho, k) → f(rho, Kgetfield (i - 1) :: k))
```

The instruction Kgetfield i accesses the ith element of the heap-allocated block that the accumulator points to.

4.7.4 Conditionals

```
let mkif (vsa, _, fa) (vsb, _, fb) (vsc, _, fc) =
  (vsa @ vsb @ vsc, None,
   fun(rho, k) →
     let l1 = new_label() in
     let l2 = new_label() in
     fa(rho,    Kbranchifnot l1
        :: fb(rho,    Kbranch l2
              :: Klabel l1
              :: fc(rho, Klabel l2 :: k))))
```

The instruction Kbranchifnot l jumps to the label l if the accumulator holds the truth-value false.

The instruction Kbranch l jumps unconditionally to the label l.

The pseudo-instruction Klabel l inserts a label in the list of symbolic instructions.

4.7.5 Free variables

```
let mkvar x =
  ([x], Some x,
   fun(rho, k) →
     match lookup rho x with
       ARG i → Kacc i :: k
     | ENV i → Kenvacc (i + 1) :: k)
```

The instruction Kacc i loads the accumulator with the ith element on the stack.

The instruction Kenvacc i loads the accumulator with the ith element in the (run-time) environment pointed to by the register env. A (run-time) environment is really a closure whose first element contains a code pointer.

In the current implementation, (compile-time) environments are lists of pairs associating variables with their position. Is is straightforward to replace this naive representation with a more efficient one.

4.7.6 Abstractions

For conciseness, here we only show the low-level first-order constructor of λ-abstractions and how it can be used in defining the higher-order uncurried constructor.

```
let mklam1 x (vsb, _, fb) =
  let free_variables = remove x vsb in
  (free_variables, None,
   fun(rho, k) →
     let l1 = new_label() in
     let rec savefv ys rho' ts =
       match ys with
         [] → (   Kclosure(l1, List.length free_variables)
                :: k @ Klabel l1 :: fb(installArg rho' x, [Kreturn 1]))
       | [v] →
           begin match lookup rho v with
             ARG i → (   Kacc (i + ts)
                       :: savefv [] (installEnv rho' v) ts)
           | ENV i → (   Kenvacc (i + 1)
                       :: savefv [] (installEnv rho' v) ts)
           end
       | v :: vs →
           begin match lookup rho v with
             ARG i → (   Kacc (i + ts)
                       :: Kpush
                       :: savefv vs (installEnv rho' v) (ts + 1))
           | ENV i → (   Kenvacc (i + 1)
                       :: Kpush
                       :: savefv vs (installEnv rho' v) (ts + 1))
           end
     in  savefv free_variables rho 0)

  let mklam f = let x = gensym() in mklam1 x (f (mkvar x))
```

The instruction `Kclosure` (c, i) generates a closure with code pointer c and with an environment given by the topmost i elements on the stack.

The instruction `Kreturn` n removes n arguments from the stack and then pops a saved return address and a saved environment off the stack.

The code for the body of the λ-abstraction immediately follows the current continuation.

4.7.7 Applications

For conciseness, we only show the unoptimizing constructor here. It uses an auxiliary function that generates the correct type of call depending on whether the call is in tail position or not.

```
let apply_before k =
  match k with
    Kreturn n      :: k' → Kappterm(1, n + 1) :: k'
  | Kappterm(i, j) :: k' → Kappterm(i + 1, j + 1) :: k'
  | _                    → Kapply 1 :: k

let mkapp (vsa, fa) (vsb, fb) =
  (vsa @ vsb,
   fun(rho, k) →
     fb(rho, (   Kpush
              :: fa(installTmp rho, apply_before k))))
```

The instruction Kappterm (i, j) performs a tail call with i arguments and where the current stack frame contains j elements.

The instruction Kapply i performs an ordinary non-tail call with i arguments.

4.7.8 Let expressions

```
let mklet ((vsa, _, fa) as v) f =
  let x = gensym() in
  let (vsb, varb, fb) = f (mkvar x) in
  if varb = Some x then
    v
  else
    (vsa @ (List.filter (fun y → not(x = y)) vsb), None,
     fun(rho, k) →
       fa(rho, (   Kpush
                :: fb(installArg rho x,
                      pop_before(1, k)))))
```

4.7.9 Imperative features

```
let mkseq (vsa, _, fa) (vsb, _, fb) =
  (vsa @ vsb, None,
   fun(rho, k) → fa(rho, fb(rho, k)))


let mkref (vs, _, f) =
  (vs, None,
   fun (rho, k) →
     f(rho, Kmakeblock (1,0) :: k))

let mkget (vs, _, f) =
  (vs, None,
   fun (rho, k) →
     f(rho, Kgetfield 0 :: k))

let mkset (vsa, _, fa) (vsb, _, fb) =
  (vsa @ vsb, None,
   fun (rho, k) →
     fb(rho,    Kpush
         :: fa(installTmp rho,
               Ksetfield 0 :: k)))
```

The instruction `Ksetfield` i stores the value in the topmost stack entry in the i'th position in the heap-allocated block that the accumulator points to.

4.7.10 Primitive operations

For conciseness, we only illustrate the case for addition here.

```
let mkadd (vsa, _, fa) (vsb, _, fb) =
  (vsa @ vsb, None,
   fun (rho, k) →
     fb(rho,    Kpush
         :: fa(installTmp rho,
               Kaddint :: k)))
```

4.8 Conclusions and issues

We have designed a library of byte-code combinators for OCaml byte code. We have illustrated their use in semantics-directed compilation, as the code-generating primitives of generating extensions, and in run-time specialization, as the code-generating primitives of type-directed partial evaluation.

Chapter 5

Goal-directed evaluation

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 chapter, 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.

Note. This chapter is based on joint work with Olivier Danvy and Bernd Grobauer presented at SAIG 2001 [37] and to appear in New Generation Computing [38].

Thanks are due to the anonymous referees for comments and to Andrzej Filinski for discussions.

5.1 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 [69].

Backtracking as a language feature complicates both semantics and implementation. Gudeman [71] 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 [19, 75, 139]. Proebsting and Townsend, the implementors of an Icon compiler in Java, observe that continuations can be compiled into efficient code [4, 74], but nevertheless dismiss them because “[they] are notoriously difficult to understand, and few target languages directly support them” [114, p.38]. Instead, their compiler is based on a translation scheme proposed by Proebsting [113], which is based on the four-port model used for describing control flow in Prolog [17]. 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 [70] 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 chapter, 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 [71] as instantiations of one generic semantics based on computational monads [104]. 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 [24, 83]. 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 this chapter is structured as follows: In Section 5.2 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 5.3 we describe semantics-directed compilation for a goal-directed language. Section 5.4 concludes.

5.2 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.

5.2.1 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)`.

5.2.2 Monads and semantics

Computational monads were introduced to structure denotational semantics [104]. 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 proofs between semantics arising from instantiation with different monads can be conducted in a modular way, using the concept of a monad morphism [137].

Monads can also be used to structure functional programs [138]. 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 5.1. For these operations, the so-called monad laws have to hold.

In Section 5.2.4 we give a denotational semantics of the goal-directed language described in Section 5.2.1. Anticipating semantics-directed compilation by partial evaluation, we describe the semantics in terms of ML, in effect defining an interpreter. The semantics $[\![\cdot]\!]_{\mathsf{M}} : Exp \rightarrow int\,\mathsf{M}$ is parameterized over a monad M, where $\alpha\,\mathsf{M}$ represents a sequence of values of type α.

$$\begin{array}{rcl} \mathit{unit}_{\mathsf{M}} & : & \alpha \rightarrow \alpha\,\mathsf{M} \\ \mathit{map}_{\mathsf{M}} & : & (\alpha \rightarrow \beta) \rightarrow \alpha\,\mathsf{M} \rightarrow \beta\,\mathsf{M} \\ \mathit{join}_{\mathsf{M}} & : & (\alpha\,\mathsf{M})\,\mathsf{M} \rightarrow \alpha\,\mathsf{M} \end{array}$$

Figure 5.1: Monad operators and their types

Standard monad operations:

$$\begin{array}{lcl} \mathit{unit}_{\mathsf{L}}\ x & = & [x] \\ \mathit{map}_{\mathsf{L}}\ f\ [\,] & = & [\,] \\ \mathit{map}_{\mathsf{L}}\ f\ (x :: xs) & = & (f\ x) :: (\mathit{map}_{\mathsf{L}}\ f\ xs) \\ \mathit{join}_{\mathsf{L}}\ [\,] & = & [\,] \\ \mathit{join}_{\mathsf{L}}\ (l :: ls) & = & l\ @\ (\mathit{join}_{\mathsf{L}}\ ls) \end{array}$$

Special operations for sequences:

$$\begin{array}{lcl} \mathit{empty}_{\mathsf{L}} & = & [\,] \\ \mathit{if\_empty}_{\mathsf{L}}\ [\,]\ ys\ zs & = & ys \\ \mathit{if\_empty}_{\mathsf{L}}\ (x :: xs)\ ys\ zs & = & zs \\ \mathit{append}_{\mathsf{L}}\ xs\ ys & = & xs\ @\ ys \end{array}$$

Figure 5.2: The list monad

5.2.3 A monad of sequences

In order to handle sequences, some structure is needed in addition to the three generic monad operations displayed in Figure 5.1. We add three operations:

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

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 5.2; for lists, “join” is sometimes also called “flatten” or, in ML, “concat”.

5.2.4 A monadic semantics

A monadic semantics of the goal-directed language described in Section 5.2.1. is given in Figure 5.3. We explain the semantics in terms of the list monad. A literal i is interpreted as

$$
\begin{array}{rcl}
[\![\cdot]\!]_{\mathsf{M}} & : & \mathit{Exp} \to \mathit{int}\ \mathsf{M} \\[6pt]
[\![i]\!]_{\mathsf{M}} & = & \mathit{unit}_{\mathsf{M}}\ i \\
[\![E_1\ \texttt{to}\ E_2]\!]_{\mathsf{M}} & = & \mathit{bind2}_{\mathsf{M}}\ (\lambda xy.\mathit{to}_{\mathsf{M}}\ x\ y)\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![E_1\ \texttt{+}\ E_2]\!]_{\mathsf{M}} & = & \mathit{bind2}_{\mathsf{M}}\ (\lambda xy.\mathit{unit}_{\mathsf{M}}\ (x+y))\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![E_1\ \texttt{<=}\ E_2]\!]_{\mathsf{M}} & = & \mathit{bind2}_{\mathsf{M}}\ (\lambda xy.\mathit{leq}_{\mathsf{M}}\ x\ y)\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}} \\
[\![\texttt{if}\ E_0\ \texttt{then}\ E_1 & & \\
\texttt{else}\ E_2]\!]_{\mathsf{M}} & = & \mathit{if\_empty}_{\mathsf{M}}\ [\![E_0]\!]_{\mathsf{M}}\ [\![E_1]\!]_{\mathsf{M}}\ [\![E_2]\!]_{\mathsf{M}}
\end{array}
$$

where

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

Figure 5.3: Monadic semantics for a subset of Icon

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 $\mathit{bind2}$ and a function of type $\mathit{int} \to \mathit{int} \to \mathit{int}$ 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 $\mathit{bind2}_{\mathsf{L}}\ f\ xs\ ys$ 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 5.2.1 of how the effect of goal-directed evaluation can be achieved in ML using lists.

5.2.5 A spectrum of semantics

In the following, we describe four possible instantiations of the semantics given in Figure 5.3. Because a semantics corresponds directly to an interpreter, we thus create four different interpreters.

A list-based interpreter

Instantiating the semantics with the list monad from Figure 5.2 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.

Standard monad operations:

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

Special operations for sequences:

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

Figure 5.4: The continuation monad

A stream-based interpreter

Implementing the list monad from Figure 5.2 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.

A continuation-based interpreter

Gudeman [71] 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 5.4. 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 [20] of the higher-order representation of lists proposed by Hughes [81]. Hughes observed that when constructing the partially applied concatenation function $\lambda ys.xs\ @\ ys$ 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 represented by a partially applied cons function:

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

Church-encoding a data type means abstracting over selector functions, in this case “ :: ”:

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

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

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 5.5: A continuation semantics

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

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 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 5.4, 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 5.4 can be conceptually linked to the list monad, the present continuation monad can be linked to the stream monad by a Church encoding of

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

where

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

Figure 5.6: A semantics with success and failure continuations

the data type of streams:

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

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

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

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

The interpreter defined by the semantics $[\![\cdot]\!]_{\mathsf{C}_2}$ is the starting point of the semantics-directed compilation described in Section 5.3. Figure 5.6 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.

5.2.6 Correctness

So far, we have related the various semantics presented in Section 5.2.5 only conceptually. Because the four different interpreters presented in Section 5.2.5 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 [137].

Definition 8 (Monad morphism) *If* M *and* N *are two monads, then* $h : \alpha\,\mathsf{M} \rightarrow \alpha\,\mathsf{N}$ *is a* monad morphism *if it preserves the monad operations*[1]*, i.e.,*

$$\begin{array}{rcl} h \circ unit_{\mathsf{M}} & = & unit_{\mathsf{N}} \\ h \circ map_{\mathsf{M}}\, f & = & map_{\mathsf{N}}\, f \circ h \\ h \circ join_{\mathsf{M}} & = & join_{\mathsf{N}} \circ h \circ map_{\mathsf{M}}\, h \\ h\ empty_{\mathsf{M}} & = & empty_{\mathsf{N}} \\ h \circ if\_empty_{\mathsf{M}} & = & \lambda xs.\lambda ys.\lambda zs.if\_empty_{\mathsf{N}}(h\, xs)(h\, ys)(h\, zs) \\ h \circ append_{\mathsf{M}} & = & \lambda xs.\lambda ys.append_{\mathsf{N}}(h\, xs)(h\, ys) \end{array}$$

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 11 *Let* M *and* N *be monads of sequences as specified in Section 5.2.3. If* h *is a monad morphism from* M *to* N*, then* $(h\, [\![E]\!]_{\mathsf{M}}) = [\![E]\!]_{\mathsf{N}}$ *for every Icon expression* E*.*

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 11 to show that the list-based interpreter and the continuation-based interpreter from Section 5.2.5 always yield comparable results:

Proposition 1 *Let* $show\ :\ \alpha\, \mathsf{C} \rightarrow \alpha\, \mathsf{L}$ *be defined as*

$$show\ f\ =\ f\ (\lambda x.\lambda xs.append_{\mathsf{L}}\ (unit_{\mathsf{L}}\ x)\ xs)\ empty_{\mathsf{L}}.$$

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

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

$$\begin{array}{rcl} h\ [\,] & = & empty_{\mathsf{C}} \\ h\ (x :: xs) & = & append_{\mathsf{C}}\ (unit_{\mathsf{C}}\ x)\ (h\ xs) \end{array}$$

is a monad morphism from L to C, and (2) the function $(show \circ h)$ is the identity function on lists. The proposition then follows immediately with Lemma 11.

5.2.7 Summary

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

[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.

```
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 5.7: The abstract syntax of Icon terms

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 [4, 74]; in the following section we show that partial evaluation is sufficient to generate efficient code from the the continuation semantics derived in the final paragraph of Section 5.2.5.

5.3 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 5.6. The source language of the interpreter is shown in Figure 5.7. In Section 5.3.1 we present the Icon interpreter written in ML. In Section 5.3.1, 5.3.2, and 5.3.3 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.

5.3.1 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 [44].

Type-directed partial evaluation is an approach to off-line specialization of higher-order programs [34]. 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.

```
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 5.8: Signature of primitive operations

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).

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 5.8. For the residualizing primitives we let the partial evaluator produce functions that generate ML programs with meaningful variable names [34].

The parameterized interpreter is shown in Figure 5.9. The main function `eval` takes an Icon term and two continuations, `k : tint → (tunit → res) → res` and `f : tunit → 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.)

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 β-redexes and it is fully η-expanded with respect to its type.

Example 21 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 ⇒
```

```
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 5.9: Parameterized interpreter

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

Avoiding code duplication

The result of specializing the interpreter in Figure 5.9 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 [83]. 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 5.3.2 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.

5.3.2 Generating C programs

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

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

imposes further restrictions: A residual program must take two arguments, a success continuation `k : tint → (tunit → res) → res` and a failure continuation `f : tunit → 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 → res`;
- applying the failure continuation `f`;
- applying the primitive `cond` to a boolean and two functions of type `tunit → res`;
- applying the primitive `fix` to a function of two arguments, $\texttt{loop}_n$: `tint → 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 → 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 → (tunit → res) → 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 → 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 5.10. (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 12 *The residual program generated from applying type-directed partial evaluation to the interpreter in Figure 5.9 can be generated by the grammar in Figure 5.10.*

The idea of generating grammars for residual programs has been studied by, e.g., Malmkjær [98] and is used in the run-time specializer Tempo to generate code templates [25].

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 5.11 and 5.12 show the translation from residual programs to C programs.

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

Figure 5.10: Grammar of residual 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 5.11 and 5.12 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 5.10 and then showing equivalence.

Example 22 Consider again the Icon term `10 + (4 to 7)` from Example 21. It is translated into the following C program.

```
        i0 = 4;                    |        goto loop0;
loop0:  if (i0<=7) goto L0;        |
        goto fail;                 | succ:  printf("%d ", value);
                                   |        goto resume;
L0:     value = 10 + i0;           |
        goto succ;                 | fail:  printf("\n");
                                   |        exit(0);
resume: i0 = i0 + 1;               |
```

The C target programs corresponds to the target programs of Proebsting's optimized template-based compiler [113]. In effect, we are automatically generating flow-chart programs from the denotation of an Icon term.

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

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

$$|\texttt{f ()}|_{\mathrm{S}} = \left\{\texttt{goto fail;}\right.$$

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

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

$$|\texttt{loop}_n\ E|_{\mathrm{S}} = \left\{\begin{array}{l} \texttt{i}_n\texttt{ = } |E|_{\mathrm{E}}\texttt{;} \\ \texttt{goto loop}_n\texttt{;} \end{array}\right.$$

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

$$|\texttt{resume}_n\texttt{ ()}|_{\mathrm{S}} = \left\{\begin{array}{l} \texttt{if (gate}_n\texttt{) goto resume}^1_n\texttt{;} \\ \texttt{goto resume}^0_n\texttt{;} \end{array}\right.$$

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

Figure 5.11: Translating residual programs into C (Statements)

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

Figure 5.12: Translating residual programs into C (Expressions)

5.3.3 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 OCaml byte-code combinators from Chapter 4 and by composing the second compiler with a hand-written compiler from flow charts into OCaml byte code.

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 [25, 94, 127]. 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 5.9 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.)

Compiling flow charts into OCaml byte code

We have modified the translation in Figure 5.11 and 5.12 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 [129].

5.3.4 Summary

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 [113].

5.4 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 [71].

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 [113].

Gudeman [71] 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 [60, 91].

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 [116], to obtain stack-based specifications and the corresponding run-time architectures.

Chapter 6

Conclusions and perspectives

Higher-order programming languages are built on the principles of abstraction and parameterization. As such, they provide an expressive power that is needed to write generic programs, which by their very nature are parameterized. In this dissertation, we have applied higher-order techniques in several areas of program generation.

Interpreters and compilers for domain-specific languages have been implemented by embedding their functionality into existing meta-languages. Recent work also embed type systems for simple types into higher-order polymorphically typed meta-languages using phantom types. We have shown (Chapter 2) that such an embedding is sound and complete for an idealized higher-order meta-language. To our knowledge, the only embedding of a higher-order object language into Haskell is the type-preserving implementation of type-directed partial evaluation that we have presented in Chapter 3. Other embedded object languages are first order. Time will tell whether there are further applications of higher-order embedded languages.

Partial evaluation has been used to specialize generic software components to specific areas of application. Thus, partial evaluation supports re-using software in several different applications. Type-directed partial evaluation, in particular, is an approach to specializing higher-order typed programs. We have derived a statically typed implementation of type-directed partial evaluation (Chapter 3), thus making it work in the context of statically typed languages such as Haskell and Standard ML. We have also shown how use the type inferencer for these languages as a theorem prover to show that type-directed partial evaluation preserves types and yields normal forms.

Traditional partial evaluation generates residual programs as text which must be compiled before they can be executed. However, adaptive software may want to specialize components on the fly. Stand-alone compilers are typically too expensive to operate at run time. Run-time code generation techniques enable quick compilation of specialized programs. We have presented (Chapter 4) a collection of byte-code combinators that enables run-time code generation for OCaml byte code. We have successfully applied them semantics-directed compilation using handwritten generating extensions. We have also applied successfully applied them to run-time specialization using type-directed partial evaluation. We have used them in an application of type-directed partial evaluation to specialize the term rewriter of a large theorem prover. Although the performance results where not as convincing as hoped for, they did indicate directions of research in the area of type-directed partial evaluation. First, the input to type-directed partial evaluation is essentially a binding-time annotated program. To make type-directed partial evaluation useful in practice, we believe it should be composed with an (almost) automatic binding-time analysis. Second, naively inserting let-expressions exercises a cost for call-by-value type-directed partial evaluation as well as for the residual programs. A possible solution is to reduce the number of let-expression in a post-processing phase. The challenge is then to specify such a phase in the context of run-time code generation.

Our final contribution is in semantics-directed compilation for a goal-directed language (Chapter 5). Existing compilers for these languages use ad hoc “template-base” compilation techniques. Instead, we have specialized a continuation-passing style interpreter for a goal-directed language yielding ML programs as output. We have then observed that the residual programs are flow-chart programs. Thus, by composing the output of partial evaluation with various back-ends we have compiled goal-directed programs into efficient C code and OCaml byte code.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt and David E. Rydeheard, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199. Springer-Verlag, 1995.

[3] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In Clarke [21].

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

[5] 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.

[6] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.

[7] David B. Bartley and John C. Jensen. The implementation of PC Scheme. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 86–93, Cambridge, Massachusetts, August 1986. ACM Press.

[8] Alan Bawden. Quasiquotation in Lisp. In Danvy [35], pages 4–12.

[9] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993.

[10] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[11] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[12] Andrew A. Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, December 1990.

[13] Anders Bondorf. Compiling laziness by partial evaluation. In Simon L. Peyton Jones, Guy Hutton, and Carsten K. Holst, editors, *Functional Programming, Glasgow 1990*, Workshops in Computing, pages 9–22, Glasgow, Scotland, 1990. Springer-Verlag.

[14] 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.

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

[16] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

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

[18] William E. Carlson, Paul Hudak, and Mark P. Jones. An experiment using Haskell to prototype 'geometric region servers' for navy command and control. Technical Report 1031, Yale University, New Haven, Connecticut, November 1993.

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

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

[21] Edmund M. Clarke, editor. *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[22] C. Consel and S.C. Khoo. Semantics-directed generation of a Prolog compiler. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 135–146, Passau, Germany, August 1991. Springer-Verlag.

[23] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[24] 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.

[25] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In Steele [128], pages 145–156.

[26] Catarina Coquand. From semantics to rules: A machine assisted analysis. In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *Proceedings of CSL'93*, number 832 in Lecture Notes in Computer Science, pages 91–105. Springer-Verlag, 1993.

[27] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[28] Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.

[29] 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.

[30] Olivier Danvy. Type-directed partial evaluation. In Steele [128], pages 242–257.

[31] Olivier Danvy. A user's guide to type-directed partial evaluation. Unpublished manuscript, 1996.

[32] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

[33] 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.

[34] 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.

[35] Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, number NS–99–1 in BRICS Note Series, San Antonio, Texas, January 1999.

[36] 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.

[37] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. In Walid Taha, editor, *Proceedings of the Second Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG 2001)*, number 2196 in Lecture Notes in Computer Science, Florence, Italy, September 2001. Springer-Verlag. An extended version appears in Vol. 20, No. 1 of New Generation Computing, Nov. 2001.

[38] Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1), 2001. A preliminary version is available in the proceedings of SAIG 2001.

[39] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995.

[40] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[41] 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.

[42] Olivier Danvy and Morten Rhiger. A simple take on typed abstract syntax in Haskell-like languages. In Herbert Kuchen and Kazunori Ueda, editors, *Proceedings of the Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages 343–358, Tokyo, Japan, March 2001. Springer-Verlag. Extended version available as the technical report BRICS RS-00-34.

[43] Olivier Danvy, Kristoffer Høgsbro Rose, and Morten Rhiger. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001. Extended version available as the technical report BRICS RS-01-16.

[44] 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.

[45] Rowan Davies. A temporal-logic approach to binding-time analysis. In Clarke [21], pages 184–195.

[46] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Steele [128], pages 258–283.

[47] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. Technical report CMU–CS–99–153, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1999. To appear in the Journal of the ACM.

[48] N. G. de Bruijn. Lambda calculus notation with nameless dummies. A tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[49] Jöelle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In P. de Groote and J. R. Hindley, editors, *Proceedings of the 3rd International Conference on Typed Lambda Calculi and Applications*, number 1210 in Lecture Notes in Computer Science, pages 147–163, Nancy, France, April 1997.

[50] Premkumar Devanbu and Jeff Poulin, editors. *Proceedings of the Fifth International Conference on Software Reuse*, Victoria, British Columbia, June 1998. IEEE Computer Society Press.

[51] R. Kent Dybvig. *Chez Scheme User's Guide*. Cadence Research Systems, 1998.

[52] Belmina Dzafic. Formalizing program transformations. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.

[53] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In Chris Ramming, editor, *First Conference on Domain-Specific Languages*, pages 285–296, Santa Barbara, California, October 1997.

[54] Conal Elliott, Sigbjorn Finne, and Oege de Moor. Compiling embedded languages. In Walid Taha, editor, *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, number 1924 in Lecture Notes in Computer Science, pages 9–27, Montréal, Canada, September 2000.

[55] Dawson R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In PLDI'96 [111], pages 160–170.

[56] 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 [128], pages 131–144.

[57] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS VI)*, pages 263–272. ACM Press, Oct 1994.

[58] Andrei P. Ershov. On the essence of compilation. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[59] Joseph H. Fasel, Paul Hudak, Simon Peyton Jones, and Philip Wadler. Haskell special issue. *SIGPLAN Notices*, 27(5), May 1992.

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

[61] 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.

[62] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, Krakow, Poland, May 2001.

[63] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 114–125, Paris, France, September 1999. ACM Press.

[64] Daniel Fridlender and Mia Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, March 2001.

[65] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems, Computers, Controls 2(5), 1971.

[66] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[67] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[68] Brian Grant, Markus Mock, Matthai Phillipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for c. Technical Report UW-CSE-97-03-03, University of Washington, May 1997.

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

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

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

[72] Jason Hickey. Nuprl-light: An implementation framework for higher-order logics. In William McCune, editor, *14th International Conference on Automated Deduction*, number 1249 in Lecture Notes in Artificial Intelligence, pages 395–399. Springer-Verlag, 1997.

[73] Jason J. Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International*

Conference, TPHOLs 2000, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer-Verlag, 2000.

[74] 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.

[75] 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.

[76] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In Giuseppe Longo, editor, *Proceedings of the Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Computer Society Press.

[77] Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(5):337–375, 1999.

[78] Paul Hudak. Modular domain specific languages and tools. In Devanbu and Poulin [50], pages 134–142.

[79] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[80] Gérard Huet. Résolution d'équations dans les langages d'ordre 1, 2, ..., ω. Thèse d'État, Université de Paris VII, Paris, France, 1976.

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

[82] Steven C. Johnson. Yacc – Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[83] 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. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[84] 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.

[85] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. In Devanbu and Poulin [50], pages 224–233.

[86] Jesper Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, January 1992.

[87] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 258–268, Albuquerque, New Mexico, January 1992. ACM Press.

[88] K.M. Kahn and M. Carlsson. The compilation of Prolog programs without the use of a Prolog compiler. In *International Conference on Fifth Generation Computer Systems, Tokyo, Japan*, pages 348–355. Tokyo: Ohmsha and Amsterdam: North-Holland, 1984.

[89] 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.

[90] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–336, 1994.

[91] 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.

[92] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[93] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In Thomas Ball, editor, *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, 1999.

[94] Mark Leone and Peter Lee. Optimizing ML with run-time code generation. In PLDI'96 [111], pages 137–148.

[95] Xavier Leroy. The ZINC experiment, an economical implementation of the ML language. Technical Report 117, INRIA, Le Chesnay, France, February 1990.

[96] Xavier Leroy. *The Objective Caml system, release 3.01*. INRIA, Rocquencourt, France, March 2001.

[97] Michael E. Lesk. Lex – A lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

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

[99] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.

[100] 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.

[101] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985. Springer-Verlag.

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

[103] John C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 8, pages 365–458. The MIT Press, 1990.

[104] 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.

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

[106] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, a formal introduction*. Wiley Professional Computing. John Wiley and Sons, 1992.

[107] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In Purush Iyer and Young il Choo, editors, *Proceedings of the IEEE International Conference on Computer Languages*, Chicago, Illinois, May 1998. IEEE Computer Society. Also available as IRISA report PI-1065.

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

[109] Petite Chez Scheme version 6.0, October 1998. Cadence Research Systems. Available from `http://www.scheme.com`.

[110] 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.

[111] *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5. ACM Press, May 1996.

[112] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A system for fast, flexible, and high-level dynamic code generation. In Cytron [29], pages 109–121.

[113] Todd A. Proebsting. Simple translation of goal-directed evaluation. In Cytron [29], pages 1–6.

[114] 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.

[115] Jonathan Rees. The Scheme of things: The June 1992 meeting. *LISP Pointers*, V(4):40–45, October-December 1992.

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

[117] John C. Reynolds. Normalization and functor categories. In Danvy and Dybjer [36].

[118] Morten Rhiger. A study in higher-order programming languages. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1997.

[119] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In Danvy [35], pages 25–29.

[120] Morten Rhiger. A foundation for embedded languages. Submitted for publication. Extended version available as a BRICS technical report, July 2001.

[121] Kristoffer Rose. Type-directed partial evaluation using type classes. In Danvy and Dybjer [36].

[122] Erik Ruf. *Topics in Online Partial Evaluation*. PhD thesis, Stanford University, Stanford, California, February 1993. Technical report CSL-TR-93-563.

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

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

[125] Dana Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993.

[126] Michael Sperber. Self-applicable online partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 465–480, Dagstuhl, Germany, February 1996. Springer-Verlag.

[127] Michael Sperber and Peter Thiemann. Two for the price of one: composing partial evaluation and compilation. In Cytron [29], pages 215–225.

[128] 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.

[129] 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.

[130] Guy L. Steele Jr. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999.

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

[132] Eijiro Sumii and Naoki Kobayashi. Online-and-offline partial evaluation: A mixed approach. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Boston, Massachusetts, January 2000. ACM Press.

[133] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type safety. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, number 1443 in Lecture Notes in Computer Science, pages 918–929. Springer-Verlag, 1998.

[134] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, June 1997. ACM Press.

[135] Peter Thiemann. Cogen in six lines. In R. Kent Dybvig, editor, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 180–189, Philadelphia, Pennsylvania, May 1996. ACM Press.

[136] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1989. Special issue on ESOP'88, the Second European Symposium on Programming, Nancy, France, March 21-24, 1988.

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

[138] 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.

[139] 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.

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

[141] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241. ACM Press, January 1982.

[142] Mitchell Wand. Embedding type structure in semantics. In Mary S. Van Deusen and Zvi Galil, editors, *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 1–6. ACM Press, January 1985.

[143] Daniel Weise, Roland Conybeare, Erik Ruf, and Scott Seligman. Automatic online partial evaluation. In John Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 165–191, Cambridge, Massachusetts, August 1991. Springer-Verlag.

[144] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

[145] 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.