# *Finally Tagless, Partially Evaluated*

*Tagless Staged Interpreters for Simpler Typed Languages*

Jacques Carette, Oleg Kiselyov and Chung-chieh Shan

**Abstract**

We have built the first family of tagless interpretations for a higher-order typed object language in a typed metalanguage (Haskell or ML) that require no dependent types, generalized algebraic data types, or postprocessing to eliminate tags. The statically type-preserving interpretations include an evaluator, a compiler (or staged evaluator), a partial evaluator, and call-by-name and call-by-value CPS transformers.

Our main idea is to encode de Bruijn or higher-order abstract syntax using cogen functions rather than data constructors. In other words, we represent object terms not in an initial algebra but using the coalgebraic structure of the $\lambda$-calculus. Our representation also simulates inductive maps from types to types, which are required for typed partial evaluation and CPS transformations.

Our encoding of an object term abstracts over the various ways to interpret it, yet statically assures that the interpreters never get stuck. To achieve self-interpretation and show Jones-optimality, we relate this exemplar of higher-rank and higher-kind polymorphism (provided by ML functors and Haskell 98 constructor classes) to plugging a term into a context of let-polymorphic bindings.

*It should also be possible to define languages with a highly refined syntactic type structure. Ideally, such a treatment should be metacircular, in the sense that the type structure used in the defined language should be adequate for the defining language.*
(Reynolds 1972)

## 1 Introduction

A popular way to define and implement a language is to embed it in another (Reynolds 1972). Embedding means to represent terms and values of the *object language* as terms and values in the *metalanguage*. Embedding is especially appropriate for domain-specific object languages because it supports rapid prototyping and integration with the host environment (Hudak 1996). If the metalanguage supports *staging*, then the embedding can compile object programs to the metalanguage and avoid the overhead of interpreting them on the fly (Pašalić et al. 2002). A staged definitional interpreter is thus a promising way to build a domain-specific language (DSL).

We focus on embedding a *typed* object language into a *typed* metalanguage. The benefit of types in this setting is to rule out meaningless object terms, thus enabling faster interpretation and assuring that our interpreters do not get stuck. To be concrete, we use the typed object language in Figure 1 throughout this paper. We aim not just for evaluation of object programs but also for compilation, partial evaluation, and other processing.

Pašalić et al. (2002) and Xi et al. (2003) motivated interpreting a typed object language in a typed metalanguage as an interesting problem. The known solutions to this problem store

$$\frac{\begin{array}{c}[x:t_1]\\ \vdots\\ e:t_2\end{array}}{\lambda x.\,e:t_1\to t_2} \qquad \frac{\begin{array}{c}[f:t_1\to t_2]\\ \vdots\\ e:t_1\to t_2\end{array}}{\mathrm{fix}\,f.\,e:t_1\to t_2} \qquad \frac{e_1:t_1\to t_2 \quad e_2:t_1}{e_1e_2:t_2} \qquad \frac{n\text{ is an integer}}{n:\mathbb{Z}} \qquad \frac{b\text{ is a boolean}}{b:\mathbb{B}}$$

$$\frac{e:\mathbb{B} \quad e_1:t \quad e_2:t}{\text{if } e \text{ then } e_1 \text{ else } e_2:t} \qquad \frac{e_1:\mathbb{Z} \quad e_2:\mathbb{Z}}{e_1+e_2:\mathbb{Z}} \qquad \frac{e_1:\mathbb{Z} \quad e_2:\mathbb{Z}}{e_1\times e_2:\mathbb{Z}} \qquad \frac{e_1:\mathbb{Z} \quad e_2:\mathbb{Z}}{e_1\le e_2:\mathbb{B}}$$

Fig. 1. Our typed object language

object terms and values in the metalanguage in a universal type, a generalized algebraic data type (GADT), or a dependent type. In the remainder of this section, we discuss these solutions, identify their drawbacks, then summarize our proposal and contributions. No matter how we represent the object language in the metalanguage, the representation can be created either by hand (for example, by entering object terms at a metalanguage interpreter's prompt) or by a parser/type-checker reading from a text string. We leave aside the solved problem of writing such a parser/type-checker, whether using dependent types (Pašalić et al. 2002) or not (Baars and Swierstra 2002).

## *1.1 The tag problem*

It is straightforward to create an algebraic data type, say in OCaml, to represent object terms such as those in Figure 1. For brevity, we elide treating integers, conditionals, and fixpoint in this section.

```
type var = VZ | VS of var
type exp = V of var | B of bool | L of exp | A of exp * exp
```

We represent each variable using a unary de Bruijn index. For example, we represent the object term $(\lambda x.x)$ true as

```
let test1 = A (L (V VZ), B true)
```

Following (Pašalić et al. 2002), we try to implement an interpreter function `eval0`. It takes an object term such as `test1` above and gives us its value. The first argument to `eval0` is the environment, initially empty, which is the list of values bound to free variables in the interpreted code.

```
let rec lookup (x::env) = function VZ -> x | VS v -> lookup env v
let rec eval0 env = function
| V v       -> lookup env v
| B b       -> b
| L e       -> fun x -> eval0 (x::env) e
| A (e1,e2) -> (eval0 env e1) (eval0 env e2)
```

If our OCaml-like metalanguage were untyped, the code above would be acceptable. The `L e` line exhibits interpretive overhead: `eval0` traverses the function body `e` every time (the result of evaluating) `L e` is applied. Staging can be used to remove this interpretive overhead (Pašalić et al. 2002; §1.1–2).

However, the function `eval0` is ill-typed if we use OCaml or some other typed language as the metalanguage. The line `B b` says that `eval0` returns a boolean, whereas the next line `L e` says the result is a function, but all branches of a pattern-match form must yield values of the same type. A related problem is the type of the environment `env`: a regular OCaml list cannot hold both boolean and function values.

The usual solution is to introduce a universal type (Pašalić et al. 2002; §1.3) containing both booleans and functions.

```
type u = UB of bool | UA of (u -> u)
```

We can then write a typed interpreter

```
let rec eval env = function
| V v       -> lookup env v
| B b       -> UB b
| L e       -> UA (fun x -> eval (x::env) e)
| A (e1,e2) -> match eval env e1 with UA f -> f (eval env e2)
```

whose inferred type is `u list -> exp -> u`. Now we can evaluate

```
let test1r = eval [] test1
val test1r : u = UB true
```

The unfortunate tag `UB` in the result reflects that `eval` is a partial function. First, the pattern match `with UA f` in the line `A (e1,e2)` is not exhaustive, so `eval` can fail if we apply a boolean, as in the ill-typed term `A (B true, B false)`.

```
let test2 = A (B true, B false)
let test2r = eval [] test2
Exception: Match_failure in eval
```

Second, the `lookup` function assumes a nonempty environment, so `eval` can fail if we evaluate an open term

```
let test3 = A (L (V (VS VZ)), B true)
let test3r = eval [] test3
Exception: Match_failure in lookup
```

After all, the type `exp` represents object terms both well-typed and ill-typed, both open and closed.

If we evaluate only closed terms that have been type-checked, then `eval` would never fail. Alas, this soundness is not obvious to the metalanguage, whose type system we must still appease with the nonexhaustive pattern matching in `lookup` and `eval` and the tags `UB` and `UA` (Pašalić et al. 2002; §1.4). In other words, the algebraic data types above fail to express in the metalanguage that the object program is well-typed. This failure necessitates tagging and nonexhaustive pattern-matching operations that incur a performance penalty in interpretation (Pašalić et al. 2002) and impair optimality in partial evaluation (Taha et al. 2001). In short, the universal-type solution is unsatisfactory because it does not preserve typing.

### *1.2 Solutions using fancier types*

It is commonly thought that to interpret a typed object language in a typed metalanguage while preserving types is difficult and requires GADTs or dependent types (Taha et al. 2001). In fact, this problem motivated much work on GADTs (Peyton Jones et al. 2006; Xi et al. 2003) and on dependent types (Fogarty et al. 2007; Pašalić et al. 2002). For a metalanguage's type system to allow the well-typed object term `test1` but disallow the ill-typed object term `test2`, fancier types such as GADTs or dependent types seem necessary. Yet other type systems have been proposed to distinguish closed terms like `test1` from open terms like `test3` (Davies and Pfenning 2001; Nanevski 2002; Nanevski and Pfenning 2005; Nanevski et al. 2007; Taha and Nielsen 2003), so that `lookup` never receives an empty environment. We discuss these proposals further in §7; here we just note that many advanced type systems have been devised to ensure statically that an object term is well-typed and closed.

### *1.3 Our final proposal*

We represent object programs using ordinary functions rather than data constructors. These functions comprise the entire interpreter, shown below.

```
let varZ env        = fst env
let varS vp env     = vp (snd env)
let b (bv:bool) env = bv
let lam e env       = fun x -> e (x,env)
let app e1 e2 env   = (e1 env) (e2 env)
```

We now represent our sample term $(\lambda x.x)$ true as

```
let testf1 = app (lam varZ) (b true)
```

This representation is almost the same as in §1.1, only written with lowercase identifiers. To evaluate an object term is to apply its representation to the empty environment.

```
let testf1r = testf1 ()
val testf1r : bool = true
```

The result has no tags: the interpreter patently uses no tags and no pattern matching. The term `b true` evaluates to a boolean and the term `lam varZ` evaluates to a function, both untagged. The app function applies `lam varZ` without pattern matching. What is more, evaluating an open term such as `testf3` below gives a type error rather than a run-time error.

```
let testf3 = app (lam (varS varZ)) (b true)
let testf3r = testf3 ()
This expression has type unit but is here used with type 'a * 'b
```

The type error correctly complains that the initial environment should be a tuple rather than `()`. In other words, the term is open.

In sum, by Church-encoding terms using ordinary functions, we achieve a tagless evaluator for a typed object language in a metalanguage with a simple type system (Hindley

1969; Milner 1978). In this *final* rather than *initial* approach, both kinds of run-time errors in §1.1 (applying a nonfunction and evaluating an open term) are reported at compile time. Because the new interpreter uses no universal type or pattern matching, it never results in a run-time error, and is in fact total. Because this safety is obvious not just to us but also to the metalanguage implementation, we avoid the serious performance penalty (Pašalić et al. 2002) of error checking. Glück (2002) explains deeper technical reasons that inevitably lead to these performance penalties.

Our solution is *not* Church-encoding the universal type. The Church encoding of the type `u` in §1.1 requires two continuations; the function `app` in the interpreter above would have to provide both to the encoding of `e1`. The continuation corresponding to the `UB` case of `u` must either raise an error or loop. For a well-typed object term, that error continuation is never invoked, yet it must be supplied. In contrast, our interpreter has no error continuation at all.

The evaluator above is wired directly into the functions `b`, `lam`, `app`, and so on. In the rest of this paper, we explain how to abstract the interpreter so as to process the *same* term in many other ways: compilation, partial evaluation, CPS conversion, and so forth.

### *1.4 Contributions*

The term "constructor" functions `b`, `lam`, `app`, and so on appear free in the encoding of an object term such as `testf1` above. Defining these functions differently gives rise to different interpreters, that is, different folds on object programs. Given the same term representation but varying the interpreter, we can

- evaluate the term to a value in the metalanguage;
- measure the size or depth of the term;
- compile the term, with staging support such as in MetaOCaml;
- partially evaluate the term, online; and
- transform the term to continuation-passing style (CPS), even call-by-name (CBN) CPS, so as to isolate the evaluation order of the object language from that of the metalanguage.

We have programmed our interpreters in OCaml (and, for staging, MetaOCaml) and standard Haskell. The complete code is available at `http://okmij.org/ftp/packages/tagless-final.tar.gz` to supplement the paper. Our examples below switch between (Meta)OCaml and Haskell even though we have implemented each example equivalently in both metalanguages, because some of our claims are more obvious in one metalanguage than the other. For example, MetaOCaml provides convenient, typed staging facilities.

We attack the problem of tagless (staged) typed-preserving interpretation exactly as it was posed by Pašalić et al. (2002) and Xi et al. (2003). We use their running examples and achieve the result they call desirable. Our contributions are as follows.

1. We build interpreters that evaluate (§2), compile (or evaluate with staging) (§3), and partially evaluate (§4) a typed higher-order object language in a typed metalanguage, in direct and continuation-passing styles (§5).
2. All these interpreters use no type tags, patently never get stuck, and need no advanced type-system features such as GADTs, dependent types, or intentional type analysis.

3. The partial evaluator avoids polymorphic lift and delays binding-time analysis. It bakes a type-to-type map into the interpreter interface to eliminate the need for GADTs and thus remain portable across Haskell 98 and ML.
4. We show the first typed, tagless, and statically type-preserving CPS transformation with simple types (§5), which is available for mainstream functional languages (ML and Haskell). Type preservation is assured by the soundness of our embedding.
5. We use the type system of the metalanguage to check statically that an object program is well-typed and closed.
6. We show clean, comparable implementations in MetaOCaml and Haskell.
7. We specify a functor signature that encompasses all our interpreters, from evaluation and compilation (§2) to partial evaluation and CPS transformation (§4).
8. We point a clear way to extend the object language with more features such as state (§5.3).
9. We describe an approach to self-interpretation compatible with the above (§6). Self-interpretation turned out to be harder than expected.

Our code is surprisingly simple and obvious in hindsight, but it has been an open problem to interpret a typed object language in a typed metalanguage without tagging or type-system extensions. For example, Taha et al. (2001) say that "expressing such an interpreter in a statically typed programming language is a rather subtle matter. In fact, it is only recently that some work on programming type-indexed values in ML (Yang 1998) has given a hint of how such a function can be expressed." We discuss related work in §7.

To reiterate, we do *not* propose any new language feature or even any new programming technique. Rather, we solve an open problem by a novel combination of simple types and existing techniques. More precisely, we use features already present in mainstream functional languages—Hindley-Milner type system with either an inference-preserving module system or constructor classes, as realized in ML and Haskell 98—and techniques which have all appeared in the literature, to solve a problem that was stated in the published record as unsolved and likely unsolvable in ML or Haskell 98 without extensions. The simplicity of our solution and its use of only mainstream features are virtues that make it more practical to build typed, embedded DSLs.

## 2 The object language and its tagless interpreters

Figure 1 shows our object language, a simply-typed $\lambda$-calculus with fixpoint, integers, booleans, and comparison. The language is close to Xi et al.'s (2003), without their polymorphic lift but with more constants so as to more conveniently express Fibonacci, factorial, and power. In contrast to §1, we encode binding using higher-order abstract syntax (HOAS) (Miller and Nadathur 1987; Pfenning and Elliott 1988) rather than de Bruijn indices. This makes the encoding convenient and also ensures that our object programs are closed.

### *2.1 How to make encoding flexible: abstract the interpreter*

We embed our language in (Meta)OCaml and Haskell. In Haskell, the functions that construct object terms are methods in a type class `Symantics` (with a parameter `repr` of kind

`* -> *`). The class is so named because its interface gives the syntax of the object language and its instances give the semantics.

```
class Symantics repr where
  int  :: Int  -> repr Int
  bool :: Bool -> repr Bool

  lam :: (repr a -> repr b) -> repr (a -> b)
  app :: repr (a -> b) -> repr a -> repr b
  fix :: (repr a -> repr a) -> repr a

  add :: repr Int -> repr Int -> repr Int
  mul :: repr Int -> repr Int -> repr Int
  leq :: repr Int -> repr Int -> repr Bool
  if_ :: repr Bool -> repr a -> repr a -> repr a
```

For example, we encode the term `test1`, or $(\lambda x.x)$ true, from §1.1 above as `app (lam (\x -> x)) (bool True)`, whose inferred type is `Symantics repr => repr Bool`. For another example, the classical *power* function is

```
testpowfix () = lam (\x -> fix (\self -> lam (\n ->
                 if_ (leq n (int 0)) (int 1)
                     (mul x (app self (add n (int (-1))))))))
```

and the partial application $\lambda x.\mathit{power}\,x\,7$ is

```
testpowfix7 () = lam (\x -> app (app (testpowfix ()) x) (int 7))
```

The dummy argument `()` above is to avoid the monomorphism restriction, to keep the type of `testpowfix` and `testpowfix7` polymorphic in `repr`. Instead of supplying this dummy argument, we could have given the terms explicit polymorphic signatures. We however prefer for Haskell to infer the object types for us. We could also avoid the dummy argument by switching off the monomorphism restriction with a compiler flag. The methods `add`, `mul`, and `leq` are quite similar, and so are `int` and `bool`. Therefore, we often show only one method of each group and elide the rest. The accompanying code has the complete implementations.

Comparing `Symantics` with Fig. 1 shows how to represent *every* typed, closed object term in the metalanguage. Moreover, the representation preserves types.

*Proposition 1*
If an object term has the object type $t$, then its representation in the metalanguage has the type `forall repr. Symantics repr => repr` $t$.

Conversely, the type system of the metalanguage statically checks that the represented object term is well-typed and closed. If we err, say replace `int 7` with `bool True` in `testpowfix7`, Haskell will complain there that the expected type `Int` does not match the inferred `Bool`. Similarly, the object term $\lambda x.xx$ and its encoding `lam (\x -> app x x)` both fail occurs-checks in type checking. Haskell's type checker also flags syntactically invalid object terms, such as if we forget `app` somewhere above.

To embed the same object language in (Meta)OCaml, we replace the `Symantics` type class and its instances by a module signature `Symantics` and its implementations. Figure 2

```
module type Symantics = sig type ('c, 'dv) repr
  val int : int  -> ('c, int) repr
  val bool: bool -> ('c, bool) repr

  val lam : (('c, 'da) repr -> ('c, 'db) repr) -> ('c, 'da -> 'db) repr
  val app : ('c, 'da -> 'db) repr -> ('c, 'da) repr -> ('c, 'db) repr
  val fix : ('x -> 'x) -> (('c, 'da -> 'db) repr as 'x)

  val add : ('c, int) repr -> ('c, int) repr -> ('c, int) repr
  val mul : ('c, int) repr -> ('c, int) repr -> ('c, int) repr
  val leq : ('c, int) repr -> ('c, int) repr -> ('c, bool) repr
  val if_ : ('c, bool) repr
            -> (unit -> 'x) -> (unit -> 'x) -> (('c, 'da) repr as 'x)
end
```

Fig. 2. A simple (Meta)OCaml embedding of our object language

```
module EX(S: Symantics) = struct open S
  let test1 () = app (lam (fun x -> x)) (bool true)
  let testpowfix () =
       lam (fun x -> fix (fun self -> lam (fun n ->
        if_ (leq n (int 0)) (fun () -> int 1)
            (fun () -> mul x (app self (add n (int (-1))))))))
  let testpowfix7 = lam (fun x -> app (app (testpowfix ()) x) (int 7))
end
```

Fig. 3. Examples using the embedding in Figure 2 of our object language

shows a simple signature that suffices until §4. The two differences are: the additional type parameter `'c`, an *environment classifier* (Taha and Nielsen 2003) required by MetaOCaml for code generation in §3; and the $\eta$-expanded type for `fix` and thunk types in `if_` since OCaml is a call-by-value language.

The functor `EX` in Fig. 3 encodes our running examples `test1` and the *power* function (`testpowfix`). The dummy argument to `test1` and `testpowfix` is an artifact of MetaOCaml, related to monomorphism: in order for us to run a piece of generated code, it must be polymorphic in its environment classifier (the type variable `'c` in Figure 2). The value restriction dictates that the definitions of our object terms must look syntactically like values. Alternatively, we could have used the rank-2 record types of OCaml to maintain the necessary polymorphism.

Thus, we represent an object expression in OCaml as a functor from `Symantics` to an appropriate semantic domain. This is essentially the same as the constraint `Symantics repr =>` in the Haskell embedding.

### *2.2 Two tagless interpreters*

Having abstracted our term representation over the interpreter, we are now ready to present a series of interpreters. Each interpreter is an instance of the `Symantics` class in Haskell and a module implementing the `Symantics` signature in MetaOCaml.

The first interpreter evaluates an object term to its value in the metalanguage. The module below interprets each object-language operation as the corresponding metalanguage operation.

```
module R = struct
  type ('c,'dv) repr = 'dv (* no wrappers *)

  let int  (x:int)  = x
  let bool (b:bool) = b
  let lam  f        = f
  let app  e1 e2    = e1 e2
  let fix  f        = let rec self n = f self n in self
  let add  e1 e2    = e1 + e2
  let mul  e1 e2    = e1 * e2
  let leq  x y      = x <= y
  let if_  eb et ee = if eb then et () else ee ()
end
```

As in §1.3, this interpreter is patently tagless, using neither a universal type nor any pattern matching: the operation `add` is really OCaml's addition, and `app` is OCaml's application. To run our examples, we instantiate the `EX` functor from §2.1 with `R`.

```
module EXR = EX(R)
```

Thus, `EXR.test1 ()` evaluates to the untagged boolean value `true`. In Haskell, we define

```
newtype R a = R {unR::a}
instance Symantics R where ...
```

Although `R` looks like a tag, it is only a `newtype`. The types `a` and `R a` are represented differently only at compile time, not at run time. Pattern matching against `R` cannot ever fail and is assuredly compiled away. In OCaml, too, it is obvious to the compiler that pattern matching cannot fail, because there is no pattern matching. Evaluation can only fail to yield a value due to interpreting `fix`.

*Proposition 2*
If an object term $e$ encoded in the metalanguage has type $t$, then evaluating $e$ in the interpreter `R` either continues indefinitely or terminates with a value of the same type $t$.

Generalizing from `R` to all interpreters, we have the following broader and more useful, if also more obvious, proposition.

*Proposition 3*
If an implementation of `Symantics` never gets stuck, then the type system of the object language is sound with respect to the dynamic semantics defined by that implementation.

These propositions follow immediately from the soundness of the metalanguage's type system. For variety, we show another interpreter, which measures the *size* of each object term, defined as the number of term constructors.

```
module L = struct
  type ('c,'dv) repr = int
```

```
    let int  (x:int)  = 1
    let bool (b:bool) = 1
    let lam  f        = f 0 + 1
    let app  e1 e2    = e1 + e2 + 1
    let fix  f        = f 0 + 1
    let add  e1 e2    = e1 + e2 + 1
    let mul  e1 e2    = e1 + e2 + 1
    let leq  x y      = x + y + 1
    let if_  eb et ee = eb + et () + ee () + 1
  end
```

Now the OCaml expression `let module E = EX(L) in E.test1 ()` evaluates to 3. This interpreter is not only tagless but also total. It "evaluates" even seemingly divergent terms, for instance `app (fix (fun self -> self)) (int 1)` evaluates to 3.

## 3 A tagless compiler (or, a staged interpreter)

Besides immediate evaluation, we can compile our object language into OCaml code using MetaOCaml's staging facilities. MetaOCaml represents future-stage expressions of type *t* at the present stage as values of type `('c, `*t*`) code` where `'c` is the environment classifier (Calcagno et al. 2004; Taha and Nielsen 2003). Code values are created by a *bracket* form `.<`*e*`>.`, which specifies that the expression *e* is to be evaluated at a future stage. The *escape* `.~`*e* must occur within a bracket and specifies that the expression *e* must be evaluated at the current stage; its result, which must be a code value, is spliced into the code being built by the enclosing bracket. The *run* form `.!`*e* evaluates the future-stage code value *e* by compiling and linking it at run-time. The bracket, escape, and run are akin to quasi-quotation, unquotation, and `eval` of Lisp.

Inserting brackets and escapes appropriately into the evaluator `R` above yields the simple compiler `C` below.

```
  module C = struct
    type ('c,'dv) repr = ('c,'dv) code

    let int (x:int)   = .<x>.
    let bool (b:bool) = .<b>.
    let lam f         = .<fun x -> .~(f .<x>.)>.
    let app e1 e2     = .<.~e1 .~e2>.
    let fix f         = .<let rec self n = .~(f .<self>.) n in self>.
    let add e1 e2     = .<.~e1 + .~e2>.
    let mul e1 e2     = .<.~e1 * .~e2>.
    let leq x y       = .<.~x <= .~y>.
    let if_ eb et ee  = .<if .~eb then .~(et ()) else .~(ee ())>.
  end
```

This is a straightforward staging of `module R`. This compiler produces unoptimized code. For example, interpreting our `test1` with

```
  let module E = EX(C) in E.test1 ()
```

gives the code value `.<(fun x_6 -> x_6) true>.` of inferred type `('c, bool) C.repr`. Interpreting `testpowfix7` with

```
let module E = EX(C) in E.testpowfix7
```

gives a code value with many apparent $\beta$- and $\eta$-redexes:

```
.<fun x_1 -> (fun x_2 -> let rec self_3 = fun n_4 ->
   (fun x_5 -> if x_5 <= 0 then 1 else x_2 * self_3 (x_5 + (-1)))
   n_4 in self_3) x_1 7>.
```

This compiler does not incur any interpretive overhead: the code produced for $\lambda x.x$ is simply `fun x_6 -> x_6` and does not call the interpreter, unlike the recursive calls to `eval0` and `eval` in the `L e` lines in §1.1. The resulting code obviously contains no tags and no pattern matching.

*Proposition 4*
The compiler `C` generates tagless code free of pattern matching.

This is syntactically clear from the body of `C`, and follows readily by structural induction and the properties of `.~` and `.< >.`.

We have also implemented this compiler in Haskell. Since Haskell has no (convenient, typed) staging facility, we had to emulate it: we defined a data type `ByteCode` with constructors such as `Var`, `Lam`, `App`, `Fix`, and `INT`. (Alternatively, we could use Template Haskell as our staging facility: `ByteCode` can be mapped to the abstract syntax of Template Haskell. The output of our compiler will then be assuredly type-correct Template Haskell.) Whereas our representation of object terms uses HOAS, our bytecode uses integer-named variables to be realistic. We then define

```
newtype C t = C (Int -> (ByteCode t, Int))
```

where `Int` is the counter for creating fresh variable names. We define the compiler by making `C` an instance of the class `Symantics`. The implementation is quite similar (but slightly more verbose) than the corresponding MetaOCaml code above. (The implementation uses GADTs because we also wanted to write a typed interpreter for the `ByteCode` *data type*.) The accompanying code gives the full details.

## 4 A tagless partial evaluator

Surprisingly, we can write a partial evaluator using the idea above, namely to build object terms using ordinary functions rather than data constructors. We present this partial evaluator in a sequence of four attempts. It uses no universal type and no tags for object types. We then discuss residualization and binding-time analysis. Our partial evaluator is a modular extension of the evaluator in §2.2 and the compiler in §3, in that it uses the former to reduce static terms and the latter to build dynamic terms.

### *4.1 Avoiding polymorphic lift*

Roughly, a partial evaluator interprets each object term to either a static (present-stage) term (using the evaluator `R`) or a dynamic (future-stage) term (using the compiler `C`). To distinguish between static and dynamic terms, one might define

```
data P0 t = S0 (R t) | E0 (C t)
```

To extract a dynamic term from this type, we create the functions

```
abstrI :: P0 Int -> C Int
abstrI (S0 r) = int (unR r)
abstrI (E0 c) = c

abstrB :: P0 Bool -> C Bool
abstrB (S0 r) = bool (unR r)
abstrB (E0 c) = c
```

We then start to define P0 as an instance of the Symantics class. Integer and boolean literals are immediate, present-stage values. Addition yields a static term (using R) if and only if both operands are static; otherwise we extract the dynamic terms from the operands and add them using C.

```
instance Symantics P0 where
  int  x = S0 (int x)
  bool x = S0 (bool x)
  add (S0 e1) (S0 e2) = S0 (add e1 e2)
  add     e1      e2  = E0 (add (abstrI e1) (abstrI e2))
```

Thus, the two uses of add above refer to add for R and add for C.

Whereas mul and leq are as easy to define as add, we encounter a problem with if_. If the first argument to if_ is a dynamic term (of type C Bool), the second a static term (of type R a) and the third a dynamic term (of type C a), then we need to convert the static term to dynamic, but there is no polymorphic "lift" function, of type a -> C a, to send a value to the future stage (Taha and Nielsen 2003; Xi et al. 2003).(By the way, if we were to add polymorphic lift to the type class Symantics repr, then repr would become an instance of Applicative and thus Functor: fmap f = app (lift f) .)

Our Symantics class only includes separate lifting methods bool and int, not a parametrically polymorphic lifting method, for good reason: When compiling to a first-order target language such as machine code, booleans, integers, and functions may well be represented differently. Thus, compiling polymorphic lift requires intensional type analysis. To avoid needing polymorphic lift, we turn to Asai's technique (Asai 2001; Sumii and Kobayashi 2001): build a dynamic term alongside every static term.

### *4.2 Delaying binding-time analysis*

We switch to the Haskell data type

```
data P1 t = P1 (Maybe (R t)) (C t)
```

so that a partially evaluated term P1 t always contains a dynamic component and sometimes contains a static component. The two alternative constructors of a Maybe value, Just and Nothing, tag each partially evaluated term with a phase: either present or future. This tag is not an object type tag: all pattern matching below is exhaustive. Because the future-stage component is always present, we can now define the polymorphic function

```
abstr1 :: P1 t -> C t
abstr1 (P1 _ dyn) = dyn
```

to extract it without requiring polymorphic lift into `C`. We then try to define the interpreter `P1`—and get as far as the first-order constructs of our object language, including `if_`.

```
instance Symantics P1 where
  int  x = P1 (Just (int  x)) (int  x)
  bool b = P1 (Just (bool b)) (bool b)

  add (P1 (Just n1) _) (P1 (Just n2) _) = int (unR (add n1 n2))
  add e1 e2 = P1 Nothing (add (abstr1 e1) (abstr1 e2))
  -- mul and leq are analogous and elided

  if_ (P1 (Just s) _) et ef = if unR s then et else ef
  if_ eb et ef = P1 Nothing
                     (if_ (abstr1 eb) (abstr1 et) (abstr1 ef))
```

When we come to functions, however, we stumble. According to our definition of `P1`, a partially evaluated object function, such as the identity $\lambda x.x$ embedded in Haskell as `lam (\x -> x) :: P1 (a -> a)`, consists of a dynamic part (type `C (a -> a)`) and maybe a static part (type `R (a -> a)`). The dynamic part is useful when this function is passed to another function that is only dynamically known, as in $\lambda k.k(\lambda x.x)$. The static part is useful when this function is applied to a static argument, as in $(\lambda x.x)$ true. Neither part, however, lets us *partially* evaluate the function, that is, compute as much as possible statically when it is applied to a mix of static and dynamic inputs. For example, the partial evaluator should turn $\lambda n.(\lambda x.x)n$ into $\lambda n.n$ by substituting $n$ for $x$ in the body of $\lambda x.x$ even though $n$ is not statically known. The same static function, applied to different static arguments, can give both static and dynamic results: we want to simplify $(\lambda y.x\times y)0$ to 0 but $(\lambda y.x\times y)1$ to $x$.

To enable these simplifications, we delay binding-time analysis for a static function until it is applied, that is, until `lam f` appears as the argument of `app`. To do so, we have to incorporate `f` as it is into the `P1` data structure: applying the type constructor `P1` to a function type `a -> b` should yield one of

```
data P1 (a -> b) = S1 (P1 a -> P1 b) | E1 (C (a -> b))
data P1 (a -> b) = P1 (Maybe (P1 a -> P1 b)) (C (a -> b))
```

That is, we need a nonparametric data type, something akin to type-indexed functions and type-indexed types, which Oliveira and Gibbons (2005) dub the *typecase* design pattern. Thus, typed partial evaluation, like typed CPS transformation, inductively defines a map from source types to target types that performs case distinction on the source type. The connection with CPS is not an accident, as we shall see in §5.1.

### *4.3 Eliminating tags from typecase*

Two common ways to provide typecase in Haskell are GADTs and type-class functional dependencies (Oliveira and Gibbons 2005). These methods are equivalent, and here we use GADTs; `incope1.hs` in the accompanying source code shows the latter. We introduce a GADT with four data constructors.

```
data P t where
  VI :: Int  -> P Int
  VB :: Bool -> P Bool
  VF :: (P a -> P b) -> P (a -> b)
  E  :: C t -> P t
```

The constructors `VI`, `VB`, and `VF` build static terms (like `S0` in §4.1), and `E` builds dynamic terms (like `E0`). However, the type `P t` is no longer parametric in `t`: the constructor `VF` takes an operand of type `P a -> P b` rather than `a -> b`. We define a function like `abstr1` above to extract a future-stage computation from a value of type `P t`.

```
abstr :: P t -> C t
abstr (VI i) = int i
abstr (VB b) = bool b
abstr (VF f) = lam (abstr . f . E)
abstr (E x)  = x
```

The cases of this function `abstr` are type-indexed. In particular, the `VF f` case uses the method `lam` of the `C` interpreter to compile `f`.

We may now make `P` an instance of `Symantics` and implement the partial evaluator as follows. We elide `mul`, `leq`, `if_`, and `fix`.

```
instance Symantics P where
  int x               = VI x
  bool b              = VB b
  add (VI n1) (VI n2) = VI (n1 + n2)
  add e1 e2           = E (add (abstr e1) (abstr e2))
  lam                 = VF
  app (VF f) ea       = f ea
  app (E f)  ea       = E (app f (abstr ea))
```

The implementations of `int`, `bool`, and `add` are like in §4.2. The interpretation of `lam f` is `VF f`, which just wraps the HOAS function `f`. We can always compile `f` to a code value, but we delay it to apply `f` to concrete arguments. The interpretation of `app ef ea` checks to see if `ef` is such a delayed HOAS function `VF f`. If it is, we apply `f` to the concrete argument `ea`, giving us a chance to perform static computations (see example `testpowfix7` in §4.4). If `ef` is a dynamic value `E f`, we residualize.

This solution using GADTs works but is not quite satisfactory. First, it cannot be ported to MetaOCaml, as GADTs are unavailable there. Second, the problem of nonexhaustive pattern-matching reappears in `app` above: the type `P t` has four constructors, of which the pattern in `app` matches only `VF` and `E`. One may say that the constructors `VI` and `VB` obviously cannot occur because they do not construct values of type `P (a -> b)` as required by the type of `app`. Indeed, the metalanguage implementation could reason thus: if we use inductive families (as in Coq) or logical frameworks with canonical forms (as in Twelf with its coverage checker), we can prove the pattern matching to be exhaustive. Then again, the metalanguage implementation may not reason thus: GHC cannot and issues warnings. Although this point may seem minor, it is the heart of the tagging problem and the purpose of tag elimination. A typed tagged interpreter contains many pattern-matching

```
module type Symantics = sig
  type ('c,'sv,'dv) repr

  val int : int  -> ('c,int,int) repr
  val bool: bool -> ('c,bool,bool) repr

  val lam : (('c,'sa,'da) repr -> ('c,'sb,'db) repr as 'x)
            -> ('c,'x,'da -> 'db) repr
  val app : ('c,'x,'da -> 'db) repr
            -> (('c,'sa,'da) repr -> ('c,'sb,'db) repr as 'x)
  val fix : ('x -> 'x) -> (('c, ('c,'sa,'da) repr -> ('c,'sb,'db) repr,
                            'da -> 'db) repr as 'x)

  val add : ('c,int,int) repr -> ('c,int,int) repr -> ('c,int,int) repr
  val mul : ('c,int,int) repr -> ('c,int,int) repr -> ('c,int,int) repr
  val leq : ('c,int,int) repr -> ('c,int,int) repr -> ('c,bool,bool) repr
  val if_ : ('c,bool,bool) repr
            -> (unit -> 'x) -> (unit -> 'x) -> (('c,'sa,'da) repr as 'x)
end
```

Fig. 4. A (Meta)OCaml embedding of our object language that supports partial evaluation

forms that look partial but never fail in reality. The goal is to make this exhaustiveness *syntactically* apparent.

### *4.4 The "final" solution*

Let us re-examine the problem in §4.2. What we would ideally like is to write

```
data P t = P (Maybe (repr_pe t)) (C t)
```

where `repr_pe` is the type function defined by

```
repr_pe Int      = Int
repr_pe Bool     = Bool
repr_pe (a -> b) = P a -> P b
```

Although we can use type classes to define this type function in Haskell, that is not portable to MetaOCaml. However, these three typecase alternatives are already present in existing methods of `Symantics`. A simple and portable solution thus emerges: we bake `repr_pe` into the signature `Symantics`. Switching to MetaOCaml to demonstrate this portability, we recall from Figure 2 in §2.1 that the `repr` type constructor took two arguments `'c` and `'dv`. We add an argument `'sv` for the result of applying `repr_pe` to `'dv`. Figure 4 shows the new signature.

The interpreters `R`, `L` and `C` above only use the old type arguments `'c` and `'dv`, which are treated by the new signature in the same way. Hence, all that needs to change in these interpreters to match the new signature is to add a phantom type argument `'sv` to `repr`. For example, the compiler `C` now begins

```
module C = struct
  type ('c,'sv,'dv) repr = ('c,'dv) code
```

```
module P = struct
  type ('c,'sv,'dv) repr = {st: 'sv option; dy: ('c,'dv) code}
  let abstr {dy = x} = x
  let pdyn x = {st = None; dy = x}

  let int  (x:int ) = {st = Some (R.int  x); dy = C.int  x}
  let bool (x:bool) = {st = Some (R.bool x); dy = C.bool x}

  let add e1 e2 = match e1, e2 with
                | {st = Some 0}, e | e, {st = Some 0} -> e
                | {st = Some m}, {st = Some n} -> int (R.add m n)
                | _ -> pdyn (C.add (abstr e1) (abstr e2))
  let if_ eb et ee = match eb with
                   | {st = Some b} -> if b then et () else ee ()
                   | _ -> pdyn (C.if_ (abstr eb) (fun () -> abstr (et ()))
                                                 (fun () -> abstr (ee ())))

  let lam f = {st = Some f; dy = C.lam (fun x -> abstr (f (pdyn x)))}
  let app ef ea = match ef with
                | {st = Some f} -> f ea
                | _ -> pdyn (C.app (abstr ef) (abstr ea))
  let fix f = let fdyn = C.fix (fun x -> abstr (f (pdyn x)))
              in let rec self = function
                              | {st = Some _} as e -> app (f (lam self)) e
                              | e -> pdyn (C.app fdyn (abstr e))
                 in {st = Some self; dy = fdyn}
end
```

Fig. 5. Our partial evaluator (`mul` and `leq` are elided)

with the rest the same. In contrast, the partial evaluator P relies on the type argument `'sv`.

Figure 5 shows the partial evaluator P. Its type `repr` literally expresses the type equation for `repr_pe` above. The function `abstr`, as in §4.3, extracts a future-stage code value from the result of partial evaluation. Conversely, the function `pdyn` injects a code value into the `repr` type. As in §4.2, we build dynamic terms alongside any static ones to avoid polymorphic lift.

To illustrate how to add optimizations, we improve `add` (and `mul`, elided) to simplify the generated code using the monoid (and ring) structure of `int`: not only is addition performed statically (using R) when both operands are statically known, but it is eliminated when one operand is statically 0; similarly for multiplication by 0 or 1. Such algebraic simplifications are easy to abstract over the specific domain (such as monoid or ring) where they apply. These simplifications and abstractions help a lot in a large language with more base types and primitive operations. Incidentally, the code actually contains a more general implementation mechanism for such features, inspired in part by previous work in generative linear algebra (Carette and Kiselyov 2005).

Any partial evaluator must decide how much to unfold recursion statically: unfolding too little can degrade the residual code, whereas unfolding too much risks nontermination. Our partial evaluator is no exception, because our object language includes `fix`. The code in Figure 5 takes the naïve approach of "going all the way", that is, unfold `fix` rather than residualize whenever the argument is static. A conservative alternative is to unfold recursion only once, then residualize:

```
let fix f = f (pdyn (C.fix (fun x -> abstr (f (pdyn x)))))
```

Many sophisticated approaches have been developed to decide how much to unfold (Jones et al. 1993), but this issue is orthogonal to our presentation. A separate concern in our treatment of `fix` is possible code bloat in the residual program, which calls for let-insertion (Swadi et al. 2006).

Given this implementation of `P`, our running example

```
let module E = EX(P) in E.test1 ()
```

evaluates to

```
{P.st = Some true; P.dy = .<true>.}
```

of type `('a, bool, bool) P.repr`. Unlike with `C` in §3, a $\beta$-reduction has been statically performed to yield `true`. More interestingly, whereas `testpowfix7` compiles to a code value with many $\beta$-redexes in §3, the partial evaluation

```
let module E = EX(P) in E.testpowfix7
```

gives the desired result

```
{P.st = Some <fun>;
 P.dy = .<fun x -> x * (x * (x * (x * (x * (x * x)))))>.}
```

Unlike the GADT approach in §4.3, all pattern-matching in `P` is *syntactically* exhaustive, so it is patent to the metalanguage implementation that `P` never gets stuck. Further, all pattern-matching occurs during partial evaluation, only to check if a value is known statically, never what type it has. In other words, our partial evaluator tags phases (with `Some` and `None`) but not object types.

Our typed partial evaluator is online and polyvariant. It reuses the compiler `C` and the evaluator `R` by composing them. This situation is simpler than Sperber and Thiemann's (1997) composition of a partial evaluator and a compiler, but the general ideas are similar.

## 5 Continuation-passing style

Our approach accommodates several variants, including a call-by-name CPS interpreter and a call-by-value CPS transformation.

### *5.1 Call-by-name CPS interpreters*

The object language generally inherits the evaluation strategy from the metalanguage—call-by-value (CBV) in OCaml, call-by-name (CBN) in Haskell. To represent a CBN object language in a CBV metalanguage, Reynolds (1972, 1974) and Plotkin (1975) introduce CPS to make the evaluation strategy of a definitional interpreter indifferent to that of the metalanguage. To achieve the same indifference in the typed setting, we build a CBN CPS interpreter for our object language in OCaml.

The interpretation of an object term is a function mapping a continuation `k` to the answer returned by `k`.

```
let int (x:int) = fun k -> k x
let add e1 e2 = fun k -> e1 (fun v1 -> e2 (fun v2 -> k (v1 + v2)))
```

In both `int` and `add`, the interpretation has type `(int -> 'w) -> 'w`, where `'w` is the (polymorphic) answer type.

Unlike CBV CPS, the CBN CPS interprets abstraction and application as follows:

```
let lam f = fun k -> k f
let app e1 e2 = fun k -> e1 (fun f -> f e2 k)
```

Characteristic of CBN, `app e1 e2` does not evaluate the argument `e2` by applying it to the continuation `k`. Rather, it passes `e2` unevaluated to the abstraction. Interpreting $\lambda x. x+1$ yields type

```
((((int -> 'w1) -> 'w1) -> (int -> 'w1) -> 'w1) -> 'w2) -> 'w2
```

We would like to collect those interpretation functions into a module with signature `Symantics`, to include the CBN CPS interpreter within our general framework. Alas, as in §4.2, the type of an object term inductively determines the type of its interpretation: the interpretation of an object term of type $t$ may not have type ($t$`->'w)->'w`, because $t$ may be a function type. Again we simulate a type function with a typecase distinction, using an extra type argument to `repr`. Happily, the type function `repr_pe` needed for the partial evaluator in §4.4 is precisely the same type function we need for CBN CPS.

```
module RCN = struct
  type ('c,'sv,'dv) repr = {ko: 'w. ('sv -> 'w) -> 'w}
  let int (x:int) = {ko = fun k -> k x}
  let add e1 e2 = {ko = fun k ->
      e1.ko (fun v1 -> e2.ko (fun v2 -> k (v1 + v2)))}
  let if_ eb et ee = {ko = fun k ->
      eb.ko (fun vb -> if vb then (et ()).ko k else (ee ()).ko k)}
  let lam f = {ko = fun k -> k f}
  let app e1 e2 = {ko = fun k -> e1.ko (fun f -> (f e2).ko k)}
  let fix f = let rec fx f n = app (f (lam (fx f))) n in lam (fx f)
  let run x = x.ko (fun v -> v)
end
```

This interpreter `RCN` is fully polymorphic over the answer type, using higher-rank polymorphism through OCaml record types. It could also be a functor parameterized over the answer type.

Because `RCN` has the signature `Symantics`, we can instantiate our previous examples with it, and all works as expected. More interesting is the example $(\lambda x. 1)\big((\mathrm{fix} f. f)\, 2\big)$, which terminates under CBN but not CBV.

```
module EXS(S: Symantics) = struct open S
 let diverg () = app (lam (fun x -> int 1))
                     (app (fix (fun f->f)) (int 2))
end
```

Interpreting `EXS` with the `R` interpreter of §2.2 does not terminate.

```
let module M = EXS(R) in M.diverg ()
```

In contrast, the CBN interpreter gives the result `1`.

```
let module M = EXS(RCN) in RCN.run (M.diverg ())
```

## *5.2 CBV CPS transformers*

Changing one definition turns our CBN CPS interpreter into CBV.

```
module RCV = struct include RCN
  let lam f = {ko = fun k -> k
      (fun e -> e.ko (fun v -> f {ko = fun k -> k v}))}
end
```

Now an applied abstraction evaluates its argument before proceeding. This approach is in line with Reynolds's (1974), albeit typed. The interpreter `RCV` is useful for CBV evaluation of the object language whether the metalanguage is CBV or CBN.

We turn to a more general approach to CBV CPS: a CPS transformer that turns any implementation of `Symantics` into a CPS interpreter, whether it is an evaluator. This functor on interpreters performs a textbook CPS transformation on the object language.

```
module CPST(S: Symantics) = struct
  let int i = S.lam (fun k -> S.app k (S.int i))
  let add e1 e2 = S.lam (fun k -> S.app e1 (S.lam (fun v1 ->
                                  S.app e2 (S.lam (fun v2 ->
                                  S.app k (S.add v1 v2))))))
  let lam f = S.lam (fun k -> S.app k
              (S.lam (fun x -> f (S.lam (fun k -> S.app k x)))))
  let app e1 e2 = S.lam (fun k -> S.app e1 (S.lam (fun f ->
                                  S.app e2 (S.lam (fun v ->
                                  S.app (S.app f v) k)))))
  let fix = S.fix
end
```

This (abbreviated) code explicitly maps CPS interpretations to (direct) interpretations performed by the base interpreter `S`.

The module returned by `CPST` does not define `repr` and thus does not have signature `Symantics`. The reason is again the type of `lam f`. Whereas `int` and `add` return the (abbreviated) type `('c, ..., (int -> 'w) -> 'w) S.repr`, the type of `lam (add (int 1))` is

```
('c, ..., ((int -> (int -> 'w1) -> 'w1) -> 'w2) -> 'w2) S.repr
```

Hence, to write the type equation defining `CPST.repr` we again need a type function with a typecase distinction, similar to `repr_pe` in §4.4. Alas, the type function we need is not identical to `repr_pe`, so we need to add another type argument to `repr` in the `Symantics` signature. As in §4.4, the terms in previous implementations of `Symantics` stay unchanged, but the `repr` type equations in those implementations have to take a new (phantom) type argument.

$$\frac{}{!\text{state} : t_s} \qquad \frac{e : t_s}{\text{state} \leftarrow e : t_s} \qquad \frac{e_1 : t_1 \quad \begin{array}{c}[x : t_1] \\ \vdots \\ e_2 : t_2\end{array}}{\text{case } e_1 \text{ of } x.\, e_2 : t_2}$$

Fig. 6. Extending our typed object language with mutable state of type $t_s$

To save space, we just use the module returned by CPST as is. Because it does not match the signature Symantics, we cannot apply the EX functor to it. Nevertheless, we can write the tests.

```
module T = struct
  module M = CPST(C)
  open M
  let test1 () = (* same as before *)
       app (lam (fun x -> x)) (bool true)
  let testpowfix () = ... (* same as before *)
  let testpowfix7 = (* same as before *)
       lam (fun x -> app (app (testpowfix ()) x) (int 7))
end
```

We instantiate CPST with the desired base interpreter C, then use the result M to interpret object terms. Those terms are *exactly* as before. Having to textually copy the terms is the price we pay for this simplified treatment. Our discussion of self-interpretation in §6 shows that this copying is not frivolous but represents plugging a term into a context, which is one of the many faces of polymorphism.

With CPST instantiated by the compiler C above, T.test1 gives

```
.<fun x_5 -> (fun x_2 -> x_2 (fun x_3 x_4 -> x_4 x_3))
             (fun x_6 -> (fun x_1 -> x_1 true)
                         (fun x_7 -> x_6 x_7 x_5))>.
```

This output is a naïve CPS transformation of $(\lambda x.x)\,$true, containing several apparent $\beta$-redexes. To reduce these redexes, we just change T to instantiate CPST with P instead.

```
{P.st = Some <fun>; P.dy = .<fun x_5 -> x_5 true>.}
```

### *5.3 State and imperative features*

We can modify a CBN or CBV CPS transformation to pass a piece of state along with the continuation. This technique lets us support mutable state. As Figure 6 shows, we extend our object language with three imperative features.

1. "!state" gets the current state;
2. "state $\leftarrow e$" sets the state to the value of $e$ and returns the previous value of the state;
3. the let-form "case $e_1$ of $x.\,e_2$" evaluates $e_1$ before $e_2$ even if $e_2$ does not use $x$.

If $x$ does not appear in $e_2$, then "case $e_1$ of $x.\,e_2$" is same as the more familiar sequencing form "$e_1; e_2$". We can embed this extended object language into OCaml by extending the Symantics signature in Figure 4.

```
module type SymSI = sig
  include Symantics
  type state
  type 'c states       (* static version of the state *)
  val lapp : (('c,'sa,'da) repr as 'x) -> ('x -> 'y)
             -> (('c,'sb,'db) repr as 'y)
  val deref : unit -> ('c, 'c states, state) repr
  val set   : (('c, 'c states, state) repr as 'x) -> 'x
end
```

In HOAS, we write case $e_1$ of $x.\,e_2$ as `lapp e1 (fun x -> e2)`; the type of `lapp` is that of function application with the two arguments swapped. We can encode the term "case !state of $x.$ state $\leftarrow 2; x+$ !state" as the OCaml functor

```
module EXSI_INT(S: SymSI
  with type state = int and type 'c states = int) = struct open S
  let test1 () = lapp (deref ()) (fun x ->
                   lapp (set (int 2)) (fun _ -> add x (deref ())))
end
```

The accompanying source code shows several more tests, including a test for higher-order state and a power function that uses state as the accumulator.

The state-passing interpreter extends the CBN CPS interpreter `RCN` of §5.1.

```
module RCPS(ST: sig
  type state
  type 'c states
  type ('c,'sv,'dv) repr =
      {ko: 'w. ('sv -> 'c states -> 'w) -> 'c states -> 'w}
end) = struct
  include ST
  type ('c, 'sv, 'dv) result = 'c states -> 'sv
  ...
  let lapp e2 e1 = {ko = fun k ->
      e2.ko (fun v -> (app (lam e1) {ko = fun k -> k v}).ko k)}
  let deref () = {ko = fun k s -> k s s}
  let set e = {ko = fun k -> e.ko (fun v s -> k s v)}
  let get_res x = fun s0 -> x.ko (fun v s -> v) s0
end
```

The implementations of `int`, `app`, `lam`, and so on are *identical* to those of `RCN` and elided. New are the extended type `repr`, which now includes the state, and the functions `lapp`, `deref`, and `set` representing imperative features. The interpreter is still CBN, so evaluating `app ef ea` might not evaluate `ea`, but evaluating `lapp ea ef` always does. For first-order state, such as of type $\mathbb{Z}$, we instantiate the interpreter as

```
module RCPSI = RCPS(struct
  type state = int
```

```
    type 'c states = int
    type ('c,'sv,'dv) repr =
        {ko: 'w. ('sv -> 'c states -> 'w) -> 'c states -> 'w}
end)
```

If the state has a higher-order type, then the types `state` and `'c states` are no longer the same, and `'s states` is mutually recursive with the type `('c,'sv,'dv) repr`, as demonstrated in the accompanying source code.

`RCPSI` matches the `Symantics` signature and implements the unextended object language: we can pass `RCPSI` to the functor `EX` (Fig. 2) and run the example `test1` from there. The main use for `RCPSI` is to interpret the extended object language.

```
module EXPSI_INT = EXSI_INT(RCPSI)
let cpsitesti1 = RCPSI.get_res (EXPSI_INT.test1 ()) 100
val cpsitesti1 : int = 102
```

It is worthwhile reiterating how close this implementation is to the CPS interpreter, and that adding state and imperative features needed no new techniques. We can also add mutable references to the object language using mutable references of the metalanguage, as shown in the accompanying code.

## 6 Self-interpretation

We turn to interpreting the object language in the object language, to clarify how expressive our typed object language can be and to argue that our partial evaluator is Jones-optimal.

Given an *encoding* of each object term $e$ as an *object* term “$e$”, a *self-interpreter* is usually defined as an object function SI such that any object term $e$ is observationally equivalent to the object application SI“$e$” (Danvy and López 2003; Jones et al. 1993; Taha et al. 2001). A partial evaluator PE maps object terms $e$ to observationally equivalent object terms PE$(e)$. It is said to be *optimal* with respect to `si` (Jones 1988) if PE(SI“$e$”) is equal to $e$ (up to $\alpha$-conversion, or in some accounts, no less efficient than $e$).

Intuitively, self-interpretation is straightforward in our framework: the functions comprising the interpreters in §2.2 may just as well be written in our object language. In particular, the following *object* functions implement an evaluator. (We use the number 0 in lieu of a unit value.)

$$\begin{aligned}
\mathit{int} &= \lambda x.\, x \\
\mathit{add} &= \lambda x.\, \lambda y.\, x+y \\
\mathit{if\_} &= \lambda b.\, \lambda t.\, \lambda e.\, \text{if } b \text{ then } t\,0 \text{ else } e\,0 \\
\mathit{lam} &= \lambda f.\, f \\
\mathit{app} &= \lambda f.\, \lambda x.\, f x \\
\mathit{fix} &= \lambda g.\, \text{fix}\, f.\, \lambda x.\, g f x
\end{aligned}$$

We thus map each object terms $e$ to an object term '$e$' as follows. We call this mapping *pre-encoding*.

$$\begin{aligned}
\text{`}x\text{'} &= x \\
\text{`}n\text{'} &= \mathit{int}\, n \\
\text{`}e_1 + e_2\text{'} &= \mathit{add}\,\text{`}e_1\text{'}\,\text{`}e_2\text{'} \\
\text{`if } b \text{ then } t \text{ else } e\text{'} &= \mathit{if\_}\,\text{`}b\text{'}\,(\lambda\_.\,\text{`}t\text{'})\,(\lambda\_.\,\text{`}e\text{'}) \\
\text{`}\lambda x.\,e\text{'} &= \mathit{lam}\,(\lambda x.\,\text{`}e\text{'}) \\
\text{`}f x\text{'} &= \mathit{app}\,\text{`}f\text{'}\,\text{`}x\text{'} \\
\text{`fix}\, f.\,e\text{'} &= \mathit{fix}\,(\lambda f.\,\text{`}e\text{'})
\end{aligned}$$

The metavariables $x$ and $n$ stand for a variable and an integer, respectively. This pre-encoding is just like how we represent object terms in the metalanguage in the preceding sections, but it produces terms in the object language rather than the metalanguage.

To evaluate '$e$', then, we instantiate the free variables in '$e$' such as *int*, *lam*, and *add* by their definitions above. For example, the familiar object term $(\lambda x.x)\,\text{true}$ pre-encodes to

$$\text{`}(\lambda x.x)\,\text{true'} = \mathit{app}\,(\mathit{lam}\,(\lambda x.x))\,(\mathit{bool}\,\text{true}),$$

and to evaluate this pre-encoded term is to evaluate the object term

$$(\lambda f.\,\lambda x.\,fx)\,((\lambda f.\,f)(\lambda x.x))\,((\lambda b.\,b)\,\text{true}).$$

Because the evaluator above mostly consists of glorified identity functions, our simple partial evaluator reduces the result of this instantiation to $e$. In general, to interpret '$e$' using an interpreter is to instantiate its free variables by that interpreter's definitions.

## *6.1 Avoiding higher polymorphism*

Any approach to self-interpretation needs to spell out first how to encode object terms $e$ to object terms "$e$", and then how to interpret "$e$" in the object language. For our approach, we want to define encoding in terms of pre-encoding, and interpretation using some notion of instantiation. Unfortunately, the simple type structure of our object language hinders both tasks. To continue with the example term above, we could try to define

$$\text{``}e\text{''} = \lambda \mathit{app}.\,\lambda \mathit{lam}.\,\lambda \mathit{bool}.\,\text{`}e\text{'}\,,$$

in particular

$$\text{``}(\lambda x.x)\,\text{true''} = \lambda \mathit{app}.\,\lambda \mathit{lam}.\,\lambda \mathit{bool}.\,\mathit{app}\,(\mathit{lam}\,(\lambda x.x))\,(\mathit{bool}\,\text{true}).$$

To type-check this encoded term, we give the variable *lam* the simple type $(\mathbb{B} \to \mathbb{B}) \to \mathbb{B} \to \mathbb{B}$. We then define the self-interpreter

$$\mathrm{SI} = \lambda e.\,e(\lambda f.\,\lambda x.\,fx)(\lambda f.\,f)(\lambda b.\,b)$$

and apply it to the encoded term. The result is the object term

$$\big(\lambda e.\,e(\lambda f.\,\lambda x.\,fx)(\lambda f.\,f)(\lambda b.\,b)\big)\,\big(\lambda \mathit{app}.\,\lambda \mathit{lam}.\,\lambda \mathit{bool}.\,\mathit{app}\,(\mathit{lam}\,(\lambda x.x))\,(\mathit{bool}\,\text{true})\big),$$

which partially evaluates to true easily. However, encoding fails on a term with multiple $\lambda$-abstractions at different types. For example, the pre-encoding

$$\text{‘}\lambda f.\,\lambda x.\,fx\text{’} = \mathit{lam}\,(\lambda f.\,\mathit{lam}\,(\lambda x.\,\mathit{app}\,f\,x))$$

does not type-check in any typing environment, because *lam* needs to take two incompatible types. In sum, we need more polymorphism in the object type system to type *lam*, *app*, *fix*, and *if_* (and “*e*” and SI). (The polytypes in `Symantics` given by Haskell’s type classes and OCaml’s modules supply this polymorphism.) Moreover, we need to encode any polymorphism of the object language *into* the object language to achieve self-interpretation.

### *6.2 Introducing let-bound polymorphism*

Instead of adding higher-rank and higher-kind polymorphism to our object language (along with polymorphism over kinds!), we add let-bound polymorphism. As usual, we can add a new typing rule

$$\frac{e_1 : t_1 \quad e_2\,\{x \mapsto e_1\} : t_2}{\text{let}\; x = e_1 \;\text{in}\; e_2 : t_2}\;.$$

The pre-encoding of a let-expression is trivial.

$$\text{‘let}\; x = e_1 \;\text{in}\; e_2\text{’} \quad = \quad \text{let}\; x = \text{‘}e_1\text{’} \;\text{in}\; \text{‘}e_2\text{’}$$

A *context* is an object term with a hole [ ]. The hole may occur under a binder, so plugging a term into the context may capture free variables of the term. By pre-encoding a hole to a hole, we extend pre-encoding from a translation on terms to one on contexts.

$$\text{‘[\,]’} \quad = \quad [\,]$$

We define an interpreter in the object language to be not a term but a context. For example, the evaluator is the context

$$\begin{array}{rll}
\mathrm{SI}[\,] = \text{let } \mathit{int} & = \lambda x.\,x & \text{in}\\
\text{let } \mathit{add} & = \lambda x.\,\lambda y.\,x+y & \text{in}\\
\text{let } \mathit{if\_} & = \lambda b.\,\lambda t.\,\lambda e.\,\text{if } b \text{ then } t\,0 \text{ else } e\,0 & \text{in}\\
\text{let } \mathit{lam} & = \lambda f.\,f & \text{in}\\
\text{let } \mathit{app} & = \lambda f.\,\lambda x.\,fx & \text{in}\\
\text{let } \mathit{fix} & = \lambda g.\,\text{fix}\,f.\,\lambda x.\,gfx & \text{in } [\,],
\end{array}$$

and the size-measurer at the end of §2.2 is the context

$$\begin{array}{rll}
\mathrm{SZ}[\,] = \text{let } \mathit{int} & = \lambda x.\,1 & \text{in}\\
\text{let } \mathit{add} & = \lambda x.\,\lambda y.\,x+y+1 & \text{in}\\
\text{let } \mathit{if\_} & = b+t\,0+e\,0 & \text{in}\\
\text{let } \mathit{lam} & = \lambda f.\,f\,0+1 & \text{in}\\
\text{let } \mathit{app} & = \lambda f.\,\lambda x.\,f+x+1 & \text{in}\\
\text{let } \mathit{fix} & = \lambda g.\,g\,0+1 & \text{in } [\,].
\end{array}$$

To interpret an object term $e$ using an interpreter $I[\,]$ is to evaluate the object term $I[\text{‘}e\text{’}]$. SI is a self-interpreter in the following sense.

*Proposition 5*
SI[‘$e$’] is observationally equivalent to $e$.

As a corollary, we can pre-encode SI itself as a context: the term SI[‘SI[‘$e$’]’] is observationally equivalent to SI[‘$e$’], and in turn to $e$. In other words, SI can interpret itself. Our partial evaluator is optimal with respect to the self-interpreter SI.

*Proposition 6*
Let PE be the partial evaluator `P` in §4.4. Then the object terms PE(SI[‘$e$’]) and PE($e$) are either both undefined or both defined and equal up to $\alpha$-conversion.

### 6.3 Contexts clarify polymorphism

We always type-check a pre-encoded term ‘$e$’ and its interpreter $I[\,]$ together, never separately. This treatment has the drawback that we must duplicate a pre-encoded term in order to interpret it in multiple ways. The meta-notions of contexts and plugging may seem ad hoc, but in fact they just reflect the type-class and module machinery that we have been using in this paper all along.

In the presence of let-bound polymorphism, we can understand a term waiting to be plugged into a context as a higher-rank and higher-kind abstraction over the context. Even though our object language does not support higher abstraction, our metalanguages do, so they can type-check an object term separately from its interpreter—either as a functor from a `Symantics` module containing a type constructor (in OCaml), or a value with a `Symantics` constraint over a type constructor (in Haskell). Thus, “context” is a euphemism for a polymorphic argument, and “plugging” is a euphemism for application.

## 7 Related work

Our initial motivation came from several papers that justify advanced type systems, in particular GADTs, by embedded interpreters (Pašalić et al. 2002; Peyton Jones et al. 2006; Taha et al. 2001; Xi et al. 2003) and CPS transformations (Chen and Xi 2003; Guillemette and Monnier 2006; Shao et al. 2005). We admire all this technical machinery, but these motivating examples do not need it. Although GADTs may indeed be more flexible and easier to use, they are unavailable in mainstream ML and implemented problematically in GHC currently. We also wanted to find the minimal set of widespread language features needed for tagless type-preserving interpretation.

Even a simply typed $\lambda$-calculus obviously supports self-interpretation, provided we use universal types (Taha et al. 2001). The ensuing tagging overhead motivated Taha et al. (2001) to propose tag elimination, which however does not statically guarantee that all tags will be removed (Pašalić et al. 2002).

Pašalić et al. (2002), Taha et al. (2001), Xi et al. (2003), and Peyton Jones et al. (2006) seem to argue as follows that a self-interpreter of a typed language cannot be tagless or Jones-optimal:

1. One needs to encode a typed language in a typed language based on a sum type (at some level of the hierarchy).

2. A *direct* interpreter for such an encoding of a typed language in a typed language requires either advanced types or tagging overhead.
3. Thus, an indirect interpreter is necessary, which needs a universal type and hence tagging.
4. Thus, any self-interpreter must have tags and cannot be Jones-optimal.

While the logic is sound, we showed that the premise in the very first step is not valid.

Danvy and López (2003) discuss Jones optimality at length and apply HOAS to typed self-interpretation. However, their source language is untyped. Therefore, their object-term encoding has tags, and their interpreter can raise run-time errors. Nevertheless, HOAS lets the partial evaluator remove all the tags. In contrast, our object encoding and interpreters do not have tags to start with and obviously cannot raise run-time errors.

A lot of effort has gone into "typing dynamic typing": to statically type-check dynamically-typed values (Baars and Swierstra 2002; Guillemette and Monnier 2006; Pašalić et al. 2002; Visser and Vytiniotis 2006), using the host language's type system to varying extents. Our object terms are statically typed, so we would need one of these techniques to interpret dynamically-typed terms such as those read from a file.

Our partial evaluator establishes a bijection `repr_pe` between static and dynamic types (the valid values of `'sv` and `'dv`), and between static and dynamic terms. It is customary to implement such a bijection using an injection-projection pair, as done for interpreters by Ramsey (2005) and Benton (2005), partial evaluation by Danvy (1996), and type-level functions by Oliveira and Gibbons (2005). As explained in §4.4, we avoid injection and projection at the type level by adding an argument to `repr`. Our solution could have been even more straightforward if MetaOCaml provided total type-level functions such as `repr_pe` in §4.4—simple type-level computations ought to become mainstream.

At the term level, we also avoid converting between static and dynamic terms by building them in parallel, using Asai's method (2001). This method type-checks in Hindley-Milner once we deforest the object term representation. Put another way, we manual apply type-level partial evaluation to our type functions (see §4.4) to obtain simpler types acceptable to MetaOCaml. Sumii and Kobayashi (2001) also use Asai's method, to combine online and offline partial evaluation. We strive for modularity by reusing interpreters for individual stages (Sperber and Thiemann 1997). It would be interesting to try to derive a *cogen* (Thiemann 1996) in the same manner.

It is common to implement an embedded DSL by providing multiple interpretations of host-language pervasives such as addition and application. It is also common to use phantom types to rule out ill-typed object terms, as done in Lava (Bjesse et al. 1998) and by Rhiger (2001). However, these approaches are not tagless because they still use universal types, such as Lava's `Bit` and `NumSig`, and Rhiger's `Raw` (his Fig. 2.2) and `Term` (his Chap. 3), which incur the attendant overhead of pattern matching. The universal type also greatly complicates the soundness and completeness proofs of embedding (Rhiger 2001), whereas our proofs are trivial. Rhiger's approach does not support typed CPS transformation (his §3.3.4).

We are not the first to implement a typed interpreter for a typed language. Läufer and Odersky (1993) use type classes to implement a metacircular interpreter (rather than a self-interpreter) of a typed version of the SK language, which is quite different from our object language. Their interpreter appears to be tagless, but they could not have implemented a compiler or partial evaluator in the same way, since they rely heavily on injection-projection pairs.

Fiore (2002) and Balat et al. (2004) also build a tagless partial evaluator, using delimited control operators. It is type-directed, so the user must represent, as a term, the type of every term to be partially evaluated. We shift this work to the type checker of the metalanguage. By avoiding term-level type representations, our approach makes it easier to perform algebraic simplifications (as in §4.4).

Using Haskell, Guillemette and Monnier (2006) implement a CPS transformation for HOAS terms and statically assure that it preserves object types. They represent proofs of type preservation as terms of a GADT, which is not sound (as they admit in §4.2) without a separate totality check because any type is trivially inhabited by a nonterminating term in Haskell. In contrast, our CPS transformations use simpler types than GADTs and assure type preservation at the (terminating) type level rather than the term level of the metalanguage. Guillemette and Monnier review other type-preserving CPS transformations (mainly in the context of typed intermediate languages), in particular Shao et al.'s (2005) and Chen and Xi's (2003). These approaches use de Bruijn indices and fancier type systems with type-level functions, GADTs, or type-equality proofs.

We encode terms in elimination form, as a coalgebraic structure. Pfenning and Lee (1991) first described this basic idea and applied it to metacircular interpretation. Our approach, however, can be implemented in mainstream ML and supports type inference, typed CPS transformation and partial evaluation. In contrast, Pfenning and Lee conclude that partial evaluation and program transformations "do not seem to be expressible" even using their extension to $F_\omega$, perhaps because their avoidance of general recursive types compels them to include the polymorphic lift that we avoid in §4.1.

Our encoding of the type function `repr_pe` in §4.4 emulates type-indexed types and is related to intensional type analysis (Harper and Morrisett 1995; Hinze et al. 2004). However, our object language and running examples in HOAS include `fix`, which intensional type analysis cannot handle (Xi et al. 2003). Our final approach seems related to Washburn and Weirich's approach to HOAS using catamorphisms and anamorphisms (2003).

We could not find work that establishes that the *typed* $\lambda$-calculus has a final coalgebra structure. Honsell and Lenisa (1995, 1999) investigate the untyped $\lambda$-calculus along this line. In particular, they use contexts with a hole (Honsell and Lenisa 1999; p. 13) to define *observational equivalence* (see our §6). Honsell and Lenisa's bibliography (1999) refers to the foundational work in this important area. Particularly intriguing is the link to the coinductive aspects of Böhm trees, as pointed out by Berarducci (1996) and Jacobs (2007; Example 4.3.4).

One way to understand our main idea is to eschew sum types and sum kinds for their dual, record types and record kinds. For the self-interpreter, we then proceed to use a Church encoding for recursive data types (Böhm and Berarducci 1985).

As §6.3 observes, higher-rank and higher-kind polymorphism lets us type-check and compile object terms separately from interpreters. This observation is consistent with the role of polymorphism in the separate compilation of modules (Shao 1998).

## 8 Conclusions

We solve the problem of embedding a typed object language in a typed metalanguage without using GADTs, dependent types, or a universal type. Our family of interpreters include an evaluator, a compiler, a partial evaluator, and CPS transformers. It is patent that they never get stuck, because we represent object types as metalanguage types. This work makes it safer and more efficient to embed DSLs in practical metalanguages such as Haskell and ML.

Our main idea is to represent object programs not in an initial algebra but using the existing coalgebraic structure of the $\lambda$-calculus. More generally, to squeeze more invariants out of a type system as simple as Hindley-Milner, we shift the burden of representation and computation from consumers to producers: encoding object terms as calls to metalanguage functions (§1.3); build dynamic terms alongside static ones (§4.1); simulating type functions for partial evaluation (§4.4) and CPS transformation (§5.1). This shift also underlies fusion, functionalization, and amortized complexity analysis. When the metalanguage does provide higher-rank and higher-kind polymorphism, we can type-check and compile an object term separately from any interpreters it may be plugged into.

Our representation of object terms in elimination form encodes primitive recursive folds over the terms. This encoding makes operations like interpretation trivial to implement. We still have to understand if and how non-primitively recursive operations can be supported.

## Acknowledgments

We thank Martin Sulzmann and Walid Taha for helpful discussions. Sam Staton, Pieter Hofstra, and Bart Jacobs kindly provided some useful references. We thank anonymous reviewers for pointers to related work.

## References

Asai, Kenichi. 2001. Binding-time analysis for both static and dynamic expressions. *New Generation Computing* 20(1):27–52.

Baars, Arthur I., and S. Doaitse Swierstra. 2002. Typing dynamic typing. In *ICFP '02: Proceedings of the ACM international conference on functional programming*, 157–166. New York: ACM Press.

Balat, Vincent, Roberto Di Cosmo, and Marcelo P. Fiore. 2004. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL '04: Conference record of the annual ACM symposium on principles of programming languages*, 64–76. New York: ACM Press.

Benton, P. Nick. 2005. Embedded interpreters. *Journal of Functional Programming* 15(4): 503–542.

Berarducci, Alessandro. 1996. Infinite lambda-calculus and non-sensible models. In *Logic and algebra*, ed. A. Ursini and P. Aglianò, vol. 180, 339–378. Marcel Dekker.

Bjesse, Per, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware design in Haskell. In *ICFP '98: Proceedings of the ACM international conference on functional programming*, vol. 34(1) of *ACM SIGPLAN Notices*, 174–184. New York: ACM Press.

Böhm, Corrado, and Alessandro Berarducci. 1985. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science* 39:135–154.

Calcagno, Cristiano, Eugenio Moggi, and Walid Taha. 2004. ML-like inference for classifiers. In *Programming languages and systems: Proceedings of ESOP 2004, 13th European symposium on programming*, ed. David A. Schmidt, 79–93. Lecture Notes in Computer Science 2986, Berlin: Springer-Verlag.

Carette, Jacques, and Oleg Kiselyov. 2005. Multi-stage programming with Functors and Monads: eliminating abstraction overhead from generic code. In *Generative programming and component-based engineering GPCE*, 256–274.

Chen, Chiyan, and Hongwei Xi. 2003. Implementing typeful program transformations. In *Proceedings of the 2003 ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation*, 20–28. New York: ACM Press.

Danvy, Olivier. 1996. Type-directed partial evaluation. In *POPL '96: Conference record of the annual ACM symposium on principles of programming languages*, 242–257. New York: ACM Press.

Danvy, Olivier, and Pablo E. Martínez López. 2003. Tagging, encoding, and Jones optimality. In *Programming languages and systems: Proceedings of ESOP 2003, 12th European symposium on programming*, ed. Pierpaolo Degano, 335–347. Lecture Notes in Computer Science 2618, Berlin: Springer-Verlag.

Davies, Rowan, and Frank Pfenning. 2001. A modal analysis of staged computation. *Journal of the ACM* 48(3):555–604.

Fiore, Marcelo P. 2002. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th international conference on principles and practice of declarative programming*, 26–37. New York: ACM Press.

Fogarty, Seth, Emir Pasalic, Jeremy Siek, and Walid Taha. 2007. Concoqtion: Indexed types now! In *Proceedings of the 2007 ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation*. New York: ACM Press.

Glück, Robert. 2002. Jones optimality, binding-time improvements, and the strength of program specializers. In *ASIA-PEPM '02: Proceedings of the ASIAN symposium on partial evaluation and semantics-based program manipulation*, 9–19. New York: ACM Press.

Guillemette, Louis-Julien, and Stefan Monnier. 2006. Statically verified type-preserving code transformations in Haskell. In *PLPV 2006: Programming languages meets program verification*, ed. Aaron Stump and Hongwei Xi, 40–53. Electronic Notes in Theoretical Computer Science 174(7), Amsterdam: Elsevier Science.

Harper, Robert, and J. Gregory Morrisett. 1995. Compiling polymorphism using intensional type analysis. In *POPL '95: Conference record of the annual ACM symposium on principles of programming languages*, 130–141. New York: ACM Press.

Hindley, J. Roger. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society* 146:29–60.

Hinze, Ralf, Johan Jeuring, and Andres Löh. 2004. Type-indexed data types. *Science of Computer Programming* 51(1-2):117–151.

Honsell, Furio, and Marina Lenisa. 1995. Final semantics for untyped lambda-calculus. In *TLCA '95: Proceedings of the 2nd international conference on typed lambda calculi and applications*, ed. Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, 249–265.

Lecture Notes in Computer Science 902, Berlin: Springer-Verlag.

———. 1999. Coinductive characterizations of applicative structures. *Mathematical Structures in Computer Science* 9(4):403–435.

Hudak, Paul. 1996. Building domain-specific embedded languages. *ACM Computing Surveys* 28(4es):196.

Jacobs, Bart. 2007. Introduction to coalgebra: Towards mathematics of states and observations. `http://www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf`. Draft book.

Jones, Neil D. 1988. Challenging problems in partial evaluation and mixed computation. *New Generation Computing* 6(2–3):291–302.

Jones, Neil D., Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Englewood Cliffs, NJ: Prentice-Hall.

Läufer, Konstantin, and Martin Odersky. 1993. Self-interpretation and reflection in a statically typed language. In *Proceedings of the 4th annual OOPSLA/ECOOP workshop on object-oriented reflection and metalevel architectures*.

Miller, Dale, and Gopalan Nadathur. 1987. A logic programming approach to manipulating formulas and programs. In *IEEE symposium on logic programming*, ed. Seif Haridi, 379–388. Washington, DC: IEEE Computer Society Press.

Milner, Robin. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17:348–375.

Nanevski, Aleksandar. 2002. Meta-programming with names and necessity. In *ICFP '02: Proceedings of the ACM international conference on functional programming*, 206–217. New York: ACM Press.

Nanevski, Aleksandar, and Frank Pfenning. 2005. Staged computation with names and necessity. *Journal of Functional Programming* 15(6):893–939.

Nanevski, Aleksandar, Frank Pfenning, and Brigitte Pientka. 2007. Contextual modal type theory. *Transactions on Computational Logic*. To appear.

Oliveira, Bruno César dos Santos, and Jeremy Gibbons. 2005. TypeCase: A design pattern for type-indexed functions. In *Proceedings of the 2005 Haskell workshop*, 98–109. New York: ACM Press.

Pašalić, Emir, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *ICFP '02: Proceedings of the ACM international conference on functional programming*, 157–166. New York: ACM Press.

Peyton Jones, Simon L., Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the ACM international conference on functional programming*, 50–61. New York: ACM Press.

Pfenning, Frank, and Conal Elliott. 1988. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM conference on programming language design and implementation*, vol. 23(7) of *ACM SIGPLAN Notices*, 199–208. New York: ACM Press.

Pfenning, Frank, and Peter Lee. 1991. Metacircularity in the polymorphic $\lambda$-calculus. *Theoretical Computer Science* 89(1):137–159.

Plotkin, Gordon D. 1975. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science* 1(2):125–159.

Ramsey, Norman. 2005. ML module mania: A type-safe, separately compiled, extensible interpreter. In *Proceedings of the 2005 workshop on ML*. Electronic Notes in Theoretical Computer Science, Amsterdam: Elsevier Science.

Reynolds, John C. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM national conference*, vol. 2, 717–740. New York: ACM Press. Reprinted with a foreword in *Higher-Order and Symbolic Computation* 11(4): 363–397.

———. 1974. On the relation between direct and continuation semantics. In *Automata, languages and programming: 2nd colloquium*, ed. Jacques Loeckx, 141–156. Lecture Notes in Computer Science 14, Berlin: Springer-Verlag.

Rhiger, Morten. 2001. Higher-Order program generation. Ph.D. thesis, BRICS Ph.D. School. Department of Computer Science, University of Aarhus, Denmark.

Shao, Zhong. 1998. Typed cross-module compilation. In *ICFP '98: Proceedings of the ACM international conference on functional programming*, vol. 34(1) of *ACM SIGPLAN Notices*, 141–152. New York: ACM Press.

Shao, Zhong, Valery Trifonov, Bratin Saha, and Nikolaos S. Papaspyrou. 2005. A type system for certified binaries. *ACM Transactions on Programming Languages and Systems* 27(1):1–45.

Sperber, Michael, and Peter Thiemann. 1997. Two for the price of one: Composing partial evaluation and compilation. In *PLDI '97: Proceedings of the ACM conference on programming language design and implementation*, 215–225. New York: ACM Press.

Sumii, Eijiro, and Naoki Kobayashi. 2001. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation* 14(2–3):101–142.

Swadi, Kedar, Walid Taha, Oleg Kiselyov, and Emir Pasalic. 2006. A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of the 2006 ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation*, 160–169. New York: ACM Press.

Taha, Walid, Henning Makholm, and John Hughes. 2001. Tag elimination and Jones-optimality. In *Proceedings of PADO 2001: 2nd symposium on programs as data objects*, ed. Olivier Danvy and Andrzej Filinski, 257–275. Lecture Notes in Computer Science 2053, Berlin: Springer-Verlag.

Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 26–37. New York: ACM Press.

Thiemann, Peter. 1996. Cogen in six lines. In *ICFP '96: Proceedings of the ACM international conference on functional programming*, vol. 31(6) of *ACM SIGPLAN Notices*, 180–189. New York: ACM Press.

Visser, Joost, and Dimitrios Vytiniotis. 2006. Simple GADT parser for the eval example. Haskell-Cafe mailing list. `http://www.haskell.org/pipermail/haskell-cafe/2006-October/019160.html` `http://www.haskell.org/pipermail/haskell-cafe/2006-October/019161.html`.

Washburn, Geoffrey, and Stephanie Weirich. 2003. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *ICFP '03: Proceedings of the ACM international conference on functional programming*, vol. 38(9) of *ACM SIGPLAN Notices*, 249–262. New York: ACM Press.

Xi, Hongwei, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *POPL '03: Conference record of the annual ACM symposium on principles of programming languages*, 224–235. New York: ACM Press.

Yang, Zhe. 1998. Encoding types in ML-like languages. In *ICFP '98: Proceedings of the ACM international conference on functional programming*, vol. 34(1) of *ACM SIGPLAN Notices*, 289–300. New York: ACM Press.