# *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 principal technique is to encode de Bruijn or higher-order abstract syntax using combinator 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 uniformly over the family of ways to interpret it, yet statically assures that the interpreters never get stuck. This family of interpreters thus demonstrates again that it is useful to abstract over higher-kinded types.

*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 (Landin 1966). Embedding means to represent terms and values of the *object language* as terms and values in the *metalanguage*, so as to interpret the former in the latter (Reynolds 1972). Embedding is especially appropriate for domain-specific languages (DSLs) because it supports rapid prototyping and integration with the host environment (Hudak 1996).

Most interpreters suffer from various kinds of overhead, making it less efficient to run object programs via the metalanguage than to implement the object language directly on the machine running the metalanguage (Jones et al. 1993). Two major sources of overhead are dispatching on the syntax of object terms and tagging the types of object values. If the metalanguage supports code generation (Bawden 1999; Gomard and Jones 1991; Nielson and Nielson 1988, 1992; Taha 1999), then the embedding can avoid the dispatching overhead by compiling object programs, that is, by specializing an interpreter to object programs (Futamura 1971). Specializing an interpreter is thus a promising way to build a DSL. However, the tagging overhead remains, especially if the object language and the metalanguage both have a sound type system. The quest to remove all interpretive

$$\frac{\begin{array}{c}[x:t_1]\\ \vdots\\ e:t_2\end{array}}{\lambda x.\,e:t_1\to t_2} \qquad \frac{\begin{array}{c}[f:t_1\to t_2]\\ \vdots\\ e:t_1\to t_2\end{array}}{\mathrm{fix}\,f.\,e:t_1\to t_2} \qquad \frac{e_0:t_1\to t \quad e_1:t_1}{e_0 e_1:t} \qquad \frac{n \text{ is an integer}}{n:\mathbb{Z}} \qquad \frac{b \text{ is a boolean}}{b:\mathbb{B}}$$

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

Fig. 1. Our typed object language

overhead, in particular by specializing the interpreter using a *Jones-optimal* partial evaluator (Jones et al. 1993), has motivated much work on typed specialization (Birkedal and Welinder 1993; Danvy 1998; Danvy and López 2003; Hughes 1998; Makholm 2000; Taha et al. 2001) and type systems (see §1.2).

This paper shows how to eliminate tagging overhead, whether in the context of code generation and whether in the presence of dispatching overhead. We use metalanguage types, without such fancy features as generalized algebraic data types (GADTs) or dependent types, to rule out ill-typed object terms statically, thus speeding up interpretation and assuring that our interpreters do not get stuck. We illustrate the problem of tagging overhead in this section using a simple evaluator as example. We apply our solution first to evaluation, then to code-generation tasks such as partial evaluation.

### *1.1 The tag problem*

To be concrete, we use the typed object language in Figure 1 throughout this paper. 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.[1] For example, we represent the object term $(\lambda x.x)\,\mathrm{true}$ as

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

Let us 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)
```

[1] We use de Bruijn indices to simplify the comparison with Pašalić et al.'s work (2002).

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. Code generation can be used to remove this interpretive overhead (Futamura 1971; Jones et al. 1993; Pašalić et al. 2002).

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 containing 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.

Although `eval` never fails on well-typed closed terms, 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`. 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 and impair optimality in partial evaluation (Jones et al. 1993; Taha et al. 2001). In short, the universal-type solution is unsatisfactory because it does not preserve the type of the encoded term.

### 1.2 Solutions using fancier types

It is commonly thought that the type-preserving interpretation of a typed object language in a typed metalanguage 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), in order for the metalanguage's type system to allow the well-typed object term `test1` but disallow the ill-typed object term `test2`. Yet other fancy 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.

### 1.3 Our final proposal

Following an old idea of Reynolds (1975), we represent object programs using ordinary functions rather than data constructors. These functions comprise the entire interpreter:

```
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 $(\lambda x.x)\,\mathrm{true}$ 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, using ordinary functions rather than data constructors to represent well-typed terms, we achieve a tagless evaluator for a typed object language in a metalanguage with a simple type system (Hindley 1969; Milner 1978). We call this approach *final* (in contrast to *initial*), because we represent each object term not by its abstract syntax but by its denotation in a semantic algebra. This representation makes it trivial to implement a primitive

recursive function over object terms, such as an evaluator. Or, as a referee puts it aptly, our proposal is "a way to write a typed fold function over a typed term."

We emphasize "typed" and "fold" in the previous sentence. We use a typed version of Mogensen's (1995) encoding of the recursive type of terms (Böhm and Berarducci 1985), which makes it much easier to write folds over terms than term functions that are not primitive recursive (or, compositional). In contrast, Mogensen's earlier encoding of the sum type of terms (1992) does not privilege folds. In exchange, we statically express object types in the metalanguage and prevent both kinds of run-time errors in §1.1, due to evaluating ill-typed or open terms. Because the new interpreter uses no universal type or pattern matching, it never gives 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) that arises from error checking at run time.

Our solution does *not* involve 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 functions such as `b`, `lam`, and `app`, whose names appear free in `testf1` above. In the rest of this paper, we explain how to abstract over these functions' definitions and apply different folds to the same object language, so as to process the same term using many other interpreters: we can

- evaluate the term to a value in the metalanguage;
- measure the length 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 in a call-by-value (CBV) metalanguage, so as to isolate the evaluation order of the object language from that of the metalanguage.

We have programmed all our interpreters and examples in OCaml (and, for staging, MetaOCaml) and standard Haskell. The complete code is available at `http://okmij.org/ftp/tagless-final/` to supplement the paper. Except for the basic definitions in §2.1, we show our examples in (Meta)OCaml even though some of our claims are more obvious in Haskell, for consistency and because MetaOCaml provides convenient, typed staging facilities.

### *1.4 Contributions*

We attack the problem of tagless (staged) type-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 the first *family* of interpreters, each instantiating the *same* signature, that evaluate (§2), compile (§3), and partially evaluate (§4) a typed higher-order object language in a typed metalanguage, in direct and continuation-passing styles (§5).

2. These interpreters use no type tags and need no advanced type-system features such as GADTs, dependent types, or intensional type analysis. Yet the type system of the metalanguage assures statically that each object program is well-typed and closed, and that each interpreter preserves types and never gets stuck. In particular, our (online) partial evaluator and CPS transformers avoid GADTs in their implementation and stay portable across Haskell 98 and ML, by expressing in their interface an inductive map from input types to output types.

3. Our clean, comparable implementations using OCaml modules and Haskell type classes show how to parametrize our final representation of object terms over multiple ways to assign them meanings.

4. We point a clear way to extend the object language with more features such as state (§5.4). Our term encoding is contravariant in the object language, so extending the language does not invalidate terms already encoded.

5. We show how to use higher-kinded abstraction to build embedded DSLs.

Our code is surprisingly simple and obvious in hindsight, but it has been cited as a difficult problem (Sumii and Kobayashi (2001) and Thiemann (1999) notwithstanding) 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 2004) has given a hint of how such a function can be expressed." We discuss related work in §6.

To reiterate, we do *not* propose any new language feature or even any new programming technique. Rather, we solve a problem that was stated in the published record as open and likely unsolvable in ML or Haskell 98 without extensions, by a novel combination of simple types and techniques already described in the literature that use features present in mainstream functional languages. In particular, we follow Yang's (2004) encoding of type-indexed values, Sperber's (1996) and Asai's (2001) construction of dynamic terms alongside static terms, and Thiemann's (1999) deforestation of syntax constructors. These techniques require just a Hindley-Milner type system with either module functors or constructor classes, as realized in all variants of ML and Haskell. The simplicity of our solution and its use of only mainstream features are virtues that make it more practical to build typed, embedded DSLs.

However we represent an object term, the representation can be created either by hand (for example, by entering object terms at a metalanguage interpreter's prompt) or by parsing and type-checking text. It is known how to write such a type checker for a higher-order object language such as ours, whether using fancy types (Guillemette and Monnier 2006; Pašalić et al. 2002) or not (Baars and Swierstra 2002). We have ourselves implemented a type checker for our object language (in the accompanying source file `IncopeTypecheck.hs`), which maps an ordinary syntax tree to (either a type error or) a finally encoded object term that can then be interpreted in multiple ways without repeated type-checking. We leave this problem aside in the rest of this paper.

## 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 similar to Plotkin's PCF (1977). It is also close to Xi et al.'s (2003), without their polymorphic lift but with more constants so as to more conveniently express examples. In contrast to §1, in the rest of the paper we use higher-order abstract syntax (HOAS) (Miller and Nadathur 1987; Pfenning and Elliott 1988) rather than de Bruijn indices to encode binding and ensure that our object programs are closed. We find HOAS to be more convenient, but we have also implemented our approach using de Bruijn indices (in §2.3 and the accompanying source file `incope-dB.ml`).

### *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 $(\lambda x.x)\,\mathrm{true}$, 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 elide all but one method of each group. The accompanying code has the complete implementations.

```
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

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 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. We shorten some of the types using OCaml's `as` syntax.

The functor `EX` in Figure 3 encodes our running examples `test1` and the *power* function (`testpowfix`). The dummy argument to `test1` and `testpowfix` is an artifact of MetaOCaml: 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), so we must define our object terms as syntactic values to satisfy the value restriction. (Alternatively, we could have used OCaml's rank-2 record types to maintain the necessary polymorphism.)

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

Comparing `Symantics` with Figure 1 shows how to represent *every* well-typed object term in the metalanguage. We formalize this representation by defining $H$ and $M$, two inductive maps from terms and types in our object language to terms and types in Haskell

and OCaml:

$$
\begin{array}{rl}
H(\mathbb{Z}) = \texttt{Int} & M(\mathbb{Z}) = \texttt{int} \\
H(\mathbb{B}) = \texttt{Bool} & M(\mathbb{B}) = \texttt{bool} \\
H(t_1 \to t_2) = H(t_1)\ \texttt{->}\ H(t_2) & M(t_1 \to t_2) = M(t_1)\ \texttt{->}\ M(t_2) \\
H(x) = x & M(x) = x \\
H(\lambda x.e) = \texttt{lam (\textbackslash}x\ \texttt{->}\ H(e)\texttt{)} & M(\lambda x.e) = \texttt{lam (fun}\ x\ \texttt{->}\ M(e)\texttt{)} \\
H(\mathrm{fix}\, f.e) = \texttt{fix (\textbackslash}x\ \texttt{->}\ H(e)\texttt{)} & M(\mathrm{fix}\, x.e) = \texttt{fix (fun}\ x\ \texttt{->}\ M(e)\texttt{)} \\
H(e_1 e_2) = \texttt{app}\ H(e_1)\ H(e_2) & M(e_1 e_2) = \texttt{app}\ M(e_1)\ M(e_2) \\
H(n) = \texttt{int}\ n & M(n) = \texttt{int}\ n \\
H(\mathrm{true}) = \texttt{bool True} & M(\mathrm{true}) = \texttt{bool true} \\
H(\mathrm{false}) = \texttt{bool False} & M(\mathrm{false}) = \texttt{bool false}
\end{array} \tag{1}
$$

$$
\begin{array}{rl}
H(\text{if } e \text{ then } e_1 \text{ else } e_2) = \texttt{if\_}\ H(e)\ H(e_1)\ H(e_2) & \\
M(\text{if } e \text{ then } e_1 \text{ else } e_2) = \texttt{if\_}\ M(e)\ \texttt{(fun () ->}\ M(e_1)\texttt{)}\ \texttt{(fun () ->}\ M(e_2)\texttt{)} & \\
H(e_1 + e_2) = \texttt{add}\ H(e_1)\ H(e_2) & M(e_1 + e_2) = \texttt{add}\ M(e_1)\ M(e_2) \\
H(e_1 \times e_2) = \texttt{mul}\ H(e_1)\ H(e_2) & M(e_1 \times e_2) = \texttt{mul}\ M(e_1)\ M(e_2) \\
H(e_1 \le e_2) = \texttt{leq}\ H(e_1)\ H(e_2) & M(e_1 \le e_2) = \texttt{leq}\ M(e_1)\ M(e_2)
\end{array} \tag{2}
$$

These definitions assume that our object language, Haskell, and OCaml use the same variable names $x$ and integer literals $n$. If $\Gamma$ is a typing context $x_1 : t_1, \ldots, x_n : t_n$ in the object language, then we define the metalanguage contexts

$$\texttt{repr}\ H(\Gamma) = x_1 : \texttt{repr}\ H(t_1), \ldots, x_n : \texttt{repr}\ H(t_n), \tag{3}$$

$$\texttt{('c, }M(\Gamma)\texttt{) repr} = x_1 : \texttt{('c, }M(t_1)\texttt{) repr}, \ldots, x_n : \texttt{('c, }M(t_n)\texttt{) repr}. \tag{4}$$

The following proposition states the trivial but fundamental fact that this representation preserves types.

*Proposition 1*
If an object term $e$ has the type $t$ in the context $\Gamma$, then the Haskell term $H(e)$ has the type $\texttt{repr}\ H(t)$ in the context

$$\texttt{repr} : \star \to \star, \quad \texttt{Symantics repr}, \quad \texttt{repr}\, H(\Gamma),$$

and the OCaml term $M(e)$ has the type $\texttt{('c, }M(t)\texttt{) repr}$ in the context

$$\texttt{S:Symantics}, \quad \texttt{open S}, \quad \texttt{'c:}\star, \quad \texttt{('c, }M(\Gamma)\texttt{) repr}.$$

*Proof*
By structural induction on the derivation in the object language that $e$ has type $t$ in $\Gamma$. □

*Corollary 2*
If a closed object term $e$ has the type $t$, then the Haskell term $H(e)$ has the type

$$\texttt{forall repr. Symantics repr => repr}\ H(t)$$

and the OCaml functor

$$\texttt{functor (S:Symantics) -> struct open S let term () =}\ M(e)\ \texttt{end}$$

has the signature

`functor (S:Symantics) -> sig val term: unit -> ('c, `$M(t)$`) S.repr end`.

Conversely, the type system of the metalanguage checks that the represented object term is well-typed and closed. If we err, say replace `int 7` with `bool True` in `testpowfix7`, the type checker 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. Both Haskell's and MetaOCaml's type checkers also flag syntactically invalid object terms, such as if we forget `app` somewhere above. Because our encoding of terms and types are so straightforward and metacircular, these error messages from the metalanguage implementation are just about as readable as those for "native" type errors such as `fun x -> x x`.

### *2.2 Two tagless interpreters*

Now that our term representation is independent of any particular interpreter, we are 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 OCaml.

The first interpreter evaluates an object term to its value in the metalanguage. The module `R` below is metacircular in that it *runs* each object-language operation by executing 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  e1 e2    = e1 <= e2
  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`. 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`. The soundness of the object language's type system with respect to the dynamic semantics specified by a definitional interpreter follows from the soundness of the metalanguage's type system.

*Proposition 3*
If a closed object term $e$ has type $t$, and the OCaml module `I` implements the signature `Symantics`, then under the OCaml module definition

```
module RESULT =
  (functor (S:Symantics) -> struct open S let term () = M(e) end)
  (I)
```

evaluating the expression `RESULT.term ()` never gets stuck: it either does not terminate or evaluates to a value of type `('c, M(t)) I.repr` (polymorphic over `'c`).

*Proof*
By Corollary 2 and the type soundness of (this fragment of) OCaml. □

*Corollary 4*
If a closed object term $e$ has type $t$, then under the OCaml module definition

```
module RESULT =
  (functor (S:Symantics) -> struct open S let term () = M(e) end)
  (R)
```

evaluating the expression `RESULT.term ()` never gets stuck: it either does not terminate or evaluates to a value of type $M(t)$.

For variety, we show another interpreter `L`, which measures the *length* 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  e1 e2    = e1 + e2 + 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.

### *2.3 Higher-order abstract syntax versus de Bruijn indices*

Because Haskell and ML allow case analysis on $\lambda$-bound variables, one might worry that our HOAS representation of the object language allows *exotic terms* and is thus inadequate. To the contrary, because the representation of an object term is *parametrically* polymorphic

```
module type Symantics = sig
  type ('c,'h,'dv) repr
  type ('c,'dv) vr                          (* variable representation *)

  val vz  : ('c, ('c,'d) vr * 'h, 'd) repr
  val vs  : ('c, 'h, 'd) repr -> ('c, _ * 'h, 'd) repr

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

  val lam : ('c, ('c,'da) vr * 'h, 'db) repr -> ('c,'h,'da->'db) repr
  val app : ('c,'h,'da->'db) repr -> ('c,'h,'da) repr -> ('c,'h,'db) repr
  val fix : ('c, ('c,'da->'db) vr * 'h, 'da->'db) repr
            -> ('c, 'h, 'da->'db) repr

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

module R = struct
  type ('c,'h,'dv) repr = 'h -> 'dv
  type ('c,'d) vr = 'd

  let vz   (x,_) = x
  let vs v (_,h) = v h

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

Fig. 4. Embedding and evaluating our object language using de Bruijn indices

over the type constructor `repr` of the interpreter, $\lambda$-bound object variables cannot be case-analyzed. We thus follow Washburn and Weirich (2008) in "enforcing term parametricity with type parametricity" to represent and fold over abstract syntax.

Although the rest of this paper continues to represent binding using HOAS, our approach is compatible with de Bruijn indices. The accompanying source file `incope-dB.ml` implements this alternative, starting with the `Symantics` signature and the `R` evaluator in Figure 4. In this encoding of the object language, `vz` represents the innermost variable, `vs vz` represents the second-to-innermost variable, and so on. The new type argument `'h` to `repr` tracks the type of the environment as a nested tuple, each of whose components is a value of type `('c,'dv) vr` representing a variable of type `'dv`. The evaluator `R` interprets each object term as a function from its environment to its value.

## 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 $t$ 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 quotes the expression $e$ for evaluation 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. Bracket, escape, and run are akin (modulo hygiene) to quasi-quotation, unquotation, and `eval` of Lisp.

To turn the evaluator `R` into a simple compiler, we bracket the computation on values to be performed at run time, then escape the code generation from terms to be performed at compile time. Adding these stage annotations yields the 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 e1 e2     = .<.~e1 <= .~e2>.
  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. The environment classifiers here, like the tuple types in §1.3, make it a type error to run an open expression.

*Proposition 5*
If an object term $e$ has the type $t$ in the context $x_1 : t_1, \ldots, x_n : t_n$, then in a MetaOCaml environment

$$\texttt{open C}, \quad x_1 \mapsto \texttt{.<}y_1\texttt{>.}, \ \ldots, \ x_n \mapsto \texttt{.<}y_n\texttt{>.}$$

where each $y_i$ is a future-stage variable of type $M(t_i)$, the MetaOCaml term $M(e)$ evaluates to a code value, of type `('c,` $M(t)$`) code` (polymorphic over `'c`), that contains no pattern-matching operations.

*Proof*
By structural induction on the typing derivation of $e$. □

*Corollary 6*
If a closed object term $e$ has type $t$, then under the OCaml module definition

```
module RESULT =
  (functor (S:Symantics) -> struct open S let term () = M(e) end)
  (C)
```

the expression `RESULT.term ()` evaluates to a code value, of type `('c,` $M(t)$`) code` (polymorphic over `'c`), that contains no pattern-matching operations.

We have also implemented this compiler in Haskell. Since Haskell has no convenient facility for typed staging, we emulate it by defining a data type `ByteCode` with constructors such as `Var`, `Lam`, `App`, `Fix`, and `INT`. (Alternatively, we could use Template Haskell (Sheard and Peyton Jones 2002) as our staging facility: `ByteCode` can be mapped to the abstract syntax of Template Haskell. The output of our compiler would 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 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, this `Symantics` interface extends to encompass an online partial evaluator that uses no universal type and no tags for object types. We present this partial evaluator in a sequence of three attempts to express the types of 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 yield 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, we might try to define `repr` in the partial evaluator as follows. In the phase tags `S0` and `D0`, the digit zero indicates our initial attempt.

```
type ('c,'dv) repr = S0 of ('c,'dv) R.repr | D0 of ('c,'dv) C.repr
```

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

```
let abstrI0 (e : ('c,int) repr) : ('c,int) C.repr =
  match e with S0 e -> C.int e | D0 e -> e
```

and a similar function `abstrB0` for dynamic boolean terms. Here, `C.int` is used to convert a static term (of type `('c,int) R.repr`, which is just `int`) to a dynamic term. We can now define the following components required by the `Symantics` signature:

```
let int  (x:int)  = S0 (R.int x)
let bool (x:bool) = S0 (R.bool x)
let add e1 e2 = match (e1,e2) with
  | (S0 e1, S0 e2) -> S0 (R.add e1 e2)
  | _ -> D0 (C.add (abstrI0 e1) (abstrI0 e2))
```

Integer and boolean literals are immediate, present-stage values. Addition yields a static term (using `R.add`) if and only if both operands are static; otherwise we extract the dynamic terms from the operands and add them using `C.add`.

Whereas `mul` and `leq` are as easy to define as `add`, we encounter a problem with `if_`. Suppose that the first argument to `if_` is a dynamic term (of type `('c,bool) C.repr`), the second a static term (of type `('c,'a) R.repr`), and the third a dynamic term (of type `('c,'a) C.repr`). We then need to convert the static term to dynamic, but there is no polymorphic "lift" function, of type `'a -> ('c,'a) C.repr`, to send a value to a future stage (Taha and Nielsen 2003; Xi et al. 2003).

Our `Symantics` signature only includes separate lifting methods `bool` and `int`, not a 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. Compiling a polymorphic lift function thus requires intensional type analysis. To avoid needing polymorphic lift, we turn to Sperber's (1996) and Asai's (2001) technique of building a dynamic term alongside every static term (Sumii and Kobayashi 2001).

### *4.2 Delaying binding-time analysis*

We start building the partial evaluator anew and switch to the data type

```
type ('c,'dv) repr = P1 of ('c,'dv) R.repr option * ('c,'dv) C.repr
```

so that a partially evaluated term always contains a dynamic component and sometimes contains a static component. The two alternative constructors of an `option` value, `Some` and `None`, tag each partially evaluated term to indicate whether its value is known statically at the present stage. This tag is not an object type tag: all pattern matching below is

exhaustive. Now that the future-stage component is always available, we can define the polymorphic function

```
let abstr1 (P1 (_,dyn) : ('c,'dv) repr) : ('c,'dv) C.repr = dyn
```

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

```
let int (x:int) = P1 (Some (R.int x), C.int x)
let add e1 e2 = match (e1,e2) with
  | (P1 (Some n1, _), P1 (Some n2, _)) -> int (R.add n1 n2)
  | _ -> P1 (None, C.add (abstr1 e1) (abstr1 e2))
let if_ eb et ee = match eb with
  | P1 (Some s, _) -> if s then et () else ee ()
  | _ -> P1 (None, C.if_ (abstr1 eb) (fun () -> abstr1 (et ()))
                                     (fun () -> abstr1 (ee ())))
```

However, we stumble on functions. Given how we just defined `repr`, a partially evaluated object function, such as the identity $\lambda x.x$ (of type $\mathbb{Z} \to \mathbb{Z}$) embedded in OCaml as `lam (fun x -> x)` (of type `('c,int->int) repr`), consists of a dynamic part (of type `('c,int->int) C.repr`) and optionally a static part (of type `('c,int->int) R.repr`). 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 $(\lambda x.x)1$. 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 is into `lam f`: the type `('c,'a->'b) repr` should be one of

```
S1 of ('c,'a) repr -> ('c,'b) repr | E1 of ('c,'a->'b) C.repr
P1 of (('c,'a) repr -> ('c,'b) repr) option * ('c,'a->'b) C.repr
```

unlike `('c,int) repr` or `('c,bool) repr`. 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 (see §5.1), inductively defines a map from source types to target types that performs case distinction on the source type. In Haskell, typecase can be implemented using either GADTs or type-class functional dependencies (Oliveira and Gibbons 2005). The accompanying code shows both approaches (`Incope.hs` and `incope1.hs`), neither of which is portable to OCaml. In addition, the problem of non-exhaustive pattern-matching reappears in the GADT approach because GHC 6.8 and prior cannot see that a particular type of GADT value precludes certain constructors. Although this is an implementation issue of GHC, it indicates that assuring exhaustive pattern match with GADTs requires non-trivial reasoning (beyond the abilities of GHC at the moment); certainly GADTs fail to make it *syntactically* apparent that pattern matching is exhaustive.

```
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. 5. A (Meta)OCaml embedding of our object language that supports partial evaluation

### *4.3 The "final" solution*

The problem in the last section is that we want to write

```
type ('c,'dv) repr = P1 of ('c,'dv) static option * ('c,'dv) C.repr
```

where `static` is the type function defined by

```
('c,int)    static = ('c,int) R.repr
('c,bool)   static = ('c,bool) R.repr
('c,'a->'b) static = ('c,'a) repr -> ('c,'b) repr
```

Although we can use type classes to define this type function in Haskell, that is not portable to OCaml. However, the three typecase alternatives of `static` are already present in existing methods of `Symantics`. Thus emerges a simple and portable solution, if a long-winded one: we bake `static` into the signature `Symantics`. In Figure 2, the `repr` type constructor took two arguments `('c,'dv)`; in Figure 5, we add an argument `'sv` for the type `('c,'dv) static`.

The interpreters `R`, `L` and `C` in §2.2 and §3 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
```

with the rest the same.

Figure 6 shows the partial evaluator `P`. Its type `repr` expresses the definition for `static` given at the start of this section, with `'sv` taking the crucial place of `('c,'dv) static`. The function `abstr` extracts a future-stage code value from the result of partial evaluation. Conversely, the function `pdyn` injects a code value into the `repr` type. Thus, `abstr` and

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

`pdyn` are like the *reify* and *reflect* functions defined in normalization by evaluation (Danvy 1996), but as in §4.2, we build dynamic terms alongside any static ones to express how the lift function is indexed by the dynamic type. Analogously, we now build a static type alongside the dynamic type to express how the static type is indexed by the dynamic type. Thus we establish a bijection `static` between static and dynamic types, without defining at the type level the injection-projection pairs customarily used to establish such bijections for interpreters (Benton 2005; Ramsey 2005), partial evaluation (Danvy 1996), and type-level functions (Oliveira and Gibbons 2005). This emulation of type-indexed types is related to intensional type analysis (Harper and Morrisett 1995; Hinze et al. 2004), but intensional type analysis cannot handle our `fix` (Xi et al. 2003).

The static portion of the interpretation of `lam f` is `Some f`, which just wraps the HOAS function `f`. The interpretation of `app ef ea` checks to see if `ef` is such a wrapped HOAS function. If it is, we apply `f` to the concrete argument `ea`, so as to perform static computations (see the example below). If `ef` has only a dynamic part, we residualize.

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. Although our basic machinery for partial evaluation is independent of such algebraic simplifications, it makes them easy to add and to abstract over the specific domains (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 accompanying 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 6 takes the naïve approach of "going all the way", that is, whenever the argument is static, we unfold `fix` rather than residualize it. 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, 1989), 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 (Bondorf and Danvy 1991).

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)))))>.}
```

If the object program does not use `fix`, then the output of `P` is $\beta$-normal. Also, `P` is correct in that, if interpreting an object term using `P` terminates, then the `dy` component of the output is equivalent to the interpretation of the same object term using `C`, modulo $\alpha$-renaming, $\beta$-reduction, and algebraic simplification. To prove this correctness by structural induction on the object term, we need to strengthen the induction hypothesis to assert that the `st` component, if not `None`, is consistent with the `dy` component.

All pattern-matching in `P` is *syntactically* exhaustive, so it is patent to the metalanguage implementation that `P` never gets stuck. Further, `P` uses pattern-matching only to check if a value is known statically, never to check what type a value has dynamically. In other words, our partial evaluator tags phases (with `Some` and `None`) but not object types, so it is patent that the *output* of `P` never gets stuck.

Our partial evaluator owes much to Thiemann (1999) and Sumii and Kobayashi (2001), who deforested the object term representation and expressed a partial evaluator as a collection of term combinators in a typed metalanguage. Like us, Sumii and Kobayashi follow

Sperber (1996) and Asai (2001) in building static and dynamic terms in tandem, to combine offline and online partial evaluation. Mogensen's earlier self-reducers for the untyped $\lambda$-calculus (1992, 1995) also build static and dynamic terms in tandem. However, they build a static term for every object term, even a bound variable, so they move some work from `app` to `pdyn` (in terms of Figure 6) and remain untyped. In contrast, we follow Sperber, Asai, and Sumii and Kobayashi in leaving the static term optional, so as to perform lifting without juggling explicit type indices in the encoding of an object term. The idea of generating static and dynamic components alongside each other is part of the tradition that developed partial evaluators such as Schism (Consel 1993; §5).

Our contribution to the literature on partial evaluation is to use mere Hindley-Milner types in the metalanguage to assure statically and patently that partially evaluating a well-typed object program not only never gets stuck but also, if it terminates, produces a well-typed output program that never gets stuck. Moreover, thanks to the online binding-time analysis performed by our partial evaluator (in contrast to Thiemann's), these types form an instance of a general `Symantics` signature that encompasses other interpreters such as evaluation and compilation. This early and manifest assurance of type safety contrasts, for example, with Birkedal and Welinder's compiler generator (cogen) for ML (1993), which transforms a program into its tagless generating extension. Because that cogen uses a universal type, the fact that it never generates an ill-typed generating extension from a well-typed input program is only manifest when each generating extension is type-checked, and the fact that the generating extension never generates an ill-typed residual program from well-typed static input is only manifest when each residual program is type-checked. Similarly, the fact that the partial evaluator of Fiore (2002) and that of Balat et al. (2004), both of which use delimited control operators, never turn well-typed code into ill-typed code is not assured by the metalanguage, whether or not as part of a typed family of interpreter modules.

Our partial evaluator 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. Of course, CPS is a well-studied topic, and Thiemann's work on program generation (1999) already includes a CPS evaluator expressed using combinator functions rather than data constructors. We focus here on expressing CPS transformations as part of a larger, typed family of interpreters.

### *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.[2] To represent a CBN ob-

[2] To be more precise, most Haskell implementations use call-by-need, which is observationally equivalent to call-by-name because sharing is not observable (Ariola et al. 1995).

ject 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, by changing the type arguments to `repr`. Luckily, the type function `static` needed for the partial evaluator in §4.3 is precisely the same type function we need for CBN CPS, so our CBN interpreter can match the `Symantics` signature in §4.3, without even using the `'dv` argument to `repr`.

```
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. To avoid this higher-rank polymorphism in the core language, we could also define `RCN` as a functor parameterized over the answer type.

```
module RCN(W: sig type w end) = struct
  type ('c,'sv,'dv) repr = ('sv -> W.w) -> W.w   ...
```

This alternative is more cumbersome to use because the functor needs to be applied once for each answer type, but it translates to, for example, Standard ML, whose core language does not support higher-rank polymorphism.

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. The interpreter `RCV` is useful for CBV evaluation of the object language whether the metalanguage is CBV or CBN. To match the same `Symantics` signature as `RCN` above, `RCV` uses Reynolds's (1974) CBV CPS transformation, in which variables denote computations (that is, functions from continuations), rather than Plotkin's (1975), in which variables denote values.

We turn to a more general approach to CBV CPS: a CPS transformer that turns any implementation of `Symantics` into a CPS version of that evaluator. This functor on interpreters performs Plotkin's (1975) 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 `static` in §4.3. Alas, the type function we need is not identical to `static`, so again we need to change the type arguments to `repr` in the `Symantics` signature. As in §4.3, 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. The verbosity of these types is the only difficulty in defining a replacement signature for `Symantics` which captures that of `CPST` as well.

For brevity, 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 () =
       app (lam (fun x -> x)) (bool true)  (* same as before *)
  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.

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 $(\lambda x.x)\,\mathrm{true}$, 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>.}
```

```
module type Symantics1 = sig
  type 'c dint
  type 'c dbool
  type ('c,'da,'db) darr
  type ('c,'dv) repr
  val int : int  -> ('c, 'c dint) repr
  val bool: bool -> ('c, 'c dbool) repr
  val lam : (('c,'da) repr -> ('c,'db) repr) -> ('c, ('c,'da,'db) darr) repr
  val app : ('c, ('c,'da,'db) darr) repr -> ('c, 'da) repr -> ('c, 'db) repr
  val fix : ('x -> 'x) -> (('c, ('c,'da,'db) darr) repr as 'x)
  val add : ('c, 'c dint) repr -> ('c, 'c dint) repr -> ('c, 'c dint) repr
  val mul : ('c, 'c dint) repr -> ('c, 'c dint) repr -> ('c, 'c dint) repr
  val leq : ('c, 'c dint) repr -> ('c, 'c dint) repr -> ('c, 'c dbool) repr
  val if_ : ('c, 'c dbool) repr
            -> (unit -> 'x) -> (unit -> 'x) -> (('c, 'da) repr as 'x)
end
```

Fig. 7. A (Meta)OCaml embedding that abstracts over an inductive map on object types

### *5.3 Abstracting over an inductive map on object types*

Having seen that each CPS interpreter above matches a differently modified `Symantics` signature, one may wonder whether `Symantics` can be generalized to encompass them all. The answer is yes: the `Symantics1` signature in Figure 7 abstracts our representation of object terms not only over the type constructor `repr` but also over the three branches that make up an inductive map, such as `static` in §4.3, from object types to metalanguage types. The first two branches (for the object types $\mathbb{Z}$ and $\mathbb{B}$) become the abstract types `dint` and `dbool`, whereas the third branch (for object types $t_1 \to t_2$) becomes the new abstract type constructor `darr`.

Almost every interpreter in this paper can be made to match the `Symantics1` signature without changing any terms, by defining `dint`, `dbool`, `darr`, and `repr` suitably. For example, the types in the evaluator `R` and the CBV CPS transformer `CPST` should be changed as follows.

```
module R = struct
  type 'c dint = int
  type 'c dbool = bool
  type ('c,'da,'db) darr = 'da -> 'db
  type ('c,'dv) repr = 'dv   ...

module CPST(S: Symantics1)(W: sig type 'c dw end) = struct open W
  type 'c dint = 'c S.dint
  type 'c dbool = 'c S.dbool
  type ('c,'da,'db) darr =
    ('c, 'da, ('c, ('c, 'db, 'c dw) S.darr, 'c dw) S.darr) S.darr
  type ('c,'dv) repr =
    ('c, ('c, ('c, 'dv, 'c dw) S.darr, 'c dw) S.darr) S.repr   ...
```

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

Fig. 8. Extending our typed object language with mutable state of type $t_s$

Modified thus, `CPST` produces modules that match `Symantics1` and can be not only evaluated or compiled but also transformed using `CPST` again. The accompanying source file `inco.ml` shows the details, including one-pass CPS transformations in the higher-order style of Danvy and Filinski (1992).

The abstract type constructors in Figure 7 exemplify the amount of polymorphism that our technique requires of the metalanguage in order to represent a given object language. Generally, our technique represents term constructions (such as $+$) by applying abstract functions (such as `add`) and represents type constructions (such as $\rightarrow$) and typing judgments (namely ':') by applying abstract type constructors (such as `darr` and `repr`). Therefore, it requires the metalanguage to support enough polymorphism to abstract over the interpretation of each inference rule for well-typed terms and well-kinded types. For example, to encode System F's term generalization rule

$$\frac{\begin{array}{c}[\alpha : \star] \\ \vdots \\ e : t\end{array}}{\Lambda\alpha.\, e : \forall\alpha.\, t,}$$

the metalanguage must let terms (representing $\Lambda\alpha.\, e$) abstract over terms (interpreting $\Lambda$) that are polymorphic both over type constructors of kind $\star \rightarrow \star$ (representing $t$ with $\alpha$ free) and over polymorphic terms of type $\forall\alpha{:}\star \ldots$ (representing $e$ with $\alpha$ free). These uses of higher-rank and higher-kind polymorphism let 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).

The only interpreter in this paper that does not fit `Symantics1` is the partial evaluator `P`. It does not fit because it uses *two* inductive maps on object types—both `'sv` and `'dv` in Figure 5. We could define a `Symantics2` signature to abstract over *two* inductive maps over object types; it would include 4 abstract types and 2 abstract type constructors in addition to `repr`. It would then be easy to turn `P` into a functor that returns a `Symantics2` module, but the input to `P` can still only match `Symantics1`. This escalation points to a need for either record-kind polymorphism (so that `'dv` in Figure 7 may be more than just one type) or type-indexed types (so that we do not need to emulate them in the first place).

### *5.4 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 (or more generally any monadic effect) by representing it using continuations (Filinski 1994). As Figure 8 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 $x$ and even if evaluation is CBN.

The form "case $e_1$ of $x.e_2$" is equivalent to "let" in Moggi's monadic metalanguage (1991). If $x$ does not appear in $e_2$, then it is same as the more familiar sequencing form "$e_1; e_2$". We embed this extended object language into OCaml by extending the `Symantics` signature in Figure 5.

```
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 the term "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 $x.(\text{state} \leftarrow 2; x + !\text{state})$" 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 ...
  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.

Because the `SymSI` signature extends `Symantics`, any encoding of a term in the pure object language (that is, any functor that takes a `Symantics` module as argument) can also be used as a term in the extended object language (for example, applied to an implementation of `SymSI`). In particular, `RCPSI` matches the `Symantics` signature and implements the unextended object language: we can pass `RCPSI` to the functor `EX` (Figure 2) and run the example `test1` from there. The main use for `RCPSI` is to interpret the extended language.

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

We reiterate that this implementation adding state and imperative features is very close to the CPS interpreter and uses 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. Yet another way to add side effects to the object language is to write a monadic interpreter (for a specific monad or a general class of monads), which can be structured as a module matching the `Symantics1` signature in Figure 7.

## 6 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 simpler and more flexible, they are unavailable in mainstream ML, and their implementation in GHC 6.8 fails to detect exhaustive pattern matching. We also wanted to find the minimal set of widespread language features needed for tagless type-preserving interpretation.

The simply typed $\lambda$-calculus can interpret itself, provided we use universal types (Taha et al. 2001). The ensuing tagging overhead motivated Makholm (2000); 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 that a typed interpreter of a typed language cannot be tagless without

advanced types, based on the premise that the only way to encode a typed language in a typed language is to use a sum type (at some level of the hierarchy). While the logic is sound, we (following Yang (2004)) showed that the premise 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.

Our separation between the `Symantics` interface and its many implementations codifies the common practice of implementing an embedded DSL by specifying an abstract syntax of object-language pervasives, such as addition and application, then providing multiple interpretations of them. The techniques we use to form such a family of interpreters find their origins in Holst's *language triplets* (1988), though in an untyped setting. Jones and Nielson (1994) also prefigured this separation when they decomposed a denotational definition of an untyped object language into a core semantics (which we call abstract syntax) and multiple interpretations.

In the typed setting, Nielson (1988) expressed families of program analyses on typed object languages using a typed $\lambda$-calculus as a metalanguage; however, the embeddings of the object language and the analyses are not type-checked in the metalanguage, unlike with our `Symantics` signature. When implementing a typed, embedded DSL, it is also common practice 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 two approaches are not tagless because they still use universal types, such as Lava's `Bit` and `NumSig`, and Rhiger's `Raw` (his Figure 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).

Thiemann and Sperber (1997) implemented a set of *binding-time polymorphic* combinators in Gofer, using many constructor classes. By merging all their classes into one and dropping polymorphic lift, they could have invented `Symantics`.

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 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.

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.

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. 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).

Other researchers have very recently realized that it is useful to abstract over higher-kinded types, like our `repr`. Moors et al. (2008) put the same power to work in Scala. Hofer et al. (2008) also use Scala and note that they are influenced by our work (Carette et al. 2007). Where we have concentrated on multiple efficient interpretations of the same language, they have concentrated on composing the languages and interpretations.

## 7 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 includes 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 improves the safety and reduces the overhead of embedding 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 (§4.3) 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.

## Acknowledgments

We thank Olivier Danvy, Neil Jones, Martin Sulzmann, Eijiro Sumii, and Walid Taha for helpful discussions. Pieter Hofstra, Bart Jacobs, Sam Staton, and Eijiro Sumii kindly provided some useful references. We thank anonymous reviewers for some wonderful suggestions as well as pointers to related work.

**References**

Ariola, Zena M., Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. The call-by-need lambda calculus. In *POPL '95: Conference record of the annual ACM symposium on principles of programming languages*, 233–246. New York: ACM Press.

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.

Bawden, Alan. 1999. Quasiquotation in Lisp. In *Proceedings of the 1999 ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation*, ed. Olivier Danvy, 4–12. Note NS-99-1, BRICS.

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.

Birkedal, Lars, and Morten Welinder. 1993. Partial evaluation of Standard ML. Master's thesis, DIKU, University of Copenhagen, Denmark. DIKU Research Report 93/22.

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.

Bondorf, Anders, and Olivier Danvy. 1991. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming* 16(2):151–195.

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.

Carette, Jacques, Oleg Kiselyov, and Chung chieh Shan. 2007. Finally tagless, partially evaluated. In *Aplas*, ed. Zhong Shao, vol. 4807 of *Lecture Notes in Computer Science*, 222–238. Springer.

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.

Consel, Charles. 1993. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the 1993 ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation*, 145–154. 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.

———. 1998. A simple solution to type specialization. In *Proceedings of ICALP'98: 25th international colloquium on automata, languages, and programming*, ed. Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, 908–917. Lecture Notes in Computer Science 1443, Berlin: Springer-Verlag.

Danvy, Olivier, and Andrzej Filinski. 1992. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2(4):361–391.

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.

Filinski, Andrzej. 1994. Representing monads. In *POPL '94: Conference record of the annual ACM symposium on principles of programming languages*, 446–457. New York: ACM Press.

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.

Futamura, Yoshihiko. 1971. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls* 2(5):45–50. Reprinted with revisions in *Higher-Order and Symbolic Computation* 12(4):381–391.

Gomard, Carsten K., and Neil D. Jones. 1991. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming* 1(1):21–69.

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.

Hofer, Christian, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of DSLs. In *ACM conference on generative programming and component engineering (GPCE'08)*.

Holst, Carsten Kehler. 1988. Language triplets: The AMIX approach. In *Partial evaluation and mixed computation*, ed. Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, 167–186. Amsterdam: North-Holland.

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.

Hughes, John. 1998. Type specialization. *ACM Computing Surveys* 30(3es:14):1–6.

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

Jones, Neil D., and Flemming Nielson. 1994. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of logic in computer science*. Oxford University Press. 527–629.

Jones, Neil D., Peter Sestoft, and Harald Søndergaard. 1989. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation* 2(1): 9–50.

Landin, Peter J. 1966. The next 700 programming languages. *Communications of the ACM* 9(3):157–166.

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*.

Makholm, Henning. 2000. On Jones-optimal specialization for strongly typed languages. In *Semantics, applications and implementation of program generation*, ed. Walid Taha, vol. 1924 of *Lecture Notes in Computer Science*, 129–148. Montreal, Canada: Springer-Verlag.

MetaOCaml. `http://www.metaocaml.org`.

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.

Mogensen, Torben Æ. 1992. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming* 2(3):345–363.

———. 1995. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of the 1995 ACM SIGPLAN workshop on partial evaluation and semantics-*

*based program manipulation*, 39–44. New York: ACM Press.

Moggi, Eugenio. 1991. Notions of computation and monads. *Information and Computation* 93(1):55–92.

Moors, Adriaan, Frank Piessens, and Martin Odersky. 2008. Generics of a higher kind. In *Oopsla '08: Proceedings of the 23rd acm sigplan conference on object oriented programming systems languages and applications*, 423–438. New York, NY, USA: ACM. General Chair-Gail E. Harris.

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.

Nielson, Flemming. 1988. Strictness analysis and denotational abstract interpretation. *Information and Computation* 76(1):29–92.

Nielson, Flemming, and Hanne Riis Nielson. 1988. Automatic binding time analysis for a typed $\lambda$-calculus. In *POPL '88: Conference record of the annual ACM symposium on principles of programming languages*, 98–106. New York: ACM Press.

———. 1992. *Two-level functional languages*. Cambridge University Press.

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.

———. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5:223–255.

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.

———. 1975. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New directions in algorithmic languages 1975*, ed. Stephen A. Schuman, 157–168. IFIP Working Group 2.1 on Algol, Rocquencourt, France: INRIA.

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.

Sheard, Tim, and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 Haskell workshop*, ed. Manuel M. T. Chakravarty, 1–16. New York: ACM Press.

Sperber, Michael. 1996. Self-applicable online partial evaluation. In *Partial evaluation*, ed. Olivier Danvy, Robert Glück, and Peter Thiemann, 465–480. Lecture Notes in Computer Science 1110, Schloß Dagstuhl, Germany: Springer-Verlag.

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.

Taha, Walid. 1999. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trival (extended abstract). In *Proceedings of the 2000 ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation*, ed. Julia L. Lawall, vol. 34(11) of *ACM SIGPLAN Notices*, 34–43. 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. 1999. Combinators for program generation. *Journal of Functional Programming* 9(5):483–525.

Thiemann, Peter, and Michael Sperber. 1997. Program generation with class. In *Proceedings informatik'97*, ed. M. Jarke, K. Pasedach, and K. Pohl, 582–592. Reihe Informatik aktuell, Aachen: Springer-Verlag.

Washburn, Geoffrey Alan, and Stephanie Weirich. 2008. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Journal of Functional Programming* 18(1):87–140.

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. 2004. Encoding types in ML-like languages. *Theoretical Computer Science* 315(1):151–190.