Finally Tagless, Partially Evaluated*

Tagless Staged Interpreters for Simpler Typed Languages

Jacques Carette[1], Oleg Kiselyov[2], and Chung-chieh Shan[3]

[1] McMaster University carette@mcmaster.ca
[2] FNMOC oleg@pobox.com
[3] Rutgers University ccshan@rutgers.edu

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

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

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

> It should also be possible to define languages with a highly refined syntactic type structure. Ideally, such a treatment should be metacircular, in the sense that the type structure used in the defined language should be adequate for the defining language. John Reynolds [28]

1 Introduction

A popular way to define and implement a language is to embed it in another [28]. Embedding means to represent terms and values of the *object language* as terms and values in the *metalanguage*. Embedding is especially appropriate for domain-specific object languages because it supports rapid prototyping and integration with the host environment [16]. If the metalanguage supports *staging*, then the embedding can compile object programs to the metalanguage and avoid the overhead of interpreting them on the fly [23]. A staged definitional interpreter is thus a promising way to build a domain-specific language (DSL).

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

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

$$\frac{b \text{ is a boolean}}{b : \mathbb{B}} \qquad \frac{e : \mathbb{B} \quad e_1 : t \quad e_2 : t}{\text{if } e \text{ then } e_1 \text{ else } e_2 : t} \qquad \frac{e_1 : \mathbb{Z} \quad e_2 : \mathbb{Z}}{e_1 + e_2 : \mathbb{Z}} \qquad \frac{e_1 : \mathbb{Z} \quad e_2 : \mathbb{Z}}{e_1 \times e_2 : \mathbb{Z}} \qquad \frac{e_1 : \mathbb{Z} \quad e_2 : \mathbb{Z}}{e_1 \leq e_2 : \mathbb{B}}$$

Fig. 1. Our typed object language

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

Pašalić et al. [23] and Xi et al. [37] motivated interpreting a typed object language in a typed metalanguage as an interesting problem. The common solutions to this problem store object terms and values in the metalanguage in a universal type, a generalized algebraic data type (GADT), or a dependent type. In the remainder of this section, we discuss these solutions, identify their drawbacks, then summarize our proposal and contributions. We leave aside the solved problem of writing a parser/type-checker, for embedding object language objects into the metalanguage (whether using dependent types [23] or not [2]), and just enter them by hand.

1.1 The tag problem

It is straightforward to create an algebraic data type, say in OCaml, Fig. 2(a), to represent object terms such as those in Figure 1. For brevity, we elide treating integers, conditionals, and fixpoint in this section. We represent each variable using a unary de Bruijn index.[4] For example, we represent the object term $(\lambda x.\, x)\,\mathrm{true}$ as `let test1 = A (L (V VZ), B true)`.

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

(b)
```
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)
```

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

(d)
```
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)
```

Fig. 2. OCaml code illustrating the tag problem

[4] We use de Bruijn indices to simplify the comparison with Pašalić et al.'s work [23].

Following [23], we try to implement an interpreter function `eval0`, Fig. 2(b). 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. If our OCaml-like metalanguage were untyped, the code above would be acceptable. The `L e` line exhibits interpretive overhead: `eval0` traverses the function body `e` every time (the result of evaluating) `L e` is applied. Staging can be used to remove this interpretive overhead [23, §1.1–2].

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

The usual solution is to introduce a universal type [23, §1.3] containing both booleans and functions, Fig. 2(c). We can then write a typed interpreter, Fig. 2(d), whose inferred type is `u list -> exp -> u`. Now we can evaluate `eval [] test1` obtaining `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)`. Second, the `lookup` function assumes a nonempty environment, so `eval` can fail if we evaluate an open term `A (L (V (VS VZ)), B true)`. After all, the type `exp` represents object terms both well-typed and ill-typed, both open and closed.

If we evaluate only closed terms that have been type-checked, then `eval` would never fail. Alas, this soundness is not obvious to the metalanguage, whose type system we must still appease with the nonexhaustive pattern matching in `lookup` and `eval` and the tags `UB` and `UA` [23, §1.4]. In other words, the algebraic data types above fail to express in the metalanguage that the object program is well-typed. This failure necessitates tagging and nonexhaustive pattern-matching operations that incur a performance penalty in interpretation [23] and impair optimality in partial evaluation [33]. In short, the universal-type solution is unsatisfactory because it does not preserve typing.

It is commonly thought that to interpret a typed object language in a typed metalanguage while preserving types is difficult and requires GADTs or dependent types [33]. In fact, this problem motivated much work on GADTs [24, 37] and on dependent types [11, 23]. Yet other type systems have been proposed to distinguish closed terms like `test1` from open terms [9, 21, 34], so that `lookup` never receives an empty environment. We discuss these proposals further in §5.

1.2 Our final proposal

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

```
let varZ env    = fst env         let b (bv:bool) env = bv
let varS vp env = vp (snd env)    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, `testf1 ()`, obtaining `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 `app (lam (varS varZ)) (b true)` gives a type error rather than a run-time error. The type error correctly complains that the initial environment should be a tuple rather than `()`. In other words, the term is open.

In sum, by Church-encoding terms using ordinary functions, we achieve a tagless evaluator for a typed object language in a metalanguage with a simple Hindley-Milner type system. In this *final* rather than *initial* approach, both kinds of run-time errors in §1.1 (applying a nonfunction and evaluating an open term) are reported at compile time. Because the new interpreter uses no universal type or pattern matching, it never results in a run-time error, and is in fact total. Because this safety is obvious not just to us but also to the metalanguage implementation, we avoid the serious performance penalty [23] of error checking. Glück [12] explains deeper technical reasons that inevitably lead to these performance penalties.

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

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

1.3 Contributions

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

- evaluate the term to a value in the metalanguage;
- measure the size or depth of the term;
- compile the term, with staging support such as in MetaOCaml;
- partially evaluate the term, online; and
- transform the term to continuation-passing style (CPS), even call-by-name (CBN) CPS, so as to isolate the evaluation order of the object language from that of the metalanguage.[5]

[5] Due to serious lack of space, we refer the reader to the accompanying code for this.

We have programmed our interpreters in OCaml (and, for staging, MetaOCaml [19]) and standard Haskell. The complete code is available at `http://okmij.org/ftp/packages/tagless-final.tar.gz` to supplement the paper. For simplicity, main examples in the paper will be in MetaOCaml; all examples have also been implemented in Haskell.

We attack the problem of tagless (staged) typed-preserving interpretation exactly as it was posed by Pašalić et al. [23] and Xi et al. [37]. We use their running examples and achieve the result they call desirable. Our contributions are as follows.

1. We build interpreters that evaluate (§2), compile (or evaluate with staging) (§3), and partially evaluate (§4) a typed higher-order object language in a typed metalanguage, in direct and continuation-passing styles.
2. All these interpreters use no type tags, patently never get stuck, and need no advanced type-system features such as GADTs, dependent types, or intentional type analysis.
3. The partial evaluator avoids polymorphic lift and delays binding-time analysis. It bakes a type-to-type map into the interpreter interface to eliminate the need for GADTs and thus remain portable across Haskell 98 and ML.
4. We use the type system of the metalanguage to check statically that an object program is well-typed and closed.
5. We show clean, comparable implementations in MetaOCaml and Haskell.
6. We specify a functor signature that encompasses all our interpreters, from evaluation and compilation (§2) to partial evaluation (§4).
7. We point a clear way to extend the object language with more features such as state.[6]
8. We describe an approach to self-interpretation compatible with the above. Self-interpretation turned out to be harder than expected.[6]

Our code is surprisingly simple and obvious in hindsight, but it has been cited as a difficult problem ([32] notwithstanding) to interpret a typed object language in a typed metalanguage without tagging or type-system extensions. For example, Taha et al. [33] 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 [38] has given a hint of how such a function can be expressed." We discuss related work in §5.

To reiterate, we do *not* propose any new language feature or new technique. We use features already present in mainstream functional languages—Hindley-Milner type system with either an inference-preserving module system or constructor classes, as realized in ML and Haskell 98—and techniques which have all appeared in the literature (in particular, [32, 38]), to solve a problem that was stated in the published record as unsolved and likely unsolvable in ML or Haskell 98 without extensions. The simplicity of our solution and its use of only mainstream features make it more practical to build typed, embedded DSLs.

[6] Again, please see our code.

2 The object language and its tagless interpreters

Figure 1 shows our object language, a simply-typed λ-calculus with fixpoint, integers, booleans, and comparison. The language is close to Xi et al.'s [37], without their polymorphic lift but with more constants so as to more conveniently express Fibonacci, factorial, and power. In contrast to §1, we encode binding using higher-order abstract syntax (HOAS) [20, 25] rather than de Bruijn indices. This makes the encoding convenient and ensures that our object programs are closed.

2.1 How to make encoding flexible: abstract the interpreter

We embed our language in (Meta)OCaml and Haskell. In Haskell, the functions that construct object terms are methods in a type class `Symantics` (with a parameter `repr` of kind `* -> *`), Fig. 3(a). The class is so named because its interface gives the syntax of the object language and its instances give the semantics. For example, we encode the term `test1`, or $(\lambda x.\, x)$ true, from §1.1 above as `app (lam (\x -> x)) (bool True)`, whose inferred type is `Symantics repr => repr Bool`. For another example, the classical *power* function is in Fig. 3(b) and the partial application $\lambda x.\, power\ x\ 7$ is in Fig. 3(c). The dummy argument `()` above is to avoid the monomorphism restriction, to keep the type of `testpowfix` and `testpowfix7` polymorphic in `repr`. The methods `add`, `mul`, and `leq` are quite similar, and so are `int` and `bool`. Therefore, we often show only one method of each group and elide the rest. The accompanying code has the complete implementations.

(a)
```
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
```

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

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

Fig. 3. Symantics in Haskell

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

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

Conversely, the type system of the metalanguage statically checks that the represented object term is well-typed and closed. If we err, say replace `int 7` with `bool True` in `testpowfix7`, Haskell will complain there that the expected type `Int` does not match the inferred `Bool`. Similarly, the object term $\lambda x.\, xx$ and

```
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

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. 4. A simple (Meta)OCaml embedding of our object language, and examples

its encoding `lam (\x -> app x x)` both fail occurs-checks in type checking. Haskell's type checker also flags syntactically invalid object terms, such as if we forget `app` somewhere above.

To embed the same object language in (Meta)OCaml, we replace the type class `Symantics` and its instances by a module signature `Symantics` and its implementations. Figure 4 shows a simple signature that suffices until §4. The two differences are: the additional type parameter `'c`, an *environment classifier* [34] required by MetaOCaml for code generation in §3; and the η-expanded type for `fix` and thunk types in `if_` since OCaml is a call-by-value language.

The functor `EX` in Fig. 4 encodes our running examples `test1` and the *power* function (`testpowfix`). The dummy argument to `test1` and `testpowfix` is an artifact of MetaOCaml, related to monomorphism: in order for us to run a piece of generated code, it must be polymorphic in its environment classifier (the type variable `'c` in Figure 4). The value restriction dictates that the definitions of our object terms must look syntactically like values. (Alternatively, we could have used the rank-2 record types of OCaml to maintain the necessary polymorphism.) Thus, we represent an object expression in OCaml as a functor from `Symantics` to an appropriate semantic domain. This is essentially the same as the constraint `Symantics repr =>` in the Haskell embedding.

2.2 Two tagless interpreters

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

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

```
module R = struct type ('c,'dv) repr = 'dv (* no wrappers *)
  let int  (x:int)  = x         let bool (b:bool) = b
  let lam  f        = f         let app  e1 e2    = e1 e2
  let fix  f        = let rec self n = f self n in self
  let add  e1 e2    = e1 + e2   let mul  e1 e2    = e1 * e2
  let leq  x y      = x <= y
  let if_  eb et ee = if eb then et () else ee () end
```

As in §1.2, 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 source code shows a total interpreter `L` that measures the size of each object term.) We can also generalize from `R` to all interpreters; these propositions follow immediately from the soundness of the metalanguage's type system.

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

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

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 [6, 34]. 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 to quasi-quotation, unquotation, and `eval` of Lisp.

Inserting brackets and escapes appropriately into the evaluator `R` above yields the simple compiler `C` in Fig. 5(a). This is a straightforward staging of `module R`. This compiler produces unoptimized code. For example, interpreting our `test1` with Fig. 5(b) gives the code value `.<(fun x_6 -> x_6) true>.` of inferred type `('c, bool) C.repr`. Interpreting `testpowfix7` with Fig. 5(c) gives a code value with many apparent β- and η-redexes, Fig. 5(d). This compiler does not incur

(a)
```
module C = struct type ('c,'dv) repr = ('c,'dv) code
    let int (x:int)  = .<x>.            let bool (b:bool) = .<b>.
    let lam f        = .<fun x -> .~(f .<x>.)>.
    let app e1 e2    = .<.~e1 .~e2>.
    let fix f =  .<let rec self n = .~(f .<self>.) n in self>.
    let add e1 e2    = .<.~e1 + .~e2>.  let mul e1 e2 = .<.~e1 * .~e2>.
    let leq x y      = .<.~x <= .~y>.
    let if_ eb et ee = .<if .~eb then .~(et ()) else .~(ee ())>. end
```
(b)
```
let module E = EX(C) in E.test1 ()
```
(c)
```
let module E = EX(C) in E.testpowfix7
```
(d)
```
.<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>.
```

Fig. 5. The tagless staged interpreter `C`

any interpretive overhead: the code produced for $\lambda x.\, x$ is simply `fun x_6 -> x_6`. The resulting code obviously contains no tags and no pattern matching. The environment classifiers here, like the tuple types in §1.2, make it a type error to run an open expression. The accompanying code shows the Haskell implementation.

4 A tagless partial evaluator

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

4.1 Avoiding polymorphic lift

Roughly, a partial evaluator interprets each object term to yield either a static (present-stage) term (using `R`) or a dynamic (future-stage) term (using `C`). To distinguish between static and dynamic terms, we might try to define `repr` in the partial evaluator as `type ('c,'dv) repr = S0 of ('c,'dv) R.repr | E0 of ('c,'dv) C.repr`. 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`. We use `C.int` to convert from the static term `('c,int) R.repr`, which is just `int`, to the dynamic term.

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 the future stage [34, 37].

Our `Symantics` only includes separate lifting methods `bool` and `int`, not a parametrically polymorphic lifting method, for good reason: When compiling

to a first-order target language such as machine code, booleans, integers, and functions may well be represented differently. Thus, compiling polymorphic lift requires intensional type analysis. To avoid needing polymorphic lift, we turn to Asai's technique [1, 32]: build a dynamic term alongside every static term.

4.2 Delaying binding-time analysis

We 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. By distributivity, the two alternative constructors of an `option` value, `Some` and `None`, tag each partially evaluated term with a phase: either present or future. This tag is not an object type tag: all pattern matching below is exhaustive. Because the future-stage component is always present, we can now define the polymorphic function `let abstr1 (P1 (_,dyn)) = dyn` of type `('c,'dv) repr -> ('c,'dv) C.repr` to extract it without requiring polymorphic lift into `C`. We then try to define the interpreter `P1`—and get as far as the first-order constructs of our object language, including `if_`.

```
module P1 : Symantics = struct
  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_ = function
    | P1 (Some s,_) -> fun et ee -> if s then et () else ee ()
    | eb -> fun et ee -> P1 (None, C.if_ (abstr1 eb)
                                    (fun () -> abstr1 (et ()))
                                    (fun () -> abstr1 (ee ())))
```

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

To enable these simplifications, we delay binding-time analysis for a static function until it is applied, that is, until `lam f` appears as the argument of `app`. To do so, we have to incorporate `f` as it is into the `P1.repr` data structure: the representation for a function type `'a->'b` 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 `P1.repr` of `int` or `bool`. That is, we need a nonparametric data type, something akin to type-indexed functions and type-indexed types, which Oliveira and Gibbons [22] dub the *typecase* design pattern. Thus, typed partial evaluation, like typed CPS transformation, inductively defines a map from source types to target types that performs case distinction on the source type. In Haskell, typecase can be equivalently implemented either with GADTs or with type-class functional dependencies [22]. The accompanying code shows both approaches, neither portable to OCaml. In addition, the problem of nonexhaustive pattern-matching reappears in the GADT approach because GHC 6.6.1 cannot see that a particular type of a GADT value precludes certain constructors. Thus GADTs fail to make it *syntactically* apparent that pattern matching is exhaustive.

4.3 The "final" solution

Let us re-examine the problem in §4.2. What we would ideally like is to write `type ('c,'dv) repr = P1 of (repr_pe ('c,'dv)) R.repr option * ('c,'dv) C.repr` where `repr_pe` is the type function defined by

```
repr_pe ('c,int) = ('c,int); repr_pe ('c,bool) = ('c,bool)
repr_pe ('c,'a->'b) = ('c,'a) repr -> ('c,'b) repr
```

Although we can use type classes to define this type function in Haskell, that is not portable to MetaOCaml. However, these three typecase alternatives are already present in existing methods of `Symantics`. A simple and portable solution thus emerges: we bake `repr_pe` into the signature `Symantics`. We recall from Figure 4 in §2.1 that the `repr` type constructor took two arguments `'c` and `'dv`. We add an argument `'sv` for the result of applying `repr_pe` to `'dv`. Figure 6 shows the new signature.

```
module type Symantics = sig type ('c,'sv,'dv) repr
  val int : int -> ('c,int,int) 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 if_ : ('c,bool,bool) repr
            -> (unit->'x) -> (unit->'x) -> (('c,'sa,'da) repr as 'x) end
```

Fig. 6. A (Meta)OCaml embedding of our object language that supports partial evaluation (`bool`, `mul`, `leq` are elided)

The interpreters `R`, `L` and `C` above only use the old type arguments `'c` and `'dv`, which are treated by the new signature in the same way. Hence, all that needs to change in these interpreters to match the new signature is to add a phantom type argument `'sv` to `repr`. For example, the compiler `C` now begins `module C = struct type ('c,'sv,'dv) repr = ('c,'dv) code` with the rest the same. In contrast, the partial evaluator `P` relies on the type argument `'sv`.

Figure 7 shows the partial evaluator `P`. Its type `repr` literally expresses the type equation for `repr_pe` above. The function `abstr` extracts a future-stage

```
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 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)) end
```

Fig. 7. Our partial evaluator (`bool`, `mul`, `leq` and `fix` are elided)

code value from the result of partial evaluation. Conversely, the function `pdyn` injects a code value into the `repr` type. As in §4.2, we build dynamic terms alongside any static ones to avoid polymorphic lift.

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`, giving us a chance 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. Such optimizations can be quite effective in a large language with more base types and primitive operations.

Any partial evaluator must decide how much to unfold recursion. Our code naïvely unfolds `fix` whenever the argument is static. In the accompanying source code is a conservative alternative `P.fix` that unfolds recursion only once, then residualizes. Many sophisticated approaches have been developed to decide how much to unfold [17], but this issue is orthogonal to our presentation.

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 β-reduction has been statically performed to yield `true`. More interestingly, whereas `testpowfix7` compiles to a code value with many β-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)))))>.}
```

All pattern-matching in `P` is *syntactically* exhaustive, so it is patent to the meta-language implementation that `P` never gets stuck. Further, all pattern-matching occurs during partial evaluation, only to check if a value is known statically,

never what type it has. In other words, our partial evaluator tags phases (with `Some` and `None`) but not object types.

5 Related work

Our initial motivation came from several papers [23, 24, 33, 37] that use embedded interpreters to justify advanced type systems, in particular GADTs. 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 6.6.1 fails to detect exhaustive pattern matching. We also wanted to find the minimal set of widespread language features needed for tagless type-preserving interpretation.

Even a simply typed λ-calculus obviously supports self-interpretation, provided we use universal types [33]. The ensuing tagging overhead motivated Taha et al. [33] to propose tag elimination, which however does not statically guarantee that all tags will be removed [23].

Pašalić et al. [23], Taha et al. [33], Xi et al. [37], and Peyton Jones et al. [24] seem to argue as follows that a self-interpreter of a typed language cannot be tagless or Jones-optimal: (1) One needs to encode a typed language in a typed language based on a sum type (at some level of the hierarchy); (2) A *direct* interpreter for such an encoding of a typed language in a typed language requires either advanced types or tagging overhead; (3) Thus, an indirect interpreter is necessary, which needs a universal type and hence tagging. While the logic is sound, we (following Yang [38]) showed that the first step's premise is not valid.

Danvy and López [8] 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 partial evaluator establishes a bijection `repr_pe` between static and dynamic types (the valid values of `'sv` and `'dv`), and between static and dynamic terms. It is customary to implement such a bijection using an injection-projection pair, as done for interpreters [4, 27], partial evaluation [7], and type-level functions [22]. As explained in §4.3, we avoid injection and projection at the type level by adding an argument to `repr`. Our solution could have been even more straightforward if MetaOCaml provided total type-level functions such as `repr_pe` in §4.3—simple type-level computations ought to become mainstream.

At the term level, we also avoid converting between static and dynamic terms by building them in parallel, using Asai's method [1]. This method type-checks in Hindley-Milner once we deforest the object term representation. Put another way, we manual apply type-level partial evaluation to our type functions (see §4.3) to obtain simpler types acceptable to MetaOCaml.

Sumii and Kobayashi [32] also use Asai's method, to combine online and offline partial evaluation. They predate us in deforesting the object term representation to enable tagless partial evaluation. We strive for modularity by reusing interpreters for individual stages [31]: our partial evaluator `P` reuses our tagless

evaluator `R` and tagless compiler `C`, so it is patent that the *output* of `P` never gets stuck. It would be interesting to try to derive a *cogen* [35] in the same manner.

It is common to implement an embedded DSL by providing multiple interpretations of host-language pervasives such as addition and application. It is also common to use phantom types to rule out ill-typed object terms, as done in Lava [5] and by Rhiger [29]. However, these approaches are not tagless because they still use universal types, such as Lava's `Bit` and `NumSig`, and Rhiger's `Raw` (his Fig. 2.2) and `Term` (his Chap. 3), which incur the attendant overhead of pattern matching. The universal type also greatly complicates the soundness and completeness proofs of embedding [29], whereas our proofs are trivial. Rhiger's approach does not support typed CPS transformation (his §3.3.4).

We are not the first to implement a typed interpreter for a typed language. Läufer and Odersky [18] use type classes to implement a metacircular interpreter (rather than a self-interpreter) of a typed version of the SK language, which is quite different from our object language. Their interpreter appears to be tagless, but they could not have implemented a compiler or partial evaluator in the same way, since they rely heavily on injection-projection pairs.

Fiore [10] and Balat et al. [3] also build a tagless partial evaluator, using delimited control operators. It is type-directed, so the user must represent, as a term, the type of every term to be partially evaluated. We shift this work to the type checker of the metalanguage. By avoiding term-level type representations, our approach makes it easier to perform algebraic simplifications (as in §4.3).

We encode terms in elimination form, as a coalgebraic structure. Pfenning and Lee [26] 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_ω, perhaps because their avoidance of general recursive types compels them to include the polymorphic lift that we avoid in §4.1.

Our encoding of the type function `repr_pe` in §4.3 emulates type-indexed types and is related to intensional type analysis [13, 14]. However, our object language and running examples in HOAS include `fix`, which intensional type analysis cannot handle [37]. Our final approach seems related to Washburn and Weirich's approach to HOAS using catamorphisms and anamorphisms [36].

We could not find work that establishes that the *typed* λ-calculus has a final coalgebra structure. (See Honsell and Lenisa [15] for the untyped case.)

We observe that higher-rank and higher-kind polymorphism lets us typecheck and compile object terms separately from interpreters. This is consistent with the role of polymorphism in the separate compilation of modules [30].

6 Conclusions

We solve the problem of embedding a typed object language in a typed metalanguage without using GADTs, dependent types, or a universal type. Our family of interpreters include an evaluator, a compiler, a partial evaluator, and CPS transformers. It is patent that they never get stuck, because we represent object

types as metalanguage types. This work makes it safer and more efficient to embed DSLs in practical metalanguages such as Haskell and ML.

Our main idea is to represent object programs not in an initial algebra but using the existing coalgebraic structure of the λ-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 (§1.2); build dynamic terms alongside static ones (§4.1); simulating type functions for partial evaluation (§4.3) and CPS transformation. This shift also underlies fusion, functionalization, and amortized complexity analysis.

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

References

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

[2] Baars, Arthur I., and S. Doaitse Swierstra. 2002. Typing dynamic typing. In *ICFP*, 157–166.

[3] 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*, 64–76.

[4] Benton, P. Nick. 2005. Embedded interpreters. *JFP* 15(4):503–542.

[5] Bjesse, Per, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware design in Haskell. In *ICFP*, 174–184.

[6] Calcagno, Cristiano, Eugenio Moggi, and Walid Taha. 2004. ML-like inference for classifiers. In *ESOP*, 79–93.

[7] Danvy, Olivier. 1996. Type-directed partial evaluation. In *POPL*, 242–257.

[8] Danvy, Olivier, and Pablo E. Martínez López. 2003. Tagging, encoding, and Jones optimality. In *ESOP*, 335–347.

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

[10] Fiore, Marcelo P. 2002. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *PPDP*, 26–37.

[11] Fogarty, Seth, Emir Pasalic, Jeremy Siek, and Walid Taha. 2007. Concoqtion: Indexed types now! In *PEPM*.

[12] Glück, Robert. 2002. Jones optimality, binding-time improvements, and the strength of program specializers. In *ASIA-PEPM*, 9–19.

[13] Harper, Robert, and J. Gregory Morrisett. 1995. Compiling polymorphism using intensional type analysis. In *POPL*, 130–141.

[14] Hinze, Ralf, Johan Jeuring, and Andres Löh. 2004. Type-indexed data types. *Sci. Comput. Program.* 51(1-2):117–151.

[15] Honsell, Furio, and Marina Lenisa. 1999. Coinductive characterizations of applicative structures. *Math. Structures in Comp. Sci.* 9(4):403–435.

[16] Hudak, Paul. 1996. Building domain-specific embedded languages. *ACM Comp. Surv.* 28(4es):196.

[17] Jones, Neil D., Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice-Hall.
[18] Läufer, Konstantin, and Martin Odersky. 1993. Self-interpretation and reflection in a statically typed language. In *OOPSLA/ECOOP workshop on object-oriented reflection and metalevel architectures*.
[19] MetaOCaml. `http://www.metaocaml.org`.
[20] Miller, Dale, and Gopalan Nadathur. 1987. A logic programming approach to manipulating formulas and programs. In *IEEE symp. on logic programming*, 379–388.
[21] Nanevski, Aleksandar, and Frank Pfenning. 2005. Staged computation with names and necessity. *JFP* 15(6):893–939.
[22] Oliveira, Bruno César dos Santos, and Jeremy Gibbons. 2005. TypeCase: A design pattern for type-indexed functions. In *Haskell workshop*, 98–109.
[23] Pašalić, Emir, Walid Taha, and Tim Sheard. 2002. Tagless staged interpreters for typed languages. In *ICFP*, 157–166.
[24] Peyton Jones, Simon L., Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple unification-based type inference for GADTs. In *ICFP*, 50–61.
[25] Pfenning, Frank, and Conal Elliott. 1988. Higher-order abstract syntax. In *PLDI*, 199–208.
[26] Pfenning, Frank, and Peter Lee. 1991. Metacircularity in the polymorphic λ-calculus. *Theor. Comp. Sci.* 89(1):137–159.
[27] Ramsey, Norman. 2005. ML module mania: A type-safe, separately compiled, extensible interpreter. In *ML workshop*.
[28] Reynolds, John C. 1972. Definitional interpreters for higher-order programming languages. In *Proc. ACM Natl. Conf.*, vol. 2, 717–740. Repr. with a foreword in *HOSC* 11(4):363–397.
[29] Rhiger, Morten. 2001. Higher-Order program generation. Ph.D. thesis, BRICS, Denmark.
[30] Shao, Zhong. 1998. Typed cross-module compilation. In *ICFP*, 141–152.
[31] Sperber, Michael, and Peter Thiemann. 1997. Two for the price of one: Composing partial evaluation and compilation. In *PLDI*, 215–225.
[32] Sumii, Eijiro, and Naoki Kobayashi. 2001. A hybrid approach to online and offline partial evaluation. *HOSC* 14(2–3):101–142.
[33] Taha, Walid, Henning Makholm, and John Hughes. 2001. Tag elimination and Jones-optimality. In *PADO*, 257–275. LNCS 2053.
[34] Taha, Walid, and Michael Florentin Nielsen. 2003. Environment classifiers. In *POPL*, 26–37.
[35] Thiemann, Peter. 1996. Cogen in six lines. In *ICFP*, 180–189.
[36] Washburn, Geoffrey, and Stephanie Weirich. 2003. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *ICFP*, 249–262.
[37] Xi, Hongwei, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *POPL*, 224–235.
[38] Yang, Zhe. 1998. Encoding types in ML-like languages. In *ICFP*, 289–300.