Boxes Go Bananas: Encoding Higher-Order Abstract Syntax with Parametric Polymorphism

Geoffrey Washburn Stephanie Weirich
Department of Computer and Information Science
University of Pennsylvania
{geoffw,sweirich}@cis.upenn.edu

Abstract

Higher-order abstract syntax is a simple technique for implementing languages with functional programming. Object variables and binders are implemented by variables and binders in the host language. By using this technique, one can avoid implementing common and tricky routines dealing with variables, such as capture-avoiding substitution. However, despite the advantages this technique provides, it is not commonly used because it is difficult to write sound elimination forms (such as folds or catamorphisms) for higher-order abstract syntax. To fold over such a datatype, one must either simultaneously define an inverse operation (which may not exist) or show that all functions embedded in the datatype are parametric.

In this paper, we show how first-class polymorphism can be used to guarantee the parametricity of functions embedded in higher-order abstract syntax. With this restriction, we implement a library of iteration operators over data-structures containing functionals. From this implementation, we derive "fusion laws" that functional programmers may use to reason about the iteration operator. Finally, we show how this use of parametric polymorphism corresponds to the Schürmann, Despeyroux and Pfenning method of enforcing parametricity through modal types. We do so by using this library to give a sound and complete encoding of their calculus into System F_ω. This encoding can serve as a starting point for reasoning about higher-order structures in polymorphic languages.

Categories and Subject Descriptors

D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and Features—*abstract data types, polymorphism, control structures*; F.3.3 [**LOGICS AND MEANINGS OF PROGRAMS**]: Software—*type structure, program and recursion schemes, functional constructs*; F.4.1 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Mathematical Logic—*Lambda calculus and related systems, modal logic*

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

General Terms

Languages

Keywords

Higher-order abstract syntax, modal type system, catamorphism, parametricity, parametric polymorphism

1 Introduction

Higher-order abstract syntax (HOAS) is an old and seductively simple technique for implementing a language with functional programming.[1] The main idea is elegant: instead of representing object variables explicitly, we use metalanguage variables. For example, we might represent the object calculus term ($\lambda x.x$) with the Haskell expression `lam (\x -> x)`. Doing so eliminates the need to implement a number of tricky routines dealing with object language variables. For example, capture-avoiding substitution is merely function application in the metalanguage. However, outside of a few specialized domains, such as theorem proving, partial evaluation [26], logical frameworks [22] and intensional type analysis [27, 30], higher-order abstract syntax has found limited use as an implementation technique.

One obstacle preventing the widespread use of this technique is the difficulty in using elimination forms, such as catamorphisms[2], for datatypes containing functions. The general form of catamorphism for these datatypes requires that an inverse be simultaneously defined for every iteration [16]. Unfortunately, many operations that we would like to define with catamorphisms require inverses that do not exist or are expensive to compute.

However, if we know that the embedded functions in a datatype are *parametric*, we can use a version of the catamorphism that does not require an inverse [9, 24]. A parametric function may not examine its argument; it may only use it abstractly or "push it around". Only allowing parametric embedded functions works well with HOAS because the terms with non-parametric embedded functions are exactly those that have no correspondence to any λ-calculus term [24]. In this paper, we use *iterator* to refer to a catamorphism restricted to arguments with parametric functions.

A type system can separate parametric functions from those that

[1] While the name comes from Pfenning and Elliott [21], the idea itself goes back to Church. [4].

[2] Catamorphisms (also called folds) are sometimes represented with the bananas $(\!| \cdot |\!)$ notation [15].

are not. For example, Fegaras and Sheard [9] add tags to mark the types of datatypes whose embedded functions are not parametric, prohibiting iteration over those datatypes. Alternatively, Schürmann, Despeyroux and Pfenning [24, 8] use the necessity modality ("box") to mark those terms that allow iteration.

However, many modern typed languages already have a mechanism to enforce that an argument be used abstractly—*parametric polymorphism*. It seems desirable to find a way to use this mechanism instead of adding a separate facility to the type system. In this paper, we show how to encode datatypes with parametric function spaces in the polymorphic λ-calculus, including iteration operators over them.

Our specific contributions are the following. For functional programmers, we provide an informal description of how restricting datatypes to parametric function spaces can be enforced in the Haskell language using first-class polymorphism. We provide a safe and easy implementation of a library for iteration over higher-order abstract syntax. This Haskell library allows the natural expression of many algorithms over the object language; to illustrate its use, we use it to implement a number of operations including Danvy and Filinski's optimizing one-pass CPS conversion algorithm [6]. Furthermore, because we encode the iteration operator within the polymorphic λ-calculus, we also derive "fusion laws" about the iteration operator that functional programmers may use to reason about their programs.

To show the generality of this technique, we use this implementation to show a formal translation from the Schürmann, Despeyroux and Pfenning modal calculus [24] (called here the SDP calculus) to System F_ω. This encoding has an added benefit to language designers who wish to incorporate reasoning about parametric function spaces. It demonstrates how systems based on the polymorphic λ-calculus may be extended with reasoning about higher-order structure.

We do not claim that this encoding will solve all of the problems with programming using higher-order abstract syntax. In particular, algorithms that require the explicit manipulation of the names of bound variables remain outside the scope of this implementation technique.

The remainder of this paper is as follows. Section 2 starts with background material on catamorphisms for HOAS, including those developed by Meijer and Hutton [16] and Fegaras and Sheard [9]. In Section 2.1 we show how to use first-class polymorphism and abstract types to provide an interface for Fegaras and Sheard's implementation that enforces the parametricity of embedded functions. Using this interface, we show some examples of iteration including CPS conversion (Section 2.2). In Section 3, we describe an implementation of that interface within the part of Haskell that corresponds to System F_ω, and describe properties of that implementation in Section 3.1. Section 4 describes the SDP calculus and Section 5 presents an encoding of that calculus into F_ω, using the implementation that we developed in Section 3. Section 6 presents future work, Section 7 presents related work, and Section 8 concludes. We include Generic Haskell code for the polytypic part of our implementation in Appendix A and the full encoding of the SDP calculus into System F_ω in Appendix B.

2 Catamorphisms for datatypes with embedded functions

The following recursive datatype represents the untyped λ-calculus using Higher-Order Abstract Syntax (HOAS).[3]

```
data Exp = Lam (Exp -> Exp) | App Exp Exp
```

The data constructor `Lam` represents λ-expressions. However, instead of explicitly representing bound λ-calculus variables, Haskell functions are used to implement binding and Haskell variables are used to represent variables. For example, we might represent the identity function $(\lambda x.x)$ as `Lam (\x -> x)` or the infinite loop $(\lambda x.(xx))(\lambda x.(xx))$ as `App (Lam (\x -> App x x)) (Lam (\x -> App x x))`.

Using this datatype, we can implement an interpreter for the λ-calculus. To do so, we must also represent the result values (also using HOAS).

```
data Value = Fn (Value -> Value)
unFn (Fn x) = x
```

It is tricky to define recursive operations, such as evaluation, over this implementation of expressions. The argument, `x`, to `Lam` below is a function of type `Exp -> Exp`. To evaluate it, we must convert `x` to a function of type `Value -> Value`. Therefore, we must also simultaneously define an inverse to evaluation, called `uneval`, such that `eval . uneval = \x -> x`. This inverse is used to convert the argument of `x` from a `Value` to an `Exp`.

```
eval :: Exp -> Value
eval (Lam x)   = Fn (eval . x . uneval)
eval (App y z) = unFn (eval y) (eval z)
uneval :: Value -> Exp
uneval (Fn x) = Lam (uneval . x . eval)
```

Consider the evaluation of $((\lambda x.x)(\lambda y.y))$. First `eval` replaces `App` with `unFn` and pushes evaluation down to the two subcomponents of the application. Next, each `Lam` is replaced by `Fn`, and the argument is composed with `eval` and `uneval`. The `unFn` cancels the first `Fn`, and the identity functions can be removed from the compositions. As `uneval` is right inverse to `eval`, we can replace each `(eval . uneval)` with the identity function.

```
eval (App (Lam (\x -> x)) (Lam (\y -> y)))
= unFn  (eval (Lam (\x -> x)))
        (eval (Lam (\y -> y)))
= unFn (Fn (eval . \x -> x . uneval))
       (Fn (eval . \y -> y . uneval))
= (eval . uneval) (Fn (eval . uneval))
= (\x -> x) (Fn (\y -> y))
= Fn (\y -> y)
```

Many functions defined over `Exp` will follow this same pattern of recursion, requiring an inverse for `Lam` and calling themselves recursively for the subcomponents of `App`. *Catamorphisms* capture the general pattern of recursion for functions defined over recursive datatypes. For example, `foldl` is a catamorphism for the list datatype and can implement many list operations. For lists of type

[3] All of the following examples are in the syntax of the Haskell language [19]. While some of the later examples require an extension of the Haskell type system—first-class polymorphism—this extension is supported by the Haskell implementations GHC and Hugs.

```
newtype Rec a = Roll (a (Rec a))

data ExpF a = Lam (a -> a) | App a a
type Exp  = Rec ExpF

lam       :: (Exp -> Exp) -> Exp
lam x     = Roll (Lam x)

app       :: Exp -> Exp -> Exp
app x y   = Roll (App x y)

xmapExpF :: (a -> b, b -> a)
   -> (ExpF  a -> ExpF  b, ExpF  b -> ExpF  a)
xmapExpF (f,g) = (\x -> case x of
                     Lam x   -> Lam (f . x . g)
                     App y z -> App (f y) (f z),
                \x -> case x of
                     Lam x   -> Lam (g . x . f)
                     App y z -> App (g y) (g z))

cata ::
  (ExpF a -> a) -> (a -> ExpF a) -> Rec ExpF -> a
cata f g (Roll x) =
  f ((fst (xmapExpF  (cata f g, ana f g))) x)

ana  ::
  (ExpF a -> a) -> (a -> ExpF a) -> a -> Rec ExpF
ana f g x =
  Roll (snd (xmapExpF (cata f g, ana f g)) (g x))
```

Figure 1. Meijer/Hutton catamorphism

[a], foldr replaces [] with a base case of type b and (:) with a function of type (a -> b -> b).

Meijer and Hutton [16] showed how to define catamorphisms for datatypes with embedded functions, such as Exp. The catamorphism for Exp systematically replaces Lam with a function of type ((a -> a) -> a) and App with a function of type (a -> a -> a). However, just as we defined eval simultaneously with uneval, the catamorphism for Exp must be simultaneously defined with an *anamorphism*. The catamorphism provides a way to consume members of type Exp and the anamorphism provides a way to generate them.

In order to easily specify this anamorphism, we use a slightly more complicated version of the Exp datatype, shown at the top of Figure 1. This version makes the recursion in the datatype explicit. The newtype Rec computes the fixed point of type constructors (functions from types to types). The type Exp is the fixed point of the type constructor ExpF, where the recursive occurrences of Exp have been replaced with the type parameter a. The first argument to cata is of type ExpF a -> a (combining the two functions mentioned above, of type ((a -> a) -> a) and (a -> a -> a)). The first argument to ana has the inverse type a -> ExpF a.

The functions cata and ana are defined in terms of xmapExpF, a generalized version of a mapping function for the type constructor ExpF. Because of the function argument to Lam, xmapExpF maps two functions, one of type a -> b and the other of type b -> a. The definition of xmapExpF is completely determined by the definition of ExpF. With Generic Haskell [5], we can define xmap and automatically generate xmapExpF from ExpF (see Appendix A).[4] That way, we can easily generalize this catamorphism to other datatypes. Unlike map, which is defined only for covariant type constructors, xmap is defined for type constructors that have both positive and negative occurrences of the bound variable. The only type constructors of F_ω for which xmap is not defined are those whose bodies contain first-class polymorphism. For example, $\lambda\alpha:\star.\forall\beta:\star.\alpha\to\beta$.

```
data Rec a b = Roll (a (Rec a b)) | Place b

data ExpF a  = Lam (a -> a) | App a a
type Exp a = Rec ExpF a

lam        :: (Exp a -> Exp a) -> Exp a
lam x      = Roll (Lam x)

app        :: Exp a -> Exp a -> Exp a
app x y    = Roll (App x y)

xmapExpF :: (a -> b, b -> a)
   -> (ExpF  a -> ExpF  b, ExpF  b -> ExpF  a)
xmapExpF (f,g) = (\x -> case x of
                     Lam x   -> Lam (f . x . g)
                     App y z -> App (f y) (f z),
                   \x -> case x of
                     Lam x   -> Lam (g . x . f)
                     App y z -> App (g y) (g z))

cata :: (ExpF a -> a) -> Exp a -> a
cata f (Roll x)  =
  f ((fst (xmapExpF (cata f, Place))) x)
cata f (Place x) = x
```

Figure 2. Fegaras/Sheard catamorphism

We can use cata to implement eval. To do so we must describe one step of turning an expression into a value (the function evalAux) and one step of turning a value into an expression (the function unevalAux).

```
evalAux :: ExpF Value -> Value
evalAux (Lam f)   = Fn f
evalAux (App x y) = (unFn x) y

unevalAux :: Value -> ExpF Value
unevalAux (Fn f) = Lam f

eval :: Exp -> Value
eval x = cata evalAux unevalAux x
```

Using cata to implement operations such as eval is convenient because the pattern of recursion is already specified. None of eval, evalAux or unevalAux are recursively defined. However, for some operations, there is no obvious (or efficient) inverse. For example, to using cata to print out expressions also requires writing a parser. Fegaras and Sheard [9] noted that sometimes the operation of the catamorphism often undoes with f what it has just done with

[4] Meijer and Hutton's version of xmapExpF only created the first component of the pair. In ana where the second component is needed, they swap the arguments. This is valid because fst (xmap (f,g)) = snd(xmap (g,f)). However, while the version that we use here is a little more complicated, it can be defined with Generic Haskell.

g. This situation occurs when the argument to `cata` contains only *parametric* functions. A parametric function is one that does not analyze its argument with `case` or `cata`.

When the argument to `cata` is *parametric*, Fegaras and Sheard showed how to implement `cata` without `ana`. The basic idea is that for parametric functions, any use of `ana` during the computation of a catamorphism will always be annihilated by `cata` in the final result. Therefore, instead of computing the anamorphism, they use a place holder to store the original argument. When `cata` reaches that place holder, it returns the stored argument.

To implement Fegaras and Sheard's catamorphism, we must redefine `Rec`. In Figure 2, we extend it with an extra branch (called `Place`) that is the place holder. Because `Place` can contain any type of value, `Rec` (and consequently `Exp`) must be parameterized with the type of the argument to `Place`. This type is the result of the catamorphism over the expression. In the implementation of `cata`, `Place` is the second argument to `xmapExpF` instead of `ana f`. It is a right inverse to `cata f` by definition.

For example, to count the number of occurrences of bound variables in an expression, we might use the following code.

```
countvarAux :: ExpF Int -> Int
countvarAux (App x y) = x + y
countvarAux (Lam f) = f 1

countvar :: Exp Int -> Int
countvar = cata countvarAux
```

The function `countvarAux` describes what to do in one step. The number of variables in an application expression is the sum of the number of variables in `x` and the number of variables in `y`. In the case of a λ-expression, `f` is a function from the number of variables in a variable expression (i.e. one) to the number of variables in the body of the `lam`. For example, to count the variables in $(\lambda x.\ x\ x)$:

```
countvar (lam (\x -> app x x))
  = (countvar . (\x -> x + x) . Place) 1
  = (\x -> (countvar (Place x))
          + (countvar (Place x))) 1
  = (countvar (Place 1)) + (countvar (Place 1))
  = 2
```

This definition of `cata` only works for arguments whose function spaces are parametric and who do not use `Place`. Informally, we call such expressions *sound* and other expressions *unsound*. Applying `cata` to an unsound expression can return a meaningless result. For example, say we define the following term:

```
badplace :: Exp Int
badplace = lam (\x -> Place 3)
```

Then `countvar badplace = 3`, even though it contains no bound variables. Even more importantly for higher-order abstract syntax, unsound datatypes do not correspond to untyped λ-calculus expressions, so it is important to be able to distinguish between sound and unsound representations.[5]

[5] It is also important to distinguish between sound and unsound members of datatypes that have meaningful non-parametric representations. For these datatypes, the behavior of the Fegaras and Sheard catamorphism on unsound arguments does not correspond to the Meijer and Hutton version.

There are two ways for parametricity to fail, corresponding to the two destructors for the type `Exp a`. A function is not parametric if it uses `cata` or `case` to examine its argument, as below:

```
badcata :: Exp Int
badcata = lam (\x -> if (countvar x == 1)
                     then app x x
                     else x)

badcase :: Exp a
badcase = lam (\x -> case x of
                Roll (App v w) -> app x x
                Roll (Lam f)   -> x
                Place v        -> x)
```

Fegaras and Sheard designed a type system to distinguish between sound and unsound expressions. Datatypes such as `Exp` were annotated with flags to indicate whether they had been examined with either `case` or `cata`, and if so, they were prevented from appearing inside of non-flagged datatypes. Furthermore, their language prevented the user from accessing `Place` by automatically generating `cata` from the definition of the user's datatype.

2.1 Enforcing parametricity with type abstraction

The type of `badcata` is `Exp Int`. This type tells us that something is wrong: the type parameter of `Exp` is constrained to be `Int`, so we can only use `cata` on this expression to produce an `Int`. The same is true for `badplace`. Whenever we use `cata` or `Place` in an expression, this parameter will be constrained. If we can ensure that only sound expressions have type `(forall a. Exp a)`, then we can use *first-class polymorphism* to enforce that the argument to a function is sound. That way, we can be assured that it will behave as expected. For example, define a version of `cata`, called `iter0` that may only be applied to sound expressions, below. The implementation of `cata` uses the argument at the specific type `(Exp a)`, so it is safe for `iter0` to require that its argument has the more general type `(forall a.  Exp a)`.

```
iter0 :: (ExpF b -> b) -> (forall a. Exp a) -> b
iter0 = cata
```

However, this new type does not prevent expressions like `badcase` from being the argument to `iter0`. We can prevent such case analysis inside `lam` expressions by ruling out case analysis for *all* terms of type `Exp t`. If the user cannot use `case`, then they cannot write `badcase`. While this restriction means that some operations cannot be naturally defined in this calculus, `cata` alone can define a large number of operations, as we demonstrate below and in Section 2.2.

There are two ways to prohibit case analysis. The first way is to reimplement `Exp` in such a way that `cata` is the only possible operation (in other words without using a Haskell datatype). We discuss this alternative in Section 3.

The second way to prohibit case analysis is to make `Rec` an abstract type constructor. If the definition of `Rec` is hidden by some module boundary, such as with the interface in Figure 3, then the only way to destruct an expression of type `Exp a` is with `cata`. Because `Roll` and `Place` are datatype constructors of `Rec`, and `cata` pattern matches these constructors, they must all be defined in the same module as `Rec`. However, because we only need to prohibit case analysis, we can export `Roll` and `Place` as the functions `roll` and `place`. With `roll` we can define the terms `app` and `lam` anywhere.

```
type Rec a b -- abstract
data ExpF a  = Lam (a -> a) | App a a
type Exp a   = Rec ExpF a

roll  :: ExpF (Exp a) -> Exp a
place :: a -> Exp a
cata  :: (ExpF a -> a) -> Exp a -> a
```

Figure 3. Iteration library interface

We can also make good use of `place`. The type `forall a. Exp a` enforces that all embedded functions are parametric, but it can only represent *closed* expressions. What if we would like to examine expressions with free variables? In HOAS, an expression with one free variable has type `Exp t -> Exp t`. To compute the catamorphism for the expression, we use `place` to provide the value for the free variable.

```
openiter1 :: (ExpF b -> b)
  -> (Exp b -> Exp b) -> (b -> b)
openiter1 f x = \y -> cata f (x (place y))
```

If we would like to make sure that the expression is sound, we must quantify over the parameter type and require that the expression have type `forall a. Exp a -> Exp a`.

```
iter1 :: (ExpF b -> b)
         -> (forall a. Exp a -> Exp a) -> (b -> b)
iter1 = openiter1
```

With `iter1` we can determine if that one free variable occurs in an expression.

```
freevarused :: (forall a. Exp a -> Exp a) -> Bool
freevarused e =
  iter1 (\x -> case x of
               (App x y) -> x || y
               (Lam f) -> f False) e True
```

An app expression uses the free variable if either the function or the argument uses it. The occurrence of the bound variable of a `lam` is not an occurrence of the free variable, so `False` is the argument to `f`, but the expression does use the free variable if it appears somewhere in the body of the abstraction. Finally, the program works by feeding in `True` for the value of the free variable. If the result is `True` then it must have appeared somewhere in the expression.

There is no reason to stop with one free variable. There are an infinite number of related iteration operators, each indexed by the type inside the `forall`. The types of several such iterators are shown below. For example, the third one, `iterList`, may analyze expressions with arbitrary numbers of free variables.

```
iter2    :: (ExpF b -> b)
           -> (forall a. Exp a -> Exp a -> Exp a)
           -> (b -> b -> b)
iterFun  :: (ExpF b -> b)
           -> (forall a. (Exp a -> Exp a) -> Exp a)
           -> ((b -> b) -> b)
iterList :: (ExpF b -> b)
           -> (forall a. ([Exp a] -> Exp a))
           -> ([b] -> b)
```

Each of these iterators is defined by using `xmap` to map `(cata f)` and `place`. Thus we can easily implement them by defining the appropriate version of `xmap`. However, because `xmap` is a polytypic function, we should be able to automatically generate all of these iterators using Generic Haskell. The following code implements these operations. Below, the notation `xmap{|g|}` generates the instance of `xmap` for the type constructor `g`.

```
openiter{|g :: * -> * |} ::
  (ExpF a -> a) -> g (Exp a) -> g a
openiter{|g|} f =
  fst (xmap{|g|} (cata f, place))


iter{|g :: * -> * |} ::
  (ExpF a -> a) -> (forall b. g (Exp b)) -> g a
iter{|g|} = openiter{|g|}
```

Unfortunately, the above Generic Haskell code cannot automatically generate all the iterators that we want, such as `iter1`, `iterFun` and `iterList`. Because of type inference, `g` can only be a type constructor that is a constant or a constant applied to type constructors [13]. In particular, we cannot represent the type constructor $(\lambda\alpha : \star.\alpha \rightarrow \alpha)$ in Haskell, so we cannot automatically generate the instance

```
iter1 :: (f b -> b)
   -> (forall a. (Exp a) -> (Exp a)) -> b -> b
```

Fortunately, using a different extension of Haskell, called functional dependencies [14], we can generate these versions of `openiter`. For each version of `iter` that we want, we still need to redefine the generated `openiter` with the more restrictive type.

```
iter1 :: (ExpF a -> a)
          -> (forall b. Exp b -> Exp b) -> a -> a
iter1 = openiter
```

The `Iterable` class defines `openiter` simultaneously with its inverse. The parameters `m` and `n` should be `g(Exp a)` and `g a`, where each instance specifies `g`. (The type `a` is a parameter of the type class so that `m` and `n` may refer to it.) Also necessary are the functional dependencies that state that `m` determines both `a` and `n`. These dependencies rule out ambiguities during type inference.

```
class Iterable a m n | m -> a, m -> n where
    openiter :: (ExpF a -> a) -> m -> n
    uniter   :: (ExpF a -> a) -> n -> m
```

If `g` is the identity type constructor, then `m` and `n` are `Exp a` and `a` respectively.

```
instance Iterable a (Exp a) a where
    openiter  = cata
    uniter f  = place
```

Using the instances for the subcomponents, we can define instances for types that contain `->`.

```
instance (Iterable a m1 n1, Iterable a m2 n2)
    => Iterable a (m1 -> m2) (n1 -> n2) where
    openiter f x = openiter f . x . uniter f
    uniter f x   = uniter f . x . openiter f
```

With these instances, we have a definition for `openiter{|`$\lambda\alpha.\alpha \rightarrow \alpha$`|}`. It is not difficult to add instances for other type constructors, such as lists and tuples.

2.2 Examples of iteration

We next present several additional examples of the expressiveness of `iter0` for arguments of type `(forall a. Exp a)`. The purpose of these examples is to demonstrate how to implement some of the common operations for λ-calculus terms without case analysis.

For example, we can use `iter0` to convert expressions to strings. So that we have different names for each nested binding occurrence, we must parameterize this iteration with a list of variable names. Haskell's list comprehension provides us with an infinite supply of strings.

```
vars :: [String]
vars = [ [i] | i <- ['a'..'z'] ] ++
       [ i : show j | j <- [1..], i <- ['a'..'z'] ]

showAux :: ExpF ([String] -> String)
             -> ([String] -> String)
showAux (App x y) vars =
 "(" ++ (x vars) ++ " " ++ (y vars) ++ ")"
showAux (Lam z) (v:vars) =
 "(fn " ++ v ++ ". " ++ (z (const v) vars) ++ ")"

show :: (forall a. Exp a) -> String
show e = iter0 showAux e vars
```

Applying `show` to an expression produces a readable form of the expression.

```
show (lam (\x -> lam (\y -> app x y)))
  = (fn a. (fn b. (a b)))
```

Another operation we might wish to perform for a λ-calculus expression is to reduce it to a simpler form. As an example, we next implement parallel reduction for a λ-calculus expression.[6] Parallel reduction differs from full reduction in that it does not reduce any newly created redexes. Therefore, it terminates even for expressions with no β-normal form. Parallel reduction may be specified by the following inductive definition.

$$\frac{}{x \Rightarrow x} \qquad \frac{M \Rightarrow M'}{\lambda x.M \Rightarrow \lambda x.M'} \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N'}{MN \Rightarrow M'N'}$$

$$\frac{M \Rightarrow M' \quad N \Rightarrow N'}{(\lambda x.M)N \Rightarrow M'\{x/N'\}}$$

We use `iter0` to implement parallel reduction below. The tricky part is the case for applications. We must determine whether the first component of an application is a `lam` expression, and if so, perform the reduction. However, we cannot do a case analysis on expressions, as the type `Exp a` is abstract. Therefore, we implement parallel reduction with a "pairing" trick[7]. As we iterate over the term we produce *two* results, stored in the following record:

```
data PAR a = PAR { par   :: Exp a,
                   apply :: Exp a -> Exp a }
```

The first component, `par`, is the actual result we want—the parallel reduction of the term. The second component, `apply`, is a function that we build up for the application case. In the case of a `lam` expression, `apply` performs the substitution in the reduced term. Otherwise, `apply` creates an `app` expression with its argument and the reduced term.[8]

```
parAux :: ExpF (PAR a) -> PAR a
parAux (Lam f) =
  PAR { par   = lam (par . f . var),
        apply = par . f . var }
  where
    var :: Exp a -> PAR a
    var x = PAR { par = x, apply = app x }
parAux (App x y) =
  PAR { par   = apply x (par y),
        apply = app (apply x (par y)) }

parallel :: (forall v. Exp v) -> (forall v. Exp v)
parallel x = par (iter0 parAux x)
```

For example:

```
show (parallel (app (lam (\x -> app x x))
                    (lam (\y -> y))))
= "((fn a. a) (fn a. a))"
```

While we could not write the most natural form of parallel reduction with `iter0`, other operations may be expressed in a very natural manner. For example, we can implement the one-pass call-by-value CPS-conversion of Danvy and Filinski [6]. This sophisticated algorithm performs "administrative" redexes at the meta-level so that the result term has no more redexes than the original expression. The algorithm is based on two mutually recursive operations: `cpsmeta` performs closure conversion given a meta-level continuation (a term of type `Exp a -> Exp a`), and `cpsobj` does the same with an object-level continuation (a term of type `Exp a`).

```
data CPS a = CPS {
   cpsmeta :: (Exp a -> Exp a) -> Exp a,
   cpsobj  :: Exp a -> Exp a }
```

If we are given a value (i.e. a λ-expression or a variable) the function `value` below describes its CPS conversion. Given a meta-continuation `k`, we apply `k` to the value. Otherwise, given an object continuation `c`, we create an object application of `c` to the value.

```
value  :: Exp a -> CPS a
value x = CPS { cpsmeta = \k -> k x,
                cpsobj  = \c -> app c x }
```

The operation `cpsAux` takes an expression whose subcomponents have already been CPS converted and CPS converts it. For application, translation is the same in both cases except that the meta-case converts the meta-continuation into an object continuation with `lam`.

```
cpsAux :: ExpF (CPS a) -> CPS a
cpsAux (App e1 e2) =
  CPS { cpsmeta = \k -> appexp (lam k),
        cpsobj  = appexp }
  where appexp c =
          (cpsmeta e1) (\y1 ->
             (cpsmeta e2) (\y2 ->
                app (app y1 y2) c))
```

[6] This example is from Schürmann et. al [24].

[7] Pairing was first used to implement the predecessor operation for Church numbers. The iteration simultaneously computes the desired result with auxiliary operations.

[8] In Haskell, the notation `apply x` projects the `apply` component from the record `x`.

```
type Rec f a = (f a -> a) -> a
data ExpF a = Lam (a -> a) | App a a
type Exp a = Rec ExpF a

roll :: ExpF (Exp a) -> Exp a
roll x =
  \f -> f (fst (xmapExpF (cata f, place)) x)

place :: a -> Exp a
place x = \f -> x

lam :: (Exp a -> Exp a) -> Exp a
lam x   = roll (Lam x)

app :: Exp a -> Exp a -> Exp a
app y z = roll (App y z)

xmapExpF :: (a -> b, b -> a)
   -> (ExpF  a -> ExpF  b, ExpF  b -> ExpF  a)
xmapExpF (f,g) = (\x -> case x of
                     Lam x   -> Lam (f . x . g)
                     App y z -> App (f y) (f z),
                \x -> case x of
                     Lam x   -> Lam (g . x . f)
                     App y z -> App (g y) (g z))

cata :: (ExpF a -> a) -> Exp a -> a
cata f x = x f

iter0 :: (ExpF a -> a) -> (forall b. Exp b) -> a
iter0 = cata
```

Figure 4. Catamorphism in the F_ω fragment of Haskell

For functions, we use value, but we must transform the function to bind both the original and continuation arguments and transform the body of the function to use this object continuation. The outer lam binds the original argument. We use value for this argument in f and cpsobj yields a body expecting an object continuation that the inner lam converts to an expression.

```
cpsAux (Lam f) =
  value (lam (lam . cpsobj . f . value))
```

Finally, we start cps with iter0 by abstracting an arbitrary dynamic context a and transforming the argument with respect to that context.

```
cps :: (forall a. Exp a) -> (forall a. Exp a)
cps x = lam (\a ->
          cpsmeta (iter0 cpsAux x) (\m -> app a m))

show (cps (lam (\x -> app x x)))
= "(fn a. (a (fn b. (fn c. ((b b) c)))))"
```

Above, a is the initial continuation, b is the argument x, and c is the continuation for the function.

3 Encoding iteration in F_ω

In the previous section, we implemented iter as a recursive function and used a recursive type, Rec, to define Exp. To prevent case analysis, we hid this definition of Rec behind a module boundary. However, this module abstraction and is not the only way to prevent case analysis. Furthermore, term and type recursion is not necessary to implement this datatype. We may define iter and Rec in the fragment of Haskell that corresponds to F_ω [10] so that iteration is the only elimination form for Rec. This implementation appears in Figure 4.

The encoding is similar to the encoding of covariant datatypes in the polymorphic λ-calculus [3] (or to the encoding of Church numerals). We encode an expression of type Exp a as its elimination form. For example, something of type Exp a should take an elimination function of type (ExpF a -> a) and return an a. To implement cata we apply the expression to the elimination function.

To create an expression, roll must encode this elimination. Therefore, roll returns a function that applies its argument f (the elimination function) to the result of iterating over x. Again, to use xmap we need a right inverse for cata f. The term place in Figure 4 is an expression that when analyzed returns its argument. We can show that place is a right inverse by expanding the above definitions:

```
cata f . place = (\x -> cata f (place x))
               = (\x -> (place x) f)
               = (\x -> ((\y -> x) f))
               = (\x -> x)
```

3.1 Reasoning about iteration

There are powerful tools for reasoning about programs written in the polymorphic λ-calculus. For example, we know that all programs that are written in F_ω will terminate. Therefore, we can argue that the examples of the previous section are total on all inputs that may be expressed in the polymorphic λ-calculus, such as app (lam (\x -> app x x))(lam (\x -> app x x)). Unfortunately, we cannot argue that these examples are total for arbitrary Haskell terms. For example, calling any of these routines on (lam (let f x = f x in f)) will certainly diverge. Furthermore, even if the arguments to iteration are written in F_ω, if the operation itself uses type or term recursion, then it could still diverge. For example, using the recursive datatype Value from Section 2, we can implement the untyped λ-calculus evaluator with iter0.

Parametricity is another way to reason about programs written in F_ω. As awkward as they may be, one of the advantages to programming with catamorphisms instead of general recursion is that we may reason about our programs using algebraic laws that follow from parametricity. While the following laws only hold for F_ω, we may be able to prove some form of them for Haskell using techniques developed by Johann [12].

Using parametricity, we can derive a *free theorem* [28] about expressions of type (forall a. (b a -> a) -> a). If x has this type, then

```
f . f' = id and f . g = h . fst (xmap|b|(f,f')) =>
f (x g) = x h
```

The equivalence in this theorem is equivalence in some parametric model of F_ω, such as the term model with βη-equivalence. Using the free theorem, we can prove a number of properties about iteration. First, we can show that iterating roll is an identity function, that iter0 roll = id. Using this result we can show the *uniqueness property* for iter, which describes when a function is equal to an application of iter. It resembles an "induction principle" for iter0.

```
f . f' = id and f . roll = h . fst (xmap|b|(f,f'))
 <=>  f = iter0 h
```

The <= direction follows directly from the implementation of `iter0` and `roll`. The => direction follows from the free theorem.

Finally, the *fusion law* can be used to combine the composition of a function `f` and an iteration into one iteration. This law follows directly from the free theorem.

```
f . f' = id  and  f . g = h . fst(xmap|b|(f,f'))  =>
f . iter0 g = iter0 h
```

However, there is an important property about this encoding of the λ-calculus that we have not proven. *Adequacy* states that if a F_ω term is of type `forall a. Exp a` and is in canonical form, then it should be the encoding of the canonical form of some λ-calculus expression. In other words, there is no extra "junk" in the type `forall a. Exp a`, such as `badcase`. As a first step towards proving this result, we next show how this F_ω library can encode a language with iteration over HOAS that itself adequately embeds the λ-calculus.

4 Enforcing parametricity with modal types

In the next section, we formally describe the connection between the interface we have provided for iteration over higher-order abstract syntax and the modal calculus of Schürmann, Despeyroux and Pfenning (SDP) [24]. We do so by using this library to give a sound and complete embedding of the SDP calculus into F_ω. First, we provide a brief overview of the static and dynamic semantics of this calculus. The syntax of the SDP calculus is shown in Figure 5.

The SDP calculus enforces the parametricity of function spaces with modal types. Modal necessity in logic is used to indicate those propositions that are true in all worlds. Consequently, these propositions can make use of only those assumptions that are also true in all worlds. In Pfenning and Davies' [20] interpretation of modal necessity, necessarily true propositions correspond to those formulae that can be shown to be valid. Validity is defined as derivable with respect to only assumptions that themselves are valid assumptions. As such, the typing judgments have two environments (also called contexts), one for valid assumptions, Ω, and one for "local" assumptions, ϒ. The terms corresponding to the introduction and elimination forms for modal necessity are **box** and **let box**. We give them the following typing rules:

$$\frac{\Omega;\varnothing \vdash M : A}{\Omega;\Upsilon \vdash \mathbf{box}\,M : \Box A}$$

$$\frac{\Omega;\Upsilon \vdash M_1 : \Box A_1 \quad \Omega \uplus \{x : A_1\};\Upsilon \vdash M_2 : A_2}{\Omega;\Upsilon \vdash \mathbf{let\ box}\ x : A_1 = M_1\ \mathbf{in}\, M_2 : A_2}$$

A **box**ed term, M, has type $\Box A$ only if it has type A with respect to the valid assumptions in Ω, and no assumptions in local environment. The **let box** elimination construct allows for the introduction of valid assumptions into Ω, binding the contents of the boxed term M_1 in the body M_2. This binding is allowed because the contents of **box**ed terms are well-typed themselves with only valid assumptions. Another way to think about modal necessity is that terms with boxed type are "closed" and do not contain any free variables, except those that are bound to closed terms themselves.

Operationally, **box**ed terms behave like suspensions, while **let box** substitutes the contents of a **box**ed term for the bound variable. Because the operational semantics is defined simultaneously with conversion to canonical forms, it is parameterized by the environment Ψ that describes the types of free local variables appearing in the expression.

$$\frac{\Psi \vdash M_1 \hookrightarrow \mathbf{box}\,M_1' : \Box A_1 \quad \Psi \vdash M_2\{M_1'/x\} \hookrightarrow V : A_2}{\Psi \vdash \mathbf{let\ box}\ x : A_1 = M_1\ \mathbf{in}\, M_2 \hookrightarrow V : A_2}$$

To enforce the separation between the iterative and parametric function spaces, the SDP calculus defines those types, B, that do not contain a $\Box$ type to be "pure". Objects in the calculus with type $\Box B$, boxed pure types, can be examined intensionally using an iteration operator, while objects of arbitrary impure type, A, cannot. This forces functions of pure type, $\lambda x : B_1..M : B_1 \to B_2$, to be *parametric*. This is because the input, x, to such a function does not have a boxed pure type, and there is no way to convert it to one — x will not be free inside of a **box**ed expression in M. Consequently, the functions of pure type may only treat their inputs extensionally, making them parametric.

(*Pure Types*)	B	$::= b \mid 1 \mid B_1 \to B_2 \mid B_1 \times B_2$
(*Types*)	A	$::= B \mid A_1 \to A_2 \mid A_1 \times A_2 \mid \Box A$
(*Terms*)	M	$::= x \mid c \mid \langle\rangle \mid \lambda x : A.M \mid M_1 M_2 \mid$ $\mathbf{box}\,M \mid \mathbf{let\ box}\ x : A = M_1\ \mathbf{in}\, M_2 \mid$ $\langle M_1, M_2\rangle \mid \mathbf{fst}\,M \mid \mathbf{snd}\,M \mid$ $\mathbf{iter}\,[A_1, A_2][\Theta]\ M$
(*Term Replacement*)	Θ	$::= \varnothing \mid \Theta \uplus \{x \mapsto M\} \mid \Theta \uplus \{c \mapsto M\}$
(*Pure Environment*)	Ψ	$::= \varnothing \mid \Psi \uplus \{x : B\}$
(*Valid Environment*)	Ω	$::= \varnothing \mid \Omega \uplus \{x : A\}$
(*Local Environment*)	Υ	$::= \varnothing \mid \Upsilon \uplus \{x : A\}$
(*Signatures*)	Σ	$::= \varnothing \mid \Sigma \uplus \{c : B \to b\}$

Figure 5. Syntax of SDP calculus

The language is parameterized by a constant type b and a *signature*, Σ, of *data constructor constants*, c, for that base type. Each of the constructors in this signature must be of type $B \to b$. Because B is a pure type, these constructors may only take *parametric* functions as arguments.

For example, consider a signature describing the untyped λ-calculus, $\Sigma = \{\mathbf{app} : b \times b \to b, \mathbf{lam} : (b \to b) \to b\}$, where the constant type b corresponds to `Exp`. Using this signature, we can write a function to count the number of bound variables in an expression, as we did in Section 2.[9]

$$\begin{array}{l} \mathbf{countvar} \triangleq \lambda x : \Box b. \\ \quad \mathbf{iter}[\Box b, \mathsf{int}][\{\mathbf{app} \mapsto \lambda y : \mathsf{int}.\lambda z : \mathsf{int}.y + z, \\ \qquad\qquad \mathbf{lam} \mapsto \lambda f : \mathsf{int} \to \mathsf{int}.f\,1\}]\ x \end{array}$$

The term **iter** intensionally examines the structure of the argument x and replaces each occurrence of **app** and **lam** with $\lambda y : \mathsf{int}.\lambda z : \mathsf{int}.y + z$ and $\lambda f : \mathsf{int} \to \mathsf{int}.f\,1$ respectively.

The typing rule for **iter** is the following:

$$\frac{\Omega;\Upsilon \vdash M : \Box B \quad \Omega;\Upsilon \vdash \Theta : A\langle\Sigma\rangle}{\Omega;\Upsilon \vdash \mathbf{iter}\,[\Box B, A][\Theta]\ M : A\langle B\rangle}$$

The argument to iteration, M, must have a pure closed type to be analyzable. Analysis proceeds via walking over M and using the

[9] For simplicity, our formal presentation of SDP (in Figure 5) does not include integers. However, it is straightforward to extend this calculus to additional base types.

replacement Θ, a finite map from constants to terms, to substitute for the constants in the term M. The type A is the type that will replace the base type b in the result of iteration. The notation $A\langle B\rangle$ substitutes A for the constant b in the pure type B. Each term in the range of the replacements must also agree with replacing b with A. We verify this fact with the judgment $\Omega;\Upsilon \vdash \Theta : A\langle\Sigma\rangle$, which requires that if $\Theta(c) = M_c$ and $\Sigma(c) = B_c$, then M_c must have type $A\langle B_c\rangle$.

Operationally, iteration in the SDP calculus works in the following fashion.

$$\frac{\begin{array}{c}\Psi \vdash M \hookrightarrow \mathbf{box}\, M' : \Box B \\ \varnothing \vdash M' \Uparrow V' : B \\ \Psi \vdash \langle A, \Psi, \Theta\rangle(V') \hookrightarrow V : A\langle B\rangle\end{array}}{\Psi \vdash \mathbf{iter}\,[\Box B, A]\,[\Theta]\; M \hookrightarrow V : A\langle B\rangle}$$

First, the argument to iteration M is evaluated, $\Psi \vdash M \hookrightarrow \mathbf{box}\, M' : \Box B$, producing a **box**ed object M'. M' is then evaluated to η-long canonical form via $\varnothing \vdash M' \Uparrow V' : B$. Next we perform elimination of that canonical form, $\langle A, \Psi, \Theta\rangle(V')$, walking over V' and using Θ to replace the occurrences of constants. Finally, we evaluate that result, $\Psi \vdash \langle A, \Psi, \Theta\rangle(V') \hookrightarrow V : A\langle B\rangle$.

In order to simplify the presentation of the encoding, we have made a few changes to the SDP calculus. First, while the language presented in this paper has only one pure base type b, the SDP calculus allows the signature Σ to contain arbitrarily many base types. However, the extension of the encoding to several base types is straightforward. Also, in order to make the constants of the pure language more closely resemble datatype constructors, we have forced them all to be of the form $B \to b$ instead of any arbitrary pure type B. To facilitate this restriction, we add unit and pairing to the pure fragment of the calculus so that constructors may take any number of arguments.

5 Encoding SDP in F_ω

The terms that we defined in Section 3, `roll` and `iter`, correspond very closely to the constructors and iteration primitive of the SDP calculus. In this section, we strengthen this observation by showing how to encode all programs written in the SDP calculus into F_ω using a variation of these terms.

There are two key ideas behind our encoding:

- We use type abstraction to ensure that the encoding of **box**ed objects obeys the closure property of the source language, and prevents variables from the local environment from appearing inside these terms. To do so, we parameterize our encoding by a type that represents the current world and maintain the invariant that all variables in the local environment mention the current world in their types. Because a term enclosed within a **box** must be well-typed in any world, when we encode a **box**ed term we use a fresh type variable to create an arbitrary world. We then encode the enclosed term with that new world and wrap the result with a type abstraction. As a consequence, the encoding of a data-structure within a **box** cannot contain free local variables because their types would mention that fresh type variable outside of the scope of the type abstraction.
- We encode constants in the source language as their elimination form with `roll`. Furthermore, we restrict the result of elimination to be of the type that is the world in which the term was encoded. However, the encoding of **box**ed expressions quantifies over that world, allowing the resulting computations to be of arbitrary type.

The encoding of the SDP calculus can be broken into four primary pieces: the encodings for signatures, types, terms, and replacements. To simplify our presentation, we extend the target language with unit, void, products, and variants. The syntax of these terms appears in Figure 6. This extension does not weaken our results as there are well known encodings of these types into F_ω. In the remainder of this section, we present the details of the encoding and describe the most interesting cases. The full specification of this encoding appears in Appendix B.

Signatures. The encoding of signatures in the SDP calculus, notated $\tau\langle\Sigma\rangle$, corresponds to generating the type constructor whose fixed point defines the recursive datatype. (For example, `ExpF` in Section 2.) The argument of the encoding, a specified world τ, corresponds to the argument of the type constructor.

For this encoding, we assume the aid of an injective function $\mathcal{L}$ that maps data constructors in the source language to distinct labels in the target language. We also need an operation called *parameterization*, notated $\tau\langle B\rangle$ and defined in Appendix B.1. This operation parameterizes pure types in the source calculus with respect to a given world in the target language, and produces a type in the target language. Essentially, $\tau\langle B\rangle$ "substitutes" the type τ for the base type, b, in B.

We encode a signature as a variant. Each field corresponds to a constant c_i in the signature, with a label according to $\mathcal{L}$, and a type that is the result of parameterizing the argument type of c_i with the provided type.

$$\frac{\forall c_i \in \mathrm{dom}(\Sigma) \quad \Sigma(c_i) = B_i \to b}{\tau\langle\Sigma\rangle \triangleq \langle \mathcal{L}(c_1) : \tau\langle B_1\rangle, \ldots, \mathcal{L}(c_n) : \tau\langle B_n\rangle\rangle}$$

We often use parameterization and the signature translation to build type constructors in the target language, so we define the following two abbreviations:

$$B^* \triangleq \lambda\alpha:\star.\alpha\langle B\rangle \qquad \Sigma^* \triangleq \lambda\alpha:\star.\alpha\langle\Sigma\rangle$$

Types. As with the encoding of signatures, the encoding of types is parameterized by the worlds in which they occur. We write the judgment for encoding a type A in the source calculus in world τ as $\Delta \vdash A \rhd_\tau \tau'$. The environment Δ tracks type variables allocated during the translation and allows us to chose variables that are not in scope. The two interesting cases for encoding types from the source calculus are those for the base type and for boxed types. The case for b corresponds to `Rec ExpF a` from Section 3. Therefore, we define the abbreviation $\mathsf{Rec}\,\Sigma^*\,\alpha \triangleq (\Sigma^*\alpha \to \alpha) \to \alpha$, intuitively a fixed point of Σ^*, to the same idea of encoding a datatype as its elimination form.

$$\frac{}{\Delta \vdash b \rhd_\tau \mathsf{Rec}\,\Sigma^*\,\tau}$$

The rule for boxed types uses type abstraction to ensure the result is parametric with respect to its world. Naïvely, we might expect to use a fresh type variable as the new world and then encode the contents of the boxed type with that type variable. This encoding ensures that the type is parametric with respect to its world and then quantifies over the result.

$$\frac{\alpha \notin \Delta \quad \Delta \uplus \{\alpha:\star\} \vdash A \rhd_\alpha \tau'}{\Delta \vdash \Box A \rhd_\tau \forall\alpha{:}\star.\tau'}\ \text{WRONG!}$$

$$
\begin{array}{lrcl}
(\textit{Kinds}) & \kappa & ::= & \star \mid \kappa_1 \to \kappa_2 \\
(\textit{Types}) & \tau & ::= & 1 \mid 0 \mid \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha{:}\kappa.\tau \mid \tau_1 \times \tau_2 \mid \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \mid \lambda\alpha:\kappa.\tau \mid \alpha \mid \tau_1\tau_2 \\
(\textit{Terms}) & e & ::= & x \mid \langle\rangle \mid \lambda x:\tau.e \mid e_1 e_2 \mid \Lambda\alpha{:}\kappa.e \mid e[\tau] \mid \langle e_1, e_2 \rangle \mid \mathbf{fst}\, e \mid \mathbf{snd}\, e \mid \mathbf{inj}_l\, e\, \mathbf{of}\, \tau \mid \\
& & & \mathbf{case}\, e\, \mathbf{of}\, \mathbf{inj}_{l_1}\, x_1\, \mathbf{in}\, e_1 \ldots \mathbf{inj}_{l_n}\, x_n\, \mathbf{in}\, e_n \\
(\textit{Type Variable Environment}) & \Delta & ::= & \varnothing \mid \Delta \uplus \{\alpha : \kappa\} \\
(\textit{Term Environment}) & \Gamma & ::= & \varnothing \mid \Gamma \uplus \{x : \tau\}
\end{array}
$$

Figure 6. Syntax of F_ω with unit, void, products, and variants

However, with this encoding we violate the invariant that the types of all free local variables mention the current world, because the encoding does not involve τ. Instead, we use the fresh type variable to create a new world from the current world and consider α as a "world transformer". During the translation, a term will be encoded with a stack of world transformers, somewhat akin to stack of environments in the implicit formulation of modal types [7].

$$\frac{\alpha \notin \Delta \quad \Delta \uplus \{\alpha : \star \to \star\} \vdash A \rhd_{\alpha\tau} \tau'}{\Delta \vdash \Box A \rhd_\tau \forall\alpha{:}\star \to \star.\tau'}$$

The naïve translation of the unit type also forgets the current world. For this reason, we add a non-standard unit to F_ω that is parameterized by the current world. In other words, the unit type 1 is of kind $\star \to \star$ and the unit term $\langle\rangle$ has type $\forall\alpha{:}\star.1(\alpha)$. Our type translation instantiates this type with the current world.

$$\frac{}{\Delta \vdash 1 \rhd_\tau 1[\tau]}$$

The remaining types in the SDP language are encoded recursively in a straightforward manner. The complete rules can be found in Appendix B.3.

Terms and replacements. We encode the source term, M, with the judgment $\Delta;\Xi \vdash M \rhd_\tau e$. In addition to the current world, τ, and the set of allocated type variables, Δ, the encoding of terms is also parameterized by a set of term variables, Ξ. This set of variables allows the encoding to distinguish between variables that were bound with λ and those bound with **let box**. We will elaborate on why this set is necessary shortly.

Our encoding of **box**ed terms follows immediately from the encoding of **box**ed types. Here we encode the argument term with respect to a fresh world transformer applied to the present world and then wrap the result with a type abstraction.

$$\frac{\alpha \notin \Delta \quad \Delta \uplus \{\alpha : \star \to \star\};\Xi \vdash M \rhd_{\alpha\tau} e}{\Delta;\Xi \vdash \mathbf{box}\, M \rhd_\tau \Lambda\alpha{:}\star \to \star.e}$$

We encode **let box** by converting it to an abstraction and application in the target language. However, one might note the discrepancy between the type of the variable we bind in the abstraction and the type we might naïvely expect.

$$\frac{\begin{array}{c}\Delta \vdash \Box A_1 \rhd_\tau \tau_1 \\ \Delta;\Xi \vdash M_1 \rhd_\tau e_1 \quad \Delta;\Xi \uplus \{x\} \vdash M_2 \rhd_\tau e_2\end{array}}{\Delta;\Xi \vdash \mathbf{let\ box}\, x : A_1 = M_1\, \mathbf{in}\, M_2 \rhd_\tau (\lambda x : \tau_1.e_2)e_1}$$

The type of x is A_1 and so one might assume that the type of x in the target should be the encoding of A_1 in the world τ. However, **let box** allows us to bind variables that are accessible in any world and using A_1 encoded against τ would allow the result to be used only in the present world. Because the encoding of M_1 will evaluate to a type abstraction, a term parametric in its world, we do not immediately unpack it by instantiating it with the current world. Instead we pass it as x and then, when x appears we instantiate it with the current world. Consequently, we use Ξ to keep track of variables bound by **let box**. When encoding variables, we check whether x occurs in Ξ and perform instantiations as necessary.

$$\frac{x \notin \Xi}{\Delta;\Xi \vdash x \rhd_\tau x} \qquad \frac{x \in \Xi}{\Delta;\Xi \vdash x \rhd_\tau x[\lambda\alpha : \star.\tau]}$$

If the variable is in Ξ, then it is applied to a world transformer that ignores its argument, and returns the present world. This essentially replaces the bottom of the world transformer stack captured by the type abstraction substituted for x with the world τ. Doing so ensures that if we substitute the encoding of a **box**ed term into the encoding of another **box**ed term, the type correctness of the embedding is maintained by correctly propagating the enclosing world.

$\mathbf{cata} : \forall\alpha{:}\star.(\Sigma^*\alpha \to \alpha) \to (\mathsf{Rec}\ \Sigma^*\ \alpha) \to \alpha$
$\mathbf{cata} \triangleq \Lambda\alpha{:}\star..\lambda f : (\Sigma^*\alpha \to \alpha).\lambda y : (\mathsf{Rec}\ \Sigma^*\ \alpha).yf$

$\mathbf{place} : \forall\alpha{:}\star.\alpha \to \mathsf{Rec}\ \Sigma^*\ \alpha$
$\mathbf{place} \triangleq \Lambda\alpha{:}\star.\lambda x : \alpha.\lambda f : (\Sigma^*\alpha \to \alpha).x$

$\mathbf{xmap}\{\!|\tau|\!\} : \forall\alpha{:}\star.\forall\beta{:}\star.(\alpha \to \beta \times \beta \to \alpha) \to$
$\quad (\tau\alpha \to \tau\beta \times \tau\beta \to \tau\alpha)$

$\mathbf{openiter}\{\!|\tau|\!\} : \forall\alpha{:}\star.(\Sigma^*\alpha \to \alpha) \to \tau(\mathsf{Rec}\ \Sigma^*\ \alpha) \to \tau\alpha$
$\mathbf{openiter}\{\!|\tau|\!\} \triangleq \Lambda\alpha{:}\star.\lambda f : \Sigma^*\alpha \to \alpha.$
$\quad \mathbf{fst}\,(\mathbf{xmap}\{\!|\tau|\!\}[\mathsf{Rec}\ \Sigma^*\ \alpha][\alpha]\langle \mathbf{cata}[\alpha]f, \mathbf{place}[\alpha]\rangle)$

$\mathbf{iter}\{\!|\tau|\!\} : \forall\gamma{:}\star.\forall\alpha{:}\star.(\Sigma^*\alpha \to \alpha) \to$
$\quad (\forall\beta{:}\star \to \star.\tau(\mathsf{Rec}\ \Sigma^*\ (\beta\gamma)) \to \tau\alpha$
$\mathbf{iter}\{\!|\tau|\!\} \triangleq \Lambda\gamma{:}\star.\Lambda\alpha{:}\star.\lambda f : \Sigma^*\alpha \to \alpha.$
$\quad \lambda x : (\forall\beta : \star \to \star.\tau(\mathsf{Rec}\ \Sigma^*\ (\beta\gamma))).\mathbf{openiter}\{\!|\tau|\!\}[\alpha]f(x[\alpha])$

$\mathbf{roll} : \forall\alpha{:}\star.\Sigma^*(\mathsf{Rec}\ \Sigma^*\ \alpha) \to \mathsf{Rec}\ \Sigma^*\ \alpha$
$\mathbf{roll} \triangleq \Lambda\alpha{:}\star.\lambda x : \Sigma^*(\mathsf{Rec}\ \Sigma^*\ \alpha).$
$\quad \lambda f : \Sigma^*\alpha \to \alpha.f(\mathbf{openiter}\{\!|\Sigma^*|\!\}[\alpha]\ f\ x)$

Figure 7. Library routines

Figure 7 shows the types and definitions of the library routines used by the encoding. The only difference between it and Figure 4 is that **iter** abstracts the current world and requires that its argument be valid in any transformation of the current world. Again, we make use of the polytypic function **xmap** to lift **cata** to arbitrary type constructors . Because **xmap** is defined by the structure of a type constructor τ, we cannot directly define it as a term in F_ω. Instead, we will think of $\mathbf{xmap}\{\!|\tau|\!\}$ as macro that expands to the mapping function for the type constructor τ. (We use the notation $\{\!|\cdot|\!\}$ to distinguish between polytypic instantiation and parametric type instantiation.) This expansion is done according to the definition in Appendix A. We do not cover the implementation here, see Hinze [11] for details.

Encoding constants in the source calculus makes straightforward use of the library routine **roll**. We simply translate the constant into an abstraction that accepts a term that is the encoding of the argument of the constant, and then uses **roll** to transform the injection into the encoding of the base type, Rec Σ^* τ.

$$\frac{\Sigma(c) = B \to b \quad \Delta \vdash B \rhd_\tau \tau_B}{\Delta; \Xi \vdash c \rhd_\tau \lambda x : \tau_B.\mathbf{roll}[\tau](\mathbf{inj}_{\mathcal{L}(c)}\, x \ \mathbf{of}\, \Sigma^*(\mathsf{Rec}\ \Sigma^*\ \tau))}$$

The encoding of iteration is similarly straightforward. We instantiate our polytypic function **iter** with a type constructor created from parameterizing B, and then apply it to the current world and the encodings of the intended result type A, the replacement term Θ and argument term M.

$$\frac{\Delta \vdash A \rhd_\tau \tau_A \quad \Delta; \Xi \vdash \Theta \rhd_\tau^{\tau_A} e_\Theta \quad \Delta; \Xi \vdash M \rhd_\tau e_M}{\Delta; \Xi \vdash \mathbf{iter}\,[\Box B, A][\Theta]\ M \rhd_\tau \mathbf{iter}\{\!|B^*|\!\}[\tau][\tau_A]\, e_\Theta\, e_M}$$

The encoding of replacements Θ is uncomplicated and analogous to the encoding of signatures. We construct an abstraction that consumes an instance of an encoded signature, dispatching the variant using a **case** expression. In each branch, the encoding of the corresponding replacement is applied to the argument of the injection.

$$\frac{\forall c_i \in \mathrm{dom}(\Theta) \quad \Delta; \Xi \vdash \Theta(C_i) \rhd_\tau e_i}{\begin{array}{l}\Delta; \Xi \vdash \Theta \rhd_\tau^{\tau_A} \lambda x : \Sigma^*\tau_A.\mathbf{case}\, x\, \mathbf{of}\, \mathbf{inj}_{\mathcal{L}(c_1)}\, y_1\, \mathbf{in}\, (e_1 y_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathbf{inj}_{\mathcal{L}(c_n)}\, y_n\, \mathbf{in}\, (e_n y_n)\end{array}}$$

The encodings for the other terms in the source language are straightforward and appear in Appendix B.4. Now that we have defined all of our encoding for any closed term M in the SDP calculus, we put everything together to construct a term e in our target calculus using the initial judgment $\varnothing; \varnothing \vdash M \rhd_0 e$. We use the void type as the initial world to enforce the parametricity of unboxed constants.

5.1 Properties of the encoding

We have proven a number of desirable properties concerning this encoding. However, before we can state these properties, we must first define the relationship between the environments in the source and target calculi. These relations hold when all types from the local environment are encoded with the current world, and all types from the valid environment are first boxed then encoded with any world.

DEFINITION 5.1 (ENCODING TYPING ENVIRONMENTS). *We write $\Delta \vdash \Upsilon \rhd_\tau \Gamma_1$ and $\Delta \vdash \Omega \rhd \Gamma_2$ to mean that*

$$\begin{array}{ll}\forall x : A \in \Upsilon, x : \tau_A \in \Gamma_1 & \textit{when } \Delta \vdash A \rhd_\tau \tau_A \\ \forall x : A \in \Omega, x : \tau_A \in \Gamma_2 & \textit{when there exists } \Delta \vdash \tau' \textit{ such that} \\ & \qquad \Delta \vdash \Box A \rhd_{\tau'} \tau_A\end{array}$$

The relation for valid environments above is not parameterized by the current world. A single valid environment may be encoded as many different target environments, depending on what worlds are chosen for each type in the environment. However, in some sense the encodings are equivalent. If the translation of M type checks with one encoding of Ω, it will type check with any other encoding of Ω.

The encoding is type preserving. If we encode a well-typed term M, the resulting term will be well-typed under the appropriately translated environment. Furthermore, the converse is also true. If the encoding of a term M is well-typed in the target language, then M must have been well-typed in the source. This means that the target language preserves the abstractions of the source language.

THEOREM 5.2 (STATIC CORRECTNESS). *Assume $\Delta \vdash \tau$ and $\Delta \vdash \Upsilon \rhd_\tau \Gamma_1$ and $\Delta \vdash \Omega \rhd \Gamma_2$.*

1. *If $\Delta; \mathrm{dom}(\Omega) \vdash M \rhd_\tau e$ then*

$$\Omega; \Upsilon \vdash M : A \textit{ and } \Delta \vdash A \rhd_\tau \tau_A \ \Leftrightarrow\ \Delta; \Gamma_1 \uplus \Gamma_2 \vdash e : \tau_A$$

2. *If $\Delta; \mathrm{dom}(\Omega) \vdash \Theta \rhd_\tau^{\tau_A} e_\Theta$ then*

$$\Omega; \Upsilon \vdash \Theta : A\langle\Sigma\rangle \textit{ and } \Delta \vdash A \rhd_\tau \tau_A \ \Leftrightarrow\ \Delta; \Gamma_1 \uplus \Gamma_2 \vdash e_\Theta : \tau_A\langle\Sigma\rangle \to \tau_A$$

PROOF. By mutual induction over the translation of terms $(\Delta; \mathrm{dom}(\Omega) \vdash M \rhd_\tau e)$ and of replacements $(\Delta; \mathrm{dom}(\Omega) \vdash \Theta \rhd_\tau^{\tau_A} e_\Theta)$. ☐

Furthermore, source evaluation and canonicalization is the same as $\beta\eta$-equivalence in the target calculus.

THEOREM 5.3 (DYNAMIC CORRECTNESS). *If $\varnothing; \Psi \vdash M : A$ and $\varnothing; \Psi \vdash M \rhd_\tau e$ and $\varnothing; \Psi \vdash V \rhd_\tau e'$ and $\varnothing \vdash A \rhd_\tau \tau_A$ and $\varnothing; \Xi \vdash \varnothing; \Psi \rhd_\tau \Gamma$ then*

1. *$\Psi \vdash M \hookrightarrow V : A \ \Leftrightarrow\ \varnothing; \Gamma \vdash e \equiv_{\beta\eta} e' : \tau_A$.*

2. *$\Psi \vdash M \Uparrow V : A \ \Leftrightarrow\ \varnothing; \Gamma \vdash e \equiv_{\beta\eta} e' : \tau_A$.*

PROOF. The forward direction follows by simultaneous induction on the evaluation of M ($\Psi \vdash M \hookrightarrow V : A$) and the conversion of M to canonical form ($\Psi \vdash M \Uparrow V : A$). The reverse direction follows from the forward direction and from the fact that evaluation in the SDP calculus is deterministic and total. ☐

6 Future work

Although we have shown a very close connection between SDP and its encoding in F_ω, we have not shown that this encoding is adequate. We would like to show that if τ is the image of an SDP type, then all terms of type τ are equivalent to the encoding of some SDP term. In other words, there is no extra "junk" of type τ. Showing this result would also show that encoding the λ-calculus with **app** and **lam** is adequate, because the SDP calculus can already adequately encode the λ-calculus.

Alternatively, we could try to show adequacy with respect to the λ-calculus directly using a different method. It may also be possible to do so for the simpler encoding of modal types, informally presented in the first part of the paper, that uses first-order quantification and discards the current world. Whereas this simpler encoding allows the translation of some terms that are rejected by the SDP calculus to type check (for example, $\lambda x : \Box b.\ \mathbf{box}\ x$), it may still be adequate for encoding the untyped λ-calculus.

One important extension of this work is the case operator. Because there are limitations to what may be defined with `iter`, the SDP calculus also includes a construct for case analysis of closed terms.

However, we have not yet found an obvious correspondence for case in our encoding.

Another further area of investigation is into the dual operation to `iter`, the anamorphism over datatypes with embedded functions. An implementation of this operation, called `coiter`, is below. The `coiter` term is an anamorphism—it generates a recursive data structure from an initial seed.

```
data Dia f a = In (f (Dia f a), a)

coroll  :: Dia f a -> f (Dia f a)
coroll (In x) = fst x
coplace :: Dia f a -> a
coplace (In x) = snd x

coiter0 :: (a -> f a) -> a -> (exists a. Dia f a)
coiter0 g b =
  In (snd (xmap (coplace, coiter0 g) (g b)), b)
```

Instead of embedding the recursive type in a sum, we embed it in a product. The two selectors from this product have the dual types to `roll` and `place`. In the definition of `coiter0` we use `coplace` as the inverse where we would have used `cata` in the definition of `ana`. A term of type `(exists a.  Dia b a)` corresponds to the possibility type ($\diamond b$) in a modal calculus. However, while a general anamorphism is an inverse of a catamorphism, `coiter` is not an inverse to `iter`. In fact, `iter` cannot consume what `coiter` produces, giving doubts to its practical use. (On the other hand, `ana` itself has seen little practical use for datatypes with embedded functions.) From a logical point of view, this restriction makes sense. Combining anamorphisms and catamorphisms (even for datatypes without embedded functions) leads to general recursion.

7 Related work

The technique we present, using polymorphism to enforce parametricity, has appeared under various guises in the literature. For example, Shao et al. [27] use this technique (one level up) to implement type-level intensional analysis of recursive types. They use higher-order abstract syntax to the represent recursive types and remark that the kind of this type constructor requires a parametric function as its argument. However, they do not make a connection with modal type systems, nor do they extend their type-level iteration operator to higher kinds. Xi et al. [31] remark on the correspondence between HOAS terms with the place operator (which they call *HOASvar*) and closed terms of Mini-ML$_e^\Box$ but do not investigate the relationship or any form of iteration.

While higher-order abstract syntax has an attractive simplicity, the difficulties programming and reasoning about structures encoded with this technique have motivated research into language extensions for working with higher-order abstract syntax or alternative approaches altogether. Dale Miller developed a small language called ML_λ [17] that introduces a type constructor for terms formed by abstracting out a parameter. These types can be thought of as function types that can be intensionally analyzed through pattern matching. Pitts and Gabbay built on the theory of FM-sets to design a language called FreshML [23] that allows for the manipulation and abstraction of fresh "names". Nanevski [18] combines fresh names with modal necessity to allow for the construction of more efficient residual terms, while still retaining the ability to evaluate them at runtime. The Delphin Project [25] by Schürmann et al. develops a functional language for manipulating datatypes that are terms in the LF logical framework. Because higher-order abstract syntax is the primary representation technique in LF, Delphin provides operations for matching over higher-order LF terms in regular worlds. The SDP calculus uses modal necessity to restrict matching to closed worlds, so regular worlds provide additional flexibility without the difficulties of matching in an open world. The Hybrid [2] logical framework provides induction over higher-order abstract syntax by evaluation to de Bruijn terms, which provide straightforward induction.

There is a long history of encoding modality in logic, but only recently has the encoding of modal type systems been explored. Acar et al. [1] use modal types in a functional language that provides control over the use of memoization, and implement it as a library in SML. Because SML does not have modal types or first-class polymorphism, they use run-time checks to enforce the correct use of modality. Davies and Pfenning [7] presented, in passing, a simple encoding of the modal λ-calculus into the simply-typed λ-calculus that preserves only the dynamic semantics. Washburn expanded upon this encoding, showing that it bisimulates the source calculus [29].

8 Conclusion

While other approaches to defining an induction operator over higher-order abstract syntax require type system extensions to ensure the parametricity of embedded function spaces, the approach that we present in this paper requires only type polymorphism. Because of this encoding, we are able to implement iteration operators for datatypes with embedded parametric functions directly in the Haskell language.

However, despite its simplicity, our approach is equivalent to previous work on induction operators for HOAS. We demonstrate this generality by showing how the modal calculus of Schüermann, Despeyroux and Pfenning may be embedded into F_ω using this technique. In fact, the analogy of representing **box**ed terms with polymorphic terms makes semantic sense: a proposition with a boxed type is valid in all worlds and polymorphism makes that quantification explicit.

Acknowledgements

We thank Margaret DeLap, Eijiro Sumi, Stephen Tse, Stephan Zdancewic and the anonymous ICFP reviewers for providing us with feedback concerning this paper.

9 References

[1] U. Acar, G. Blelloch, and R. Harper. Selective memoization. In *Thirtieth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New Orleans, LA, Jan. 2002.

[2] S. Ambler, R. L. Crole, and A. Momigliano. Combining higer order abstract syntax with tactical theorem proving and (co)induction. In *Theorem Proving in Higher Order Logics, 15th International Conference, TPHOLs 2002, Hampton, VA, USA, August 20-23, 2002, Proceedings*, volume 2410 of *Lecture Notes in Computer Science*. Springer, 2002.

[3] C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[4] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[5] D. Clarke, R. Hinze, J. Jeuring, A. Löh, and J. de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[6] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, Dec. 1992.

[7] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[8] J. Despeyroux and P. Leleu. Recursion over objects of functional type. *Mathematical Structures in Computer Science*, 11:555–572, 2001.

[9] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Twenty-Third ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, FL,USA, 1996.

[10] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[11] R. Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002. MPC Special Issue.

[12] P. Johann. A generalization of short-cut fution and its correctness proof. *Higher-Order and Symbolic Computation*, 15:273–300, 2002.

[13] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), Jan. 1995.

[14] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany*, number 1782 in LNCS. Springer-Verlag, 2000.

[15] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, 1991.

[16] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, La Jolla, CA, June 1995.

[17] D. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, May 1990.

[18] A. Nanevski. Meta-programming with names and necessity. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 206–217. ACM Press, 2002.

[19] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[20] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, Aug. 2001.

[21] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *1988 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, Atlanta, GA, USA, June 1988.

[22] F. Pfenning and C. Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999.

[23] A. M. Pitts and M. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction*, pages 230–255, 2000.

[24] C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1–2):1–58, Sept. 2001.

[25] C. Schürmann, R. Fontana, and Y. Liao. Delphin: Functional programming with deductive systems. Available at `http://cs-www.cs.yale.edu/homes/carsten/`, 2002.

[26] E. Sumii and N. Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[27] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, Montreal, Sept. 2000.

[28] P. Wadler. Theorems for free! In *FPCA89: Conference on Functional Programming Languages and Computer Architecture*, London, Sept. 1989.

[29] G. Washburn. Modal typing for specifying run-time code generation. Available from `http://www.cis.upenn.edu/~geoffw/research/`, 2001.

[30] S. Weirich. Higher-order intensional type analysis in type-erasure semantics. Available from `http://www.cis.upenn.edu/~sweirich/`, 2003.

[31] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Thirtieth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, LA, USA, Jan. 2003.

A Generic Haskell implementation of xmap

```
type XMap {[*]} t1 t2 = (t1 -> t2, t2 -> t1)
type XMap {[k -> l]} t1 t2 = forall u1 u2.
  XMap {[k]} u1 u2 -> XMap {[l]}(t1 u1)(t2 u2)

xmap {| t :: k |} :: XMap {[k]} t t
xmap {| Unit |}  = (id,id)
xmap {| :+: |}  (xmapA1,xmapA2) (xmapB1,xmapB2) =
      (\x -> case x of
       (Inl a) -> Inl (xmapA1 a)
       (Inr b) -> Inr (xmapB1 b),
       \x -> case x of
       (Inl a) -> Inl (xmapA2 a)
       (Inr b) -> Inr (xmapB2 b))
xmap {| :*: |}  (xmapA1,xmapA2) (xmapB1,xmapB2) =
      (\(a :*: b) -> (xmapA1 a) :*: (xmapB1 b),
       \(a :*: b) -> (xmapA2 a) :*: (xmapB2 b))
xmap {| (->) |}  (xmapA1,xmapA2) (xmapB1,xmapB2) =
      (\f -> xmapB1 . f . xmapA2,
       \f -> xmapB2 . f . xmapA1)
xmap {| Int  |}  = (id, id)
```

```
xmap {| Bool |}  = (id, id)
xmap {| IO |} (xmapA1,xmapA2) =
     (fmap xmapA1, fmap xmapA2)
xmap {| [] |} (xmapA1,xmapA2) =
     (map xmapA1, map xmapA2)
```

B Full encoding of SDP

B.1 Parameterization

$$\frac{}{\tau\langle b\rangle \triangleq \tau} \qquad \frac{}{\tau\langle 1\rangle \triangleq 1} \qquad \frac{\tau\langle B_1\rangle \triangleq \tau_1 \quad \tau\langle B_2\rangle \triangleq \tau_2}{\tau\langle B_1 \to B_2\rangle \triangleq \tau_1 \to \tau_2}$$

$$\frac{\tau\langle B_1\rangle \triangleq \tau_1 \quad \tau\langle B_2\rangle \triangleq \tau_2}{\tau\langle B_1 \times B_2\rangle \triangleq \tau_1 \times \tau_2}$$

B.2 Signatures

$$\frac{\forall c_i \in \mathrm{dom}(\Sigma) \quad \Sigma(c_i) = B_i \to b}{\tau\langle\Sigma\rangle \triangleq \langle \mathcal{L}(c_1) : \tau\langle B_1\rangle, \ldots, \mathcal{L}(c_n) : \tau\langle B_n\rangle\rangle}$$

B.3 Types

$$\frac{\Delta \vdash \tau}{\Delta \vdash b \rhd_\tau \mathsf{Rec}\;\Sigma^*\;\tau} \qquad \frac{\alpha \notin \Delta \quad \Delta \uplus \{\alpha : \star \to \star\} \vdash A \rhd_{\alpha\tau} \tau'}{\Delta \vdash \Box A \rhd_\tau \forall \alpha{:}\star \to \star.\tau'}$$

$$\frac{}{\Delta \vdash 1 \rhd_\tau 1(\tau)} \qquad \frac{\Delta \vdash A_1 \rhd_\tau \tau_1 \quad \Delta \vdash A_2 \rhd_\tau \tau_2}{\Delta \vdash A_1 \to A_2 \rhd_\tau \tau_1 \to \tau_2}$$

$$\frac{\Delta \vdash A_1 \rhd_\tau \tau_1 \quad \Delta \vdash A_2 \rhd_\tau \tau_2}{\Delta \vdash A_1 \times A_2 \rhd_\tau \tau_1 \times \tau_2}$$

B.4 Terms

$$\frac{x \notin \Xi}{\Delta;\Xi \vdash x \rhd_\tau x} \qquad \frac{x \in \Xi}{\Delta;\Xi \vdash x \rhd_\tau x[\lambda\alpha : \star.\tau]}$$

$$\frac{\alpha \notin \Delta \quad \Delta \uplus \{\alpha : \star \to \star\};\Xi \vdash M \rhd_{\alpha\tau} e}{\Delta;\Xi \vdash \mathbf{box}\,M \rhd_\tau \Lambda\alpha{:}\star \to \star.e} \qquad \frac{}{\Delta;\Xi \vdash \langle\rangle \rhd_\tau \langle\rangle[\tau]}$$

$$\frac{\Sigma(c) = B \to b \quad \Delta \vdash B \rhd_\tau \tau_B}{\Delta;\Xi \vdash c \rhd_\tau \lambda x : \tau_B.\mathbf{roll}[\tau](\mathbf{inj}_{\mathcal{L}(c)}\,x\,\mathbf{of}\,(\mathsf{Rec}\;\Sigma^*\;\tau)\langle\Sigma\rangle)}$$

$$\frac{\Delta;\Xi \vdash M \rhd_\tau e \quad \Delta \vdash A_1 \rhd_\tau \tau_1}{\Delta;\Xi \vdash \lambda x : A_1.M \rhd_\tau \lambda x : \tau_1.e}$$

$$\frac{\Delta;\Xi \vdash M_1 \rhd_\tau e_1 \quad \Delta;\Xi \vdash M_2 \rhd_\tau e_2}{\Delta;\Xi \vdash M_1 M_2 \rhd_\tau e_1 e_2}$$

$$\frac{\begin{gathered}\Delta \vdash \Box A_1 \rhd_\tau \tau_1 \\ \Delta;\Xi \vdash M_1 \rhd_\tau e_1 \quad \Delta;\Xi \uplus \{x\} \vdash M_2 \rhd_\tau e_2\end{gathered}}{\Delta;\Xi \vdash \mathbf{let\ box}\,x : A_1 = M_1\,\mathbf{in}\,M_2 \rhd_\tau (\lambda x : \forall\alpha.\tau_1.e_2)e_1}$$

$$\frac{\Delta;\Xi \vdash M_1 \rhd_\tau e_1 \quad \Delta;\Xi \vdash M_2 \rhd_\tau e_2}{\Delta;\Xi \vdash \langle M_1, M_2\rangle \rhd_\tau \langle e_1, e_2\rangle} \qquad \frac{\Delta;\Xi \vdash M \rhd_\tau e}{\Delta;\Xi \vdash \mathbf{fst}\,M \rhd_\tau \mathbf{fst}\,e}$$

$$\frac{\Delta;\Xi \vdash M \rhd_\tau e}{\Delta;\Xi \vdash \mathbf{snd}\,M \rhd_\tau \mathbf{snd}\,e}$$

$$\frac{\Delta \vdash A \rhd_\tau \tau_A \quad \Delta;\Xi \vdash \Theta \rhd_\tau^{\tau_A} e_\Theta \quad \Delta;\Xi \vdash M \rhd_\tau e_M}{\Delta;\Xi \vdash \mathbf{iter}\,[\Box B, A][\Theta]\;M \rhd_\tau \mathbf{iter}\{\!|B^*|\!\}[\tau][\tau_A]\;e_\Theta\;e_M}$$

B.5 Replacements

$$\frac{\forall c_i \in \mathrm{dom}(\Theta) \quad \Delta;\Xi \vdash \Theta(c_i) \rhd_\tau e_i}{\begin{aligned}\Delta;\Xi \vdash \Theta \rhd_\tau^{\tau_A} \lambda x : \Sigma^*\tau_A.\mathbf{case}\,x\,\mathbf{of}\,\mathbf{inj}_{\mathcal{L}(c_1)}\,y_1\,\mathbf{in}\,(e_1 y_1)& \\ \ldots& \\ \mathbf{inj}_{\mathcal{L}(c_n)}\,y_n\,\mathbf{in}\,(e_n y_n)&\end{aligned}}$$