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

GEOFFREY WASHBURN

STEPHANIE WEIRICH

Department of Computer and Information Science

University of Pennsylvania, Philadelphia, Pennsylvania 19104 United States
(*e-mail:* {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 $\mathbb{F}_\omega$. This encoding can serve as a starting point for reasoning about higher-order structures in polymorphic languages.

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,

∗ This is an extended version of the paper that appeared in The 8th ACM SIGPLAN International Conference on Functional Programming (Washburn & Weirich, 2003).

[1] While the name comes from Pfenning and Elliott (1988), the idea itself goes back to Church. (1940).

capture-avoiding substitution is merely function application in the metalanguage. However, outside of a few specialized domains, such as theorem proving, partial evaluation (Sumii & Kobayashi, 2001), logical frameworks (Pfenning & Schürmann, 1999) and intensional type analysis (Trifonov *et al.*, 2000; Weirich, 2006), 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 (Meijer & Hutton, 1995). 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 (Fegaras & Sheard, 1996; Schürmann *et al.*, 2001). 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 (Schürmann *et al.*, 2001). In this paper, we use the term *iterator* to refer to a catamorphism restricted to arguments with parametric functions.

A type system can separate parametric functions from those that are not. For example, Fegaras and Sheard (1996) add tags to mark the types of datatypes whose embedded functions are not parametric, prohibiting iteration over those datatypes. Alternatively, Schürmann, Despeyroux and Pfenning (2001) and Despeyroux and LeLeu (2001) 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 (Peyton Jones *et al.*, 2005). 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 provide a number of examples including Danvy and Filinski's optimizing one-pass CPS conversion algorithm (Danvy & Filinski, 1992).

[2] Catamorphisms (also called folds) are sometimes represented with the bananas $(\!| \cdot |\!)$ notation (Meijer *et al.*, 1991).

- Since we encode our 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 our technique, we use this implementation to show a formal translation from the Schürmann, Despeyroux and Pfenning modal calculus (2001) (called here the SDP calculus) to System $\mathbb{F}_\omega$. 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 our 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 (1995) and Fegaras and Sheard (1996). In Section 2.2 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 2.3). In Section 3, we describe an implementation of that interface within the part of Haskell that corresponds to System $\mathbb{F}_\omega$, 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 $\mathbb{F}_\omega$, using the implementation that we developed in Section 3. Section 8 presents future work, Section 9 presents related work, and Section 10 concludes.

2 Catamorphisms for datatypes with embedded functions

2.1 Examples of catamorphisms

The following recursive data type represents the untyped λ-calculus using 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 (λx.x) as `Lam (\x -> x)` or the diverging expression (λx.(xx))(λx.(xx)) as `App (Lam (\x -> App x x)) (Lam (\x -> App x x))`.

[3] All of the following examples are in the syntax of the Haskell language (Peyton Jones, 2003). 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.

Using this data type, we can implement an interpreter for the λ-calculus. To do so, we must also provide a representation of values (also using HOAS).

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

unFn (Fn x) = x
```

It is tricky to define recursive operations, such as call-by-value 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 a right 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`, in the evaluation of `Lam x`.

```
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 the encoding 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, `foldr` is a catamorphism for the list datatype and can implement many list operations. For lists of type `[a]`, `foldr` replaces `[]` with a base case of type `b` and `(:)` with a function of type `a -> b -> b`.

Meijer and Hutton (1995) 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.

```
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))
```

Fig. 1. Meijer/Hutton catamorphism

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 (Clarke *et al.*, 2001), we can define `xmap` and automatically generate `xmapExpF` from `ExpF` (see Figure 2).[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 $\mathbb{F}_\omega$ for which `xmap` is not defined are those whose bodies contain first-class polymorphism. For example, $\lambda\alpha : \star.\forall\beta : \star.\alpha \to \beta$.

[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.

```
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)
```

Fig. 2. Generic Haskell implementation of **xmap**

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 (1996) noted that the catamorphism often undoes with `f` what it has just done with `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`.

```
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
```

Fig. 3. Fegaras/Sheard catamorphism

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 3, we extend it with an extra constructor (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`.

Counting the number of variables in (λx. x x) will evaluate as follows:

```
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` may sometimes 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.

We can approximate when `cata` will be meaningful by separating expressions into two classes. *Sound* expressions do not use `Place` and furthermore, do not contain any non-parametric function spaces (see below). All other expressions are *unsound*.

There are two ways for function-space 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)
```

The distinction between sound and unsound terms is important for representing the untyped λ-calculus because with higher-order abstract syntax, because some unsound terms may not correspond to any untyped λ-calculus expressions. However, all λ-calculus expressions may be encoded by sound terms.[5]

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.

[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.

2.2 Enforcing term parametricity with type parametricity

The terms `badplace`, `badcata`, and `badcase` are *exotic terms*, values in the meta-language that do not correspond to any members of the object language. In other words, our encoding of untyped λ-calculus expressions as Haskell terms of type `Exp t`, for some `t`, is not *adequate*—there are some terms in this type that are not equivalent to the encoding of any object language term. If our encoding were adequate, there would be no exotic terms. The distinction between sound and unsound expressions approximates the distinction between non-exotic and exotic terms—sound expressions are guaranteed to be non-exotic (i.e. correspond to some object language term).

The reason that our encoding is not adequate can be seen from the type of `badplace` and `badcata`. The type parameter of `Exp` is constrained to be `Int`, so we can only use `cata` on this expression to produce an `Int`. Whenever we use `cata` or `Place` inside an expression, this parameter will be constrained. Therefore, to eliminate unsound terms containing `cata` or `Place` we can use *higher-rank polymorphism* to ensure that the type parameter of `Exp` is always abstract. Instead of considering terms of type `Exp t`, for some `t`, we will only consider terms of type `forall a. Exp a`.

We can then define a version of `cata`, called `iter0` that may only be applied to expressions of type `forall a. Exp a`, 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
```

Consequently, we will not be able to use `badplace` and `badcata` with `iter0`.

However, our encoding of untyped λ-terms as expressions of type `forall a. Exp a` is not yet quite adequate. It does not prevent terms like `badcase`. 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 2.3.

There are two ways to prohibit case analysis. The first way is to re-implement `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 4, then the only way to destruct an expression of type `Exp t` 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
```

Fig. 4. 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 (Jones, 1995). In particular, we cannot represent the type constructor $(\lambda\alpha : \star.\alpha \to \alpha)$ 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 (Jones, 2000), 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{| \a.a -> a |}`. It is not difficult to add instances for other type constructors, such as lists and tuples.

2.3 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. A Haskell 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.

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

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 }
```

[6] This example is from Schürmann et. al (2001).

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

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 (1992). 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 }
```

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

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 $\mathbb{F}_\omega$

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 is not the only way to prevent case analysis. Furthermore, term and type recursion is not necessary to implement this data type. We may define `iter` and `Rec` in the fragment of Haskell that corresponds to $\mathbb{F}_\omega$ (Girard, 1971) so that iteration is the only elimination form for `Rec`. This implementation appears in Figure 5.

The encoding is similar to the encoding of covariant data types in the polymorphic λ-calculus (Böhm & Berarducci, 1985) 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.

```
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
```

Fig. 5. Catamorphism in the $\mathbb{F}_\omega$ fragment of Haskell

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 5 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 $\mathbb{F}_\omega$ 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 $\mathbb{F}_\omega$, 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 $\mathbb{F}_\omega$ (Reynolds, 1983). 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 $\mathbb{F}_\omega$, we may be able to prove some form of them for Haskell using techniques developed by Johann (2002).

Using parametricity, we can derive what are know as *free theorem* (Wadler, 1989) about expressions. They are "free" in the sense that we do not actually need to know anything about the implementation of the expression, they just are available merely as consequence of the expression type-checking. For example, any expression `x` of type `forall a. (b a -> a) -> a`, we know that the following free theorem holds

$$(\texttt{f . f' = id} \wedge \texttt{f . g = h . fst (xmap\{|b|\}(f,f'))}) \Rightarrow \texttt{f (x g) = x h}.$$

In particular, we know it holds for expressions of type `forall a. Exp a` by substituting `ExpF` for `b`. This is not entirely obvious because `Exp` is defined in terms of other constructors: `forall a. Exp a` is the same as `forall a. (Rec ExpF a)`, which in turn is equal to `forall a. (ExpF a -> a) -> a`.

The equivalence `=` in this theorem is equivalence in some parametric model of $\mathbb{F}_\omega$, 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.

Theorem 3.1
`iter0 roll = id`.

Proof
By the definitions of `iter0` and `cata` we know that for `iter0 roll = id` to be true, it must be the case that `x roll = x`, for all `x`. If we instantiate the free theorem such that `f` is `iter0 h`, `f'` is `place`, and `g` is `roll` we have that

$$(\texttt{iter0 h . place = id} \wedge \texttt{iter0 h . roll = h . fst (xmap\{|ExpF|\}(iter0 h, place))}) \Rightarrow \texttt{iter0 h (x roll) = x h}.$$

Then by unfolding the definitions of `iter0` and `place`

```
iter0 h . place  =  (\x -> x h) . (\y -> \f -> y)
                 =  \x -> (\y -> \f . y) x h
                 =  \x -> (\f -> x) h
                 =  \x -> x
```

we can see that we have the first equality required by the implication. Next we need to prove that it is the case that

$$\texttt{iter0 h . roll = h . fst (xmap\{|ExpF|\}(iter0 h, place))}.$$

By extensionality we can conclude this equality will only be true if for any `x`,

```
iter0 h (roll x) = h (fst (xmap{|ExpF|}(iter0 h, place))) x.
```

Here we can note that on the right-hand side of the equation,

```
h (fst (xmap{|ExpF|}(iter0 h, place))) x,
```

is just the expansion of `roll x h`. Dually, the left-hand side of the equation, `iter0 h (roll x)`, reduces to `roll x h`. Therefore, we have our second premise.

Our free theorem then gives us `iter0 h (x roll) = x h`, which is the same as `x roll h = x h` by reducing `iter0`. Finally, we can conclude `x roll = x` by extensionality. □

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`.

Theorem 3.2 (*Uniqueness*)

$$\exists \texttt{f'.f . f' = id} \wedge \texttt{f . roll = h . fst (xmap\{|ExpF|\}(f,f'))} \Leftrightarrow \texttt{f = iter0 h.}$$

Proof
The reverse direction of the biconditional follows directly from the definitions of `iter0` and `roll`. The forward direction of the biconditional follows from the free theorem.

For the forward direction we need to show that

$$(\texttt{f . f'} \wedge \texttt{f . roll = h . fst (xmap\{|ExpF|\}(f,f'))}) \Rightarrow \texttt{f = iter0 h.}$$

Therefore, we assume `f . f'` and `f . roll = h . fst (xmap{|ExpF|}(f,f'))`. By our free theorem we know that

$$(\texttt{f . f'} \wedge \texttt{f . roll = h . fst (xmap\{|ExpF|\}(f,f'))}) \Rightarrow \texttt{f (x roll) = x h.}$$

So by our assumptions we know that `f (x roll) = x h`. From our previous result, we know that `iter0 roll = id`, which is the same as `x roll = x` by extensionality and reduction. Given `x roll = x`, it is the case that `f x = x h` holds. By definition `iter0 h x = x h` which implies `f x = iter0 h x` which in turn implies `f = iter0 h` by extensionality.

For the other direction, assume that `f = iter 0 h`. Next we need to show there exists a `f'` such that the equalities `f . f' = id` and `f . roll = h . fst (xmap{|ExpF|}(f, f'))` are true. We guess that `place` would be a good choice for `f'`, because we know `iter0 h . place = id` from Theorem 3.1. Therefore choosing `place` for `f'` provides us with the first equality. Next we want to prove that `f . roll = h . fst(xmap{|ExpF|}(f, place))`. This equality holds if `iter0 h . roll = h . fst(xmap{|ExpF|}(iter 0 h, place))` does. However, we have already shown that both sides of this equation are equal to `roll x h` as part of Theorem 3.1, so the proof is complete. □

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.

Theorem 3.3 (*Fusion*)

$$\texttt{(f . f' = id} \wedge \texttt{f . g = h . fst(xmap\{|ExpF|\}(f,f')))} \Rightarrow \texttt{f . iter0 g = iter0 h.}$$

Proof
By using the free theorem, we know

$$\texttt{(f . f'} \wedge \texttt{f . roll = h . fst (xmap\{|ExpF|\}(f,f')))} \Rightarrow \texttt{f (x roll) = x h.}$$

Then begin by assuming `f . f' = id` and `f . g = h . fst(xmap{|ExpF|}(f,f'))` to obtain `f (x g) = x h`. From `f (x g) = x h` we want to prove `f . iter0 g = iter0 h`. This holds if `f (x g) = x h` does, which we know is true from the definition of `iter0`. □

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) (2001). We do so by using this library to give a sound and complete embedding of the SDP calculus into $\mathbb{F}_\omega$. 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 6.

The static semantics of the SDP calculus in shown in Figure 8. The SDP type system is defined in terms of a judgment $\Omega; \Upsilon \stackrel{\text{SDP}}{\vDash} M : A$ which is read as "term M has type A with respect to the valid environment Ω and the local environment Υ". A separate typing judgment $\Omega; \Upsilon \stackrel{\text{SDP}}{\vDash} \Theta : A\langle\Sigma\rangle$ is used to mean "the replacement Θ maps constants in Σ to their well-typed terms iterated with type A with respect to the valid environment Ω and the local environment Υ". Iteration types are defined in Figure 7, and will be explained in more detail shortly. In addition to well-formed terms, the SDP static semantics also defines what it means for a term to be *atomic* or in *canonical form*. This is indicated with the judgments $\Psi \stackrel{\text{SDP}}{\vDash} M \downarrow B$ and $\Psi \stackrel{\text{SDP}}{\vDash} M \Uparrow B$ respectively. Canonical forms in SDP are β-normal, η-long terms.

The dynamic semantics of the SDP calculus in described in Figure 9 and 10. The dynamic semantics consist of two judgments and a rewrite system: The first judgment is $\Psi \stackrel{\text{SDP}}{\vDash} M \hookrightarrow V : A$, read as "term M evaluates to value V with type A", and the second judgment is $\Psi \stackrel{\text{SDP}}{\vDash} M \Uparrow V : B$, which is read as "term M is canonicalized to value V with type B". Replacements are evaluated using a rewrite system called *elimination*, written as $\langle A, \Psi, \Theta\rangle(V)$ and read as "value V is eliminated with respect to the pure context Ψ and the type A". Elimination always rewrites to another value, and is analogous to the application of a catamorphism.

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

Fig. 6. Syntax of SDP calculus

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's (2001) 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:

$$\frac{\Omega; \varnothing \vDash^{\text{SDP}} M : A}{\Omega; \Upsilon \vDash^{\text{SDP}} \mathbf{box}\, M : \Box A}\ \mathsf{tp\_box}$$

$$\frac{\Omega; \Upsilon \vDash^{\text{SDP}} M_1 : \Box A_1 \quad \Omega \uplus \{x : A_1\}; \Upsilon \vDash^{\text{SDP}} M_2 : A_2}{\Omega; \Upsilon \vDash^{\text{SDP}} \mathbf{let\ box}\ x : A_1 = M_1\ \mathbf{in}\ M_2 : A_2}\ \mathsf{tp\_letb}$$

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.

$$\frac{\Psi \vDash^{\text{SDP}} M_1 \hookrightarrow \mathbf{box}\, M_1' : \Box A_1 \quad \Psi \vDash^{\text{SDP}} M_2\{M_1'/x\} \hookrightarrow V : A_2}{\Psi \vDash^{\text{SDP}} \mathbf{let\ box}\ x : A_1 = M_1\ \mathbf{in}\ M_2 \hookrightarrow V : A_2}\ \mathsf{ev\_letb}$$

$$\begin{array}{rcl} \mathsf{A}\langle \mathsf{b} \rangle & \triangleq & \mathsf{A} \\ \mathsf{A}\langle \mathsf{1} \rangle & \triangleq & \mathsf{1} \\ \mathsf{A}\langle \mathsf{B}_1 \to \mathsf{B}_2 \rangle & \triangleq & \mathsf{A}\langle \mathsf{B}_1 \rangle \to \mathsf{A}\langle \mathsf{B}_2 \rangle \\ \mathsf{A}\langle \mathsf{B}_1 \times \mathsf{B}_2 \rangle & \triangleq & \mathsf{A}\langle \mathsf{B}_1 \rangle \times \mathsf{A}\langle \mathsf{B}_2 \rangle \end{array}$$

Fig. 7. Iteration types

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 $\Box\mathsf{B}$, 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, $\lambda \mathsf{x} : \mathsf{B}_1.\mathsf{M} : \mathsf{B}_1 \to \mathsf{B}_2$, 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.

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 $\mathsf{B} \to \mathsf{b}$. Because B is a pure type, these constructors may only take *parametric* functions as arguments. In the original presentation of SDP multiple base types were allowed, however this required keeping track of the *subordination* relationship which characterizes the dependencies between constants in the signature. We felt that the added complication did not add anything to our results.

Consider a signature describing the untyped λ-calculus, $\Sigma = \{\mathbf{app} : \mathsf{b} \times \mathsf{b} \to \mathsf{b}, \mathbf{lam} : (\mathsf{b} \to \mathsf{b}) \to \mathsf{b}\}$, 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]

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

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

The argument to iteration, M, must have a pure closed type to be analyzable. Analysis proceeds via walking over M and using the *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 $\mathsf{A}\langle \mathsf{B} \rangle$, defined in Figure 7, 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 $\Omega; \Upsilon \vDash^{\text{SDP}} \Theta : \mathsf{A}\langle \Sigma \rangle$, which requires that if $\Theta(\mathsf{c}) = \mathsf{M}_\mathsf{c}$ and $\Sigma(\mathsf{c}) = \mathsf{B}_\mathsf{c}$, then M_c must have type $\mathsf{A}\langle \mathsf{B}_\mathsf{c} \rangle$.

[9] For simplicity, our formal presentation of SDP (in Figure 6) does not include integers. However, it is straightforward to extend the language with integer types distinct from pure types.

$\boxed{\Psi \vdash^{\text{SDP}} M \downarrow B}$ *Atomic terms*

$$\frac{\Psi(x) = B}{\Psi \vdash^{\text{SDP}} x \downarrow B}\ \mathsf{at\_var} \qquad \frac{\Sigma(c) = B \to b}{\Psi \vdash^{\text{SDP}} c \downarrow B \to b}\ \mathsf{at\_cons}$$

$$\frac{\Psi \vdash^{\text{SDP}} V_1 \downarrow B_2 \to B_1 \quad \Psi \vdash^{\text{SDP}} V_2 \Uparrow B_2}{\Psi \vdash^{\text{SDP}} V_1 V_2 \downarrow B_1}\ \mathsf{at\_app} \qquad \frac{\Psi \vdash^{\text{SDP}} V \downarrow B_1 \times B_2}{\Psi \vdash^{\text{SDP}} \mathbf{fst}\, V \downarrow B_1}\ \mathsf{at\_fst}$$

$$\frac{\Psi \vdash^{\text{SDP}} V \downarrow B_1 \times B_2}{\Psi \vdash^{\text{SDP}} \mathbf{snd}\, V \downarrow B_2}\ \mathsf{at\_snd}$$

$\boxed{\Psi \vdash^{\text{SDP}} M \Uparrow B}$ *Canonical terms*

$$\frac{\Psi \vdash^{\text{SDP}} V \downarrow b}{\Psi \vdash^{\text{SDP}} V \Uparrow b}\ \mathsf{can\_at} \qquad \frac{\Psi \uplus \{x : B_1\} \vdash^{\text{SDP}} V \Uparrow B_2}{\Psi \vdash^{\text{SDP}} \lambda x : B_1.V \Uparrow B_1 \to B_2}\ \mathsf{can\_lam}$$

$$\frac{\Psi \vdash^{\text{SDP}} V_1 \Uparrow B_1 \quad \Psi \vdash^{\text{SDP}} V_2 \Uparrow B_2}{\Psi \vdash^{\text{SDP}} \langle V_1, V_2 \rangle \Uparrow B_1 \times B_2}\ \mathsf{can\_prod}$$

$\boxed{\Omega; \Upsilon \vdash^{\text{SDP}} \Theta : A\langle\Sigma\rangle}$ *Replacement typing rules*

$$\frac{\forall c_i \in \mathrm{dom}(\Sigma) \quad \mathrm{dom}(\Theta) = \mathrm{dom}(\Sigma) \quad \Sigma(c_i) = B_i \quad \Omega; \Upsilon \vdash^{\text{SDP}} \Theta(c_i) : A\langle B_i \rangle}{\Omega; \Upsilon \vdash^{\text{SDP}} \Theta : A\langle\Sigma\rangle}\ \mathsf{tp\_rep}$$

$\boxed{\Omega; \Upsilon \vdash^{\text{SDP}} M : A}$ *Term typing rules*

$$\frac{x \notin \mathrm{dom}(\Omega) \quad \Upsilon(x) = A}{\Omega; \Upsilon \vdash^{\text{SDP}} x : A}\ \mathsf{tp\_var} \qquad \frac{x \notin \mathrm{dom}(\Upsilon) \quad \Omega(x) = A}{\Omega; \Upsilon \vdash^{\text{SDP}} x : A}\ \mathsf{tp\_bvar}$$

$$\frac{}{\Omega; \Upsilon \vdash^{\text{SDP}} \langle\rangle : 1}\ \mathsf{tp\_unit} \qquad \frac{\Sigma(c) = B \to b}{\Omega; \Upsilon \vdash^{\text{SDP}} c : B \to b}\ \mathsf{tp\_con}$$

$$\frac{\Omega; \Upsilon \uplus \{x : A_1\} \vdash^{\text{SDP}} M : A_2}{\Omega; \Upsilon \vdash^{\text{SDP}} \lambda x : A_1.M : A_1 \to A_2}\ \mathsf{tp\_abs}$$

$$\frac{\Omega; \Upsilon \vdash^{\text{SDP}} M_1 : A_1 \to A_2 \quad \Omega; \Upsilon \vdash^{\text{SDP}} M_2 : A_1}{\Omega; \Upsilon \vdash^{\text{SDP}} M_1 M_1 : A_2}\ \mathsf{tp\_app}$$

$$\frac{\Omega; \Upsilon \vdash^{\text{SDP}} M_1 : \Box A_1 \quad \Omega \uplus \{x : A_1\}; \Upsilon \vdash^{\text{SDP}} M_2 : A_2}{\Omega; \Upsilon \vdash^{\text{SDP}} \mathbf{let\ box}\ x : A_1 = M_1\ \mathbf{in}\, M_2 : A_2}\ \mathsf{tp\_letb} \qquad \frac{\Omega; \varnothing \vdash^{\text{SDP}} M : A}{\Omega; \Upsilon \vdash^{\text{SDP}} \mathbf{box}\, M : \Box A}\ \mathsf{tp\_box}$$

$$\frac{\Omega; \Upsilon \vdash^{\text{SDP}} M_1 : A_1 \quad \Omega; \Upsilon \vdash^{\text{SDP}} M_2 : A_2}{\Omega; \Upsilon \vdash^{\text{SDP}} \langle M_1, M_2 \rangle : A_1 \times A_2}\ \mathsf{tp\_pair} \qquad \frac{\Omega; \Upsilon \vdash^{\text{SDP}} M : A_1 \times A_2}{\Omega; \Upsilon \vdash^{\text{SDP}} \mathbf{fst}\, M : A_1}\ \mathsf{tp\_fst}$$

$$\frac{\Omega; \Upsilon \vdash^{\text{SDP}} M : A_1 \times A_2}{\Omega; \Upsilon \vdash^{\text{SDP}} \mathbf{snd}\, M : A_2}\ \mathsf{tp\_snd} \qquad \frac{\Omega; \Upsilon \vdash^{\text{SDP}} M : \Box B \quad \Omega; \Upsilon \vdash^{\text{SDP}} \Theta : A\langle\Sigma\rangle}{\Omega; \Upsilon \vdash^{\text{SDP}} \mathbf{iter}\, [\Box B, A][\Theta]\ M : A\langle B\rangle}\ \mathsf{tp\_iter}$$

Fig. 8. SDP static semantics

$\boxed{\Psi \vDash^{\text{SDP}} M \Uparrow V : B}$ *Canonicalization*

$$\frac{\Psi \vDash^{\text{SDP}} M \hookrightarrow V : b}{\Psi \vDash^{\text{SDP}} M \Uparrow V : b}\ \text{ec\_at} \qquad \frac{\Psi \uplus \{x : B_1\} \vDash^{\text{SDP}} Mx \Uparrow V : B_2}{\Psi \vDash^{\text{SDP}} M \Uparrow \lambda x : B_1.V : B_1 \to B_2}\ \text{ec\_arr}$$

$$\frac{\Psi \vDash^{\text{SDP}} \mathbf{fst}\, M \Uparrow V_1 : B_1 \quad \Psi \vDash^{\text{SDP}} \mathbf{snd}\, M \Uparrow V_2 : B_2}{\Psi \vDash^{\text{SDP}} M \Uparrow \langle V_1, V_2\rangle : B_1 \times B_2}\ \text{ec\_pair} \qquad \frac{}{\Psi \vDash^{\text{SDP}} M \Uparrow \langle\rangle : 1}\ \text{ec\_unit}$$

$\boxed{\Psi \vDash^{\text{SDP}} M \hookrightarrow V : A}$ *Evaluation*

$$\frac{}{\Psi \vDash^{\text{SDP}} x \hookrightarrow x : B}\ \text{ev\_var} \qquad \frac{}{\Psi \vDash^{\text{SDP}} c \hookrightarrow c : B \to b}\ \text{ev\_const} \qquad \frac{}{\Psi \vDash^{\text{SDP}} \langle\rangle \hookrightarrow \langle\rangle : 1}\ \text{ev\_unit}$$

$$\frac{\varnothing; \Psi \uplus \{x : A_2\} \vDash^{\text{SDP}} M : A_2}{\Psi \vDash^{\text{SDP}} \lambda x : A_1.M \hookrightarrow \lambda x : A_1.M : A_1 \to A_2}\ \text{ev\_lam}$$

$$\frac{\begin{array}{c}\Psi \vDash^{\text{SDP}} M_1 \hookrightarrow \lambda x : A_2.M_1' : A_2 \to A_1 \\ \Psi \vDash^{\text{SDP}} M_2 \hookrightarrow V_2 : A_2 \\ \Psi \vDash^{\text{SDP}} M_1'\{V_2/x\} \hookrightarrow V : A_1\end{array}}{\Psi \vDash^{\text{SDP}} M_1 M_2 \hookrightarrow V : A_1}\ \text{ev\_app} \qquad \frac{\begin{array}{c}\Psi \vDash^{\text{SDP}} M_1 \hookrightarrow V_1 : B_2 \to B_1 \\ \Psi \vDash^{\text{SDP}} V_1 \downarrow B_2 \to B_1 \\ \Psi \vDash^{\text{SDP}} M_2 \Uparrow V_2 : B_2\end{array}}{\Psi \vDash^{\text{SDP}} M_1 M_2 \hookrightarrow V_1 V_2 : B_1}\ \text{ev\_at}$$

$$\frac{\varnothing; \Psi \vDash^{\text{SDP}} M_1 : B_1 \quad \varnothing; \Psi \vDash^{\text{SDP}} M_2 : B_2}{\Psi \vDash^{\text{SDP}} \langle M_1, M_2\rangle \hookrightarrow \langle M_1, M_2\rangle : B_1 \times B_2}\ \text{ev\_pair}$$

$$\frac{\Psi \vDash^{\text{SDP}} M \hookrightarrow \langle M_1, M_2\rangle : A_1 \times A_2 \quad \Psi \vDash^{\text{SDP}} M_1 \hookrightarrow V : A_1}{\Psi \vDash^{\text{SDP}} \mathbf{fst}\, M \hookrightarrow V : A_1}\ \text{ev\_fst}$$

$$\frac{\Psi \vDash^{\text{SDP}} M \Uparrow \langle V_1, V_2\rangle : B_1 \times B_2}{\Psi \vDash^{\text{SDP}} \mathbf{fst}\, M \hookrightarrow V_1 : B_1}\ \text{ev\_fst\_at}$$

$$\frac{\Psi \vDash^{\text{SDP}} M \hookrightarrow \langle M_1, M_2\rangle : A_1 \times A_2 \quad \Psi \vDash^{\text{SDP}} M_2 \hookrightarrow V : A_2}{\Psi \vDash^{\text{SDP}} \mathbf{snd}\, M \hookrightarrow V : A_2}\ \text{ev\_snd}$$

$$\frac{\Psi \vDash^{\text{SDP}} M \Uparrow \langle V_1, V_2\rangle : B_1 \times B_2}{\Psi \vDash^{\text{SDP}} \mathbf{snd}\, M \hookrightarrow V_2 : B_2}\ \text{ev\_snd\_at} \qquad \frac{\varnothing; \varnothing \vDash^{\text{SDP}} M : A}{\Psi \vDash^{\text{SDP}} \mathbf{box}\, M \hookrightarrow \mathbf{box}\, M : \Box A}\ \text{ev\_box}$$

$$\frac{\Psi \vDash^{\text{SDP}} M_1 \hookrightarrow \mathbf{box}\, M_1' : \Box A_1 \quad \Psi \vDash^{\text{SDP}} M_2\{M_1'/x\} \hookrightarrow V : A_2}{\Psi \vDash^{\text{SDP}} \mathbf{let\ box}\ x : A_1 = M_1\ \mathbf{in}\, M_2 \hookrightarrow V : A_2}\ \text{ev\_letb}$$

$$\frac{\begin{array}{c}\Psi \vDash^{\text{SDP}} M \hookrightarrow \mathbf{box}\, M' : \Box B \\ \varnothing \vDash^{\text{SDP}} M' \Uparrow V' : B \\ \Psi \vDash^{\text{SDP}} \langle A, \varnothing, \Theta\rangle(V') \hookrightarrow V : A\langle B\rangle\end{array}}{\Psi \vDash^{\text{SDP}} \mathbf{iter}\, [\Box B, A][\Theta]\ M \hookrightarrow V : A\langle B\rangle}\ \text{ev\_it}$$

Fig. 9. SDP evaluation rules

$\boxed{\langle A, \Psi, \Theta\rangle(M)}$ *Elimination*

$$\frac{}{\langle A, \Psi, \Theta\rangle(x) \triangleq \Theta(x)}\ \mathsf{el\_var} \qquad \frac{}{\langle A, \Psi, \Theta\rangle(c) \triangleq \Theta(c)}\ \mathsf{el\_const}$$

$$\frac{\langle A, \Psi \uplus \{x : B\}, \Theta\rangle(V) \triangleq M}{\langle A, \Psi, \Theta\rangle(\lambda x : B.V) \triangleq \lambda x : A\langle B\rangle.M}\ \mathsf{el\_lam}$$

$$\frac{\langle A, \Psi, \Theta\rangle(V_1) \triangleq M_1 \quad \langle A, \Psi, \Theta\rangle(V_2) \triangleq M_2}{\langle A, \Psi, \Theta\rangle(V_1\, V_2) \triangleq M_1 M_2}\ \mathsf{el\_app} \qquad \frac{\langle A, \Psi, \Theta\rangle(V) \triangleq M}{\langle A, \Psi, \Theta\rangle(\mathbf{fst}\, V) \triangleq \mathbf{fst}\, M}\ \mathsf{el\_fst}$$

$$\frac{\langle A, \Psi, \Theta\rangle(V) \triangleq M}{\langle A, \Psi, \Theta\rangle(\mathbf{snd}\, V) \triangleq \mathbf{snd}\, M}\ \mathsf{el\_snd} \qquad \frac{\langle A, \Psi, \Theta\rangle(V_1) \triangleq M_1 \quad \langle A, \Psi, \Theta\rangle(V_2) \triangleq M_2}{\langle A, \Psi, \Theta\rangle(\langle V_1, V_2\rangle) \triangleq \langle M_1, M_2\rangle}\ \mathsf{el\_prod}$$

$$\frac{}{\langle A, \Psi, \Theta\rangle(\langle\rangle) \triangleq \langle\rangle}\ \mathsf{el\_unit}$$

Fig. 10. SDP elimination rules

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

$$\frac{\begin{array}{c}\Psi \Vdash^{\text{SDP}} M \hookrightarrow \mathbf{box}\, M' : \Box B \\ \varnothing \Vdash^{\text{SDP}} M' \Uparrow V' : B \\ \Psi \Vdash^{\text{SDP}} \langle A, \Psi, \Theta\rangle(V') \hookrightarrow V : A\langle B\rangle\end{array}}{\Psi \Vdash^{\text{SDP}} \mathbf{iter}\, [\Box B, A][\Theta]\ M \hookrightarrow V : A\langle B\rangle}\ \mathsf{ev\_it}$$

First, the argument to iteration M is evaluated, $\Psi \Vdash^{\text{SDP}} M \hookrightarrow \mathbf{box}\, M' : \Box B$, producing a **box**ed object M'. M' is then evaluated to η-long canonical form via $\varnothing \Vdash^{\text{SDP}} M' \Uparrow V' : B$. 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, $\Psi \Vdash^{\text{SDP}} \langle A, \Psi, \Theta\rangle(V') \hookrightarrow V : A\langle B\rangle$.

Elimination, described in Figure 10, is used to describe the structure of a term after iteration. The only interesting cases to elimination are those for variables, constants, and abstractions. When elimination encounters an abstraction, it chooses a fresh variable and adds it to the mapping Θ. It then eliminates recursively on the body M of the abstraction, wrapping the result with an abstraction of the correct type, one where b is replaced by A. The variable and the constant cases use the mappings in the replacement Θ.

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.

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

Fig. 11. Syntax of $\mathbb{F}_\omega$ with unit, void, products, and variants

$\boxed{\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau : \kappa}$ *Well-formed types*

$$\frac{\Delta(\alpha) = \kappa}{\Delta \overset{\mathbb{F}_\omega}{\vdash} \alpha : \kappa}\ \mathsf{wf\_tvar} \qquad \frac{\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 : \star \quad \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_2 : \star}{\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \to \tau_2 : \star}\ \mathsf{wf\_arrow} \qquad \frac{\Delta \uplus \{\alpha : \kappa\} \overset{\mathbb{F}_\omega}{\vdash} \tau : \star}{\Delta \overset{\mathbb{F}_\omega}{\vdash} \forall \alpha : \kappa.\tau : \star}\ \mathsf{wf\_forall}$$

$$\frac{}{\Delta \overset{\mathbb{F}_\omega}{\vdash} 1 : \star \to \star}\ \mathsf{wf\_unit} \qquad \frac{}{\Delta \overset{\mathbb{F}_\omega}{\vdash} 0 : \star}\ \mathsf{wf\_void} \qquad \frac{\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 : \star \quad \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_2 : \star}{\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \times \tau_2 : \star}\ \mathsf{wf\_times}$$

$$\frac{\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 : \star \quad \ldots \quad \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_n : \star}{\Delta \overset{\mathbb{F}_\omega}{\vdash} \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle : \star}\ \mathsf{wf\_variant} \qquad \frac{\Delta \uplus \{\alpha : \kappa_1\} \overset{\mathbb{F}_\omega}{\vdash} \tau : \kappa_2}{\Delta \overset{\mathbb{F}_\omega}{\vdash} \lambda \alpha : \kappa_1.\tau : \kappa_1 \to \kappa_2}\ \mathsf{wf\_abs}$$

$$\frac{\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 : \kappa_1 \to \kappa_2 \quad \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_2 : \kappa_1}{\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \tau_2 : \kappa_2}\ \mathsf{wf\_app}$$

$\boxed{\Delta \vdash \Gamma}$ *Well-formed environments*

$$\frac{\forall x : \tau \in \Gamma \quad \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau : \star}{\Delta \overset{\mathbb{F}_\omega}{\vdash} \Gamma}\ \mathsf{wf\_env}$$

Fig. 12. Well-formedness of types and environments for $\mathbb{F}_\omega$

5 Encoding SDP in $\mathbb{F}_\omega$

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 $\mathbb{F}_\omega$ 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 cre-

$\boxed{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e : \tau}$

$$\frac{\Delta \vdash \Gamma \quad \Gamma(x) = \tau}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} x : \tau}\ \textsf{tp\_var} \qquad \frac{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e : \tau \quad \Delta \vdash^{\mathbb{F}\omega} \tau \equiv_{\beta\eta} \tau' : \star}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e : \tau'}\ \textsf{tp\_eq}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} \tau_1 : \star \quad \Delta;\Gamma \uplus \{x : \tau_1\} \vdash^{\mathbb{F}\omega} e : \tau_2}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} \lambda x : \tau_1.e : \tau_1 \to \tau_2}\ \textsf{tp\_abs}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e_1 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash^{\mathbb{F}\omega} e_2 : \tau_1}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e_1 e_2 : \tau_2}\ \textsf{tp\_app} \qquad \frac{\Delta \vdash \Gamma}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} \langle\rangle : \forall\alpha : \star.1(\alpha)}\ \textsf{tp\_unit}$$

$$\frac{\Delta \uplus \{\alpha : \kappa\};\Gamma \vdash^{\mathbb{F}\omega} e : \tau}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} \Lambda\alpha{:}\kappa.e : \forall\alpha{:}\kappa.\tau}\ \textsf{tp\_tabs} \qquad \frac{\Delta \vdash^{\mathbb{F}\omega} \tau' : \kappa \quad \Delta;\Gamma \vdash^{\mathbb{F}\omega} e : \forall\alpha{:}\kappa.\tau}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e[\tau'] : \tau\{\tau'/\alpha\}}\ \textsf{tp\_tapp}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e_1 : \tau_1 \quad \Delta;\Gamma \vdash^{\mathbb{F}\omega} e_2 : \tau_2}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} \langle e_1, e_2\rangle : \tau_1 \times \tau_2}\ \textsf{tp\_pair} \qquad \frac{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} \mathbf{fst}\, e : \tau_1}\ \textsf{tp\_fst}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}\omega} e : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} \mathbf{snd}\, e : \tau_2}\ \textsf{tp\_snd}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} \tau_1 : \star \quad \ldots \quad \Delta;\Gamma \vdash^{\mathbb{F}\omega} e : \tau_i \quad \ldots \quad \Delta \vdash^{\mathbb{F}\omega} \tau_n : \star}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} \mathbf{inj}_{l_i}\, e\, \mathbf{of}\, \langle l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\rangle : \langle l_1 : \tau_1, \ldots, l_i : \tau_i, \ldots, l_n : \tau_n\rangle}\ \textsf{tp\_variant}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash^{\mathbb{F}\omega} e : \langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle \\ \Delta;\Gamma \uplus \{x_1 : \tau_1\} \vdash^{\mathbb{F}\omega} e_1 : \tau \quad \ldots \quad \Delta;\Gamma \uplus \{x_n : \tau_n\} \vdash^{\mathbb{F}\omega} e_n : \tau\end{array}}{\Delta;\Gamma \vdash^{\mathbb{F}\omega} \mathbf{case}\, e\, \mathbf{of}\, \mathbf{inj}_{l_1}\, x_1\, \mathbf{in}\, e_1\ \ldots\ \mathbf{inj}_{l_n}\, x_n\, \mathbf{in}\, e_n : \tau}\ \textsf{tp\_case}$$

Fig. 13. Typing rules for $\mathbb{F}_\omega$

ate 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. The essence of this encoding was also used by Honsell and Miculan in formalizing dynamic logic within a logical framework (Honsell & Miculan, 1995).

- 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 presen-

$\boxed{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa}$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} \tau : \kappa}{\Delta \vdash^{\mathbb{F}\omega} \tau \equiv_{\beta\eta} \tau : \kappa}\ \textsf{tp\_eq\_refl} \qquad \frac{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa}{\Delta \vdash^{\mathbb{F}\omega} \tau_2 \equiv_{\beta\eta} \tau_1 : \kappa}\ \textsf{tp\_eq\_sym}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa \quad \Delta \vdash^{\mathbb{F}\omega} \tau_2 \equiv_{\beta\eta} \tau_3 : \kappa}{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_3 : \kappa}\ \textsf{tp\_eq\_trans} \qquad \frac{\Delta(\alpha) = \kappa}{\Delta \vdash^{\mathbb{F}\omega} \alpha \equiv_{\beta\eta} \alpha : \kappa}\ \textsf{tp\_eq\_var}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} (\lambda\alpha : \kappa_1.\tau)\tau' : \kappa \quad \text{or} \quad \Delta \vdash^{\mathbb{F}\omega} \tau\{\tau'/\alpha\} : \kappa}{\Delta \vdash^{\mathbb{F}\omega} (\lambda\alpha : \kappa_1.\tau)\tau' \equiv_{\beta\eta} \tau\{\tau'/\alpha\} : \kappa}\ \textsf{tp\_eq\_abs\_beta}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} (\lambda\alpha : \kappa_1.\tau\alpha) : \kappa_1 \to \kappa_2 \quad \text{or} \quad \Delta \vdash^{\mathbb{F}\omega} \tau : \kappa_1 \to \kappa_2 \quad \alpha \notin \mathrm{FTV}(\tau)}{\Delta \vdash^{\mathbb{F}\omega} (\lambda\alpha : \tau_1.\tau\alpha) \equiv_{\beta\eta} \tau : \kappa_1 \to \kappa_2}\ \textsf{tp\_eq\_abs\_eta}$$

$$\frac{\Delta \uplus \{\alpha : \kappa_1\} \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa_2}{\Delta \vdash^{\mathbb{F}\omega} \lambda\alpha : \kappa_1.\tau_1 \equiv_{\beta\eta} \lambda\alpha : \kappa_1.\tau_2 : \kappa_1 \to \kappa_2}\ \textsf{tp\_eq\_abs}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_3 : \kappa_1 \to \kappa_2 \quad \Delta \vdash^{\mathbb{F}\omega} \tau_2 \equiv_{\beta\eta} \tau_4 : \kappa_1}{\Delta \vdash^{\mathbb{F}\omega} \tau_1\tau_2 \equiv_{\beta\eta} \tau_3\tau_4 : \kappa_2}\ \textsf{tp\_eq\_app}$$

$$\frac{}{\Delta \vdash^{\mathbb{F}\omega} 1 \equiv_{\beta\eta} 1 : \star \to \star}\ \textsf{tp\_eq\_unit} \qquad \frac{}{\Delta \vdash^{\mathbb{F}\omega} 0 \equiv_{\beta\eta} 0 : \star}\ \textsf{tp\_eq\_void}$$

$$\frac{\Delta \uplus \{\alpha : \kappa\} \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_2 : \star}{\Delta \vdash^{\mathbb{F}\omega} \forall\alpha : \kappa.\tau_1 \equiv_{\beta\eta} \forall\alpha : \kappa.\tau_2 : \star}\ \textsf{tp\_eq\_forall}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_3 : \star \quad \Delta \vdash^{\mathbb{F}\omega} \tau_2 \equiv_{\beta\eta} \tau_4 : \star}{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \to \tau_2 \equiv_{\beta\eta} \tau_3 \to \tau_4 : \star}\ \textsf{tp\_eq\_arrow}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_3 : \star \quad \Delta \vdash^{\mathbb{F}\omega} \tau_2 \equiv_{\beta\eta} \tau_4 : \star}{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \times \tau_2 \equiv_{\beta\eta} \tau_3 \times \tau_4 : \star}\ \textsf{tp\_eq\_times}$$

$$\frac{\Delta \vdash^{\mathbb{F}\omega} \tau_1 \equiv_{\beta\eta} \tau_1' : \star \quad \ldots \quad \Delta \vdash^{\mathbb{F}\omega} \tau_n \equiv_{\beta\eta} \tau_n' : \star}{\Delta \vdash^{\mathbb{F}\omega} \langle l_1 : \tau_1, \ldots, l_n : \tau_n\rangle \equiv_{\beta\eta} \langle l_1 : \tau_1', \ldots, l_n : \tau_n'\rangle : \star}\ \textsf{tp\_eq\_variant}$$

Fig. 14. Type congruences for $\mathbb{F}_\omega$

tation, we extend the target language with unit, void, products, and variants. The syntax of these terms appears in Figure 11. In the remainder of this section, we present the details of the encoding and describe the most interesting cases.

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

$\boxed{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_1 \equiv_{\beta\eta} e_2 : \tau}$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e : \tau}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e \equiv_{\beta\eta} e : \tau}\ \textsf{eq\_refl} \qquad \frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_1 \equiv_{\beta\eta} e_2 : \tau}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_2 \equiv_{\beta\eta} e_1 : \tau}\ \textsf{eq\_sym}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_1 \equiv_{\beta\eta} e_2 : \tau \quad \Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_2 \equiv_{\beta\eta} e_3 : \tau}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_1 \equiv_{\beta\eta} e_3 : \tau}\ \textsf{eq\_trans}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e \equiv_{\beta\eta} e' : \tau \quad \Delta \vdash^{\mathbb{F}_\omega} \tau \equiv_{\beta\eta} \tau' : \star}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e \equiv_{\beta\eta} e' : \tau'}\ \textsf{eq\_tp\_eq} \qquad \frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e : 1(\tau)}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e \equiv_{\beta\eta} \langle\rangle[\tau] : 1\tau}\ \textsf{eq\_unit}$$

$$\frac{\Delta \vdash^{\mathbb{F}_\omega} \Gamma \quad \Gamma(x) = \tau}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} x \equiv_{\beta\eta} x : \tau}\ \textsf{eq\_var}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} (\lambda x : \tau_1.e)e' : \tau \quad \text{or} \quad \Delta;\Gamma \vdash^{\mathbb{F}_\omega} e\{e'/x\} : \tau}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} (\lambda x : \tau_1.e)e' \equiv_{\beta\eta} e\{e'/x\} : \tau}\ \textsf{eq\_abs\_beta}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} (\lambda x : \tau_1.ex) : \tau_1 \to \tau_2 \quad \text{or} \quad \Delta;\Gamma \vdash^{\mathbb{F}_\omega} e : \tau_1 \to \tau_2 \quad x \notin \mathrm{FV}(e)}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} (\lambda x : \tau_1.ex) \equiv_{\beta\eta} e : \tau_1 \to \tau_2}\ \textsf{eq\_abs\_eta}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} (\Lambda\alpha : \kappa.e)[\tau] : \tau' \quad \text{or} \quad \Delta;\Gamma \vdash^{\mathbb{F}_\omega} e\{\tau/\alpha\} : \tau'}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} (\Lambda\alpha : \kappa.e)[\tau] \equiv_{\beta\eta} e\{\tau/\alpha\} : \tau'}\ \textsf{eq\_tabs\_beta}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} \Lambda\alpha : \kappa.e[\alpha] : \forall\alpha : \kappa.\tau \quad \text{or} \quad \Delta;\Gamma \vdash^{\mathbb{F}_\omega} e : \forall\alpha : \kappa.\tau \quad \alpha \notin \mathrm{FTV}(e)}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} (\Lambda\alpha : \kappa.e[\alpha]) \equiv_{\beta\eta} e : \forall\alpha : \kappa.\tau}\ \textsf{eq\_tabs\_eta}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} \langle e_1, e_2\rangle : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} \mathbf{fst}\,\langle e_1, e_2\rangle \equiv_{\beta\eta} e_1 : \tau_1}\ \textsf{eq\_pair\_beta1}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} \langle e_1, e_2\rangle : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} \mathbf{snd}\,\langle e_1, e_2\rangle \equiv_{\beta\eta} e_2 : \tau_2}\ \textsf{eq\_pair\_beta2}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} \langle \mathbf{fst}\, e, \mathbf{snd}\, e\rangle \equiv_{\beta\eta} e : \tau_1 \times \tau_2}\ \textsf{eq\_pair\_eta}$$

$$\frac{\Delta;\Gamma \uplus \{x : \tau_1\} \vdash^{\mathbb{F}_\omega} e_1 \equiv_{\beta\eta} e_2 : \tau_3 \quad \Delta \vdash^{\mathbb{F}_\omega} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} \lambda x : \tau_1.e_1 \equiv_{\beta\eta} \lambda x : \tau_2.e_2 : \tau_1 \to \tau_3}\ \textsf{eq\_abs}$$

$$\frac{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_1 \equiv_{\beta\eta} e_3 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_2 \equiv_{\beta\eta} e_4 : \tau_1}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} e_1 e_2 \equiv_{\beta\eta} e_3 e_4 : \tau_2}\ \textsf{eq\_app}$$

$$\frac{\Delta \uplus \{\alpha : \kappa\};\Gamma \vdash^{\mathbb{F}_\omega} e_1 \equiv_{\beta\eta} e_2 : \tau}{\Delta;\Gamma \vdash^{\mathbb{F}_\omega} \Lambda\alpha : \kappa.e_1 \equiv_{\beta\eta} \Lambda\alpha : \kappa.e_2 : \forall\alpha : \kappa.\tau}\ \textsf{eq\_tabs}$$

Fig. 15. Term congruences for $\mathbb{F}_\omega$ (part one)

$\boxed{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_1 \equiv_{\beta\eta} e_2 : \tau}$

$$\frac{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_1 \equiv_{\beta\eta} e_2 : \forall \alpha : \kappa.\tau \quad \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa}{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_1[\tau_1] \equiv_{\beta\eta} e_2[\tau_2] : \tau\{\tau_1/\alpha\}} \text{ eq\_tapp}$$

$$\frac{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_1 \equiv_{\beta\eta} e_3 : \tau_1 \quad \Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_2 \equiv_{\beta\eta} e_4 : \tau_2}{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} \langle e_1, e_2 \rangle \equiv_{\beta\eta} \langle e_3, e_4 \rangle : \tau_1 \times \tau_2} \text{ eq\_pair}$$

$$\frac{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_1 \equiv_{\beta\eta} e_2 : \tau_1 \times \tau_2}{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} \mathbf{fst}\, e_1 \equiv_{\beta\eta} \mathbf{fst}\, e_2 : \tau_1} \text{ eq\_fst} \qquad \frac{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_1 \equiv_{\beta\eta} e_2 : \tau_1 \times \tau_2}{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} \mathbf{snd}\, e_1 \equiv_{\beta\eta} \mathbf{snd}\, e_2 : \tau_2} \text{ eq\_snd}$$

$$\frac{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_1 \equiv_{\beta\eta} e_2 : \tau_1 \quad \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa}{\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} \mathbf{inj}_l\, e_1\, \mathbf{of}\, \tau_1 \equiv_{\beta\eta} \mathbf{inj}_l\, e_2\, \mathbf{of}\, \tau_2 : \langle \ldots, l : \tau_1, \ldots \rangle} \text{ eq\_inj}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} e_1 \equiv_{\beta\eta} e_2 : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \\ \Delta; \Gamma \uplus \{y_1 : \tau_1\} \overset{\mathbb{F}_\omega}{\vdash} e_1' \equiv_{\beta\eta} e_1'' : \tau' \quad \ldots \quad \Delta; \Gamma \uplus \{y_n : \tau_n\} \overset{\mathbb{F}_\omega}{\vdash} e_n' \equiv_{\beta\eta} e_n'' : \tau'\end{array}}{\begin{array}{l}\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} \mathbf{case}\, e_1\, \mathbf{of}\, \mathbf{inj}_{l_1}\, y_1\, \mathbf{in}\, e_1' \\ \qquad \ldots \\ \qquad \mathbf{inj}_{l_n}\, y_n\, \mathbf{in}\, e_n'\end{array} \equiv_{\beta\eta} \begin{array}{l}\mathbf{case}\, e_2\, \mathbf{of}\, \mathbf{inj}_{l_1}\, y_1\, \mathbf{in}\, e_1'' \\ \qquad \ldots \\ \qquad \mathbf{inj}_{l_n}\, y_n\, \mathbf{in}\, e_n''\end{array} : \tau_2} \text{ eq\_case}$$

$$\frac{\Delta; \Gamma \uplus \{y_1 : \tau_1\} \overset{\mathbb{F}_\omega}{\vdash} e_1 : \tau' \quad \ldots \quad \Delta; \Gamma \uplus \{y_n : \tau_n\} \overset{\mathbb{F}_\omega}{\vdash} e_n : \tau'}{\begin{array}{l}\Delta; \Gamma \overset{\mathbb{F}_\omega}{\vdash} \mathbf{case}\, (\mathbf{inj}_{l_i}\, e\, \mathbf{of}\,)\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle\, \mathbf{of} \\ \qquad \mathbf{inj}_{l_1}\, y_1\, \mathbf{in}\, e_1 \\ \qquad \ldots \\ \qquad \mathbf{inj}_{l_n}\, y_n\, \mathbf{in}\, e_n\end{array} \equiv_{\beta\eta} e_i\{e/y_i\} : \tau_2} \text{ eq\_case\_beta}$$

Fig. 16. Term congruences for $\mathbb{F}_\omega$ (part two)

need an operation called *parameterization*, notated $\tau\langle B \rangle$ and defined in Figure 18. 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.

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

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

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

If were to start from the version of SDP that includes multiple base types, each disjoint base type in the signature would be translated to a different variant type. If there was a mutually recursive dependency between two or more base types, we

$\boxed{\tau_1\langle\Sigma\rangle \triangleq \tau_2}$ *Signatures*

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

$\boxed{\Delta \vdash A \rhd_{\tau_1} \tau_2}$ *Types*

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

$$\frac{}{\Delta \vdash 1 \rhd_\tau 1(\tau)}\ \textsf{en\_tp\_unit} \qquad \frac{\Delta \vdash A_1 \rhd_\tau \tau_1 \quad \Delta \vdash A_2 \rhd_\tau \tau_2}{\Delta \vdash A_1 \to A_2 \rhd_\tau \tau_1 \to \tau_2}\ \textsf{en\_tp\_arrow}$$

$$\frac{\Delta \vdash A_1 \rhd_\tau \tau_1 \quad \Delta \vdash A_2 \rhd_\tau \tau_2}{\Delta \vdash A_1 \times A_2 \rhd_\tau \tau_1 \times \tau_2}\ \textsf{en\_tp\_prod}$$

$\boxed{\Delta;\Xi \vdash M \rhd_\tau e}$ *Terms*

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

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

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

$$\frac{\Delta;\Xi \vdash M \rhd_\tau e \quad \Delta \vdash A_1 \rhd_\tau \tau_1}{\Delta;\Xi \vdash \lambda x : A_1.M \rhd_\tau \lambda x : \tau_1.e}\ \textsf{en\_abs} \qquad \frac{\Delta;\Xi \vdash M_1 \rhd_\tau e_1 \quad \Delta;\Xi \vdash M_2 \rhd_\tau e_2}{\Delta;\Xi \vdash M_1 M_2 \rhd_\tau e_1 e_2}\ \textsf{en\_app}$$

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

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

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

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

$\boxed{\Delta;\Xi \vdash \Theta \rhd_{\tau_2}^{\tau_1} e}$ *Replacements*

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

Fig. 17. Full encoding of SDP

$$\begin{array}{rcl} \tau\langle \mathsf{b} \rangle & \triangleq & \tau \\ \tau\langle 1 \rangle & \triangleq & 1(\tau) \\ \tau\langle \mathsf{B}_1 \to \mathsf{B}_2 \rangle & \triangleq & \tau\langle \mathsf{B}_1 \rangle \to \tau\langle \mathsf{B}_2 \rangle \\ \tau\langle \mathsf{B}_1 \times \mathsf{B}_2 \rangle & \triangleq & \tau\langle \mathsf{B}_1 \rangle \times \tau\langle \mathsf{B}_2 \rangle \end{array}$$

Fig. 18. Parameterization

would need to combine them into a single base type using a technique like the Bekić-Leszczyłowski Theorem, and then encode that new type as a single variant in $\mathbb{F}_\omega$ (1984; 2005).

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 choose 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.

$$\frac{}{\Delta \vdash \mathsf{b} \rhd_\tau \mathsf{Rec}\ \Sigma^*\ \tau}\ \mathsf{en\_tp\_b}$$

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.

$$\frac{\alpha \notin \Delta \quad \Delta \uplus \{\alpha : \star\} \vdash A \rhd_\alpha \tau'}{\Delta \vdash \Box A \rhd_\tau \forall \alpha{:}\star.\tau'}\ \mathsf{en\_tp\_box\_wrong}$$

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 by Pfenning and Davies (Davies & Pfenning, 2001). Their type system is inspired by the semantic interpretation of S4 modal logic in terms of Kripke models (Kripke, 1959).

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

The naïve translation of the unit type also forgets the current world. For this reason, we add a non-standard unit to $\mathbb{F}_\omega$ that is parameterized by the current

world. In other words, the unit type $\mathsf{1}$ is of kind $\star \to \star$ and the unit term $\langle\rangle$ has type $\forall\alpha{:}\star.\mathsf{1}(\alpha)$.

Our type translation instantiates this type with the current world.

$$\frac{}{\Delta \vdash \mathsf{1} \rhd_\tau \mathsf{1}(\tau)}\ \mathsf{en\_tp\_unit}$$

This unconventional definition for unit is not a semantic problem, as it is derivable from a more standard unit, with term `unit` of type `Unit`, as follows:

$$\begin{array}{rcl} \mathsf{1} & \triangleq & \lambda\alpha{::}\star.\alpha \to \mathtt{Unit} \\ \langle\rangle & \triangleq & \Lambda\alpha{::}\star.\lambda x{:}\alpha.\mathtt{unit} \end{array}$$

The semantics of our $\mathsf{1}$ and $\langle\rangle$, including the η-rule $\mathsf{eq\_unit}$ in Figure 15, are fully derivable from the standard semantics for `unit` and `Unit`.

The remaining types in the SDP language are encoded recursively in a straightforward manner. The complete rules can be found in Figure 17.

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.

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

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.

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

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.

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

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

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

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

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

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

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

Fig. 19. Library routines

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.

Figure 19 shows the types and definitions of the library routines used by the encoding. The only difference between it and Figure 5 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 $\mathbb{F}_\omega$. 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 Figure 2. We do not cover the implementation here, see Hinze (2002) 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, $\mathsf{Rec}\ \Sigma^*\ \tau$.

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

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.

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

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.

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

The encodings for the other terms in the source language are straightforward and appear in Figure 17. 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.

6 Static correctness

Our notion of static correctness is that 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.

However, because we allow for the encoding of open terms, before we can begin to reason about static correctness and other properties, we must first define a relationship between source and target language environments.

Definition 6.1 (*Encoding typing environments*)
We write $\Delta \vdash \Upsilon \rhd_\tau \Gamma_1$ and $\Delta \vdash \Omega \rhd \Gamma_2$ to mean that

$$\begin{array}{ll}\forall x.x : A \in \Upsilon \ \Leftrightarrow\ x : \tau_A \in \Gamma_1 & \text{where } \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau : \star \text{ and } \Delta \vdash A \rhd_\tau \tau_A \\ \forall x.x : A \in \Omega \ \Leftrightarrow\ x : \tau_A \in \Gamma_2 & \text{where there exists some } \Delta \overset{\mathbb{F}_\omega}{\vdash} \tau' : \star \text{ such that} \\ & \qquad \Delta \vdash \Box A \rhd_{\tau'} \tau_A\end{array}$$

The relation for valid environments above is not parameterized by the current world. A single valid environment may be encoded at many different target environments, depending on what worlds are chosen for each type in the environment. However, in a 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 following theorem proves our primary static correctness result, supported by a number of lemmas that appear after the proof.

Theorem 6.2 (*Static correctness*)

1. If $\Delta; \mathrm{dom}(\Omega) \vdash M \rhd_\tau e$ then
   if $\Delta \vdash \Upsilon \rhd_\tau \Gamma_1$
   and $\Delta \vdash \Omega \rhd \Gamma_2$
   and $\Delta \vdash A \rhd_\tau \tau_A$
   then $\Omega; \Upsilon \overset{\text{SDP}}{\vdash} M : A \iff \Delta; \Gamma_1 \uplus \Gamma_2 \overset{\mathbb{F}\omega}{\vdash} e : \tau_A$.
2. If $\Delta; \mathrm{dom}(\Omega) \vdash \Theta \rhd_\tau^{\tau_A} e_\Theta$ then
   if $\Delta \vdash \Upsilon \rhd_\tau \Gamma_1$
   and $\Delta \vdash \Omega \rhd \Gamma_2$
   and $\Delta \vdash A \rhd_\tau \tau_A$
   then $\Omega; \Upsilon \overset{\text{SDP}}{\vdash} \Theta : A\langle\Sigma\rangle \iff \Delta; \Gamma_1 \uplus \Gamma_2 \overset{\mathbb{F}\omega}{\vdash} e_\Theta : \Sigma^*\tau_A \to \tau_A$.

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

Most cases make use of Definition 6.1 (environment encoding), Lemma A.12 (environment encoding well-formedness), and Lemma A.3 (inversion). Additionally many cases involving explicitly typed terms make use of Lemma A.7 (uniqueness of type encoding), Lemma A.2 (well-formedness of type encoding), Lemma A.6 (type encodings is total and decidable), and Lemma 6.6 (type encoding with congruent results). The last, we cover in more detail later.

- The case for en_bvar requires Lemma A.9 (world substitution on type encoding), Lemma A.8 (encoding under congruent worlds), and Lemma 6.6 (typing encoding with congruent results).
- The case for en_box requires the most difficult lemma, Lemma 6.3 (local strengthening) which we cover in more detail later.
- The case for en_con requires Lemma A.10 (commutativity of parameterization and type encoding).
- Finally, the case for en_iter makes use of Lemma A.10 (commutativity of parameterization and type encoding) and Lemma A.11 (commutativity of iteration types and type encoding).

□

An important lemma is required for **box**ed terms in the backward direction. To show that the **box**ed term is well-typed in the source language, we need to show that the local environment is empty.

We use the following lemma to do so, which guarantees that if the term is encoded with respect to some world containing a type variable α, if the local environment is encoded with respect to a world that does not contain the type variable α, then those bindings must be unnecessary for the typing derivation.

Lemma 6.3 (*Local strengthening*)

- Assume $\Delta \uplus \{\alpha : \star \to \star\} \vdash \Omega \rhd \Gamma_1$ and $\Delta \vdash \Upsilon \rhd_\tau \Gamma_2$ and $\alpha \notin \mathrm{FTV}(\tau)$.
- If $\Delta \uplus \{\alpha : \star \to \star\}; \mathrm{dom}(\Omega) \vdash M \rhd_{\alpha\tau} e$
  and $\Delta \uplus \{\alpha : \star \to \star\}; \Gamma_1 \uplus \Gamma_2 \overset{\mathrm{SDP}}{\vdash} e : \tau'$
  where $\Delta \uplus \{\alpha : \star \to \star\} \vdash A \rhd_{\alpha\tau} \tau'$
  then $\Delta \uplus \{\alpha : \star \to \star\}; \Gamma_1 \overset{\mathbb{F}_\omega}{\vdash} e : \tau'$

Proof
We cannot prove this lemma directly, but must instead generalize the induction hypothesis, yielding the next Lemma 6.4 (superfluous context elimination). It then follows by instantiating Lemma 6.4 with $\Upsilon_i \in \{\Upsilon\}$ and $\Upsilon = \varnothing$. □

Lemma 6.4 (*Superfluous context elimination*)

1. If $\Delta \uplus \{\alpha : \star \to \star\}; \mathrm{dom}(\Omega) \vdash M \rhd_{\alpha\tau} e$
   and $\Delta \vdash \Upsilon_i \rhd_{\tau_i} \Gamma_i$
   and $\alpha \notin \mathrm{FTV}(\Gamma_i)$
   and $\Delta \uplus \{\alpha : \star \to \star\} \vdash \Omega \rhd \Gamma$
   and $\Delta \uplus \{\alpha : \star \to \star\} \vdash \Upsilon' \rhd_{\alpha\tau} \Gamma'$
   and $\Delta \uplus \{\alpha : \star \to \star\}; \Gamma \uplus \Gamma_1 \uplus \ldots \uplus \Gamma_n \uplus \Gamma' \overset{\mathbb{F}_\omega}{\vdash} e : \tau'$
   where $\Delta \uplus \{\alpha : \star \to \star\} \vdash A \rhd_{\alpha\tau} \tau'$
   then $\Delta \uplus \{\alpha : \star \to \star\}; \Gamma \uplus \Gamma' \overset{\mathbb{F}_\omega}{\vdash} e : \tau'$.
2. If $\Delta \uplus \{\alpha : \star \to \star\}; \mathrm{dom}(\Omega) \vdash \Theta \rhd^{\tau_A}_{\alpha\tau} e_\Theta$
   and $\Delta \vdash \Upsilon_i \rhd_{\tau_i} \Gamma_i$
   and $\alpha \notin \mathrm{FTV}(\Gamma_i)$
   and $\Delta \uplus \{\alpha : \star \to \star\} \vdash \Omega \rhd \Gamma$
   and $\Delta \uplus \{\alpha : \star \to \star\} \vdash \Upsilon' \rhd_{\alpha\tau} \Gamma'$
   and$\Delta \uplus \{\alpha : \star \to \star\}; \Gamma \uplus \Gamma_1 \uplus \ldots \uplus \Gamma_n \uplus \Gamma' \overset{\mathbb{F}_\omega}{\vdash} e_\Theta : \Sigma^* \tau_A \to \tau_A$
   where $\Delta \uplus \{\alpha : \star \to \star\} \vdash A \rhd_{\alpha\tau} \tau_A$
   then $\Delta \uplus \{\alpha : \star \to \star\}; \Gamma \uplus \Gamma' \overset{\mathbb{F}_\omega}{\vdash} e_\Theta : \Sigma^* \tau_A \to \tau_A$.

Proof
By mutual induction over the structure of $\Delta \uplus \{\alpha : \star \to \star\}; \mathrm{dom}(\Omega) \vdash M \rhd_{\alpha\tau} e$ and $\Delta \uplus \{\alpha : \star \to \star\}; \mathrm{dom}(\Omega) \vdash \Theta \rhd^{\tau_A}_{\alpha\tau} e_\Theta$.

Most cases only require Definition 6.1 (environment encoding), Lemma A.3 (inversion for typing derivations), and Lemma A.3 (inversion for type encoding). Additionally, the case for en_var uses Lemma A.13 (type containment) and the case for en_rep uses Lemma A.10 (commutativity for iteration types). □

Since System $\mathbb{F}_\omega$ treats types identical up the equivalence relation $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa$, inversion lemmas that rely on the structure of types, such as inversion on typing derivations, type congruences, and type encoding do not follow trivially by inspection. However, it is possible to strengthen some of these inversion lemmas by recognizing that type encoding always produces types in $\mathbb{F}_\omega$ that are in weak head normal form. We use the judgment $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau \upharpoonright \kappa$ to indicate that type τ with kind κ is in weak head normal form with respect to Δ. See Figure 20 for the definition.

$$\frac{\Delta(\alpha) = \kappa}{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \alpha \upharpoonright \kappa}\ \textsf{whnf\_var} \qquad \frac{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 : \star \quad \Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_2 : \star}{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 \to \tau_2 \upharpoonright \star}\ \textsf{whnf\_arrow}$$

$$\frac{\Delta \uplus \{\alpha : \kappa\} \stackrel{\mathbb{F}\omega}{\vdash} \tau : \star}{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \forall \alpha : \kappa.\tau \upharpoonright \star}\ \textsf{whnf\_forall} \qquad \frac{}{\Delta \stackrel{\mathbb{F}\omega}{\vdash} 1 \upharpoonright \star \to \star}\ \textsf{whnf\_unit} \qquad \frac{}{\Delta \stackrel{\mathbb{F}\omega}{\vdash} 0 \upharpoonright \star}\ \textsf{whnf\_void}$$

$$\frac{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 : \star \quad \Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_2 : \star}{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 \times \tau_2 \upharpoonright \star}\ \textsf{whnf\_times} \qquad \frac{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 : \star \quad \ldots \quad \Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_2 : \star}{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \upharpoonright \star}\ \textsf{whnf\_variant}$$

$$\frac{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 \upharpoonright \kappa_1 \to \kappa_2 \quad \Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_2 : \kappa_1}{\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 \tau_2 \upharpoonright \kappa_2}\ \textsf{whnf\_app}$$

Fig. 20. Definition of weak-head normal types for $\mathbb{F}_\omega$

Lemma 6.5 (*Type encodings are weak head normal forms*)
If $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau : \star$ and $\Delta \vdash A \rhd_\tau \tau_A$ then $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_A \upharpoonright \star$.

Proof
By straightforward induction over the structure of $\Delta \vdash A \rhd_\tau \tau_A$. □

Another difficulty that arises in the backward direction of the static correctness proof is showing that two types, known only to be congruent, are the result of encoding the same source language type. It is possible to further strengthen the conclusion of the following lemma to also state that τ_1 and τ_2 must also be syntactically in addition to semantically equivalent using Lemma A.6, but it is not necessary for the proofs.

Lemma 6.6 (*Type encoding with congruent results*)
If $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau : \star$ and $\Delta \vdash A_1 \rhd_\tau \tau_1$ and $\Delta \vdash A_2 \rhd_\tau \tau_2$ where $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_2 : \star$
then $A_1 = A_2$.

Proof
By induction over the structure of $\Delta \vdash A_1 \rhd_\tau \tau_1$ using inversion on $\Delta \vdash A_2 \rhd_\tau \tau_2$. □

7 Dynamic correctness

We prove the dynamic correctness of our encoding with respect to the equivalence relation $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} e \equiv_{\beta\eta} e' : \tau$ between target terms of type τ. This congruence relation includes the standard β and η-equivalences for functions, products and unit. The complete definition can be found in Figures 14, 15, and 16. We will use the equals symbol, $=$, when we intend syntactic equality.

Theorem 7.1 (*Dynamic Correctness*)
If $\varnothing;\Psi \vDash^{\text{SDP}} M : A$ and $\varnothing;\varnothing \vdash M \rhd_\tau e$ and $\varnothing;\varnothing \vdash V \rhd_\tau e'$ and $\varnothing \vdash A \rhd_\tau \tau_A$ and $\varnothing \vdash \Psi \rhd_\tau \Gamma$ and

1. if $\Psi \vDash^{\text{SDP}} M \hookrightarrow V : A \;\Leftrightarrow\; \varnothing;\Gamma \vDash^{\mathbb{F}\omega} e \equiv_{\beta\eta} e' : \tau_A$.
2. if $\Psi \vDash^{\text{SDP}} M \Uparrow V : A \;\Leftrightarrow\; \varnothing;\Gamma \vDash^{\mathbb{F}\omega} e \equiv_{\beta\eta} e' : \tau_A$.

Proof
The backward direction follows from the forward direction and from the fact that evaluation in the SDP calculus is deterministic and total(Schürmann *et al.*, 2001).

The forward direction follows by mutual induction over the structure of $\Psi \vDash^{\text{SDP}} M \hookrightarrow V : A$ and $\Psi \vDash^{\text{SDP}} M \Uparrow V : A$. The cases for $\Psi \vDash^{\text{SDP}} M \Uparrow V : A$ are uncomplicated. For $\Psi \vDash^{\text{SDP}} M \hookrightarrow V : A$, only ev_app, ev_letb, and ev_it are nontrivial. All these cases make use of Lemma B.7 (term encoding is total and decidable), type preservation (Schürmann *et al.*, 2001), and β-equivalence.

Additionally,

- The case for ev_app makes use of Lemma A.6 (type encoding total and decidable), Lemma A.7 (uniqueness of type encoding), and Lemma B.6 (substitution for encoding regular term variables).
- The case for ev_letb makes use of Lemma B.9 (substitution for the encoding of modal variables).
- Finally, the case for ev_it requires the properties of evaluation results (Schürmann *et al.*, 2001), Lemma A.6 (type encoding total and decidable), Lemma A.2 (well-formedness of encoding), Lemma B.8 (world substitution for terms), Lemma B.3 (replacements are well-formed dynamic replacements), Lemma B.10 (elimination typing). The proof of ev_it requires Lemma 7.6 (dynamic correctness of elimination) which we will cover in greater detail later.

□

Most of the lemmas for the proof of Theorem 7.1 are straightforward, but proving Lemma 7.6 (dynamic correctness of elimination) requires a considerable amount of technical machinery which is established in the remainder of this section.

In order to aid in reasoning about the operational behavior of iteration, we first define an inverse to **openiter**, called **uniter**, constructed from the second component of **xmap**.

Definition 7.2 (**uniter**)

$$\begin{aligned}
&\mathbf{uniter}\{\!|\tau : \star \to \star|\!\} : \forall\alpha : \star.(\Sigma^*\,\alpha \to \alpha) \to \tau\alpha \to \tau(\mathsf{Rec}\ \Sigma^*\ \alpha)\\
&\mathbf{uniter}\{\!|\tau : \star \to \star|\!\} =\\
&\quad \Lambda\alpha : \star.\lambda f : \Sigma^*\alpha \to \alpha.\mathbf{snd}\,(\mathbf{xmap}\{\!|\tau|\!\}[\mathsf{Rec}\ \Sigma^*\ \alpha][\alpha]\langle\mathbf{cata}[\alpha]f, \mathbf{place}[\alpha]\rangle)
\end{aligned}$$

Statically the source language only allows for replacements for constants, but during iteration mappings for free variables are added to replacements. Therefore, in order to reason about the dynamic correctness of iteration, we need to have some

notion of well-formedness for replacements that contain variable mappings. Furthermore, it is useful to define the notation $\llcorner\Theta\lrcorner$ to indicate a replacement restricting to constants, or more formally $\llcorner\Theta\lrcorner = \{c_i \mapsto M_i \mid \forall c_i \mapsto M_i \in \Theta\}$.

Definition 7.3 (*Well-formed dynamic replacements*)

$$\frac{\begin{array}{ll}\forall c_i \in \mathrm{dom}(\Sigma) & \Sigma(c_i) = B_i \quad \Omega;\Upsilon \vDash^{\text{SDP}} \Theta(c_i) : A\langle B_i\rangle \\ \forall x_i \in \mathrm{dom}(\Psi) & \Psi(x_i) = B'_i \quad \Omega;\Upsilon \vDash^{\text{SDP}} \Theta(x_i) : A\langle B'_i\rangle\end{array}}{\Omega;\Upsilon \vDash^{\text{SDP}} \Theta : A\langle\Psi{:}\Sigma\rangle}\ \textsf{tp\_rep\_vars}$$

Because the operational semantics of the SDP calculus depends on the definition of elimination $\langle A, \Psi, \Theta\rangle(V)$ we must define an encoding from an elimination form to a term in the target calculus so that we may prove dynamic correctness of the encoding. The first step is to define a substitution for all of the free variables in V. We will replace each variable with an **uniter** term that will hold its mapping from Θ. For these derived encodings we will use a black triangle, $\blacktriangleright$, rather than an a white one, $\triangleright$, to help distinguish between them and the standard encodings. We create a substitution (notated $\Delta;\Psi;\Theta;e_\Theta \blacktriangleright^{\tau_A}_{\tau}\ S$) as follows:

Definition 7.4 (*Elimination Substitution*)

$$\frac{}{\Delta;\varnothing;\Theta;e_\Theta \blacktriangleright^{\tau_A}_{\tau}\ \{\}}\ \textsf{sub\_empty}$$

$$\frac{\Delta;\Psi;\Theta;e_\Theta \blacktriangleright^{\tau_A}_{\tau}\ S \quad \Delta;\varnothing \vdash \Theta(x) \rhd_\tau e'}{\Delta;\Psi \uplus \{x : B\};\Theta;e_\Theta \blacktriangleright^{\tau_A}_{\tau}\ S \cdot \{(\mathbf{uniter}[\![B^*]\!][\tau_A]\ e_\Theta\ e')/x\}}\ \textsf{sub\_cons}$$

Then given an elimination, we may encode it with **openiter** as follows:

Definition 7.5 (*Encoding of elimination*)

$$\frac{\begin{array}{c}\Psi \vDash^{\text{SDP}} V \Uparrow B \quad \Delta \vdash A \rhd_\tau \tau_A \\ \Delta;\Xi \vdash \llcorner\Theta\lrcorner \rhd^{\tau_A}_{\tau}\ e_\Theta \quad \Delta;\varnothing \vdash V \rhd_{\tau_A} e' \quad \Delta;\Psi;\Theta;e_\Theta \blacktriangleright^{\tau_A}_{\tau}\ S\end{array}}{\Delta;\Xi \vdash \langle A,\Psi,\Theta\rangle(V) \blacktriangleright_\tau \mathbf{openiter}[\![B^*]\!][\tau_A]\ e_\Theta\ S(e')}\ \textsf{en\_elim}$$

The next lemma states that the encoding of an elimination is β,η-equivalent to the encoding of the result of elimination over M in the source calculus.

Lemma 7.6 (*Dynamic correctness of elimination*)
If $\Omega;\Upsilon \vDash^{\text{SDP}} \Theta : A\langle\Psi{:}\Sigma\rangle$
and $\langle A,\Psi,\Theta\rangle(V) = M$
and $\Delta;\mathrm{dom}(\Omega) \vdash \langle A,\Psi,\Theta\rangle(V) \blacktriangleright^{\tau_A}_{\tau}\ e$
and $\Delta;\mathrm{dom}(\Omega) \vdash M \rhd_\tau e'$
and $\Delta \vdash \Omega \rhd \Gamma_1$
and $\Delta \vdash \Upsilon \rhd_\tau \Gamma_2$
then $\Delta;\Gamma_1 \uplus \Gamma_2 \vDash^{\mathbb{E}\omega} e \equiv_{\beta\eta} e' : B^*\tau_A$.

Proof
By induction on $\langle \mathsf{A}, \Psi, \Theta \rangle(\mathsf{V})$. All cases make use of inversion, Lemma B.2 (properties of **iter**, and the definition of congruence.

- The case for el_var requires the proof typing of atomic and canonical forms (Schürmann *et al.*, 2001), Lemma B.11 (elimination substitution application), Lemma B.10 (typing for elimination), Lemma A.11 (commutativity of encoding on iteration types), and Theorem 6.2 (static correctness, forward direction).
- The case for el_const needs Lemma B.4 (well-formed dynamic replacements are well-typed replacements) and Theorem 6.2 (static correctness, forward direction).
- The most complicated case is el_app which requires a Lemma 7.7 (iteration on atomic applications) which we will discuss in detail later.
- The case for el_lam makes use of Lemma A.11 (commutativity of encoding on iteration types), Lemma A.7 (uniqueness of type encoding), as well as Definition 7.3 (well-formed dynamic replacements) and Definition 6.1 (environment encoding).

□

For the case where $\mathsf{V} = \mathsf{V}_1\mathsf{V}_2$ in the above proof we require the following lemma about how iteration interacts with application:

Lemma 7.7 (*Iteration and atomic applications*)
If $\Omega; \Upsilon \vDash^{\text{SDP}} \Theta : \mathsf{A}\langle\Psi{:}\Sigma\rangle$
and $\Delta \vdash \Omega \rhd \Gamma_1$
and $\Delta \vdash \Upsilon \rhd_\tau \Gamma_2$
and $\Psi \vDash^{\text{SDP}} \mathsf{V}_1\mathsf{V}_2 \downarrow \mathsf{B}_2$
and $\Psi \vDash^{\text{SDP}} \mathsf{V}_1 \downarrow \mathsf{B}_1 \to \mathsf{B}_2$
and $\Psi \vDash^{\text{SDP}} \mathsf{V}_2 \Uparrow \mathsf{B}_1$
and $\Delta; \varnothing \vdash \mathsf{V}_1 \rhd_{\tau_\mathsf{A}} e_1$
and $\Delta; \varnothing \vdash \mathsf{V}_2 \rhd_{\tau_\mathsf{A}} e_2$
and $\Delta; \Psi; \Theta; e_\Theta \blacktriangleright^{\tau_\mathsf{A}}_{\tau} \mathsf{S}$
then
$\Delta; \Gamma_1 \uplus \Gamma_2 \vDash^{\mathbb{E}\omega}$
$\mathbf{openiter}\{\!|\mathsf{B}_2^*|\!\}[\tau_\mathsf{A}]\ e_\Theta\ \mathsf{S}(e_1\ e_2) \equiv_{\beta\eta}$
$(\mathbf{openiter}\{\!|(\mathsf{B}_1 \to \mathsf{B}_2)^*|\!\}[\tau_\mathsf{A}]\ e_\Theta\ \mathsf{S}(e_1))(\mathbf{openiter}\{\!|\mathsf{B}_1^*|\!\}[\tau_\mathsf{A}]\ e_\Theta\ \mathsf{S}(e_2)) : \mathsf{B}^*\tau_\mathsf{A}$

Proof
We cannot prove this lemma directly, but it follows from the more general Lemma 7.17 (Iteration and atomic forms). □

To generalize the induction hypothesis of Lemma 7.7 sufficiently requires the introduction of formal machinery we will call iteration contexts. Iteration contexts provide convenient a formalism to reason about the dynamic behavior of iteration over atomic terms. Our iteration contexts are similar in flavor to evaluation contexts, as they describe a computation that needs a term to proceed. However, iteration contexts describe the computation from the inside out, instead of the outside in.

Definition 7.8 (*Iteration contexts*)

$$\begin{array}{lrcl} (\textit{Iteration Contexts}) & \mathsf{E} & ::= & \bullet \mid \mathsf{E}\{\!\{\bullet\ e\}\!\} \mid \mathsf{E}\{\!\{\mathbf{fst}\ \bullet\}\!\} \mid \mathsf{E}\{\!\{\mathbf{snd}\ \bullet\}\!\} \\ (\textit{Pure Context Types}) & \mathsf{D} & ::= & \bullet \mid \mathsf{B} \to \mathsf{D} \mid \mathsf{D} \times \mathsf{B} \mid \mathsf{B} \times \mathsf{D} \\ (\textit{Context Types}) & \mathsf{C} & ::= & \bullet \mid \mathsf{A} \to \mathsf{C} \mid \mathsf{C} \times \mathsf{A} \mid \mathsf{A} \times \mathsf{C} \end{array}$$

Because of our universal usage of the asterisk type constructor notation, B^*, for pure source language types, it proves convenient to describe iteration contexts types in terms of source language types, despite the fact that the contexts themselves are defined in terms of the target language. Furthermore, because iteration does not necessarily yield pure types in the source language, we also must make a distinction between normal and pure context types. In addition we define a notation of iterated contexts types, analogous to iterated types in the source language.

Definition 7.9 (*Iteration context algebra*)

$$\begin{array}{rcl} \bullet\{\!\{e\}\!\} & \triangleq & e \\ \mathsf{E}\{\!\{\mathbf{fst}\ \bullet\}\!\}\{\!\{e\}\!\} & \triangleq & \mathsf{E}\{\!\{\mathbf{fst}\ e\}\!\} \\ \mathsf{E}\{\!\{\mathbf{snd}\ \bullet\}\!\}\{\!\{e\}\!\} & \triangleq & \mathsf{E}\{\!\{\mathbf{snd}\ e\}\!\} \\ \mathsf{E}\{\!\{\bullet\ e'\}\!\}\{\!\{e\}\!\} & \triangleq & \mathsf{E}\{\!\{e e'\}\!\} \end{array}$$

Definition 7.10 (*Context type algebra*)

$$\begin{array}{rcl} \bullet\{\!\{\mathsf{A}\}\!\} & \triangleq & \mathsf{A} \\ (\mathsf{C} \times \mathsf{A}')\{\!\{\mathsf{A}\}\!\} & \triangleq & \mathsf{C}\{\!\{\mathsf{A}\}\!\} \times \mathsf{A}' \\ (\mathsf{A}' \times \mathsf{C})\{\!\{\mathsf{A}\}\!\} & \triangleq & \mathsf{A}' \times \mathsf{C}\{\!\{\mathsf{A}\}\!\} \\ (\mathsf{A}' \to \mathsf{C})\{\!\{\mathsf{A}\}\!\} & \triangleq & \mathsf{A}' \to \mathsf{C}\{\!\{\mathsf{A}\}\!\} \end{array}$$

Definition 7.11 (*Iterated context types*)

$$\begin{array}{rcl} \mathsf{A}\langle\bullet\rangle & \triangleq & \bullet \\ \mathsf{A}\langle\mathsf{B} \to \mathsf{D}\rangle & \triangleq & \mathsf{A}\langle\mathsf{B}\rangle \to \mathsf{A}\langle\mathsf{D}\rangle \\ \mathsf{A}\langle\mathsf{D} \times \mathsf{B}\rangle & \triangleq & \mathsf{A}\langle\mathsf{D}\rangle \times \mathsf{A}\langle\mathsf{B}\rangle \\ \mathsf{A}\langle\mathsf{B} \times \mathsf{D}\rangle & \triangleq & \mathsf{A}\langle\mathsf{B}\rangle \times \mathsf{A}\langle\mathsf{D}\rangle \end{array}$$

Definition 7.12 (*Context typing rules*)

$$\frac{}{\Delta; \Gamma \vdash_\tau \bullet : \bullet}\ \mathsf{ctp\_bullet} \qquad \frac{\Delta; \Gamma \vdash_\tau \mathsf{E} : \mathsf{C}}{\Delta; \Gamma \vdash_\tau \mathsf{E}\{\!\{\mathbf{fst}\ \bullet\}\!\} : \mathsf{C} \times \mathsf{A}}\ \mathsf{ctp\_fst}$$

$$\frac{\Delta; \Gamma \vdash_\tau \mathsf{E} : \mathsf{C}}{\Delta; \Gamma \vdash_\tau \mathsf{E}\{\!\{\mathbf{snd}\ \bullet\}\!\} : \mathsf{A} \times \mathsf{C}}\ \mathsf{ctp\_snd}$$

$$\frac{\Delta; \Gamma \vdash_\tau \mathsf{E} : \mathsf{C} \quad \Delta \vdash \mathsf{A} \rhd_\tau \tau_\mathsf{A} \quad \Delta; \Gamma \overset{\mathbb{F}^\omega}{\vdash} e : \tau_\mathsf{A}}{\Delta; \Gamma \vdash_\tau \mathsf{E}\{\!\{\bullet\ e\}\!\} : \mathsf{A} \to \mathsf{C}}\ \mathsf{ctp\_app}$$

Finally, we define a formalism to describe the result of iteration over an iteration context.

Definition 7.13 (*Iterated contexts*)

$$\frac{\Delta;\Gamma \models^{\mathbb{E}\omega} e_\Theta : \Sigma^*\tau \to \tau}{\Delta;\Gamma \vdash \bullet \blacktriangleright^{\tau}_{e_\Theta} \bullet}\ \textsf{itc\_bullet} \qquad \frac{\Delta;\Gamma \vdash E \blacktriangleright^{\tau}_{e_\Theta} E'}{\Delta;\Gamma \vdash E\{\!\{\mathbf{fst}\,\bullet\}\!\} \blacktriangleright^{\tau}_{e_\Theta} E'\{\!\{\mathbf{fst}\,\bullet\}\!\}}\ \textsf{itc\_fst}$$

$$\frac{\Delta;\Gamma \vdash E \blacktriangleright^{\tau}_{e_\Theta} E'}{\Delta;\Gamma \vdash E\{\!\{\mathbf{snd}\,\bullet\}\!\} \blacktriangleright^{\tau}_{e_\Theta} E'\{\!\{\mathbf{snd}\,\bullet\}\!\}}\ \textsf{itc\_snd}$$

$$\frac{\Delta;\Gamma \vdash E \blacktriangleright^{\tau}_{e_\Theta} E' \quad \Delta;\Gamma \models^{\mathbb{E}\omega} e : B^*(\mathsf{Rec}\ \Sigma^*\ \tau)}{\Delta;\Gamma \vdash E\{\!\{\bullet\ e\}\!\} \blacktriangleright^{\tau}_{e_\Theta} E'\{\!\{\bullet\ (\mathbf{openiter}\{\!\{B^*\}\!\}[\tau]\ e_\Theta\ e)\}\!\}}\ \textsf{itc\_app}$$

The following two lemmas lift congruence to iteration contexts.

Lemma 7.14 (*Congruence under iteration contexts*)
If $\Delta;\Gamma \vdash_{\tau'} E : C$ and $\Delta;\Gamma \models^{\mathbb{E}\omega} e_1 \equiv_{\beta\eta} e_2 : C\{\!\{B\}\!\}^*\tau$ then $\Delta;\Gamma \models^{\mathbb{E}\omega} E\{\!\{e_1\}\!\} \equiv_{\beta\eta} E\{\!\{e_2\}\!\} : B^*\tau$.

Proof
By induction over the structure of $\Delta;\Gamma \vdash_\tau E : C$. □

Lemma 7.15 (*Congruence under iterated contexts*)
If $\Delta;\Gamma \vdash_\tau E : A\langle D\rangle$
and $\Delta;\Gamma \models^{\mathbb{E}\omega} e_1 \equiv_{\beta\eta} e_2 : D\{\!\{B\}\!\}^*\tau_A$
and $\Delta \vdash A \rhd_\tau \tau_A$
then $\Delta;\Gamma \models^{\mathbb{E}\omega} E\{\!\{e_1\}\!\} \equiv_{\beta\eta} E\{\!\{e_2\}\!\} : B^*\tau_A$.

Proof
By induction over the structure of $\Delta;\Gamma \vdash_\tau E : A\langle B\rangle$. □

One more lemma is required to prove our generalization of Lemma 7.7 (iteration and atomic applications) to all atomic forms. Because variables are one of the possible atomic forms, and elimination substitutions are used to replace them with an occurrence of **uniter**, it is necessary to prove that **openiter** will cancel with **uniter**. For simple terms, Lemma B.2 (properties of iteration, part 1) suffices. However, this is not sufficient if **openiter** and **uniter** are separated by an intervening sequence of projections and applications, as is the case when considering terms embedded in iteration contexts. Therefore, we lift the right inverse property to apply to this situation.

Lemma 7.16 (*Lifting right inverse property to iteration contexts*)
If $\Delta;\Gamma \models^{\mathbb{E}\omega} e_\Theta : \Sigma^*\tau_A \to \tau_A$
and $\Delta;\Gamma \vdash_\tau E : D$
then for all $\Delta;\Gamma \models^{\mathbb{E}\omega} e' : D\{\!\{B\}\!\}^*\tau_A$,
$\Delta;\Gamma \models^{\mathbb{E}\omega} \mathbf{openiter}\{\!\{B^*\}\!\}[\tau_A]\ e_\Theta(E\{\!\{\mathbf{uniter}\{\!\{D\{\!\{B\}\!\}^*\}\!\}[\tau_A]\ e_\Theta\ e'\}\!\}) \equiv_{\beta\eta}$
$E'\{\!\{e'\}\!\} : B^*\tau_A$
where $\Delta;\Gamma \vdash E \blacktriangleright^{\tau_A}_{e_\Theta} E'$.

Proof
By induction over the context typing derivation, $\Delta; \Gamma \vdash_\tau \mathsf{E} : \mathsf{D}$.

All cases begin by assuming an arbitrary $\Delta; \Gamma \overset{\mathbb{E}^\omega}{\vDash} e' : \mathsf{D}\{\!|\mathsf{B}|\!\}^*\tau_A$, and then make use of Lemma B.2 (properties of iteration) and Lemma B.1 (substitution for congruences, part two). Additionally the cases for ctp_fst, ctp_snd, and ctp_app all use Lemma 7.14 (congruence for iteration contexts). Finally, the case for ctp_app requires Lemma A.10 (commutativity for parameterization and type encoding). □

Lemma 7.17 (*Iteration and atomic forms*)
If $\Psi \overset{\text{SDP}}{\vDash} \mathsf{V} \downarrow \mathsf{B}_2$
and $\Delta \vdash \Omega \rhd \Gamma_1$
and $\Delta \vdash \Upsilon \rhd_\tau \Gamma_2$
and $\Delta \vdash \mathsf{A} \rhd_\tau \tau_A$
and $\Delta; \Gamma_1 \uplus \Gamma_2 \overset{\mathbb{E}^\omega}{\vDash} e_\Theta : \Sigma^*\tau_A \to \tau_A$
and $\Omega; \Upsilon \overset{\text{SDP}}{\vDash} \Theta : \mathsf{A}\langle\Psi{:}\Sigma\rangle$
and $\Delta; \varnothing \vdash \mathsf{V} \rhd_{\tau_A} e$
and $\Delta; \Psi; \Theta; e_\Theta \blacktriangleright^{\tau_A}_{\tau} \mathsf{S}$
then for all $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash_\tau \mathsf{E} : \mathsf{D}$
where $\mathsf{B}_2 = \mathsf{D}\{\!|\mathsf{B}|\!\}$,
and $\Delta; \Gamma_1 \uplus \Gamma_2 \overset{\mathbb{E}^\omega}{\vDash} \mathbf{openiter}\{\!|\mathsf{B}^*|\!\}[\tau_A]\ e_\Theta\ \mathsf{E}\{\!|\mathsf{S}(e)|\!\} \equiv_{\beta\eta}$
$\mathsf{E}'\{\!|\mathbf{openiter}\{\!|\mathsf{D}\{\!|\mathsf{B}|\!\}^*|\!\}[\tau_A]\ e_\Theta\ \mathsf{S}(e)|\!\} : \mathsf{B}^*\tau_A$
where $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \mathsf{E} \blacktriangleright^{\tau_A}_{e_\Theta} \mathsf{E}'$.

Proof
By induction on $\Psi \overset{\text{SDP}}{\vDash} \mathsf{V} \downarrow \mathsf{B}_2$.

All cases begin by assuming an arbitrary $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash_\tau \mathsf{E} : \mathsf{D}$ where $\mathsf{B}_2 = \mathsf{D}\{\!|\mathsf{B}|\!\}$, and then proceed by uses of inversion and congruence.

- The case for at_var is quite involved and requires Lemma B.11 (substitution elimination), Lemma B.10 (typing for elimination) Lemma A.6 (type encoding total and decidable), Lemma A.10 (commutativity for parameterization and type encoding) Theorem 6.2 (static correctness, forward direction), Lemma 7.16 (lifting right inverse), Lemma B.5 (iterated context typing), Lemma B.2 (properties of iteration), Lemma B.1 (substitution for congruences, part two), and Lemma 7.15 (congruence under iterated contexts).
- The case for at_cons makes use of Lemma A.10 (commutativity for parameterization and type encoding), Lemma B.2 (properties of iteration), and Lemma B.1 (substitution for congruences, part two).
- The case for at_app requires the proof of typing of atomic and canonical forms (Schürmann *et al.*, 2001), Lemma A.6 (type encoding total and decidable), Lemma B.12 (static correctness with substitution), Lemma A.10 (commutativity for parameterization and type encoding) Lemma B.5 (iterated context typing), and Lemma 7.14 (congruence under iteration contexts).
- The cases for at_fst and at_snd need Lemma B.5 (iterated context typing), Lemma B.2 (properties of iteration), and Lemma 7.14 (congruence of iterated contexts).

□

8 Future work

Although we have shown a very close connection between SDP and its encoding in $\mathbb{F}_\omega$, 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 $\mathbb{F}_\omega$ terms of type τ are equivalent to the encoding of some SDP term. In other words, there is no extra "junk" of type τ. One of the primary difficulties in such a proof would be that there are clearly terms in $\mathbb{F}_\omega$ inhabiting τ that have no equivalent in SDP. For example, any term that contains type abstraction and type application. Therefore, it is not possible to prove a theorem as strong as "all $\mathbb{F}_\omega$ terms of type τ are equivalent to the encoding of some SDP term". We believe that constraining the theorem to "all $\mathbb{F}_\omega$ terms of type τ are $\beta\eta$-equivalent to a canonical form that is the encoding of some SDP term". From there, we expect the proof would follow by induction on canonical terms of type τ.

Additionally it would be worthwhile to show the adequacy of encoding of the untyped λ-calculus we presented informally in the first part of the paper. That encoding uses first-order quantification and discards the current world. This simpler representation allows the encoding of some terms that are rejected by the SDP calculus to type check (for example, $\lambda x : \Box b.\ \mathbf{box}\ x$), but we conjecture that it is still an adequate encoding of the untyped λ-calculus. Again, the proof of adequacy would need to be defined in terms of Haskell or $\mathbb{F}_\omega$ canonical forms.

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 data types 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 data types with embedded functions.) From a logical point of view, this restriction makes sense. Combining anamorphisms and catamorphisms (even for data types without embedded functions) leads to general recursion.

9 Related work

The technique we present, using polymorphism to enforce parametricity, has appeared under various guises in the literature. For example, Shao et al. (2000) use this technique, at the type instead of term level, 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. (2003) 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.

One popular alternative to HOAS is what is called *weak higher-order abstract syntax*. The idea behind weak HOAS, is to abstract over names rather than terms of the object language. As a consequence of abstracting over names, weak HOAS requires an explicit constructor for representing variables, and substitution must be implemented as a metalanguage function. The simplest version of weak HOAS uses an explicit type for names. In such a case, representing the untyped λ-calculus in Haskell using weak HOAS might be done using the following data type.

```
type Name = String
data Exp = Var Name
         | Lam (Name -> Exp)
         | App Exp Exp
```

However, this version of weak HOAS still suffers from the problem of exotic terms because the embedded function may not be parametric in names. Consider the following example

```
badexp :: Exp
badexp = Lam (\x -> if (x == "y") then
                      App (Var x) (Var x)
                    else
                      Var x)
```

If the type of names is kept abstract it is possible to prevent this particular exotic term. This is the approach taken by Honsell, Miculan, and Scagnetto in their Theory of Contexts (2001), and Despeyroux, Felty, and Hirshowitz in their study of encoding higher-order abstract in Coq (1995).

However, if there is no way to compare names programming becomes cumbersome in practice. Therefore it becomes necessary for the `Exp` data type to be packaged with functions for substituting for abstract names, calculating free variables, etc. However, at that point it becomes straightforward to write exotic terms again:

```
e1 :: Exp -- Some other expressions
e2 :: Exp
badexp' :: Exp
badexp' = Lam (\x -> if (freevar x e1) then e1 else e2)
```

Despeyroux, Felty, and Hirshowitz resolve this problem by introducing an additional validity predicate for their expressions.

The benefit of is using weak HOAS is that it is much easier to developing reasoning principles because there is no need for negative occurrences of the data type. In some cases weak HOAS is an ideal choice. For example, in languages like the π-calculus, there is no abstraction over terms of the object language, just names. Consequently, Despeyroux was able to use weak HOAS to provide an elegant mechanical formalization of the π-calculus (Despeyroux, 2000).

A weakness of our approach and the SDP calculus is that it is not possible explicitly reason about variables in the object-language. There has been a significant amount of research on manipulating "open" terms of an object-language. Dale Miller developed a small language called ML_λ (Miller, 1990) 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 (Pitts & Gabbay, 2000) that allows for the manipulation and abstraction of fresh "names". Nanevski (2002) 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.

Similar to fresh names as developed by Pitts and Gabbay, and name generation in the π-calculus, Miller and Tiu have recently developed a logic, $FO\lambda^{\Delta\nabla}$ (Miller & Tiu, 2005), with a built in abstraction operator ∇. The ∇ quantifier abstracts over variables that are guaranteed to be distinct, even from universally quantified terms. Gabbay and Cheney noted that the ∇ quantifier, like the $\mathcal{N}$ of fresh logic, is self-dual and commutes with propositional connectives, but that it does not satisfy the same tautologies (Gabbay & Cheney, 2004). Regardless, they were able show that $FO\lambda^{\nabla}$ can be soundly embedded into fresh logic.

The Delphin Project (Schürmann *et al.*, 2002) by Schürmann et al. is aimed at developing a functional language for manipulating data types 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 latest core-calculus proposed for Delphin is the ∇-calculus (Schürmann *et al.*, 2004). Despite seeming similarities with the ∇ of Miller and Tiu, the ∇ quantifier in this language is used to indicate a non-deterministic matching of a free variable in the environment. The ∇-calculus also makes use of a separate operator for introducing new parameters and a modal type system to ensure variables are restricted to appropriate scopes.

The Hybrid (Ambler *et al.*, 2002) 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, for example, Honsell and Miculan's formalization of dynamic logic within a logical framework (Honsell & Miculan, 1995). Only recently has the encoding of modal type systems been explored. Acar et al. (2002) use modal types in a functional language that provides control over the use of memoization, and implement it as a library in Standard ML. 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 (2001) 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 (Washburn, 2001).

10 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 $\mathbb{F}_\omega$ 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 reachable worlds and polymorphism over world transformers makes that quantification explicit.

Acknowledgments

We thank Margaret DeLap, Eijiro Sumi, Stephen Tse, Stephan Zdancewic, and the anonymous ICFP reviewers for providing us with feedback concerning this paper. The first author would like to thank Frank Pfenning for instilling within him an attention to detail.

References

Acar, Umut, Blelloch, Guy, & Harper, Robert. (2002). Selective memoization. *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New Orleans, LA: ACM Press.

Ambler, Simon, Crole, Roy L., & Momigliano, Alberto. (2002). Combining higher order abstract syntax with tactical theorem proving and (co)induction. *15th International Conference on Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science, vol. 2410. Hampton, VA: Springer.

Bekić, Hans. (1984). Definable operation in general algebras, and the theory of automata and flowcharts. *Programming languages and their definition*. Springer-Verlag. LNCS vol 177.

Böhm, Corrado, & Berarducci, Allessandro. (1985). Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, **39**, 135–154.

Church, Alonzo. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, **5**, 56–68.

Clarke, Dave, Hinze, Ralf, Jeuring, Johan, Löh, Andres, & de Wit, Jan. (2001). *The Generic Haskell user's guide*. Tech. rept. UU-CS-2001-26. Utrecht University.

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

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

Despeyroux, Joelle. (2000). A higher-order specification of the π–calculus. *IFIP International Theoretical Computer Science*. Sendai, Japan: Springer.

Despeyroux, Joëlle, & Leleu, Pierre. (2001). Recursion over objects of functional type. *Mathematical Structures in Computer Science*, **11**, 555–572.

Despeyroux, Joëlle, Felty, Amy P., & Hirschowitz, André. (1995). Higher-order abstract syntax in Coq. *Second International Conference on Typed Lambda Calculi and Applications*. London, UK: Springer-Verlag.

Fegaras, Leonidas, & Sheard, Tim. (1996). Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersburg Beach, FL: ACM Press.

Gabbay, Murdoch J., & Cheney, James. (2004). A sequent calculus for nominal logic. *19th IEEE Symposium on Logic in Computer Science*.

Girard, Jean-Yves. (1971). 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. Fenstad, J. E. (ed), *Second Scandinavian Logic Symposium*. North-Holland Publishing Co.

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

Honsell, Furio, & Miculan, Marino. 1995 (June). A natural deduction approach to dynamic logic. Berardi, Coppo (ed), *TYPES 1995*. Published in LNCS 1158, 1996.

Honsell, Furio, Miculan, Marino, & Scagnetto, Ivan. (2001). An axiomatic approach to metareasoning on nominal algebras in HOAS. *Lecture Notes in Computer Sciece*, **2076**.

Johann, Patricia. (2002). A generalization of short-cut futions and its correctness proof. *Higher-order and Symbolic Computation*, **15**, 273–300.

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

Jones, Mark P. (2000). Type classes with functional dependencies. *Ninth European Symposium on Programming*. LNCS, no. 1782. Berlin, Germany: Springer-Verlag.

Kripke, Saul A. (1959). A completeness theorem in modal logic. *Journal of Symbolic Logic*, **24**, 1–15.

Leszczyłowski, Jacek. (2005). A theory on resolving equations in the space of languages. *Bulletin of the Polish Academy of the Sciences*, **19**(Oct.), 967–970.

Meijer, Erik, & Hutton, Graham. (1995). Bananas in space: Extending fold and unfold to exponential types. *Conference on functional programming languages and computer architecture*. La Jolla, CA: ACM Press.

Meijer, Erik, Fokkinga, Maarten M., & Paterson, Ross. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Conference on functional programming languages and computer architecture*. Cambridge, MA: Springer-Verlag.

Miller, Dale. 1990 (May). An extension to ML to handle bound variables in data structures: Preliminary report. *Proceedings of the Logical Frameworks BRA Workshop*.

Miller, Dale, & Tiu, Alwen. (2005). A proof theory for generic judgments. *ACM Transactions Computational Logic*, **6**(4), 749–783.

Nanevski, Aleksandar. (2002). Meta-programming with names and necessity. *Pages 206–217 of: Seventh ACM SIGPLAN International Conference on Functional Programming*. ACM Press.

Peyton Jones, Simon (ed). (2003). *Haskell 98 language and libraries: The revised report*. Cambridge University Press.

Peyton Jones, Simon, Vytiniotis, Dimitrios, Weirich, Stephanie, & Shields, Mark. (2005). Practical type inference for arbitrary-rank types. *Journal of Functional Programming*. Accepted for publication to the Journal of Functional Programming.

Pfenning, Frank, & Davies, Rowan. (2001). A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, **11**(4), 511–540.

Pfenning, Frank, & Elliott, Conal. (1988). Higher-order abstract syntax. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, GA: ACM Press.

Pfenning, Frank, & Schürmann, Carsten. (1999). System description: Twelf—a meta-logical framework for deductive systems. Ganzinger, H. (ed), *16th International Conference on Automated Deduction*. Trento, Italy: Springer-Verlag.

Pitts, Andrew M., & Gabbay, Murdoch J. (2000). A metalanguage for programming with bound names modulo renaming. *Mathematics of Program Construction*. Port de Lima, Portugal: Springer-Verlag.

Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Information processing '83*. North-Holland. Proceedings of the IFIP 9th World Computer Congress.

Schürmann, Carsten, Despeyroux, Joëlle, & Pfenning, Frank. (2001). Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, **266**(1–2), 1–58.

Schürmann, Carsten, Fontana, Richard, & Liao, Yu. (2002). *Delphin: Functional programming with deductive systems*. Available at `http://cs-www.cs.yale.edu/homes/carsten/`.

Schürmann, Carsten, Poswolsky, Adam, & Sarnat, Jeffrey. 2004 (Nov.). *The ∇-calculus: Functional programming with higher-order encodings*. Tech. rept. YALEU/DCS/TR-1272. Yale University.

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

Trifonov, Valery, Saha, Bratin, & Shao, Zhong. (2000). Fully reflexive intensional type analysis. *Fifth ACM SIGPLAN International Conference on Functional Programming*. Montreal, Canada: ACM Press. Extended version is YALEU/DCS/TR-1194.

Wadler, Philip. (1989). Theorems for free! *Conference on functional programming languages and computer architecture*. London, United Kingdom: ACM Press.

Washburn, Geoffrey. (2001). *Modal typing for specifying run-time code generation*. Available from `http://www.cis.upenn.edu/~geoffw/research/`.

Washburn, Geoffrey, & Weirich, Stephanie. (2003). Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *Eighth ACM SIGPLAN International Conference on Functional Programming*. Uppsala, Sweden: ACM Press, for ACM SIGPLAN.

Weirich, Stephanie. (2006). Type-safe run-time polytypic programming. *Journal of Functional Programming*. To appear.

Xi, Hongwei, Chen, Chiyan, & Chen, Gang. (2003). Guarded recursive datatype constructors. *30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New Orleans, LA, USA: ACM Press.

A Static correctness

Lemma A.1 (*Weak head types are well-formed types*)
If $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \restriction \star$ then $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau : \star$.

Proof
By trivial induction over the structure of $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \restriction \star$. □

Lemma A.2 (*Well-formedness of type encoding*)
If $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau : \star$ and $\Delta \vdash A \rhd_\tau \tau_A$ then $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_A : \star$.

Proof
Follows directly from Lemma 6.5 and Lemma A.1. □

Lemma A.3 (*Inversion on typing derivations*)

1. If $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} x : \tau$ then $\Gamma(x) = \tau'$ where $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \tau' : \star$.
2. If $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} e_1 e_2 : \tau$ then $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} e_1 : \tau_1 \to \tau_2$ and $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} e_2 : \tau_1$ where $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \tau_2 : \star$.
3. If $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} \lambda x : \tau_1 . e : \tau$ then $\Delta; \Gamma \uplus \{x : \tau_1\} \stackrel{\mathbb{F}\omega}{\vdash} e : \tau'$ where $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \tau_1 \to \tau' : \star$.
4. If $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} \langle\rangle : \tau$ then $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \forall \alpha : \star . 1(\alpha) : \star$.
5. If $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} \Lambda \alpha : \kappa . e : \tau$ then $\Delta \uplus \{\alpha : \kappa\}; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} e : \tau'$ where $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \forall \alpha : \kappa . \tau' : \star$.
6. If $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} e[\tau_1] : \tau$ then $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} e : \forall \alpha : \kappa . \tau'$ where $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 : \kappa$ and $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \tau'\{\tau_1'/\alpha\} : \star$ and $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_1' : \kappa$.
7. If $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} \mathbf{case}\, e\, \mathbf{of}\, \mathbf{inj}_{l_1}\, x_1\, \mathbf{in}\, e_1 \ \ldots\ \mathbf{inj}_{l_n}\, x_n\, \mathbf{in}\, e_n : \tau$ then $\Delta; \Gamma \stackrel{\mathbb{F}\omega}{\vdash} e : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$ and $\Delta; \Gamma \uplus \{x_i : \tau_i\} \stackrel{\mathbb{F}\omega}{\vdash} e_i : \tau'$ for each e_i where $\Delta \stackrel{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \tau' : \star$.

Proof
By straightforward induction over the number of uses of `tp_eq` used before the final derivation step. □

Lemma A.4 (*Inversion for type congruences*)

1. If $\Delta \overset{\mathbb{F}\omega}{\vdash} 1(\tau) \equiv_{\beta\eta} \tau' : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau' \upharpoonright \star$ then $\tau' = 1(\tau'')$ where $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \tau'' : \star$.
2. If $\Delta \overset{\mathbb{F}\omega}{\vdash} \mathsf{Rec}\ \Sigma^*\ \tau \equiv_{\beta\eta} \tau' : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau' \upharpoonright \star$ then $\tau' = \tau_1 \to \tau_2$ where $\Delta \overset{\mathbb{F}\omega}{\vdash} \Sigma^*\tau \to \tau \equiv_{\beta\eta} \tau_1 : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \tau_2 : \star$.
3. If $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1 \to \tau_2 \equiv_{\beta\eta} \tau' : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau' \upharpoonright \star$ then $\tau' = \tau_1' \to \tau_2'$ where $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_1' : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_2 \equiv_{\beta\eta} \tau_2' : \star$.
4. If $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1 \times \tau_2 \equiv_{\beta\eta} \tau' : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau' \upharpoonright \star$ then $\tau' = \tau_1' \times \tau_2'$ where $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_1' : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_2 \equiv_{\beta\eta} \tau_2' : \star$.
5. If $\Delta \overset{\mathbb{F}\omega}{\vdash} \forall\alpha : \star \to \star.\tau \equiv_{\beta\eta} \tau' : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau' \upharpoonright \star$ then $\tau' = \forall\alpha : \star \to \star.\tau''$ where $\Delta \uplus \{\alpha : \star \to \star\} \overset{\mathbb{F}\omega}{\vdash} \tau \equiv_{\beta\eta} \tau'' : \star$.

Proof
By induction over the structure of the type congruences. □

Lemma A.5 (*Inversion for type encoding*)

1. If $\Delta \vdash A \rhd_\tau \tau_A$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_A \equiv_{\beta\eta} \mathsf{Rec}\ \Sigma^*\ \tau : \star$ then $A = \mathsf{b}$.
2. If $\Delta \vdash A \rhd_\tau \tau_A$ and $\Delta \vdash A_1 \rhd_\tau \tau_1$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_A \equiv_{\beta\eta} \tau_1 \to \tau_2 : \star$ then $\Delta \vdash A_1' \rhd_\tau \tau_1'$ and $\Delta \vdash A_2 \rhd_\tau \tau_2'$ where $A = A_1' \to A_2$ and $\tau_A = \tau_1' \to \tau_2'$.
3. If $\Delta \vdash A \rhd_\tau \tau_A$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_A \equiv_{\beta\eta} \forall\alpha : \star \to \star.\tau_A' : \star$ then $\Delta \uplus \{\alpha : \star \to \star\} \vdash A' \rhd_{\alpha\tau} \tau_A''$ where $\Delta \uplus \{\alpha : \star \to \star\} \overset{\mathbb{F}\omega}{\vdash} \tau_A' \equiv_{\beta\eta} \tau_A'' : \star$ and $A = \Box A'$ and $\tau_A = \forall\alpha : \star \to \star.\tau_A''$.
4. If $\Delta \vdash A \rhd_\tau \tau_A$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_A \equiv_{\beta\eta} 1(\tau') : \star$ then $A = 1$.
5. If $\Delta \vdash A \rhd_\tau \tau_A$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_A \equiv_{\beta\eta} \tau_1 \times \tau_2 : \star$ then $\Delta \vdash A_1 \rhd_\tau \tau_1'$ where $\Delta \vdash A_2 \rhd_\tau \tau_2'$ where $A = A_1 \times A_2$ and $\tau_A = \tau_1' \times \tau_2'$.

Proof
By inversion over the structure of the type congruence. For Part 1:

- By Lemma 6.5 (type encodings are weak head normal) on $\Delta \vdash A \rhd_\tau \tau_A$ we know that $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_A \upharpoonright \star$. Using Lemma A.4 (inversion) on $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_A \equiv_{\beta\eta} \mathsf{Rec}\ \Sigma^*\ \tau : \star$ we know that $\tau_A = \tau_1 \to \tau_2$ where $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \Sigma^*\tau \to \tau : \star$. Given that $\Delta \vdash A \rhd_\tau \tau_1 \to \tau_2$, either $A = \mathsf{b}$ or $A = A_1 \to A_2$ for some A_1, A_2.
- Assume that $A = A_1 \to A_2$. Then by inversion on $\Delta \vdash A_1 \to A_2 \rhd_\tau \tau_1 \to \tau_2$ we have that $\Delta \vdash A_1 \rhd_\tau \tau_1$. Using Lemma 6.5 again on $\Delta \vdash A_1 \rhd_\tau \tau_1$ we know that $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1 \upharpoonright \star$. Again by Lemma A.4 on $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \Sigma^*\tau \to \tau : \star$ we have that $\tau_1 = \tau_1' \to \tau_1''$ where $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1' \equiv_{\beta\eta} \Sigma^*\tau : \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1'' \equiv_{\beta\eta} \tau : \star$. As before $A_1 = \mathsf{b}$ or $A_1 = A_1' \to A_1''$ for some A_1', A_1''.
- Assume $A_1 = \mathsf{b}$. Then $\tau_1' = \Sigma^*\tau \to \tau$. However, $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1' \equiv_{\beta\eta} \Sigma^*\tau : \star$. $\Sigma^*\tau \to \tau$ and $\Sigma^*\tau$ have different head normal forms therefore it cannot be the case that $A_1 = \mathsf{b}$.

- Therefore $A_1 = A_1' \to A_1''$. By inversion on $\Delta \vdash A_1' \to A_1'' \rhd_\tau \tau_1' \to \tau_1''$ we have that $\Delta \vdash A_1' \rhd_\tau \tau_1'$. However, we know that $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1' \equiv_{\beta\eta} \Sigma^*\tau : \star$. There are no types in the image of the encoding where the head constructor is equivalent to a variant. So our assumption that $A = A_1 \to A_2$ must be false, and $A = b$.

For Part 2:

- By Lemma 6.5 (type encodings are weak head normal) on $\Delta \vdash A \rhd_\tau \tau_A$ we know that $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_A \upharpoonright \star$. Using Lemma A.4 (inversion) on $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_A \equiv_{\beta\eta} \tau_1 \to \tau_2 :$ we know that $\tau_A = \tau_1' \to \tau_2'$ where $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_1' : \star$. Given that $\Delta \vdash A \rhd_\tau \tau_1' \to \tau_2'$, either $A = b$ or $A = A_1' \to A_2'$ for some A_1', A_2'.
- Assume $A = b$. Then $\tau_1' = \Sigma^*\tau \to \tau$. However, $\Delta \vdash A_1 \rhd_\tau \tau_1$ and $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_1' : \star$. There are no types in the image of the encoding where the head constructor is equivalent to a variant. So our assumption that $A = b$ must false.
- Therefore, $A = A_1' \to A_2'$ for some A_1', A_2'. By inversion on $\Delta \vdash A_1' \to A_2' \rhd_\tau \tau_1' \to \tau_2'$ we can conclude that $\Delta \vdash A_1' \rhd_\tau \tau_1'$ and $\Delta \vdash A_2' \rhd_\tau \tau_2'$.

□

Lemma A.6 (*Type encoding is total and decidable*)
Given a type, A, in the source calculus and a τ in $\mathbb{F}_\omega$ we can construct $\Delta \vdash A \rhd_\tau \tau_A$.

Proof
By straightforward induction over the structure of A. □

Lemma A.7 (*Uniqueness of type encoding*)

1. If $\Delta \vdash A \rhd_\tau \tau_A$ and $\Delta \vdash A \rhd_\tau \tau_A'$ then $\tau_A = \tau_A'$.
2. If $\Delta \vdash A \rhd_\tau \tau$ and $\Delta \vdash A' \rhd_\tau \tau$ then $A = A'$.

Proof
Both properties follow by straightforward simultaneous induction on the type encoding derivations. □

Lemma A.8 (*Type encoding under congruent worlds*)
If $\Delta \vdash A \rhd_{\tau_1} \tau_A$ and $\Delta \vdash A \rhd_{\tau_2} \tau_A'$ where $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_2 : \star$ then
$\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau_A \equiv_{\beta\eta} \tau_A' : \star$

Proof
By straightforward simultaneous induction on the type encoding derivations. □

Lemma A.9 (*World substitution for type encoding*)
If $\Delta \uplus \{\alpha : \star \to \star\} \vdash A \rhd_{\alpha\tau'} \tau_A$ and $\Delta \overset{\mathbb{F}_\omega}{\vdash} \tau : \star \to \star$ then $\Delta \vdash A \rhd_{\tau\tau'} \tau_A\{\tau/\alpha\}$.

Proof
By straightforward induction over the structure of $\Delta \uplus \{\alpha : \star \to \star\} \vdash A \rhd_{\alpha\tau'} \tau_A$.
□

Lemma A.10 (*Commutativity for parameterization and type encoding*)
If $\Delta \vdash B \rhd_\tau \tau_B$ then $\tau_B = (\mathsf{Rec}\ \Sigma^*\ \tau)\langle B\rangle$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_B \equiv_{\beta\eta} B^*(\mathsf{Rec}\ \Sigma^*\ \tau) : \star$.

Proof
By straightforward induction over the structure of $\Delta \vdash B \rhd_\tau \tau_B$. □

Lemma A.11 (*Commutativity for iteration types and type encoding*)
If $\Delta \vdash A \rhd_\tau \tau_A$ then $\Delta \vdash A\langle B\rangle \rhd_\tau \tau_A\langle B\rangle$

Proof
By straightforward induction over the structure of $A\langle B\rangle$. □

Lemma A.12 (*Encoding produces well-formed environments*)
Assume $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau : \star$.

1. If $\Delta \vdash \Upsilon \rhd_\tau \Gamma_1$ then $\Delta \overset{\mathbb{F}\omega}{\vdash} \Gamma_1$.
2. If $\Delta \vdash \Omega \rhd \Gamma_2$ then $\Delta \overset{\mathbb{F}\omega}{\vdash} \Gamma_2$.

Proof
Straightforward from the definitions and Lemma A.2. □

Lemma A.13 (*Type containment*)
Given a derivation $\Delta \vdash A \rhd_\tau \tau_A$ we know that $\mathrm{FTV}(\tau) = \mathrm{FTV}(\tau_A)$.

Proof
By straightforward induction over the structure of $\Delta \vdash A \rhd_\tau \tau_A$. □

B Dynamic correctness

Lemma B.1 (*Substitution for congruences*)

1. If $\Delta;\Gamma \overset{\mathbb{F}\omega}{\vdash} e_1 \equiv_{\beta\eta} e_2 : \tau'$ and $\Delta;\Gamma \uplus \{x : \tau'\} \overset{\mathbb{F}\omega}{\vdash} e_3 \equiv_{\beta\eta} e_4 : \tau$ then $\Delta;\Gamma \overset{\mathbb{F}\omega}{\vdash} e_3\{e_1/x\} \equiv_{\beta\eta} e_4\{e_2/x\} : \tau$.
2. If $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_1 \equiv_{\beta\eta} \tau_2 : \kappa'$ and $\Delta \uplus \{\alpha : \kappa'\} \overset{\mathbb{F}\omega}{\vdash} \tau_3 \equiv_{\beta\eta} \tau_4 : \kappa$ then $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau_3\{\tau_1/\alpha\} \equiv_{\beta\eta} \tau_4\{\tau_2/\alpha\} : \kappa$

Proof
By straightforward induction over the structure of $\Delta;\Gamma \uplus \{x : \tau'\} \overset{\mathbb{F}\omega}{\vdash} e_3 \equiv_{\beta\eta} e_4 : \tau$ and $\Delta \uplus \{\alpha : \kappa'\} \overset{\mathbb{F}\omega}{\vdash} \tau_3 \equiv_{\beta\eta} \tau_4 : \kappa$ respectively. □

Lemma B.2 (*Properties of* **openiter** *and* **uniter**)
Assuming $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau : \star \to \star$ and $\Delta \overset{\mathbb{F}\omega}{\vdash} \tau' : \star$.

1. $\Delta;\{f : \Sigma^*\tau' \to \tau'\} \overset{\mathbb{F}\omega}{\vdash} (\mathbf{openiter}\{\!|\tau|\!\}[\tau']\ f) \circ (\mathbf{uniter}\{\!|\tau|\!\}[\tau']\ f) \equiv_{\beta\eta} \lambda x : \tau\tau'.x : \tau\tau' \to \tau\tau'$
2. $\Delta;\{f : \Sigma^*\tau' \to \tau', e : b^*(\mathsf{Rec}\ \Sigma^*\ \tau')\} \overset{\mathbb{F}\omega}{\vdash} \mathbf{openiter}\{\!|b^*|\!\}[\tau']\ f\ e \equiv_{\beta\eta} ef : \tau'$
3. $\Delta;\{f : \Sigma^*\tau' \to \tau', e : (B_1 \to B_2)^*(\mathsf{Rec}\ \Sigma^*\ \tau')\} \overset{\mathbb{F}\omega}{\vdash}$
   $\mathbf{openiter}\{\!|(B_1 \to B_2)^*|\!\}[\tau']\ f\ e \equiv_{\beta\eta}$
   $(\mathbf{openiter}\{\!|B_2^*|\!\}[\tau']\ f) \circ\ e\ \circ (\mathbf{uniter}\{\!|B_1^*|\!\}[\tau']\ f) : (B_1 \to B_2)^*\tau'$
4. $\Delta;\{f : \Sigma^*\tau' \to \tau', e : (B_1 \to B_2)^*\tau'\} \overset{\mathbb{F}\omega}{\vdash}$
   $\mathbf{uniter}\{\!|(B_1 \to B_2)^*|\!\}[\tau']\ f\ e \equiv_{\beta\eta}$
   $(\mathbf{uniter}\{\!|B_2^*|\!\}[\tau']\ f) \circ\ e\ \circ (\mathbf{openiter}\{\!|B_1^*|\!\}[\tau']\ f) : (B_1 \to B_2)^*(\mathsf{Rec}\ \Sigma^*$

5. $\Delta; \{f : \Sigma^*\tau' \to \tau', e : (B_1 \times B_2)^*(\mathsf{Rec}\ \Sigma^*\ \tau')\} \models^{\mathbb{E}\omega}$
   $\mathbf{openiter}\{\!|(B_1 \times B_2)^*|\!\}[\tau']\ f\ e \equiv_{\beta\eta}$
   $\langle \mathbf{openiter}\{\!|B_1^*|\!\}[\tau']\ f\ (\mathbf{fst}\, e), \mathbf{openiter}\{\!|B_2^*|\!\}[\tau']\ f\ (\mathbf{snd}\, e)\rangle : (B_1 \times B_2)^*\tau'$
6. $\Delta; \{f : \Sigma^*\tau' \to \tau', e : (B_1 \times B_2)^*\tau'\} \models^{\mathbb{E}\omega}$
   $\mathbf{uniter}\{\!|(B_1 \times B_2)^*|\!\}[\tau']\ f\ e \equiv_{\beta\eta}$
   $\langle \mathbf{uniter}\{\!|B_1^*|\!\}[\tau']\ f\ (\mathbf{fst}\, e), \mathbf{uniter}\{\!|B_2^*|\!\}[\tau']\ f\ (\mathbf{snd}\, e)\rangle : (B_1 \times B_2)^*(\mathsf{Rec}\ \Sigma^*\ \tau')$
7. $\Delta; \{f : \Sigma^*\tau' \to \tau', e : B_i^*(\mathsf{Rec}\ \Sigma^*\ \tau')\} \models^{\mathbb{E}\omega}$
   $\mathbf{openiter}\{\!|\Sigma^*|\!\}[\tau']\ f\ (\mathbf{inj}_{l_i}\ e\ \mathbf{of}\ \Sigma^*(\mathsf{Rec}\ \Sigma^*\ \tau')) \equiv_{\beta\eta}$
   $\mathbf{inj}_{l_i}\ (\mathbf{openiter}\{\!|B_i^*|\!\}[\tau']\ f\ e)\ \mathbf{of}\ \Sigma^*\tau' : \Sigma^*\tau'$
8. $\Delta; \{f : \Sigma^*\tau' \to \tau'\} \models^{\mathbb{E}\omega}$
   $\mathbf{openiter}\{\!|(B_i \to b)^*|\!\}[\tau']\ f\ (\lambda x : \tau_{B_i}.\mathbf{roll}[\tau](\mathbf{inj}_{\mathcal{L}(c_i)}\ x\ \mathbf{of}\ \Sigma^*(\mathsf{Rec}\ \Sigma^*\ \tau'))) \equiv_{\beta\eta}$
   $\lambda x : B_i^*\tau'.f(\mathbf{inj}_{\mathcal{L}(c_i)}\ x\ \mathbf{of}\ \Sigma^*\tau') : (B_i \to b)^*\tau'$

Proof
Property 1 is by straightforward induction on the structure B. The proofs of properties 2, 3, 4, 5, 6, 7, and 8 follow directly from the rules term of congruence and the definitions of **openiter**, **xmap** and **uniter**. □

Lemma B.3 (*Well-typed replacements are well-formed dynamic replacements*)
If $\Omega; \Upsilon \models^{SDP} \Theta : A\langle\Sigma\rangle$ then $\Omega; \Upsilon \models^{SDP} \Theta : A\langle\varnothing{:}\Sigma\rangle$.

Proof
Follows trivially from the definitions. □

Lemma B.4 (*Restricted dynamic replacements are Well-typed replacements*)
If $\Omega; \Upsilon \models^{SDP} \Theta : A\langle\Psi{:}\Sigma\rangle$ then $\Omega; \Upsilon \models^{SDP} \llcorner\Theta\lrcorner : A\langle\Sigma\rangle$.

Proof
Follows trivially from the definitions. □

Lemma B.5 (*Iterated context typing*)
If $\Delta; \Gamma \vdash_\tau E : D$ and $\Delta \vdash A \rhd_\tau \tau_A$ and $\Delta; \Gamma \vdash E \blacktriangleright_{e_\Theta}^{\tau_A} E'$ then $\Delta; \Gamma \vdash_\tau E' : A\langle D\rangle$.

Proof
By induction over the structure $\Delta; \Gamma \vdash_\tau E : D$. □

Lemma B.6 (*Substitution for encoding of regular term variables*)
If $\Delta; \Xi \vdash M_1 \rhd_\tau e_1$ and $\Delta; \Xi \vdash M_2 \rhd_\tau e_2$ and $\Delta; \Xi \vdash M_2\{M_1/x\} \rhd_\tau e$ where $x \notin \Xi$ then $e = e_2\{e_1/x\}$.

Proof
By straightforward induction over the structure of $\Delta; \Xi \vdash M_2 \rhd_\tau e_2$. □

Lemma B.7 (*Replacement and term encoding are total and decidable*)

1. If $\Omega; \Upsilon \models^{SDP} M : A$ and $\Delta \models^{\mathbb{E}\omega} \tau : \star$ we can construct $\Delta; \mathrm{dom}(\Omega) \vdash M \rhd_\tau e$.
2. If $\Omega; \Upsilon \models^{SDP} \Theta : A\langle\Sigma\rangle$ and $\Delta \models^{\mathbb{E}\omega} \tau : \star$ we can construct $\Delta; \mathrm{dom}(\Omega) \vdash \Theta \rhd_\tau^{\tau_A} e_\Theta$.

Proof
By mutual induction over the structure of $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} M : A$ and $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \Theta : A\langle\Sigma\rangle$, and use of Lemma A.6 (type encoding is total and decidable). □

Lemma B.8 (*World substitution for term encoding*)
If $\Delta \uplus \{\alpha : \star \to \star\}; \Xi \vdash M \rhd_{\alpha\tau'} e$ then $\Delta; \Xi \vdash M \rhd_\tau e\{\lambda\beta : \star.\tau/\alpha\}$.

Proof
Follows by straightforward induction over the structure of $\Delta \uplus \{\alpha : \star \to \star\}; \Xi \vdash M \rhd_{\alpha\tau'} e$ and Lemma A.9 (world substitution for type encoding). □

Lemma B.9 (*Substitution of for encoding modal variables*)
If $\Delta \vdash \Upsilon \rhd_\tau \Gamma_1$
and $\Delta \vdash \Omega \rhd \Gamma_2$
and $\Delta \vdash A_1 \rhd_\tau \tau_{A_1}$
and $\Omega \uplus \{x : A_2\}; \Upsilon \stackrel{\text{SDP}}{\vdash} M_1 : A_1$
and $\Delta; \mathrm{dom}(\Omega) \uplus \{x\} \vdash M_1 \rhd_\tau e_1$
and $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} M_2 : A_2$
and $\Delta \uplus \{\alpha : \star \to \star\}; \mathrm{dom}(\Omega) \vdash M_2 \rhd_{\alpha\tau'} e_2$ then
then $\Delta; \mathrm{dom}(\Omega) \vdash M_1\{M_2/x\} \rhd_\tau e_1'$ where
$\Delta; \Gamma_1 \uplus \Gamma_2 \stackrel{\mathbb{E}_\omega}{\vdash} e_1' \equiv_{\beta\eta} e_1\{\Lambda\alpha : \star \to \star.e_2/x\} : \tau_{A_1}$.

Proof
By induction over the structure of $\Delta; \mathrm{dom}(\Omega) \uplus \{x\} \vdash M_1 \rhd_\tau e_1$. The only interesting case is for en_bvar which uses Lemma B.8 (world substitution for term encodings). □

Lemma B.10 (*Typing for elimination*)

1. If $\Psi \stackrel{\text{SDP}}{\vdash} V \Uparrow B$ and $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \Theta : A\langle\Psi{:}\Sigma\rangle$ then $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \langle A, \Psi, \Theta\rangle(V) : A\langle B\rangle$.
2. If $\Psi \stackrel{\text{SDP}}{\vdash} V \downarrow B$ and $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \Theta : A\langle\Psi{:}\Sigma\rangle$ then $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \langle A, \Psi, \Theta\rangle(V) : A\langle B\rangle$.
3. If $\Psi \stackrel{\text{SDP}}{\vdash} x \downarrow B$ and $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \Theta : A\langle\Psi{:}\Sigma\rangle$ then $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \Theta(x) : A\langle B\rangle$.
4. If $\Psi \stackrel{\text{SDP}}{\vdash} c \downarrow B \to b$ and $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \Theta : A\langle\Psi{:}\Sigma\rangle$ then $\Omega; \Upsilon \stackrel{\text{SDP}}{\vdash} \Theta(c) : A\langle B \to b\rangle$.

Proof
Parts 1 and 2 follow by mutual induction over the structure of $\Psi \stackrel{\text{SDP}}{\vdash} V \Uparrow B$ and $\Psi \stackrel{\text{SDP}}{\vdash} V \downarrow B$. Parts 3 and 4 follow as corollaries.
□

Lemma B.11 (*Substitution application*)
If $\Delta; \Psi; \Theta; e_\Theta \blacktriangleright_\tau^{\tau_A} S$ and $\Psi(x) = B$ then $S(x) = \mathbf{uniter}[\![B^*]\!][\tau_A]\ e_\Theta\ e'$ where $\Delta; \varnothing \vdash \Theta(x) \rhd_\tau e'$.

Proof
Straightforward induction on the structure of $\Delta; \Psi; \Theta; e_\Theta \blacktriangleright_\tau^{\tau_A} S$. □

Lemma B.12 (*Static correctness with substitution*)
If $\Delta; \varnothing \vdash V \rhd_{\tau_A} e$ and $\Psi \stackrel{\text{SDP}}{\vdash} V \Uparrow B$ and $\Delta \vdash B \rhd_{\tau_A} \tau_B$ and $\Delta; \Psi; \Theta; e_\Theta \blacktriangleright_\tau^{\tau_A} S$ then $\Delta; \varnothing \stackrel{\text{SDP}}{\vdash} S(e) : \tau_B$.

Proof
Follows from Theorem 6.2 (static correctness, forward direction), Definition 6.1 (environment encoding), Lemma B.11 (substitution application), and Lemma B.10 (elimination typing). □

Lemma B.13 (*Equivalences between replacements and their restrictions*)

1. $\Theta(\mathsf{c_i}) = \llcorner\Theta\lrcorner(\mathsf{c_i})$.
2. $\Delta; \mathrm{dom}(\Omega) \vdash \Theta \rhd^{\tau_A}_{\tau} \mathsf{e}_\Theta \;\Leftrightarrow\; \Delta; \mathrm{dom}(\Omega) \vdash \llcorner\Theta\lrcorner \rhd^{\tau_A}_{\tau} \mathsf{e}_\Theta$.

Proof
Both parts follow trivially from the definition. □