September 2003

Boxes Go Bananas: Encoding Higher-Order Abstract Syntax with Parametric Polymorphism (Extended Version)

Geoffrey Washburn  
University of Pennsylvania

Stephanie Weirich  
University of Pennsylvania, sweirich@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
http://repository.upenn.edu/cis_reports/43


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/43
For more information, please contact repository@pobox.upenn.edu.
Boxes Go Bananas: Encoding Higher-Order Abstract Syntax with Parametric Polymorphism (Extended Version)

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

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

Comments
Abstract

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

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

1 Introduction

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

One obstacle preventing the widespread use of this technique is the difficulty in using elimination forms, such as catamorphisms\(^2\), for datatypes containing functions. The general form of catamorphism for these datatypes requires that an inverse be simultaneously defined for every iteration [16]. Unfortunately, many

\(^1\)While the name comes from Pfenning and Elliott [21], the idea itself goes back to Church. [4].
\(^2\)Catamorphisms (also called folds) are sometimes represented with the bananas (\ · \) notation [15].
operations that we would like to define with catamorphisms require inverses that do not exist or are expensive to compute.

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

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

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

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

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

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

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

2 Catamorphisms for datatypes with embedded functions

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

---

All of the following examples are in the syntax of the Haskell language [19]. While some of the later examples require an extension of the Haskell type system—first-class polymorphism—this extension is supported by the Haskell implementations GHC and Hugs.
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 \( \lambda x \rightarrow x \) or the infinite loop \( \lambda x.((\lambda x.x)\lambda x.((\lambda x.x)\lambda x.((\lambda x.x)))) \) as App (Lam \( \lambda x \rightarrow \text{App} x x \)) (Lam \( \lambda x \rightarrow \text{App} x x \)).

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

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

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

eval :: Exp \( \rightarrow \) Value
eval (Lam x) = Fn (eval \( \cdot \) x \( \cdot \) uneval)
eval (App y z) = unFn (eval y) (eval z)
uneval :: Value \( \rightarrow \) Exp
uneval (Fn x) = Lam (uneval \( \cdot \) x \( \cdot \) eval)

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

eval (App (Lam (\( \lambda x \rightarrow x \))) (Lam (\( \lambda y \rightarrow y \))))
= unFn (eval (Lam (\( \lambda x \rightarrow x \))))
  (eval (Lam (\( \lambda y \rightarrow y \))))
= unFn (Fn (eval \( \cdot \) \( \lambda x \rightarrow x \) \( \cdot \) uneval))
  (Fn (eval \( \cdot \) \( \lambda y \rightarrow y \) \( \cdot \) uneval))
= (eval \( \cdot \) uneval) (Fn (eval \( \cdot \) uneval))
= (\( \lambda x \rightarrow x \)) (Fn (\( \lambda y \rightarrow y \)))
= Fn (\( \lambda y \rightarrow y \))

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 \( \rightarrow \) b \( \rightarrow \) b).

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

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

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

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

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

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

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

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

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

Figure 2: Fegaras/Sheard catamorphism

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

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

Using cata to implement operations such as eval is convenient because the pattern of recursion is already specified. None of eval, evalAux or unevalAux are recursively defined. However, for some operations, there is no obvious (or efficient) inverse. For example, to using cata to print out expressions also requires writing a parser. Fegaras and Sheard [9] noted that sometimes the operation of the catamorphism often undoes with f what it has just done with g. This situation occurs when the argument to cata contains only parametric functions. A parametric function is one that does not analyze its argument with case or cata.

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

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

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

\[
\text{countvar} (\text{lam} (\lambda x . \text{app} x x)) \\
= (\text{countvar} . (\lambda x . \, x + x) . \text{Place}) \, 1 \\
= (\lambda x . \, (\text{countvar} (\text{Place} x)) \\
+ (\text{countvar} (\text{Place} x))) \, 1 \\
= (\text{countvar} (\text{Place} 1)) + (\text{countvar} (\text{Place} 1)) \\
= 2
\]

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

```
badplace :: Exp Int 
badplace = \(x \rightarrow \text{Place} \, 3\)
```

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

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

```
badcata :: Exp Int 
badcata = lam (\(x \rightarrow \text{if} \, (\text{countvar} \, x = 1) \)
then app x x 
else x)
```

```
badcata :: Exp a 
badcata = lam (\(x \rightarrow \text{case} \, x \, \text{of} \)
Roll (App \(v\) \(w\)) -> app x x 
Roll (Lam \(f\)) -> x 
Place \(v\) -> x)
```

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

---

5It 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.
2.1 Enforcing parametricity with type abstraction

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

\begin{verbatim}
iter0 :: (ExpF b -> b) -> (forall a. Exp a) -> b
iter0 = cata
\end{verbatim}

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

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

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

\begin{verbatim}
openiter1 :: (ExpF b -> b) -> (Exp b -> Exp b) -> (forall a. Exp a) -> b
openiter1 f x = \y -> cata f (x (place y))
\end{verbatim}

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 \texttt{forall a. Exp a -> Exp a}.

\begin{verbatim}
iter1 :: (ExpF b -> b) -> (forall a. Exp a -> Exp a) -> b
iter1 = openiter1
\end{verbatim}
With \texttt{iter1} we can determine if that one free variable occurs in an expression.

\begin{verbatim}
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
\end{verbatim}

An \texttt{app} expression uses the free variable if either the function or the argument uses it. The occurrence of the bound variable of a \texttt{lam} is not an occurrence of the free variable, so \texttt{False} is the argument to \texttt{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 \texttt{True} for the value of the free variable. If the result is \texttt{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 \texttt{forall}. The types of several such iterators are shown below. For example, the third one, \texttt{iterList}, may analyze expressions with arbitrary numbers of free variables.

\begin{verbatim}
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)
\end{verbatim}

Each of these iterators is defined by using \texttt{xmap} to map \texttt{(cata f)} and \texttt{place}. Thus we can easily implement them by defining the appropriate version of \texttt{xmap}. However, because \texttt{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 \texttt{xmap\{g\}} generates the instance of \texttt{xmap} for the type constructor \texttt{g}.

\begin{verbatim}
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|}
\end{verbatim}

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

\begin{verbatim}
iter1 :: (f b -> b)
  -> (forall a. (Exp a -> (Exp a)) -> b -> b)
\end{verbatim}

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

\begin{verbatim}
iter1 :: (ExpF a -> a)
  -> (forall b. Exp b -> Exp b) -> a -> a
iter1 = openiter
\end{verbatim}
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.

```haskell
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.

```haskell
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 `->`.

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

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

### 2.2 Examples of iteration

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

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

```haskell
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.

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

Another operation we might wish to perform for a \(\lambda\)-calculus expression is to reduce it to a simpler form. As an example, we next implement parallel reduction for a \(\lambda\)-calculus expression.\(^6\) Parallel reduction differs

---

\(^6\)This example is from Schümmerm et. al [24].
from full reduction in that it does not reduce any newly created redexes. Therefore, it terminates even for expressions with no $\beta$-normal form. Parallel reduction may be specified by the following inductive definition.

\[
M \Rightarrow M' \quad \lambda x. M \Rightarrow \lambda x. M' \\
MN \Rightarrow M'N' \quad (\lambda x. M)N \Rightarrow M'(x/N')
\]

We use \texttt{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 \texttt{lam} expression, and if so, perform the reduction. However, we cannot do a case analysis on expressions, as the type \texttt{Exp a} is abstract. Therefore, we implement parallel reduction with a “pairing” trick\textsuperscript{7}. As we iterate over the term we produce \texttt{two} results, stored in the following record:

\[
\texttt{data PAR a = PAR \{ par :: Exp a, apply :: Exp a -> Exp a \}}
\]

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

\[
\texttt{parAux :: ExpF (PAR a) \rightarrow PAR a}
\]

\[
\texttt{parAux (Lam f) = PAR \{ par = lam (par . f . var), apply = par . f . var \}}
\]

\[
\texttt{where}
\]

\[
\texttt{var :: Exp a \rightarrow PAR a}
\]

\[
\texttt{var x = PAR \{ par = x, apply = app x \}}
\]

\[
\texttt{parAux (App x y) = PAR \{ par = apply x (par y), apply = app (apply x (par y)) \}}
\]

\[
\texttt{parallel :: (forall v. Exp v) \rightarrow (forall v. Exp v)}
\]

\[
\texttt{parallel x = par (iter0 parAux x)}
\]

For example:

\[
\texttt{show (parallel (app (lam (\x \rightarrow app x x)) (lam (\y \rightarrow y))))} = "((fn a. a) (fn a. a))"
\]

While we could not write the most natural form of parallel reduction with \texttt{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 \cite{6}. This sophisticated algorithm performs “administrative” redexes at the meta-level so that the result term has no more redexes than the original expression. The algorithm is based on two mutually recursive operations: \texttt{cpsmeta} performs closure conversion given a meta-level continuation (a term of type \texttt{Exp a -> Exp a}), and \texttt{cpsobj} does the same with an object-level continuation (a term of type \texttt{Exp a}).

\[
\texttt{data CPS a = CPS \{}
\]

\[
\texttt{cpsmeta :: (Exp a -> Exp a) \rightarrow Exp a,}
\]

\[
\texttt{cpsobj :: Exp a -> Exp a} \}
\]

\textsuperscript{7}Pairing was first used to implement the predecessor operation for Church numbers. The iteration simultaneously computes the desired result with auxiliary operations.

\textsuperscript{8}In Haskell, the notation \texttt{apply x} projects the \texttt{apply} component from the record \texttt{x}. 

10
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.

\[
\text{value} :: \text{Exp} \ a \rightarrow \text{CPS} \ a
\]
\[
\text{value} \ x = \text{CPS} \ \{ \ \text{cpsmeta} = \ k \rightarrow k \ x, \ \\
\text{cpsobj} = c \rightarrow \text{app} \ c \ x \ \}
\]

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

\[
\text{cpsAux} :: \text{ExpF} (\text{CPS} \ a) \rightarrow \text{CPS} \ a
\]
\[
\text{cpsAux} \ (\text{App} \ e1 \ e2) = \ \\
\text{CPS} \ \{ \ \text{cpsmeta} = \ k \rightarrow \text{appexp} (\text{lam} \ k), \ \\
\text{cpsobj} = \text{appexp} \ \}
\]
\[
\text{where} \ \\
\text{appexp} \ c = \ \\
(\text{cpsmeta} \ e1) (\ y1 \rightarrow \ \\
(\text{cpsmeta} \ e2) (\ y2 \rightarrow \ \\
\text{app} (\text{app} \ y1 \ y2) \ c))
\]

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.

\[
\text{cpsAux} \ (\text{Lam} \ f) = \ \\
\text{value} \ (\text{lam} \ (\text{lam} \ . \ \text{cpsobj} \ . \ f \ . \ \text{value}))
\]

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

\[
\text{cps} :: (\forall a. \ \text{Exp} \ a) \rightarrow (\forall a. \ \text{Exp} \ a)
\]
\[
\text{cps} \ x = \text{lam} \ (\ a \rightarrow \ \\
\text{cpsmeta} \ (\text{iter0} \ \text{cpsAux} \ x) \ (\ m \rightarrow \text{app} \ a \ m))
\]

show (\text{cps} (\text{lam} \ ((x \rightarrow \text{app} \ x) x)))
= "(fn a. (fn b. (fn c. (((b b) c)))))"

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

3 Encoding iteration in F_ω

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

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

To create an expression, roll must encode this elimination. Therefore, roll returns a function that applies its argument f (the elimination function) to the result of iterating over x. Again, to use xmap we
type Rec f a = (f a -> a) -> a
data ExpF a = Lam (a -> a) | App a a
type Exp a = Rec ExpF a

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

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

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

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

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

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

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

Figure 4: Catamorphism in the $F_\omega$ fragment of Haskell
need a right inverse for \( \text{cata } f \). The term \( \text{place} \) in Figure 4 is an expression that when analyzed returns its argument. We can show that \( \text{place} \) is a right inverse by expanding the above definitions:

\[
\text{cata } f \cdot \text{place} = (\lambda x \rightarrow \text{cata } f (\text{place } x)) \\
= (\lambda x \rightarrow (\text{place } x) f) \\
= (\lambda x \rightarrow ((\lambda y \rightarrow x) f)) \\
= (\lambda x \rightarrow x)
\]

### 3.1 Reasoning about iteration

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

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

Using parametricity, we can derive a *free theorem* [28] about expressions of type \((\forall a. \text{Exp } a)\). If \( x \) has this type, then

\[ f \cdot f' = \text{id} \quad \text{and} \quad f \cdot g = h \cdot \text{fst}(\text{xmap{|b|}}(f,f')) \Rightarrow f \cdot (x \cdot g) = x \cdot h \]

The equivalence in this theorem is equivalence in some parametric model of \( F_\omega \), such as the term model with \( \beta\eta \)-equivalence. Using the free theorem, we can prove a number of properties about iteration. First, we can show that iterating \( \text{roll} \) is an identity function, that \( \text{iter0 } \text{roll} = \text{id} \). Using this result we can show the *uniqueness property* for \( \text{iter} \), which describes when a function is equal to an application of \( \text{iter} \).

It resembles an “induction principle” for \( \text{iter0} \).

\[ f \cdot f' = \text{id} \quad \text{and} \quad f \cdot \text{roll} = h \cdot \text{fst}(\text{xmap{|b|}}(f,f')) \Leftrightarrow f = \text{iter0 } h \]

The \( \Leftrightarrow \) direction follows directly from the implementation of \( \text{iter0} \) and \( \text{roll} \). The \( \Rightarrow \) direction follows from the free theorem.

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

\[ f \cdot f' = \text{id} \quad \text{and} \quad f \cdot g = h \cdot \text{fst}(\text{xmap{|b|}}(f,f')) \Rightarrow f \cdot \text{iter0 } g = \text{iter0 } h \]

However, there is an important property about this encoding of the \( \lambda \)-calculus that we have not proven. *Adequacy* states that if a \( F_\omega \) term is of type \( (\forall a. \text{Exp } a) \), then it should be the encoding of some \( \lambda \)-calculus expression. In other words, there is no extra “junk” in the type \( (\forall a. \text{Exp } a) \), such as \( \text{badcase} \). As a first step towards proving this result, we next show how this \( F_\omega \) library can encode a language with iteration over HOAS that itself adequately embeds the \( \lambda \)-calculus.

### 4 Enforcing parametricity with modal types

In the next section, we formally describe the connection between the interface we have provided for iteration over higher-order abstract syntax and the modal calculus of Schürrmann, Despeyroux and Pfenning (SDP) [24]. We do so by using this library to give a sound and complete embedding of the SDP calculus.
into $F_\omega$. 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 5.

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

$$
\frac{\emptyset; \emptyset \vdash M : A}{\Omega; \emptyset \vdash \text{box} M : \Box A} \quad \frac{\emptyset; \emptyset \vdash x : A}{\emptyset; \emptyset \vdash \text{let box } x : M = M_1 \text{ in } M_2 : A} \quad \frac{\emptyset; \emptyset \vdash x : A_1, \Upsilon \vdash M_2 : A_2}{\emptyset; \emptyset \vdash \text{let box } x : A_1 \rightarrow \text{in } M_2 : A_2}
$$

A $\text{boxed}$ term, $M$, has type $\Box A$ only if it has type $A$ with respect to the valid assumptions in $\Omega$, and no assumptions in local environment. The $\text{let box}$ elimination construct allows for the introduction of valid assumptions into $\Omega$, binding the contents of the boxed term $M_1$ in the body $M_2$. This binding is allowed because the contents of $\text{boxed}$ 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, $\text{boxed}$ terms behave like suspensions, while $\text{let box}$ substitutes the contents of a $\text{boxed}$ term for the bound variable. Because the operational semantics is defined simultaneously with conversion to canonical forms, it is parameterized by the environment $\Psi$ that describes the types of free local variables appearing in the expression.

$$
\frac{\emptyset; \emptyset \vdash M_1 \leftarrow \text{box } M'_1 : \Box A_1}{\emptyset; \emptyset \vdash \text{let box } x : A_1 \rightarrow \text{in } M_2 \leftarrow V : A_2}
$$

To enforce the separation between the iterative and parametric function spaces, the SDP calculus defines those types, $B$, that do not contain a $\Box$ type to be “pure”. Objects in the calculus with type $\Box B$, boxed pure types, can be examined intensionally using an iteration operator, while objects of arbitrary impure type, $A$, cannot. This forces functions of pure type, $\lambda x : B \rightarrow M : B_1 \rightarrow B_2$, 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 $\text{boxed}$ 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, $\Sigma$, of data constructor constants, $c$, for that base type. Each of the constructors in this signature must be of type $B \rightarrow b$. Because $B$ is a pure type, these constructors may only take parametric functions as arguments.
For example, consider a signature describing the untyped $\lambda$-calculus, $\Sigma = \{\text{app} : b \times b \to b, \text{lam} : (b \to b) \to b\}$, where the constant type $b$ corresponds to $\text{Exp}$. Using this signature, we can write a function to count the number of bound variables in an expression, as we did in Section 2.

\[
\text{countvar} \triangleq \lambda x : b. \\
\text{iter}[\square b, \text{int}][\{\text{app} \mapsto \lambda y : \text{int}.\lambda z : \text{int}.y + z, \\
\text{lam} \mapsto \lambda f : \text{int} \to \text{int}.f\}] x
\]

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

The typing rule for $\text{iter}$ is the following:

\[
\frac{\Omega; \Upsilon \vdash M : \square B \\
\Omega; \Upsilon \vdash \Theta : A(\Sigma)}{\Omega; \Upsilon \vdash \text{iter}[\square B, A] [\Theta] M : A(B)} \quad \text{tp\_iter}
\]

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

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

\[
\Psi \vdash M \to \text{box} M' : \square B \\
\Theta \vdash M' \uparrow V' : B \\
\Psi \vdash \langle A, \Psi, \Theta \rangle(V') \to V : A(B) \\
\Psi \vdash \text{iter}[\square B, A][\Theta] M \to V : A(B) \quad \text{ev\_iter}
\]

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

Elimination is used to describe the structure of a term after iteration. The only interesting cases to elimination are those for variables, constants, and abstractions.

\[
\langle A, \Psi, \Theta \rangle(x) \triangleq \Theta(x) \quad \text{el\_var} \\
\langle A, \Psi, \Theta \rangle(c) \triangleq \Theta(c) \quad \text{el\_const}
\]

\[
\langle A, \Psi \uplus \{x' : B\}, \Theta \uplus \{x \mapsto x'\} \rangle(V) \triangleq M \\
\langle A, \Psi, \Theta \rangle(\lambda x : B. V) \triangleq \lambda x' : A(B). M \quad \text{el\_lam}
\]

When elimination encounters an abstraction, it chooses a fresh variable and adds it to the mapping $\Theta$. 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 $\Theta$.

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 $\Sigma$ 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.

\footnote{For simplicity, our formal presentation of SDP (in Figure 5) does not include integers. However, it is straightforward to extend this calculus to additional base types.}
5 Encoding SDP in $F_\omega$

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 $F_\omega$, using a variation of these terms.

There are two key ideas behind our encoding:

- We use type abstraction to ensure that the encoding of boxed 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 boxed term we use a fresh type variable to create an arbitrary world. We then encode the enclosed term with that new world and wrap the result with a type abstraction. As a consequence, the encoding of a data-structure within a box cannot contain free local variables because their types would mention that fresh type variable outside of the scope of the type abstraction.

- We encode constants in the source language as their elimination form with roll. Furthermore, we restrict the result of elimination to be of the type that is the world in which the term was encoded. However, the encoding of boxed expressions quantifies over that world, allowing the resulting computations to be of arbitrary type.

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

Signatures. The encoding of signatures in the SDP calculus, notated $\tau(\Sigma)$, 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 $\tau$, corresponds to the argument of the type constructor.

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

\[
\forall c_i \in \text{dom}(\Sigma) \quad \Sigma(c_i) = B_i \rightarrow b \\
\tau(\Sigma) \triangleq (\mathcal{L}(c_1) : \tau(B_1), \ldots, \mathcal{L}(c_n) : \tau(B_n))
\]

\[\text{en_sig}\]
We often use parameterization and the signature translation to build type constructors in the target language, so we define the following two abbreviations:

\[ B^* \triangleq \lambda \alpha : \star.B \]  
\[ \Sigma^* \triangleq \lambda \alpha : \star.\alpha \langle \Sigma \rangle \]

**Types.** As with the encoding of signatures, the encoding of types is parameterized by the worlds in which they occur. We write the judgment for encoding a type \( A \) in the source calculus in world \( \tau \) as \( \Delta \vdash A \uparrow \tau \).

The environment \( \Delta \) 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 \( \text{Rec ExpF a} \) from Section 3. Therefore, we define the abbreviation \( \text{Rec} \Sigma^* \alpha \triangleq (\Sigma^* \alpha \to \alpha) \to \alpha \), intuitively a fixed point of \( \Sigma^* \), to the same idea of encoding a datatype as its elimination form.

\[ \Delta \vdash b \uparrow \tau \text{ Rec } \Sigma^* \tau \text{ en tp b} \]

The rule for boxed types uses type abstraction to ensure the result is parametric with respect to its world. Naively, 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.

\[ \alpha \notin \Delta \quad \Delta \uplus \{ \alpha : \star \} \vdash A \uparrow \alpha \tau' \quad \Delta \vdash \Box A \uparrow \forall \alpha : \star.\tau' \text{ en tp box wrong} \]

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 \( \tau \). Instead, we use the fresh type variable to create a new world from the current world and consider \( \alpha \) as a “world transformer”. During the translation, a term will be encoded with a stack of world transformers, somewhat akin to stack of environments in the implicit formulation of modal types [7].

\[ \alpha \notin \Delta \quad \Delta \uplus \{ \alpha : \star \to \star \} \vdash A \uparrow \alpha \tau' \quad \Delta \vdash \Box A \uparrow \forall \alpha : \star \to \star.\tau' \text{ en tp box} \]

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

\[ \Delta \vdash 1 \uparrow \tau \text{ 1(}\tau\text{ en tp unit)} \]

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

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

Our encoding of boxed terms follows immediately from the encoding of boxed 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.

\[ \alpha \notin \Delta \quad \Delta \uplus \{ \alpha : \star \to \star \}; \Xi \vdash M \uparrow \alpha \tau \ e \quad \Delta; \Xi \vdash \text{box } M \uparrow \tau \Lambda \alpha : \star \to \star.\text{e en box} \]
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 na"ively expect.

\[
\begin{array}{c}
\Delta \vdash \Box A_1 \triangleright_\tau \tau_1 \\
\Delta ; \Xi \vdash M_1 \triangleright_\tau e_1 \\
\Delta ; \Xi \triangleright_\tau \{ x \} \vdash M_2 \triangleright_\tau e_2 \\
\Delta ; \Xi \vdash \text{let } x : A_1 = M_1 \text{ in } M_2 \triangleright_\tau (\lambda x : \tau_1 . e_2) e_1 \end{array}
\]

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 \( \tau \). However, let box allows us to bind variables that are accessible in any world and using \( A_1 \) encoded against \( \tau \) 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 \( \Xi \) to keep track of variables bound by let box. When encoding variables, we check whether \( x \) occurs in \( \Xi \) and perform instantiations as necessary.

\[
\begin{array}{c}
x \notin \Xi \\
\Delta ; \Xi \vdash x \triangleright_\tau x \end{array}
\]

en_var

\[
\begin{array}{c}
x \in \Xi \\
\Delta ; \Xi \vdash x \triangleright_\tau x [ \lambda \alpha : \star . \tau ] \end{array}
\]

en_bvar

If the variable is in \( \Xi \), 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 \( \tau \). Doing so ensures that if we substitute the encoding of a boxed term into the encoding of another boxed term, the type correctness of the embedding is maintained by correctly propagating the enclosing world.

Figure 7 shows the types and definitions of the library routines used by the encoding. The only difference between it and Figure 4 is that iter abstracts the current world and requires that its argument be valid in any transformation of the current world. Again, we make use of the polytypic function xmap to lift cata to arbitrary type constructors. Because xmap is defined by the structure of a type constructor \( \tau \), we cannot directly define it as a term in \( F_\omega \). Instead, we will think of \( \text{xmap} \{ \tau \} \) as macro that expands to the mapping function for the type constructor \( \tau \). (We use the notation \( \{ \} \) to distinguish between polytypic instantiation

\[
\begin{array}{c}
\text{let box} : \forall \alpha : \star . (\Sigma^* \alpha \rightarrow \alpha) \rightarrow (\text{Rec} \Sigma^* \alpha) \rightarrow \alpha \\
\text{cata} \triangleq \Lambda \alpha : \star . \lambda f : (\Sigma^* \alpha \rightarrow \alpha) . \lambda y : (\text{Rec} \Sigma^* \alpha) . y f \\
\text{place} : \forall \alpha : \star . \alpha \rightarrow \text{Rec} \Sigma^* \alpha \\
\text{place} \triangleq \Lambda \alpha : \star . \lambda x : (\Sigma^* \alpha \rightarrow \alpha) . x \\
\text{xmap} \{ \tau : \star \rightarrow \star \} : \forall \alpha : \forall \beta : \star . (\alpha \rightarrow \beta \times \beta \rightarrow \alpha) \rightarrow (\tau \alpha \rightarrow \tau \beta \times \tau \beta \rightarrow \tau \alpha) \\
\text{openiter} \{ \tau : \star \rightarrow \star \} : \forall \alpha : \forall \beta : \star . (\Sigma^* \alpha \rightarrow \alpha) \rightarrow \tau (\text{Rec} \Sigma^* \alpha) \rightarrow \tau \alpha \\
\text{openiter} \{ \tau : \star \rightarrow \star \} \triangleq \Lambda \alpha : \forall \beta : \star \rightarrow \star . \lambda f : \Sigma^* \alpha \rightarrow \alpha . \text{fst} (\text{xmap} \{ \tau \} \{ \text{Rec} \Sigma^* \alpha \} \{ \text{cata} \{ \alpha \} f \} . \text{place} \{ \alpha \}) \\
\text{iter} \{ \tau : \star \rightarrow \star \} : \forall \gamma : \forall \alpha : \star . (\Sigma^* \alpha \rightarrow \alpha) \rightarrow (\forall \beta : \star \rightarrow \star . \tau (\text{Rec} \Sigma^* (\beta \gamma)) \rightarrow \tau \alpha \\
\text{iter} \{ \tau : \star \rightarrow \star \} \triangleq \Lambda \gamma : \forall \alpha : \forall \beta : \star \rightarrow \star . \tau (\text{Rec} \Sigma^* (\beta \gamma)) . \text{openiter} \{ \tau \} \{ \alpha \} f (x \mid \lambda : \star . \alpha \}) \\
\text{roll} : \forall \alpha : \Sigma^* (\text{Rec} \Sigma^* \alpha) \rightarrow \text{Rec} \Sigma^* \alpha \\
\text{roll} \triangleq \Lambda \alpha : \lambda x : (\Sigma^* \alpha) . \lambda f : \Sigma^* \alpha \rightarrow \alpha . f (\text{openiter} \{ \Sigma^* \} \{ \alpha \} f x)
\end{array}
\]

Figure 7: Library routines
and parametric type instantiation.) This expansion is done according to the definition in Appendix A. We do not cover the implementation here, see Hinze [11] for details.

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

\[
\Sigma(c) = B \rightarrow b \quad \Delta \vdash B \triangleright_{\tau} \tau_B
\]

\[\Delta; \Xi \vdash e \triangleright_{\tau} \lambda x : \tau_B. \text{roll}[\tau]\left[\text{inj}_{\ell(c)} x\right] \text{ of } \Sigma^* (\text{Rec } \Sigma^* \tau) \quad \text{en\_con}
\]

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 \( \Theta \) and argument term \( M \).

\[\Delta \vdash A \triangleright_{\tau} \tau_A \quad \Delta; \Xi \vdash \Theta \triangleright_{\tau^A} e_\Theta \quad \Delta; \Xi \vdash M \triangleright_{\tau} e_M
\]

\[\Delta; \Xi \vdash \text{iter } \triangleright_{\tau} \text{iter } \triangleright_{\tau^A} e_\Theta e_M \quad \text{en\_iter}
\]

The encoding of replacement \( \Theta \) 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.

\[\forall c_i \in \text{dom}(\Theta) \quad \Delta; \Xi \vdash \Theta(C_i) \triangleright_{\tau} e_{c_i}
\]

\[\Delta; \Xi \vdash \Theta \triangleright_{\tau^A} \lambda x : \Sigma^* \tau_A. \text{case } x \text{ of } \text{inj}_{\ell(c_1)} y_1 \text{ in } (e_1 y_1) \]

\[\ldots \]

\[\text{inj}_{\ell(c_n)} y_n \text{ in } (e_n y_n) \quad \text{en\_rep}
\]

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

### 5.1 Properties of the encoding

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

**Definition 5.1 (Encoding typing environments).** We write \( \Delta \vdash \Upsilon \triangleright_{\tau} \Gamma_1 \) and \( \Delta \vdash \Omega \triangleright_{\tau} \Gamma_2 \) to mean that

\[\forall x : A \in \Upsilon \quad \exists x : \tau_A \in \Gamma_1 \quad \text{where } \Delta \vdash \tau : * \text{ and } \Delta \vdash A \triangleright_{\tau} \tau_A
\]

\[\forall x : A \in \Omega \quad \exists x : \tau_A \in \Gamma_2 \quad \text{where there exists some } \Delta \vdash \tau' : * \text{ such that } \Delta \vdash \square A \triangleright_{\tau'} \tau_A
\]

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

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

**Theorem 5.2 (Static correctness).** Assume \( \Delta \vdash \Upsilon \triangleright_{\tau} \Gamma_1 \) and \( \Delta \vdash \Omega \triangleright_{\tau} \Gamma_2 \) and \( \Delta \vdash A \triangleright_{\tau} \tau_A \).

1. If \( \Delta; \text{dom}(\Omega) \vdash M \triangleright_{\tau} e \) then \( \Omega; \Upsilon \vdash M : A \iff \Delta; \Gamma_1 \cup \Gamma_2 \vdash e : \tau_A \).
2. If $\Delta; \text{dom}(\Omega) \vdash \Theta \subseteq \tau \ e_\theta$ then $\Omega; \Upsilon \vdash \Theta : A \langle \Sigma \rangle \iff \Delta; \Gamma_1 \sqcup \Gamma_2 \vdash e_\theta : \Sigma^* \tau_A \rightarrow \tau_A$.

Proof. By mutual induction over the translation of terms ($\Delta; \text{dom}(\Omega) \vdash M \uparrow \tau \ e$) and of replacements ($\Delta; \text{dom}(\Omega) \vdash \Theta \subseteq \tau \ A \tau \ e$).

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

Theorem 5.3 (Dynamic correctness). If $\emptyset; \Psi \vdash M : A$ and $\emptyset; \Psi \vdash M \uparrow \tau \ e$ and $\emptyset; \Psi \vdash V \uparrow \tau \ e'$ and $\emptyset \vdash A \uparrow \tau \ \tau_A$ and $\Delta \vdash \Psi \uparrow \tau \ \Gamma$ then

1. $\Psi \vdash M \equiv V : A \iff \emptyset; \Gamma \vdash e \equiv_{\beta\eta} e' : \tau_A$.
2. $\Psi \vdash M \equiv V : A \iff \emptyset; \Gamma \vdash e \equiv_{\beta\eta} e' : \tau_A$.

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

6 Future work

Although we have shown a very close connection between SDP and its encoding in $F_{\omega}$, we have not shown that this encoding is adequate. We would like to show that if $\tau$ is the image of an SDP type, then all terms of type $\tau$ are equivalent to the encoding of some SDP term. In other words, there is no extra “junk” of type $\tau$. Showing this result would also show that encoding the $\lambda$-calculus with $\text{app}$ and $\text{lam}$ is adequate, because the SDP calculus can already adequately encode the $\lambda$-calculus.

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

One important extension of this work is the case operator. Because there are limitations to what may be defined with $\text{iter}$, 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 $\text{iter}$, the anamorphism over datatypes with embedded functions. An implementation of this operation, called $\text{coiter}$, is below. The $\text{coiter}$ term is an anamorphism—it generates a recursive data structure from an initial seed.

```haskell
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 $\text{roll}$ and $\text{place}$. In the definition of $\text{coiter0}$ we use $\text{coplace}$ as the inverse where we would have used $\text{cata}$ in the definition of $\text{ana}$. A term of type $(\exists a. \text{Dia } f a)$ corresponds to the possibility type $(\Diamond b)$ in a modal calculus. However, while a general anamorphism is an inverse of a catamorphism, $\text{coiter}$ is not an inverse to $\text{iter}$. In fact, $\text{iter}$ cannot consume what $\text{coiter}$ produces, giving doubts to its practical use. (On the other hand, $\text{ana}$ itself has seen little practical use for datatypes with embedded functions.) From a logical point of view, this restriction makes sense. Combining anamorphisms and catamorphisms (even for datatypes without embedded functions) leads to general recursion.
7 Related work

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

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

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

8 Conclusion

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

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

Acknowledgements

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


A  Generic Haskell implementation of xmap

type XMap {*[]} t1 t2 = (t1 -> t2, t2 -> t1)

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

xmap {| t :: k |} :: XMap {[k]} t t
xmap {| Unit |} = (id,id)

xmap {| :+: |} (xmapA1,xmapA2) (xmapB1,xmapB2) =
  (\x -> case x of
    (Inl a) -> Inl (xmapA1 a)
    (Inr b) -> Inr (xmapB1 b),

23
\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 {| [] |} (xmapA1,xmapA2) =
  (fmap xmapA1, fmap xmapA2)

xmap {| [] |} (xmapA1,xmapA2) =
  (map xmapA1, map xmapA2)

B Full encoding of SDP

B.1 Parameterization

\[
\frac{\tau(b) \triangleq \tau}{\text{par}_b} \quad \frac{\tau(1) \triangleq 1}{\text{par}_\text{unit}} \quad \frac{\tau(B_1) \triangleq \tau_1 \quad \tau(B_2) \triangleq \tau_2}{\text{par}_\text{arrow}} \quad \frac{\tau(B_1) \triangleq \tau_1 \quad \tau(B_2) \triangleq \tau_2}{\text{par}_\text{times}}
\]

B.2 Signatures

\[
\forall c_i \in \text{dom}(\Sigma) \quad \Sigma(c_i) = B_i \rightarrow b \quad \tau(\Sigma) \triangleq (L(c_1) : \tau(B_1), \ldots, L(c_n) : \tau(B_n)) \quad \text{en}\_\text{sig}
\]

B.3 Types

\[
\frac{\Delta \vdash b : \tau \quad \text{Rec} \Sigma^* \chi \quad \tau}{\text{en}\_\text{tp}_b} \quad \frac{\alpha \notin \Delta \quad \Delta \uplus \{ \alpha : \tau \rightarrow \tau \}' \vdash A : \alpha : \tau \rightarrow \tau'}{\Delta \vdash \Box A : \forall \alpha : \tau \rightarrow \tau'. \tau'} \quad \text{en}\_\text{tp}_\text{box}
\]

\[
\frac{\Delta \vdash 1 : \tau(1)}{\text{en}\_\text{tp}_\text{unit}} \quad \frac{\Delta \vdash A_1 : \tau_1 \quad \Delta \vdash A_2 : \tau_2}{\Delta \vdash A_1 \rightarrow \tau_1 \rightarrow \tau_2} \quad \text{en}\_\text{tp}_\text{arrow}
\]

\[
\frac{\Delta \vdash A_1 : \tau_1 \quad \Delta \vdash A_2 : \tau_2}{\Delta \vdash A_1 \times A_2 : \tau_1 \times \tau_2} \quad \text{en}\_\text{tp}_\text{prod}
\]

24
B.4 Terms

\[
\frac{x \not\in \Xi}{\Delta; \Xi \vdash \Delta; \Xi \vdash x \triangleright_{\tau} x} \quad \text{en_var} \quad \frac{x \in \Xi}{\Delta; \Xi \vdash \Delta; \Xi \vdash x [\lambda \alpha : \star \tau]} \quad \text{en_bvar}
\]

\[
\frac{\alpha \not\in \Delta}{\Delta; \Xi \vdash \Delta; \Xi \vdash \alpha \triangleright_{\alpha \tau} e} \quad \text{en_box}
\]

\[
\frac{\Sigma(e) = B \rightarrow b}{\Delta; \Xi \vdash \Delta; \Xi \vdash \Sigma(e) \triangleright \tau B \triangleright \tau B \text{ roll}[\tau](\text{inj}_{\Sigma(e)} x \text{ of } \Sigma^{*}(\text{Rec } \Sigma^{*} \tau))} \quad \text{en_con}
\]

\[
\frac{\Delta; \Xi \vdash \Delta; \Xi \vdash M \triangleright_{\tau} e \quad \Delta; \Xi \vdash \Delta; \Xi \vdash A_{1} \triangleright_{\tau} \tau_{1}}{\Delta; \Xi \vdash \Delta; \Xi \vdash \lambda x : A_{1} M \triangleright_{\tau} \lambda x : \tau_{1} e} \quad \text{en_abs}
\]

\[
\frac{\Delta; \Xi \vdash \Delta; \Xi \vdash M_{1} \triangleright_{\tau} e_{1} \quad \Delta; \Xi \vdash \Delta; \Xi \vdash M_{2} \triangleright_{\tau} e_{2}}{\Delta; \Xi \vdash M_{1} M_{2} \triangleright_{\tau} (e_{1}, e_{2})} \quad \text{en_app}
\]

\[
\frac{\Delta; \Xi \vdash \Delta; \Xi \vdash M \triangleright_{\tau} e}{\Delta; \Xi \vdash \Delta; \Xi \vdash \text{snd } M \triangleright_{\tau} \text{snd } e} \quad \text{en_snd}
\]

B.5 Replacements

\[
\frac{\forall c \in \text{dom}(\Theta)}{\Delta; \Xi \vdash \Delta; \Xi \vdash \Theta(c) \triangleright_{\tau} e_{i} \quad \text{en_rep}}
\]

\[
\frac{\forall x : A \in \Upsilon}{\Delta; \Xi \vdash \Delta; \Xi \vdash x : \tau_{A} \in \Gamma_{1} \quad \text{where } \Delta \vdash \tau : \star \text{ and } \Delta \vdash A \triangleright_{\tau} \tau_{A}}
\]

\[
\forall x : A \in \Omega \quad x : \tau_{A} \in \Gamma_{2} \quad \text{where there exists some } \Delta \vdash \tau' : \star \text{ such that } \Delta \vdash \square A \triangleright_{\tau'} \tau_{A}
\]

C 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 C.1 (Encoding typing environments).** We write \( \Delta \vdash \Upsilon \triangleright_{\tau} \Gamma_{1} \text{ and } \Delta \vdash \Omega \triangleright \Upsilon \triangleright_{\tau} \Gamma_{2} \) to mean that

\[
\forall x : A \in \Upsilon \iff x : \tau_{A} \in \Gamma_{1} \quad \text{where } \Delta \vdash \tau : \star \text{ and } \Delta \vdash A \triangleright_{\tau} \tau_{A}
\]

\[
\forall x : A \in \Omega \iff x : \tau_{A} \in \Gamma_{2} \quad \text{where there exists some } \Delta \vdash \tau' : \star \text{ such that } \Delta \vdash \square A \triangleright_{\tau'} \tau_{A}
\]

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 \( \Omega \), it will type check with any other encoding of \( \Omega \).

The following theorem proves our primary static correctness result, supported by a number of lemmas that follow it.
Theorem C.2 (Static correctness).

1. If $\Delta; \text{dom}(\Omega) \vdash M \triangleright e$ then if $\Delta \vdash \Upsilon \triangleright_\rho \Gamma_1$ and $\Delta \vdash \Omega \triangleright \Gamma_2$ and $\Delta \vdash A \triangleright_\rho \tau_A$ then $\Omega; \Upsilon \vdash M : A \iff \Delta; \Gamma_1 \cup \Gamma_2 \vdash e : \tau_A$.

2. If $\Delta; \text{dom}(\Omega) \vdash \Theta \triangleright_\rho^x e_\Theta$ then if $\Delta \vdash \Upsilon \triangleright_\rho \Gamma_1$ and $\Delta \vdash \Omega \triangleright \Gamma_2$ and $\Delta \vdash A \triangleright_\rho \tau_A$ then $\Omega; \Theta : A(\Sigma) \iff \Delta; \Gamma_1 \cup \Gamma_2 \vdash e_\Theta : \Sigma \tau_A \rightarrow \tau_A$.

Proof. By mutual induction over the structure of $\Delta; \text{dom}(\Omega) \vdash M \triangleright e$ and $\Delta; \text{dom}(\Omega) \vdash \Theta \triangleright^x e_\Theta$. The cases for former:

Case

$$\frac{x \not\in \text{dom}(\Omega)}{\Delta; \text{dom}(\Omega) \vdash x \triangleright x} \text{ en\_var}$$

Forward direction:

- By inversion on $\Omega; \Upsilon \vdash x : A$ and $x \not\in \text{dom}(\Omega)$ we can conclude that $A = \Upsilon(x)$.
- By Definition C.1 (environment encoding), if $\Upsilon(x) = A$ then $\Gamma_1(x) = \tau'_A$ where $\Delta \vdash A \triangleright \tau'_A$.
- Lemma C.13 (uniqueness of type encoding) tells us that if $\Delta \vdash A \triangleright \tau_A$ and $\Delta \vdash A \triangleright \tau'_A$ we have $\tau_A = \tau'_A$.
- Using Lemma C.20 (environment encoding well-formedness) we have $\Delta \vdash \Gamma_1$ and $\Delta \vdash \Gamma_2$, and along with $\Gamma_1(x) = \tau_A$, the variable typing rule $(\text{tp\_var})$, and weakening we can conclude $\Delta; \Gamma_1 \cup \Gamma_2 \vdash x : \tau_A$.

Backward direction:

- By Lemma C.8 (inversion) on $\Delta; \Gamma_1 \cup \Gamma_2 \vdash x : \tau_A$ we know that $x : \tau'_A \in \Gamma_1 \cup \Gamma_2$ where $\Delta \vdash \tau_A \equiv_{\beta\eta} \tau'_A : \ast$. Furthermore, we know that $x \not\in \text{dom}(\Omega)$ so by Definition C.1 (environment encoding) $x \not\in \text{dom}(\Gamma_2)$ and the disjointness of contexts means that $x : \tau'_A \in \Gamma_1$.
- Definition C.1 (environment encoding) tells us that if $x : \tau'_A \in \Gamma_1$ then $x : A' \in \Upsilon$ where $\Delta \vdash A' \triangleright \tau_A$. Using Lemma C.12 (type encoding with congruent results) on $\Delta \vdash A \triangleright \tau_A$ and $\Delta \vdash A' \triangleright \tau'_A$ with $\Delta \vdash \tau_A \equiv_{\beta\eta} \tau'_A : \ast$, we can conclude $A = A'$. Therefore by the local variable typing rule $(\text{tp\_var})$ on $x : A \in \Upsilon$ we can conclude $\Omega; \Upsilon \vdash x : A$.

Case

$$\frac{x \in \text{dom}(\Omega)}{\Delta; \text{dom}(\Omega) \vdash x \triangleright x[\lambda \alpha : \ast. \tau]} \text{ en\_bvar}$$

Forward direction:

- By inversion on $\Omega; \Upsilon \vdash x : A$ and $x \in \text{dom}(\Omega)$ we can conclude that $A = \Omega(x)$.
- By Definition C.1 (environment encoding), if $\Omega(x) = A$ then $\Gamma_2(x) = \tau'_A$ where $\Delta \vdash \square A \triangleright \tau'_A$ for some $\Delta \vdash \tau' : \ast$. By inversion, we know that $\Delta \cup \{ \beta : \ast \rightarrow \ast \} \vdash A \triangleright \tau''_A$ where $\tau''_A = \forall \beta : \ast \rightarrow \ast. \tau''_A$.
- Using Lemma C.20 (environment encoding well-formedness) we have $\Delta \vdash \Gamma_1$ and $\Delta \vdash \Gamma_2$, and along $\Gamma_2(x) = \forall \beta : \ast \rightarrow \ast. \tau''_A$ and the variable typing rule $(\text{tp\_var})$, the type application rule $(\text{tp\_tapp})$ and weakening we can derive $\Delta; \Gamma_1 \cup \Gamma_2 \vdash x[\lambda \alpha : \ast. \tau] : \tau''_A[\lambda \alpha : \ast. \tau/\beta]$.
- By Lemma C.15 (world substitution on type encoding) on $\Delta \vdash \lambda \alpha : \ast. \tau : \ast \rightarrow \ast$ and $\Delta \cup \{ \beta : \ast \rightarrow \ast \} \vdash A \triangleright_{\lambda \alpha : \ast. \tau} \tau''_A$ we know that $\Delta \vdash A \triangleright_{(\lambda \alpha : \ast. \tau)} \tau''_A[\lambda \alpha : \ast. \tau/\beta]$. By $\beta$-equivalence $(\text{tp\_eq\_abs\_beta})$ we know that $\Delta \vdash \equiv_{\beta\eta} (\lambda \alpha : \ast. \tau)\tau''_A : \ast$, so by Lemma C.14 (encoding under congruent worlds) on $\Delta \vdash A \triangleright \tau_A$ and $\Delta \vdash A \triangleright_{(\lambda \alpha : \ast. \tau)} \tau''_A[\lambda \alpha : \ast. \tau/\beta]$ we know that $\Delta \vdash \tau_A \equiv_{\beta\eta} \tau''_A[\lambda \alpha : \ast. \tau/\beta] : \ast$. 

26
Therefore, by type equivalence (\(\text{tp\_eq}\)) on \(\Delta \vdash \tau_A \equiv_{\beta\eta} \tau''_A\{\lambda: \star.\tau/\beta\} : \star\) we can conclude \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash x[\lambda: \star.\tau] : \tau_A\).

**Backward direction:**

- From Lemma C.8 (inversion) on \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash x[\lambda: \star.\tau] : \tau_A\) we know \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash x : \forall \beta: \kappa.\tau'_A\) and \(\Delta \vdash \lambda: \star.\tau : \kappa\) where \(\Delta \vdash \tau_A \equiv_{\beta\eta} \tau'_A\{\tau_1/\beta\} : \star\) and \(\Delta \vdash \tau_1 \equiv_{\beta\eta} \lambda: \star.\tau : \kappa\). By further inversion on \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash x : \forall \beta: \kappa.\tau'_A\) we can conclude \(x : \tau''_A \in \Gamma_1 \cup \Gamma_2\) for \(\Delta \vdash \forall \beta: \kappa.\tau'_A \equiv_{\beta\eta} \tau''_A : \star\). By However, we know that \(x \in \text{dom}(\Omega)\) so by Definition C.1 (environment encoding) we have that \(x \in \text{dom}(\Gamma_2)\). Given that contexts are disjoint, \(x : \tau''_A \in \Gamma_2\).

- By inversion on \(\Delta \vdash \lambda: \star.\tau : \kappa\) we know that \(\kappa = \star \rightarrow \star\).

Definition C.1 (environment encoding) also allows us to conclude that given \(x : \tau''_A \in \Gamma_2\), we have \(x : A' \in \Omega\) where \(\Delta \vdash \Box A' \triangleright_{\tau'} \tau''_A\) for some \(\Delta \vdash \tau' : \star\).

- By using the typing rule for valid variables (\(\text{tp\_bvar}\)) on \(x : A' \in \Omega\) we can conclude \(\Omega; \Upsilon \vdash x : A'\).

- By inversion on \(\Delta \vdash \Box A' \triangleright_{\tau'} \tau''_A\) we have that \(\Delta \cup \{\beta: \star \rightarrow \star\} \vdash A' \triangleright_{\beta\tau'} \tau''_A\) where \(\tau''_A = \forall \beta: \star \rightarrow \star.\tau''_A\).

- Using Lemma C.15 (world substitution) on \(\Delta \vdash \lambda: \star.\tau : \star \rightarrow \star\) and \(\Delta \cup \{\beta: \star \rightarrow \star\} \vdash A' \triangleright_{\beta\tau'} \tau''_A\) we can conclude \(A' \triangleright_{(\lambda, \star.\tau')\beta} \tau''_A\{\lambda: \star.\tau/\beta\}\). By \(\beta\)-equivalence (\(\text{tp\_eqabs\_beta}\)) we have that \(\Delta \vdash (\lambda: \star.\tau') =_{\beta\eta} \tau : \star\). From Lemma C.11 (type encoding total and decidable) on \(\Delta \vdash \tau : \star\) that \(\Delta \vdash A' \triangleright_{\tau} \tau_2\). Using Lemma C.14 (encoding under congruent worlds) on \(\Delta \vdash A' \triangleright_{(\lambda, \star.\tau')\beta} \tau''_A\{\lambda: \star.\tau/\beta\}\) and \(\Delta \vdash A' \triangleright_{\tau} \tau_2\) and \(\Delta \vdash (\lambda: \star.\tau') =_{\beta\eta} \tau : \star\) we can conclude \(\Delta \vdash \tau''_A\{\lambda: \star.\tau/\beta\} \equiv_{\beta\eta} \tau_2 : \star\).

- By inversion on \(\Delta \vdash \forall \beta: \star \rightarrow \star.\tau'_A \equiv_{\beta\eta} \forall \beta: \star \rightarrow \star.\tau''_A\) we have that \(\Delta \cup \{\beta: \star \rightarrow \star\} \vdash \tau'_A \equiv_{\beta\eta} \tau''_A\{\lambda: \star.\tau/\beta\}\). Using this congruence with type equivalence for substitution (\(\text{tp\_eqsubst}\)) and \(\Delta \vdash \tau_1 \equiv_{\beta\eta} \lambda: \star.\tau : \star \rightarrow \star\) we can conclude \(\Delta \vdash \tau_A\{\tau_1/\beta\} \equiv_{\beta\eta} \tau''_A\{\lambda: \star.\tau/\beta\}\{\lambda: \star.\tau/\beta\}\{\lambda: \star.\tau/\beta\}\). By transitivity of type congruence (\(\text{tp\_eqtrans}\)) on \(\Delta \vdash \tau_A\{\tau_1/\beta\} \equiv_{\beta\eta} \tau''_A\{\lambda: \star.\tau/\beta\}\{\lambda: \star.\tau/\beta\}\{\lambda: \star.\tau/\beta\}\) and \(\Delta \vdash \tau''_A\{\lambda: \star.\tau/\beta\} \equiv_{\beta\eta} \tau_2 : \star\) and \(\Delta \vdash \tau_A \equiv_{\beta\eta} \tau''_A\{\tau_1/\beta\}\{\lambda: \star.\tau/\beta\}\{\lambda: \star.\tau/\beta\}\{\lambda: \star.\tau/\beta\}\) we have that \(\Delta \vdash \tau_A \equiv_{\beta\eta} \tau_2 : \star\).

- Lemma C.12 (typing encoding with congruent results) on \(\Delta \vdash A' \triangleright_{\tau} \tau_2\) and \(\Delta \vdash A \triangleright_{\tau} \tau_A\) with \(\Delta \vdash \tau_A \equiv_{\beta\eta} \tau_2 : \star\) we can conclude that \(A' = A\).

**Case**

\[
\begin{array}{c}
\alpha \notin \Delta \\
\Delta \cup \{\alpha: \star \rightarrow \star\}; \text{dom}(\Omega) \vdash M \triangleright_{\alpha\tau} e \\
\Delta; \text{dom}(\Omega) \vdash \boxed{M \triangleright_{\alpha\tau} \lambda\alpha: \star \rightarrow \star.e}
\end{array}
\]

**en\_box**

**Common:**

- We have \(\Delta \cup \{\alpha: \star \rightarrow \star\} \vdash \Omega \triangleright \Gamma_2\) by weakening and \(\Delta \cup \{\alpha: \star \rightarrow \star\} \vdash \emptyset \triangleright_{\alpha\tau} \emptyset\) by Definition C.1 (environment encoding).

The forward direction follows from straightforward use of induction:

- Using inversion on \(\Omega; \Upsilon \vdash \boxed{M: A}\) we can conclude \(\Omega; \emptyset \vdash M : A'\) where \(A = \Box A'\).

- By inversion on \(\Delta \vdash \Box A' \triangleright_{\tau} \tau_A\) we have that \(\Delta \cup \{\alpha: \star \rightarrow \star\} \vdash A' \triangleright_{\tau} \tau'_A\) where \(\tau_A = \forall \alpha: \star \rightarrow \star.\tau'_A\).

- Appealing to the induction hypothesis on \(\Delta \cup \{\alpha: \star \rightarrow \star\}; \text{dom}(\Omega) \vdash M \triangleright_{\alpha\tau} e\), with the auxiliary judgements \(\Delta \cup \{\alpha: \star \rightarrow \star\} \vdash \emptyset \triangleright_{\alpha\tau} \emptyset\) and \(\Delta \cup \{\alpha: \star \rightarrow \star\} \vdash \Omega \triangleright \Gamma_2\) and \(\Delta \cup \{\alpha: \star \rightarrow \star\} \vdash A' \triangleright_{\alpha\tau} \tau'_A\) and and \(\Omega; \emptyset \vdash M : A'\) we have a derivation \(\Delta \cup \{\alpha: \star \rightarrow \star\}; \Gamma_2 \vdash e : \tau'_A\).

- Via the typing rule for type abstraction (\(\text{tp\_tabs}\)) and weakening we can conclude \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash \alpha: \star \rightarrow \star.e; \forall \alpha: \star \rightarrow \star.\tau'_A\).
Backward direction:

- By Lemma C.8 (inversion) on $\Delta;\Gamma_1 \cup \Gamma_2 \vdash \Delta \alpha : \ast \rightarrow \ast.e : \tau_A$ we can conclude that 
  $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast.e : \tau_A'$ where $\Delta \vdash \tau_A \equiv_{\beta \eta} \forall \alpha : \ast \rightarrow \ast.\tau_A'$.

- From Lemma C.10 (inversion) on $\Delta \vdash A \vdash \tau_A$ and $\Delta \vdash \tau_A \equiv_{\beta \eta} \forall \alpha : \ast \rightarrow \ast.\tau_A'$ can conclude 
  $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast.\tau_A'$ which is $A = \square A'$.

- Given that $\alpha \not\in \Delta$ and $\Delta \vdash \tau : \ast$, we know that $\alpha \not\in \text{FTV}(\tau)$. Using Lemma C.3 (local strengthening) on $\Delta \vdash \Upsilon \vdash \Gamma_1$ and $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast \vdash M \vdash \tau$ and 
  $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast \vdash M \vdash \tau$ and $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast \vdash M \vdash \tau$ we can conclude $\Delta \vdash M \vdash \tau$.

- By appealing to the induction hypothesis on $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast \vdash M \vdash \tau$, with the auxiliary judgements $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast \vdash \Upsilon \vdash \Omega$ and $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast \vdash \Upsilon \vdash \Omega$ and 
  $\Delta \vdash \Delta \alpha : \ast \rightarrow \ast \vdash \Upsilon \vdash \Omega$ we have a derivation 
  $\Omega \vdash \Upsilon \vdash M : A'$.

- Using the typing rule for box (tp.box) on $\Omega ; \emptyset \vdash M : A'$ we can conclude $\Omega ; \Upsilon \vdash M : \square A'$.

Case

$$\Delta; \text{dom}(\Omega) \vdash \Upsilon : \tau \vdash (\tau)[\tau] \text{ en-unit}$$

Forward direction:

- By the unit typing rule (tp.unit) and weakening we can conclude $\Delta; \Gamma_1 \vdash \Upsilon : \tau \vdash (\tau)[\tau] : \forall \alpha : \ast.1(\alpha)$.

- Using the type application rule (tp.app) we have that $\Delta; \Gamma_1 \vdash \Upsilon : \tau \vdash (\tau)[\tau] : \forall \alpha : \ast.1(\alpha)$. Which by substitution is the same as $\Delta; \Gamma_1 \vdash \Upsilon : \tau \vdash (\tau)[\tau] : 1(\tau)$.

- From inversion on $\Omega; \Upsilon \vdash (\tau) : A$ we know that $A = 1$, so by inversion on $\Delta \vdash 1 \vdash \tau$ we know 
  $\tau_A = 1(\tau)$.

Backward direction:

- We can trivially conclude by the axioms for unit (tp.unit and en.tp.unit) that $\Omega; \Upsilon \vdash (\tau) : 1$ and 
  $\Delta \vdash 1 \vdash(\tau)$.

Case

$$\Delta; \text{dom}(\Omega) \vdash M \vdash e \quad \Delta \vdash A_1 \vdash \tau_1 \quad \Delta; \text{dom}(\Omega) \vdash \lambda x : A_1. M \vdash \lambda x : \tau_1 \vdash e \quad \text{en-abs}$$

Common:

- Using Definition C.1 (environment encoding) on $\Delta \vdash \Upsilon \vdash \Gamma_1$ and $\Delta \vdash \lambda x : A_1. M \vdash \lambda x : (\tau_1.e)$ we can conclude 
  $\Delta \vdash \Upsilon \vdash \{ x : A_1 \} \vdash \tau_1 \vdash \Gamma_1 \vdash \{ x : \tau_1 \}$.

The forward direction follows from straightforward use of induction:

- By inversion on $\Omega; \Upsilon \vdash \lambda x : A_1. M : A$ we know that $A = A_1 \rightarrow A_2$ and that 
  $\Omega; \Upsilon \vdash \{ x : A_1 \} \vdash M : A_2$.

- We know that $\Delta \vdash \lambda x : A_1. M : A_2$ therefore we can conclude $\Delta \vdash \Upsilon \vdash \{ x : A_1 \} \vdash \Gamma_1 \vdash \{ x : \tau_1 \}$.

- From inversion $\Delta \vdash \lambda x : A_1. M \vdash \lambda x : (\tau_1.e)$, we know that $\Delta \vdash \lambda x : A_1 \vdash (\tau_1 \vdash \tau_2) \vdash \tau_A = \tau_1 \rightarrow \tau_2$. Using Lemma C.13 (uniqueness of type encoding) on $\Delta \vdash \lambda x : A_1. M \vdash \lambda x : (\tau_1 \vdash \tau_2)$ and 
  $\Delta \vdash \lambda x : A_1 \vdash (\tau_1 \vdash \tau_2) \vdash \text{we can conclude we have} \tau_1 = \tau_1'$.

- By application of the induction hypothesis on $\Delta; \text{dom}(\Omega) \vdash M \vdash e$, with the following auxiliary judgements $\Delta \vdash \Upsilon \vdash \{ x : A_1 \} \vdash \Gamma_1 \vdash \{ x : \tau_1 \}$ and $\Delta \vdash \Upsilon \vdash \{ x : A_1 \} \vdash \Gamma_2 \vdash \{ x : \tau_2 \}$ and 
  $\Omega; \Upsilon \vdash \{ x : A_1 \} \vdash M : A_2$, we have that $\Delta; \Gamma_1 \vdash \{ x : \tau_1 \} \vdash \Gamma_2 \vdash e : \tau_2$.
– By the typing rule for abstraction (\texttt{tp_abs}) on \(\Delta; \Gamma_1 \uplus \{ x : \tau_A \} \uplus \Gamma_2 \vdash e : \tau_2\) we can conclude \(\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \lambda x : \tau_1.e : \tau_1 \rightarrow \tau_2\).

Backward direction:

– From Lemma C.8 (inversion) on \(\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \lambda x : \tau_1.e : \tau_A\) we know that \(\Delta; \Gamma_1 \uplus \Gamma_2 \uplus \{ x : \tau_1 \} \vdash e : \tau_2\) where \(\Delta \vdash \tau_A \equiv_{\beta\eta} \tau_1 \rightarrow \tau_2 : \star\).

– From Lemma C.10 (inversion) on \(\Delta \vdash A \triangleright_\tau \tau_A\) and \(\Delta \vdash A_1 \triangleright_\tau \tau_1\) and \(\Delta \vdash \tau_A \equiv_{\beta\eta} \tau_1 \rightarrow \tau_2 : \star\) we know that \(\Delta \vdash A_2 \triangleright_\tau \tau_2\) where \(A = A_1 \rightarrow A_2\).

– Appealing to the induction hypothesis on \(\Delta; \text{dom}(\Omega) \vdash M \triangleright_\tau e\), with the auxiliary judgements \(\Delta \vdash \Theta \vdash \{ x : A_1 \} \vdash \Gamma_1 \uplus \{ x : \tau_1 \}\) and \(\Delta \vdash \Omega \vdash \Gamma_2\) and \(\Delta \vdash A_1 \triangleright_\tau \tau_2\) and \(\Delta; \Gamma_1 \uplus \Gamma_2 \uplus \{ x : \tau_1 \} \vdash e : \tau_2\) we have a derivation \(\Omega; \Upsilon \vdash \{ x : A_1 \} \vdash M : A_2\).

– Using the typing rule for abstraction (\texttt{tp_abs}) on \(\Omega; \Upsilon \vdash \{ x : A_1 \} \vdash M : A_2\) we have \(\Omega; \Upsilon \vdash \lambda x : A_1. \Theta \vdash \{ x : A_1 \} \vdash M : A_1 \rightarrow A_2\).

Case

\[
\Sigma(c) = B \rightarrow b \quad \Delta \vdash B \triangleright_\tau \tau_B
\]

\(\Delta; \text{dom}(\Omega) \vdash c \triangleright_\tau \lambda x : \tau_B. \text{roll}()\) on \(\text{inj}_{L(c)}\) x of \(\Sigma^*(\text{Rec} \Sigma^*)\)

\text{en_con}

Common:

– By Lemma C.16 (commutivity of parameterization and type encoding) on \(\Delta \vdash B \triangleright_\tau \tau_B\) we know that \(\tau_B = (\text{Rec} \Sigma^*) \tau(B)\).

– By Lemma C.7 (well-formedness of type encoding) we can conclude \(\Delta \vdash \{ x : \tau_B \}\). Using this fact, the variable typing rule (\texttt{tp_var}), the injection typing rule (\texttt{tp_variant}), and signature encoding (\texttt{en_sig}), \(\tau_B = (\text{Rec} \Sigma^*) \tau(B)\), and type equivalence we can conclude that \(\Delta; \{ x : \tau_B \} \vdash \text{inj}_{L(c)} x of \Sigma^*(\text{Rec} \Sigma^*)\) : \(\Sigma^*(\text{Rec} \Sigma^*)\).

– By Lemma C.18 (\texttt{roll} typing) we know that \(\emptyset; \emptyset \vdash \text{roll} : \forall \alpha. \Sigma^*(\text{Rec} \Sigma^* \alpha) \rightarrow \text{Rec} \Sigma^* \alpha\). By using the type application rule (\texttt{tp_app}), weakening, the typing rule for applications (\texttt{tp_app}), and finally the abstraction typing rule (\texttt{tp_abs}) along with weakening we can conclude \(\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \lambda x : \tau_B. \text{roll}()\) on \(\text{inj}_{L(c)}\) x of \(\Sigma^*(\text{Rec} \Sigma^*)\) : \(\tau_B \rightarrow \Sigma^* \tau\).

Forward direction:

– From inversion on \(\Omega; \Upsilon \vdash c : A\), we know that \(\Sigma(c) = B \rightarrow b\) and \(A = B \rightarrow b\). By inversion on \(\Delta \vdash B \rightarrow b \triangleright_\tau \tau_A\) we know that \(\Delta \vdash B \triangleright_\tau \tau_B^\prime\) and \(\Delta \vdash b \triangleright_\tau \tau_b^\prime\) where \(\tau_A = \tau_B^\prime \rightarrow \tau_b^\prime\). Using Lemma C.13 (uniqueness of type encoding) on \(\Delta \vdash B \triangleright_\tau \tau_B^\prime\) and \(\Delta \vdash B \triangleright_\tau \tau_B\) we have that \(\tau_B = \tau_B^\prime\). Finally, by inversion on \(\Delta \vdash b \triangleright_\tau \tau_b^\prime\) we have that \(\tau_b^\prime = \text{Rec} \Sigma^* \tau\). Therefore, \(\Delta \vdash A \triangleright_\tau \tau_B \rightarrow \text{Rec} \Sigma^* \tau\).

Backward direction:

– From inversion on \(\Omega; \Upsilon \vdash c : B\), and the typing rule for constants (\texttt{tp_con}) we can conclude \(\Omega; \Upsilon \vdash c : B \rightarrow b\).

– By using the rule for encoding functions (\texttt{en_tp_arrow}) on \(\Delta \vdash B \triangleright_\tau \tau_B\) and the axiom \(\Delta \vdash b \triangleright_\tau \text{Rec} \Sigma^* \tau \) (\texttt{en_tp_b}) we can conclude \(\Delta \vdash B \rightarrow b \triangleright_\tau \tau_B \rightarrow \text{Rec} \Sigma^* \tau\).

Case

\[
\Delta \vdash A \triangleright_\tau \tau_A \quad \Delta; \text{dom}(\Omega) \vdash \Theta \triangleright_\tau^\Theta e_\Theta \quad \Delta; \text{dom}(\Omega) \vdash M \triangleright_\tau e_M
\]

\(\Delta; \text{dom}(\Omega) \vdash \text{iter} \{ \square B, A | \Theta \} M \triangleright_\tau \text{iter} \{ B^* \} \tau_A | e_\Theta e_M\) \text{ en_iter}

Common:

– Lemma C.11 (type encodings is total and decidable) tells us that for \(\square B\) and \(\Delta \vdash \tau : \star\) we can construct a derivation \(\Delta \vdash \square B \triangleright_\tau \tau_B\).
The forward direction follows from straightforward use of induction:

- By inversion on $\Delta \vdash \Box B \triangleright_{\tau} \tau_B$ we know that $\tau_B = \forall \beta : \star \rightarrow \star; \tau'_B$ and $\Delta \uplus \{ \beta : \star \rightarrow \star \} \vdash B \triangleright_{\beta; \tau'_B}$. By Lemma C.16 (commutivity of parameterization and type encoding) we know that $\tau'_B = (\text{Rec } \Sigma^* \beta \tau)(B)$.

- By Lemma C.17 (commutivity of iteration types and type encoding) on $\Delta \vdash A \triangleright_{\tau_{\Lambda}} \tau_B$ we have that $\Delta \vdash A(B) \triangleright_{\tau} \tau_A(B)$.

The forward direction follows from straightforward use of induction:

- By inversion on $\Omega; \Upsilon \vdash \text{iter } \Box B, A|[\Theta] \ M : A'$ we know that $A' = A(B)$ and $\Omega; \Upsilon \vdash \Theta : A(\Sigma)$ and $\Omega; \Upsilon \vdash M : \Box B$.

- By appealing to the induction hypothesis on $\Omega; \Upsilon \vdash \Box; \text{dom}(\Omega) \vdash M \triangleright_{\tau} e_M$, with the expected auxiliary judgements, we have $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash e_M : \forall \beta : \star \rightarrow \star; \text{Rec } \Sigma^* \beta \tau(B)$ From this derivation, beta-equivalence (\text{tp}_\text{eqabs}_\text{beta}), and type equivalence (\text{tp}_\text{eq}) we can derive $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash e_M : \forall \beta : \star \rightarrow \star; \text{Rec } \Sigma^* \beta \tau(B)$.

- By appealing to the induction hypothesis on $\Omega; \Upsilon \vdash \Theta \triangleright_{\tau^\Delta} e_\Theta$, with the expected auxiliary judgements, we can conclude that $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash e_\Theta : \Sigma^* \tau_A \rightarrow \tau_A$.

- By Lemma C.19 (\text{iter} typing) we know that $\not\exists; \not\exists \vdash \text{iter } \{ B^* \} : \forall \gamma : \star; \not\exists : \star; (\text{Rec } \Sigma^* \beta \gamma) \rightarrow B^* \alpha$. Using type application (\text{tp}_\text{app}), weakening, the application typing rule (\text{tp}_\text{app}) twice, and type equivalence we can conclude $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{iter } \{ B^* \} [\tau]\{ \tau_A \} e_\Theta e_M : \tau_M : \tau_A(B)$.

Backward direction:

- By Lemma C.19 (\text{iter} typing) we know that $\not\exists; \not\exists \vdash \text{iter } \{ B^* \} : \forall \gamma : \star; \not\exists : \star; (\text{Rec } \Sigma^* \beta \gamma) \rightarrow B^* \alpha$. From repeated use of Lemma C.8 (inversion) on $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{iter } \{ B^* \} [\tau]\{ \tau_A \} e_\Theta e_M : \tau_M$, followed by type equivalence (\text{tp}_\text{eq}) we can conclude that $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash e_M : \forall \beta : \star \rightarrow \star; \text{Rec } \Sigma^* \beta \tau(B)$ and $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash e_\Theta : \Sigma^* \tau_A \rightarrow \tau_A$.

- Using the induction hypothesis on $\Omega; \Upsilon \vdash \Box; \text{dom}(\Omega) \vdash M \triangleright_{\tau} e_M$, with the expected auxiliary judgements, we can conclude $\Omega; \Upsilon \vdash M : \Box B$.

- Appealing to the induction hypothesis on $\Omega; \Upsilon \vdash \Theta \triangleright_{\tau^A} e_\Theta$, with the expected auxiliary judgments, gives a derivation $\Omega; \Upsilon \vdash \Theta : A(\Sigma)$.

- Using the iteration typing rule (\text{tp}_\text{iter}) on $\Omega; \Upsilon \vdash M : \Box B$ and $\Omega; \Upsilon \vdash \Theta : A(\Sigma)$ we know that $\Omega; \Upsilon \vdash \text{iter } \Box B, A|[\Theta] \ M : A(B)$.

Case

$$\Delta \vdash \Box \ A_1 \triangleright_{\tau} \tau_1 \quad \Delta; \text{dom}(\Omega) \vdash M_1 \triangleright_{\tau} e_1 \quad \Delta; \text{dom}(\Omega) \uplus \{ x : A_1 \} \vdash M_2 \triangleright_{\tau} e_2$$

From $\not\exists; \not\exists \vdash \text{let } \text{box } x : A_1 = M_1 \text{ in } M_2 \triangleright_{\tau} (\lambda x : \tau_1.e_2)e_1$, we can conclude $\Omega; \Upsilon \vdash \text{let } \text{box } x : A_1 = M_1 \text{ in } M_2 : A$.

The forward direction follows by straightforward use of induction:

- From inversion on $\Omega; \Upsilon \vdash \text{let } \text{box } x : A_1 = M_1 \text{ in } M_2 : A$ we can conclude $\Omega; \Upsilon \vdash \Theta : A(\Sigma)$ and $\Omega \uplus \{ x : A_1 \} ; \Upsilon \vdash M_2 : A$.

- Appealing to the induction hypothesis on $\Delta; \text{dom}(\Omega) \vdash M_1 \triangleright_{\tau} e_1$, and the expected auxiliary judgements, we have that $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash e_1 : \tau_1$. 30
Appealing to the induction hypothesis on $\Delta; \text{dom}(\Omega) \cup \{ x \} \vdash M_2 \triangleright_{\tau} e_2$, with the auxiliary judgements $\Delta \vdash Y \triangleright_{\tau} \Gamma_1$ and $\Delta \vdash \Omega \cup \{ x : A_1 \} \triangleright \Gamma_2 \cup \{ x : \tau_1 \}$ and $\Delta \vdash A \triangleright_{\tau_A}$ and $\Omega \cup \{ x : A_1 \}; \Upsilon \vdash M_2 : A$, we can produce a derivation $\Delta; \Gamma_1 \cup \Gamma_2 \cup \{ x : \tau_1 \} \vdash e_2 : \tau_A$.

Using the typing rules for abstraction (tp.abs) and application (tp.app) on $\Delta; \Gamma_1 \cup \Gamma_2 \cup \{ x : \tau_1 \} \vdash e_2 : \tau_A$ and $\Delta; \Gamma_1 \cup \Gamma_2 \cup e_1 : \tau_1$ we can conclude the desired result $\Delta; \Gamma_1 \cup \Gamma_2 \vdash (\lambda x : \tau_1. e_2) e_1 : \tau_A$.

Backward direction:

By repeated use of Lemma C.8 (inversion) on $\Delta; \Gamma_1 \cup \Gamma_2 \vdash (\lambda x : \tau_1. e_2) e_1 : \tau_A$ we have that $\Delta; \Gamma_1 \cup \Gamma_2 \vdash e_1 : \tau_1'$ and $\Delta; \Gamma_1 \cup \Gamma_2 \vdash \{ x : \tau_1 \} \vdash e_2 : \tau'$ where $\Delta \vdash \tau_A \equiv_{\beta\eta} \tau_2 : *$ and $\Delta \vdash \tau_1' \rightarrow \tau' \equiv_{\beta\eta} \tau_1 \rightarrow \tau_2 : *$.

By inversion on $\Delta \vdash \tau_1' \rightarrow \tau' \equiv_{\beta\eta} \tau_1 \rightarrow \tau_2 : *$ we can conclude $\Delta \vdash \tau_1' \equiv_{\beta\eta} \tau_1 : *$. Therefore, we have $\Delta; \Gamma_1 \cup \Gamma_2 \vdash e_1 : \tau_1'$ by type equivalence (tp.eq). Similarly for $\Delta; \Gamma_1 \cup \Gamma_2 \vdash \{ x : \tau_1 \} \vdash e_2 : \tau_A$.

Using the induction hypothesis on $\Delta; \text{dom}(\Omega) \vdash \Theta \triangleright_{\tau} e_1$, with the expected auxiliary judgements, we can conclude that $\Omega; \Upsilon \vdash M_1 : \square A_1$.

Again using the induction hypothesis on $\Delta; \text{dom}(\Omega) \cup \{ x \} \vdash M_2 \triangleright_{\tau} e_2$, with the auxiliary judgements $\Delta \vdash Y \triangleright_{\tau} \Gamma_1$ and $\Delta \vdash \Omega \cup \{ x : A_1 \} \triangleright \Gamma_2 \cup \{ x : \tau_1 \}$ and $\Delta \vdash A \triangleright_{\tau_A}$ and $\Delta; \Gamma_1 \cup \Gamma_2 \vdash \{ x : \tau_1 \} \vdash e_2 : \tau_A$, we have that $\Omega \cup \{ x : A_1 \}; \Upsilon \vdash M_2 : A$.

By using the typing rule for letbox (tp.let) on $\Omega; \Upsilon \vdash M_1 : \square A_1$ and $\Omega \cup \{ x : A_1 \}; \Upsilon \vdash M_2 : A$ we have the desired result $\Omega; \Upsilon \vdash \text{let box } x : A_1 = M_1 \text{ in } M_2 : A$.

**Cases** The remaining cases procedure by straightforward application of the induction hypothesis and inversion to the subderivations.

The case for $\Delta; \text{dom}(\Omega) \vdash \Theta \triangleright_{\tau} e_\Theta$

<table>
<thead>
<tr>
<th>Case</th>
<th>$\forall c_i \in \text{dom}(\Theta) \vdash \Delta; \text{dom}(\Omega) \vdash \Theta(c_i) \triangleright_{\tau} e_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Delta; \text{dom}(\Omega) \vdash \Theta \triangleright_{\tau} \lambda x : \Sigma^*<em>{\tau_A} \text{case } x \text{ of } \text{inj}</em>{\xi(c_i)} y_1 \text{ in } (e_1y_1) \ldots \text{inj}_{\xi(c_n)} y_n \text{ in } (e_ny_n)$</td>
</tr>
</tbody>
</table>

Common:

- Lemma C.11 (type encoding total and decidable) on tells us that we can construct $\Delta \vdash A(B_1 \rightarrow b) \triangleright_{\tau} \tau_i$ for each $B_i$, where $\Sigma(c_i) = B_i \rightarrow b$. From the definition of iteration types, $A(B_1 \rightarrow b) = A(B_i) \rightarrow A$. Therefore by inversion on $\Delta \vdash A(B_i) \rightarrow A \triangleright_{\tau} \tau_i$ we know that $\Delta \vdash A(B_i) \triangleright_{\tau} \tau_i$ and $\Delta \vdash A \triangleright_{\tau_A}$ where $\tau_i = \tau_i \rightarrow \tau_A$. Lemma C.13 (uniqueness of type encoding) on $\Delta \vdash A \triangleright_{\tau_A}$ and $\Delta \vdash A \triangleright_{\tau_A}$ tells us that $\tau_A = \tau_i$.

The forward direction follows by straightforward use of the induction:

- Using inversion on $\Omega; \Upsilon \vdash \Theta : A(\Sigma)$ we can conclude for all $c_i \in \text{dom}(\Sigma)$ that $\Omega; \Upsilon \vdash \Theta(c_i) : A(B_i \rightarrow b)$ where $\Sigma(c_i) = B_i \rightarrow b$.

- Appealing to the induction hypothesis on $\Delta; \text{dom}(\Omega) \vdash \Theta(c_i) \triangleright_{\tau} e_i$, with the auxiliary judgements $\Delta \vdash Y \triangleright_{\tau} \Gamma_1$ and $\Delta \vdash \Omega \cup \{ x : A_1 \} \triangleright \Gamma_2$ and $\Delta \vdash A(B_i \rightarrow b) \triangleright_{\tau} \tau_i$ and and $\Omega; \Upsilon \vdash \Theta(c_i) : A(B_i \rightarrow b)$, we have that $\Delta; \Gamma_1 \cup \Gamma_2 \vdash e_i : \tau_i \rightarrow \tau_A$.

- Using the typing rule for case (tp.case) and abstraction (tp.abs) allows us to conclude $\Delta; \Gamma_1 \cup \Gamma_2 \vdash \lambda x : \Sigma^*_{\tau_A} \text{case } x \text{ of } \text{inj}_{\xi(c_i)} y_1 \text{ in } (e_1y_1) \ldots \text{inj}_{\xi(c_n)} y_n \text{ in } (e_ny_n) : \Sigma^*_{\tau_A} \rightarrow \tau_A$. 31
Proof. By mutual induction over the structure of $\Delta \vdash \{ \alpha : * \rightarrow \star \}; \text{dom}(\Omega) \vdash M \triangleright_{\alpha \tau} e$ and $\Delta \vdash \{ \alpha : * \rightarrow \star \}; \text{dom}(\Omega) \vdash \Theta \triangleright_{\alpha \tau} e_\Theta$. The cases for the former:

Case

$$x \notin \text{dom}(\Omega)$$

$\Delta \vdash \{ \alpha : * \rightarrow \star \}; \text{dom}(\Omega) \vdash x \triangleright_{\alpha \tau} x$
By assumption we have that $\Delta \cup \{\alpha : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash x : \tau'$. Therefore by Lemma C.8 (inversion), we can conclude that $(x : \tau'' \in \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma')$, where $\Delta \cup \{\alpha : * \rightarrow *\} \vdash \tau' \equiv \beta_\eta \tau'' : *$. By the fact that $x \notin \text{dom}(\Omega)$ and Definition C.1 (environment encoding) we know that $x \notin \text{dom}(\Gamma)$. Furthermore that $x : \tau'' \in \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma'$. 

Given that the contexts are disjoint $x : \tau'' \in \Gamma_1$ or $x : \tau'' \in \Gamma_1$. By Lemma C.21 (type containment) on $\Delta \cup \{\alpha : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash x : \tau'$. Therefore by Lemma C.8 (inversion), we can conclude that $x : \tau'' \in \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma'$ where $\Delta \cup \{\alpha : * \rightarrow *\} \vdash \tau' \equiv \beta_\eta \tau'' : *$. However, we assumed that $\alpha \notin \text{FTV}(\Gamma_1)$, so $x : \tau''$ cannot be in $\Gamma_1$ and it must be the case that $x : \tau'' \in \Gamma_1$. Therefore, we are may conclude $\Delta \cup \{\alpha : * \rightarrow *\}; \Gamma \cup \Gamma' \vdash x : \tau'$ by the variable typing rule ($\text{tp}\_\text{var}$) followed by type equivalence ($\text{tp}\_\text{eq}$).

Case

$$x \in \text{dom}(\Omega)$$

$\Delta \cup \{\alpha : * \rightarrow *\}; \text{dom}(\Omega) \vdash x \triangleright_{\tau} x[\lambda \alpha : * . \tau]$ envars

By assumption we have that $\Delta \cup \{\alpha : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash x : \tau'$. Therefore by Lemma C.8 (inversion), we can conclude that $x : \tau'' \in \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma'$ where $\Delta \cup \{\alpha : * \rightarrow *\} \vdash \tau' \equiv \beta_\eta \tau'' : *$. However, we know that $x \in \text{dom}(\Omega)$ which means by Definition C.1 (environment encoding) that $x \in \text{dom}(\Gamma)$. Because the union of contexts must be disjoint, we are allowed to conclude $x : \tau'' \in \Gamma \cup \Gamma'$, by which the by the variable typing rule ($\text{tp}\_\text{var}$) and type equivalence ($\text{tp}\_\text{eq}$) we have $\Delta \cup \{\alpha : * \rightarrow *\}; \Gamma \cup \Gamma' \vdash x : \tau'$.

Case

$$\beta \notin \Delta \quad \Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \text{dom}(\Omega) \vdash M \triangleright_{\beta(\alpha \tau)} e$$

$\Delta \cup \{\alpha : * \rightarrow *\}; \text{dom}(\Omega) \vdash \text{box} M \triangleright_{\alpha \beta} \Lambda \beta : * \rightarrow \ast . e$ enbox

By inversion on $\Delta \cup \{\alpha : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash \Delta \beta : * \rightarrow \ast . e : \tau'$ we know that $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash e : \tau''$ where $\Delta \cup \{\alpha : * \rightarrow *\} \vdash \tau' \equiv \beta_\eta \forall \beta : * \rightarrow * . \tau'' : *$. By Lemma C.10 (inversion) on $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash e : \tau''$ and $\Delta \cup \{\alpha : * \rightarrow *\} \vdash \tau' \equiv \beta_\eta \forall \beta : * \rightarrow * . \tau'' : *$ we can conclude $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\} \vdash A' \triangleright_{\beta(\alpha \tau)} \tau_A'$ where $\Delta \cup \{\alpha : * \rightarrow *\} \vdash \tau_A' \equiv \beta_\eta \tau'' : *$. We have that $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash e : \tau''$ and $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\} \vdash \tau_A' \equiv \beta_\eta \tau'' : *$ by Definition C.1 (environment encoding).

Using type equivalence ($\text{tp}\_\text{eq}$) on $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash e : \tau''$ and $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\} \vdash \tau_A' \equiv \beta_\eta \tau'' : *$ we have that $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash e : \tau_A'$.

By use of the induction hypothesis, with respect to $\beta$, on $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \text{dom}(\Omega) \vdash M \triangleright_{\beta(\alpha \tau)} e$, with the auxiliary judgements $\Delta \cup \{\alpha : * \rightarrow *\} \vdash \gamma_j \triangleright_{\tau_j} \gamma_j$ (where we have added $\Gamma'$ to the set of contexts to be eliminated) and $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\} \vdash \Omega \triangleright_{\Gamma} \Gamma$ and $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\} \vdash \Theta \triangleright_{\beta(\alpha \tau)} \Theta$ and $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma' \vdash e : \tau''$ and $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\} \vdash A' \triangleright_{\beta(\alpha \tau)} \tau_A'$ we have that $\Delta \cup \{\alpha : * \rightarrow *\} \cup \{\beta : * \rightarrow *\}; \Gamma \vdash e : \tau_A'$.

By the type abstraction typing rule ($\text{tp}\_\text{abs}$), weakening, and type equivalence ($\text{tp}\_\text{eq}$) we have the desired result $\Delta \cup \{\alpha : * \rightarrow *\}; \Gamma \vdash \Lambda \beta : * \rightarrow \ast . e : \tau'$.
Case

\[ \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \text{dom}(\Omega) \vdash M \triangleright_{\alpha}^\tau_1 \ e \quad \Delta \uplus \{ \alpha : \star \rightarrow \star \} \vdash A_1 \triangleright_{\alpha}^\tau \ \tau_1 \ \text{en.abs} \]

- By Lemma C.8 (inversion) on \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \Gamma \uplus \Gamma_1 \uplus \ldots \uplus \Gamma_n \uplus \Gamma' \vdash \lambda x : \tau_1.e : \tau' \) we have that \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \Gamma \uplus \Gamma_1 \uplus \ldots \uplus \Gamma_n \uplus \Gamma' \uplus \{ x : \tau_1 \} \vdash e : \tau_2 \) and that \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} \vdash \tau' \equiv_{\beta\eta} \tau_1 \rightarrow \tau_2 : \star \).

- From Lemma C.10 (inversion) on \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} \vdash A \triangleright_{\alpha}^\tau \ \tau_1 \rightarrow \tau_2 \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} \vdash \tau' \equiv_{\beta\eta} \tau_1 \rightarrow \tau_2 : \star \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} \vdash A_1 \triangleright_{\alpha}^\tau \ \tau_1 \) we know \( A = A_1 \rightarrow A_2 \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} \vdash A_2 \triangleright_{\alpha}^\tau \ \tau_2 \).

- Therefore by application of the induction hypothesis to \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \text{dom}(\Omega) \vdash M \triangleright_{\alpha}^\tau \ e \) with the expected auxiliary judgements we can conclude \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \Gamma \uplus \Gamma' \vdash \lambda x : \tau_1.e : \tau' \).

- Using the abstraction typing rule (tp.abs) and type equivalence (tp.eq) we have the desired result, \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \Gamma \uplus \Gamma' \vdash \lambda x : \tau_1.e : \tau' \).

Case

\[ \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \text{dom}(\Omega) \vdash \square A_1 \triangleright_{\alpha}^\tau \ \tau_1 \quad \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \text{dom}(\Omega) \vdash M_1 \triangleright_{\alpha}^\tau \ e_1 \quad \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \text{dom}(\Omega) \vdash \{ x \} \vdash M_2 \triangleright_{\alpha}^\tau \ e_2 \ \text{en.letb} \]

- By Lemma C.8 (inversion) on \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \Gamma \uplus \Gamma_1 \uplus \ldots \uplus \Gamma_n \uplus \Gamma' \vdash (\lambda x : \tau_1.e_2)e_1 : \tau' \) we know that \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \Gamma \uplus \Gamma_1 \uplus \ldots \uplus \Gamma_n \uplus \Gamma' \uplus \{ x : \tau_1 \} \vdash e_2 : \tau'' \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \Gamma \uplus \Gamma_1 \uplus \ldots \uplus \Gamma_n \uplus \Gamma' \vdash \Gamma \uplus \{ x : \tau_1 \} \vdash \tau'' \equiv_{\beta\eta} \tau : \star \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} \vdash \tau' \equiv_{\beta\eta} \tau'' : \star \).

- Given \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} ; \emptyset \vdash A_1 \triangleright_{\alpha}^\tau \ \tau_1 \) and Definition C.1 (environment encoding) we know that \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} ; \emptyset \vdash \{ x : A_1 \} \vdash \Gamma \uplus \{ x : \tau_1 \} \).

- By inversion on \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} ; \emptyset \vdash \tau' \equiv_{\beta\eta} \tau_1 \rightarrow \tau'' : \star \) we have that \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} ; \emptyset \vdash \tau'' \equiv_{\beta\eta} \tau''' : \star \). Using this congruence and type equivalence (tp.eq) we can conclude on \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} ; \emptyset \vdash \tau'' : \star \). By application of the induction hypothesis to \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \text{dom}(\Omega) \uplus \{ x \} \vdash M_2 \triangleright_{\alpha}^\tau \ e_2 \), with the auxiliary judgements \( \Delta \vdash \Gamma_1 \triangleright_{\alpha}^\tau \ \Gamma_1 \) and \( \alpha \notin \text{FTV}(\Gamma_1) \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \emptyset \vdash \{ x : A_1 \} \vdash \Gamma \uplus \{ x : \tau_1 \} \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \emptyset \vdash \Gamma \vdash \Gamma' \triangleright_{\alpha}^\tau \ \Gamma' \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \emptyset \vdash \Gamma \uplus \{ x : \tau_1 \} \vdash e_2 : \tau' \) and \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} ; \emptyset \vdash A \triangleright_{\alpha}^\tau \ \tau' \) we have that \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} ; \emptyset \vdash \Gamma \uplus \{ x : \tau_1 \} \vdash e_2 : \tau' \).

- Similarly, using the induction on \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \text{dom}(\Omega) \vdash M_1 \triangleright_{\alpha}^\tau \ e_1 \), with the expected auxiliary judgements, \( \Delta \uplus \{ \alpha : \star \rightarrow \star \} ; \emptyset \vdash \lambda x : \tau_1.e_2 : \tau' \).

- Finally, using the typing rules for abstraction (tp.abs), application (tp.app), and type equivalence we have the desired conclusion \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \emptyset \vdash (\lambda x : \tau_1.e_2)e_1 : \tau' \).

Cases

The remaining cases follow by straightforward inversion and application of the induction hypothesis.

The case for \( \Delta \uplus \{ \alpha : \star \rightarrow \star \}; \text{dom}(\Omega) \vdash \Theta \triangleright_{\alpha}^\tau_1 \ e_{\Theta} \):
Proof.

\[ \Delta \vdash \{ \alpha : \star \rightarrow \star \}; \Gamma \vdash \tau_1 \equiv_{\beta_1} \tau_2 : \kappa, \] inversion lemmas by recognizing that type encoding always produces types in \( F \) or weak head atomic form with respect to \( \Delta \).

By Lemma C.16 (commutativity for iteration types) on and \( \Delta \vdash \{ \alpha : \star \rightarrow \star \}; \Gamma \vdash \tau \rightarrow \tau \) we can conclude \( \Delta \vdash \{ \alpha : \star \rightarrow \star \}; \Gamma \vdash \tau(B_i) \rightarrow \tau \) for each \( e_i \). Furthermore, we can strengthen the typing derivations to \( \Delta \vdash \{ \alpha : \star \rightarrow \star \}; \Gamma \vdash \tau(B_i) \rightarrow \tau \).

By Lemma C.16 (commutativity for application) on each \( \Delta \vdash \{ \alpha : \star \rightarrow \star \}; \Gamma \vdash \tau \rightarrow \tau \) we have the desired result
\[
\Delta \vdash \{ \alpha : \star \rightarrow \star \}; \Gamma \vdash \tau \rightarrow \tau \]

Since System \( F_\omega \) treats types identical up to the equivalence relation \( \Delta \vdash \tau \equiv_{\beta_1} \tau \equiv \kappa \), 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 \( F_\omega \) that are in weak head normal form.

Lemma C.5 (Type encodings are weak head normal forms). If \( \Delta \vdash \tau : \star \) and \( \Delta \vdash A \vdash_\tau \tau \) then \( \Delta \vdash \tau \rightarrow \star \).

Proof. By straightforward induction over the structure of \( \Delta \vdash A \vdash_\tau \tau \).

Lemma C.6 (Weak head types are well-formed types).

1. If \( \Delta \vdash \tau \vdash \star \) then \( \Delta \vdash \tau : \star \).
2. If \( \Delta \vdash \tau \vdash \star \) then \( \Delta \vdash \tau : \star \).

Proof. By trivial mutual induction over the structure of \( \Delta \vdash \tau \vdash \star \) and \( \Delta \vdash \tau \vdash \star \).

Lemma C.7 (Well-formedness of type encoding). If \( \Delta \vdash \tau : \star \) and \( \Delta \vdash A \vdash_\tau \tau \) then \( \Delta \vdash \tau \rightarrow \star \).

Proof. Follows directly from Lemma C.5 and Lemma C.6.

Lemma C.8 (Inversion on typing derivations).

1. If \( \Delta; \Gamma \vdash x : \tau \) then \( \Gamma(x) = \tau' \) where \( \Delta \vdash \tau \equiv_{\beta_1} \tau' : \star \).
2. If \( \Delta; \Gamma \vdash e_1 e_2 : \tau \) then \( \Delta; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \) and \( \Delta; \Gamma \vdash e_2 : \tau_1 \) where \( \Delta \vdash \tau \equiv_{\beta_1} \tau_2 : \star \).
3. If \( \Delta; \Gamma \vdash \lambda x : \tau_1 . e : \tau \) then \( \Delta; \Gamma \vdash \{ x : \tau_1 \} \vdash e : \tau' \) where \( \Delta \vdash \tau \equiv_{\beta_1} \tau_1 \rightarrow \tau' : \star \).
4. If \( \Delta; \Gamma \vdash A : \tau \) then \( \Delta \vdash \tau \equiv_{\beta_1} \forall \alpha : \star. (\alpha : \star) : \star \).
Proof. By straightforward induction over the number of uses of $\text{tp.eq}$ used before the final derivation step.

**Lemma C.9 (Inversion for type congruences).**

1. If $\Delta \vdash 1(\tau) \equiv \beta_\eta \tau' : *$ and $\Delta \vdash \tau' \parallel \tau$ then $\tau' = 1(\tau''')$ where $\Delta \vdash \tau \equiv \beta_\eta \tau''$.

2. If $\Delta \vdash \text{Rec} \Sigma^* \tau \equiv \beta_\eta \tau' : *$ and $\Delta \vdash \tau' \parallel \tau$ then $\tau' = \tau_1 \rightarrow \tau_2$ where $\Delta \vdash \Sigma^* \tau \rightarrow \tau \equiv \beta_\eta \tau_1 : *$ and $\Delta \vdash \tau \equiv \beta_\eta \tau_2 : *$.

3. If $\Delta \vdash \tau_1 \rightarrow \tau_2 \equiv \beta_\eta \tau' : *$ and $\Delta \vdash \tau' \parallel \tau$ then $\tau' = \tau_1' \rightarrow \tau_2'$ where $\Delta \vdash \tau_1 \equiv \beta_\eta \tau_1' : *$ and $\Delta \vdash \tau_2 \equiv \beta_\eta \tau_2' : *$.

4. If $\Delta \vdash \tau_1 \times \tau_2 \equiv \beta_\eta \tau' : *$ and $\Delta \vdash \tau' \parallel \tau$ then $\tau' = \tau_1' \times \tau_2'$ where $\Delta \vdash \tau_1 \equiv \beta_\eta \tau_1' : *$ and $\Delta \vdash \tau_2 \equiv \beta_\eta \tau_2' : *$.

5. If $\Delta \vdash \forall \alpha : \ast \rightarrow \ast, \tau \equiv \beta_\eta \tau' : *$ and $\Delta \vdash \tau' \parallel \tau$ then $\tau' = \forall \alpha : \ast \rightarrow \ast, \tau''$ where $\Delta \vdash \{ \alpha : \ast \rightarrow \ast \} ; \tau \rightarrow \tau''$.

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

**Lemma C.10 (Inversion for type encoding).**

1. If $\Delta \vdash A \triangleright_\tau \tau_A$ and $\Delta \vdash \tau_A \equiv \beta_\eta \text{Rec} \Sigma^* \tau : *$ then $A = b$.

2. If $\Delta \vdash A \triangleright_\tau \tau_A$ and $\Delta \vdash A_1 \triangleright_\tau \tau_1$ and $\Delta \vdash \tau_A \equiv \beta_\eta \tau_1 \rightarrow \tau_2 : *$ then $\Delta \vdash A_2 \triangleright_\tau \tau_2'$ where $A = A_1 \rightarrow A_2$ and $\tau_A = \tau_1 \rightarrow \tau_2'$.

3. If $\Delta \vdash A \triangleright_\tau \tau_A$ and $\Delta \vdash \tau_A \equiv \beta_\eta \forall \alpha : \ast \rightarrow \ast, \tau_A : *$ then $\Delta \vdash \{ \alpha : \ast \rightarrow \ast \} ; \tau' \equiv \beta_\eta \tau_A' : *$ and $A = \square A'$ and $\tau_A = \forall \alpha : \ast \rightarrow \ast, \tau''_A$.

4. If $\Delta \vdash A \triangleright_\tau \tau_A$ and $\Delta \vdash \tau_A \equiv \beta_\eta \ast(\tau') : *$ then $A = 1$.

5. If $\Delta \vdash A \triangleright_\tau \tau_A$ and $\Delta \vdash \tau_A \equiv \beta_\eta \tau_1 \times \tau_2 : *$ then $\Delta \vdash A \triangleright_\tau \tau_A' \equiv \beta_\eta \tau_1' \times \tau_2'$ 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 C.5 (type encodings are weak head normal) on $\Delta \vdash A \triangleright_\tau \tau_A$ we know that $\Delta \vdash \tau_A \parallel \tau$.

Using Lemma C.9 (inversion) on $\Delta \vdash \tau_A \equiv \beta_\eta \text{Rec} \Sigma^* \tau : *$ we know that $\tau_A = \tau_1 \rightarrow \tau_2$ where $\Delta \vdash \tau_1 \equiv \beta_\eta \Sigma^* \tau \rightarrow \tau : *$. Given that $\Delta \vdash A \triangleright_\tau \tau_1 \rightarrow \tau_2$, either $A = b$ or $A = A_1 \rightarrow A_2$ for some $A_1, A_2$.

- Assume that $A = A_1 \rightarrow A_2$. Then by inversion on $\Delta \vdash A_1 \rightarrow A_2 \triangleright_\tau \tau_1 \rightarrow \tau_2$ we have that $\Delta \vdash A_1 \triangleright_\tau \tau_1$. Using Lemma C.5 again on $\Delta \vdash A_1 \triangleright_\tau \tau_1$ we know that $\Delta \vdash \tau_A \parallel \tau$. Again by Lemma C.9 on $\Delta \vdash \tau_1 \equiv \beta_\eta \Sigma^* \tau \rightarrow \tau : *$ we have that $\tau_1 = \tau_1' \rightarrow \tau''$ where $\Delta \vdash \tau_1' \equiv \beta_\eta \Sigma^* \tau : *$ and $\Delta \vdash \tau_1'' \equiv \beta_\eta \tau : *$. As before $A_1 = b$ or $A_1 = A_1' \rightarrow A_1''$ for some $A_1', A_1''$. 

36
• Assume $A_1 = b$. Then $\tau_1' = \Sigma^* \tau \rightarrow \tau$ and $\tau_1'' = \tau$. However, $\Delta \vdash \Sigma^* \tau \rightarrow \tau \not= \beta_\eta \Sigma^* \tau : *$ as there is no way to make a variant and function equivalent.

• Therefore $A_1 = A'_1 \rightarrow A''_1$. By inversion on $\Delta \vdash A'_1 \rightarrow A''_1 \triangleright \tau \tau_1' \rightarrow \tau_1''$ we have that $\Delta \vdash A'_1 \triangleright \tau \tau_1'$. Using Lemma C.5 yet again we know that $\Delta \vdash \tau_1' \ntriangleright *. However, we know that $\Delta \vdash \tau_1' \equiv_{\beta \eta} \Sigma^* \tau : *$.

There are no types in the image of the encoding where the head constructor is equivalent to a variant. So our assumption that $A = A_1 \rightarrow A_2$ must be false, and $A = b$.

For Part 2:

• By Lemma C.5 (type encodings are weak head normal) on $\Delta \vdash A \triangleright \tau A$ we know that $\Delta \vdash \tau A \ntriangleright *$. Using Lemma C.9 (inversion) on $\Delta \vdash \tau A \equiv_{\beta \eta} \tau_1 \rightarrow \tau_2 : *$ we know that $\tau A = \tau_1' \rightarrow \tau_2'$ where $\Delta \vdash \tau_1 \equiv_{\beta \eta} \tau_1' : *$. Given that $\Delta \vdash A \triangleright \tau_1' \rightarrow \tau_2'$, either $A = b$ or $A = A'_1 \rightarrow A''_2$ for some $A'_1, A''_2$.

• Assume $A = b$. \[ ^{10} \]

Lemma C.11 (Type encoding is total and decidable). Given a type, $A$, in the source calculus and a $\tau$ in $F_\omega$ we can construct $\Delta \vdash A \triangleright \tau A$.

Proof. By straightforward induction over the structure of $A$. \[ \]

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 following lemma to also state that $\tau_1$ and $\tau_2$ must also be syntactically equivalent in addition to semantically equivalent using Lemma C.11, but it is not necessary for the proofs.

Lemma C.12 (Type encoding with congruent results). If $\Delta \vdash \tau : *$ and $\Delta \vdash A \triangleright \tau_1$ and $\Delta \vdash A_2 \triangleright \tau_2$ where $\Delta \vdash \tau_1 \equiv_{\beta \eta} \tau_2 : *$ then $A_1 = A_2$.

Proof. By induction over the structure of $\Delta \vdash A_1 \triangleright \tau_1$ using inversion on $\Delta \vdash A_2 \triangleright \tau_2$. \[ \]

Lemma C.13 (Uniqueness of type encoding).

1. If $\Delta \vdash A \triangleright \tau A$ and $\Delta \vdash A \triangleright \tau'_A$ then $\tau_A = \tau'_A$.

2. If $\Delta \vdash A \triangleright \tau$ and $\Delta \vdash A' \triangleright \tau$ then $A = A'$.

Proof. Both properties follow by straightforward simultaneous induction on the type encoding derivations. \[ \]

Lemma C.14 (Type encoding under congruent worlds). If $\Delta \vdash A \triangleright \tau_1 \tau A$ and $\Delta \vdash A \triangleright \tau_2 \tau'_A$ where $\Delta \vdash \tau_1 \equiv_{\beta \eta} \tau_2 : *$ then $\Delta \vdash \tau A \equiv_{\beta \eta} \tau'_A : *$

Proof. By straightforward simultaneous induction on the type encoding derivations. \[ \]

Lemma C.15 (World substitution for type encoding). If $\Delta \triangleright \{ \alpha : \tau \rightarrow * \} \vdash A \triangleright \alpha \tau A$ and $\Delta \vdash \tau : * \rightarrow *$ then $\Delta \vdash A \triangleright \alpha \tau A \{ \tau / \alpha \}$.

Proof. By straightforward induction over the structure of $\Delta \triangleright \{ \alpha : \tau \rightarrow * \} \vdash A \triangleright \alpha \tau A$. \[ \]

Lemma C.16 (Commutativity for parameterization and type encoding). If $\Delta \vdash B \triangleright \tau B$ then

1. $\tau_B = (\text{Rec } \Sigma^* \tau)(B)$.

\[ ^{10} \text{finish} \]
2. $\Delta \vdash \tau_B \equiv_{\beta\eta} B^* (\text{Rec } \Sigma^* \tau) : *$.

Proof. By straightforward induction over the structure of $\Delta \vdash B \triangleright \tau_B$.

Lemma C.17 (Commutativity for iteration types and type encoding). If $\Delta \vdash A \triangleright \tau_A$ then $\Delta \vdash A(B) \triangleright \tau_A(B)$.

Proof. By straightforward induction over the structure of $A(B)$.

Lemma C.18 (roll typing). $\emptyset; \emptyset \vdash \text{roll} : \forall \alpha : * . \Sigma^* \alpha \rightarrow \text{Rec } \Sigma^* \alpha$

Proof. By inspection of the definition in Figure 7 (Library Routines).

Lemma C.19 (iter typing). $\emptyset; \emptyset \vdash \text{iter} \{ | \tau : * \rightarrow * | \} : \forall \gamma : * . \forall \alpha : * . (\Sigma^* \alpha \rightarrow \alpha) \rightarrow (\forall \beta : * \rightarrow * . \tau(\text{Rec } \Sigma^* \beta \gamma)) \rightarrow \tau \alpha$

Proof. By inspection of the definition in Figure 7 (Library Routines).

Lemma C.20 (Encoding produces well-formed environments). Assume $\Delta \vdash \tau : *$.

1. If $\Delta \vdash \Upsilon \triangleright \Gamma_1$ then $\Delta \vdash \Gamma_1$.
2. If $\Delta \vdash \Omega \triangleright \Gamma_2$ then $\Delta \vdash \Gamma_2$.

Proof. Straightforward from the definitions and Lemma C.7.

Lemma C.21 (Type containment). Given a derivation $\Delta \vdash A \triangleright \tau_A$ we know that $\text{FTV}(\tau) = \text{FTV}(\tau_A)$.

Proof. By straightforward induction over the structure of $\Delta \vdash A \triangleright \tau_A$.

D Dynamic correctness

We prove the dynamic correctness of our encoding with respect to the equivalence relation $\Delta; \Gamma \vdash e \equiv_{\beta\eta} e' : \tau$ between target terms of type $\tau$. This congruence relation includes the standard $\beta$ and $\eta$-equivalences for functions, products and unit. The complete definition can be found in Appendix G.6. We will use the equals symbol, $=$, when we intend syntactic equality.

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

Definition D.1 (uniter).

$$\text{uniter} \{ \tau : * \rightarrow * \} : \forall \alpha : *(\Sigma^* \alpha \rightarrow \alpha) \rightarrow \tau \alpha \rightarrow \tau (\text{Rec } \Sigma^* \alpha)$$

$$\text{uniter} \{ \tau : * \rightarrow * \} = \Lambda \alpha : * . \Lambda f : \Sigma^* \alpha \rightarrow \alpha. \text{snd} (\text{xmap} \{ \tau \} \{ \text{Rec } \Sigma^* \alpha \} \{ \alpha \} (\text{cata} \{ \alpha | f \} \{ \text{place} \{ \alpha \} )))$$

Throughout the proofs in this section the following equivalences will be required many times, so for conciseness we state them all here.

Lemma D.2 (Properties of openiter and uniter). Assuming $\Delta \vdash \tau : *$ and $\Delta \vdash \tau' : *$.

1. $\Delta ; \{ f : \Sigma^* \tau' \rightarrow \tau' \} \vdash (\text{openiter} \{ \tau \} \{ \tau' \} f) \circ (\text{uniter} \{ \tau \} \{ \tau' \} f) \equiv_{\beta\eta} \lambda x : \tau \tau'. x : \tau \tau' \rightarrow \tau \tau'$
2. $\Delta ; \{ f : \Sigma^* \tau' \rightarrow \tau', e : b^* (\text{Rec } \Sigma^* \tau') \} \vdash \text{openiter} \{ b^* \} \{ \tau' \} f e \equiv_{\beta\eta} e f : \tau'$
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 $\text{openiter}$, $\text{xmap}$ and $\text{uniter}$.

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.

Definition D.3 (Well-formed dynamic replacements).

\[
\forall c_i \in \text{dom}(\Sigma) \quad \Sigma(c_i) = B_i \quad \Omega; \Psi \vdash \Theta(c_i) : A(B_i)
\]

\[
\forall x_i \in \text{dom}(\Psi) \quad \Psi(x_i) = B'_i \quad \Omega; \Psi \vdash \Theta(x_i) : A(B'_i)
\]

\[\Omega; \Theta : A(\Psi; \Sigma)\] 

Lemma D.4 (Well-typed replacements are well-formed dynamic replacements). If $\Omega; \Psi \vdash \Theta : A(\Sigma)$ then $\Omega; \Psi \vdash \Theta : A(\emptyset; \Sigma)$.

Proof. Follows trivially from the definitions. \[\square\]

Lemma D.5 (Typing for elimination).

1. If $\Psi \vdash V \downarrow B$ and $\Omega; \Psi \vdash \Theta : A(\Psi; \Sigma)$ then $\Omega; \Psi \vdash \langle A, \Psi, \Theta \rangle(V) : A(B)$.

2. If $\Psi \vdash V \downarrow B$ and $\Omega; \Psi \vdash \Theta : A(\Psi; \Sigma)$ then $\Omega; \Psi \vdash \langle A, \Psi, \Theta \rangle(V) : A(B)$.

3. If $\Psi \vdash x \downarrow B$ and $\Omega; \Psi \vdash \Theta : A(\Psi; \Sigma)$ then $\Omega; \Psi \vdash \Theta(x) : A(B)$.

4. If $\Psi \downarrow c \downarrow B \rightarrow b$ and $\Omega; \Psi \vdash \Theta : A(\Psi; \Sigma)$ then $\Omega; \Psi \vdash \Theta(c) : A(B \rightarrow b)$.
Proof. Parts 1 and 2 follow by mutual induction over the structure of $\Psi \vdash V \uparrow B$ and $\Psi \vdash V \downarrow B$. Parts 3 and 4 follow as corollaries.

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 $\Theta$. For these derived encodings we will use a black triangle, $\triangleright$, rather than an a white one, $\triangleright$, to help distinguish between them and the standard encodings. We create a substitution (notated $\Delta; \Theta; e_\Theta \triangleright S$) as follows:

Definition D.6 (Elimination Substitution).

\[
\begin{array}{l}
\text{sub_empty} \\
\Delta; \emptyset; \Theta; e_\Theta \triangleright S \\
\end{array}
\]

Lemma D.7 (Substitution application). If $\Delta; \Psi; e_\Theta \triangleright S$ and $\Psi(x) = B$ then $S(x) = \text{uniter}\{B^*\}[\tau_A] e_\Theta e'$ where $\Delta; \emptyset; \Theta(x) \triangleright e'$.

Proof. Straightforward induction on the structure of $\Delta; \Psi; e_\Theta \triangleright S$.

Lemma D.8 (Static correctness with substitution). If $\Delta; \emptyset; \triangleright \tau_A e$ and $\Psi \vdash V \uparrow B$ and $\Delta \vdash B \triangleright \tau_A$ and $\Delta; \Psi; e_\Theta \triangleright S$ then $\Delta; \emptyset; S(e) \vdash \tau_B$.

Proof. Follows from Theorem C.2 (static correctness, forward direction), Definition C.1 (environment encoding), Lemma D.7 (substitution application), and Lemma D.5 (elimination typing).

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

Definition D.9 (Encoding of elimination).

\[
\begin{array}{l}
\Psi \vdash V \uparrow B \\
\Delta \vdash A \triangleright \tau_A \\
\Delta; \Xi \vdash \Theta(x) \triangleright e' \\
\Delta; \emptyset; \triangleright \tau_A \\
\end{array}
\]

Lemma D.10 (Dynamic correctness of elimination). If $\Omega; \Upsilon : \Theta : A(\Psi ; \Sigma)$ and $\langle A, \Psi, \Theta \rangle(V) = M$ and $\Delta; \text{dom}(\Omega) \vdash \langle A, \Psi, \Theta \rangle(V) \triangleright \tau_A e$ and $\Delta; \text{dom}(\Omega) \vdash M \triangleright \tau e'$ and $\Delta \vdash \Upsilon \triangleright \Gamma_1$ and $\Delta \vdash \Upsilon \triangleright \Gamma_2$ then $\Delta; \Gamma_1 \triangleright \Gamma_2 \triangleright e \equiv_{\beta \eta} e' : B^* \tau_A$.

Proof. By induction on $\langle A, \Psi, \Theta \rangle(V)$.

Case

\[
\langle A, \Psi, \Theta \rangle(x) \triangleq \Theta(x)
\]

- By inversion on $\Delta; \text{dom}(\Omega) \vdash \langle A, \Psi, \Theta \rangle(x) \triangleright \tau_A e$ we know that $e = \text{openiter}\{B^*\}[\tau_A] e_\Theta S(e_v)$ and $\Delta; \Psi; e_\Theta \triangleright S$ and $\Delta \vdash A \triangleright \tau_A$ and and $\Psi \vdash x \uparrow B$.

- It follows from typing of atomic and canonical forms[24] on $\Psi \vdash x \uparrow B$ that $\emptyset; \Psi \vdash x : B$. By inversion on $\emptyset; \Psi \vdash x : B$ we know that $\Psi(x) = B$.

- By Lemma D.7 (elimination substitution application) on $\Delta; \Psi; e_\Theta \triangleright S$ and $\Psi(x) = B$ we have that $S(x) = \text{uniter}\{B^*\}[\tau_A] e_\Theta e'$ where $\Delta; \emptyset; \Theta(x) \triangleright e'$.
Using Lemma D.5 (typing for elimination) on $\Omega; \Upsilon \vdash \Theta : A(\Psi; \Sigma)$ and $(A, \Psi, \Theta)(x) = M$ and $\Psi \vdash x \uparrow B$ we have that $\Omega; \Upsilon \vdash M : A(B)$. By use of Lemma C.17 (commutativity of encoding on iteration types) on $\Delta \vdash A \triangleright_{\tau_A} \tau_A$ we can conclude $\Delta \vdash A(B) \triangleright_{\tau_A} \tau_A(B)$.

By using Theorem C.2 (static correctness, forward direction) on $\Delta; \emptyset \vdash M \triangleright_{\tau} \tau'$, with the auxiliary judgements $\Delta \vdash \emptyset \triangleright_{\tau} \Gamma_1$ and $\Delta \vdash \emptyset \triangleright_{\tau} \emptyset \vdash A(B) \triangleright_{\tau} \tau_A(B)$ and $\emptyset; \emptyset \vdash M : A(B)$, we have that $\Delta; \emptyset \triangleright_{\tau} \emptyset \vdash \tau'(B)$. By type equivalence (tp_eq) we know that $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \tau'(B) : B^*\tau_A$.

By Lemma D.2 (properties of iter, part 3) and $\emptyset; \emptyset \vdash (\text{opener}[B^*][\tau_A] e_{\Theta}) \circ (\text{uniter}[B^*][\tau_A] e_{\Theta}) \equiv_{\beta\eta} \lambda z : B^*\tau_A, z : B^*\tau_A \rightarrow B^*\tau_A$. By reflection (eq_refl) on $\Delta, \Gamma_1 \uplus \Gamma_2 \vdash \tau'(B) \triangleright_{\tau} \tau_A$ we know that $\Delta, \Gamma_1 \uplus \Gamma_2 \vdash \tau'(B) \triangleright_{\tau} \tau_A$. This equivalence along with weakening, congruence of application (eq_app), $\beta$-equivalence (eq_abs_beta), and transitivity (eq_trans) means we can conclude $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash (\text{opener}[B^*][\tau_A] e_{\Theta}) \circ (\text{uniter}[B^*][\tau_A] e_{\Theta}) \equiv_{\beta\eta} \tau'(B) : B^*\tau_A$. Consequently, by undoing the composition and using the definition of the substitution $S$, we have the desired result $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{opener}[B^*][\tau_A] e_{\Theta} [S(x) \equiv_{\beta\eta} \tau'(B)]$.

Case

By inversion on $\Delta; \emptyset \vdash \Theta : A(\Psi; \Sigma)$ we know that $\Delta; \emptyset \vdash \Theta : A(\Psi; \Sigma)$ and $\emptyset; \emptyset \vdash \emptyset \vdash \emptyset \vdash B$. By inversion on $\Psi : \Theta(c) \uplus B$ where $\Delta; \emptyset \vdash \emptyset \vdash B = B_1 \rightarrow b$ and $\Sigma(c_1) = B_1 \rightarrow b$.

By inversion on $\Delta; \emptyset \vdash \emptyset \vdash B$ we have that $\Psi \vdash c_1 \uparrow B$ where $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$ we have that $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$ we have that $\emptyset; \emptyset \vdash \emptyset \vdash B$.

From Lemma C.17 (commutativity for iteration types and type encoding) on $\Delta \vdash A \triangleright_{\tau_A} \tau_A$ and $\emptyset; \emptyset \vdash \emptyset \vdash B$ we have that $\Delta \vdash A(B_1 \rightarrow b) \triangleright_{\tau_A} \tau_A(B_1 \rightarrow b)$.

Theorem C.2 (static correctness, forward direction) on $\Delta; \emptyset \vdash \Theta(c) \uplus B$ and $\emptyset; \emptyset \vdash \emptyset \vdash B$, we have that $\Delta \vdash \emptyset \vdash A(B_1 \rightarrow b) \triangleright_{\tau_A} \tau_A(B_1 \rightarrow b)$ and $\emptyset; \emptyset \vdash \Theta(c) : A(B_1 \rightarrow b)$ gives us $\Delta; \emptyset \vdash \emptyset \vdash B \triangleright_{\tau_A} \tau_A \triangleright_{\tau_A} \tau_A$. Using type equivalence (tp_eq) this is the same as $\Delta; \emptyset \vdash \emptyset \vdash B \triangleright_{\tau_A} \tau_A$.

From Lemma D.2 (properties of iteration, part 8) on $\Delta; \emptyset \vdash \emptyset \vdash \emptyset \vdash B \triangleright_{\tau_A} \tau_A$, we can conclude $\Delta; \emptyset \vdash \emptyset \vdash \emptyset \vdash B \triangleright_{\tau_A} \tau_A$.

Using transitivity (eq_trans) with $\Delta; \emptyset \vdash \emptyset \vdash \emptyset \vdash \emptyset \vdash B \triangleright_{\tau_A} \tau_A$, we have the desired result $\Delta; \emptyset \vdash \emptyset \vdash \emptyset \vdash B \triangleright_{\tau_A} \tau_A$.

Case

By inversion on $\Delta; \emptyset \vdash \Theta : A(\Psi; \Sigma)$ we know that $\Delta; \emptyset \vdash \Theta : A(\Psi; \Sigma)$ and $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$ we have that $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$ we have that $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$ we have that $\emptyset; \emptyset \vdash \emptyset \vdash B$.

Case

By inversion on $\Delta; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$. By inversion on $\emptyset; \emptyset \vdash \emptyset \vdash B$.

$\text{fix}$
By inversion on $\Psi \vdash V_1 V_2 \uparrow B$ we have that $B = b$ and $\Psi \vdash V_1 V_2 \downarrow b$. Futhermore, by inversion on $\Psi \vdash V_1 V_2 \downarrow b$ we know that $\Psi \vdash V_1 \downarrow B_1 \to b$ and $\Psi \vdash V_2 \uparrow B_1$. We can conclude $\Psi \vdash V_1 \uparrow B_1 \to b$ by conversion from atomic to canonical form (canat).

By inversion to $\Delta; \text{dom}(\Omega) \vdash V_1 V_2 \triangleright_\tau \sigma \epsilon_\nu$ we know that $\epsilon_\nu = \epsilon'_\nu, \epsilon''_\nu$ and $\Delta; \text{dom}(\Omega) \vdash V_1 \triangleright_\tau \sigma \epsilon'_\nu$ and $\Delta; \text{dom}(\Omega) \vdash V_2 \triangleright_\tau \sigma \epsilon''_\nu$.

By inversion on $\Delta; \not\vdash M_1 M_2 \triangleright_\tau \epsilon'$ we have that $\epsilon' = \epsilon'_1 \epsilon'_2$ and $\Delta; \not\vdash M_1 \triangleright_\tau \epsilon'_1$ and $\Delta; \not\vdash M_2 \triangleright_\tau \epsilon'_2$.

Using $\Delta; \Psi; \Theta; \epsilon_\theta \triangleright_\tau^A S \text{ and } \Delta \vdash A \triangleright_\tau \tau_A$ and $\Delta; \text{dom}(\Omega) \vdash \Theta \triangleright_\tau^A \epsilon_\theta$ and $\Delta; \not\vdash V_1 \triangleright_\tau \epsilon'_\nu$ and $\Psi \vdash V_1 \uparrow B_1 \to b$ we can conclude $\Delta; \text{dom}(\Omega) \vdash \langle A, \Psi, \Theta \rangle (V_1) \triangleright_\tau^A \text{ opener} \{ (B_1 \to b)^* \} [\tau_A] \epsilon_\theta \ S(\epsilon'_\nu)$ by the definition of elimination encoding (en_elim).

Similarly, using $\Delta; \Psi; \Theta; \epsilon_\theta \triangleright_\tau^A S \text{ and } \Delta \vdash A \triangleright_\tau \tau_A$ and $\Delta; \text{dom}(\Omega) \vdash \Theta \triangleright_\tau^A \epsilon_\theta$ and $\Delta; \not\vdash V_2 \triangleright_\tau \epsilon''_\nu$ and $\Psi \vdash V_2 \uparrow B_1$ we can conclude $\Delta; \text{dom}(\Omega) \vdash \langle A, \Psi, \Theta \rangle (V_2) \triangleright_\tau^A \text{ opener} \{ B_1^* \} [\tau_A] \epsilon_\theta \ S(\epsilon''_\nu)$ by the definition of elimination encoding (en_elim).

By application of the induction hypothesis to $\Omega; \Upsilon \vdash \Theta : \langle A; \Psi ; \Sigma \rangle$ and $\langle A; \Psi, \Theta \rangle (V_1) = M_1$ and $\Delta; \text{dom}(\Omega) \vdash \langle A; \Psi, \Theta \rangle (V_1) \triangleright_\tau^A \text{ opener} \{ (B_1 \to b)^* \} [\tau_A] \epsilon_\theta \ S(\epsilon'_\nu)$ and $\Delta; \text{dom}(\Omega) \vdash M_1 \triangleright_\tau \epsilon'_1$ and $\Delta; \not\vdash \Omega \triangleright_\tau \Gamma_1$ and $\Delta; \not\vdash \Upsilon \triangleright_\tau \Gamma_2$ we can conclude $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{ opener} \{ (B_1 \to b)^* \} [\tau_A] \epsilon_\theta \ S(\epsilon'_\nu) \equiv_{\beta \eta} \epsilon'_1 : (B_1 \to b)^* A$.

Similarly for $\Omega; \Upsilon \vdash \Theta : \langle A; \Psi ; \Sigma \rangle$ and $\langle A; \Psi, \Theta \rangle (V_2) = M_2$ and $\Delta; \text{dom}(\Omega) \vdash \langle A; \Psi, \Theta \rangle (V_2) \triangleright_\tau^A \text{ opener} \{ B_1^* \} [\tau_A] \epsilon_\theta \ S(\epsilon''_\nu)$ and $\Delta; \text{dom}(\Omega) \vdash M_2 \triangleright_\tau \epsilon''_2$ $\Delta; \not\vdash \Omega \triangleright_\tau \Gamma_1$ and $\Delta; \not\vdash \Upsilon \triangleright_\tau \Gamma_2$ we know that that $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{ opener} \{ B_1^* \} [\tau_A] \epsilon_\theta \ S(\epsilon''_\nu) \equiv_{\beta \eta} \epsilon''_2 : B_1^* A$.

Finally, by Lemma D.11 (iteration on atomic applications) on $\Omega; \Upsilon \vdash \Theta : \langle A; \Psi ; \Sigma \rangle$ and $\Delta \vdash \Omega \triangleright_\tau \Gamma_1$ and $\Delta \vdash \Upsilon \triangleright_\tau \Gamma_2$ and $\Psi \vdash V_1 V_2 \downarrow b$ and $\Psi \vdash V_1 \downarrow B_1 \to b$ and $\Psi \vdash V_2 \uparrow B_1$ and $\Delta; \not\vdash V_1 \triangleright_\tau \epsilon'_\nu$ and $\Delta; \not\vdash V_2 \triangleright_\tau \epsilon''_\nu$ and $\Delta; \Psi; \Theta; \epsilon_\theta \triangleright_\tau^A S$ we have that

$$\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{ opener} \{ b^* \} [\tau_A] \epsilon_\theta \ S(\epsilon'_\nu, \epsilon''_\nu) \equiv_{\beta \eta} \epsilon'_1\epsilon''_2 : b^* A$$

By transitivity (eq_trans) and application congruence (eq_app) on $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{ opener} \{ b^* \} [\tau_A] \epsilon_\theta \ S(\epsilon'_\nu, \epsilon''_\nu) \equiv_{\beta \eta} \epsilon'_1\epsilon''_2 : b^* A$.

$$\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{ opener} \{ B_1^* \} [\tau_A] \epsilon_\theta \ S(\epsilon''_\nu) \equiv_{\beta \eta} \epsilon''_2 : B_1^* A$$

We can then use the cons rule for elimination substitutions (sub_cons) on $\Delta; \not\vdash x' \triangleright_\tau x'$ and $\Delta; \Psi; \Theta; \epsilon_\theta \triangleright_\tau^A S$ to conclude $\Delta; \Psi \cup \{ x : B_1 \} \cup \{ x \to x' \} ; \epsilon_\theta \triangleright_\tau^A S \cdot \text{ uniter} \{ B_1^* \} [\tau_A] \epsilon_\theta \ x' / x$.

By inversion on $\Psi \vdash \lambda x : B_1. V \uparrow B$ we have that $B = B_1 \to B_2$ and $\Psi \cup \{ x : B_1 \} \vdash V \uparrow B_2$.

By inversion on $\Delta; \not\vdash \lambda x : B_1. V \triangleright_\tau \epsilon_\nu$ we know that $\epsilon_\nu = \lambda x : \tau B. \epsilon'_\nu$ and $\Delta; \not\vdash \lambda x \triangleright_\tau \epsilon'_\nu$ and $\Delta \vdash B_1 \triangleright_\tau \tau B$.  

\[ \langle A, \Psi \cup \{ x : B_1 \}, \Theta \cup \{ x \to x' \} \rangle (V) \triangleq M \]

\[ (A, \Psi, \Theta)(\lambda x : B_1. V) \triangleq \lambda x' : A(B_1). M \]  

\[ \text{elJam} \]

\[ \text{By inversion on } \Delta; \text{dom}(\Omega) \vdash \langle A, \Psi, \Theta \rangle (\lambda x : B_1. V) \triangleright_\tau^A e \text{ we know that } e = \text{ opener} \{ B_1^* \} [\tau_A] \epsilon_\theta \ S(\epsilon''_\nu) \text{ and } \Delta; \Psi; \Theta; \epsilon_\theta \triangleright_\tau^A S \text{ and } \Delta; \not\vdash \lambda x : B_1. V \triangleright_\tau \epsilon_\nu \text{ and } \Delta \vdash \lambda x : \tau B. \epsilon'_\nu \text{ and } \Delta; \not\vdash \lambda x \triangleright_\tau \epsilon'_\nu \text{ and } \Delta \vdash B_1 \triangleright_\tau \tau B. \]

\[ \text{By inversion on } \Delta; \not\vdash \lambda x : B_1. V \uparrow B \text{ we have that } B = B_1 \to B_2 \text{ and } \Psi \cup \{ x : B_1 \} \vdash V \uparrow B_2. \]

\[ \text{By inversion on } \Delta; \not\vdash \lambda x : B_1. V \triangleright_\tau \epsilon_\nu \text{ we know that } \epsilon_\nu = \lambda x : \tau B. \epsilon'_\nu \text{ and } \Delta; \not\vdash \lambda x \triangleright_\tau \epsilon'_\nu \text{ and } \Delta \vdash B_1 \triangleright_\tau \tau B. \]
By the definition of replacement encoding (en_rep) $\Delta; \text{dom}(\Omega) \vdash (x \mapsto x') \triangleright A e_{\Theta}$ is equivalent to $\Delta; \text{dom}(\Omega) \vdash \Theta \cup \{x \mapsto x'\} \triangleright A e_{\Theta}$.

Using all of these facts, we can use the encoding of eliminations (en_elim) on $\Psi \cup \{x : B_1\} \vdash V \upharpoonright B_2$ and $\Delta; \emptyset \vdash V \triangleright A e_{\Theta}$ and $\Delta; \emptyset \vdash A \triangleright A e_{\Theta}$ and $\Delta; \Psi \cup \{x \mapsto x'\} \triangleright S \cdot \{\text{uniter}(B_1)|[\tau_A| e_{\Theta} x'/x]\}$ and $\Delta; \text{dom}(\Omega) \vdash \Theta \cup \{x \mapsto x'\} \triangleright A e_{\Theta}$ to conclude

$$\Delta; \text{dom}(\Omega) \vdash (A, \Psi \cup \{x : B_1\}, \Theta \cup \{x \mapsto x'\}) \triangleright A$$

openiter($B_2|\tau_A| e_{\Theta} (S \cdot \{\text{uniter}(B_1)|[\tau_A| e_{\Theta} x'/x\}))(e'_\lambda)$

By inversion on $\Delta; \text{dom}(\Omega) \vdash \lambda x : A(B).M \triangleright e'$ we know that $e' = \lambda x : \tau'_A e''$ and $\Delta \vdash A(B_1) \triangleright \tau'_A$ and $\Delta; \text{dom}(\Omega) \vdash M \triangleright e''$. By use of Lemma C.17 (communitivity of encoding on iteration types) on $\Delta \vdash A(B_1) \triangleright \tau_A$ we can conclude $\Delta \vdash A(B_1) \triangleright \tau_A B_1)$. Therefore, by Lemma C.13 (uniqueness of type encoding) that $\tau'_A = \tau_A B_1$.

By the typing rule for variables (tp_var) we can conclude $\Omega; \Psi \cup \{x' : A(B_1)\} \vdash x' : A(B_1)$. Using this derivation, weakening, and Definition D.3 (well-formed dynamic replacements) on $\Omega; \Psi \cup \{x' : A(B_1)\} \vdash \Theta \cup \{x \mapsto x'\} : A(\Psi \cup \{x : B_1\}; \Sigma)$.

We have $\Delta \vdash \Psi \triangleright \Gamma_2$ and $\Delta \vdash A(B_1) \triangleright \tau_A B_1)$ so by Definition C.1 (environment encoding) we can conclude $\Delta \vdash \Psi \cup \{x' : A(B_1)\} \triangleright \Gamma_2 \vdash \{x : \tau_A B_1\}$.

Therefore, by induction on $\Omega; \Psi \cup \{x' : A(B_1)\} \vdash \Theta \cup \{x \mapsto x'\} : A(\Psi \cup \{x : B_1\}; \Sigma)$ and $\Delta; \text{dom}(\Omega) \vdash M \triangleright \tau_A e_{\Theta}$ and $\Delta; \text{dom}(\Omega) \vdash M \triangleright e''$. By use of Lemma C.17 (communitivity of encoding on iteration types) on $\Delta \vdash A(B_1) \triangleright \tau_A$ we can conclude $\Delta; \text{dom}(\Omega) \vdash M \triangleright \tau_A e_{\Theta}$ and $\Delta \vdash A(B_1) \triangleright \tau_A$. We can conclude

$$\Delta; \text{dom}(\Omega) \vdash \Delta_1 \cup \Delta_2 \vdash \lambda x' : \tau_A B_1).\text{openiter}(B_2|\tau_A| e_{\Theta} (S \cdot \{\text{uniter}(B)|[\tau_A| e_{\Theta} x'/x\}))(e'_\sim)(e'' \equiv \beta_\eta : B_2^* \tau_A)$$

By the term congruence rule for abstraction (eq.abs) and type equivalence (eq.tp_eq) we can conclude

$$\Delta_1 \cup \Delta_2 \vdash \lambda x' : \tau_A B_1).\text{openiter}(B_2|\tau_A| e_{\Theta} (S \cdot \{\text{uniter}(B)|[\tau_A| e_{\Theta} x'/x\}))(e'_{\sim})(\beta_\eta : \tau_A B_1).e'' : (B_1 \rightarrow B_2)^* \tau_A$$

By pulling out the substitution $\{\text{uniter}(B)|[\tau_A| e_{\Theta} x'/x\}, \beta$-equivalence (eq.abs.beta) and Lemma D.2 (properties of iteration, part 3) we can conclude the desired result

$$\Delta; \text{dom}(\Omega) \vdash \Delta_1 \cup \Delta_2 \vdash \text{openiter}(B_2|\tau_A| e_{\Theta} (S(e'_1))(\text{openiter}(B_2|\tau_A| e_{\Theta} S(e_2)) : B^* \tau_A$$

Cases: The remaining cases are uncomplicated uses of the induction hypothesis and congruences.

\[\square\]
Proof. We cannot prove this lemma directly, but it follows from the more general Lemma D.22 (Iteration and atomic forms).

To generalize the induction hypothesis of Lemma D.11 sufficently requires the introduction of formal machinery we will call iteration contexts. Iteration contexts provide convienient 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 D.12 (Iteration contexts).

\[
E ::= \bullet \mid E\{\bullet\, e\} \mid E\{\text{fst}\, \bullet\} \mid E\{\text{snd}\, \bullet\}
\]

\[
D ::= \bullet \mid B \rightarrow D \mid D \times B \mid B \times D
\]

\[
C ::= \bullet \mid A \rightarrow C \mid C \times A \mid A \times C
\]

Because of our universal usage of the asterisk type constructor notation, \(B^*\), for pure source language types, it proves convienient 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, analagous to iterated types in the source language.

Definition D.13 (Iteration context algebra).

\[
\begin{align*}
\bullet\{\bullet\, e\} &\triangleq e \\
E\{\text{fst}\, \bullet\, e\} &\triangleq e' \\
E\{\text{snd}\, \bullet\, e\} &\triangleq e'
\end{align*}
\]

Definition D.14 (Context type algebra).

\[
\begin{align*}
\bullet\{A\} &\triangleq A \\
(C \times A')\{A\} &\triangleq A' \times A' \\
(A' \times C)\{A\} &\triangleq A' \times A'' \\
A(\bullet) &\triangleq \bullet \\
A(B \rightarrow D) &\triangleq A(B) \rightarrow C \\
A(B \times D) &\triangleq A(B) \times C \\
A(D) &\triangleq C
\end{align*}
\]

Definition D.15 (Iterated context types).

\[
\begin{align*}
A(D) &\triangleq C \\
A(B \rightarrow D) &\triangleq A(B) \rightarrow C \\
A(B \times D) &\triangleq A(B) \times C \\
A(D) &\triangleq C
\end{align*}
\]

Definition D.16 (Context typing rules).

\[
\begin{align*}
\Delta; \Gamma \vdash \bullet : \bullet &\quad \text{ctp\_bullet} \\
\Delta; \Gamma \vdash E : C &\quad \text{ctp\_fist} \\
\Delta; \Gamma \vdash E\{\text{snd}\, \bullet\} : A \times C &\quad \text{ctp\_snd}
\end{align*}
\]

\[
\begin{align*}
\Delta; \Gamma \vdash \bullet : \bullet &\quad \text{ctp\_bullet} \\
\Delta; \Gamma \vdash \bullet : \bullet &\quad \text{ctp\_fist} \\
\Delta; \Gamma \vdash E\{\text{snd}\, \bullet\} : A \times C &\quad \text{ctp\_snd}
\end{align*}
\]
Finally, we define a formalism to describe the result of iteration over an iteration context.

**Definition D.17 (Iterated contexts).**

\[
\begin{align*}
\Delta; \Gamma \vdash e_\Theta : \Sigma^* \tau & \rightarrow \tau \quad \text{itr.bullet} \\
\Delta; \Gamma \vdash \bullet \openiter \ e_\Theta & \rightarrow \tau \quad \text{itr.fst} \\
\Delta; \Gamma \vdash E \openiter \bullet & \rightarrow \tau \\
\Delta; \Gamma \vdash E \{\text{snd} \bullet\} \openiter \bullet & \rightarrow \tau \\
\end{align*}
\]

**Lemma D.18 (Iterated context typing).** If \( \Delta; \Gamma \vdash E : D \) and \( \Delta; \Gamma \vdash A \triangleleft \tau_A \) and \( \Delta; \Gamma \vdash E \openiter e_\Theta \rightarrow \tau \) then \( \Delta; \Gamma \vdash E' : A(D) \).

**Proof.** By induction over the structure \( \Delta; \Gamma \vdash E : D \).

The following two lemmas lift congruence to iteration contexts.

**Lemma D.19 (Congruence under iteration contexts).** If \( \Delta; \Gamma \vdash E : C \) and \( \Delta; \Gamma \vdash e_1 \equiv_{\beta\eta} e_2 : C \{B\}^* \tau \) then \( \Delta; \Gamma \vdash E\{e_1\} \equiv_{\beta\eta} E\{e_2\} : B^* \tau \).

**Proof.** By induction over the structure of \( \Delta; \Gamma \vdash E : C \).

**Lemma D.20 (Congruence under iterated contexts).** If \( \Delta; \Gamma \vdash E : A(D) \) and \( \Delta; \Gamma \vdash e_1 \equiv_{\beta\eta} e_2 : D \{B\}^* \tau_A \) and \( \Delta; \Gamma \vdash A \triangleleft \tau_A \) then \( \Delta; \Gamma \vdash E\{e_1\} \equiv_{\beta\eta} E\{e_2\} : B^* \tau_A \).

**Proof.** By induction over the structure of \( \Delta; \Gamma \vdash E : A(B) \).

**Lemma D.21 (Lifting right inverse property to iteration contexts).** If \( \Delta; \Gamma \vdash e_\Theta : \Sigma^* \tau_A \rightarrow \tau_A \) and \( \Delta; \Gamma \vdash E : D \) and for all \( \Delta; \Gamma \vdash e' : D \{B\}^* \tau_A \),
\[
\Delta; \Gamma \vdash \text{openiter} \{B^*\} \{\tau_A\} e_\Theta \left( E \left( \text{uniter} \{D \{B\}^* \{\tau_A\} e_\Theta\{e'\}\right)\right) \equiv_{\beta\eta} E\{e'\} : B^* \tau_A \]
where \( \Delta; \Gamma \vdash E \openiter e_\Theta \rightarrow \tau \).

**Proof.** We do this by induction on \( \Delta; \Gamma \vdash E : D \).

**Case**

\[
\Delta; \Gamma \vdash \bullet : \bullet \quad \text{ctp.bullet}
\]

- Assume an arbitrary \( \Delta; \Gamma \vdash e' : \bullet \{B\}^* \tau_A \).
- Given the syntactic equivalence for context bullet types (tag.bullet) on \( \Delta; \Gamma \vdash e' : \bullet \{B\}^* \tau_A \), we can conclude \( \Delta; \Gamma \vdash e' : B^* \tau_A \).
- Using Lemma D.2 (properties of iteration, part 1), congruence of substitution (eqsubst) on \( \Delta; \Gamma \vdash e_\Theta : \Sigma^* \tau_A \rightarrow \tau_A \), and congruence of application (eqapp) on \( \Delta; \Gamma \vdash e' : B^* \tau_A \) gives us \( \Delta; \Gamma \vdash \text{openiter} \{B^*\} \{\tau_A\} e_\Theta \left( \text{uniter} \{B^*\} \{\tau_A\} e_\Theta\{e'\}\right) \equiv_{\beta\eta} e' : B^* \tau_A \) The desired result follows from follows from the syntactic equivalence for bullet iteration contexts (cat.bullet).
- From the rule for iterated bullet contexts (itr.bullet) and \( \Delta; \Gamma \vdash e_\Theta : \Sigma^* \tau_A \rightarrow \tau_A \) we have that \( \Delta; \Gamma \vdash \bullet \openiter e_\Theta \rightarrow \tau \).

**Case**

\[
\Delta; \Gamma \vdash E : D \\
\Delta; \Gamma \vdash E \{\text{fst} \bullet\} : D \times B_1 \quad \text{ctp.fst}
\]

\[^{12}\text{Something here}\]
Assume an arbitrary $\Delta; \Gamma \vdash e' : \langle D \times B_1 \rangle \{B\}^* \tau_A$. From syntactic equivalence of product context types (tag_prod,left) and type equivalence (tp_eq) we have that $\Delta; \Gamma \vdash e' : \langle D \{B\}^* \tau_A \rangle \times (B_1^* \tau_A)$.

By application of the induction hypothesis to $\Delta; \Gamma$ we have that for all $\Delta; \Gamma \vdash e'' : \langle D \{B\}^* \tau_A \rangle$, $\Delta; \Gamma \vdash E : D$ we have that

$$\Delta; \Gamma \vdash \text{fst } e'' : D \{B\}^* \tau_A.$$

For which we obtain $\Delta; \Gamma \vdash \text{fst } e' : D \{B\}^* \tau_A$ allowing us to conclude

$$\Delta; \Gamma \vdash \text{openiter } B^* \{\tau_A \in \text{fst } e' \} : B^* \tau_A.$$

From Lemma D.2 (properties of iteration, part 6), congruence of substitution (eq subst) on $\Delta; \Gamma \vdash e : \Sigma^* \tau_A \rightarrow \tau_A$ and $\Delta; \Gamma \vdash e' : \langle D \{B\}^* \tau_A \rangle \times (B_1^* \tau_A)$, and $\beta$-equivalence for products (eq_pair_beta1) we have that

$$\Delta; \Gamma \vdash \text{fst } e' : D \{B\}^* \tau_A$$

Using Lemma D.19 (congruence for iteration contexts) on $\Delta; \Gamma \vdash E : D$ we can conclude

$$\Delta; \Gamma \vdash \text{openiter } B^* \{\tau_A \in \text{fst } e' \} : B^* \tau_A.$$

Using this equivalence along with the congruence of application (eq_app) and transitivity (eq_trans) we can conclude

$$\Delta; \Gamma \vdash \text{openiter } B^* \{\tau_A \in \text{fst } e' \} : B^* \tau_A.$$

Finally using the syntactic equivalence of projection iteration contexts (cag fst) and the syntactic equivalence of context product types we have the desired result

$$\Delta; \Gamma \vdash \text{openiter } B^* \{\tau_A \in \text{fst } e' \} : B^* \tau_A.$$

Given that $\Delta; \Gamma \vdash E \triangleright^\tau_A \tau_A e' \; E'$ we can conclude $\Delta; \Gamma \vdash E \{\text{fst } e' \} \triangleright^\tau_A \tau_A e' \; E'$ by the rule for iterated projection contexts (itc fst).

**Case** The case for ctp_snd is symmetric to ctp fst.

**Case**

$$\Delta; \Gamma \vdash e : D \quad \Delta; \Gamma \vdash B_1 \triangleright^\tau_B \; \Delta; \Gamma \vdash e : \tau_B \quad \Delta; \Gamma \vdash e : \tau_B \rightarrow D$$

-- Assume an arbitrary $\Delta; \Gamma \vdash e' : (B_1 \rightarrow D) \{B\}^* \tau_A$. Using the rule for syntactic equivalence of context function types (tag_arrow) and type equivalence (tp eq) we know that $\Delta; \Gamma \vdash e' : (B_1^* \tau_A) \rightarrow (D \{B\}^* \tau_A)$.

From Lemma C.16 (commutativity for parameterization and type encoding) on $\Delta \vdash B_1 \triangleright^\tau_B B_2$ we can conclude that $\Delta; \Gamma \vdash \tau_B \equiv_{\beta \eta} B_1^*(\text{Rec } \Sigma^* \tau_A) : \tau$. Therefore, by type equivalence (eq tp) we have that $\Delta; \Gamma \vdash e : B_1^*(\text{Rec } \Sigma^* \tau_A)$. Consequently, using the typing rules for type and term application (tp_app, tp app) we can conclude $\Delta; \Gamma \vdash \text{openiter } B_1^* \{\tau_A \in \text{fst } e \} : B_1^* \tau_A$. Finally by using the typing rule for application on $\Delta; \Gamma \vdash e' : (B_1^* \tau_A) \rightarrow (D \{B\}^* \tau_A)$ we have that $\Delta; \Gamma \vdash e'(\text{openiter } B_1^* \{\tau_A \in \text{fst } e \} : D \{B\}^* \tau_A)$. 46
By application of the induction hypothesis to $\Delta; \Gamma \vdash e_\Theta : \Sigma^* \tau \to \tau_A$ and $\Delta; \Gamma \vdash E : D$ we know that for all $\Delta: \Gamma \vdash e'' : D\{B_2\}^*\tau_A$

$\Delta; \Gamma \vdash \text{openiter}(B_2^\tau)[\tau_A] e_\Theta(E\{\text{uniter}(D\{B_2\}^*\tau_A) e_\Theta e''\}) \equiv_{\beta\eta} E'(e'') : B_2^\tau_A$ where

$\Delta; \Gamma \vdash E \triangleright^\tau e'$. By instantiating this derivation with

$\Delta; \Gamma \vdash e'(\text{openiter}(B_1^\tau)[\tau_A] e_\Theta e) : D\{B\}^*\tau_A$ we have a derivation

$\Delta; \Gamma \vdash \text{openiter}(B^\tau)[\tau_A] e_\Theta(E\{\text{uniter}(D\{B\}^*\tau_A) e_\Theta (e'(\text{openiter}(B_1^\tau)[\tau_A] e_\Theta e))) \equiv_{\beta\eta}$

$E'(e'(\text{openiter}(B_1^\tau)[\tau_A] e_\Theta e)) : B^\tau_A$

From Lemma D.2 (properties of iteration, part 4), congruence of substitution (eq\_subst) on $\Delta; \Gamma \vdash e_\Theta : \Sigma^* \tau \to \tau_A$ and $\Delta; \Gamma \vdash e' : (B_1^\tau \tau_A) \to (D\{B\}^*\tau_A)$, congruence for application (eq\_app), and $\beta$-equivalence for abstractions (eq\_abs\_beta), and the syntactic equivalence of context function types (tag\_arrow) we have that

$\Delta; \Gamma \vdash (\text{uniter}((B_1 \to D)\{B\}^*\tau_A) f e')e \equiv_{\beta\eta}$

$\text{uniter}(D\{B\}^*\tau_A) e_\Theta (e'(\text{openiter}(B_1^\tau)[\tau_A] e_\Theta e)) : D\{B\}^*(\text{Rec } \Sigma^* \tau_A)$

From this congruence, Lemma D.19 (congruence for iteration contexts) on $\Delta; \Gamma \vdash E : D$, and congruence on application (eq\_app) we can conclude

$\Delta; \Gamma \vdash \text{openiter}(B^\tau)[\tau_A] e_\Theta(E\{\text{uniter}((B_1 \to D)\{B\}^*\tau_A) f e')e) \equiv_{\beta\eta}$

$E'(e'(\text{openiter}(B_1^\tau)[\tau_A] e_\Theta e)) : B^\tau_A$

Using our algebra on iteration contexts, we can pull out the applications (cag\_app) to produce our desired result

$\Delta; \Gamma \vdash \text{openiter}(B^\tau)[\tau_A] e_\Theta(E\{\text{uniter}((B_1 \to D)\{B\}^*\tau_A) f e')e) \equiv_{\beta\eta}$

$E'(\text{openiter}(B_1^\tau)[\tau_A] e_\Theta e) : B^\tau_A$

From $\Delta; \Gamma \vdash E \triangleright^\tau e'$ and the rule for iterated application contexts (itc\_app) on $\Delta; \Gamma \vdash e : B_1^\tau(\text{Rec } \Sigma^* \tau_A)$ we can conclude $\Delta; \Gamma \vdash E\{e\} \triangleright^\tau e' : B^\tau_A$

\[\square\]

**Lemma D.22 (Iteration and atomic forms).** If $\Psi \vdash V \downarrow B_2$ and $\Delta \vdash \Omega \uparrow \Gamma_1$ and $\Delta \vdash \Upsilon \uparrow \Gamma_2$ and $\Delta \vdash A \uparrow \tau_A$ and $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash e_\Theta : \Sigma^* \tau_A \to \tau_A$ and $\Omega; \Upsilon \vdash \Theta : A(\Psi; \Sigma)$ and $\Delta; \emptyset \vdash V \uparrow \tau_A e$ and $\Delta; \Psi; \Theta; e_\Theta \triangleright^\tau S$ then for all $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash E : D$ where $B_2 = D\{B\}$, $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash \text{openiter}(B^\tau)[\tau_A] e_\Theta E\{S(e)\} \equiv_{\beta\eta} E'(\text{openiter}(D\{B\}^*\tau_A) e_\Theta S(e)) : B^\tau_A$ where $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash E \triangleright^\tau e'$.

**Proof.** By induction on $\Psi \vdash V \downarrow B_2$.

**Case**

\[\Psi \vdash x \downarrow B_2 \text{ at } \text{var}\]

- Assume an arbitrary $\Delta; \Gamma_1 \uplus \Gamma_2 \vdash E : D$, where $B_2 = D\{B\}$.

- By inversion on $\Delta; \emptyset \vdash x \uparrow \tau_A e$ we have that $e = x$. Furthermore, given that $\Psi(x) = B_2 = D\{B\}$ and $\Delta; \Psi; e_\Theta \triangleright^\tau S$ using Lemma D.7 (substitution elimination) we can conclude

$S(x) = \text{uniter}(D\{B\}^*\tau_A) e_\Theta e''$ where $\Delta; \emptyset \vdash \Theta(x) \uparrow \tau_A e''$.

- From Lemma D.5 (typing for elimination) on $\Psi \vdash x \downarrow D\{B\}$ and $\Omega; \Upsilon \vdash \Theta : A(\Psi; \Sigma)$ we can conclude $\Omega; \Upsilon \vdash \Theta(x) : A(D\{B\})$. Using Lemma C.11 (type encoding total and decidable) on $A(D\{B\})$ and Lemma C.16 (commutativity for parameterization and type encoding) we have a derivation $\Delta \vdash A(D\{B\}) \uparrow \tau_C$ where $\Delta \vdash \tau_C \equiv_{\beta\eta} D\{B\}^*\tau_A : *$.

47
Using Theorem C.2 (static correctness, forward direction) on $\Delta; \emptyset \vdash \Theta(x) \triangleright e''$, with the auxiliary judgements, $\Delta; \emptyset \vdash x : \tau_a$ and $\Delta \vdash Y : \Gamma_1$ and $\Delta \vdash \Omega : \tau_B$ and $\Delta \vdash A(D\{B\}) \vdash \tau_C$ and $\Omega; Y \vdash \Theta(x) : A(D\{B\})$, we can conclude that $\Delta; \Gamma_1 \vdash e'' : \tau_C$.

Furthermore, given $\Delta \vdash e_C \equiv \beta_t D\{B\}^*\tau_A : \star$ we know that $\Delta; \Gamma_1 \vdash e'' : D\{B\}^*\tau_A$ by type equivalence (tp_eq).

From Lemma D.21 (lifting right inverse) on $\Delta; \Gamma_1 \vdash e_\Theta : \Sigma^*\tau_A \rightarrow \tau_A$ and $\Delta; \Gamma_1 \vdash e'' : D\{B\}^*\tau_A$, and congruence of substitution (eqsubs), we can conclude that $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

From Lemma C.16 (commutativity for parameterization and type encoding) on $\Delta; \Gamma_1 \vdash e'' : D\{B\}^*\tau_A$ and $\Delta \vdash \Lambda \rightarrow \tau_A$, we know that $\Delta; \Gamma_1 \vdash e'' : A(D)$.

From Lemma D.2 (properties of iteration, part 1), congruence of substitution (eqsubs) on $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$ and $\Theta$-equivalence (eqabs_beta), we can conclude that $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

From Lemma D.19 (iterated context typing) on $\Delta; \Gamma_1 \vdash e'' : D\{B\}^*\tau_A$ and $\Delta \vdash A \rightarrow \tau_A$ we know that $\Delta; \Gamma_1 \vdash e'' : A(D)$.

This equivalence with Lemma D.20 (congruence for iterated contexts) on $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$ and $\Theta$-equivalence (eqabs_beta), we can conclude that $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

Rolling the substitution $S = \text{uniter}(D\{B\}^*\tau_A) \equiv \beta_t \beta_t$ back up gives us $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

Using transitivity (eqtrans) on $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$ and $\Theta$-equivalence (eqabs_beta), we can conclude that $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

Consequently, we have $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

Case

Assume an arbitrary $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$. By inversion on $\Delta; \emptyset \vdash e'' : \equiv \beta_t \beta_t$.

Using inversion on $\Delta; \emptyset \vdash e'' : \equiv \beta_t \beta_t$.

From inversion on $\Delta; \emptyset \vdash e'' : \equiv \beta_t \beta_t$.

From Lemma C.16 (commutativity for parameterization and type encoding) on $\Delta \vdash B_2 \rightarrow \tau_B$ and $\Delta \vdash \Lambda \rightarrow \tau_A$ we can conclude that $\Delta \vdash \tau_B = \equiv \beta_t \beta_t$.

Using congruence with type equivalence (eqtp) gives us $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

By Lemma D.2 (properties of iteration, part 7), and congruence of substitution (eqsubs) on $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$ and $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

Consequently, we have $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$.

By congruence of application (eqapp) on $\Delta; \Gamma_1 \vdash e'' : \equiv \beta_t \beta_t$ and $\Delta \vdash \Lambda \rightarrow \tau_A$.

48
By Lemma D.2 (properties of iteration, part 2) and \(\beta\)-equivalence again we have that

\[
\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openter}\{b^\ast\}[\tau_A] \ e_\Theta \ (\lambda x : \tau_B \bullet \ \text{openter}\{B_2\}[\tau_A] e_\Theta \ e) \equiv_{\beta_\eta} \ (Ax : \tau_B \bullet \text{openter}\{B_2\}[\tau_A] e_\Theta \ e) : b^\ast \tau_A
\]

Again using Lemma D.2 (properties of iteration, part 8), with congruence of substitution on

\[
\Delta; \Gamma_1 \cup \Gamma_2 \vdash e_\Theta : \Sigma^* \tau_A \rightarrow \tau_A, \ congruence \ of \ application, \ and \ rolling \ up
\]

\[
e = \lambda x : \tau_B \bullet \text{openter}\{B_2\}[\tau_A] e_\Theta \ e \equiv_{\beta_\eta} \text{openter}\{B_2\}[\tau_A] e_\Theta \ e) : b^\ast \tau_A
\]

If we use the syntactic equivalence of bullet and application contexts (cag_bul,b,cag_app) we have the desired result.

\[
\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openter}\{b^\ast\}[\tau_A] \ (\bullet \bullet \ e) \equiv_{\beta_\eta} \text{openter}\{B_2\}[\tau_A] e_\Theta \ e) : b^\ast \tau_A
\]

– It follows from the rule for iterated bullets and application (itc_bulc, itc_app) on

\[
\Delta; \Gamma_1 \cup \Gamma_2 \vdash e' : B_2^\ast \ (\text{Rec \ } \Sigma^* \tau_A) \ we \ can \ conclude \ that \ where
\]

\[
\Delta; \Gamma_1 \cup \Gamma_2 \vdash (\bullet \bullet \ e') \equiv_{\beta_\eta} \text{openter}\{B_2\}[\tau_A] e_\Theta \ e') : \tau_A
\]

Case

\[
\Psi \vdash V_1 \downarrow B_1 \rightarrow B_2, \ \Psi \vdash V_2 \uparrow B_1 \at \text{app}
\]

– Assume an arbitrary \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash e : D\) where \(B_2 = D \{B\}\).

– From typing of atomic and canonical forms[24] on \(\Psi \vdash V_2 \uparrow B_1\) we can conclude \(\emptyset; \Psi \vdash V_2 : B_1\).

By inversion on \(\Delta; \emptyset \vdash V_1 V_2 \uparrow \tau \ a\) we have that \(e = e_1 \ e_2\) where \(\emptyset; \emptyset \vdash V_1 \downarrow \tau_s \ e_1\) and

\[
\emptyset; \emptyset \vdash V_2 \downarrow \tau_s \ e_2.
\]

– By Lemma C.11 (type encoding total and decidable) we know that \(\Delta \vdash B_1 \downarrow \tau_B\). Using Lemma D.8 (static correctness with substitution) on \(\emptyset; \emptyset \vdash V_2 \downarrow \tau_s \ e_2\) and \(\Delta \vdash B_1 \downarrow \tau_s \ e_2\) and

\[
\Psi \vdash V_2 \uparrow B_1 \emptyset; \emptyset \vdash S(e_2) : \tau_B\) using Lemma C.16 (commutativity for parameterization and type encoding), weakening, and type equivalence (tp_eq) we know that \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash (S(e_2)' : B_1^\prime (\text{Rec } \Sigma^* \tau_A)\).

– Using the context typing rule for application (ctp_app) on \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash e : D\) and

\[
\Delta \vdash B_1 \downarrow \tau_s \ B_1 \text{ and } B_1 \vdash \text{openter}\{B_1^\prime\}[\tau_A] e_\Theta \ E \{S(e_2)'\} : B_1^\prime \downarrow \tau_s \ B_1
\]

– Appealing to the induction hypothesis on \(\Psi \vdash V_1 \downarrow B_1 \rightarrow B_2\), with the auxiliary judgements

\[
\Delta \vdash \emptyset; \emptyset : A (\Psi : \Sigma) \text{ and } \Delta \vdash V_1 \downarrow \tau_s \ e_1\text{ and}
\]

\[
\Delta; \Psi; \emptyset : \text{openter}\{B_2\}[\tau_A] e_\Theta \ E \{S(e_2)'\} \equiv_{\beta_\eta} \text{openter}\{B_2\}[\tau_A] e_\Theta \ S(e_2)' : B_2^\ast \tau_A
\]

where \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \{S(e_2)\} : \emptyset; \emptyset \vdash e : \Sigma^* \tau_A\).

If we use the syntactic equivalence of application contexts (cag_app) and the definition of substitution we can conclude

\[
\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openter}\{B^\ast\}[\tau_A] e_\Theta \ E \{S(e_2)'\} \equiv_{\beta_\eta} \text{openter}\{B^\ast\}[\tau_A] e_\Theta \ S(e_2)' : B^\ast \tau_A
\]

49
– By inversion on \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \bullet S(e'_2)\) \(\Rightarrow \tau^a E' \bullet (\text{openiter}\{B_1\}[\tau_A] e_\Theta S(e'_2))\) we have
  that \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \Rightarrow \tau^a E'\).

– Given Lemma D.18 (iterated context typing) on \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \triangleright A \vdash \tau : D\) and \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \Rightarrow \tau^a A \vdash \tau : E'\) and
  and \(\Delta \triangleright A \vdash \tau : \tau_A\) we know that \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \triangleright A(D)\).

– If we instantiate for all \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E_1 : D_1\) where \(B_1 \rightarrow b = D_1\{B_3\}\),
  \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openiter}\{B_2\}[\tau_A] e_\Theta E_1(S(e'_1)) \equiv_{\beta_\eta} E'_1(\text{openiter}\{D_1; B_3\}[\tau_A] e_\Theta S(e'_1)) : B_3^* \tau_A\)
  where \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E_1 \Rightarrow \tau^a A \vdash \tau : E'\) with \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \bullet \bullet S(e'_2)\) : \(B_1 \rightarrow \bullet\) and \(D\{B\}\), we have
  \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openiter}\{D\{B\}\}[\tau_A] e_\Theta \bullet \bullet S(e'_2) \equiv_{\beta_\eta} \text{openiter}\{(B_1 \rightarrow \bullet) D\{B\}\}[\tau_A] e_\Theta S(e'_1)) : D\{B\}\vdash A\).

Using Lemma D.19 (congruence of iterated contexts) on \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E' \vdash A(D)\) and
  \(\Delta \vdash A \Rightarrow \tau : \tau_A\) we can conclude
  \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E' \text{openiter}\{D\{B\}\}[\tau_A] e_\Theta S(e'_2) \equiv_{\beta_\eta} E' \text{openiter}\{(B_1 \rightarrow \bullet) D\{B\}\}[\tau_A] e_\Theta S(e'_1)) : B^* \tau_A\)

Using this equivalence along with transitivity (eq.trans) and
  \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openiter}\{B^*\}[\tau_A] e_\Theta E \bullet \bullet S(e'_2) \equiv_{\beta_\eta} E' \text{openiter}\{(B_1 \rightarrow \bullet) D\{B\}\}[\tau_A] e_\Theta S(e'_1)) : B^* \tau_A\)

we have the desired result

\[\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openiter}\{B\} \triangleright \tau : \tau_A \vdash E \bullet S(e'_2) \equiv_{\beta_\eta} E' \text{openiter}\{D\{B\}\} \triangleright \tau : \tau_A \vdash E \bullet S(e'_2) : B^* \tau_A\]

Case

\[\Psi \vdash V \downarrow B_2 \times B_1\]

\[\Psi \vdash \text{fst } V \downarrow B_2\]

at \(\text{fst}\)

– Assume an arbitrary \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E : D\) where \(B_2 = D\{B\}\). Using the context typing rule for
  projection (ctp.fst) we can conclude \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{fst} \bullet : D \times B_1\).

– Using inversion on \(\Delta; \emptyset \triangleright \text{fst } V \triangleright \tau :e\) we can conclude \(\Delta; \emptyset \vdash V \triangleright \tau :e'\) where \(\text{e} = \text{fst } e\).

– Appealing to the induction hypothesis on \(\Psi \vdash V \downarrow B_2 \times B_1\), with the auxiliary judgements
  \(\Delta \vdash \Omega \triangleright \Gamma_1\) and \(\Delta \vdash \emptyset \triangleright \Gamma_2\) and \(\Delta \vdash A \triangleright \tau : \tau_A\) and \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash e_\Theta : \Sigma^* \tau_A \triangleright \tau_A\) and
  \(\Omega; \emptyset \vdash \emptyset : A(\Psi; \Sigma)\) and \(\Delta \vdash \emptyset \triangleright \Gamma_2\) and \(\Delta \vdash \emptyset \triangleright \tau : \tau_A\) and
  \(\Delta; \emptyset \vdash \emptyset \triangleright \tau : e'\) and
  \(\Delta; \Psi; e_\Theta \triangleright \tau : S\), we then have forall, \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{fst } E_1 : D_1\) where \(B_2 \times B_1 = D_1\{B_3\}\),

\[
\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openiter}\{B_3\}[\tau_A] e_\Theta E_1(S(e')) \equiv_{\beta_\eta} E'_1(\text{openiter}\{D_1; B_3\}[\tau_A] e_\Theta S(e')) : B^* \tau_A
\]

where \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E_1 \Rightarrow \tau^a A \vdash \tau : E'_1\). If we instantiate this derivation with
\(\Delta; \Gamma_1 \cup \Gamma_2 \vdash \text{openiter}\{B\} \triangleright \tau : \tau_A \vdash E \bullet S(e'_2) \equiv_{\beta_\eta} E' \text{openiter}\{(D \times B_1)\} \triangleright \tau : \tau_A \vdash E \bullet S(e'_2) : B^* \tau_A\)

where \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \Rightarrow \tau^a A \vdash \tau : E'\). By inversion on
\(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \Rightarrow \tau^a A \vdash \tau : E'\) we can conclude \(\Delta; \Gamma_1 \cup \Gamma_2 \vdash E \Rightarrow \tau^a A \vdash \tau : E'\).
Proof. From inversion on Ψ ⊢ τ A 2. if the backward direction follows from the forward direction and from the fact that evaluation in the Ψ ⊢ τ A.

Cases The case for at_snd is symmetric to at fst.

Theorem D.23 (Dynamic Correctness). If ⊢; Ψ ⊢ M : A and ⊢; Ψ ⊢ M ⊢ e and ⊢; Ψ ⊢ V ⊢ e' and ⊢; Ψ ⊢ Γ and

1. if ⊢; Ψ ⊢ M ⊢ V : A ⇔ ⊢; Ψ ⊢ e ≡ βη e' : τ A.
2. if ⊢; Ψ ⊢ M ⊢ V : A ⇔ ⊢; Ψ ⊢ e ≡ βη e' : τ A.

Proof. The backward direction follows from the forward direction and from the fact that evaluation in the SDP calculus is deterministic and total[24]. The forward direction follows by mutual induction over the structure of ⊢; Ψ ⊢ M ⊢ V : A and ⊢; Ψ ⊢ M ⊢ V : A. The cases for ⊢; Ψ ⊢ M ⊢ V : A are uncomplicated. For ⊢; Ψ ⊢ M ⊢ V : A:

Case

\[
\begin{align*}
\Psi &\vdash M_1 \leftarrow \lambda x : A_2. M'_1 : A_2 \to A_1 \\
\Psi &\vdash M_2 \leftarrow V_2 : A_2 \\
\Psi &\vdash M'_1 \{V_2/x\} \leftarrow V : A_1 \\
\Psi &\vdash M_1M_2 \leftarrow V : A_1 \\
\end{align*}
\]

− By inversion on ⊢; Ψ ⊢ M_1M_2 : A_1 we can conclude \( \emptyset; \Psi \vdash M_1 : A_3 \to A_1 \) and \( \emptyset; \Psi \vdash M_2 : A_3 \).
− By type preservation[24] on \( \emptyset; \Psi \vdash M_1 : A_3 \to A_1 \) and \( \Psi \vdash M_1 \leftarrow \lambda x : A_2. M'_1 : A_2 \to A_1 \) we know that \( \emptyset; \Psi \vdash \lambda x : A_2. M'_1 : A_3 \to A_1 \). By inversion we have that \( \emptyset; \Psi \vdash \{ x : A_2 \} \vdash M'_1 : A_1 \) and that \( A_3 = A_2 \).
− From inversion on \( \emptyset; \Psi \vdash M_1M_2 \vdash e \) we have that \( e = e_1e_2 \) where \( \emptyset; \Psi \vdash M_1 \vdash e_1 \) and \( \emptyset; \Psi \vdash M_2 \vdash e_2 \).
− By Lemma D.25 (term encoding is total and decidable) we have \( \emptyset; \Psi \vdash V_2 \vdash \epsilon'_2 \) and \( \emptyset; \Psi \vdash \lambda x : A_2. M'_1 \vdash e'_1 \). By inversion we can conclude that \( e'_1 = \lambda x : \tau'_A \epsilon'_1 \) where \( \emptyset \vdash A_2 \vdash \tau'_A \). \( \emptyset; \Psi \vdash M'_1 \vdash \epsilon''_1 \). Using Lemma C.11 (type encoding total and decidable) we can construct \( \emptyset \vdash A_2 \vdash A_1 \vdash \tau'_A \). By inversion we can conclude that \( \tau'_A = \tau_1 \to \tau_2 \) where \( \emptyset \vdash A_2 \vdash \tau_1 \) and \( \emptyset \vdash A_1 \vdash \tau_2 \). From Lemma C.13 (uniqueness of type encoding) can conclude that \( \tau'_A = \tau_1 \).

51
Using Lemma D.27 (substitution for the encoding of modal variables) on \(\varnothing; \Psi \vdash M_1 : A_2 \rightarrow A_1\) and \(\varnothing; \varnothing \vdash M_1 \triangleright e_1\) and \(\varnothing; \varnothing \vdash M_1 \triangleright e_1\) and \(\varnothing; \varnothing \vdash A_2 \rightarrow A_1 \triangleright \tau_1 \rightarrow \tau_2\) and \(\varnothing; \Psi \triangleright \Gamma\) and \(\Psi \vdash M_1 \leftarrow \lambda x : A_2 : M_1' : A_2 \rightarrow A_1\) we have that \(\varnothing; \Gamma \vdash e_1 \equiv_{\beta\eta} e_1' : \tau_1 \rightarrow \tau_2\). Similarly, applying the induction hypothesis to \(\varnothing; \Psi \vdash M_2 : A_2\) and \(\varnothing; \varnothing \vdash M_2 \triangleright e_2\) and \(\varnothing; \varnothing \vdash V \triangleright e_2'\) and \(\varnothing \vdash A_2 \triangleright \tau_1\) and \(\varnothing \vdash \Psi \triangleright \tau_1\) and we have a derivation that \(\varnothing; \Gamma \vdash e_2 \equiv_{\beta\eta} e_2' : \tau_1\).

By type preservation[24] on \(\varnothing; \Psi \vdash M_2 : A_2\) and \(\varnothing; \varnothing \vdash V \triangleright e_2\) we know that \(\varnothing; \Psi \vdash V : A_2\). By substitution on \(\varnothing; \Psi \vdash V : A_2\) and \(\varnothing; \Psi \cup \{x : A_2\} \vdash M_1' : A_1\) we have that \(\varnothing; \Psi \vdash V : A_2\).

By Lemma D.25 (term encoding is total and decidable) we have \(\varnothing; \varnothing \vdash M_1'\{V_2/x\} \triangleright e''\).

From appealing to the induction hypothesis on \(\varnothing; \varnothing \vdash M_1'\{V_2/x\} \triangleright e''\), with the auxiliary judgments \(\varnothing; \Psi \vdash M_1'\{V_2/x\} : A_1\) and \(\varnothing; \varnothing \vdash V \triangleright e'\) and \(\varnothing \vdash A_1 \triangleright \tau_A\) and \(\varnothing \vdash \Psi \triangleright \Gamma\) and \(\Psi \vdash M_1'\{V_2/x\} \rightarrow V : A_1\), we can conclude that \(\varnothing; \Gamma \vdash e'' \equiv_{\beta\eta} e' : \tau_A\).

- By Lemma D.24 (substitution for encoding regular term variables) on \(\varnothing; \varnothing \vdash V \triangleright e_2\) and \(\varnothing; \varnothing \vdash M_1' \triangleright e_2\). Similarly, applying \(\varnothing; \varnothing \vdash M_1' \triangleright e_2'\). By inversion on \(\varnothing; \varnothing \vdash M_1' \triangleright e_1'\) and \(\varnothing; \varnothing \vdash M_1' \triangleright e_1''\). By inversion we know that \(e_1' = A \lambda \alpha : \rightarrow \rightarrow \rightarrow e_1''\) and \(\varnothing \vdash \Box A_1 \triangleright \tau_1\) and \(\varnothing \vdash M_1 \triangleright e_1\) and \(\varnothing \vdash \{x\} \vdash M_2 \triangleright e_2\).

- By Lemma D.25 (term encoding is total and decidable) we have \(\varnothing; \varnothing \vdash M_1' \triangleright e_1'\). By inversion on \(\varnothing; \varnothing \vdash \Box \Box A_1\) and \(\Box \Box A_1\) we know that \(\varnothing \vdash M_1 \triangleright e_1\) and \(\varnothing \vdash M_1 \triangleright e_1''\).

Therefore, by induction on the derivation \(\varnothing; \varnothing \vdash M_1 \triangleright e_1\), with the auxiliary judgments \(\varnothing; \Psi \vdash M_1 : \Box A_1\) and \(\varnothing; \varnothing \vdash \Box M_1' \triangleright \Lambda \alpha : \rightarrow \rightarrow e_1''\) and \(\varnothing \vdash \Box A_1 \triangleright \tau_1\) and \(\varnothing \vdash \Psi \triangleright \Gamma\) and \(\Psi \vdash M_1 \leftarrow \boxdot M_1' : \Box A_1\), we can conclude \(\varnothing; \Gamma \vdash e_1 \equiv_{\beta\eta} \Lambda \alpha : \rightarrow \rightarrow e_1'' : \tau_1\).

By type preservation[24] on \(\varnothing; \Psi \vdash M_1 : \Box A_1\) and \(\varnothing; \varnothing \vdash M_1 \leftarrow \Box M_1' : \Box A_1\) we know that \(\varnothing; \Psi \vdash \Box M_1' : \Box A_1\). By inversion on \(\varnothing; \varnothing \vdash \Box M_1' : \Box A_1\) we have that \(\varnothing; \varnothing \vdash M_1' : A_1\). Therefore, by substitution on \(\varnothing \vdash \{x : A_1\}\); \(\varnothing \vdash M_2 \{M_1'/x\} \triangleright e_2\{e_1'/x\}\). Which we know by the above is just \(\varnothing; \varnothing \vdash M_2 \{M_1'/x\} \triangleright e_2\{e_1'/x\}\).

- Finally by \(\beta\)-equivalence and transitivity (eq.abs_beta and eq.trans) on \(\varnothing; \Gamma \vdash e_2\{e_1'/x\} \equiv_{\beta\eta} e' : \tau_A\) we have the desired \(\varnothing; \Gamma \vdash (\lambda x : \tau_1, e_2) e_1 \equiv_{\beta\eta} e' : \tau_A\).

Case

\[\Psi \vdash M \leftarrow \Box M' : \Box B\]
\[\varnothing \vdash M' \triangleright V' : B\]
\[\Psi \vdash (A', \varnothing, \Theta)(V') \rightarrow V : A'(B)\]
\[\Psi \vdash \text{iter} [\Box B, A'|\Theta] M \rightarrow V : A'(B)\]
– By inversion on $\emptyset; \psi \vdash \text{iter} \langle B \rangle A' \langle \theta \rangle M : A$ we know that $A = A' \langle B \rangle$ and $\emptyset; \psi \vdash M : \square B$ and $\emptyset; \psi \vdash \emptyset : A'(\Sigma)$.

– By inversion on $\emptyset; \emptyset \vdash \text{iter} \langle B \rangle A' \langle \theta \rangle M \triangleright e$ we know that $e = \text{iter} \langle B^* \rangle \tau_\alpha [\tau'_A] e_\theta e_M$ and $\emptyset; \emptyset \vdash A' \triangleright, \tau'_A$ and $\emptyset; \emptyset \vdash \theta \triangleright e_\theta$ and $\emptyset; \emptyset \vdash M \triangleright e_M$.

– By Lemma D.25 (term encoding is total and decidable) we can construct $\emptyset; \emptyset \vdash V' \triangleright \tau'_A e'_v$ and $\emptyset; \emptyset \vdash (A', \emptyset, \theta) (V') \triangleright e_i$.

– By the property of evaluation results [24] on $\emptyset \vdash M' \uparrow V' : B$ we can conclude $\emptyset \vdash V' \uparrow B$.

– By the rule for empty elimination substitutions (sub_empty) we have $\emptyset; \emptyset; \theta; e_\theta \triangleright \tau'_A \{ \}$. From $\emptyset \vdash V' \uparrow B$ and $\emptyset; \emptyset \vdash \tau'_A$ and $\emptyset; \emptyset \vdash \theta \triangleright e_\theta$ and $\emptyset; \emptyset \vdash V' \triangleright \tau'_A e'_v$ and $\emptyset; \emptyset; \theta; e_\theta \triangleright \tau'_A \{ \}$ we can construct $\emptyset; \emptyset \vdash (A', \emptyset, \theta) (V') \triangleright \tau'_A \text{openiter} \langle B^* \rangle [\tau'_A] e_\theta \{ \}(e'_v)$ by the rule for encoding eliminations (en_elim).

– Using Lemma D.10 (dynamic correctness of elimination) on $(A', \emptyset, \theta)(V')$ and $\emptyset; \emptyset \vdash (A', \emptyset, \theta) (V') \triangleright \tau'_A \text{openiter} \langle B^* \rangle [\tau'_A] e_\theta \{ \}(e'_v)$ and $\emptyset; \emptyset \vdash (A', \emptyset, \theta)(V') \triangleright e_i$ and $\emptyset; \emptyset \vdash \tau'_A$ we have that $\emptyset; \emptyset \vdash \text{openiter} \langle B^* \rangle [\tau'_A] e_\theta \{ \}(e'_v) \equiv_{\beta_e} e_i : \tau'_A$. Furthermore, by the definition of substitution we know $\emptyset; \emptyset \vdash \text{openiter} \langle B^* \rangle [\tau'_A] e_\theta e'_v \equiv_{\beta_e} e_i : \tau'_A$.

– By Lemma D.25 (term encoding is total and decidable) we have $\emptyset; \emptyset \vdash \text{box} M' \triangleright e'_M$. By inversion on $\emptyset; \emptyset \vdash \text{box} M' \triangleright e'_M$ we have that $e'_M = A\alpha : \star \rightarrow \star. e'_M$ and $\{ \alpha : \star \rightarrow \star \}; \emptyset \vdash M' \triangleright e'_M$.

– Using Lemma C.11 (type encoding total and decidable) we can construct $\emptyset; \emptyset \vdash \square B \triangleright B$ and and $\emptyset; \emptyset \vdash B \triangleright \tau'A \tau'B$.

– By application of the induction hypothesis to the derivation $\emptyset; \emptyset \vdash M \triangleright e_M$, with the auxiliary judgments $\emptyset; \emptyset \vdash M : \square B$ and $\emptyset; \emptyset \vdash \text{box} M' \triangleright \tau'A \tau'B$ and $\emptyset; \emptyset \vdash \psi \triangleright \Gamma$ and $\emptyset; \emptyset \vdash M \rightarrow \text{box} M' : \square B$, we can conclude that $\emptyset; \emptyset \vdash e_M \equiv_{\beta_e} \psi \rightarrow \star. e'_M \equiv_{\beta_e} \tau'B$.

– Because $\tau'_A$ was encoded in a empty context, by Lemma C.7 (well-formedness of encoding) we know that $\emptyset \vdash \tau'_A : \star$. Then, using Lemma D.26 (world substitution for terms) on $\emptyset \vdash \tau'_A : \star$ and $\emptyset \vdash A \alpha : \star \rightarrow \star. e'_M$ we know from that $\emptyset; \emptyset \vdash M' \triangleright \tau'_A e'_M \equiv_{\beta_e} \tau'_A$ and $\emptyset; \emptyset \vdash \psi \rightarrow \psi \rightarrow \star. e'_M \equiv_{\beta_e} \tau'B$.

– By type preservation [24] on $\emptyset \vdash M \rightarrow \text{box} M' : \square B$ and $\emptyset; \emptyset \vdash M : \square B$ we know that $\emptyset; \emptyset \vdash \text{box} M' : \square B$. By inversion on $\emptyset; \emptyset \vdash \text{box} M' : \square B$ we have that $\emptyset; \emptyset \vdash M' : B$.

– By application of the induction hypothesis to $\emptyset; \emptyset \vdash M' \triangleright \tau'_A e'_M \equiv_{\beta_e} \tau'_A$ and $\emptyset; \emptyset \vdash M' \rightarrow \tau'A \tau'B$ and $\emptyset; \emptyset \vdash B \rightarrow \tau'A \tau'B$. By $\emptyset; \emptyset \vdash \psi \rightarrow \Gamma$ and $\emptyset; \emptyset \vdash M' \rightarrow V' : B$, we can conclude that $\emptyset; \emptyset \vdash e'_M \equiv_{\beta_e} \tau'B$.

– Using Lemma D.4 (replacements are well-formed dynamic replacements) on $\emptyset; \emptyset \vdash \tau'A \equiv_{\beta_e} \tau'A$ we have that $\emptyset; \emptyset \vdash \psi \rightarrow \psi \rightarrow A' \equiv_{\beta_e} \tau'A$. Using this encoding along with $\emptyset \vdash V' \uparrow B$ and Lemma D.5 (elimination typing) we can conclude $\emptyset; \emptyset \vdash \psi \rightarrow \psi \rightarrow \tau'A \rightarrow \tau'B$.

– Appealing to the induction hypothesis on $\emptyset; \emptyset \vdash \tau'A \equiv_{\beta_e} \tau'A$ and $\emptyset; \emptyset \vdash \tau'A \rightarrow \tau'A$ and $\emptyset; \emptyset \vdash \psi \rightarrow \tau'A$ and $\emptyset; \emptyset \vdash \psi \rightarrow \psi \rightarrow \tau'A$ and $\emptyset; \emptyset \vdash \psi \rightarrow \tau'A \rightarrow \tau'B$ we can conclude that $\emptyset; \emptyset \vdash e'_M \equiv_{\beta_e} \psi \rightarrow \tau'A$.

– By $\beta$-equivalence (eq_tabs_beta) on $\emptyset; \emptyset \vdash e'_M \equiv_{\beta_e} \psi \rightarrow \tau'A \equiv_{\beta_e} \tau'B$ we have $\emptyset; \emptyset \vdash A : \star \rightarrow \star. e'_M : \psi \rightarrow \star. e'_M$ and $\emptyset; \emptyset \vdash \psi \rightarrow \tau'A : \star \rightarrow \star. e'_M : \tau'B$ and $\emptyset; \emptyset \vdash e'_M \equiv_{\beta_e} \psi \rightarrow \star. e'_M : \tau'B$.

– By $\beta$-equivalence (eq_japp) on $\emptyset; \emptyset \vdash e'_M \equiv_{\beta_e} \psi \rightarrow \star. e'_M : \tau'B$ we use of application congruence (eq_app). By $\emptyset; \emptyset \vdash \text{openiter} \langle B^* \rangle [\tau'_A] e_\theta \equiv_{\beta_e} \psi \rightarrow \star. e'_M : \tau'A$ and $\emptyset; \emptyset \vdash \text{openiter} \langle B^* \rangle [\tau'_A] e_\theta e'_v : \tau_A$ we can use transitivity (eq_trans) to conclude $\emptyset; \emptyset \vdash \text{openiter} \langle B^* \rangle [\tau'_A] e_\theta e'_v : \tau_A$ and $\emptyset; \emptyset \vdash \text{iter} \langle B^* \rangle [\tau'_A] e_\theta e'_v : \tau_A$ and finally by rolling up the definition of $\text{iter}$, we have the desired $\emptyset; \emptyset \vdash \text{iter} \langle B^* \rangle [\tau'_A] e_\theta e'_v : \tau_A$. 53
Cases The remaining cases follow by straightforward application of of the induction hypothesis and congruence rules.

Lemma D.24 (Substitution for encoding of regular term variables). If $\Delta; \Xi \vdash M_1 \triangleright_{\tau} e_1$ and $\Delta; \Xi \vdash M_2 \triangleright_{\tau} e_2$ and $\Delta; \Xi \vdash M_2 \{M_1/x\} \triangleright_{\tau} e$ where $x \notin \Xi$ then $e = e_2\{e_1/x\}$.

Proof. By straightforward induction over the structure of $\Delta; \Xi \vdash M_2 \triangleright_{\tau} e_2$.

Lemma D.25 (Replacement and term encoding are total and decidable).

1. If $\Omega; \Upsilon \vdash M : A$ and $\Delta \triangleright_{\tau} \tau : *$ we can construct $\Delta; \text{dom}(\Omega) \vdash M \triangleright_{\tau} e$.

2. If $\Omega; \Upsilon \vdash \Theta : A(\Sigma)$ and $\Delta \triangleright_{\tau} \tau : *$ we can construct $\Delta; \text{dom}(\Omega) \vdash \Theta \triangleright_{\tau}^\Sigma e_\Theta$.

Proof. By mutual induction over the structure of $\Omega; \Upsilon \vdash M : A$ and $\Omega; \Upsilon \vdash \Theta : A(\Sigma)$, and use of Lemma C.11 (type encoding is total and decidable).

Lemma D.26 (World substitution for term encoding). If $\Delta \triangleright_{\tau} \{\alpha : * \rightarrow *\}; \Xi \vdash M \triangleright_{\alpha\tau} e$ then $\Delta; \Xi \vdash M \triangleright_{\tau} e\{\lambda\beta : \tau/\alpha\}$.

Proof. Follows by straightforward induction over the structure of $\Delta \triangleright_{\tau} \{\alpha : * \rightarrow *\}; \Xi \vdash M \triangleright_{\alpha\tau} e$ and Lemma C.15 (world substitution for type encoding).

Lemma D.27 (Substitution of for encoding modal variables). If $\Delta \triangleright_{\Upsilon \triangleright_{\tau} \Gamma_1} \triangleright_{\tau} A$ and $\Delta \triangleright_{\Omega \triangleright_{\tau} \Gamma_2} \triangleright_{\tau} A$ and $\Delta \vdash M_1 \triangleright_{\alpha\tau} e_1$ and $\Delta \vdash M_2 \triangleright_{\alpha\tau} e_2$ then $\Delta \vdash \text{dom}(\Omega) \triangleright_{\tau} M_2 \{M_1/x\} \triangleright_{\tau} e_2\{x/\alpha\}$ where $\Delta; \Gamma_1 \triangleright_{\tau} e_1 \equiv_{\beta\eta} e_1\{\Lambda\alpha : * \rightarrow \tau; e_2/x\} : \tau_{A_1}$.

Proof. By induction over the structure of $\Delta; \text{dom}(\Omega) \triangleright_{\tau} M_1 \triangleright_{\alpha\tau} e_1$.

Case $x \in \text{dom}(\Omega) \triangleright_{\tau} x \triangleright_{\tau} x[\lambda\beta : \tau/\alpha]$ en_bvar

- By using Lemma D.26 (world substitution for term encodings) on the derivation $\Delta \triangleright_{\tau} \{\alpha : * \rightarrow *\}; \text{dom}(\Omega) \triangleright_{\tau} M_2 \triangleright_{\alpha\tau} e_2 \{\Lambda\beta : \tau/\alpha\}$, which means by the definition of substitution that $\Delta \vdash \text{dom}(\Omega) \triangleright_{\tau} M_2 \{M_1/x\} \triangleright_{\tau} e_2\{\lambda\beta : \tau/\alpha\}$.

- By modal type substitution on $\Omega \triangleright_{\tau} M_1 \triangleright_{\alpha\tau} e_1$ and $\Omega \triangleright_{\tau} M_2 \triangleright_{\alpha\tau} e_2$ and $\text{dom}(\Omega) \triangleright_{\tau} x \{M_2/x\} : A_1$.

- By using Theorem C.2 (static correctness, forward direction) on $\Delta \vdash \text{dom}(\Omega) \triangleright_{\tau} x \{M_2/x\} \triangleright_{\tau} e_2\{\lambda\beta : \tau/\alpha\}$, with the auxiliary judgements $\Delta \vdash \Upsilon \triangleright_{\tau} \Gamma_1$ and $\Delta \vdash \Omega \triangleright_{\tau} \Gamma_2$ and $\Delta \vdash A_1 \triangleright_{\tau} A$ and $\Omega \triangleright_{\tau} x \{M_2/x\} : A_1$; we have that $\Delta; \Gamma_1 \triangleright_{\tau} e_2\{\lambda\beta : \tau/\alpha\} : \tau_{A_1}$.

- The congruence $\Delta; \Gamma \triangleright_{\tau} e_2\{\lambda\beta : \tau/\alpha\} \equiv_{\beta\eta} (\Lambda\alpha : * \rightarrow \tau; e_2)[\lambda\beta : \tau/\alpha] : \tau_{A_1}$, follows by $\beta$-equivalence for type abstraction (eq.tabs_beta) applied to $\Delta; \Gamma \triangleright_{\tau} e_2\{\lambda\beta : \tau/\alpha\} : \tau_{A_1}$. Finally, by the definition of substitution, we know that $\Delta; \Gamma \triangleright_{\tau} e_2\{\lambda\beta : \tau/\alpha\} \equiv_{\beta\eta} (\Lambda\alpha : * \rightarrow \tau; e_2)[\lambda\beta : \tau/\alpha] : \tau_{A_1}$.

Cases The remaining cases follow from straightforward uses of the induction hypothesis and congruence rules.
E  Static semantics of SDP calculus

E.1 Atomic and canonical terms

\[
\begin{align*}
\frac{\Psi(x) = B}{\Psi \vdash x \downarrow B} & \quad \text{at_var} & \frac{\Sigma(c) = B \rightarrow b}{\Psi \vdash c \downarrow B \rightarrow b} & \quad \text{at_cons} & \frac{\Psi \vdash V_1 \downarrow B_2 \rightarrow B_1 \quad \Psi \vdash V_2 \uparrow B_2}{\Psi \vdash V_1 V_2 \downarrow B_1} & \quad \text{at_app} \\
\frac{\Psi \vdash V \downarrow B_1 \times B_2}{\Psi \vdash \text{fst} V \downarrow B_1} & \quad \text{at_fst} & \frac{\Psi \vdash V \downarrow B_1 \times B_2}{\Psi \vdash \text{snd} V \downarrow B_2} & \quad \text{at_snd} & \frac{\Psi \vdash V \downarrow b}{\Psi \vdash \text{can}_\text{at}} & \quad \text{can_at} \\
\frac{\Psi \vdash \lambda x : B_1 . V \uparrow B_1 \rightarrow B_2}{\Psi \vdash \text{can}_\text{jam}} & \quad \text{can_jam} & \frac{\Psi \vdash V_1 \uparrow B_1 \quad \Psi \vdash V_2 \uparrow B_2}{\Psi \vdash \langle V_1, V_2 \rangle \uparrow B_1 \times B_2} & \quad \text{can_prod}
\end{align*}
\]

E.2 Iteration types

\[
\begin{align*}
A(b) & \triangleq A & A(1) & \triangleq 1 & A(B_1) & \triangleq A'_1 & A(B_2) & \triangleq A'_2 & \text{tp_itarrow} \\
A(B_1 \times B_2) & \triangleq A'_1 \times A'_2 & & & & & & & \text{tp_it_times}
\end{align*}
\]

E.3 Replacement typing rules

\[
\begin{align*}
\forall c_i \in \text{dom}(\Sigma) & \quad \Sigma(c_i) = B_i & \Omega; \Upsilon ; \Theta(c_i) : A(B_i) \\
\frac{\Omega; \Upsilon : \Theta : A(\Sigma) & \text{tp_rep}}
\end{align*}
\]

E.4 Term typing rules

\[
\begin{align*}
\frac{x \not\in \text{dom}(\Omega) & \quad \Upsilon(x) = A}{\Omega; \Upsilon \vdash x : A} & \quad \text{tp_var} & \frac{x \not\in \text{dom}(\Upsilon) & \quad \Omega(x) = A}{\Omega; \Upsilon \vdash x : A} & \quad \text{tp_bvar} & \frac{\Omega; \Upsilon \vdash \langle \rangle : 1}{\Omega; \Upsilon \vdash \langle \rangle} & \quad \text{tp_unit} \\
\frac{\Sigma(c) = B \rightarrow b}{\Omega; \Upsilon \vdash c : B \rightarrow b} & \quad \text{tp_con} & \frac{\Omega; \Upsilon \vdash \{ x : A_1 \} \vdash M : A_2}{\text{tp_abs}} & \frac{\Omega; \Upsilon \vdash \lambda x : A_1 . M : A_1 \rightarrow A_2}{\text{tp_abs}} & \frac{\Omega; \Upsilon \vdash \text{let} \; x : A_1 = M_1 \; \text{in} \; M_2 : A_2}{\text{tp_letb}} \\
\frac{\Omega ; \Upsilon \vdash M_1 : A_1 \rightarrow A_2}{\Omega ; \Upsilon \vdash M_1 M_2 : A_2} & \quad \text{tp_app} & \frac{\Omega ; \Upsilon \vdash M_1 : A_1}{\Omega ; \Upsilon \vdash \text{box} \; M : \square A} & \quad \text{tp_box} & \frac{\Omega ; \Upsilon \vdash \langle M_1, M_2 \rangle : A_1 \times A_2}{\Omega ; \Upsilon \vdash \text{pair}} & \quad \text{tp_pair} \\
\frac{\Omega ; \Upsilon \vdash \text{box} \; M : \square A}{\Omega ; \Upsilon \vdash \text{snd} \; M : A_2} & \quad \text{tp_snd} & \frac{\Omega ; \Upsilon \vdash M : A_1 \times A_2}{\Omega ; \Upsilon \vdash \text{fst} \; M : A_1} & \quad \text{tp_fst} & \frac{\Omega ; \Upsilon \vdash M : \square B}{\Omega ; \Upsilon \vdash \Theta : A(\Sigma)} & \quad \text{tp_iter}
\end{align*}
\]

55
F Dynamic semantics of SDP calculus

F.1 Evaluation

\[
\begin{array}{c}
\frac{\Psi \vdash M \to V : b}{\Psi \vdash M \uparrow V : b} \text{ ev-at} & \frac{\Psi \uplus \{ x : B_1 \} \vdash M \uparrow V : B_2}{\Psi \vdash M \uparrow \lambda x : B_1, V : B_1 \to B_2} \text{ ev-arr} \\
\frac{\Psi \vdash \text{fst} M \uparrow V_1 : B_1 \quad \Psi \vdash \text{snd} M \uparrow V_2 : B_2}{\Psi \vdash \langle V_1, V_2 \rangle : B_1 \times B_2} \text{ ev-pair} & \frac{\Psi \vdash M \uparrow () : 1}{\Psi \vdash x \mapsto x : B} \text{ ev-var} \\
\frac{\Psi \vdash c \leftarrow c : B \to b}{\Psi \vdash () \leftarrow () : 1} \text{ ev-unit} & \frac{\emptyset ; \Psi \uplus \{ x : A_1 \} \vdash M : A_2}{\Psi \vdash \lambda x : A_1, M \to \lambda x : A_1, M : A_1 \to A_2} \text{ ev-lam} \\
\frac{\Psi \vdash M_1 \leftarrow \lambda x : A_2, M'_1 : A_2 \to A_1}{\Psi \vdash M_1 \leftarrow \langle V_2/x \rangle : V : A_1} \text{ ev-app} & \frac{\Psi \vdash M_1 \leftarrow V_1 : B_2 \to B_1}{\Psi \vdash V_1 \downarrow B_2 \to B_1} \text{ ev-at} \\
\frac{\emptyset ; \Psi \vdash M_1 : B_1 \quad \emptyset ; \Psi \vdash M_2 : B_2}{\emptyset ; \Psi \vdash \langle M_1, M_2 \rangle : B_1 \times B_2} \text{ ev-pair} & \frac{\Psi \vdash M \leftarrow \langle M_1, M_2 \rangle : A_1 \times A_2 \quad \Psi \vdash M_2 \leftarrow V : A_2}{\Psi \vdash \langle M_1, M_2 \rangle \leftarrow V : A_1} \text{ ev-fst} \\
\frac{\Psi \vdash \text{fst} M \leftarrow V_1 : B_1 \quad \Psi \vdash \text{snd} M \leftarrow V_2 : B_2}{\Psi \vdash \langle V_1, V_2 \rangle : B_1 \times B_2} \text{ ev-fst-at} & \frac{\Psi \vdash M \leftarrow \langle M_1, M_2 \rangle : A_1 \times A_2 \quad \Psi \vdash M_1 \leftarrow V : A_1}{\Psi \vdash \langle M_1, M_2 \rangle \leftarrow V : A_1} \text{ ev-snd} \\
\frac{\Psi \vdash M_1 \leftarrow \text{box} M'_1 : \Box A_1 \quad \Psi \vdash M_2 \leftarrow \langle V_2/x \rangle : V : A_2}{\Psi \vdash \text{let} \ x : A_1 = M_1 \ in \ M_2 \leftarrow V : A_2} \text{ ev-jetb} & \frac{\emptyset ; \Psi \vdash M \mapsto \text{box} M : \Box A}{\emptyset ; \Psi \vdash M \mapsto \text{box} M : \Box A} \text{ ev-box} \\
\frac{\Psi \vdash M \leftarrow \text{box} M' : \Box B}{\emptyset ; \Psi \vdash M' \uparrow V' : B} \text{ ev-at} & \frac{\Psi \vdash \langle A, \emptyset, \Theta \rangle(V) \leftarrow V : A(B)}{\Psi \vdash \text{iter} \ [\Box B, A][\Theta] M \leftarrow V : A(B)} \text{ ev-it}
\end{array}
\]

F.2 Elimination

\[
\begin{array}{c}
\frac{\langle A, \Psi, \Theta \rangle(x) \triangleq \Theta(x)}{\langle A, \Psi, \Theta \rangle(c) \triangleq \Theta(c)} \text{ el-var} & \frac{\langle A, \Psi, \Theta \rangle(c) \triangleq \Theta(c)}{\langle A, \Psi, \Theta \rangle(c) \triangleq \Theta(c)} \text{ el-const} \\
\frac{\langle A, \Psi \uplus \{ x : B \}, \Theta \uplus \{ x \mapsto x' \} \rangle(V) \triangleq M}{\langle A, \Psi, \Theta \rangle(\lambda x : B. V) \triangleq \lambda x' : A(B). M} \text{ el-lam} & \frac{\langle A, \Psi, \Theta \rangle(V_1) \triangleq M_1 \quad \langle A, \Psi, \Theta \rangle(V_2) \triangleq M_2}{\langle A, \Psi, \Theta \rangle(V_1 V_2) \triangleq M_1 M_2} \text{ el-app} \\
\frac{\langle A, \Psi, \Theta \rangle(V) \triangleq M}{\langle A, \Psi, \Theta \rangle(\text{fst} V) \triangleq \text{fst} M} \text{ el-fst} & \frac{\langle A, \Psi, \Theta \rangle(V) \triangleq M}{\langle A, \Psi, \Theta \rangle(\text{snd} V) \triangleq \text{snd} M} \text{ el-snd} \\
\frac{\langle A, \Psi, \Theta \rangle(V_1) \triangleq M_1 \quad \langle A, \Psi, \Theta \rangle(V_2) \triangleq M_2}{\langle A, \Psi, \Theta \rangle(V_1, V_2) \triangleq \langle M_1, M_2 \rangle} \text{ el-prod} & \frac{\langle A, \Psi, \Theta \rangle(\langle \rangle) \triangleq \langle \rangle}{\langle A, \Psi, \Theta \rangle(\langle \rangle) \triangleq \langle \rangle} \text{ el-unit}
\end{array}
\]
G  Semantics of System $F_\omega$

G.1  Well-formed types

$$
\begin{align*}
\Delta(\alpha) &= \kappa & \Delta \vdash \alpha : \kappa & \text{wf_var} \\
\Delta \vdash \alpha : \kappa & \text{wf_unit} \\
\Delta \vdash \tau_1 : * & \Delta \vdash \tau_2 : * & \Delta \vdash \alpha : \kappa & \tau : * & \text{wf forall} \\
\Delta \vdash \tau_1 : * & \Delta \vdash \tau_2 : * & \Delta \vdash \tau_1 \times \tau_2 : * & \text{wf_times} \\
\Delta \vdash \tau_1 : * & \ldots & \Delta \vdash \tau_n : * & \text{wf_variant} \\
\Delta \vdash \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle : * & \text{wf variant} \\
\Delta \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2 & \Delta \vdash \tau_2 : \kappa_1 & \Delta \vdash \tau_1 \tau_2 : \kappa_2 & \text{wf_app} \\
\Delta \vdash \tau_1 : \kappa_1 & \Delta \vdash \tau_2 : \kappa_2 & \Delta \vdash \tau_1 \tau_2 : \kappa_2 & \text{wf_app} \\
\end{align*}
$$

G.2  Weak head atomic and normal types

$$
\begin{align*}
\Delta(\alpha) &= \kappa & \text{whaf_var} \\
\Delta \vdash \tau_1 : * & \Delta \vdash \tau_2 : * & \text{whaf_arrow} \\
\Delta \vdash \alpha : \kappa & \tau : * & \text{whaf forall} \\
\Delta \vdash \tau_1 : * & \Delta \vdash \tau_2 : * & \Delta \vdash \tau_1 \times \tau_2 : * & \text{whaf_times} \\
\Delta \vdash \tau_1 : * & \ldots & \Delta \vdash \tau_n : * & \text{whaf_variant} \\
\Delta \vdash \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle : * & \text{whaf variant} \\
\Delta \vdash \tau_1 : \kappa_1 & \Delta \vdash \tau_2 : \kappa_2 & \Delta \vdash \tau_1 \tau_2 : \kappa_2 & \text{whnf_app} \\
\Delta \vdash \tau_1 : \kappa_1 \rightarrow \kappa_2 & \Delta \vdash \tau_2 : \kappa_1 & \Delta \vdash \tau_1 \tau_2 : \kappa_2 & \text{whnf_app} \\
\Delta \vdash \tau_1 : \kappa_1 & \Delta \vdash \tau_2 : \kappa_2 & \Delta \vdash \tau_1 \tau_2 : \kappa_2 & \text{whnf_app} \\
\end{align*}
$$

G.3  Well-formed environments

$$
\begin{align*}
\forall \alpha : \tau \in \Gamma & \Delta \vdash \tau : * & \text{wf_env} \\
\end{align*}
$$
G.4 Typing rules

\[
\begin{align*}
\Delta \vdash \Gamma \quad \Gamma(x) = \tau & \quad \text{tp}_{\text{var}} \\
\Delta; \Gamma \vdash x : \tau & \\
\Delta \vdash \tau_1 : \ast & \quad \Delta; \Gamma \cup \{x : \tau_1\} \vdash e : \tau_2 & \quad \text{tp}_{\text{abs}} \\
\Delta; \Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 & \quad \text{tp}_{\text{app}} \\
\Delta; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 & \quad \Delta; \Gamma \vdash e_2 : \tau_1 & \\
\Delta; \Gamma \vdash e_1 \cdot e_2 : \tau_2 & \\
\Delta; \Gamma \vdash \ell : \forall \alpha : \ast.1(\alpha) & \quad \Delta \vdash \{\alpha : \kappa\} ; \Gamma \vdash e : \tau & \quad \text{tp}_{\text{unit}} \\
\Delta ; \Gamma \vdash \Lambda \alpha : \kappa . e : \forall \alpha : \kappa . \tau & \quad \text{tp}_{\text{tabs}} \\
\Delta ; \Gamma \vdash e[\tau'] : \tau\{\tau'/\alpha\} & \quad \text{tp}_{\text{app}} \\
\Delta ; \Gamma \vdash e_1 : \tau_1 \times \tau_2 & \quad \text{tp}_{\text{fst}} \\
\Delta ; \Gamma \vdash e_2 : \tau_1 \times \tau_2 & \quad \text{tp}_{\text{snd}} \\
\Delta ; \Gamma \vdash e_1 = e_2 & \quad \text{tp}_{\text{eq}} \\
\Delta ; \Gamma \vdash \tau_1 \equiv \beta \eta \tau'_1 : \ast & \quad \text{tp}_{\text{abs}} \\
\Delta ; \Gamma \vdash e_1 : \tau_1 & \quad \Delta ; \Gamma \vdash e_2 : \tau_1 \\
\Delta ; \Gamma \vdash e_1 \cdot e_2 : \tau_2 & \\
\Delta ; \Gamma \vdash \ell_1 : \tau_1 & \quad \ldots \quad \Delta ; \Gamma \vdash \ell_i : \tau_i & \quad \ldots \quad \Delta ; \Gamma \vdash \ell_n : \tau_n & \quad \text{tp}_{\text{snd}} \\
\Delta ; \Gamma \vdash \text{inj}_l \ell_1 \text{ of } \{\ell_1 : \tau_1, \ldots, \ell_i : \tau_i, \ldots, \ell_n : \tau_n\} & \\
\Delta ; \Gamma \vdash e_1 : \tau & \quad \ldots \quad \Delta ; \Gamma \vdash e_n : \tau & \quad \text{tp}_{\text{case}} \\
\Delta ; \Gamma \vdash \epsilon_1 \text{ in } e_1 \ldots \text{ inj}_n \ell_n \text{ in } e_n : \tau & \quad \text{tp}_{\text{case}} \\
\end{align*}
\]
G.5 Congruence for types

\[ \frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \tau \equiv \beta \gamma \tau : \kappa} \] \[ \text{tp}_\text{eq_refl} \]

\[ \frac{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_2 : \kappa}{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_3 : \kappa} \] \[ \frac{\Delta \vdash \tau_2 \equiv \beta \gamma \tau_3 : \kappa}{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_2 : \kappa} \] \[ \text{tp}_\text{eq_sym} \]

\[ \frac{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_2 : \kappa}{\Delta \vdash \psi \{ \alpha : \kappa \} \vdash \tau_3 \equiv \beta \gamma \tau_4 : \kappa} \] \[ \text{tp}_\text{trans} \]

\[ \frac{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_2 : \kappa'}{\Delta \vdash \tau_3 \{ \tau_1 / \alpha \} \equiv \beta \gamma \tau_4 \{ \tau_2 / \alpha \} : \kappa} \] \[ \text{tp}_\text{eqsubst} \]

\[ \frac{\Delta \vdash (\lambda \alpha : \kappa_1. \tau) \equiv \kappa \quad \Delta \vdash \tau \equiv \beta \gamma \tau : \kappa}{\Delta \vdash (\lambda \alpha : \kappa_1. \tau) \{ \tau / \alpha \} \equiv \beta \gamma \tau : \kappa} \] \[ \text{tp}_\text{abs} \]

\[ \frac{\Delta \vdash (\lambda \alpha : \kappa_1. \tau \alpha) : \kappa_1 \rightarrow \kappa_2 \quad \Delta \vdash \tau \equiv \beta \gamma \tau : \kappa_1}{\Delta \vdash (\lambda \alpha : \kappa_1. \tau \alpha) \{ \tau \{ \alpha / \alpha \} \} \equiv \beta \gamma \tau : \kappa_1} \] \[ \text{tp}_\text{abs_eta} \]

\[ \frac{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_2 : \kappa_1 \quad \Delta \vdash \tau_2 \equiv \beta \gamma \tau_4 : \kappa_1}{\Delta \vdash \tau_1 \tau_2 \equiv \beta \gamma \tau_3 \tau_4 : \kappa_2} \] \[ \text{tp}_\text{eq_app} \]

\[ \frac{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_2 \equiv \beta \gamma \tau_3 \equiv \beta \gamma \tau_4 : \kappa_1}{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_3 \equiv \beta \gamma \tau_4 : \kappa_1} \] \[ \text{tp}_\text{forall} \]

\[ \frac{\Delta \vdash 0 \equiv \beta \gamma 0 : \kappa}{\Delta \vdash \forall \alpha : \kappa. \tau_1 \equiv \beta \gamma \tau_2 : \kappa} \] \[ \text{tp}_\text{forall} \]

\[ \frac{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_2 \equiv \beta \gamma \tau_3 \equiv \beta \gamma \tau_4 : \kappa}{\Delta \vdash \tau_1 \rightarrow \tau_2 \equiv \beta \gamma \tau_3 \rightarrow \tau_4 : \kappa} \] \[ \text{tp}_\text{arrow} \]

\[ \frac{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_2 : \kappa \quad \Delta \vdash \tau_2 \equiv \beta \gamma \tau_4 : \kappa}{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_3 : \kappa \equiv \beta \gamma \tau_4 : \kappa} \] \[ \text{tp}_\text{times} \]

\[ \frac{\Delta \vdash \tau_1 \equiv \beta \gamma \tau_1' : \kappa \quad \ldots \quad \Delta \vdash \tau_n \equiv \beta \gamma \tau_n' : \kappa}{\Delta \vdash \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle \equiv \beta \gamma \langle l_1' : \tau_1', \ldots, l_n' : \tau_n' \rangle : \kappa} \] \[ \text{tp}_\text{variant} \]
G.6 Congruence for terms

\[
\begin{align*}
\Delta, \Gamma \vdash e : \tau &\quad \text{eq_refl} \\
\Delta, \Gamma \vdash e \equiv_{\beta_\eta} e : \tau &\quad \text{eq_sym} \\
\Delta, \Gamma \vdash e_1 \equiv_{\beta_\eta} e_2 : \tau &\quad \text{eq_trans} \\
\Delta, \Gamma \vdash e \equiv_{\beta_\eta} e' : \tau &\quad \Delta \vdash \tau \equiv_{\beta_\eta} \tau' : \ast &\quad \text{eq_tp_eq} \\
\Delta, \Gamma \vdash (x : \tau_1 \land \tau_2) &\quad \Delta, \Gamma \vdash x : \tau_1 \land \tau_2 &\quad \text{eq_var} \\
\Delta, \Gamma \vdash (\lambda x : \tau_1.e) : \tau &\quad \Delta, \Gamma \vdash e \equiv_{\beta_\eta} e' : \tau &\quad \text{eq_abs_beta} \\
\Delta, \Gamma \vdash (\lambda x : \tau_1.e)[\tau] : \tau' &\quad \Delta, \Gamma \vdash e[\tau/\alpha] : \tau' &\quad \text{eq_tabs_beta} \\
\Delta, \Gamma \vdash \lambda \alpha : \kappa.e : \forall \alpha : \kappa.\tau &\quad \Delta, \Gamma \vdash e : \forall \alpha : \kappa.\tau &\quad \text{eq_tabs_eta} \\
\Delta, \Gamma \vdash e_1 \equiv_{\beta_\eta} e_2 : 1(\tau) &\quad \Delta, \Gamma \vdash e_2 : 1(\tau) &\quad \text{eq_unit} \\
\Delta, \Gamma \vdash \text{fst}(e_1, e_2) \equiv_{\beta_\eta} e_1 : \tau_1 &\quad \Delta, \Gamma \vdash e : \tau_1 \times \tau_2 &\quad \text{eq_pair_1} \\
\Delta, \Gamma \vdash \text{fst} \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 &\quad \text{eq_pair_1} \\
\Delta, \Gamma \vdash \text{snd} \langle e_1, e_2 \rangle \equiv_{\beta_\eta} e_2 : \tau_2 &\quad \text{eq_pair_2} \\
\Delta, \Gamma \vdash \text{snd}(e_1, e_2) : \tau_2 &\quad \text{eq_pair_2} \\
\Delta, \Gamma \vdash \text{case}\ e_1 \text{ of } \text{inj}_1 \text{ y} \text{ in } e_1' &\quad \Delta, \Gamma \vdash \text{case}\ e_2 \text{ of } \text{inj}_2 \text{ y} \text{ in } e_2' &\quad \text{eq_case} \\
\Delta, \Gamma \vdash \text{case}\ e_1 \text{ of } \text{inj}_1 \text{ y in } e_1' &\quad \Delta, \Gamma \vdash \text{case}\ e_2 \text{ of } \text{inj}_2 \text{ y in } e_2' &\quad \text{eq_case} \\
\Delta, \Gamma \vdash \text{case}\ \text{inj}_1 \text{ e of } \langle \cdots, \text{inj}_1 \text{ y in } e_1' \rangle &\quad \Delta, \Gamma \vdash \text{case}\ \text{inj}_2 \text{ e of } \langle \cdots, \text{inj}_2 \text{ y in } e_2' \rangle &\quad \text{eq_case} \\
\Delta, \Gamma \vdash \text{case}\ \text{inj}_1 \text{ y in } e_1' &\quad \Delta, \Gamma \vdash \text{case}\ \text{inj}_2 \text{ y in } e_2' &\quad \text{eq_case} \\
\end{align*}
\]