A Resourceful Monad for IO

Luke Lau

A DISSERTATION

Presented to the University of Dublin, Trinity College
In Partial Fulfilment of the Requirements for the Degree of

Masters in Computer Science

Supervisor of Dissertation
Glenn Strong

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

Luke Lau
April 2020

Contents

Chapter 1

Introduction

The immutability and purity of some functional languages make them seem like a perfect fit for parallelism and concurrency. The lack of side effects mean we are free to compute expressions in whatever order we please, and there is no shared mutable state to plague us with race conditions. However in the real world, code is never completely pure. Useful programs need to interact with the real world at some point, whether that be by reading from a keyboard or sending packets over a network. That is, they need to carry out side effects: And the order in which they are carried out is important.

So when it comes to side effects in a language with referential transparency, the language must model the sequence in which effects are carried out. One of the most successful approaches has been to capture the state of the outside world, and have functions with side effects that write to that state. This way, if writing to `stdout` causes some particles to be perturbed on a display monitor, then we can track that by saying the state of the world has changed, and that it matters what order it changed. We don't actually track what particles are there, where they might be or what their current charge is. All we need to know is that the world is slightly different than what it was before.

Concurrent Clean models this by threading the world in and out of functions. Uniqueness types guarantee that the same world is only used once, so that the programmer doesn't create an alternative timeline by duplicating it.

```
readFile :: !String !*World -> (!MaybeError FileError String, !*World)
```

Haskell also treats the world as a state, but without the uniquness guarantee. Any function that interacts with the world returns a function, which returns a new version of the World alongside the function result.[1]

```
type IO a = World -> (World, a)
```

How Haskell ensures that an old world isn't erroneously used is by hiding the updating of the state from the user, tucking it away into a *monad*. This idea of

[1] The actual definition in GHC is

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

using monads to sequence together stateful computations was first introduced by Peyton Jones and Wadler [15] [18]. The programmer no longer needs to keep track of the state of the world. They can keep their imperative code imperative, and their pure code pure. This marriage of monads and IO is one of the crown jewels of functional language research to come out of Haskell.

```haskell
instance Monad IO where
  return x = \w -> (w, x)
  x >>= f = \w -> let (w', y) = x w in f y w'
```

Now IO actions can be easily chained together in a type-safe way that ensures their ordering. Unfortunately, this ordering imposes limitations. One of the main benefits of pure functional languages is that since expressions do not have side effects, there is no restriction on what order they need to be evaluated in. Take for example the following program:

```haskell
f, g, h :: Int -> Int
f x = g x + h x
```

`g` could be evaluated before `h`, or `h` could be evaluated before `g`. It wouldn't make a difference because there are no side effects. One might be tempted then to evaluate the two expressions concurrently, and indeed that would be safe to do so. The same cannot be said for impure IO actions however, and Haskell's type system is well aware of that.

```haskell
f, g, h :: Int -> IO Int
f x = g x >>= \y -> y + h x
```

We need to explicitly bind the actions and sequence evaluation. Does this mean that concurrency is impossible for IO actions? Not at all, many languages provide primitives to run these actions concurrently in a type safe way. Haskell's `base` library has `forkIO`, but for simpliclty we are going to assume the existence of a higher level function that runs two IO actions simultaneously and collects the results.

```haskell
concurrently :: IO a -> IO b -> IO (a, b)
```

Now we can use it to run our two IO actions side by side safely.

```haskell
f, g, h :: Int -> IO Int
f x = g x `concurrently` h x >>= \(a, b) -> return (a + b)
```

But what if `g` and `h` actually looked like this?

```haskell
g x = do
  txt <- readFile "foo.txt"
  return (x + (read txt))
h x = do
  writeFile "foo.txt" "hello"
  return (42 - x)

readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Running these two functions concurrently could be disastrous, as the order in which they execute could affect the outcome of the program. These innocuous looking IO actions then end up introducing non-determinism and race conditions. We know statically that a program such as

```
writeFile "foo.txt" "a" `concurrently` writeFile "foo.txt" "b"
```

should probably not be allowed, because it is blatantly accessing the same resource simultaneously. But then why did the type system allow it? Has it failed us? The goal of the type system is to disallow as many incorrect programs as possible whilst allowing all correct programs. It is a fine line as to what programs are deemed correct and what are deemed incorrect. A type system too lenient and buggy programs will creep through: but a type system too strict and the programmer will waste time fighting the type checker.

In this dissertation we aim to find a point in the design space that rejects such programs as the one above. We do **not** want to allow programs that have glaring race conditions, where we can see that there is a contentious access of resources. Our work is based around the idea of keeping track of what resources are in use, at the type level. We first imagine what this might look like in Haskell by adding another type parameter to our IO type, representing what resource an IO action uses:

```
type IO r a = World -> (World, a)
```

r is a phantom type parameter, which only exists at the type level. Now our type signatures could look like this, annotating the API with what resources it might use.

```
data Resource = FileSystem | Net | Database | OpenGL | ...
readFile :: FilePath -> IO FileSystem ()
writeFile :: FilePath -> IO FileSystem String
readSocket :: Socket -> IO Net ()
runQuery :: Query a -> IO Database a
swapBuffers :: IO OpenGL ()
```

Keep in mind we are painting in broad strokes when we use the word "resource". In the running example the resource has been a file, `foo.txt`, but the notion of a resource can be as broad or as specific as a function needs it to be. It could represent a specific database instance, or a network socket. For simplicity in our example we will consider the entire file system as a single resource, the entire network as a single resource, and so on.

Now that we know what resources each IO action is using, we would like to change the type of our concurrent function to take advantage of this new information. Perhaps we would like to reject any two functions that use the same resource, i.e. it only accepts IO actions with distinct resources.

```
concurrently :: r \~ s => IO r a -> IO s b -> IO ? (a, b)
```

You can read `r \~ s` as "r is distinct from s", or the opposite of a `r ~ s` equality constraint that one might see in a type signature. This, however, does not exist in Haskell. And what does it exactly mean for two resources to "be distinct"? And what resources would the returned IO use?

These questions are answered with a formal definition of a type system that tracks resource usage. We explore a specific point in the design space, where the type system rejects programs like

$$\mathsf{readFile} \curlyvee \mathsf{readFile}$$

but accepts and assigns types to programs such as

$$\mathsf{readFile} \curlyvee \mathsf{readNet} : \mathsf{IO}_{\mathsf{File} \cup \mathsf{Net}}\ \square \times \square$$

In short, we aim to create a type system that keeps tracks of the resources being used, so that the programmer doesn't have to. It **does not** aim to solve concurrency — there will still be programs that have concurrency errors that the type system will still allow. We just aim to narrow down the scope of valid programs, by ruling out those with blatant, concurrent resource access errors.

1.1 Overview

This dissertation has been written in a style that doesn't assume prior knowledge of type theory, and so it should double as a tutorial along our journey to create our resourceful type system.

In Chapter 2 we will talk about the inspirations of the type system, namely separation logic, and how it parallels with our monadic language. We will also briefly go over what we mean by a monad formally, and then look at the original design of Hindley-Damas-Milner which we will build upon. Chapter 3 introduces the language, its type system and its semantics. It gives a complete definition of all the parts needed to prove properties. We prove these properties in Chapter 4, in which we eventually build up and present a proof of its soundness. This proof, and the others that accompany it, are then mechanised within the dependently typed language and proof assistant Agda. The methodology of this is discussed in Chapter 5. Finally, Chapter 6 evaluates our type system. We look at how it might be refined, and find many areas of further work that would be interesting to explore.

Chapter 2

Background

2.1 Separation logic

Much of the original inspiration for this work came from separation logic. Separation logic [14, 16] is an extension to Hoare logic, a system for formalising properties about imperative programs, and has been used to prove properties about resource access in concurrent programs [13]. It is formalised through *specifications*, which comprise of some code alongside a *precondition* a *postcondition*.

$$\{\mathit{precondition}\}\ \mathit{code}\ \{\mathit{postcondition}\}$$

The preconditions and postconditions make assertions about what points to what, inside the *heap*. In the example that O'Hearn gives, a cyclic list is constructed from two pointers. The precondition asserts that x and y point to 0 initially, and then the postcondition asserts that they point to each other afterwards.

$$\begin{gathered}\{x \mapsto 0 * y \mapsto 0\}\\ [x] = y;\\ [y] = x;\\ \{x \mapsto y * y \mapsto x\}\end{gathered}$$

The code sets the value of x to the location of y, and vice-versa. Separation logic extends Hoare logic with the notion of splitting the heap. In the example above, instead of asserting $x \mapsto y \wedge y \mapsto x$, the separation conjunction $*$ is used which says "x points to y *and separately* y points to x". The heap must be able to split into two disjoint subheaps, such that $x \mapsto y$ holds in one and $y \mapsto x$ in the other. As a result x and y cannot point to the same location in the heap. These preconditions and postconditions can be derived from so-called small axioms. As an example, an imperative language might contain an axiom for writing to a pointer such as

$$\{x \mapsto -\}[x] = v\{x \mapsto v\}$$

amongst others for primitive operations like reading, allocating and freeing pointers.

The *frame rule* allows a specification to be extended by adding extra predicates that do not mention any variables used by the code.

$$\frac{\{p\}\ code\ \{q\}}{\{p * r\}\ code\ \{q * r\}}\ \text{where no variable occurring free in r is modified by } code$$

This rule ends up being quite significant, as its the reason why separation logic allows for programs to be reasoned about locally — the predicates involving the heap are shrunk so that they only contain variables that the code modifies.

There is also the *concurrency rule* for reasoning about running two programs side by side. It states that the preconditions and postconditions of the two programs must hold separately to each other.

$$\frac{\{p_1\}\ code_1\ \{q_1\} \qquad \{p_2\}\ code_2\ \{q_2\}}{\{p_1 * p_2\}\ code_1\ ||\ code_2\ \{q_1 * q_2\}}$$

We will see later on in Chapter 6 how the system we end up developing parallels with this.

Krishnaswami built upon separation logic with a higher-order ML like language [9] with types. Like our system, it utilised monadic binding for sequencing, allowing imperative computation within an otherwise pure language. Unlike our system, this included separation logic predicates directly within the language. We will not work with predicates, and instead push all our assertions into the type level.

2.2 Monads categorically

As the language we will develop in Chapter 3 revolves around monads at its core, it is a good idea to take a brief look at what exactly they are from a mathematical standpoint, and dip our toes into the cool waters of category theory. There is an infamous definition[1] of a monad that goes along the lines of

> *A monad is just a monoid in the category of endofunctors, what's the problem?*

Let's break this down. A *category* is a collection of objects (like a set), and morphisms (like functions between sets). In the same way, Haskell has Hask — the category of types. Its objects are types such as `Bool` and `Int`, and its morphisms are functions taking a type and returning another.

```
Bool, Int :: * -- Bool and Int are types
True :: Bool
42 :: Int
f : Bool -> Int -- a morphism between Bool and Int
```

[1] James Iry A Brief, Incomplete, and Mostly Wrong History of Programming Languages

A *functor* is a mapping between categories that preserves the structure. That is, a functor maps objects and morphisms from one category to objects and morphisms in another category. In the same way, a `Functor` in Haskell maps a type to another type, and its functions to other functions with `fmap`.

```
Maybe :: * -> *
Maybe Bool :: * -- Maps Bool to Maybe Bool
fmap :: (a -> b) -> (f a -> f b) -- Maps (a -> b) to (f a -> f b)
```

Functors can have mappings to other functors, and these are known as *natural transformations*. An *endofunctor* is a *functor* from one category to the same category. That is, it maps objects and morphisms from one category, to objects and morphisms in the same category. But what does that mean in Haskell? Well, since `Functor` maps types to types, we are mapping from Hask to Hask — all `Functor`s in Haskell are endofunctors! They exist as type constructors with the kind `* -> *`.

A *monad* is then defined as:

1. An endofunctor $T : X \to X$
2. With a natural transformation to flatten two monads together
   $\mu : T(T(X)) \to T(X)$
3. And another natural transformation $\eta : X \to T(X)$
4. That satisfy some properties (the monad laws)

Or in Haskell,

```
class Functor m => Monad m a where
  join :: m (m a) -> m a
  return :: a -> m a
```

However as you might be aware, the actual definition in Haskell uses bind `>>=`, not the mathematical definition of `join`. It can be implemented in terms of `>>=` instead, and vice-versa using the fact that a monad is also a functor.

```
(>>=) :: m a -> (a -> m b) -> m b
join m = m >>= \x -> x
m >>= f = join (fmap f m)
```

The "monoid in the category of endofunctors" bit then comes about because a monoid is defined as:

1. A set X
2. With an associative binary operation $\circ : X \times X \to X$
3. And an identity element $e : X$
4. That satisfy some properties (the monoid laws)

A monoid is the same deal as a monad, except functor composition is replaced with the Cartesian product and the unit function is replaced with the identity element. And if monoids allow chaining together elements, then monads in programming allow chaining together computation. In our system we will be using the more pragmatic Haskell definition of a monad that uses bind rather than join. But the fundamental idea of sequencing remains the same.

2.3 Type-level programming within GHC

Before creating the system in Chapter 3, we originally attempted to embed a resourceful system within GHC's type system. Whilst the original Haskell 2010 language specification was based on System F, there has been much work [7] [19] carried out to add dependent types to GHC via a plethora of language extensions. Namely, `TypeFamilies` and `GADT` provide a lot of the power needed to carry out type-level programming, and we leveraged this expressiveness to emulate a resource-tracking version of the **`IO`** monad, dubbed **`SubIO`**. It carried around a heap with a phantom type parameter:

```
newtype SubIO (a :: [j]) b = SubIO (IO b)
  deriving (Functor, Applicative, Monad)
```

We then defined a subclass of **`Monad`** that extended it with a sequencing operator and concurrency operator, both which merged the heap parameters together.

```
class Monad (m j) => HeapMonad (m :: [x] -> * -> *) j where
  (>>>=) :: m j a -> (a -> m k b) -> m (j ** k) b
  (|||) :: m j a -> m k b -> m (j ** k) b
  (|||) x f = x >>>= const f
```

It merged the two heaps together through the ** type family, which threw a type error if they overlapped.

```
type family Overlap a b :: Bool where
  Overlap '[] b = 'False
  Overlap (a ': as) b = If (MemberP a b) 'True (Overlap as b)

type family a ** b :: [c] where
  (a ** b) = If (Overlap a b)
                (TypeError ('Text "Heaps overlap!"))
                (a :++ b)
```

An instance for **`SubIO`** was given, and with it some file operations from **`System.IO`**, transformed to accept the file argument as a type argument, so it could be tracked in the heap.

```
instance HeapMonad SubIO j where
  (SubIO x) >>>= f = SubIO (x >>= \z -> let (SubIO y) = f z in y)
  (SubIO x) ||| (SubIO y) = SubIO (forkIO (x >> pure ()) >> y)
```

```
readFile :: forall filePath. KnownSymbol filePath => SubIO '[filePath]
↪  String
readFile = fileOp Prelude.readFile

appendFile :: forall filePath. KnownSymbol filePath => String -> SubIO
↪  '[filePath] ()
appendFile x = fileOp (`Prelude.appendFile` x)

writeFile :: forall filePath. KnownSymbol filePath => String -> SubIO
↪  '[filePath] ()
writeFile x = fileOp (`Prelude.writeFile` x)

fileOp :: forall filePath a. KnownSymbol filePath => (FilePath -> IO a)
↪  -> SubIO '[filePath] a
fileOp f = let fp = symbolVal (Proxy :: Proxy filePath)
            in SubIO (f fp)
```

All together, this meant that functions could be written with the familiar do syntax, and even better run concurrently with `|||` — provided their heaps didn't overlap.

```
someIO = do
  writeFile @"foo.txt" "hello"
  s <- readFile
  return s

someMoreIO x = do
  appendFile @"bar.txt" x
  readFile >>= SubIO . putStrLn

concExample = runSubIO $ someIO ||| someMoreIO "blah"
--- this *should* type error
-- badConcExample = runSubIO $ someIO ||| someIO
```

However we ultimately discovered that overlapping instances don't always throw a type error, as the phantom type parameter keeping track of the heap was evaluated **lazily**. One workaround involved lowering the heap down to the term level and forcing evaluation via `Data.Proxy` and `seq`. Still, we think there are most likely better ways to embed this within GHC's type system.

2.4 Hindley-Damas-Milner

One of our aims whilst developing this type system was to see how well it integrates with existing functional programming languages, specifically those in the ML family. Therefore we based our system on the Hindley-Damas-Milner

(HDM) type system [4]. Originally designed to formalise the type system of ML, it is one of the first formalisations of a polymorphic type system. It was heavily influential at the time and still continues to be so today, paving the way for newer type systems such as System F.

2.4.1 Syntax

Before we can talk about a type system, we need to talk about the language that it operates on. HDM uses an applicative language: A language in which you can apply abstractions.

$\langle \textit{expression } e \rangle ::= x \mid \lambda x.e \mid e\ e' \mid \mathsf{let}\ x = e'\ \mathsf{in}\ e$

$\langle \textit{type } \tau \rangle ::= \square \mid \alpha \mid \tau' \to \tau$

$\langle \textit{type scheme } \sigma \rangle ::= \tau \mid \forall \alpha.\sigma$

$\langle \textit{context } \Gamma \rangle ::= \cdot \mid \Gamma, x : \sigma$

Expressions are an extension of the venerable lambda calculus, with the addition of a new let expression that binds an expression to a variable. This is notably different from abstraction as it allows for polymorphism, which we will see later.

Types are either unit types, type variables or function types. However expressions are not assigned types directly, instead they are given *type schemes* which quantify over type variables. The distinction between the two is necessary so that quantifiers can only appear at the top level.

A *context* is a map from variables to type schemes, represented as a kind of linked list. Whenever we need to work out the type of a variable expression like x, we traverse the context to find its type scheme.

There is also the notion of *free type variables*, which are type variables *inside a type scheme* which have not been bound (quantified over).

$$\begin{aligned} \mathrm{FTV}(\forall \alpha.\tau) &= \mathrm{FTV}(\tau) \setminus \{\alpha\} \\ \mathrm{FTV}(\alpha) &= \{\alpha\} \\ \mathrm{FTV}(\tau \to \tau') &= \mathrm{FTV}(\tau') \cup \mathrm{FTV}(\tau) \end{aligned}$$

2.4.2 Static semantics

A type system describes how we assign *valid* types to our expressions. Most of the time, it's through a typing relation like

$$\Gamma \vdash e : \tau$$

You can read this as "*In the context* Γ*,* e *has the type* τ". This is a relation between a context, an expression and a type, in the same way that $a \leq b$ is a relation between two numbers. With relations we can form *judgements*, such as $1 \leq 2$, or $\Gamma \vdash \lambda x.(y\ x) : \alpha \to \tau$. These are just statements that we can make —

they might be true, they might be false. Is $42 \leq 19$? Does $\Gamma \vdash z : \beta$? I don't know, you tell me. We need something else to be able to tell whether or not they make sense, so we have *typing rules* that dictate what typing judgements we can prove. A typing judgement that we can prove tells us that the expression in it is *well typed*, and that it does indeed have the type it claims to have.

$$\frac{x : \sigma \in \Gamma \qquad \sigma > \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}\ \textsc{App}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e' : \tau' \qquad \Gamma, x : \bar{\Gamma}(\tau') \vdash e : \tau}{\Gamma \vdash \mathsf{let}\ x = e'\ \mathsf{in}\ e : \tau}\ \textsc{Let}$$

These rules consist of some *premises* above a line, and a *conclusion* below it. Premises and conclusions are just other judgements, that we have been able to prove. The gist of these rules is that if you have proof of all the premises above, then you can infer the conclusion at the bottom. So for example, you can read App as *"If e has the type $\tau' \to \tau$ in Γ and e' has the type τ' also in Γ, then e' applied to e has the type τ in Γ."*.

App is one of the four typing rules in the syntax-directed HDM type system, and tells us what happens to the types when we apply an argument to a function. Let's take a look at what the others mean. Abs relates to functions, sometimes called abstractions. It says if e has the type τ in the context Γ, *extended* with x having the type scheme τ', then $\lambda x.e$ has the type $\tau' \to \tau$ in Γ. In other words, if e can have type τ provided it has access to $x : \tau'$, then we can make an abstraction out of it.

If we have a variable expression x, then Var tells us how we can get the type for it. First, we need to make sure x exists in the context Γ. It will have some type scheme σ, but we can't directly assign that to an expression — the typing relation assigns types to terms, not type schemes. Instead, $\sigma > \tau$ says there needs to be a type τ that σ can be *instantiated* to. This instantiation relation says that if σ is a type scheme $\forall \alpha_1 \ldots \alpha_n.\tau'$, then there exists a mapping of type variables to types $\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n$ which we can apply to those bound type variables in τ' to give us τ.

As it turns out, these mappings from type variables to types are really common, and they are called *substitutions*. The whole rule then means, if we can look up a type scheme σ for x, and then instantiate it to some type τ, then we can infer that x has the type τ in Γ.

The Let rule introduces polymorphism to the language. That is, it allows one expression to be used for multiple types. Consider the identity function

$$\lambda x.x$$

What type should it have? It could be used both as a $\mathsf{Int} \to \mathsf{Int}$. But the reality is that we don't care about the underlying type, and want it to work on *anything*.

$$\Gamma \vdash \lambda x.x : \alpha \to \alpha$$

With type variables instead, we have abstracted over the concrete types. Assuming we have a context like $\Gamma = \bullet, a : \mathsf{Int}$, we might try to use it multiple times, each with different types, by passing it in as an argument to abstraction.

$$\Gamma \vdash (\lambda y.(y\ y)\ (y\ a))\ (\lambda x.x) : \mathsf{Int}$$

But this will not work, and it is in fact ill-typed. Our identity function will get passed around as $y : \alpha \to \alpha$ through Abs. When either $(y\ y)$ or $(y\ a)$ try to look up y in Var, they will find that they can't instantiate it to the type they want, because α is not quantified over!

Instead, we need to pass around the identity function as $y : \forall\alpha.\alpha \to \alpha$. With a let expression, we can instead say

$$\Gamma \vdash \mathsf{let}\ y = \lambda x.x\ \mathsf{in}\ (y\ y)\ (y\ a) : \mathsf{Int}$$

If we look at the first premise for Let, we have proof that $\Gamma \vdash \lambda x.x : \alpha \to \alpha$. In the second premise, we put it into the context as $y : \overline{\Gamma}(\alpha \to \alpha)$. $\overline{\Gamma}(\alpha \to \alpha)$ is the close function, which is defined as

$$\overline{\Gamma}(\tau) = \forall\alpha_1 \ldots \alpha_n.\tau\ \mathsf{where}\ \{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\Gamma)$$

This will take the dangling free type variables in our type and create a new type scheme that binds them by quantifying over them: It *closes* over free type variables.

$$\overline{\Gamma}(\alpha \to \alpha) = \forall\alpha.\alpha \to \alpha$$

And now that the type variable α is quantified, Var is able to instantiate y to *both* an $(\alpha \to \alpha) \to (\alpha \to \alpha)$ and an $\mathsf{Int} \to \mathsf{Int}$.

Syntax-directed

Note that we have been using the syntax-directed rules: a more modern treatment of the original system. In the syntax-directed rules, the typing relation assigns types to terms [17, p.15], not type schemes as in [4]. The original system had six rules:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{Taut} \qquad \frac{\Gamma \vdash e : \sigma \qquad \sigma > \sigma'}{\Gamma \vdash e : \sigma'}\ \textsc{Inst} \qquad \frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathrm{FTV}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}\ \textsc{Gen}$$

$$\frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}\ \textsc{Comb} \qquad \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau}\ \textsc{Let}$$

As we can see, the syntax-directed rules have merged Taut and Inst into Var, and Gen into Let via $\overline{\Gamma}$. It is possible to prove that these two systems are equivalent, but the benefit of having just four rules is that we now have exactly one rule for each syntactic form of expression. This means that *the shape of the proof is identical to the shape of the syntax*. This is discussed in more detail in Section 6.4.

2.4.3 Dynamic Semantics

You might have noticed that the previous section was titled *Static Semantics*. Whenever we defined the typing relation and the typing rules, we assigned a sense of meaning to these types. For example, something of type $\tau' \to \tau$ is going to be a function, because our Abs rule dictates so. These semantics are known *statically*. Without knowing what any of our programs evaluate to, we can work out their types.

Usually want to be able to show that these static semantics are *sound*, which in essence means if a program has a type τ, it actually evaluates to something of type τ[2]. But to show this, we first need to define what it means to evaluate a program. This is called the *dynamic semantics* of a language.

There are multiple ways to define the dynamic semantics. In the original work, Milner and Damas defined a *denotational* semantics for the language [11, 3], which describes evaluation in terms of mathematical objects and functions operating on them. For brevity, we will omit the details here, and relegate them to Appendix A. The important part is that once the dynamic semantics are in place, it is possible to define semantic entailment:

$$\Gamma \vDash e : \tau$$

Which can be read as, *"In the context* Γ*,* e *evaluates to a value in* τ*"*. Soundness then boils down to showing that if e has type τ, e does actually evaluate to a value in τ.

$$\Gamma \vdash e : \tau \to \Gamma \vDash e : \tau$$

[2]Type soundness is discussed in more detail in Section 4.1

Chapter 3

The Resourceful System

In this chapter we will formalise our resource-tracking, monadic IO language, based on the Hindley-Damas-Milner type system that we saw in the previous chapter. Just like before we will define the syntax, static semantics and dynamic semantics, but to avoid ambiguity we will also redefine concepts from HDM in more detail.

3.1 Syntax

We begin by giving the grammar as shown in figure 3.1. The expression language is an extension of the lambda calculus with let polymorphism. As before we use x, $\lambda x.e$ and $e\ e'$ to refer to variable lookup, abstraction and application respectively, and $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ for polymorphically binding e to x inside e'.

Our language also contains product types, $\tau \times \tau'$, which are the same as tuples `(a, b)` in Haskell. New product types can be introduced with $e \times e'$, and can similarly be eliminated with the projection expressions $\pi_1\ e$ and $\pi_2\ e$. $\square$ is the unit type (`()` in Haskell), and has the eponymous constructor $\square$.

More interestingly we have the monadic additions. Unlike Krishnaswami [9] we do not separate the language into expression and computation languages. $[\![e]\!]$, lifts a regular expression into the IO monad (`return` in Haskell), and $e \gg\!= e'$ is the standard monadic sequencing or binding operation.

Resources r represent something that we want to keep track of and prevent from being accessed concurrently, for example a file system or database. The resources we chose for the grammar are such examples.

Heaps ρ are then formed from resources. They differ from the notion of heaps in separation logic, and are constructed from either single resources or by merging other heaps. They are used to tag the resources that a computation in an IO monad might be accessing. There is also the notion of a World heap which encapsulates all possible resources and sub-heaps. It can be thought of as the ordinary `IO` monad in Haskell, where the outside world is one indivisible resource causing everything to be executed sequentially.

$$
\begin{array}{lll}
\text{type } \tau & ::= & \square \mid \alpha \mid \tau \to \tau' \mid \tau \times \tau' \mid \mathsf{IO}_\rho \tau \\
\\
\text{type scheme } \sigma & ::= & \forall \alpha.\sigma \mid \tau \\
\\
\text{context } \Gamma & ::= & \cdot \mid \Gamma, x : \sigma \\
\\
\text{expression } e & ::= & \square \mid e \times e' \mid \pi_1\ e\ \pi_2\ e \\
& \mid & x \mid \lambda x.e \mid e\ e' \mid \mathsf{let}\ x = e\ \mathsf{in}\ e' \\
& \mid & [\![e]\!] \mid [\![e]\!]_r \mid e \gg\!= e' \mid e \curlyvee e' \\
\\
\text{resource } r & ::= & \mathsf{File} \mid \mathsf{Network} \mid \mathsf{Database} \mid \ldots \\
\\
\text{heap } \rho & ::= & r \mid \rho \cup \rho' \mid \mathsf{World}
\end{array}
$$

Figure 3.1: The grammar for our resourceful language.

$[\![e]\!]_r$, pronounced *e using r*, lifts a value into the IO monad similarly to $[\![e]\!]$, but creates a new heap with the resource r and tags the monad with it. Novel to our language is the $e \curlyvee e'$ operator. It joins two monadic computations together into one that uses both their resources, returning a new heap made by merging e and e's heaps — provided that they do not overlap and are well formed. Heap well formedness will be defined later on.

Free type variables are defined on types, type schemes and contexts as follows:

$$
\begin{aligned}
\mathrm{FTV}(\square) &= \emptyset \\
\mathrm{FTV}(\alpha) &= \{\alpha\} \\
\mathrm{FTV}(\tau \to \tau') &= \mathrm{FTV}(\tau) \cup \mathrm{FTV}(\tau') \\
\mathrm{FTV}(\mathsf{IO}_\rho \tau) &= \mathsf{IO}_\rho\, \mathrm{FTV}(\tau) \\
\mathrm{FTV}(\forall \alpha_1, \ldots, \alpha_n.\tau) &= \mathrm{FTV}(\tau) - \{\alpha_1, \ldots, \alpha_n\} \\
\mathrm{FTV}(\Gamma) &= \bigcup_{x:\sigma \in \Gamma} \mathrm{FTV}(\sigma)
\end{aligned}
$$

Free variables are defined on terms — be careful not to mix these up with free type variables!

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} & \mathrm{FV}(\lambda x.e) &= \mathrm{FV}(e) \setminus \{x\} \\
\mathrm{FV}(e\ e') &= \mathrm{FV}(e) \cup \mathrm{FV}(e') & \mathrm{FV}(\mathsf{let}\ x = e\ \mathsf{in}\ e') &= \mathrm{FV}(e) \cup (\mathrm{FV}(e') \setminus \{x\}) \\
\mathrm{FV}([\![e]\!]) &= \mathrm{FV}(e) & \mathrm{FV}(e \gg\!= e') &= \mathrm{FV}(e) \cup \mathrm{FV}(e') \\
\mathrm{FV}(\square) &= \emptyset & \mathrm{FV}([\![e]\!]_r) &= \mathrm{FV}(e) \\
\mathrm{FV}(e_1 \times e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) & \mathrm{FV}(\pi_i e) &= \mathrm{FV}\,(e)_{i=1,2} \\
\mathrm{FV}(e_1 \curlyvee e_2) &= \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2)
\end{aligned}$$

3.1.1 Substitution

Substitution may seem self-explanatory, but it is important we define it crystal clear. It plays a vital role in proving properties about our system, and there exists subtle differences with how it is defined on terms, types and type schemes. A substitution is a map from type variables to types, written as $\tau'[\tau/\alpha]$ to replace the type the type τ for the type variable α in the type τ'.

$$\begin{aligned}
\square[\tau/\alpha] &= \square \\
\alpha'[\tau/\alpha] &= \begin{cases} \tau & \mathsf{if}\ \alpha' = \alpha \\ \alpha' & \mathsf{otherwise} \end{cases} \\
(e \times e')[\tau/\alpha] &= e[\tau/\alpha] \times e'[\tau/\alpha] \\
(\tau_1 \to \tau_2)[\tau/\alpha] &= (\tau_1[\tau/\alpha] \to \tau_2[\tau/\alpha]) \\
(\mathsf{IO}_\rho \tau')[\tau/\alpha] &= \mathsf{IO}_\rho \tau'[\tau/\alpha]
\end{aligned}$$

Substitution is associative, and a substitution that substitutes multiple type variables at once can be written like $[\tau_1/\alpha_1, \ldots, \tau_m/\alpha_m]$. It is also extended to type schemes, but note that this only substitutes *free* type variables.

$$\begin{aligned}
\forall\alpha'.\sigma[\tau/\alpha] &= \begin{cases} \forall\alpha'.\sigma & \mathsf{if}\ \alpha' = \alpha \\ \forall\alpha'.\sigma[\tau/\alpha] & \mathsf{otherwise} \end{cases} \\
\tau[\tau/\alpha] &= \tau[\tau/\alpha]\ \text{(Substituion on type)}
\end{aligned}$$

It should not be confused with instantiation, where the *bound* type variables $\forall\alpha_1, \ldots, \alpha_n$ are substituted inside the type of a type scheme. Instantiation is not a function though, it is a relation $\sigma > \tau$ in which we say σ can be instantiated to τ.

$$\boxed{\sigma > \tau}$$

$$\frac{\mathrm{dom}(s) = \{\alpha_1, \ldots, \alpha_n\} \qquad \tau'[s] = \tau}{\forall\alpha_1, \ldots, \alpha_n.\tau' > \tau}$$

The domain of a substitution, $\mathrm{dom}(s)$, is the set of type variables that it will replace, for example:

$$\mathrm{dom}([\tau_1/\alpha, \tau_2/\beta]) = \{\alpha, \beta\}$$

Instantiation can then be read as, if there exists a substitution s that substitutes exactly all the bound type variables in the type scheme $\forall\alpha_1, \ldots, \alpha_n.\tau'$ to give the type τ, then $\forall\alpha_1, \ldots, \alpha_n.\tau' > \tau$.

Furthermore we define the relation $\sigma > \sigma'$ on type schemes as well, and say that σ *is more general than* σ' if for all τ, $\sigma' > \tau \to \sigma > \tau$.

Substitution (not instantiation) is also defined on contexts. Substitution on contexts only substitutes *free* type variables.

$$\begin{aligned} \bullet[\tau/\alpha] &= \bullet \\ (\Gamma, x : \sigma)[\tau/\alpha] &= \Gamma[\tau/\alpha], x : \sigma[\tau/\alpha] \end{aligned}$$

There also exist term-level substitutions which map variables to other terms, and can be applied as follows:

$$\begin{aligned} \Box[v/\alpha] &= \Box \\ \alpha'[v/\alpha] &= \begin{cases} v & \text{if } \alpha' = \alpha \\ \alpha' & \text{otherwise} \end{cases} \\ (e\ e')[v/\alpha] &= e[v/\alpha]\ e'[v/\alpha] \\ \lambda x.e[v/\alpha] &= \begin{cases} \lambda x.e & \text{if } x = \alpha \\ \lambda x.(e[v/\alpha]) & \text{otherwise} \end{cases} \\ \mathsf{let}\ x = e'\ \mathsf{in}\ e[v/\alpha] &= \begin{cases} \mathsf{let}\ x = (e'[v/\alpha])\ \mathsf{in}\ e & \text{if } x = \alpha \\ \mathsf{let}\ x = (e'[v/\alpha])\ \mathsf{in}\ (e[v/\alpha]) & \text{otherwise} \end{cases} \\ (e_1 \times e_2)[v/\alpha] &= e_1[v/\alpha] \times e_2[v/\alpha] \\ (\pi_i\ e)[v/\alpha] &= \pi_i(e[v/\alpha])_{i=1,2} \\ [\![e]\!][v/\alpha] &= [\![e[v/\alpha]]\!] \\ [\![e]\!]_r[v/\alpha] &= [\![e[v/\alpha]]\!]_r \\ (e \gg\!= e')[v/\alpha] &= e[v/\alpha] \gg\!= e'[v/\alpha] \\ (e_1 \curlyvee e_2)[v/\alpha] &= e_1[v/\alpha] \curlyvee e_2[v/\alpha] \end{aligned}$$

3.1.2 Barendregt's variable convention

Sometimes, we will end up in a scenario with two separate expressions such as $\lambda x.x$ and $(y\ x)$. We know that the x inside the first expression is distinct from the x in the second expression, but when working with proofs we will need to show this somehow. We will use the Barendregt variable convention [1] to deal with this: If we want to show that the x inside $\lambda x.x$ is different from the x in $(y\ x)$, we can use *alpha conversion* to rename x to z and get $\lambda z.z$. This new expression is in fact equivalent to $\lambda x.x$, and we can always choose a new

unique name to avoid collisions with any *free variables*. The Barendregt variable convention says that whenever we have a bound variable (an x inside a $\lambda x.e$ or $\mathsf{let}\ x = e'\ \mathsf{in}\ e$), we can just assume that we have performed alpha conversion on it so that the variable name is unique.

3.2 Static semantics

Our static semantics begin with the syntax-directed rules of the Hindley-Damas-Milner system for the typing relation $\Gamma \vdash e : \tau$.

$$\frac{x : \sigma \in \Gamma \qquad \sigma > \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}\ \textsc{App}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e' : \tau' \qquad \Gamma, x : \overline{\Gamma}(\tau') \vdash e : \tau}{\Gamma \vdash \mathsf{let}\ x = e'\ \mathsf{in}\ e : \tau}\ \textsc{Let}$$

Like before, $\overline{\Gamma}$ is defined as

$$\overline{\Gamma}(\tau) = \forall \alpha_1 \ldots \alpha_n.\tau\ \mathsf{where}\ \{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\Gamma)$$

The typing rule for $\square$ expressions introduces the monomorphic type, and the rule for products introduces $e \times e'$.

$$\frac{}{\Gamma \vdash \square : \square}\ \textsc{Unit} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e \times e' : \tau \times \tau'}\ \textsc{Product}$$

Product types also have two eliminators, which project out the inner type.

$$\frac{\Gamma \vdash e : \tau \times \tau'}{\Gamma \vdash \pi_1 e : \tau}\ \textsc{Proj1} \qquad \frac{\Gamma \vdash e : \tau \times \tau'}{\Gamma \vdash \pi_2 e : \tau'}\ \textsc{Proj2}$$

Now we introduce the monadic parts of the language. Our language only has one type of monad, the IO monad, which is parameterised by both its *heap* ρ and its encapsulated type. Monadic values can be introduced into the language with $[\![e]\!]$, which lifts a pure term into **any** *well formed* heap.

$$\frac{\Gamma \vdash e : \tau \qquad \mathsf{ok}\ \rho}{\Gamma \vdash [\![e]\!] : \mathsf{IO}_\rho \tau}\ \textsc{Lift}$$

We need to be careful what heaps we allow terms to be lifted into, as the entire point of this system is to avoid heaps containing duplicate resources. It would be all too easy to introduce a nonsensical type like $\mathsf{IO}_{\mathsf{File} \cup \mathsf{File}} \tau$ with Lift, if it were not for the premise $\mathsf{ok}\ \rho$.

$\mathsf{ok}\ \rho$ is a new relation we define in Figure 3.2 to establish what heaps we consider to be well formed, in a similar vein to Krishnaswami [9]. All heaps

$$\boxed{\mathsf{ok}\ \rho} \qquad \frac{}{\mathsf{ok\ World}} \qquad \frac{}{\mathsf{ok}\ r} \qquad \frac{\mathsf{ok}\ \rho \qquad \mathsf{ok}\ \rho' \qquad \rho \cap \rho' = \emptyset}{\mathsf{ok}\ \rho \cup \rho'}$$

Figure 3.2: Rules for heap well formedness.

consisting of a single resource are well formed, as well as heaps made by merging two other well formed heaps *that are distinct*, i.e. they do not share any of the same resources.

Whilst Lift lets us construct monads in any heap, we will eventually want to have constructors for IO monads that use specific resources. When first designing the system, we used placeholder functions that just used fixed resources:

$$\frac{}{\Gamma \vdash \mathsf{readFile} : \mathsf{IO}_{\mathsf{File}}\square} \qquad \frac{}{\Gamma \vdash \mathsf{readNetwork} : \mathsf{IO}_{\mathsf{Network}}\square}$$

readFile and readNet are examples of typical operations that can consume a specific resource — their heap consists of just a single resource. This was then generalised to $[\![e]\!]_r$, which lifts any pure term into an IO monad with a heap consisting of the resource r.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [\![e]\!]_r : \mathsf{IO}_r e}\ \textsc{Use}$$

Note that it only uses a single resource, not any arbitrary heap. A heap with just one resource is always well formed, so there is no need for an ok r premise. Use is meant to be used to annotate terms with that a computation needs for itself, whilst on the other hand Lift is for bringing pure expressions into a monadic computation.

Once we have an IO value, we can sequence computation by binding it with a function that returns another IO.

$$\frac{\Gamma \vdash e : \mathsf{IO}_\rho \tau' \qquad \Gamma \vdash e' : \tau' \to \mathsf{IO}_\rho \tau}{\Gamma \vdash e \gg\!= e' : \mathsf{IO}_\rho \tau}\ \textsc{Bind}$$

Note that the types of the two IOs must use the same resources. However, we want expressions such as

$$[\![\square]\!]_{\mathsf{File}} \gg\!= \lambda x.[\![\square]\!]_{\mathsf{Net}}$$

to also be well typed. In particular, we want the above expression to be of type $\mathsf{IO}_{\mathsf{File} \cup \mathsf{Net}}\square$. But if Bind requires the heaps to be the same, we must first somehow "cast" readFile and readNet to the same type. This is the purpose of the Sub (subsumption) rule:

$$\frac{\Gamma \vdash e : \mathsf{IO}_{\rho'} \tau \qquad \rho' \geq: \rho \qquad \mathsf{ok}\ \rho}{\Gamma \vdash e : \mathsf{IO}_\rho \tau}\ \textsc{Sub}$$

$$\boxed{\rho' \geq: \rho}$$

$$\frac{}{\mathsf{World} \geq: \rho}\ \textsc{Top} \qquad \frac{}{\rho \geq: \rho}\ \textsc{Refl} \qquad \frac{\rho' \geq: \rho}{\rho' \cup \rho'' \geq: \rho}\ \textsc{UnionL} \qquad \frac{\rho' \geq: \rho}{\rho'' \cup \rho' \geq: \rho}\ \textsc{UnionR}$$

Figure 3.3: Subheap rules

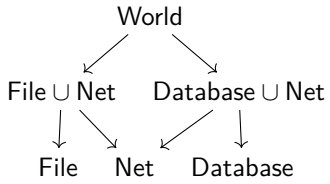

Figure 3.4: An example of some heaps and their subheap orderings.

It lets something of type $\mathsf{IO}_{\rho'}\tau$ become a $\mathsf{IO}_{\rho}\tau$, provided that the heap ρ is a *subheap* of ρ'. The subheap rules for heaps, shown in figure 3.3, define the $\geq:$ relation.

Intuitively, a heap σ can thought of being a subheap of another heap σ', if σ' subsumes σ, similarly to how subtyping works. For example, $\mathsf{Net} \geq: \mathsf{Net} \cup \mathsf{File}$, since $\mathsf{Net} \cup \mathsf{File}$ "overlaps" with the heap Net.

If one views this relation as an ordering, then we have World defined as the least upper bound — this represents using all possible resources, and as mentioned earlier $\mathsf{IO}_{\mathsf{World}}$ can be thought of as the *IO* monad in Haskell, where sequencing interacts with the state of the entire world. In a sense this could be viewed as a form of subtyping, but by constraining it to just resources and not actual types, we avoid the extra overhead and complexity subtyping would normally give us in the presence of type inference [5].

The subheaping relation is both reflexive (by definition), and transitive.

Theorem 1. *For all a, b, c, if $c \geq: b$ and $b \geq: a$ then $c \geq: a$.*

Proof. By induction on $c \geq: b$. See proof B.1 in the appendix. □

The purpose of this type system is to allow programs to be run concurrently, but reject the ones that concurrently access the same resource. In this regard, the Conc rule is the heart and soul of the system. It takes two monadic expressions, merges their heaps and returns a product of their two inner types, inside IO.

$$\frac{\Gamma \vdash e_1 : \mathsf{IO}_{\rho_1}\tau_1 \qquad \Gamma \vdash e_2 : \mathsf{IO}_{\rho_2}\tau_2 \qquad \mathsf{ok}\ \rho_1 \cup \rho_2}{\Gamma \vdash e_1 \curlyvee e_2 : \mathsf{IO}_{\rho_1 \cup \rho_2}\ \tau_1 \times \tau_2}\ \textsc{Conc}$$

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{x : \sigma \in \Gamma \qquad \sigma > \tau}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}\ \textsc{App} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash e' : \tau' \qquad \Gamma, x : \overline{\Gamma}(\tau') \vdash e : \tau}{\Gamma \vdash \mathsf{let}\ x = e'\ \mathsf{in}\ e : \tau}\ \textsc{Let} \qquad \frac{}{\Gamma \vdash \square : \square}\ \textsc{Unit}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e \times e' : \tau \times \tau'}\ \textsc{Product} \qquad \frac{\Gamma \vdash e : \tau \times \tau'}{\Gamma \vdash \pi_1 : \tau}\ \textsc{Proj1} \qquad \frac{\Gamma \vdash e : \tau \times \tau'}{\Gamma \vdash \pi_2 : \tau'}\ \textsc{Proj2}$$

$$\frac{\Gamma \vdash e : \tau \qquad \mathsf{ok}\ \rho}{\Gamma \vdash [\![e]\!] : \mathsf{IO}_\rho \tau}\ \textsc{Lift} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash [\![e]\!]_r : \mathsf{IO}_r \tau}\ \textsc{Use}$$

$$\frac{\Gamma \vdash e : \mathsf{IO}_\rho \tau' \qquad \Gamma \vdash e' : \tau' \to \mathsf{IO}_\rho \tau}{\Gamma \vdash e \gg\!= e' : \mathsf{IO}_\rho \tau}\ \textsc{Bind}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{IO}_{\rho_1} \tau_1 \qquad \Gamma \vdash e_2 : \mathsf{IO}_{\rho_2} \tau_2 \qquad \mathsf{ok}\ \rho_1 \cup \rho_2}{\Gamma \vdash e_1 \curlyvee e_2 : \mathsf{IO}_{\rho_1 \cup \rho_2}\ \tau_1 \times \tau_2}\ \textsc{Conc}$$

$$\frac{\Gamma \vdash e : \mathsf{IO}_{\rho'} \tau \qquad \rho' \geq: \rho \qquad \mathsf{ok}\ \rho}{\Gamma \vdash e : \mathsf{IO}_\rho \tau}\ \textsc{Sub}$$

Figure 3.5: The typing rules for our resourceful language.

The premises ensure that the new merged heap must be well formed, as we do not want to allow programs that try to use the same resource concurrently, such as

$$[\![\square]\!]_{\mathsf{File}} \curlyvee [\![\square]\!]_{\mathsf{File}}$$

We do however, want to allow programs that run two expressions that do not share any resources:

$$[\![\square]\!]_{\mathsf{File}} \curlyvee [\![\square]\!]_{\mathsf{Net}} : \mathsf{IO}_{\mathsf{File} \cup \mathsf{Net}}$$

All together these rules define the typing relation, and are displayed in full in Figure 3.5. To give a better idea of how they are used in practice, we will look at some examples of various programs and their types.

3.2.1 Examples

Monadic binding

Here we show a proof tree for the expression

$$\mathsf{let}\ x = \lambda y.[\![\Box]\!]\ \mathsf{in}\ (x\ \Box) \gg\!= \lambda z.[\![z]\!]_{\mathsf{File}}$$

The lift operator in $\lambda y.[\![\Box]\!]$ can lift into any heap. But the smallest heap we can use for this overall program is File, which is needed by $\lambda z.[\![z]\!]_{\mathsf{File}}$. So when we choose a heap to lift $\Box$ into, we lift it into File.

$$\dfrac{\dfrac{\bullet, y : \alpha \vdash \Box : \Box}{\bullet, y : \alpha \vdash [\![\Box]\!] : \mathsf{IO}_{\mathsf{File}}\Box}\ \mathsf{ok\ File}}{\bullet \vdash \lambda y.[\![\Box]\!] : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box \quad (\mathbf{1})}$$

$$\dfrac{\dfrac{\bullet, x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box \vdash x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box \qquad \bullet, x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box \vdash \Box : \Box}{\bullet, x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box \vdash x\ \Box : \mathsf{IO}_{\mathsf{File}}\Box} \qquad \dfrac{\dfrac{z : \alpha \in \bullet, x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box, z : \alpha \quad \alpha > \Box}{\bullet, x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box, z : \alpha \vdash z : \Box}}{\bullet, x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box, z : \alpha \vdash [\![z]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\Box} \over \bullet, x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box \vdash \lambda z.[\![z]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\Box}}{\bullet, x : \alpha \to \mathsf{IO}_{\mathsf{File}}\Box \vdash (x\ \Box) \gg\!= \lambda z.[\![z]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\Box \quad (\mathbf{2})}$$

$$\dfrac{(\mathbf{1}) \quad (\mathbf{2})}{\bullet \vdash \mathsf{let}\ x = \lambda y.[\![\Box]\!]\ \mathsf{in}\ (x\ \Box) \gg\!= \lambda z.[\![z]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\Box}$$

Let polymorphism

This example doesn't contain any resourceful elements, but is just an example of how let polymorphism allows the same variable to be given different types at each call site. Note how the premises in (**1**) and (**2**) assign x different types.

$$\dfrac{\cdot, x : \alpha \to \alpha \vdash x : (\Box \to \Box) \to (\Box \to \Box) \qquad \cdot, x : \alpha \to \alpha \vdash x : \Box \to \Box}{\bullet, x : \alpha \to \alpha \vdash x\ x : \Box \to \Box \qquad (\mathbf{1})}$$

$$\dfrac{\cdot, x : \alpha \to \alpha \vdash x : \Box \to \Box \qquad \cdot, x : \alpha \to \alpha \vdash \Box : \Box}{\bullet, x : \alpha \to \alpha \vdash x\ \Box : \Box \qquad (\mathbf{2})}$$

$$\dfrac{\dfrac{\dfrac{y : \alpha \in \bullet, y : \alpha \qquad \alpha > \alpha}{\bullet, y : \alpha \vdash y : \alpha}}{\bullet \vdash \lambda y.y : \alpha \to \alpha} \qquad \dfrac{(\mathbf{1}) \qquad (\mathbf{2})}{\bullet, x : \alpha \to \alpha \vdash (x\ x)\ (x\ \Box) : \Box}}{\bullet \vdash \mathsf{let}\ x = \lambda y.y\ \mathsf{in}\ (x\ x)\ (x\ \Box) : \Box}$$

Concurrency

Here is our first example of accessing two resources concurrently — with a bit of imagination one can think of this as reading a file from a disk whilst

simultaneously fetching data over the network.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\blacksquare, x : \alpha \vdash x : \alpha}\ \mathsf{ok\ File}}{\blacksquare, x : \alpha \vdash [\![x]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\alpha}}{\blacksquare \vdash \lambda x.[\![x]\!]_{\mathsf{File}} : \alpha \to \mathsf{IO}_{\mathsf{File}}\alpha}}{\blacksquare \vdash (\lambda x.[\![x]\!]_{\mathsf{File}})\ \square : \mathsf{IO}_{\mathsf{File}}\square} \qquad \dfrac{\dfrac{}{\blacksquare \vdash \square : \square}\ \mathsf{ok\ Net}}{\blacksquare \vdash [\![\square]\!]_{\mathsf{Net}} : \mathsf{IO}_{\mathsf{Net}}\square}}{\blacksquare \vdash (\lambda x.[\![x]\!]_{\mathsf{File}})\ \square) \curlyvee [\![\square]\!]_{\mathsf{Net}} : \mathsf{IO}_{\mathsf{File}\cup\mathsf{Net}}\square \times \square}\ \mathsf{File} \cap \mathsf{Net} = \emptyset$$

Subsumption

Assume there is a function inside our context called writeFile with the type $\square \to \mathsf{IO}_{\mathsf{File}}\square$. We can subsume its heap to be part of a larger heap, like File∪Net. We will show this with the expression

$$[\![\square]\!]_{\mathsf{File}} \curlyvee [\![\square]\!]_{\mathsf{Net}} \gg\!= \lambda x.\mathsf{writeFile}\ (\pi_1\ x)$$

$$\Gamma = \blacksquare, \mathsf{writeFile} : \square \to \mathsf{IO}_{\mathsf{File}}\square$$

$$\dfrac{\dfrac{\Gamma \vdash \square : \square}{\Gamma \vdash [\![\square]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\square} \qquad \dfrac{\Gamma \vdash \square : \square}{\Gamma \vdash [\![\square]\!]_{\mathsf{Net}} : \mathsf{IO}_{\mathsf{Net}}\square}\ \mathsf{File} \cap \mathsf{Net} = \emptyset}{\Gamma \vdash [\![\square]\!]_{\mathsf{File}} \curlyvee [\![\square]\!]_{\mathsf{Net}} : \mathsf{IO}_{\mathsf{File}\cup\mathsf{Net}}\square}\ (\mathbf{1})$$

$$\dfrac{\Gamma, x : \square \times \square \vdash \mathsf{writeFile} : \square \to \mathsf{IO}_{\mathsf{File}}\square \qquad \dfrac{\Gamma, x : \square \times \square \vdash x : \square \times \square}{\Gamma, x : \square \times \square \vdash \pi_1\ x : \square}}{\Gamma, x : \square \times \square \vdash \mathsf{writeFile}\ (\pi_1\ x) : \mathsf{IO}_{\mathsf{File}}\square}\ (\mathbf{2})$$

$$\dfrac{(\mathbf{1}) \qquad \dfrac{\dfrac{(\mathbf{2}) \qquad \mathsf{File} \geq : \mathsf{File} \cup \mathsf{Net} \qquad \mathsf{ok\ File} \cup \mathsf{Net}}{\Gamma, x : \square \times \square \vdash \mathsf{writeFile}\ (\pi_1\ x) : \mathsf{IO}_{\mathsf{File}\cup\mathsf{Net}}\square}\ \textsc{Sub}}{\Gamma \vdash \lambda x.\mathsf{writeFile}\ (\pi_1\ x) : \mathsf{IO}_{\mathsf{File}\cup\mathsf{Net}}\square}}{\Gamma \vdash [\![\square]\!]_{\mathsf{File}} \curlyvee [\![\square]\!]_{\mathsf{Net}} \gg\!= \lambda x.\mathsf{writeFile}\ (\pi_1\ x) : \mathsf{IO}_{\mathsf{File}\cup\mathsf{Net}}\square}$$

3.3 Dynamic semantics

As shown in Chapter 2, the dynamic semantics in the original Hindley-Damas-Milner system was based on denotational semantics. In our type system, we will use operational semantics. Operational semantics are similar to what we have seen before in the definition of the static semantics. We define a bunch of inference rules, and from these build up proofs. Tofte [17] had the idea of using operational semantics for not just the typing rules, but also for the dynamic semantics. We choose this approach over denotational semantics as it unifies our approach to types and evaluation, and as Tofte said, *"it seems a bit unfortunate that we should have to understand domain theory to be able to investigate whether a type inference system admits faulty programs"*.

$$\boxed{\mathsf{value}\ e}$$

$$\frac{}{\mathsf{value}\ \square} \qquad \frac{\mathsf{value}\ e_1 \qquad \mathsf{value}\ e_2}{\mathsf{value}\ e_1 \times e_2} \qquad \frac{}{\mathsf{value}\ [\![e]\!]} \qquad \frac{}{\mathsf{value}\ [\![e]\!]_r} \qquad \frac{}{\mathsf{value}\ \lambda x.e}$$

Figure 3.6: Terminal values

3.3.1 Values

Before we can talk about how we evaluate a program, we need to define what constitutes a fully evaluated program. That is, what terms are a result of a completed computation. We define a unary relation called value, and give rules describing what terms can be considered values in Figure 3.6.

For example, we cannot evaluate the unit type any further, therefore all $\square$s are considered values. The same goes for product types, but only if both inner components are values themselves: $\square \times \square$ is a finished value, but $f\ e \times \square$ might still have evaluation left to do. A lambda on its own is a value too. Without being applied to an argument it cannot be evaluated any further: The computation inside of it is suspended. In a similar fashion a lifted computation cannot be computed any further *on its own*. We will see later how binding can run this computation, but by itself it will not evaluate to anything.

3.3.2 Small-step semantics

The approach to operational semantics we will be taking is *small-step operational semantics*. In small-step operational semantics we define a step relation $a \rightsquigarrow b$ which says that in one "step", *a reduces to b*. b might then go onto reduce further if it is able to, or it could be a finished value. So reduction can be thought of as evaluation of a program, bit by bit. Small-step semantics differs from big-step semantics, where the relation $a \Downarrow b$ says that at the end of the day, a will reduce to b, and b will not reduce any further.

As an example, if an expression e_1 reduces to e_1', i.e. $e_1 \rightsquigarrow e_1'$, then we want the application $e_1\ e_2$ to reduce as well. We can write this as

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2}$$

We call this type of reduction rule which takes smaller reductions and updates it within a bigger structure, ξ-reduction. There are ξ-reductions for other expressions with structure inside, namely product types, the concurrent operator and the bind operator.

Another type of reduction rule is β-reduction, which comes from the lambda calculus. When we apply an argument to an abstraction, we substitute the bound variable inside the abstraction with the argument. The β-reduction rule

defines this.

$$\frac{\mathsf{value}\ e_2}{(\lambda x.e_1)\ e_2 \rightsquigarrow e_1[e_2/x]}$$

It is important to note the premise here that states the argument being applied must be a value. This enforces a strict evaluation order, since in order for the argument to be a value it must be completely reduced. A lazily evaluated semantics might forgo this extra requirement, so that the argument can be reduced after substitution.

As mentioned earlier, lifted expressions $[\![e]\!]$ are suspended much like lambdas, and so are values since they cannot reduce any further *on their own*. However, with the bind operator, the value inside them can be extracted out and fed into an abstraction.

$$\frac{}{[\![e]\!] \gg\!= e' \rightsquigarrow e'\ e}$$

Unlike the semantic rule for β-reduction, there is no premise enforcing that $[\![e]\!]$ is a value, since all lifted terms are values anyway.

For an expression lifted into a resourceful IO monad with $[\![e]\!]_\rho$, one might be tempted to just reduce this to a $[\![e]\!]$.

$$\frac{}{[\![e]\!]_r \rightsquigarrow [\![e]\!]}$$

And we could then also define the reduction for concurrency like so:

$$\frac{}{[\![v]\!] \curlyvee [\![w]\!] \rightsquigarrow [\![v \times w]\!]}$$

However, the intermediate $[\![e]\!]$ can be confusing. The purpose of the lift operator is to lift a pure value into any possible resource bound monad. When we see a lift, we think of the typing judgement LIFT that allows it to fit any heap, when in fact a use should restrict what heaps it can go into. For these reasons, we instead define its reduction identically to lift.

$$\frac{}{[\![e]\!]_r \gg\!= e' \rightsquigarrow e'\ e}$$

Concurrency is then defined as chaining together two binds, and returning the lifted product of the two results.

$$\frac{}{v \curlyvee w \rightsquigarrow v \gg\!= \lambda v.(w \gg\!= \lambda w.[\![v \times w]\!])}$$

This might seem like the opposite of concurrency — executing the computation in sequence — but because our monad does not have any state (see Section 6.5),

$$v \gg\!= \lambda v.(w \gg\!= (\lambda w.[\![v \times w]\!]))$$

is identical to

$$w \gg\!= \lambda w.(v \gg\!= (\lambda v.[\![v \times w]\!]))$$

In fact the previous definition for concurrency as $[\![v]\!] \curlyvee [\![w]\!] \rightsquigarrow [\![v \times w]\!]$ will have the same reduction steps at the end of the day. In a real-world program-

$$\boxed{e \rightsquigarrow e'}$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2} \qquad \frac{e_2 \rightsquigarrow e_2'}{e_1\ e_2 \rightsquigarrow e_1\ e_2'} \qquad \frac{\mathsf{value}\ e'}{(\lambda x.e)\ e' \rightsquigarrow e[e'/x]}$$

$$\frac{}{\mathsf{let}\ x = e'\ \mathsf{in}\ e \rightsquigarrow e[e'/x]}$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \times e_2 \rightsquigarrow e_1' \times e_2} \qquad \frac{e_2 \rightsquigarrow e_2'}{e_1 \times e_2 \rightsquigarrow e_1 \times e_2'} \qquad \frac{e \rightsquigarrow e'}{\pi_1 e \rightsquigarrow \pi_1 e'} \qquad \frac{e \rightsquigarrow e'}{\pi_2 e \rightsquigarrow \pi_2 e'}$$

$$\frac{}{\pi_1(e_1 \times e_2) \rightsquigarrow e_1} \qquad \frac{}{\pi_2(e_1 \times e_2) \rightsquigarrow e_2} \qquad \frac{e_1 \rightsquigarrow e_1'}{e_1 \gg\!= e_2 \rightsquigarrow e_1' \gg\!= e_2}$$

$$\frac{}{[\![v]\!] \gg\!= e_2 \rightsquigarrow e_2\ v} \qquad \frac{}{[\![e]\!]_r \gg\!= e' \rightsquigarrow e'\ e}$$

$$\frac{}{v \curlyvee w \rightsquigarrow v \gg\!= \lambda v.(w \gg\!= \lambda w.[\![v \times w]\!])}$$

Figure 3.7: Dynamic Semantics

ming language implementation, the evaluation would actually be implemented concurrently.

Although the $v \curlyvee w$ might evaluate to two separate $\gg\!=$s, the static semantics of concurrency and binding are different. For example, given $\Gamma \vdash [\![v]\!] : \mathsf{IO}_\rho \tau$ we are allowed to bind $liftv$ with itself:

$$\Gamma \vdash [\![v]\!] \gg\!= \lambda v.([\![v]\!] \gg\!= (\lambda v.[\![v \times v]\!])) : \mathsf{IO}_\rho(\tau \times \tau)$$

But with the concurrent operator, this is a type error, since the premise $\mathsf{ok}\ \rho \cup \rho$ does not hold: There is no type in any context that can be given to $v \curlyvee v$.

3.3.3 The reflexive and transitive closure

We can go a *step* further and define $\twoheadrightarrow$ as the reflexive, transitive closure of $\rightsquigarrow$. What does that mean? If $\rightsquigarrow$ is a binary relation of two terms, then the transitive closure $\twoheadrightarrow$ is a new relation that maintains all the relations of $\rightsquigarrow$ and is transitive, i.e. if $a \twoheadrightarrow b$ and $b \twoheadrightarrow c$, then $a \twoheadrightarrow c$. A reflexive transitive closure extends this so that for any a, $a \twoheadrightarrow a$.

Intuitively speaking, if the small-step inference rule $a \rightsquigarrow b$ says that a reduces to b in exactly one step, then $a \twoheadrightarrow b$ says a reduces to b in zero or more steps. Instead of reducing one step at a time, it can go all the way.

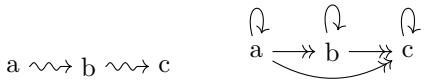

Figure 3.8: Left: the $\rightsquigarrow$ relation. Right: The reflexive, transitive closure of $\rightsquigarrow$, the reduction relation $\twoheadrightarrow$.

For example, if we have the expression $(\lambda x.x\ \square)\ (\lambda y.y)$ then have the following relations:

$$(\lambda x.x\ \square)\ (\lambda y.y) \rightsquigarrow (\lambda y.y)\ \square \rightsquigarrow \square$$

$$(\lambda x.x\ \square)\ (\lambda y.y) \twoheadrightarrow \square$$

We do not use this relation to prove any properties about our system, but it sheds light on how our expression language eventually reduces to a value in the end.

Chapter 4

Properties

A type system is no good unless we can trust that the programs actually evaluate to the type they are given. In this section we will prove a number of lemmas and theorems which will eventually show that the type system is sound.

Before we begin, we need to show a couple of properties about contexts that crop up later on in our proofs. The first states that if we have two variables in the context with the same names but different type schemes, then we can ignore the first one as it is overshadowed by the second — any reference to x will result in $x : \sigma'$.

Lemma 1 (Drop)**.** *If* $\Gamma, x : \sigma, x : \sigma' \vdash e : \tau$*, then* $\Gamma, x : \sigma' \vdash e : \tau$

We can also say that if two variables x and y are indeed different, we are free to permute them, swapping them about.

Lemma 2 (Swap)**.** *If* $x \neq y$ *and* $\Gamma, x : \sigma, y : \sigma' \vdash e : \tau$*, then* $\Gamma, y : \sigma', x : \sigma \vdash e : \tau$

And if we have a typing judgement in an empty context, we can weaken the judgement to extend this to any other context.

Lemma 3 (Weaken)**.** *If* $\cdot \vdash e : \tau$*, then for* Γ*,* $\Gamma \vdash e : \tau$*.*

We will also need to show this property about instantiation and the close function.

Lemma 4. *If* $\sigma > \sigma'$ *then* $\overline{\Gamma, x : \sigma}(\tau) > \overline{\Gamma, x : \sigma'}(\tau)$*.*

Proof. We provide an informal proof as follows.

1. If $\sigma > \sigma'$, then for every τ that $\sigma > \tau$, $\sigma' > \tau$.
2. So σ must parameterise over at least the same number of type variables if not *more* than σ'.
3. So σ has at least the same number of free type variables if not *less* than σ'.

4. By definition of the close function, $\overline{\Gamma, x : \sigma}(\tau)$ will then have at least the same number of bound type variables if not *more* than $\overline{\Gamma, x : \sigma'}(\tau)$.

5. So if $\overline{\Gamma, x : \sigma'}(\tau)$ can be instantiated to some τ, then $\overline{\Gamma, x : \sigma}(\tau)$ can also be instantiated to that τ, as it has enough bound type variables to handle everything the former can — if it had less type variables than the former, then it would not be able to instantiate these types, but for the converse excess type variables can be mapped to whatever.

□

Now we prove the generalisation theorem, which states that if a type scheme inside the context is an instantiation of another type scheme, then we can use the more general type scheme and preserve how expressions are typed. This is an adaptation of a lemma from Wright and Felleisen [20, Lemma 4.6], which is in turn an adaptation of a lemma from Damas and Milner.

Theorem 2 (Generalisation)**.** *If* $\Gamma, x : \sigma' \vdash e : \tau$ *and* $\sigma > \sigma'$*, then* $\Gamma, x : \sigma \vdash e : \tau$*.*

Proof. Begin with induction on the proof for $\Gamma, x : \sigma' \vdash e : \tau$.

Var From the premises we have $x : \sigma' \in \Gamma$ and $\sigma' > \tau$. We also have $\sigma > \sigma'$ but by definition of $\sigma > \sigma'$, if $\sigma' > \tau$ then $\sigma > \tau$. And we also have by definition $x : \sigma \in \Gamma, x : \sigma$. So putting the pieces together, we can use Var to get

$$\frac{x : \sigma \in \Gamma, x : \sigma \qquad \sigma > \tau}{\Gamma, x : \sigma \vdash x : \tau}$$

Abs We have $\Gamma, x : \sigma' \vdash \lambda y.e : \tau$. We want to be able to apply the swap lemma, Lemma 2, so we can get $\Gamma, y : \sigma'', x : \sigma' \vdash e : \tau$. But for that we need proof that $x \neq y$. It is possible to consider the case where $x = y$ and prove it separately, but to make life easier we will use the Barendregt variable convention and instead assume $\lambda y.e$ has undergone some α-conversion such that $x \neq y$.

Now with our $\Gamma, y : \sigma'', x : \sigma' \vdash e : \tau$, using the induction hypothesis results in $\Gamma, y : \sigma'', x : \sigma \vdash e : \tau$. We can then swap x back again to get $\Gamma, x : \sigma, y : \sigma'' \vdash e : \tau$, and then use Abs for $\Gamma, x : \sigma \vdash \lambda y.e : \tau$ as needed.

Let For $\mathsf{let}\ y = e'\ \mathsf{in}\ e$, we have $\Gamma, x : \sigma' \vdash e' : \tau'$ and $\Gamma, x : \sigma', y : \overline{\Gamma, x : \sigma'}(\tau') \vdash e : \tau$ and we aim to show

$$\Gamma, x : \sigma \vdash \mathsf{let}\ y = e'\ \mathsf{in}\ e : \tau$$

First off, we need to convert the $y : \overline{\Gamma, x : \sigma'}(\tau')$ to a $y : \overline{\Gamma, x : \sigma}(\tau')$ somehow. But it can be shown that if $\sigma > \sigma'$, then $\overline{\Gamma, x : \sigma}(\tau) > \overline{\Gamma, x : \sigma'}(\tau)$ due to Lemma 4. So by the inductive hypothesis, we are able to get $\Gamma, x : \sigma', y : \overline{\Gamma, x : \sigma}(\tau') \vdash e : \tau$.

Because y is bound, we know that $x \neq y$ from Barendregt's variable convention. We take the following steps:

$$\begin{aligned}
&\Gamma, x : \sigma', y : \overline{\Gamma, x : \sigma}(\tau') \vdash e : \tau \\
&\Gamma, y : \overline{\Gamma, x : \sigma}(\tau'), x : \sigma' \vdash e : \tau && \text{by swapping, Lemma 2} \\
&\Gamma, y : \overline{\Gamma, x : \sigma}(\tau'), x : \sigma \vdash e : \tau && \text{by inductive hypothesis} \\
&\Gamma, x : \sigma, y : \overline{\Gamma, x : \sigma}(\tau') \vdash e : \tau && \text{by swapping again}
\end{aligned}$$

And proceed to construct the proof for LET as previously.

APP From the premises we have $\Gamma, x : \sigma' \vdash e : \tau' \to \tau$ and $\Gamma, x : \sigma' \vdash e' : \tau'$. We wish to show $\Gamma, x : \sigma \vdash e\ e' : \tau$. We apply the induction hypothesis to get $\Gamma, x : \sigma \vdash e : \tau' \to \tau$ and $\Gamma, x : \sigma \vdash e' : \tau$. Then use APP to build up a proof of $\Gamma, x : \sigma \vdash e\ e' : \tau$.

The remaining cases The rest of the possible proofs for $\Gamma, x : \sigma' \vdash e : \tau$ can all be proved by applying the induction hypothesis on their structure, much like the case for APP, and so are omitted for brevity.

□

If a context has all the same type schemes for each free variable in an expression as another context, then we can make the same typing judgements with that other context. In other words, we are free to ignore extra variables not used by the expression.

Lemma 5. *If for all $x \in \mathrm{FV}(e)$ where $x : \sigma \in \Gamma$, $x : \sigma \in \Delta$, and if $\Gamma \vdash e : \tau$ then $\Delta \vdash e : \tau$.*

We can also show that we can apply a substitution to both the context and type given in a judgement.

Lemma 6. *If $\Gamma \vdash e : \tau$, then for any substitution s, $\Gamma[s] \vdash e : \tau[s]$.*

An important lemma that we need to show is the substitution lemma, which relates to substituting a variable for an expression, whose type we can prove. The following proof is taken from Wright and Felleisen [20].

Lemma 7 (Substitution)**.** *If $\Gamma \vdash e : \tau$ and $\Gamma, x : \forall\alpha_1, \ldots, \alpha_n.\tau \vdash e' : \tau'$, and $x \notin \mathrm{dom}(\Gamma)$ and $\alpha_1, \ldots, \alpha_n \cap \mathrm{FTV}(\Gamma) = \emptyset$, then $\Gamma \vdash e'[e/x] : \tau'$.*

Note that the domain of a context $\mathrm{dom}(\_, x_1 : \sigma_1, \ldots, x_n : \sigma_n)$ is defined as $\{x_1, \ldots, x_n\}$.

Proof. Begin with induction on the proof of $\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\tau \vdash e' : \tau'$.

VAR We have $\Gamma, x : \forall\alpha_1 \ldots \alpha_n \vdash y : \tau'$ and want to show $\Gamma \vdash y[e/x] : \tau'$.

If $y \neq x$, then by definition $y[e/x] = y$, so we just need to show $\Gamma \vdash y : \tau'$. From the premises we also have $y : \tau' \in \Gamma, x : \forall\alpha_1 \ldots \alpha_n$, but since $y \neq x$ we have $y : \tau' \in \Gamma$. And from here we can use VAR to get $\Gamma \vdash y : \tau'$.

If $y = x$, then we need to show $\Gamma \vdash x : \tau'$. With the premise

$$\Gamma, x : \forall \alpha_1 \dots \alpha_n . \tau \vdash x : \tau'$$

we also know $\forall \alpha_1 \dots \alpha_n . \tau > \tau'$, so there exists a substitution s which replaces exactly $\alpha_1, \dots, \alpha_n$, such that $\tau[s] = \tau'$.

We can use Lemma 6 to get $\Gamma[s] \vdash e : \tau[s]$, or $\Gamma[s] \vdash e : \tau'$. And furthermore, because $\alpha_1, \dots, \alpha_n \cap \text{FTV}(\Gamma) = \emptyset$, applying s to Γ will do nothing since Γ's free type variables are distinct, i.e. $\Gamma = \Gamma[s]$. Leaving us with $\Gamma \vdash e : \tau'$ as required.

Abs We have $\Gamma, x : \forall \alpha_1 \dots \alpha_n \vdash \lambda y.e' : \tau_1 \to \tau_2$ and want to show

$$\Gamma \vdash (\lambda y.e')[e/x] : \tau_1 \to \tau_2$$

y is bound, so by convention $y \neq x$. From the premises we have

$$\Gamma, x : \forall \alpha_1 \dots \alpha_n . \tau, y : \tau_1 \vdash e' : \tau_2$$

We start by choosing a substitution s, which maps $\alpha_1, \dots, \alpha_n$ to fresh type variables $\alpha'_1, \dots, \alpha'_n$. Additionally, they are distinct as follows.

$$\text{FTV}(\Gamma) \cap \alpha_1, \dots, \alpha_n \cap \alpha'_1, \dots, \alpha'_n = \emptyset$$

Now we manipulate the premise in the following order.

$$\begin{aligned}
\Gamma, y : \tau_1, x : \forall \alpha_1 \dots \alpha_n . \tau \vdash e' : \tau_2 \quad & \text{by swapping, Lemma 2} \\
(\Gamma, y : \tau_1, x : \forall \alpha_1 \dots \alpha_n . \tau)[s] \vdash e' : \tau_2 \quad & \text{by Lemma 6} \\
\Gamma, y : \tau_1[s], x : (\forall \alpha_1 \dots \alpha_n . \tau)[s] \vdash e' : \tau_2 \quad & \text{since } \Gamma[s] = \Gamma
\end{aligned}$$

And since the range of s is exactly $\alpha_1, \dots, \alpha_n$, we have $(\forall \alpha_1 \dots \alpha_n . \tau)[s] = \forall \alpha_1 \dots \alpha_n . \tau$ — substitution only substitutes free type variables, not bound type variables, and here any possible free type variables are being shadowed.

$$\Gamma, y : \tau_1[s], x : \forall \alpha_1 \dots \alpha_n . \tau \vdash e' : \tau_2 \tag{4.1}$$

We can apply Lemma 5 to $\Gamma \vdash e : \tau$, giving us

$$\Gamma, y : \tau_1[s] \vdash e : \tau \tag{4.2}$$

And when we combine $x \notin \text{dom}(\Gamma)$ with the fact that $x \neq y$, we get

$$x \notin \text{dom}(\Gamma, y : \tau_1[s]) \tag{4.3}$$

Because of how we chose s, we also have[1]

$$\text{FTV}(\Gamma, y : \tau_1[s]) \cap \alpha_1, \dots, \alpha_n = \emptyset \tag{4.4}$$

[1] y is distinct here because of the Barendregt variable convention.

Eventually, we use (4.2), (4.1), (4.3) and (4.4) with the induction hypothesis to arrive at

$$\Gamma, y : \tau_1[s] \vdash e'[e/x] : \tau_2[s]$$

But s is a bijection due to how we chose it: That means an inverse, s^{-1} exists. We "apply it to both sides" with Lemma 6

$$\begin{aligned}(\Gamma, y : \tau_1[s])[s^{-1}] &\vdash e'[e/x] : (\tau_2[s])[s^{-1}] \\ (\Gamma, y : \tau_1[s])[s^{-1}] &\vdash e'[e/x] : \tau_2 \\ \Gamma[s^{-1}], y : \tau_1[s][s^{-1}] &\vdash e'[e/x] : \tau_2 \\ \Gamma[s^{-1}], y : \tau_1 &\vdash e'[e/x] : \tau_2\end{aligned}$$

And because $\alpha'_1, \ldots, \alpha'_n \cap \mathrm{FTV}(\Gamma) = \emptyset$, we can get rid of that last substitution and arrive at $\Gamma, y : \tau_1 \vdash e'[e/x] : \tau_2$. From here, we build our way back up with Abs.

$$\begin{aligned}&\Gamma \vdash \lambda y.(e'[e/x]) : \tau_1 \to \tau_2 \\ &\Gamma \vdash (\lambda y.e')[e/x] : \tau_1 \to \tau_2 \qquad \text{because } x \neq y\end{aligned}$$

Let We have the proof $\Gamma, x : \sigma \vdash \mathsf{let}\ y = e_1\ \mathsf{in}\ e_2 : \tau'$ and want to show

$$\Gamma \vdash (\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2)[e/x] : \tau'$$

The first premise can be fed directly into the induction hypothesis

$$\begin{aligned}\Gamma, x : \forall \alpha_1 \ldots \alpha_n.\tau &\vdash e_1 : \tau_1 \\ \Gamma &\vdash e_1[e/x] : \tau_1\end{aligned} \tag{4.5}$$

Since y is bound and thus $y \neq x$, we use the swap lemma with the second premise to get

$$\Gamma, y : \overline{\Gamma, x : \forall \alpha_1 \ldots \alpha_n.\tau}(\tau_1), x : \forall \alpha_1 \ldots \alpha_n.\tau \vdash e_2 : \tau' \tag{4.6}$$

And since we have $\Gamma \vdash e : \tau$, from Lemma 5 there is

$$\Gamma, y : \overline{\Gamma, x : \forall \alpha_1 \ldots \alpha_n.\tau}(\tau_1) \vdash e : \tau \tag{4.7}$$

Now we want to call the inductive hypothesis on (4.6) and (4.7), but that means we first need to show

$$\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\Gamma, y : \overline{\Gamma, x : \forall \alpha_1 \ldots \alpha_n.\tau}(\tau_1)) = \emptyset$$

We do this as follows:

$$
\begin{aligned}
&\{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\Gamma, y : \overline{\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\tau}(\tau_1)) \\
&\subseteq \{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\overline{\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\tau}(\tau_1))) \\
&= \{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{FTV}(\overline{\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\tau}(\tau_1))) \\
&= \{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{FTV}(\tau_1) \setminus (\mathrm{FTV}(\tau_1) \setminus \mathrm{FTV}(\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\tau))) \\
&= \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\tau_1) \cap \mathrm{FTV}(\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\tau) \\
&\subseteq \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\tau) \\
&= \{\alpha_1, \ldots, \alpha_n\} \cap (\mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\forall\alpha_1 \ldots \alpha_n.\tau)) \\
&= \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\forall\alpha_1 \ldots \alpha_n.\tau) \\
&= \{\alpha_1, \ldots, \alpha_n\} \cap \mathrm{FTV}(\tau) \setminus \{\alpha_1, \ldots, \alpha_n\} \\
&= \emptyset
\end{aligned}
$$

And then we can use it to arrive at

$$\Gamma, y : \overline{\Gamma, x : \forall\alpha_1 \ldots \alpha_n.\tau}(\tau_1) \vdash e_2[e/x] : \tau'$$

The next part involves an observation, that for any x, τ and σ in any Γ, $\overline{\Gamma}(\tau) > \overline{\Gamma, x : \sigma}(\tau)$, as closing over a context with a *smaller* domain will result in the type scheme having *more* bound type variables — i.e. it is more general. Therefore we can use the generalisation lemma, Lemma 2, to get

$$
\begin{array}{rll}
\Gamma, y : \overline{\Gamma}(\tau_1) \vdash e_2[e/x] : \tau' & & \\
\Gamma \vdash \mathsf{let}\ y = e_1[e/x]\ \mathsf{in}\ e_2[e/x] : \tau' & & \text{with LET and (4.5)} \\
\Gamma \vdash (\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2)[e/x] : \tau' & & \text{by definition of substitution}
\end{array}
$$

The remaining cases The rest of the proofs for $\Gamma \vdash e'[e/x] : \tau'$ are proved by applying the induction hypothesis on their structure.

□

4.1 Type soundness

Now that we have the prerequisite lemmas out of the way, we can move onto proving that the system is **sound**. Soundness is a property of a logical system, stating that every judgement that can be proved within the system is *valid*. In the type theory realm, a typing judgement is valid if the expression actually evaluates to a member of that type. So a type system is considered to be sound if and only if, for every typing judgement that says an expression is well-typed, the expression evaluates to a member of that type — or as Milner put it, "*well-typed programs can't go wrong*" [11] [§3.7].

$$\Gamma \vdash e : \tau \to \Gamma \vDash e : \tau$$

We saw how this was shown with denotational semantics in Chapter 2, but here we have modelled our dynamic semantics with operational semantics. Instead, we prove the soundness of our type system *syntactically*. This approach was first introduced by Wright and Felleisen and 1994 [20], and has since become the de facto method for proving soundness with operational semantics.

Syntactic type soundness involves proving two properties of a type system, *type progress* and *type preservation*. The former states that for any typing judgement we can prove, the expression is either a terminal value, or can be reduced further. The latter property says that when an expression has a type, reducing it to another expression does not change the type.

We will prove the closed variants of these properties, that is they hold in the empty context. The proofs for these properties within our system are as follows.

Theorem 3 (Progress)**.** *If* $\cdot \vdash e : \tau$*, then either* $\mathsf{value}\ e$ *or there exists a* e' *such that* $e \rightsquigarrow e'$*.*

Proof. Begin by induction on $\cdot \vdash e : \tau$.

Var This case is not possible — it is impossible to construct a proof for $x \in \cdot$, and thus impossible to arrive at $\cdot \vdash x : \tau$.

Abs Any lambda abstraction is a value due to the rule

$$\frac{}{\mathsf{value}\ \lambda x.e}$$

App We have $\cdot \vdash e_1\ e_2 : \tau$, so by the premises of App, we have $\cdot \vdash e_1 : \tau' \to \tau$ and $\cdot \vdash e_2 : \tau'$.

Consider the induction hypothesis on the first premise. Either $\mathsf{value}\ e_1$, or $e_1 \rightsquigarrow e_1'$. In the latter case where e_1 reduces, we can then construct a proof that $e_1\ e_2$ reduces, and we're done.

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2}$$

In the former case where e_1 is a value, we then apply the induction hypothesis to $\cdot \vdash e_2 : \tau'$. Again, if e_2 reduces further then we can also construct a proof that $e_1\ e_2$ reduces.

But if $\mathsf{value}\ e_2$, we cannot reduce in the same way again. Instead, since we know e_1 is a value, and since we also know that $\cdot \vdash e_1 : \tau' \to \tau$, we can deduce that e_1 must be an abstraction: it has the form $\lambda x.e$. Furthermore, because e_2 is a value, we can then apply the beta reduction rule.

$$\frac{\mathsf{value}\ e_2}{(\lambda x.e)\ e_2 \rightsquigarrow e[e_2/x]}$$

We are calling a function in this case, and now have proof that it reduces.

LET Our dynamic semantics for LET are lazy: $\mathsf{let}\ x = e'\ \mathsf{in}\ e$ can reduce without evaluating e' first. And so for every let statement, we can immediately reduce.

$$\frac{}{\mathsf{let}\ x = e'\ \mathsf{in}\ e \rightsquigarrow e[e'/x]}$$

UNIT All unit expressions are values.

$$\frac{}{\mathsf{value}\ \square}$$

PRODUCT For $\cdot \vdash e_1 \times e_2 : \tau_1 \times \tau_2$, we have the premises $\cdot \vdash e_1 : \tau_1$ and $\cdot \vdash e_2 : \tau_2$. Using the induction hypothesis on either, if either of them reduce further, then we can also reduce $e_1 \times e_2$.

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \times e_2 \rightsquigarrow e_1' \times e_2} \qquad \frac{e_2 \rightsquigarrow e_2'}{e_1 \times e_2 \rightsquigarrow e_1 \times e_2'}$$

If both of them are values, then the product itself is a value.

$$\frac{\mathsf{value}\ e_1 \qquad \mathsf{value}\ e_2}{\mathsf{value}\ e_1 \times e_2}$$

PROJ1 For $\cdot \vdash \pi_1\ e : \tau$, we have the premise $\cdot \vdash e : \tau \times \tau'$. Applying the induction hypothesis to the premise, if it reduces further, then we can reduce the expression as well.

$$\frac{e \rightsquigarrow e'}{\pi_1\ e \rightsquigarrow \pi_1\ e'}$$

However, if e is a value then we know it must be of the form $e_1 \times e_2$, as that is the only expression that can have a product type and be a value. Thus we can use the reduction rule

$$\frac{}{\pi_1\ (e_1 \times e_2) \rightsquigarrow e_1}$$

PROJ2 The proof for PROJ2 is pretty much identical to that of PROJ1, so we will omit it for brevity.

LIFT All lifted expressions are also values by definition.

$$\frac{}{\mathsf{value}\ [\![e]\!]}$$

USE And likewise, all use expressions are values by definition.

$$\frac{}{\mathsf{value}\ [\![e]\!]_r}$$

BIND For $\cdot \vdash e_1 \gg\!= e_2 : \tau$, we have two premises

$$\cdot \vdash e_1 : \mathsf{IO}_\rho \tau' \qquad\qquad \cdot \vdash e_2 : \tau' \to \mathsf{IO}_\rho \tau$$

Applying the induction hypothesis to the first premise, if e_1 reduces further then we can also say $e_1 \gg\!= e_2$ reduces further

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 \gg\!= e_2 \rightsquigarrow e_1' \gg\!= e_2}$$

If e_1 turns out to be a value instead, then because it has the type $\mathsf{IO}_\rho \tau'$, it must be either of the form $[\![e]\!]$ or $[\![e]\!]_r$. These are the only expressions that can both be values and have the type $\mathsf{IO}_\rho \tau'$. In either case, there exist reduction rules for both of these.

$$\frac{}{[\![e]\!] \gg\!= e_2 \rightsquigarrow e_2\ e} \qquad\qquad \frac{}{[\![e]\!]_r \gg\!= e_2 \rightsquigarrow e_2\ e}$$

CONC We can immediately reduce any expression of the form $v \curlyvee w$.

$$\frac{}{v \curlyvee w \rightsquigarrow v \gg\!= \lambda v.(w \gg\!= \lambda w.[\![v \times w]\!])}$$

SUB If we have a subsumption judgement resulting in $\cdot \vdash e : \mathsf{IO}'_\rho \tau$, then we also have the premise $\cdot \vdash e : \mathsf{IO}_\rho \tau$ for some other ρ. But we can just call the induction hypothesis on this premise to show that either e reduces or e is a value. In any case, it is the same e — so we are done.

□

Theorem 4 (Preservation). *If* $\cdot \vdash e : \tau$ *and* $e \rightsquigarrow e'$*, then* $\cdot \vdash e' : \tau$*.*

Proof. As usual, start with induction on the proof for $\cdot \vdash e : \tau$.

VAR Like with the proof for progress, it is not possible to have a typing judgement for a variable in the empty context.

ABS Whilst it is possible to have a typing judgement of $\cdot \vdash \lambda x.e$, it is not possible to have a reduction of the form $(\lambda x.e) \rightsquigarrow e'$. A lambda abstraction cannot reduce any further, and so we do not need to consider this case.

APP We have $\cdot \vdash e_1\ e_2 : \tau$. Consider the possible cases for $e_1\ e_2 \rightsquigarrow e'$: either e_1 or e_2 must have reduced. When e_1 was reduced

$$\frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1' \rightsquigarrow e_2}$$

We can use the induction hypothesis with this alongside the first premise $\cdot \vdash e_1 : \tau' \to \tau$ to get

$$\cdot \vdash e_1' : \tau' \to \tau$$

And then using App and the second premise, we build up a proof for

$$\cdot \vdash e_1' \ e_2 : \tau$$

If e_2 reduced to e_2' instead, the steps are the same: Use the induction hypothesis to get $\cdot \vdash e_2 : \tau'$, and then $\cdot \vdash e_1 \ e_2' : \tau$. However, there is one other scenario in which an application can reduce, and that is through beta reduction:

$$\frac{\mathsf{value}\ e_2}{(\lambda x.e) \ e_2 \rightsquigarrow e[e_2/x]}$$

We need to show that $\cdot \vdash e[e_2/x] : \tau$. And in this case e_1 must be a lambda abstraction, so the typing rule for the first premise must be an Abs.

$$\textsc{App}\ \frac{\textsc{Abs}\ \dfrac{\cdot, x : \tau' \vdash e : \tau}{\cdot \vdash \lambda x.e : \tau' \to \tau} \qquad \cdot \vdash e_2 : \tau'}{\cdot \vdash (\lambda x.e) \ e_2 : \tau}$$

Furthermore, we know that $\mathrm{dom}(\cdot)$ and the bound type variables in τ' are distinct — the domain of $\cdot$ is empty, and there are no bound type variables in the type scheme τ'. This allows us to use the substitution lemma, Lemma 7, with $\cdot \vdash e_2 : \tau'$ and $\cdot, x : \tau' \vdash e : \tau$, giving us

$$\cdot \vdash e[e_2/x] : \tau$$

Let There is only way a let expression can reduce, and that is by

$$\mathsf{let}\ x = e'\ \mathsf{in}\ e \rightsquigarrow e[e'/x]$$

The premises are

$$\cdot \vdash e' : \tau'$$
$$\cdot, x : \tau' \vdash e : \tau$$

And we know that there are no bound type variables in the type scheme τ', and so they are naturally distinct form $\mathrm{dom}(\cdot)$. Therefore we can just use the substitution lemma again to show that

$$\cdot \vdash e[e'/x] : \tau$$

Unit In this case, the judgement is $\cdot : \square \vdash \square$. But there is no way for a unit $\square$ to reduce, i.e. $\square \not\rightsquigarrow e$, so we do not need to consider this case.

Product An expression of the form $\cdot \vdash e_1 \times e_2 : \tau_1 \times \tau_2$ can be reduced in either two ways. Either $e_1 \rightsquigarrow e_1'$ and $e_1 \times e_2 \rightsquigarrow e_1' \times e_2$, in which case

$$\begin{aligned} \cdot \vdash e_1 : \tau_1 \qquad & \text{by the premises} \\ \cdot \vdash e_1' : \tau_1 \qquad & \text{by induction hypothesis} \\ \cdot \vdash e_1' \times e_2 : \tau_1 \times \tau_2 \qquad & \text{by Product} \end{aligned}$$

Or if $e_2 \rightsquigarrow e_2'$, then we repeat the same steps but acting on the second premise.

PROJ1 $\cdot \vdash \pi_1\ e : \tau$ can reduce in two ways as well. If $e \rightsquigarrow e'$ and $\pi_1\ e \rightsquigarrow \pi_1\ e'$ then

$$\begin{aligned} \cdot &\vdash e : \tau \times \tau' && \text{by the premises} \\ \cdot &\vdash e' : \tau \times \tau' && \text{by induction hypothesis} \\ \cdot &\vdash \pi_1\ e' : \tau && \text{by PROJ1} \end{aligned}$$

However it can also be reduced if e happens to be a product, namely $\pi_1(e_1 \times e_2) \rightsquigarrow e_1$. So we just need to show that $\cdot \vdash e_1 : \tau$, which can be done by climbing up the proof tree.

$$\textsc{Proj1}\ \dfrac{\textsc{Product}\ \dfrac{\cdot \vdash e_1 : \tau \qquad \cdot \vdash e_2 : \tau}{\cdot \vdash e_1 \times e_2 : \tau \times \tau'}}{\cdot \vdash \pi_1\ (e_1 \times e_2) : \tau}$$

PROJ2 Again, the proof for PROJ2 is pretty much identical to that of PROJ1 and so is omitted for brevity.

LIFT For $\cdot \vdash [\![e]\!] : \mathsf{IO}_\rho \tau$, just like $\square$ there is no way for $[\![e]\!]$ to reduce any further — they are both values. There is nothing needed to be done for this case.

USE The same also applies for $[\![e]\!]_r$.

BIND The proof we are handling looks like this

$$\textsc{Bind}\ \frac{\cdot \vdash e_1 : \mathsf{IO}_\rho \tau' \qquad \cdot \vdash e_2 : \tau' \to \mathsf{IO}_\rho \tau}{\cdot \vdash e_1 \gg\!= e_2 : \mathsf{IO}_\rho \tau}$$

The first thing to note is that there are multiple possible ways in which reduction could have occurred: We will look at each of them one by one.

$e_1 \gg\!= e_2 \rightsquigarrow e_1' \gg\!= e_2$ The premise for this reduction is $e_1 \rightsquigarrow e_1'$, so we take the following steps

$$\begin{aligned} \cdot &\vdash e_1 : \mathsf{IO}_\rho \tau' && \text{by the premises} \\ \cdot &\vdash e_1' : \mathsf{IO}_\rho \tau' && \text{by induction hypothesis} \\ \cdot &\vdash e_1' \gg\!= e_2 : \mathsf{IO}_\rho \tau && \text{by BIND} \end{aligned}$$

$[\![e]\!] \gg\!= e_2 \rightsquigarrow e_2\ e$ We know $\cdot \vdash [\![e]\!] : \mathsf{IO}_\rho \tau'$, but that does not necessarily mean that the proof for it came from the LIFT rule — it could have also been derived from subsumption, SUB. To help when this scenario arises, we use a small helper lemma listed in the appendix, Lemma 10, that says if $\Gamma \vdash [\![e]\!] : \mathsf{IO}_\rho \tau'$, then $\Gamma \vdash e : \tau'$. Now we can use ABS to construct $\cdot \vdash e_2\ e : \mathsf{IO}_\rho \tau$ as needed.

$[\![e]\!]_r \gg\!= e_2 \rightsquigarrow e_2\ e$ This is identical to the case above, except we use Lemma 11 instead.

Conc We have the proof and premises

$$\textsc{Conc}\ \frac{\centerdot \vdash v : \mathsf{IO}_{\rho_1}\tau_1 \qquad \centerdot \vdash w : \mathsf{IO}_{\rho_2}\tau_2 \qquad \mathsf{ok}\ \rho_1 \cup \rho_2}{\centerdot \vdash v \curlyvee w : \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1 \times \tau_2}$$

There is also only one possible reduction that could have occurred

$$v \curlyvee w \rightsquigarrow v \gg\!= \lambda v.(w \gg\!= \lambda w.[\![v \times w]\!])$$

So we need to show $\centerdot \vdash v \gg\!= \lambda v.(w \gg\!= \lambda w.[\![v \times w]\!]) : \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1 \times \tau_2$ — not exactly the prettiest judgement — which we can construct eventually from a Bind.

To begin, we will first show that both v and w can be subsumed into the larger combined heap, $\rho_1 \cup \rho_2$.

$\centerdot \vdash v : \mathsf{IO}_{\rho_1}\tau_1$	from premises
$\rho_1 \geq: \rho_1 \cup \rho_2$	by UnionL
$\mathsf{ok}\ \rho_1 \cup \rho_2$	from premises
$\centerdot \vdash v : \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1$	by Sub

$\centerdot \vdash w : \mathsf{IO}_{\rho_2}\tau_2$	from premises
$\centerdot, v : \tau_1 \vdash w : \mathsf{IO}_{\rho_2}\tau_2$	by Lemma 3
$\rho_2 \geq: \rho_1 \cup \rho_2$	by UnionR
$\mathsf{ok}\ \rho_1 \cup \rho_2$	from premises
$\centerdot, v : \tau_1 \vdash w : \mathsf{IO}_{\rho_1\cup\rho_2}\tau_2$	by Sub

Then we show that we can make a lambda, taking a *pure* τ_2, and making a $\tau_1 \times \tau_2$ inside that new IO monad.

$$\dfrac{\dfrac{\dfrac{\centerdot, v : \tau_1, w : \tau_2 \vdash v : \tau_1 \qquad \centerdot, v : \tau_1, w : \tau_2 \vdash w : \tau_2}{\centerdot, v : \tau_1, w : \tau_2 \vdash v \times w : \tau_1 \times \tau_2}}{\centerdot, v : \tau_1, w : \tau_2 \vdash [\![v \times w]\!] : \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1 \times \tau_2}}{\centerdot, v : \tau_1 \vdash \lambda w.[\![v \times w]\!] : \tau_2 \to \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1 \times \tau_2}$$

We then take this lambda and bind our *monadic* w to it

$$\dfrac{\centerdot, v : \tau_1 \vdash w : \mathsf{IO}_{\rho_1\cup\rho_2}\tau_2 \qquad \centerdot, v : \tau_1 \vdash \lambda w.[\![v \times w]\!] : \tau_2 \to \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1 \times \tau_2}{\centerdot, v : \tau_1 \vdash w \gg\!= (\lambda w.[\![v \times w]\!]) : \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1 \times \tau_2}$$

Now we can make another lambda, this time taking in the *pure* τ_1

$$\dfrac{\centerdot, v : \tau_1 \vdash w \gg\!= (\lambda w.[\![v \times w]\!]) : \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1 \times \tau_2}{\centerdot \vdash \lambda v.w \gg\!= (\lambda w.[\![v \times w]\!]) : \tau_1 \to \mathsf{IO}_{\rho_1\cup\rho_2}\tau_1 \times \tau_2}$$

And then we finally bind it with our *monadic* v

$$\frac{\centerdot \vdash v : \mathsf{IO}_{\rho_1 \cup \rho_2} \tau_1 \qquad \centerdot \vdash \lambda v.w \gg\!= \lambda w.[\![v \times w]\!] : \tau_1 \to \mathsf{IO}_{\rho_1 \cup \rho_2} \tau_1 \times \tau_2}{\centerdot \vdash v \gg\!= (\lambda v.w \gg\!= (\lambda w.[\![v \times w]\!])) : \mathsf{IO}_{\rho_1 \cup \rho_2} \tau_1 \times \tau_2}$$

The proof is quite long, but since our typing system is syntax directed, it closely matches the expression it reduces to, making it easier to construct.

Sub Last but not least, we have a subsumption typing judgement

$$\textsc{Sub}\ \frac{\centerdot \vdash e : \mathsf{IO}_{\rho'} \tau \qquad \rho \geq: \rho' \qquad \mathsf{ok}\ \rho}{\centerdot \vdash e : \mathsf{IO}_{\rho} \tau}$$

However because we know nothing of the syntax of e, e could reduce to anything and we are left with some e', such that $e \rightsquigarrow e'$. Nevertheless, applying the induction hypothesis with $\centerdot \vdash e : \mathsf{IO}_{\rho'} \tau$ and $e \rightsquigarrow e'$ gives us

$$\centerdot \vdash e' : \mathsf{IO}_{\rho'} \tau$$

Which we can stick back into Sub to get

$$\centerdot \vdash e' : \mathsf{IO}_{\rho} \tau$$

□

Together these two properties can ensure that no expression can go wrong, which within operational semantics means it cannot get *stuck*. A stuck expression is an expression which cannot be reduced any further (in its normal form) and is not a value. The progress theorem has shown that evaluation of any well-typed expression cannot end up in this state, so it does indeed evaluate to something in the end. And the preservation theorem guarantees us that the type of the final value will remain the same. Therefore we have shown the operational equivalent of the denotational

$$\Gamma \vdash e : \tau \to \Gamma \Vdash e : \tau$$

Chapter 5

Mechanisation in Agda

In this chapter, we will look at how the proofs in Chapter 4 were mechanically formalised — in other words, proved within a proof assistant. The proofs were mechanised within Agda [12], a dependently typed programming language with an ML syntax, similar to that of Haskell's. It can be used for general purpose programming, but since it is rooted in Martin-Löf intuitionistic type theory [10], it can also be used as a prove formal properties.

Properties are proven in Agda by writing programs that satisfy types. This approach takes advantage of the Curry-Howard correspondence, which is that propositions are analogous to types, and proofs are analogous to programs that fulfil that type. We first create types that represent our propositions. Then we can then prove our propositions by constructing a value for the program that satisfies the type — hence why intuitionistic logic is also known as constructive logic.

As a brief example, a proposition of the form $\forall a.a \to b$, translates into a program of type `∀ a → b`. If you read the proposition as "*if a, then b*", then you can read the type as "*given any proof of a, then I can construct a proof of b*". And that is indeed what the program should do: It is a function that takes an argument of a, and returns something of type b.

The vast majority of the work in mechanising was in the proofs related to polymorphism in the Hindley-Damas-Milner system, not the resourceful parts. The syntax-directed Hindley-Damas-Milner system has been successfully formalised in Coq [6], and a formalisation for System F was recently created in Agda [2]. However as it stands, we are not aware of any formalizations of Hindley-Damas-Milner within Agda. Regardless, the techniques and approaches here are similar, and we base much of the work on the proofs in [20].

5.1 Definitions

The framework within Agda for working with the type system is built upon the formalisation of the simply typed lambda calculus in Agda by Kokke et al. [8]

```
data Term : Set where
  `_         : Id → Term
  ƛ_⇒_       : Id → Term → Term
  _·_        : Term → Term → Term
  lt_⇐_in'_  : Id → Term → Term → Term
  ⟦_⟧         : Term → Term
  _>>=_      : Term → Term → Term
  □          : Term
  use        : Resource → Term → Term
  _×_        : Term → Term → Term
  π₁         : Term → Term
  π₂         : Term → Term
  _⋎_        : Term → Term → Term
```

```
data Heap : Set where
  World : Heap
  `_    : Resource → Heap
  _∪_   : Heap → Heap → Heap

data Type : Set where
  `_  : Id → Type
  _⇒_ : Type → Type → Type
  IO  : Heap → Type → Type
  □   : Type
  _×_ : Type → Type → Type

data TypeScheme : Set where
  V_·_ : Id → TypeScheme → TypeScheme
  `_   : Type → TypeScheme
```

Listing 1: Grammar definitions. Note that some of the notation (e.g. ∀) had to be substituted due to overlaps with keywords in Agda.

It begins with the grammar, shown in Listing 1.

For relations, such as the heap well-formed relation ok ρ, we use a pattern of defining a new data type. The type then acts as a proposition that the relation holds, and creating an instance of the data type constitutes as providing proof. Furthermore, the constructors for the data type directly correspond to the rules for the relation.

```
data Ok : Heap → Set where
  OkZ : ∀ {r}
        --------
      → Ok (` r)
  OkS : ∀ {a b}
      → Ok a
      → Ok b
      → a ∩ b =∅
        ----------
      → Ok (a ∪ b)
  OkWorld : --------
            Ok World
```

We also define a data type to represent proof that a heap is a subheap of another heap. Since subheaping is a binary relation over two heaps, the data type is also parameterised over two heaps, which appear in the type.

```
data _≥:_ : Heap → Heap → Set where
  ≥:World : ∀ {ρ} → World ≥: ρ
  ≥:refl : ∀ {ρ} → ρ ≥: ρ
  ≥:Uˡ : ∀ {ρ ρ' ρ''}
       → ρ' ≥: ρ
         ------------
       → ρ' ∪ ρ'' ≥: ρ
  ≥:Uʳ : ∀ {ρ ρ' ρ''}
       → ρ' ≥: ρ
         ------------
       → ρ'' ∪ ρ' ≥: ρ
```

If we wanted to show that $\mathsf{Net} \cup \mathsf{File} \geq: \mathsf{Net}$, we would follow the same steps that we would carry out in a proof tree, in order to construct a value that inhabits the type `` ` Net ∪ ` File ≥: ` Net ``.

```
_ : ` Net ∪ ` File ≥: ` Net
_ = ≥:Uˡ ≥:refl
```

$$\textsc{UnionL}\ \dfrac{\textsc{Refl}\ \dfrac{}{\mathsf{Net} \geq: \mathsf{Net}}}{\mathsf{Net} \cup \mathsf{File} \geq: \mathsf{Net}}$$

The most important relation however, is the typing relation. With Agda's Unicode support we are able to define the rules, shown in Listing 2, with a notation similar to what we used in Chapter 3. Typing judgements can be can constructed as follows:

```
_ : ∅ ⊢ ƛ "x" ⇒ ` "x" ₈ (` "α" ⇒ ` "α")
_ = ⊢ƛ (⊢` Z (Inst SZ refl refl))
```

$$\textsc{Abs}\ \dfrac{\textsc{Var}\ \dfrac{x : \alpha \in \centerdot, x : \alpha \qquad \alpha > \alpha}{\centerdot, x : \alpha \vdash x : \alpha}}{\centerdot \vdash \lambda x.x : \alpha \to \alpha}$$

Where `Inst SZ refl refl` is proof that the type scheme α can be instantiated to the type α (by not instantiating anything type variables at all).

```
data _>_ : TypeScheme → Type → Set where
  Inst : ∀ {σ τ}
    → (s : Substitution)
    → subDomain s ≡ TSvars σ
    → subT s (TStype σ) ≡ τ
    --------------------
    → σ > τ
```

The small-step relation is defined in a similar way to the typing relation.

```
data _⇝_ : Term → Term → Set where
  ξ-·₁ : ∀ {e₁ e₂ e₁'}
       → e₁ ⇝ e₁'
         ------------------
       → e₁ · e₂ ⇝ e₁' · e₂

  ξ-·₂ : ∀ {e₁ e₂ e₂'}
       → e₂ ⇝ e₂'
         ------------------
       → e₁ · e₂ ⇝ e₁ · e₂'

  β-ƛ : ∀ {x e e'}
      → Value e'
        ------------------
      → (ƛ x ⇒ e) · e' ⇝ e [ x := e' ]
  -- and so on ...
```

5.2 Type schemes and type variables

One of the main design decisions made early on was how to represent type schemes. Quantified type variables in type schemes are often represented as a sequence $\forall \alpha_1, \ldots, \alpha_n \cdot \tau$. This expands out to $\forall \alpha_1 \cdot \forall \ldots \cdot \forall \alpha_n \cdot \tau$ in the end, and is how type schemes are ultimately defined in the grammar. But we need to be able to reason about type variables in the sequence format for some of the proofs. A helper function `VV` was created for this reason. It allows for type schemes to be created and manipulated in terms of lists.

```
VV : List Id → Type → TypeScheme
VV (α ∷ αs) τ = V α · (VV αs τ)
VV [] τ = ` τ
```

Now we can rewrite propositions, such as the substitution lemma (Lemma 7), in a way that lets us bind the list of quantified type variables and use it elsewhere.

```
subst : ∀ {Γ x e e' αs τ τ'}
      → Γ ⊢ e ꞉ τ
      → Γ , x ꞉ VV αs τ ⊢ e' ꞉ τ'
      → Disjoint αs (FTVC Γ)
        ----------------------
      → Γ ⊢ e' [ x := e ] ꞉ τ'
```

There are also functions to extract the quantified type variables and type from a type scheme, and equivalence proofs that can be used to convince the type checker that they are equivalent to their `VV` form.

```
data _⊢_∋_ : Context → Term → Type →
↪  Set where

  ⊢` : ∀ {Γ x σ τ}
    → x ∋ σ ∈ Γ
    → σ > τ
      ---------
    → Γ ⊢ ` x ∋ τ

  ⊢ƛ : ∀ {Γ x τ' τ e}
    → Γ , x ∋ ` τ' ⊢ e ∋ τ
      --------------------
    → Γ ⊢ ƛ x ⇒ e ∋ (τ' ⇒ τ)

  ⊢· : ∀ {Γ e e' τ τ'}
    → Γ ⊢ e ∋ τ' ⇒ τ
    → Γ ⊢ e' ∋ τ'
      ---------------
    → Γ ⊢ e · e' ∋ τ

  ⊢lt : ∀ {Γ e e' τ τ' x}
      → Γ ⊢ e' ∋ τ'
      → Γ , x ∋ close Γ τ' ⊢ e ∋ τ
        ------------------------
      → Γ ⊢ lt x ⇐ e' in' e ∋ τ

  ⊢× : ∀ {Γ e e' τ τ'}
    → Γ ⊢ e ∋ τ
    → Γ ⊢ e' ∋ τ'
      ----------------------
    → Γ ⊢ e × e' ∋ τ × τ'

  ⊢π₁ : ∀ {Γ e τ τ'}
      → Γ ⊢ e ∋ τ × τ'
        ---------------
      → Γ ⊢ π₁ e ∋ τ

  ⊢π₂ : ∀ {Γ e τ τ'}
      → Γ ⊢ e ∋ τ × τ'
        ---------------
      → Γ ⊢ π₂ e ∋ τ'

  ⊢□ : ∀ {Γ}

       ----------
    → Γ ⊢ □ ∋ □

  -- Monadic rules

  ⊢[[]] : ∀ {Γ e τ ρ}
      → Γ ⊢ e ∋ τ
      → Ok ρ
        ------------------
      → Γ ⊢ [[ e ]] ∋ IO ρ τ

  ⊢use : ∀ {Γ e τ r}
       → Γ ⊢ e ∋ τ
         ----------------
       → Γ ⊢ use r e ∋ IO (` r) τ

  ⊢>>= : ∀ {Γ e e' τ τ' ρ}
       → Γ ⊢ e ∋ (IO ρ τ')
       → Γ ⊢ e' ∋ (τ' ⇒ IO ρ τ)
         --------------------
       → Γ ⊢ e >>= e' ∋ IO ρ τ

  ⊢ϒ : ∀ {Γ e₁ e₂ τ₁ τ₂ ρ₁ ρ₂}
    → Γ ⊢ e₁ ∋ IO ρ₁ τ₁
    → Γ ⊢ e₂ ∋ IO ρ₂ τ₂
    → Ok (ρ₁ ∪ ρ₂)
      ------------------------
    → Γ ⊢ e₁ ϒ e₂ ∋ IO (ρ₁ ∪ ρ₂) (τ₁
    ↪  × τ₂)

  ⊢sub : ∀ {Γ e τ ρ ρ'}
       → Γ ⊢ e ∋ IO ρ' τ
       → ρ ≥: ρ'
       → Ok ρ
         ---------------
       → Γ ⊢ e ∋ IO ρ τ
```

Listing 2: The typing rules as they are defined in Agda.

```
TStype : TypeScheme → Type
TStype (V _ · σ) = TStype σ
TStype (` τ) = τ

TSvars : TypeScheme → List Id
TSvars (V α · σ) = α :: TSvars σ
TSvars (` τ) = []

TStype≡ : ∀ {αs τ} → TStype (VV αs τ) ≡ τ
TStype≡ {[]} {τ} = refl
TStype≡ {α :: αs} {τ} = TStype≡ {αs}

TSvars≡ : ∀ {αs τ} → TSvars (VV αs τ) ≡ αs
TSvars≡ {[]} = refl
TSvars≡ {α :: αs} = cong (_::_ α) TSvars≡
```

5.3 Properties

The type preservation theorem (Theorem 4) is proven with the following function — given proof that e has type τ in the empty context, and proof that e reduces to e' in one step, then we can provide proof that e' also has the type τ in the empty context.

```
preservation : ∀ {e e' τ}
             → ∅ ⊢ e ⦂ τ
             → e ⇝ e'
               ----------
             → ∅ ⊢ e' ⦂ τ
```

For progress (Theorem 3), we need to be able to say either the expression is a value or it reduces to something else. We create a new data type to represent the two cases where this can happen.

```
data Progress (e : Term) : Set where
  step : ∀ {e'}
       → e ⇝ e'
         ------
       → Progress e
  done : Value e
         ----------
       → Progress e

progress : ∀ {e τ}
         → ∅ ⊢ e ⦂ τ
           ----------
         → Progress e
```

5.4 Postulations

There are number of properties that are postulated throughout, and these should eventually be proven at some point. Postulations do not require any proofs and are just assumed by Agda to be true, so they are used sparingly here. They are mainly used in cases where we know something to be true that is usually needed to prove another property, but the proof for them is large and non-trivial.

One such example is when dealing with the Barendregt variable and convention. On paper it is fine to implicitly assume all bound variables are unique, but Agda is not so lenient. We need to explicitly perform the alpha conversion and show that the new variable is distinct from any free variable and that the two expressions are equivalent

```
α-conv : ∀ {x e}
       → ∃[ y ] ((∀ {e'} → y ∉ FV e') → (ƛ x ⇒ e ≡ ƛ y ⇒ (e [ x := ` y ])))
```

Working with existential quantification and these non-trivial equivalences can get pretty unwieldly, so in some proofs we simply just make the postulation

```
postulate x∉FVv : x ∉ FV v
```

Chapter 6

Evaluation and further work

6.1 Separation logic

In Section 2.1 we briefly touched separation logic, but now that we have defined our system we can look at some parallels between them. Take the frame rule again, which allows additional predicates to be inferred in a specification, under the condition that the code does not use any of the variables in the predicate.

$$\frac{\{p\}\ \mathit{code}\ \{q\}}{\{p * r\}\ \mathit{code}\ \{q * r\}}\ \begin{array}{l}\text{where no variable occurring free}\\ \text{in r is modified by } \mathit{code}\end{array}$$

Our well-formed heap judgement has a rule that allows heaps to be merged, under the condition that they do not overlap:

$$\frac{\mathsf{ok}\ \rho \qquad \mathsf{ok}\ \rho' \qquad \rho \cap \rho' = \emptyset}{\mathsf{ok}\ \rho \cup \rho'}$$

This mirrors the idea in separation logic that we only need to worry about the variables relevant to our code, or in our terminology, the resources relevant to our type. So our subsumption rule, Sub, that lets heaps get promoted to a larger heap, ends up being our version of the frame rule. And the concurrency rule from separation logic

$$\frac{\{p_1\}\ \mathit{code}_1\ \{q_1\} \qquad \{p_2\}\ \mathit{code}_2\ \{q_2\}}{\{p_1 * p_2\}\ \mathit{code}_1\ ||\ \mathit{code}_2\ \{q_1 * q_2\}}$$

Ends up becoming our concurrency typing rule, conc. The preconditions and postconditions are the heaps in the IO type, and the separating conjunction $*$ is our union $\cup$.

$$\frac{\Gamma \vdash e_1 : \mathsf{IO}_{\rho_1}\tau_1 \qquad \Gamma \vdash e_2 : \mathsf{IO}_{\rho_2}\tau_2 \qquad \mathsf{ok}\ \rho_1 \cup \rho_2}{\Gamma \vdash e_1 \curlyvee e_2 : \mathsf{IO}_{\rho_1 \cup \rho_2}\ \tau_1 \times \tau_2}$$

6.2 Well formed heaps

We have proved the system defined in Chapter 3 is sound, and moreover we can show that every concurrent expression has a well formed heap — that is a resource is not used more than once in the heap, and thus not accessed concurrently.

Theorem 5 (Concurrent heap is well formed)**.** *If* $\Gamma \vdash e_1 \curlyvee e_2 : \mathsf{IO}_\rho \tau$*, then* $\mathsf{ok}\ \rho$

Proof. By induction on the possible proofs for $\Gamma \vdash e_1 \curlyvee e_2 : \mathsf{IO}_\rho \tau$.

Conc We need to show $\mathsf{ok}\ \rho_1 \cup \rho_2$, and from the premises we have $\mathsf{ok}\ \rho_1 \cup \rho_2$.

Sub We need to show $\mathsf{ok}\ \rho'$ and from the premises, we also have $\mathsf{ok}\ \rho'$.

□

As you can see, this just follows from the definition of the concurrency and subsumption rules. What we would really like to prove is that *all* monadic expressions have well formed heaps.

Conjecture 1. *If* $\blacksquare \vdash e : \mathsf{IO}_\rho \tau$*, then* $\mathsf{ok}\ \rho$

This is a stronger version of Theorem 5. However this is not made easy in the system as it stands, for one simple yet annoying reason: Types are not necessarily well formed, and Abs allows any type to be introduced. This can be easily shown with the identity function.

$$\dfrac{\dfrac{x : \mathsf{IO}_{\mathsf{File}\cup\mathsf{File}}\tau \in \blacksquare, x : \mathsf{IO}_{\mathsf{File}\cup\mathsf{File}}\tau \qquad \mathsf{IO}_{\mathsf{File}\cup\mathsf{File}}\tau > \mathsf{IO}_{\mathsf{File}\cup\mathsf{File}}\tau}{\blacksquare, x : \mathsf{IO}_{\mathsf{File}\cup\mathsf{File}}\tau \vdash x : \mathsf{IO}_{\mathsf{File}\cup\mathsf{File}}\tau}\ \textsc{Var}}{\blacksquare \vdash \lambda x.x : \mathsf{IO}_{\mathsf{File}\cup\mathsf{File}}\tau \to \mathsf{IO}_{\mathsf{File}\cup\mathsf{File}}\tau}\ \textsc{Abs}$$

Abs acts as a mechanism to generate types with malformed heaps, such as an IO monad that contains two Files in its heap. In reality this is not an issue, as there is no way to construct a value to pass into this lambda. Because of this I believe that Conjecture 1 is still provable, but only when the context only contains well formed type schemes, and hence why it only applies for the empty context.

We can extend this to include a relation for well formed contexts and well formed types, and use this in our definition.

$$\boxed{\mathsf{ok}\ \Gamma}$$

$$\frac{}{\mathsf{ok}\ \cdot} \qquad \frac{\mathsf{ok}\ \Gamma \qquad \mathsf{ok}\ \sigma}{\mathsf{ok}\ \Gamma, x : \sigma}$$

$$\boxed{\mathsf{ok}\ \sigma}$$

$$\frac{\mathsf{ok}\ \tau}{\mathsf{ok}\ \forall \alpha_1, \ldots, \alpha_n \cdot \tau}$$

$$\boxed{\mathsf{ok}\ \tau}$$

$$\frac{}{\mathsf{ok}\ \square} \qquad \frac{}{\mathsf{ok}\ \alpha} \qquad \frac{\mathsf{ok}\ \tau \qquad \mathsf{ok}\ \tau'}{\mathsf{ok}\ \tau \to \tau'} \qquad \frac{\mathsf{ok}\ \tau \qquad \mathsf{ok}\ \tau'}{\mathsf{ok}\ \tau \times \tau'} \qquad \frac{\mathsf{ok}\ \tau \qquad \mathsf{ok}\ \rho}{\mathsf{ok}\ \mathsf{IO}_\rho \tau}$$

Conjecture 2. *If* $\mathsf{ok}\ \Gamma$ *and* $\Gamma \vdash e : \mathsf{IO}_\rho \tau$*, then* $\mathsf{ok}\ \rho$

So far two lemmas have been proven to aid in the proof of the above conjecture.

Lemma 8. *If* $\mathsf{ok}\ \tau$ *then for any substitution* s*,* $\mathsf{ok}\ \tau[s]$

Lemma 9. *If* $\mathsf{ok}\ \sigma$ *and* $\sigma > \tau$*, then* $\mathsf{ok}\ \tau$

This idea of well formed types adds quite a bit of extra overhead to the system, so it is left as further work.

6.3 Let polymorphism

All this theory was motivated by a very practical cause: to provide some simple concurrency guarantees at the type level in a programming language. Thus the intention is that some of this may eventually make its way into a type system of an existing or novel language.

So instead of starting with the simply typed lambda calculus, the system was based off of the Hindley-Damas-Milner system with its polymorphic type discipline. It cannot be understated the amount of extra complexity and work this introduced into the proofs for soundness, but polymorphism is an essential feature in today's modern functional programming languages. The let polymorphism does not interact much with the monadic and resourceful parts of the type system, but it is desirable to show that the two are compatible with each other regardless.

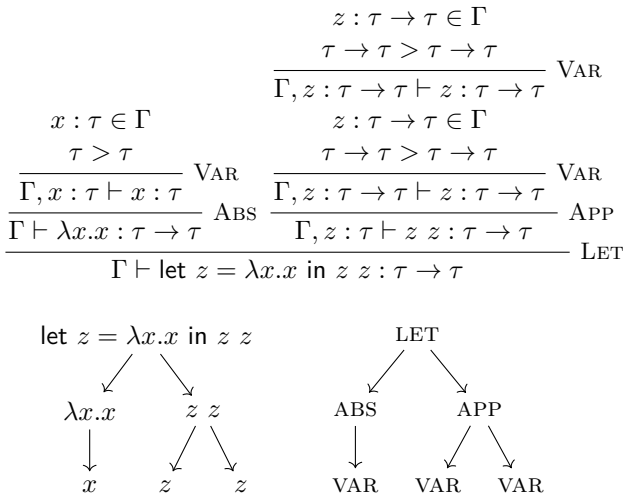

Figure 6.1: How syntax directed typing rules mean that the proof tree is isomorphic to the syntax tree.

6.4 Almost syntax directed

As mentioned earlier, the system was based off a syntax directed treatment of Hindley-Damas-Milner. That meant that there were only four rules instead of the usual six, where instantiation and generalisation are rolled into VAR and LET respectively. The main benefit of this was that it meant there was exactly one typing judgement for each form of syntax, so constructing proofs for a program was a breeze — you just follow the corresponding rules for each term. For example, in the expression $\mathsf{let}\ z = \lambda x.x\ \mathsf{in}\ z\ z$, a let consisting of an abstraction and an application, the proof begins with a LET, then the premises ABS and APP as shown in figure 6.1.

In our system however, the subsumption rule SUB prevents this. For every typing judgement of the form $\Gamma \vdash e : \mathsf{IO}_\rho \tau$, there are two possible proofs that can be constructed for it: the proof corresponding to the syntax of e, and SUB. Furthermore, because the subheap relation is reflexive, its possible to have an infinite proof tree with repeated applications of SUB as shown in figure 6.2. When working with proofs this did not turn out to be too big of an issue, but it is undoubtedly a little bit unsatisfying to lose such a nice property of the system.

6.5 Modelling state within the monad

The modelling of the IO monad is extremely simplified, and does nothing more than sequence computation. To illustrate this point, the reduction rule for

$$\textsc{Sub}\ \frac{\textsc{Sub}\ \dfrac{\textsc{Sub}\ \dfrac{\vdots\quad \mathsf{File} \geq: \mathsf{File} \quad \mathsf{ok}\ \mathsf{File}}{\Gamma \vdash [\![\Box]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\Box} \qquad \mathsf{File} \geq: \mathsf{File} \quad \mathsf{ok}\ \mathsf{File}}{\Gamma \vdash [\![\Box]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\Box} \qquad \mathsf{File} \geq: \mathsf{File} \quad \mathsf{ok}\ \mathsf{File}}{\Gamma \vdash [\![\Box]\!]_{\mathsf{File}} : \mathsf{IO}_{\mathsf{File}}\Box}$$

$$[\![\Box]\!]_{\mathsf{File}} \downarrow \Box \qquad \curvearrowright \textsc{sub}$$

Figure 6.2: How an infinite chain of SUB can be produced, leading nowhere.

concurrency is completely sequential.

$$\frac{}{v \curlyvee w \rightsquigarrow v \gg\!= \lambda v.(w \gg\!= \lambda w.[\![v \times w]\!])}$$

In the real world, this would be pointless. We want to be able to compute these expressions concurrently, so rewriting everything in terms of $\gg\!=$ won't cut it. By modelling the state of the world, like what actually happens in the `IO` monad of Haskell, we can perform more advanced reasoning about things such as concurrency. One such possible method would be to create a new rule for reduction of monadic computations, that passes about state. Then the concurrency rule could be written something like this:

$$\boxed{\langle e, s\rangle \rightsquigarrow \langle e', s'\rangle} \qquad \frac{\langle e_1, s\rangle \rightsquigarrow \langle e_1', s_1'\rangle \qquad \langle e_2, s\rangle \rightsquigarrow \langle e_2', s_2'\rangle}{\langle e_1 \curlyvee e_2, s\rangle \rightsquigarrow \langle e_1' \times e_2', s_1' \cup s_2'\rangle}$$

We could then reason further about how the two states $s_1' \cup s_2'$ are merged — something that separation logic could be used for.

6.6 Extending the inference algorithm

One of the main contributions of the Hindley-Damas-Milner system was the inclusion of a type inference algorithm, Algorithm W [11]. A type system on its own only lets us ascertain that a program is well-typed, given that we supply it an expression and a type. But the design of Hindley-Damas-milner made it possible for the type to be inferred algorithmically, from the program alone. Algorithm W also always infers the *principal type*: the most general, polymorphic type an expression could have. Algorithm W freed ML programmers from having to explicitly write type annotations, and is one of its most iconic features. Naturally, we should explore extending Algorithm W to handle our new resourceful types.

6.7 Heap polymorphism

It would be convenient if we were able to have expressions like the following well-typed:

$$\Gamma = \centerdot, f : \square \to \mathsf{IO}_{\mathsf{File}}\square, g : \square \to \mathsf{IO}_{\mathsf{Net}}\square$$
$$\Gamma \vdash \mathsf{let}\ x = [\![\square]\!]\ \mathsf{in}\ (x \gg\!= f) \curlyvee (x \gg\!= g) : \mathsf{IO}_{\mathsf{File}\cup\mathsf{Net}}\square \times \square$$

This is not possible in the current type system: x needs to be both $\mathsf{IO}_{\mathsf{File}}$ and $\mathsf{IO}_{\mathsf{Net}}$ simultaneously! This is very much the same question that Hindley, Damas and Milner set out to solve, except instead of polymorphic *types* we want to have polymorphic *heaps*. The basic idea for this would involve extending the definition of a heap to allow for variable lookup, **h**:

$$\text{heap } \rho \qquad ::= \ r \mid \sigma \cup \sigma' \mid \mathsf{World} \mid \mathbf{h}$$

And then redefining substitutions to be pairs of a map from type variables to types, and a map from heap variables to heaps.

$$S : (\alpha \mapsto \tau \times h \mapsto \rho)$$

Then in the definition of substitution, whenever a mapping from a heap variable to heap is encountered we would apply the substitution.

$$\begin{aligned}
(\mathsf{IO}_{\rho'}\tau)[h/\rho] &= \mathsf{IO}_{\rho'[h/\rho]}\tau \\
r[h/\rho] &= r \\
(\rho_1 \cup \rho_2)[h/\rho] &= \rho_1[h/\rho] \cup \rho_2[h/\rho] \\
\mathsf{World}[h/\rho] &= \mathsf{World} \\
h'[h/\rho] &= \begin{cases} \rho & \mathsf{if} h' = h \\ h' & \mathsf{otherwise} \end{cases}
\end{aligned}$$

6.8 Dependently typed resources

So far our resources have just been simple static placeholders that represented system resources, for example the filesystem and the network. This is just for the sake of argument: resources can be as general or as specific as is needed for the programmer's use case.

Suppose a programmer wanted to perform multiple concurrent operations on various files. A resource representing the entire filesystem would be too coarse-grained for this purpose. Instead they might want to have a separate resource for each file.

$$\text{resource } r ::= \mathsf{foo.txt} \mid \mathsf{bar.txt} \mid \mathsf{baz.txt} \ldots$$

But this is very static. If the programmer writes some code that operates on a new file, then the new file would need to be appended to the list of these

predefined resources. And what if they don't know what files they will be touching at typecheck time? Instead, it would be much more natural if the programmer could write programs like

$$\mathsf{readFile}\ \text{“foo.txt”} : \mathsf{IO}_{\mathsf{foo.txt}}\mathsf{String}$$
$$\mathsf{writeFile}\ \text{“bar.txt”} : \mathsf{String} \to \mathsf{IO}_{\mathsf{bar.txt}}\square$$

and have the file automatically *lifted* from the expression and into the heap of the type. This is a job for **dependent types**: having the type depend on the term. However in our system, our resources and types are not unified. In a dependently typed lambda calculus, the Π rule substitutes values into types like so[1]

$$\frac{\Gamma \vdash \Pi(x : \tau').\tau : \star \qquad \Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e : \Pi(x : \tau').\tau}$$

$$\frac{\Gamma \vdash e : \Pi(x : \tau').\tau \qquad e' : \tau'}{\Gamma \vdash e\ e' : \tau[e'/\tau']}$$

We would need to combine the syntaxes for resources and types together for this rule, or flesh out types to include type constructors through which resources and heaps could be embedded in. Alternatively, we could be lazy and avoid this by making it dependent only in the heap. We could introduce a type of lambda for IO monads that tags it with a placeholder resource, which gets substituted when an argument is applied.

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash (\lambda_{\mathsf{IO}} x.e) : \tau' \to_{\mathsf{IO}} \tau}$$

$$\frac{\Gamma \vdash e : \tau' \to_{\mathsf{IO}} \tau \qquad \mathsf{Value}\ e' \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \mathsf{IO}_{e'}\tau}$$

It would be wise to restrict this to values, so we aren't going about tagging resources with half-evaluated computations. We would also need a notion of equality for values so that we can safely tell what values are disjoint within a heap. This will pose some interesting challenges in practice, like knowing statically what variable expressions might represent the same resource. Does the typechecker here know that $x = y$?

$$x \leftarrow \mathsf{readLine}$$
$$\mathsf{readFile}\ x$$
$$\mathsf{let}\ y = x\ \mathsf{in}\ \mathsf{writeFile}\ y\ \text{“hello”}$$

[1] $\Gamma \vdash \tau : \star$ is a judgement saying that τ is a type in some universe.

6.9 Casting heaps

We have only been passing about the primitive unit type, but realistically we are going to want to have functions that actually have functionality, like the aforementioned

$$\mathsf{readFile} : \mathsf{String} \to \mathsf{IO}_{\rho}\mathsf{String}$$

A function like this is not going to be built-in, instead it will probably be part of a standard library. So how will it be implemented? The easiest solution would be to wrap around an existing function that reads files in a plain IO monad. Remember that plain IO is equivalent to $\mathsf{IO}_{\mathsf{World}}$:

$$\mathsf{readFile'} : \mathsf{String} \to \mathsf{IO}_{\mathsf{World}}\mathsf{String}$$

How do we bring the IO monad into a File heap? If it were a pure computation, then we could have just used the use operator:

$$\lambda s \to [\![\mathsf{readFile'}\ s]\!]_{\mathsf{File}}$$

But it's a monadic computation, so we would end up with a doubly nested monad like $\mathsf{IO}_{\mathsf{File}}(\mathsf{IO}_{\mathsf{World}}\mathsf{String})$. Can we subsume it? No! Because World is at the top of the sub-heap chain. It already uses all possible resources, and we can't claim that it uses less than what it actually does! The type is overly cautious in this case.

In an ideal world, standard library functions would built up from a handful of built-ins for extremely low level primitives, that keep track of things like file descriptors. Then when they are put together, we could subsume them to keep track of higher-level resources. For example, after opening several file descriptors, we could say we are using them to access a database. But practically, we are going to have to port functions that use the full $\mathsf{IO}_{\mathsf{World}}$ monad to, a monad that uses only the resources that we *know* the function is accessing.

Haskell's `base` library contains the `unsafePerformIO` function, which has the blasphemous type signature `IO a -> a`. But it provides an escape mechanism for library authors to tell the type checker that something really doesn't have any side effects, and is safe to use as such. If we are to allow library authors to annotate their heaps correctly, one such solution would be to provide an unsafe function like this, except instead of discarding the IO part, it would discard heap annotations.

$$\mathsf{unsafeCastHeap} : \mathsf{IO}_{\rho}\alpha \to \mathsf{IO}_{\rho'}\alpha$$

There is an implicit element of trust here that this will be used selectively by library authors, only when they know that a function accesses a specific set of resources, and only those resources.

Another more radical approach to this would be to flip World on its head: Introduce a notion of an "empty" heap which contains no resources, and as such is distinct from all other heaps. Existing code in a plain IO monad would have this empty heap by default, and when a library author wants to tag resources a function might use, they would subsume it upwards into the heap they want.

Chapter 7

Conclusion

We have presented a pragmatic approach to reasoning about concurrent access of resources within a type system. We build upon the Hindley-Damas-Milner type system, incorporating the idea of modelling effects with monads, and extend it with our resourceful constructs. We can prove it retains type soundness, and in the process of doing so we have also created a framework for mechanising the proof of HDM's soundness within Agda. Furthermore, what we have built here serves an excellent basis for future work. There are many different approaches yet to explore that could extend the system in interesting directions.

By allowing heaps to be merged and subsumed into using more resources than necessary, we draw parallels to the local reasoning of separation logic's frame rule. The frame rule is what allows separation logic to scale well, and so we also have good reason to believe that our system will scale for the same reasons.

And although it is simple, the system occupies a point in the design space that fits well into existing functional programming languages, providing a lot of utility for very little complexity. Concurrent programming is a notoriously difficult task, so a type system that can help whittle down cases of concurrent resource access means its one less thing the programmer needs to worry about.

Bibliography

[1] Barendregt, H. P. *The lambda calculus*, vol. 3. North-Holland Amsterdam, 1984.

[2] Chapman, J., Kireev, R., Nester, C., and Wadler, P. System f in agda, for fun and profit. In *System F in Agda, for fun and profit* (2019), vol. International Conference on Mathematics of Program Construction, Springer, pp. 255–297.

[3] Damas, L. *Type assignment in programming languages*. PhD thesis, The University of Edinburgh, 1984.

[4] Damas, L., and Milner, R. Principal type-schemes for functional programs. In *Principal Type-Schemes for Functional Programs.* (1982), vol. POPL 82, ACM, pp. 207–212.

[5] Dolan, S., and Mycroft, A. Polymorphism, subtyping, and type inference in mlsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (2017), pp. 60–72.

[6] Dubois, C. Proving ml type soundness within coq. In *Proving ML type soundness within Coq* (2000), vol. International Conference on Theorem Proving in Higher Order Logics, Springer, pp. 126–144.

[7] Eisenberg, R. A. *Dependent types in haskell: Theory and practice*. University of Pennsylvania, 2016.

[8] Kokke, W., Siek, J. G., and Wadler, P. Programming language foundations in agda. *Science of Computer Programming* (2020), 102440.

[9] Krishnaswami, N. Separation logic for a higher-order typed language. In *Separation logic for a higher-order typed language* (2006), pp. 73–82. Presented at Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE 2006.

[10] Martin-Löf, P., and Sambin, G. *Intuitionistic type theory*, vol. 9. Bibliopolis Naples, 1984.

[11] Milner, R. A theory of type polymorphism in programming. *Journal of computer and system sciences 17*, 3 (1978), 348–375.

[12] Norell, U. Dependently typed programming in agda. In *Advanced Functional Programming: Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 230–266.

[13] O'Hearn, P. Resources, concurrency, and local reasoning. *Theoretical computer science 375*, 1-3 (2007), 271–307.

[14] O'Hearn, P. Separation logic. *Communications of the ACM 62*, 2 (2019), 86–95.

[15] Peyton Jones, S. L., and Wadler, P. Imperative functional programming. In *Imperative functional programming* (1993), vol. Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 71–84.

[16] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *Separation logic: A logic for shared mutable data structures* (2002), vol. Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, IEEE, pp. 55–74.

[17] Tofte, M. *Operational semantics and polymorphic type inference*. PhD thesis, The University of Edinburgh, 1988.

[18] Wadler, P. Monads for functional programming. In *Monads for functional programming* (1995), vol. International School on Advanced Functional Programming, Springer, pp. 24–52.

[19] Weirich, S., Voizard, A., de Amorim, P. H. A., and Eisenberg, R. A. A specification for dependent types in haskell. *Proceedings of the ACM on Programming Languages 1*, ICFP (2017), 31.

[20] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. *Information and computation 115*, 1 (1994), 38–94.

Appendix A

Denotational semantics of Hindley-Damas-Milner

The *semantic domain* of the expression language defines the possible values an expression can have. It is a *complete partial order*, defined as

$$\begin{aligned}\mathbb{V} &= \mathbb{B}_0 + \cdots + \mathbb{B}_n + \mathbb{F} + \mathbb{W} \\ \mathbb{F} &= \mathbb{V} \to \mathbb{V} \\ \mathbb{W} &= \{\cdot\}\end{aligned}$$

$\mathbb{F}$ is the set of functions, $\mathbb{W}$ is the set of error values and $\mathbb{V}$ is the set of possible values. $\mathbb{B}_0, \ldots, \mathbb{B}_n$ represent the sets of "basic" values, such as booleans and naturals.

A complete partial order (cpo) is a pair $(D, \sqsubseteq)$ consisting of a set D and a partial order $\sqsubseteq$ (a function that orders elements in D, but not necessarily all of them, hence the term partial), such that

1. there is a least element $\bot$
2. each directed subset $x_0 \sqsubseteq \ldots \sqsubseteq x_n \sqsubseteq \ldots$ has a least upper bound (*lub*)

Since $\mathbb{V}$ represents all possible data values, we can extract an *ideal* of it to model the values of certain types. A subset I of our cpo $\mathbb{V}$ is called an ideal, iff it satisfies the following properties:

1. it is downwards closed: $\forall v_0 \in \mathbb{V}, v_1 \in \mathbb{V}, v_0 \sqsubseteq v_1 \to v_0 \in I \to v_1 \in I$.
2. it is closed under lubs of ω-chains ($v_0 \sqsubseteq v_1 \sqsubseteq v_2 \sqsubseteq \ldots$)

Bottom $\bot$ is useful for representing programs that do not terminate, like below:

$$\mathsf{let}\ x = \lambda y.y\ \mathsf{in}\ x\ x$$

What type does should this program have? It should assume the type of whatever is needed. For instance, we would expect this program to have a type of $\square$ as the argument is not used.

$$(\lambda z.\square)\ (\mathsf{let}\ x = \lambda y.y\ \mathsf{in}\ x\ x) : \square$$

An environment η is defined as a map from identifiers to values

$$\mathsf{Environment} : \mathsf{Id} \to \mathbb{V}$$

The semantic equation for the expression language describes how the syntax (enclosed in $[\![\ ]\!]$) is evaluated in a given environment.

$$\begin{aligned}
\mathcal{E} &: \mathsf{Expression} \to \mathsf{Environment} \to \mathbb{V} \\
\mathcal{E}[\![x]\!]\eta &= \eta[\![x]\!] \\
\mathcal{E}[\![e_1 e_2]\!]\eta &= \begin{cases} \bot & \mathsf{if}\ v_1 = \bot \\ (v_1 | \mathbb{F}) v_2 & \mathsf{if}\ v_1\ \mathbb{F} \\ \mathsf{wrong} & \mathsf{otherwise} \end{cases} \\
&\quad \mathsf{where}\ v_i = \mathcal{E}[\![e_i]\!]\eta,\ i = \{1, 2\} \\
\mathcal{E}[\![\lambda x.\ e]\!]\eta &= (\lambda v.\ \mathcal{E}[\![e]\!]\eta[v/x])\ \mathsf{in}\ \mathbb{V} \\
\mathcal{E}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]\eta &= \mathcal{E}[\![e_2]\!]\ \eta[\mathcal{E}[\![e_1]\!]\rho/x]
\end{aligned}$$

The notation used here is described in [11, 3]. There is also a type evaluation function $\mathcal{T} : \mathsf{Type} \to \mathsf{Valuation} \to \overline{\mathbb{V}}$

$$\begin{aligned}
\mathcal{T}[\![\square]\!]\psi &= \mathbb{B}_{\square} \\
\mathcal{T}[\![\mathsf{Bool}]\!]\psi &= \mathbb{B}_{\mathsf{Bool}} \\
\mathcal{T}[\![\alpha]\!]\psi &= \psi[\![\alpha]\!] \\
\mathcal{T}[\![\tau \to \tau']\!]\psi &= \mathcal{T}[\![\tau]\!]\psi\ \to\ \mathcal{T}[\![\tau']\!]\psi
\end{aligned}$$

Where $\overline{\mathbb{V}}$ is the set of all ideals in $\mathbb{V}$ that do not contain wrong. Given a valuation, we can define a relation between values and types.

$$v :_{\psi} \tau \iff v \in \mathcal{T}[\![\tau]\!]\psi$$

With it, we can also define a relation between environments η and type environments Γ (contexts), that says an environment respects a type environment if each binding in the context has the same type in the environment.

$$\eta :_{\psi} \Gamma \iff \forall x : \tau \in \Gamma\ \eta[\![x]\!] :_{\psi} \tau$$

Finally, we can then define what semantic entailment means given a type environment, expression and type.

$$\Gamma \vDash e : \tau \iff \forall\psi\forall\eta\ \eta :_{\psi} \Gamma \to \mathcal{E}[\![e]\!]\eta :_{\psi} \tau$$

Appendix B

Proofs

B.1 Transitivity of the subheap relation

Proof. We want to show $c \geq: a$. Proceed with induction on $c \geq: b$.

Top c must be World, so from Top we have $c \geq: a$.

Refl By definition of Refl, $b = c$, and so we get $c \geq: a$ from $b \geq: a$.

UnionL c is of the form $\rho' \cup \rho''$, and from the premise we have $\rho' \geq: b$. Use the induction hypothesis with $b \geq: a$ and $\rho' \geq: b$ to get $\rho' \geq: a$, and then UnionL gives us $\rho' \cup \rho'' \geq: a$.

UnionR c is of the form $\rho'' \cup \rho'$, and from the premise we have $\rho'' \geq: b$. Use the induction hypothesis with $b \geq: a$ and $\rho'' \geq: b$ to get $\rho'' \geq: a$, and then UnionR gives us $\rho'' \cup \rho' \geq: a$.

□

B.2 Helper lemmas

Lemma 10. *If* $\Gamma \vdash [\![e]\!] : \mathsf{IO}_\rho \tau$, *then* $\Gamma \vdash e : \tau$.

Proof. This might seem obvious, but because of subsumption we need to unwravel for the proof for it first. Begin with induction on $\Gamma \vdash [\![e]\!] : \mathsf{IO}_\rho \tau$.

Lift Straight from the premises, $\Gamma \vdash e : \tau$.

Sub We have the premise $\Gamma \vdash [\![e]\!] : \mathsf{IO}_{\rho'} \tau$. Just put this back into the induction hypothesis to get $\Gamma \vdash e : \tau$.

□

Lemma 11. *If* $\Gamma \vdash [\![e]\!]_r : \mathsf{IO}_\rho \tau$, *then* $\Gamma \vdash e : \tau$.

Proof. Identical to that of Lemma 10, except we handle the case Use instead of Lift. □