1 Labeling Of Unlabeled Object

1.1 Basic Example

Assume we have a `File` resource in the global context Γ. Consider the following program, which takes a logger as input and uses it to record an error message.

```
new_d x ⇒ {
   def errmsg(y : { def log : Str }) : Unit =
      y.log(‘‘runtime exception’’)
}.errmsg(
   new_d x ⇒ {
      def log(z : Str) : Unit =
         File.close
   }
)
```

Because the input and output types of the provided logger's `log` method are value-types, then we may do straight-forward inference: the free variables of the body intersect with Γ is `File`, so we label the method as having every effect on `File`. The labelled version looks like this:

```
new_σ x ⇒ {
   def log(z: Str) : Unit with { File.close, File.write, ... } =
      File.close
}
```

1.2 Naked Objects

Here's an object literal which records error messages through a given logger, but is deeply unlabelled. Here the type of `Logger` is something which takes a `String` and returns `Unit`. After labelling, what should its type be?

```
new_d x ⇒ {
   def errmsg(y : { def log : Str }) : Unit =
      y.log(‘‘runtime exception’’)
}
```

There doesn't seem to be any sensible answer as to what to label `log`. If we know this occurs as a subexpression of a method-call, we may solve this problem by labelling the type of the expression passed into the method-call before labelling the type of the formal argument.

If this doesn't appear as a subexpression (i.e. it's a top-level, naked object) then it doesn't matter what the labelling is as `errmsg` is never called. Therefore we might decide to leave it undefined. However, this means the image of `label` won't necessarily be an e_l term (which may only contain fully-labelled types). Another idea: give it a "dummy" labelling, just so it conforms to the form of e_l?

If you have to label `def` $m(y : \tau_{u,A}) : \tau_{u,B} = e_{body}$ then when you label $\tau_{u,B}$ you need to pass e_{body} with it; and when you label $\tau_{u,A}$ you need to either signify that this happens inside a naked object, or else this is a subexpression in a method-call on your parent object, so you want to pass in the argument to that method-call.

2 Encodings

2.1 No Arguments

```
def m() : τ = e
```

$\Downarrow$

```
def m(y : Unit) : τ = e
```

2.2 Let Expressions

Transformation

```
let y = $e_{l,1}$ : $\tau_{l,1}$ in $e_{l,2}$ : $\tau_{l,2}$
```

$\Downarrow$

```
new$_d$ x ⇒ {
    def eval(y : $\tau_{l,1}$) : $\tau_{l,2}$ =
        $e_{l,2}$
}.eval($e_{l,1}$)
```

2.3 Tuples

The pair $< e_1 : \tau_1, e_2 : \tau_2 >$ becomes:

```
new$_d$ x ⇒ {
    def fst(): $\tau_1$ = $e_1$
    def snd(): $\tau_2$ = $e_2$
}
```

Referring to e_1 is shorthand for calling `fst` on this pair; e_2 for calling `snd`.

2.4 Booleans

(Types in this section are probably wrong! Could do it with a `Dyn` type, but can you get away without it?)

The literal `True` is:

```
new$_d$ x ⇒ {
    def eval(y: <$e_1$:Unit, $e_2$:Unit>): Unit =
        y.fst()
}
```

The literal `False` is:

```
new$_d$ x ⇒ {
    def eval(y: <$e_1$:Unit, $e_2$:Unit>): Unit =
        y.snd()
}
```

The type `Bool` is syntactic sugar for the following type:

```
{ def eval(y: <$e_1$:Unit, $e_2$:Unit>): Unit }
```

$(b_1 : \texttt{Bool}) \wedge (b_2 : \texttt{Bool})$ is sugar for the following:

```
new$_d$ x ⇒ {
    def eval(y: <$b_1$:Bool, $b_2$:Bool>): Unit =
        $b_1$.eval(<$b_2$.eval(<True, False>), False>)
}
```

$\neg(b : \texttt{Bool})$ is sugar for the following:

```
new$_d$ x ⇒ {
    def eval(y: <$b$:Bool, _:Unit>): Unit =
        b.eval(<False, True>)
}
```

Because $\{\neg, \wedge\}$ is a functionally complete set of connectives this gives you propositional boolean logic.

2.5 Conditionals

```
if x: Bool then e_1: τ else e_2: τ
```

$\Downarrow$

```
x.eval(<e_1, e_2>)
```

2.6 Encoding $\mathbb{N}$

Define $0 : \mathbb{N}$ in the following way.

```
new_d x ⇒ {
    def isZero(): Bool = True
    def previous(): ℕ = x
}
```

For $n : \mathbb{N}$, define $\texttt{succ}(n) : \mathbb{N}$ as follows.

```
new_d x ⇒ {
    def isZero(): Bool = False
    def previous(): ℕ = n
}
```

Then the type $\mathbb{N}$ is an alias for an object with two methods, `isZero` and `previous` (not actually valid because no recursive types, but eh). Something like 2 has the following representation:

```
new_d x ⇒ {
    def isZero(): Bool = False
    def previous(): ℕ =
        new_σ x ⇒ {
            def isZero(): Bool = False
            def previous(): ℕ =
                new_σ x ⇒ {
                    def isZero(): Bool = True
                    def previous(): ℕ = n
                }
        }
}
```

2.7 N-Tuples

With pairs and $\mathbb{N}$ we can define an n-tuple in the following way. Given an n-tuple $t_n =< e_1 : \tau_1, ..., e_n : \tau_n >$, the $(n+1)$-tuple $t_{n+1} =< e_1 : \tau_1, ..., e_n : \tau_n, e_{n+1} : \tau_{n+1} >$ is the following:

```
new_d x ⇒ {
    def nth(index: ℕ): τ_{n+1} =
        if index.equals(n+1)
            then e_{n+1}
            else τ_n.nth(index.previous())
}
```

With this we may encode an n-ary method as a unary method, which takes a single n-tuple.