Language Spec

Terts Diepraam

1 Informal description

This language features implicit elaborations for higher-order effects. Higher-order effects are prefixed with ! to distinguish them from first-order effects. Any higher-order elaborations in scope are applied with the `elab` keyword. This means that while handlers for first-order effects are values, elaborations are not. A program consists of a list of modules. Declarations are private by default, unless prefixed with `pub`. Declarations are order-dependent: only previous declarations can be accessed per module and only previous modules can be imported. This simplifies type checking to a single-pass and prevents cyclic dependencies. Unbound type identifiers are automatically type parameters. For simplicity, they can not be constrained nor annotated, but the types can be inferred. The execution order of expressions is left to right and functions have call by value semantics.

In the sections below, the built-in type are restricted `Bool` and () . This is easily extended with strings, integers, floats and tuples. More complicated types (like lists) can be defined with custom types.

In this model, elaborations are essentially typed macros that are resolved based on where elab is used, not where the macro is used. Might be an interesting link to make. Maybe the syntax of a "macro call" should also be different from a function call (like Rust's ! suffix). If I use $x!$ for higher-order effects then it would even be symmetric between type identifier and usage. This might even be necessary to define the right order of operations in the semantics? At least, it makes it easier.

Would partial elaboration ever be useful? I.e. only elaborate the higher-order effects for which an elaboration is in scope but leave the rest? Currently, my typing judgments say that all effects must be elaborated.

2 Syntax definition

$$\text{program } p ::= m \dots m$$
$$\text{module } m ::= \texttt{mod}\ \ x\ \{d \dots d\}$$

$$
\begin{aligned}
\text{declaration } d ::= & \ \texttt{pub}\ d' \mid d' \\
d' ::= & \ \texttt{let}\ x = e \\
\mid & \ \texttt{import}\ x \\
\mid & \ \texttt{elaboration}\ x! \to \Delta\ \{o, \ldots, o\} \\
\mid & \ \texttt{effect}\ \phi\ \{s, \ldots, s\} \\
\mid & \ \texttt{type}\ x\ \{s, \ldots, s\} \\
\text{expression } e ::= & \ x \\
\mid & \ () \mid \texttt{true} \mid \texttt{false} \\
\mid & \ \texttt{fn}\ (x : T, \ldots, x : T)\ \{e\} \\
\mid & \ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\mid & \ e(e, \ldots, e) \\
\mid & \ x!(e, \ldots, e) \\
\mid & \ \texttt{handler}\ \{\texttt{return}\ (x)\{e\}, o, \ldots, o\} \\
\mid & \ \texttt{handle}\ e\ e \\
\mid & \ \texttt{elab}\ e \\
\mid & \ \texttt{let}\ x = e; e \\
\mid & \ \{e\} \\
\\
\text{signature } s ::= & \ x(T, \ldots, T)\ T \\
\text{effect clause } o ::= & \ x(x, \ldots, x)\ \{e\} \\
\\
\text{type scheme } \sigma ::= & \ T \mid \forall \alpha.\sigma \\
\text{type } T ::= & \ \Delta\ \tau \\
\mid & \ \texttt{handler}\ x\ \tau\ \tau \\
\text{value type } \tau ::= & \ x \mid () \mid \texttt{Bool} \\
\mid & \ (T, \ldots, T) \to T \\
\text{effect row } \Delta ::= & \ \langle\rangle \mid x \mid \langle \phi | \Delta \rangle \\
\text{effect } \phi ::= & \ x \mid x!
\end{aligned}
$$

3 Typing judgments

The context $\Gamma = (\Gamma_M, \Gamma_V, \Gamma_E, \Gamma_\Phi)$ consists of the following parts:

$\Gamma_M : x \to (\Gamma_V, \Gamma_E, \Gamma_\Phi)$	module to context
$\Gamma_V : x \to \sigma$	variable to type scheme
$\Gamma_E : x \to (\Delta, \{f_1, \ldots, f_n\})$	higher-order effect to elaboration type
$\Gamma_\Phi : x \to \{s_1, \ldots, s_n\}$	effect to operation signatures

A Γ_T for data types might be added.

Whenever one of these is extended, the others are implicitly passed on too, but when declared separately, they not implicitly passed. For example, Γ'' is empty except for the single $x : T$, whereas Γ' implicitly contains Γ_M, Γ_E & Γ_Φ.

$$\Gamma'_V = \Gamma_V, x : T \qquad \Gamma''_V = x : T$$

If the following invariants are violated there should be a type error:

- The operations of all effects in scope must be disjoint.
- Module names are unique in every scope.
- Effect names are unique in every scope.

3.1 Effect row semantics

We treat effect rows as multisets. That means that the row $\langle A, B, B, C\rangle$ is simply the multiset $\{A, B, B, C\}$. The | symbol signifies extension of the effect row with another (possibly arbitrary) effect row. The order of the effects is insignificant, though the multiplicity is. We define the operation set as follows:

$$\begin{aligned}\mathrm{set}(\varepsilon) = \mathrm{set}(\langle\rangle) &= \emptyset \\ \mathrm{set}(\langle A_1, \ldots, A_n\rangle) &= \{A_1, \ldots, A_n\} \\ \mathrm{set}(\langle A_1, \ldots, A_n | R\rangle) &= \mathrm{set}(\langle A_1, \ldots, A_n\rangle) + \mathrm{set}(R).\end{aligned}$$

Note that the extension uses the sum, not the union of the two sets. This means that $\mathrm{set}(\langle A|\langle A\rangle\rangle)$ should yield $\{A, A\}$ instead of $\{A\}$.

Then we get the following equality relation between effect rows A and B:

$$A \cong B \iff \mathrm{set}(A) = \mathrm{set}(B).$$

In typing judgments, the effect row is an overapproximation of the effects that actually used by the expression. We freely use set operations in the typing judgments, implicitly calling the the set function on the operands where required. An omitted effect row is treated as an empty effect row ($\langle\rangle$).

Any effect prefixed with a ! is a higher-order effect, which must elaborated instead of handled. Due to this distinction, we define the operations $H(R)$ and $A(R)$ representing the higher-order and first-order subsets of the effect rows, respectively. The same operators are applied as predicates on individual effects, so the operations on rows are defined as:

$$H(\Delta) = \{\phi \in \Delta \mid H(\phi)\} \qquad \text{and} \qquad A(\Delta) = \{\phi \in \Delta \mid A(\phi)\}.$$

3.2 Type inference

We have the usual generalize and instantiate rules. But, the generalize rule requires an empty effect row.

Koka requires an empty effect row. Why?

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma} \qquad \frac{\Gamma \vdash e : \forall \alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha \mapsto T']}$$

Where ftv refers to the free type variables in the context.

Let's see if I get this: in the generalize rule, we abstract over some unbound name, which makes so that we don't need explicit parameters. We have to check that the variable does not refer to any of the datatypes in the context. So custom data types must be in that context.

Is this all we need? I think so, because if we now have any effect row extended by an arbitrary effect row, we can match it to any effect row that includes the necessary effects.

3.3 Expressions

We freely write τ to mean that a type has an empty effect row. That is, we use τ and a shorthand for $\langle\rangle\ \tau$. The Δ stands for an arbitrary effect row. We start with everything but the handlers and elaborations and put them in a separate section.

It's possible to use the braces as a syntax for computations, which is kinda like Koka. The current use is more like Rust's braces. Since the rules for it are so simple, I'm ignoring it for semantics.

$$\frac{\Gamma_V(x) = \Delta\ \tau}{\Gamma \vdash x : \Delta\ \tau} \qquad \frac{\Gamma \vdash e : \Delta\ \tau}{\Gamma \vdash \{e\} : \Delta\ \tau} \qquad \frac{\Gamma \vdash e_1 : \Delta\ \tau \qquad \Gamma_V, x : \tau \vdash e_2 : \Delta\ \tau'}{\Gamma \vdash \texttt{let}\ x = e_1; e_2 : \Delta\ \tau'}$$

$$\frac{}{\Gamma \vdash ()\ : \Delta\ ()} \qquad \frac{}{\Gamma \vdash \texttt{true}\ : \Delta\ \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false}\ : \Delta\ \texttt{Bool}}$$

$$\frac{\Gamma_V, x_1 : T_1, \ldots, x_n : T_n \vdash c : T \qquad T_i = \langle\rangle \tau_i}{\Gamma \vdash \texttt{fn}\ (x_1 : T_1, \ldots, x_n : T_n)\ T\ \{e\} : \Delta\ (T_1, \ldots, T_n) \to T}$$

$$\frac{\Gamma \vdash e_1 : \Delta\ \texttt{Bool} \qquad \Gamma \vdash e_2 : \Delta\ \tau \qquad \Gamma \vdash e_3 : \Delta\ \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \Delta\ \tau}$$

$$\frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \to \Delta\ \tau \qquad \Gamma \vdash e_i : \Delta\ \tau_i}{\Gamma \vdash e(e_1, \ldots, e_n) : \Delta\ \tau}$$

3.4 Declarations and Modules

The modules are gathered into Γ_M and the variables that are in scope are gathered in Γ_V. Each module has a the type of its public declarations. Note that these are not accumulative; they only contain the bindings generated by that declaration. Each declaration has the type of both private and public bindings. Without modifier, the public declarations are empty, but with the `pub` keyword, the private bindings are copied into the public declarations.

$$\frac{\Gamma_{i-1} \vdash m_i : \Gamma_{m_i} \qquad \Gamma_{M,i} = \Gamma_{M,i-1}, \Gamma_{m_i}}{\Gamma_0 \vdash m_1 \ldots m_n : ()}$$

$$\frac{\Gamma_{i-1} \vdash d_i : (\Gamma'_i; \Gamma'_{\text{pub},i}) \qquad \Gamma_i = \Gamma_{i-1}, \Gamma'_i \qquad \Gamma \vdash \Gamma'_{\text{pub},1}, \ldots, \Gamma'_{\text{pub},n}}{\Gamma_0 \vdash \texttt{mod}\ x\ \{d_1 \ldots d_n\} : (x : \Gamma)}$$

$$\frac{\Gamma \vdash d : \Gamma'}{\Gamma \vdash d : (\Gamma'; \varepsilon)} \qquad \frac{\Gamma \vdash d : \Gamma'}{\Gamma \vdash \texttt{pub}\ d : (\Gamma'; \Gamma')} \qquad \frac{}{\Gamma \vdash \texttt{import}\ x : \Gamma_M(x)}$$

$$\frac{\begin{array}{c} f_i = \forall \alpha.(\tau_{i,1}, \ldots, \tau_{i,n_i}) \to \alpha\ x \\ \Gamma'_V = x_1 : f_1, \ldots, x_m : f_m \end{array}}{\Gamma \vdash \texttt{type}\ x\ \{x_1(\tau_{1,1}, \ldots, \tau_{1,n_1}), \ldots, x_m(\tau_{m,1}, \ldots, \tau_{m,n_m})\} : \Gamma'}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{let}\ x = e : (x : T)}$$

3.5 First-Order Effects and Handlers

Effects are declared with the `effect` keyword. The signatures of the operations are stored in Γ_Φ. The types of the arguments and resumption must all have no effects.

A handler must have operations of the same signatures as one of the effects in the context. The names must match up, as well as the number of arguments and the return type of the expression, given the types of the arguments and the resumption. The handler type then includes the handled effect ϕ, an "input" type τ and an "output" type τ'. In most cases, these will be at least partially generic.

The handle expression will simply add the handled effect to the effect row of the inner expression and use the the input and output type.

$$\frac{s_i = op_i(\tau_{i,1}, \ldots, \tau_{i,n_i}) : \tau_i \qquad \Gamma'_\Phi(x) = \{s_1, \ldots, s_n\}}{\Gamma \vdash \texttt{effect}\ x\ \{s_1, \ldots, s_n\} : \Gamma'}$$

$$\frac{\Gamma \vdash e_h : \texttt{handler}\ \phi\ \tau\ \tau' \qquad \Gamma \vdash e_c : \langle \phi | \Delta \rangle\ \tau}{\Gamma \vdash \texttt{handle}\ e_h\ e_c : \Delta\ \tau'}$$

$$\frac{A(\phi) \qquad \Gamma_\Phi(\phi) = \{s_1, \ldots, s_n\} \qquad \Gamma, x : \tau \vdash e_{\text{ret}} : \tau' \qquad \left[\begin{array}{c} s_i = x_i(\tau_{i,1}, \ldots, \tau_{i,m_i}) \to \tau_i \qquad o_i = x_i(x_{i,1}, \ldots, x_{i,m_i})\ \{e_i\} \\ \Gamma_V, resume : (\tau_i) \to \tau', x_{i,1} : \tau_{i,1}, \ldots, x_{i,i_m} : \tau_{i,i_m} \vdash e_i : \tau' \end{array}\right]_{1 \le i \le n}}{\Gamma \vdash \texttt{handler}\ \{\texttt{return}\ (x)\{e_{\text{ret}}\}, o_1, \ldots, o_n\} : \texttt{handler}\ \phi\ \tau\ \tau'}$$

3.6 Higher-Order Effects and Elaborations

The declaration of higher-order effects is similar to first-order effects, but with exclamation marks after the effect name and all operations. This will help distinguish them from first-order effects.

Elaborations are of course similar to handlers, but we explicitly state the higher-order effect $x!$ they elaborate and which first-order effects Δ they elaborate into. The operations do not get a continuation, so the type checking is a bit different there. As arguments they take the effectless types they specified along with the effect row Δ. Elaborations are not added to the value context, but to a special elaboration context mapping the effect identifier to the row of effects to elaborate into.

Later, we could add more precise syntax for which effects need to be present in the arguments of the elaboration operations.

The elab expression then checks that a elaboration for all higher-order effects in the inner expression are in scope and that all effects they elaborate into are handled.

It is not possible to elaborate only some of the higher-order effects. We could change the behaviour to allow this later.

$$\frac{s_i = op_i!(\tau_{i,1}, \ldots, \tau_{i,n_i}) : \tau_i \qquad \Gamma'_\Phi(x!) = \{s_1, \ldots, s_n\}}{\Gamma \vdash \texttt{effect}\ x!\ \{s_1, \ldots, s_n\} : \Gamma'}$$

$$\frac{\Gamma_\Phi(x!) = \{s_1, \ldots, s_n\} \qquad \Gamma'_E(x!) = \Delta \qquad \left[\begin{array}{c} s_i = x_i!(\tau_{i,1}, \ldots, \tau_{i,m_i})\ \tau_i \qquad o_i = x_i!(x_{i,1}, \ldots, x_{i,m_i})\{e_i\} \\ \Gamma, x_{i,1} : \Delta\ \tau_{i,1}, \ldots, x_{i,n_i} : \Delta\ \tau_{i,n_i} \vdash e_i : \Delta\ \tau_i \end{array}\right]_{1 \le i \le n}}{\Gamma \vdash \texttt{elaboration}\ x! \to \Delta\ \{o_1, \ldots, o_n\} : \Gamma'}$$

$$\frac{\left[\Gamma_E(\phi) \subseteq \Delta\right]_{\phi \in H(\Delta')} \qquad \Gamma \vdash e : \Delta'\ \tau \qquad \Delta = A(\Delta')}{\Gamma \vdash \texttt{elab}\ e : \Delta\ \tau}$$

4 Desugaring

Before we move on to semantics, we remove some of the typing information and higher-level features by desugaring.

To desugar fold the operation D over the syntax tree of an expression, where D is defined by the following equations:

$$\begin{aligned} D(\texttt{fn}\ (x_1 : T_1, \ldots, x_n : T_n)\ T\ \{e\}) &= \lambda x_1, \ldots, x_n.e \\ D(\texttt{let}\ x = e_1; e_2) &= (\lambda x.e_2)(e_1) \\ D(\{e\}) &= e \\ D(e) &= e \end{aligned}$$

5 Semantics

I'm only specifying the expression reduction. The declarations should be fairly self-explanatory, because they are basically the same as the typing rules, but with values instead of values.

Some informal remarks:

- Evaluation order is from left to right.
- The syntax does not yet have a value sort, so just assume that booleans, unit, integers, strings and handlers are values. Functions are not values directly, but the corresponding lambda expression is.
- *op*!(...) calls behave like macros. Otherwise we never assign computations to variables.
- The rest is fairly standard Koka-like semantics.
- The choice to make `elab` elaborate all higher-order effects is paying off, because it makes the X_{op} case for `elab` easy.

5.1 Reduction contexts

There are two very similar contexts, one for general expressions and one that does not go through handlers, to find the only the operations that a handler can use. Note that there is only one rule involving *op*! calls, that's because they are call by name and not call by value (they are macro-like in that sense).

$$
\begin{aligned}
E ::= {} & [] \mid E(e_1, \ldots, e_n) \mid v(v_1, \ldots, v_n, E, e_1, \ldots, e_m) \\
& \mid \texttt{if } E \texttt{ then } e \texttt{ else } e \\
& \mid \texttt{let } x = E; e \\
& \mid \texttt{handle } E\ e \mid \texttt{handle } v\ E \\
& \mid \texttt{elab } E \\
X_{op} ::= {} & [] \mid X_{op}(e_1, \ldots, e_n) \mid v(v_1, \ldots, v_n, X_{op}, e_1, \ldots, e_m) \\
& \mid \texttt{if } X_{op} \texttt{ then } e_1 \texttt{ else } e_2 \\
& \mid \texttt{let } x = X_{op}; e \\
& \mid \texttt{handle } X_{op}\ e \mid \texttt{handle } h\ X_{op} \text{ if } op \notin h \\
& \mid \texttt{elab } X_{op} \text{ if } A(op)
\end{aligned}
$$

5.2 Elaboration resolution

Elaboration resolution

5.3 Reduction rules

If I understand correctly, the δ rule is to define built-ins and constants, or something. Anyway it's probably useful.

$$
\begin{aligned}
c(v_1, \ldots, v_n) &\longrightarrow \delta(c, v_1, \ldots, v_n) \\
&\qquad \text{if } \delta(c, v_1, \ldots, v_n) \text{ defined} \\
(\lambda x_1, \ldots, x_n.e)(v_1, \ldots, v_n) &\longrightarrow e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\
\texttt{if true then } e_1 \texttt{ else } e_2 &\longrightarrow e_1 \\
\texttt{if false then } e_1 \texttt{ else } e_2 &\longrightarrow e_2 \\
\\
\texttt{handle } h\ v &\longrightarrow e[x \mapsto v] \\
&\qquad \text{where } \texttt{return } (x)\{e\} \in h \\
\texttt{handle } h\ (X_{op}[op(v_1, \ldots, v_n)]) &\longrightarrow e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, resume \mapsto k] \\
&\qquad \text{where } op(x_1, \ldots, x_n)\{e\} \in h \\
&\qquad\qquad k = \lambda y\ .\ \texttt{handle } h\ (X_{op}[y]) \\
\texttt{elab } v &\longrightarrow v
\end{aligned}
$$

$$\texttt{elab}\ (X_{op!}[op!(e_1, \ldots, e_n)]) \quad \longrightarrow \quad \texttt{elab}\ X_{op!}[e[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]] \quad \text{where } op!(x_1, \ldots, x_n)\{e\} \in \Gamma_E$$

We need some sort of context for the elaborations, because they are implicit. Assume that it's populated with the elaborations that are in scope.

6 Desugaring/Compilation of Elaborations

We want to define a transformation which removes all elaborations from the program, while retaining the semantics.

6.1 The Syntax Approach

Silly idea 1:

- Wrap everything in elabs recursively.
- This removes a "scope": each elab now only goes one level deep.
- Then we fold over this, by applying modified versions of each elab.

Silly idea 2:

- We distribute each elab down using some algebraic laws
- We need to prove those or take them from the hefty algebras paper?
- Which means only higher-order operations are wrapped in elab
- Now, we can use the same rule as our operational semantics to reduce the elab/operation combo.
- This is great because if we prove first laws to be equivalent in semantics, then we're only left with the same rule applied at different times.

Each elab below is resolved and carries the elaborations that it performs.

$$
\begin{aligned}
T(\texttt{elab}_\texttt{x}\ x) &\Rightarrow x \\
T(\texttt{elab}_\texttt{x}\ ()\ ) &\Rightarrow () \\
T(\texttt{elab}_\texttt{x}\ \texttt{true}\ ) &\Rightarrow \texttt{true} \\
T(\texttt{elab}_\texttt{x}\ \texttt{false}\ ) &\Rightarrow \texttt{false} \\
T(\texttt{elab}_\texttt{x}\ \lambda x_1, \ldots, x_n.e) &\Rightarrow \lambda x_1, \ldots, x_n.e \\
T(\texttt{elab}_\texttt{x}\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &\Rightarrow \texttt{if}\ \texttt{elab}_\texttt{x}\ e_1\ \texttt{then}\ \texttt{elab}_\texttt{x}\ e_2\ \texttt{else}\ \texttt{elab}_\texttt{x}\ e_3 \\
T(\texttt{elab}_\texttt{x}\ e(e_1, \ldots, e_n)) &\Rightarrow (T'(\texttt{elab}_\texttt{x}\ e))(\texttt{elab}_\texttt{x}\ e_1, \ldots, \texttt{elab}_\texttt{x}\ e_n) \\
T(\texttt{elab}_\texttt{x}\ \texttt{handler return}\ (x)\{e\}, s_1\{e_1\}, \ldots, s_n\{e_n\}) &\Rightarrow \texttt{handler return}\ (x)\{\texttt{elab}\ e\}, s_1\{\texttt{elab}\ e_1\}, \ldots, s_n\{\texttt{ela} \\
T(\texttt{elab}_\texttt{x}\ \texttt{handle}\ e_1\ e_2) &\Rightarrow \texttt{handle}\ \texttt{elab}_\texttt{x}\ e_1\ \texttt{elab}_\texttt{x}\ e_2 \\
T(e) &\Rightarrow e
\end{aligned}
$$

$$
\begin{aligned}
T'(\texttt{elab}_\texttt{x}\ \lambda x_1, \ldots, x_n.e) &= \lambda x_1, \ldots, x_n.T(\texttt{elab}_\texttt{x}\ e) \\
T'(e) &= T(e)
\end{aligned}
$$

Both T and T' are folds i don't know how to write that nicely.

Note that $\texttt{elab}_\texttt{x}\ x!(e_1, \ldots, e_n)$ is the only expression wrapped in an `elab` that is preserved. So now we can elaborate those (assuming that by well-typedness $\texttt{elab}_\texttt{x}$ contains the elaboration for $x!$). Under this assumption, the reduction rule for elab becomes:

$$
\begin{aligned}
\texttt{elab}_\Gamma\ x!(e_1, \ldots, e_n) \quad \longrightarrow \quad & \texttt{elab}_\Gamma\ e[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n] \\
& \text{where } x!(x_1, \ldots, x_n)\ e \in \Gamma
\end{aligned}
$$

Because we are just applying the rule from the reduction semantics, this is guaranteed to preserve semantics.

Is this always true? Is there known proof for this?

Now we repeat this operation until there are no more elabs in the program.

If we now show that each of these operations preserve semantics, we prove that the entire compilation procedure preserves semantics.

So that was a dead end.

The problem lies with the following case:

$$
\texttt{elab}_1\ \{\texttt{elab}_2\ \{\lambda x.x()\}(\lambda x.a!())\}
$$

First not how, if we just return a lambda value, no elaboration takes place in the body, because the lambda is not evaluated in that context. So $\texttt{elab}_1$ does not elaborate $a!$, but, it should, because the lambda with $a!$ is passed to the other lambda and applied within $\texttt{elab}_1$. Hence, figuring out which elaboration should be applied to each operation becomes a whole program analysis and is therefore infeasible.

6.2 The Type Approach

It turns out that there is a correlation between the types and which elaborations should be applied. Take the previous example. If we add unique identifiers to each higher-order operation, we get:

$$\texttt{elab}_1\ \{\texttt{elab}_2\ \{\lambda x.x()\}(\lambda x.a_1!())\}$$

Now we add those identifiers to the effect row. So the type for $a_1!()$ is $\langle A! : \{1\}\rangle()$ and we keep those identifiers throughout the type analysis. The type of the subexpression of $\texttt{elab}_1$ is then also $\langle A! : \{1\}\rangle()$ and we can substitute the each indentifier in the effect row with the elaboration.

This works quite well, because it will always find an appropriate elaboration, but it still breaks down.

The problem is that one operation in the syntax tree might be associated with multiple elaborations. Take this program for example:

```
elab₁ {
  let g = elab₂ {
    let f = λx.a!()
    if k then
      f
    else {
      let r = f()
      λx.()
    }
  }
  g()
}
```

Following the type analysis we just established, the subexpression of $\texttt{elab}_2$ has the type:

$$\langle A! : \{1\}\rangle\Big(() \ \to \langle A! : \{1\}\rangle() \ \Big).$$

The identifier 1 shows up multiple times in the signature and could therefore be elaborated by either $\texttt{elab}_1$ or $\texttt{elab}_2$, depending on the value of k. This means that it's impossible in general to find a unique elaboration for each operation in the program, which definitely throws a wrench in our plans.

Of course, we could still remove *most* higher-order effects with this approach, as long as the identifiers are unique in signature.

6.3 The Handler Approach

So now we know that some elaborations need to be determined at runtime, but that does not mean that we can't desugar them. The idea is as follows.

Let's see what we learned from the previous attempt: figuring out the right elaboration is hard (or impossible), but elab and handle use very similar reduction semantics, so we might be able to exploit that.

So we leave the "decision" on the elaboration to the runtime evaluation. Imagine a function that takes an elaboration identifier and evaluates the result of applying that elaboration. Now we just need some way to determine which elaboration to apply. We can use (algebraic) effects for that, because of the similar reduction semantics.

Say we have an effect $A!$ with operation $a!$ and elaborations $E_1, \ldots, E_N$. Let $E_i[a!(\ldots)]$ be the elaboration of $a!$ by E_i. We then substitute each operation $a!(\ldots)$ with

$$
\begin{array}{l}
\{ \\
\quad \texttt{let } e = ask_{A!}() \\
\quad \texttt{if } eq(e, 1) \texttt{ then } E_1[a!(\ldots)] \\
\quad \texttt{else if } eq(e, 2) \texttt{ then } E_2[a!(\ldots)] \\
\quad \ldots \\
\quad \texttt{else if } eq(e, N-1) \texttt{ then } E_{N-1}[a!(\ldots)] \\
\quad \texttt{else } E_N[a!(\ldots)] \\
\}
\end{array}
$$

We might need to make e a unique variable here. It could of course also be inlined, but this clearer for demonstration purposes.

Note that this expression depends on the subexpression of the operation. It cannot be expressed as a function, only as a macro, but those are not in the language.

Then, we substitute each $\texttt{elab } \{e\}$ with an elaboration E_i with

$$\texttt{handle } H_{A!}(i) \ \{e\}$$

where H is defined as

$$
\begin{array}{l}
\texttt{let } H_{A!} = \lambda i.\texttt{handler } \{ \\
\quad \texttt{return } (x)\{x\} \\
\quad ask_{A!}()\{resume(i)\} \\
\}
\end{array}
$$

The *ask* operation is used to evoke the semantics of the reader effect (without *local*, so just the algebraic part). It essentially checks what handler is being used.

From here, we can then simplify if we want to by reasoning about which handlers apply in which situations, but that is not necessary.

Of course, `elab` could elaborate multiple effects, which means that it would compile to nested handlers.

Alternatively, there is a single *Elab* effect, with an operation *ask* which takes two arguments: the higher-order effect indentifier and the elaboration identifier. It remains to be seens which one is easier. The semantics should be equivalent.

What's nice about this approach is that it is purely local and it exploits the similarities between the evaluation of handlers and elaborations. It does not require a full analysis of the program. It only requires collecting all the elaborations and giving them unique identifiers.

What's weird is that this adds additional effects, so we need some additional arguments to prove that the transformation is not just correct but also well-typed. Proving the operational semantics equivalent should be fairly straightforward though, because handlers and elaborations depend on the same context in the reduction semantics.

Another cool thing here is that `elab` can be partial, i.e. elaborate only some effects, not all of them.

This could still be combined with the type approach, where the type approach is applied when the elaboration is unambiguous and the handler approach is used as fallback in more difficult cases. But that means that we would have to prove both approaches correct (and their combination), which is much harder of course.

6.4 How do we prove the handler approach to be correct

Let P be the original program and Q the transformed program.

Obviously, any program without higher-order effects and elaborations will behave the same, before and after the transformation. Similarly, any subexpression without elab and higher-order effects will behave the same.

By well-typedness any higher-order operation is a subexpression of an elab.

We should be able to make the following arguments:

- Until the transformed elab P and Q have the same trace.
- Inside the elab in P and handlers in Q, the trace is the same until the context focuses on an operation.
- With a few steps in Q we then transform the expression into the same expression as in P.
- We repeat the last two steps.
- When the subexpression of the elab/handlers reduces to a value, they both reduce to the value.
- Then we again have identical traces until the next elab.

Now actually doing this proof is difficult. Let's see if we can write this down somehow:

Actually write this out.