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

$$\begin{aligned}
\text{program } p &::= m \ldots m \\
\text{module } m &::= \texttt{mod} \ x \ \{d \ldots d\}
\end{aligned}$$

$$\text{declaration } d ::= \texttt{pub } d' \mid d'$$
$$d' ::= \texttt{let } x = e$$
$$\mid \texttt{import } x$$
$$\mid \texttt{elaboration } x! \to \Delta \ \{o, \ldots, o\}$$
$$\mid \texttt{effect } \phi \ \{s, \ldots, s\}$$
$$\mid \texttt{type } x \ \{s, \ldots, s\}$$
$$\text{expression } e ::= x$$
$$\mid () \mid \texttt{true} \mid \texttt{false}$$
$$\mid \texttt{fn } x(x : T, \ldots, x : T) \ \{e\}$$
$$\mid \texttt{if } e \texttt{ then } e \texttt{ else } e$$
$$\mid e(e, \ldots, e)$$
$$\mid x!(e, \ldots, e)$$
$$\mid \texttt{handler } \{\texttt{return } (x)\{e\}, o, \ldots, o\}$$
$$\mid \texttt{handle } e \ e$$
$$\mid \texttt{elab } e$$
$$\mid \texttt{let } x = e; e$$
$$\mid \{e\}$$

$$\text{signature } s ::= x(T, \ldots, T) \ T$$
$$\text{effect clause } o ::= x(x, \ldots, x) \ \{e\}$$

$$\text{type scheme } \sigma ::= T \mid \forall \alpha.\sigma$$
$$\text{type } T ::= \Delta \ \tau$$
$$\mid \texttt{handler } x \ \tau \ \tau$$
$$\text{value type } \tau ::= x \mid () \mid \texttt{Bool}$$
$$\mid (T, \ldots, T) \to T$$
$$\text{effect row } \Delta ::= \langle \rangle \mid x \mid \langle \phi | \Delta \rangle$$
$$\text{effect } \phi ::= x \mid x!$$

3 Typing judgments

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

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

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?

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

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.

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

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

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

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

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

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.

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

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

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

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.

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

$$\Gamma_\Phi(x!) = \{s_1, \ldots, s_n\} \qquad \Gamma'_E(x!) = \Delta$$

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

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

4 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.

- $op!(\ldots)$ calls behave like macros. Otherwise we never assign computations to variables.

- The rest is fairly standard Koka-like semantics.

- The choice to make $\mathtt{elab}$ elaborate all higher-order effects is paying off, because it makes the X_{op} case for $\mathtt{elab}$ easy.

4.1 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).

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

4.2 Reduction rules

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

$$
\begin{aligned}
c(v_1, \ldots, v_n) &\longrightarrow \delta(c, v_1, \ldots, v_n) \\
&\qquad \text{if } \delta(c, v_1, \ldots, v_n) \text{ defined} \\
(\lambda x_1, \ldots, x_n.e)(v_1, \ldots, v_n) &\longrightarrow e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\
\texttt{let } x = v; e &\longrightarrow e[x \mapsto v] \\
\texttt{fn } x(x_1 : T_1, \ldots, x_n : T_n)\ T\ \{e\} &\longrightarrow \lambda x_1, \ldots, x_n\ .\ e \\
\texttt{if true then } e_1 \texttt{ else } e_2 &\longrightarrow e_1 \\
\texttt{if false then } e_1 \texttt{ else } e_2 &\longrightarrow e_2
\end{aligned}
$$

$$
\begin{aligned}
\texttt{handle } h\ v &\longrightarrow e[x \mapsto v] \\
&\qquad \text{where } \texttt{return } (x)\{e\} \in h \\
\texttt{handle } h\ (X_{op}[op(v_1, \ldots, v_n)]) &\longrightarrow e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, resume \mapsto k] \\
&\qquad \text{where } op(x_1, \ldots, x_n)\{e\} \in h \\
&\qquad\qquad k = \lambda y\ .\ \texttt{handle } h\ (X_{op}[y]) \\
\texttt{elab } v &\longrightarrow v \\
\texttt{elab } (X_{op!}[op!(e_1, \ldots, e_n)]) &\longrightarrow \texttt{elab } X_{op!}[e[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]] \\
&\qquad \text{where } op!(x_1, \ldots, x_n)\{e\} \in \Gamma_E
\end{aligned}
$$

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.