Elaine: Elaborations of Higher-Order Effects as First-Class Language Feature

Terts Diepraam

September 19, 2023

Elaine: Elaborations of Higher-Order Effects as First-Class Language Feature

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Terts Diepraam
born in Amsterdam, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Rubin's Vase.

Elaine: Elaborations of Higher-Order Effects as First-Class Language Feature

Author: Terts Diepraam
Student id: 5652235
Email: t.diepraam@student.tudelft.nl

Abstract

Thesis Committee:

Chair:	Prof. dr. M. Spaan, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Bach Poulsen, Faculty EEMCS, TU Delft
University Supervisor:	Ir. C. van der Rest, Faculty EEMCS, TU Delft

Contents

Todo list

Chapter 1

Elaine Specification

TODO: Custom type declarations are in the language but not explained in this chapter yet.

This chapter contains the detailed specification for Elaine: the syntax, semantics, type inference rules, and the functions provided by the standard library.

1.1 Syntax Definition

The Elaine syntax was designed to be relatively easy to parse. The grammar is white-space insensitive and most constructs are unambiguously identified with keywords at the start.

Based on the previous chapters, the `elab` without an elaboration might be surprising. The use of that syntax is explained in **??**.

The full syntax definition is given in Figure 1.1. For convenience, we use several extensions to BNF:

- tokens are written in `monospace font`, this includes the tokens `[]`, `<>`, `|`, and `!`, which might be confused with the syntax of BNF,
- $[p]$ indicates that the sort p is optional,
- $p \ldots p$ indicates that the sort p can be repeated zero or more times, and
- $p \texttt{,} \ldots \texttt{,} p$ indicates that the sort p can be repeated zero or more times, separated by commas.

1.2 Effect row semantics

Elaine's type checker uses multisets to model effect rows, meaning that the row $\langle A, B, B, C \rangle$ is represented by the multiset $\{A, B, B, C\}$. This yields a semantics where the multiplicity of effects is significant, but the order is not.

Since the effect row of a computation must match the effect row of the context in which it is used, the effect row of the computation is an overapproximation of the effects that are necessary. Therefore, we should allow effect row polymorphism, so that the same expression can be used within multiple contexts.

Effect row polymorphism is enabled via the *row tail*, which is denoted with the `|` symbol followed by an identifier.

The | symbol signifies extension of the effect row with another (possibly arbitrary) effect row. We determine compatibility between effect rows by unifying them. That is

$$
\begin{aligned}
\text{program } p &::= d \ldots d \\
\\
\text{declaration } d &::= [\texttt{pub}]\ \texttt{mod}\ x\ \{d \ldots d\} \\
&\mid [\texttt{pub}]\ \texttt{use}\ x\texttt{;} \\
&\mid [\texttt{pub}]\ \texttt{let}\ [\texttt{rec}]\ p\ \texttt{=}\ e\texttt{;} \\
&\mid [\texttt{pub}]\ \texttt{effect}\ \phi\ \{s\texttt{, } \ldots\texttt{, } s\} \\
&\mid [\texttt{pub}]\ \texttt{type}\ x\ \{s\texttt{, } \ldots\texttt{, } s\} \\
\\
\text{block } b &::= \texttt{\{}\ es\ \texttt{\}} \\
\text{expression list } es &::= e\texttt{;}\ es \\
&\mid \texttt{let}\ [\texttt{rec}]\ p\ \texttt{=}\ e\texttt{;}\ es \\
&\mid e \\
\text{expression } e &::= x \\
&\mid \texttt{()} \mid \texttt{true} \mid \texttt{false} \mid \mathit{number} \mid \mathit{string} \\
&\mid (e\texttt{, } \ldots\texttt{, } e) \\
&\mid \texttt{fn}(p\texttt{, } \ldots\texttt{, } p)\ [T]\ b \\
&\mid \texttt{if}\ e\ b\ \texttt{else}\ b \\
&\mid e(e\texttt{, } \ldots\texttt{, } e) \mid \phi(e\texttt{, } \ldots\texttt{, } e) \\
&\mid \texttt{handler}\ \{\texttt{return}(x)\ b\texttt{, } o\texttt{, } \ldots\texttt{, } o\} \\
&\mid \texttt{handle}[e]\ e \\
&\mid \texttt{elaboration}\ x\texttt{!}\ \texttt{->}\ \Delta\ \{o\texttt{, } \ldots\texttt{, } o\} \\
&\mid \texttt{elab}[e]\ e \mid \texttt{elab}\ e \\
&\mid es \\
\\
\text{annotatable variable } p &::= x\ \texttt{:}\ T \mid x \\
\text{signature } s &::= x(T\texttt{, } \ldots\texttt{, } T)\ T \\
\text{effect clause } o &::= x(x\texttt{, } \ldots\texttt{, } x)\ b \\
\\
\text{type } T &::= \Delta\ \tau \mid \tau \\
\text{value type } \tau &::= x \\
&\mid \texttt{()} \mid \texttt{Bool} \mid \texttt{Int} \mid \texttt{String} \\
&\mid \texttt{fn}(T\texttt{, } \ldots\texttt{, } T)\ T \\
&\mid \texttt{handler}\ x\ \tau\ \tau \\
&\mid \texttt{elaboration}\ x\texttt{!}\ \Delta \\
\text{effect row } \Delta &::= \texttt{<}\phi\texttt{, } \ldots\texttt{, } \phi[\texttt{|}x]\texttt{>} \\
\text{effect } \phi &::= x \mid x\texttt{!}
\end{aligned}
$$

Figure 1.1: Syntax definition of Elaine

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 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:

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

TODO: Talk about (Leijen 2005, 2014).

During type checking effect rows are represented as a pair consisting of a multiset of effects and an optional extension variable. In this section we will use a more explicit notation than the syntax of Elaine by using the multiset representation directly. Hence, a row $\langle A_1, \ldots, A_n | e_A\rangle$ is represented as the multiset $\{A_1, \ldots, A_n\} + e_A$.

Like with regular Hindley-Milner type inference, two rows can be unified if we can find a substitution of effect row variables that make the rows equal. For effect rows, this yields 3 distinct cases.

If both rows are closed (i.e. have no extension variable) there are no variables to be substituted, and we just employ multiset equality. That is, to unify rows A and B we check that $A = B$. If that is true, we do not need to unify further and unification has succeeded. Otherwise, we cannot make any substitutions to make them equal and unification has failed.

If one of the rows is open, then the set of effects in that row need to be a subset of the effects in the other row. To unify the rows

$$A + e_A \quad \text{and} \quad B$$

we assert that $A \subseteq B$. If that is true, we can substitute e_n for the effects in $B - A$.

Finally, there is the case where both rows are open:

$$A + e_A \quad \text{and} \quad B + e_B.$$

In this case, unification is always possible, because both rows can be extended with the effects of the other. We create a fresh effect row variable e_C with the following substitutions:

$$\begin{aligned}e_A &\to (B - A) + e_C \\ e_B &\to (A - B) + e_C.\end{aligned}$$

In other words, A is extended with the effects that are in B but not in A and, similarly, B is extended with the effects in A but not in A.

TODO: Formalize this here and move some stuff to the tour chapter.

1.3 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
$\Gamma_T : x \to T$ identifier to custom type$\Gamma_\Phi : x \to \{s_1, \ldots, s_n\}$ effect to operation signatures

INFO: 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.

1.3.1 Type inference

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

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

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

1.3.2 Expressions

The typing rules for expressions are given below. The rules for handlers and elaborations are worth considering separately and are listed in **????**.

$$\frac{\Gamma_V(x) = \Delta\ \tau}{\Gamma \vdash x : \Delta\ \tau} \qquad \frac{\Gamma \vdash e : \Delta\ \tau}{\Gamma \vdash \{e\} : \Delta\ \tau} \qquad \frac{}{\Gamma \vdash \texttt{()} : \Delta\ \texttt{()}} \qquad \frac{}{\Gamma \vdash \mathit{number} : \Delta\ \texttt{Int}}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \Delta\ \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \Delta\ \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \mathit{string} : \Delta\ \texttt{String}}$$

$$\frac{[\Gamma \vdash e_i : T_i]_{1 \le i \le n}}{\Gamma \vdash \texttt{(}e_1\texttt{, ...,}e_n\texttt{)} : (T_1, \ldots, T_n)} \qquad \frac{\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \to \Delta\ \tau \qquad \Gamma \vdash e_i : \Delta\ \tau_i}{\Gamma \vdash e\texttt{(}e_1\texttt{, ...,}e_n\texttt{)} : \Delta\ \tau}$$

$$\frac{\Gamma_V, x_1 : \langle\rangle\tau_1, \ldots, x_n : \langle\rangle\tau_n \vdash c : T}{\Gamma \vdash \texttt{fn(}x_1\texttt{: }\tau_1\texttt{, ..., }x_n\texttt{: }\tau_n\texttt{) } T\ \{e\} : \Delta\ (\tau_1, \ldots, \tau_n) \to T}$$

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

$$\frac{\Gamma \vdash e_1 : \Delta\ \tau \qquad \Gamma_V, x : \tau \vdash e_2 : \Delta\ \tau'}{\Gamma \vdash \texttt{let } x\texttt{ = } e_1\texttt{; } e_2 : \Delta\ \tau'} \qquad \frac{\Gamma_V, x : \tau \vdash e_1 : \Delta\ \tau_1 \qquad \Gamma_V, x : \tau \vdash e_2 : \Delta\ \tau_2}{\Gamma \vdash \texttt{let rec } x\texttt{ = } e_1\texttt{; } e_2 : \Delta\ \tau_2}$$

$$\frac{\Gamma \vdash e : \Delta x \qquad C_1, \ldots, C_n = \Gamma_T(x) \qquad \left[\begin{array}{c} c_i\texttt{(}x_{i,1}\texttt{, ...,}x_{i,m_i}\texttt{)} = p_i \qquad c_i(\tau_{i,1}, \ldots, \tau_{i,m_i}) = C_i \\ \Gamma, x_{i,1} : \tau_{i,1}, \ldots, x_{i,m_i} : \tau_{i,m_i} \vdash e_i : \Delta\tau \end{array}\right]_{1 \le i \le n}}{\Gamma \vdash \texttt{match } e\ \texttt{\{ } p_1\ \texttt{=>}\ e_1\texttt{, ...,}p_n\ \texttt{=>}\ e_n\ \texttt{\}} : \Delta\tau}$$

1.3.3 Declarations and Modules

The modules are gathered into Γ_M and the variables that are in scope are gathered in Γ_V. The $\Rightarrow$ relation specifies the bindings that a declaration generates. The right-hand side of this relation is a public with the private and public declarations. Without modifier, the public declarations are empty, but with the `pub` keyword, the private bindings are copied into the public declarations.

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

$$\frac{\Gamma \vdash d_1 \ldots s_n \Rightarrow (\Gamma_{\text{priv}}; \Gamma_{\text{pub}}) \qquad \Gamma_{M,x} = x : \Gamma_{\text{pub}}}{\Gamma \vdash \texttt{mod}\ x\ \{d_1 \ldots d_n\} \Rightarrow (\Gamma_{M,x}; \epsilon)}$$

$$\frac{\Gamma \vdash d \Rightarrow (\Gamma', \epsilon)}{\Gamma \vdash \texttt{pub}\ d \Rightarrow (\Gamma'; \Gamma')} \qquad \frac{}{\Gamma \vdash \texttt{use}\ x\texttt{;} \Rightarrow (\Gamma_M(x), \epsilon)}$$

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

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma'_V = x : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e\texttt{;} \Rightarrow (\Gamma', \epsilon)} \qquad \frac{\Gamma, x : \tau \vdash e : \tau \qquad \Gamma'_V = x : \tau}{\Gamma \vdash \texttt{let rec}\ x\ \texttt{=}\ e\texttt{;} \Rightarrow (\Gamma'; \epsilon)}$$

1.3.4 Algebraic 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 input and output type.

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

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

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

1.3.5 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.

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

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

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

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

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

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

1.4 Desugaring

To simplify the reduction rules, we simplify the AST by desugaring some constructs. This transform is given by a fold over the syntax tree with the following operation:

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

1.5 Semantics

The semantics of Elaine are defined as reduction semantics.

We use two separate contexts to evaluate expressions. The E context is for all constructs except effect operations, such as **if**, **let**, and function applications. The X_{op} context is the context in which a handler can reduce an operation *op*.

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

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

TODO: Add some explanation

$$
\begin{aligned}
c(v_1, \ldots, v_n) &\longrightarrow \delta(c, v_1, \ldots, v_n) \\
&\qquad \text{if } \delta(c, v_1, \ldots, v_n) \text{ defined} \\
(\lambda x_1, \ldots, x_n.e)(v_1, \ldots, v_n) &\longrightarrow e[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\
\texttt{if true}\ \{e_1\}\ \texttt{else}\ \{e_2\} &\longrightarrow e_1 \\
\texttt{if false}\ \{e_1\}\ \texttt{else}\ \{e_2\} &\longrightarrow e_2
\end{aligned}
$$

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

1.6 Standard Library

Elaine does not include any operators. This choice was made to simplify parsing of the language. For the lack of operators, any manipulation of primitives needs to be done via the standard library of built-in functions.

These functions reside in the `std` module, which can be imported like any other module with the **`use`** statement to bring its contents into scope.

The full list of functions available in the `std` module, along with their signatures and descriptions, is given in Figure 1.2.

TODO: Add all the other things I added to the standard library here too

	Name	Type signature		Description
Arithmetic	`add`	`fn(Int, Int)`	`Int`	addition
	`sub`	`fn(Int, Int)`	`Int`	subtraction
	`neg`	`fn(Int)`	`Int`	negation
	`mul`	`fn(Int, Int)`	`Int`	multiplication
	`div`	`fn(Int, Int)`	`Int`	division
	`modulo`	`fn(Int, Int)`	`Int`	modulo
	`pow`	`fn(Int, Int)`	`Int`	exponentiation
Comparisons	`eq`	`fn(Int, Int)`	`Bool`	equality
	`neq`	`fn(Int, Int)`	`Bool`	inequality
	`gt`	`fn(Int, Int)`	`Bool`	greater than
	`geq`	`fn(Int, Int)`	`Bool`	greater than or equal
	`lt`	`fn(Int, Int)`	`Bool`	less than
	`leq`	`fn(Int, Int)`	`Bool`	less than or equal
Boolean operations	`not`	`fn(Bool)`	`Bool`	boolean negation
	`and`	`fn(Bool, Bool)`	`Bool`	boolean and
	`or`	`fn(Bool, Bool)`	`Bool`	boolean or
String operations	`concat`	`fn(Bool, Bool)`	`Bool`	string concatenation
Conversions	`show_int`	`fn(Int)`	`String`	integer to string
	`show_bool`	`fn(Bool)`	`String`	integer to string

Figure 1.2: Overview of the functions in the `std` module in Elaine.

Bibliography

Hutton, Graham and Erik Meijer (1996). *Monadic parser combinators*. URL: https://nottingham-repository.worktribe.com/output/1024440.

Leijen, Daan (July 23, 2005). "Extensible records with scoped labels". In.

— (June 5, 2014). "Koka: Programming with Row Polymorphic Effect Types". In: *Electronic Proceedings in Theoretical Computer Science* 153, pp. 100–126. ISSN: 2075-2180. DOI: 10.4204/EPTCS.153.8. URL: http://arxiv.org/abs/1406.2061v1 (visited on 06/16/2023).

Appendix A

Elaine Example Programs

This chapter contains longer Elaine samples with some additional explanation.

A.1 A naive SAT solver

This program is a naive brute-forcing SAT solver. We first define a `Yield` effect, so we can yield multiple values from the computation. We will use this to find all possible combinations of boolean inputs that satisfy the formula. The `Logic` effect has two operations. The `branch` operation will call the continuation twice; once with **`false`** and once **`true`**. With `fail`, we can indicate that a branch has failed. To find all solutions, we just `branch` on all inputs and `yield` when a correct solution has been found and `fail` when the formula is not satisfied. In the listing below, we check for solutions of the equation $\neg a \wedge b$.

Elaine

```
use std;

effect Yield {
   yield(String) ()
}

effect Logic {
    branch() Bool
    fail() a
}

let hYield = handler {
    return(x) { "" }
    yield(m) {
        concat(concat(m, "\n"), resume(()))
    }
};

let hLogic = handler {
    return(x) { () }
    branch() {
        resume(true);
        resume(false)
    }
    fail() { () }
};
```

```
27
28 let show_bools = fn(a, b, c) {
29     let a = concat(show_bool(a), ", ");
30     let b = concat(show_bool(b), ", ");
31     concat(concat(a, b), show_bool(c))
32 };
33
34 let f = fn(a, b, c) { and(not(a), b) };
35
36 let assert = fn(f, a, b, c) <Logic,Yield> () {
37     if f(a, b, c) {
38         yield(show_bools(a, b, c))
39     } else {
40         fail()
41     }
42 };
43
44 let main = handle[hYield] handle[hLogic] {
45     assert(f, branch(), branch(), branch());
46 };
```

A.2 Reader Effect

The implementation of the reader effect is a standard application for higher-order effects. We start with a higher-order `Reader!` effect with an operation `local!` and an algebraic `Ask` effect. The `local!` operation is elaborated into a computation that handles the `Ask` with the modified value.

This effect corresponds to the `Reader` monad as defined by Haskell's mtl library.

Elaine

```
1  use std;
2
3  effect Ask {
4      ask() Int
5  }
6
7  effect Reader! {
8      local!(fn(Int) Int, a) a
9  }
10
11 let hAsk = fn(v: Int) {
12     handler {
13         return(x) { x }
14         ask() { resume(v) }
15     }
16 };
17
18 let eReader = elaboration Reader! -> <Ask> {
19     local!(f, c) {
20         handle[hAsk(f(ask()))] c
21     }
22 };
```

```
23
24 let double = fn(x) { mul(2, x) };
25
26 let main = handle[hAsk(2)] elab[eReader] {
27     local!(double, add(ask(), ask()));
28 };
```

A.3 Writer Effect

The implementation of the writer effect is similar to the implementation of the reader effect. Again, we elaborate a higher-order effect, `Writer!`, into an algebraic effect, `Out`, with a subset of the operations. The higher-order `censor!` operation handles the algebraic effect to access the output and applies the censoring function to it.

This effect corresponds to the `Writer` monad as defined by Haskell's mtl library.

Elaine

```
1  use std;
2
3  effect Writer! {
4      censor!(fn(String) String, a) a
5      tell!(String) ()
6  }
7
8  effect Out {
9      tell(String) ()
10 }
11
12 type Output[a] {
13     Output(String, a)
14 }
15
16 let hOut = handler {
17     return(x) { Output("", x) }
18     tell(s) {
19         match resume(()) {
20             Output(s', x) => Output(concat(s, s'), x)
21         }
22     }
23 };
24
25 let eWriter = elaboration Writer! -> <Out> {
26     tell!(s) { tell(s) }
27     censor!(f, c) {
28         match handle[hOut] c {
29             Output(s, x) => {
30                 tell(f(s));
31                 x
32             }
33         }
34     }
35 };
36
```

```
let main = handle[hOut] elab {
    tell("foo");
    censor!(fn(s) { "bar" }, {
        tell("baz");
        5
    });
};
```

A.4 Structured Logging

Since higher-order effects are suitable for delimiting the scope of effects, we can make an effect for structured logging. The idea is that every **log** call appends a message to the output, but the message is prefixed with some context. This context start out as the empty string, but within every `context!` call, a string is added to this context.

Elaine

```
use std;

effect Write {
    write(String) ()
}

effect Read {
    ask() String
}

effect Log! {
    context!(String, a) a
    log!(String) ()
}

let hRead = fn(v: String) {
    handler {
        return(x) { x }
        ask() { resume(v) }
    }
};

let hWrite = handler {
    return(x) { "" }
    write(m) {
        let rest = resume(());
        let msg = concat(m, "\n");
        concat(msg, rest)
    }
};

let eLog = elaboration Log! -> <Read,Write> {
    context!(s, c) {
        let new_context = concat(concat(ask(), s), ":");
        handle[hRead(new_context)] c
    }
```

```
    log!(m) {
        write(concat(concat(ask(), " "), m))
    }
};

let main = handle[hRead("")] handle[hWrite] elab[eLog] {
    context!("main", {
        log!("msg1");
        context!("foo", {
            log!("msg2")
        });
        context!("bar", {
            log!("msg3")
        })
    })
};
```

A.5 Parser Combinators

Monadic parser combinators (Hutton and Meijer 1996) are a popular technique for constructing parsers. The parser for Elaine is also written using `megaparsec`[1], which is a monadic parser combinator library for Haskell. Attempts have been made to implement parser combinators using algebraic effects. However, it requires higher-order combinators for a full feature set matching that of monadic parser combinators. For example, the `alt` combinator takes two branches and attempts to parse the first branch and tries the second branch if the first one fails. This is remarkably similar to the catch operation of the exception effect and is indeed higher-order.

Below is a full listing of a JSON parser written in Elaine using a variation on parser combinators using effects. It is implemented using a higher-order `Parse!` effect, which is elaborated into a state and an abort effect, which are imported from the standard library. The `try!` effect is a higher-order effect which takes an effectful computation as an argument. It applies the computation and returns its value if it succeeds, otherwise it will reset the state and return `Nothing()`.

Higher-order effects are convenient for parser combinators, but not necessary. Instead of the **`try`**`!` operation, the non-determinism effect can be used to write a backtracking parser. An implementation of that technique in Effekt available at `https://effekt-lang.org/docs/casestudies/parser`.

Elaine

```
use std;
use maybe;
use list;
use state_str;
use abort;

effect Parse! {
    # Signal that this branch has failed to parse
    fail!() a
    # Try to apply the parser, reset the state if it fails
    try!(a) Maybe[a]
```

[1] `https://github.com/mrkkrp/megaparsec`

```
    # Remove and return the first character of the input
    eat!() String
}

let eParse = elaboration Parse! -> <State,Abort> {
    fail!() { abort() }
    try!(x) {
        let old_state = get();
        match handle[hAbort] x {
            Just(res) => Just(res),
            Nothing() => {
                put(old_state);
                Nothing()
            }
        }
    }
    eat!() {
        let state = get();
        put(drop(1, state));
        take(1, state)
    }
};

### Combinators
let alt2 = fn(a, b) {
    match try!(a()) {
        Just(x) => x,
        Nothing() => b(),
    }
};

let rec alt = fn(parsers) {
    match parsers {
        Cons(p, ps) => alt2(p, fn() { alt(ps) }),
        Nil() => fail!(),
    }
};

let rec many = fn(p) {
    match try!(p()) {
        Just(x) => Cons(x, many(p)),
        Nothing() => Nil(),
    }
};

let separated = fn(
    p: fn() <Parse!> a,
    separator: fn() <Parse!> b,
) <Parse!> List[a] {
    match try!(p()) {
        Just(x) => Cons(x, many(fn() {separator(); p()})),
```

```
        Nothing() => Nil()
    }
};

### Parsers
# Parse a token specified as a string
let token = fn(s) {
    let c = eat!();
    if str_eq(s, c) {
        c
    } else {
        fail!()
    }
};

let rec contains_str = fn(s, l) {
    match l {
        Cons(x, xs) => {
            if str_eq(x, s) {
                true
            } else {
                contains_str(s, xs)
            }
        },
        Nil() => false,
    }
};

let one_of = fn(s) {
    let list_of_chars = explode(s);
    fn() {
        let c = eat!();
        if contains_str(c, list_of_chars) {
            c
        } else {
            fail!()
        }
    }
};

# Parse a single digit
let digit = one_of("0123456789");
let str_char = one_of(join([
    "0123456789",
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
    "abcdefghijklmnopqrstuvwxyz",
    "_- ?!",
]));
let white_one = one_of(" \n\t");
let white = fn() { many(white_one); () };

```

```
let tokenws = fn(s) {
    let t = token(s);
    white();
    t
};

let comma_separated = fn(p: fn() <Parse!> a) <Parse!> List[a] {
    separated(p, fn() { tokenws(",") })
};

# Parse as many digits as possible
let number = fn() { join(many(digit)) };

type Json {
    JsonString(String),
    JsonInt(String),
    JsonArray(List[Json]),
    JsonObject(List[(String, Json)]),
}

let string = fn() <Parse!> String {
    token("\"");
    let s = join(many(str_char));
    tokenws("\"");
    s
};

let key_value = fn(value: fn() <Parse!> Json) <Parse!> (String, Json) {
    let k = string();
    tokenws(":");
    (k, value())
};

let object = fn(value: fn() <Parse!> Json) <Parse!> Json {
    tokenws("{");
    let kvs = comma_separated(fn() { key_value(value) });
    tokenws("}");
    JsonObject(kvs)
};

let array = fn(value) {
    tokenws("[");
    let values = comma_separated(value);
    tokenws("]");
    JsonArray(values)
};

let rec value = fn() {
    alt([
        fn() { array(value) },
        fn() { object(value) },
```

```
        fn() { JsonString(string()) },
        fn() { JsonInt(number()) },
    ])
};

let parse = fn(parser, input) {
    let f = handle[hState] handle[hAbort] elab[eParse] parser();
    f(input)
};

let main = parse(
    value,
    "{\"key1\": 123, \"key2\": [1,2,3], \"key3\": \"some string\"}"
);
```