Higher-Order Effects with Implicitly Resolved Elaborations

Version of May 1, 2023

Terts Diepraam

Higher-Order Effects with Implicitly Resolved Elaborations

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`

Higher-Order Effects with Implicitly Resolved Elaborations

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

Abstract

Thesis Committee:

Chair:	Prof. dr. C. Hair, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bee, Faculty EEMCS, TU Delft
Committee Member:	Dr. C. Dee, Faculty EEMCS, TU Delft
University Supervisor:	Ir. E. Ef, Faculty EEMCS, TU Delft

Preface

Preface here.

Terts Diepraam
Delft, the Netherlands
May 1, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This is currently mostly a short history of effects, not quite an introduction yet.

In standard λ-calculus, there is no model for effects of the computation, only of the result. Speaking broadly, effects concern the aspects of the program besides the pure computation. [6]

In their seminal paper, Moggi [6] unified monads with computational effects, or notions of computation as they call it. Functional programming languages often rely on monads to perform effectful computations. Haskell, for example, uses an `IO` monad for printing output and reading input. The computation is then kept pure and impure operations (like reading and printing) are delegated to the system. However, a limitation of treating effects as monads is that monads do not compose well.

Plotkin and Power [7] then showed that effects can be represented as equational theories. The effects that can be represented in this framework are called algebraic effects. The `State` and the `Maybe` monads can for example be expressed in this framework.

Plotkin and Pretnar [8] then introduced effect handlers, allowing the programmer to destruct effects by placing handlers around effectful expressions. This provides a way to treat exception handling using effects. However, the scope in which an effect is handled can only be changed by adding handlers. Effect operations cannot define their own scope. To support this, a system for higher-order effects is required, which are effects that take effectful operations as parameters.

A solution to this was the framework of scoped effects [10]. However, scoped effects require a significant increase in complexity and cannot express effects that are neither algebraic nor scoped, such as lambda abstractions [3]. Latent effects [3] were subsequently introduced as an alternative that encapsulates a larger set of effects.

As an alternative approach to latent effects, Bach Poulsen and Rest [1] introduced hefty algebras. With hefty algebras, higher-order effects are treated separately from algebraic effects. Higher-order effects are not handled, but elaborated into algebraic effects, which can then be handled. The advantage is that the treatment of algebraic effects remains intact and that the process of elaboration is relatively simple.

In parallel with the work to define theoretical frameworks for effects, several libraries and languages have been designed that include effects as first-class concepts, allowing the programmer to define their own effects and handlers. For example, there are some libraries available for Haskell, like `fused-effects`[1] , `polysemy`[2] , `freer-simple`[3] and `eff`[4] , each encoding effects in a slightly different way.

[1]`https://github.com/fused-effects/fused-effects`
[2]`https://github.com/polysemy-research/polysemy`
[3]`https://github.com/lexi-lambda/freer-simple`
[4]`https://github.com/hasura/eff`

Notable examples of languages with support for algebraic effects are Eff [2], Koka [4] and Frank [5]. OCaml also gained support for effects [9]. By building algebraic effects into the language, instead of delegating to a library, is advantageous because the language can provide more convenient syntax. In these languages, some concepts that were traditionally only available with language support, can be expressed by the programmer. This includes exception handling and asynchronous programming.

These languages either only support algebraic effects or scoped effects. This means that their support for higher-order effects is limited. The exception is `heft`[5], which is produced in conjunction with the work in [1].

The aim of this thesis is to build on the work by Bach Poulsen and Rest and create a new language which

- supports higher-order effects using hefty algebras, with separate elaborations and handlers,
- implicitly resolves elaborations, and
- can be compiled to a representation with only algebraic effects.

To this end, we need to show that the compilation of implicitly resolved effects is equivalent to the operational semantics for elaborations. By showing that elaborations can be removed at compile-time, we show that it is possible to compile programs in our language using the techniques from [4].

In addition, we provide an implementation of this language, which is published on GitHub under the name `elaine`[6] (sharing a prefix with "elaboration") as part of the repository that hosts this thesis.

```
let f = \x . {
   let _ = a!()
   ()
}
elab[e1] {
   let g = elab[e2] {
      if <runtime condition> then
         f
      else {
         f()
         \x . ()
      }
   }
   g()
}
```

[5]`https://github.com/heft-lang/heft`
[6]`https://github.com/tertsdiepraam/thesis/tree/main/elaine`

Chapter 2

Basic Syntax and Semantics

Set up the basic syntax and semantics of Elaine, used throughout the thesis.

Chapter 3

Algebraic Effects

Introduce algebraic effects, effect rows, and handlers.

Chapter 4

Higher-Order Effects and Hefty Algebras

Introduce higher-order effects, the composability problem, scoped effects, hefty algebras etc.

Chapter 5

Elaboration Resolution

Chapter 6

Compiling Elaborations

Chapter 7

Conclusion

Bibliography

[1] Casper Bach Poulsen and Cas van der Rest. “Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects”. In: *Proceedings of the ACM on Programming Languages* 7 (POPL Jan. 9, 2023), pp. 1801–1831. ISSN: 2475-1421. DOI: 10.1145/3571255. URL: https://dl.acm.org/doi/10.1145/3571255 (visited on 01/26/2023).

[2] Andrej Bauer and Matija Pretnar. “Programming with algebraic effects and handlers”. In: *Journal of Logical and Algebraic Methods in Programming* 84.1 (Jan. 2015), pp. 108–123. ISSN: 23522208. DOI: 10.1016/j.jlamp.2014.02.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S2352220814000194 (visited on 04/03/2023).

[3] Birthe van den Berg et al. “Latent Effects for Reusable Language Components”. In: *Programming Languages and Systems*. Ed. by Hakjoo Oh. Vol. 13008. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 182–201. ISBN: 978-3-030-89050-6 978-3-030-89051-3. DOI: 10.1007/978-3-030-89051-3_11. URL: https://link.springer.com/10.1007/978-3-030-89051-3_11 (visited on 01/12/2023).

[4] Daan Leijen. “Type directed compilation of row-typed algebraic effects”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17: The 44th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. Paris France: ACM, Jan. 2017, pp. 486–499. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009872. URL: https://dl.acm.org/doi/10.1145/3009837.3009872 (visited on 04/08/2023).

[5] Sam Lindley, Conor McBride, and Craig McLaughlin. *Do be do be do*. Oct. 3, 2017. arXiv: 1611.09259[cs]. URL: http://arxiv.org/abs/1611.09259 (visited on 04/08/2023).

[6] E. Moggi. “Computational lambda-calculus and monads”. In: *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. Pacific Grove, CA, USA: IEEE Comput. Soc. Press, 1989, pp. 14–23. ISBN: 978-0-8186-1954-0. DOI: 10.1109/LICS.1989.39155. URL: http://ieeexplore.ieee.org/document/39155/ (visited on 04/08/2023).

[7] Gordon Plotkin and John Power. “Adequacy for Algebraic Effects”. In: *Foundations of Software Science and Computation Structures*. Ed. by Furio Honsell and Marino Miculan. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 2030. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–24. ISBN: 978-3-540-41864-1 978-3-540-45315-4. DOI: 10.1007/3-540-45315-6_1. URL: http://link.springer.com/10.1007/3-540-45315-6_1 (visited on 04/08/2023).

[8] Gordon Plotkin and Matija Pretnar. "Handlers of Algebraic Effects". In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Vol. 5502. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. isbn: 978-3-642-00589-3 978-3-642-00590-9. doi: `10.1007/978-3-642-00590-9_7`. url: `http://link.springer.com/10.1007/978-3-642-00590-9_7` (visited on 04/08/2023).

[9] K. C. Sivaramakrishnan et al. "Retrofitting Effect Handlers onto OCaml". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. June 19, 2021, pp. 206–221. doi: `10.1145/3453483.3454039`. arXiv: `2104.00250[cs]`. url: `http://arxiv.org/abs/2104.00250` (visited on 04/08/2023).

[10] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. *Effect Handlers in Scope*. June 10, 2014.

Appendix A

Elaine Specification

A.1 Syntax definition

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

$$
\begin{aligned}
&\mid (T, \ldots, T) \to T \\
&\mid \texttt{handler}\ x\ \tau\ \tau \\
&\mid \texttt{elaboration}\ x!\ \Delta \\
\text{effect row } \Delta &::= \langle\rangle \mid x \mid \langle \phi | \Delta \rangle \\
\text{effect } \phi &::= x \mid x!
\end{aligned}
$$

A.2 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.

A.2.1 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:

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

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

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

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

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:

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

A.2.2 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.

A.2.3 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.

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

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

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

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

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

A.2.4 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.

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

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

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

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

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

A.2.5 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.

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

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

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

A.2.6 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

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

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

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

A.3 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:

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

A.4 Elaboration resolution

A.5 Semantics

A.5.1 Reduction contexts

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

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

A.5.2 Reduction rules

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