A Quantitative Understanding of Exceptions

Miguel Ramos[1], Riccardo Treglia[2], and Delia Kesner[3]
jmiguelsramos@gmail.com riccardo.treglia@unibo.it kesner@irif.fr

[1]LIACC, DCC, Faculdade de Ciências, Universidade do Porto
[2]Department of Computer Science, University of Bologna
[3]IRIF, CNRS, Université Paris Cité & Institut Universitaire de France

1 Introduction

We show that non-idempotent intersection types can be used to obtain a quantitative relational model for a computational version of Plotkin's open call-by-value λ-calculus equipped with effectful operations given by exception handling. To achieve this result we adopt the monadic approach proposed by Moggi [Mog89] and popularized by Wadler [Wad95], which reflect the monad semantics into the operational rules of the language and its associated type system. Indeed, according to Moggi, call-by-value effectful programs can be abstracted by a generic calculus based on a few common treats, which are then combined with complementary specific rules for the effectful computations. The call-by-value λ-calculus that is going to be presented, equipped with handling operations over all possible exceptions, is an instance of this generic formalism. In fact, we combine at the computational level both nullary algebraic operators of the form $\mathtt{raise}_e()$ for raising an exception e, and specific effects handlers of the form $\mathtt{handle}_e(t, u)$, which are syntactical constructors for handling raised exceptions. Actually, both operators can be seen as families of operators indexed over a set of possible exceptions $\mathcal{E}$.

The relational model that we propose is specified as a monadic intersection type system. More precisely, we use recent *tightness* techniques [AGK20], which, instead of providing upper bounds for the evaluation lengths of terms, give *exact* measures. Indeed, judgments are decorated with counters capturing both the number of evaluation steps, as well as the size of normal forms. Besides the usual counter for the number of β steps, we have two additional counters that measure the effectful behaviour of the language, one to count the successful handling of exceptions and another one to count both exceptions that are propagated, and unsuccessful handling. We show quantitative soundness and completeness of the type system with respect to the language's operational semantics.

2 Syntax and Operational Semantics

Let $\mathcal{E}$ be a set of constants to identify exceptions. The **set of terms** is given by the following grammar:

$$\begin{array}{rcl} v, w & ::= & x \mid \lambda x.t \\ t, u, p & ::= & v \mid vt \mid \mathtt{handle}_e(t, u) \mid \mathtt{raise}_e() \end{array}$$

We define different predicates over terms in order to simplify the presentation of the reduction rules: we write $\mathtt{isvalue}(t)$ iff t is a variable or of the form $\lambda x.u$; $\mathtt{israise}_e(t)$ iff t is of the form $\mathtt{raise}_e()$; and $\mathtt{israise}(t)$ whenever we do not care about e.

The **reduction relation** are defined by the following rules:

$$\frac{}{(\lambda x.t)v \to t\{x \backslash v\}}\ (\mathtt{b}) \qquad \frac{}{v\ \mathtt{raise}_e() \to \mathtt{raise}_e()}\ (\mathtt{r})$$

$$\frac{}{\mathtt{handle}_e(\mathtt{raise}_e(), t) \to t}\ (\mathtt{h1}) \qquad \frac{[\mathtt{isvalue}(t)] \text{ or } [\mathtt{israise}_{e'}(t)\ (e' \neq e)]}{\mathtt{handle}_e(t, u) \to t}\ (\mathtt{h2})$$

$$\frac{t \to u}{\mathtt{handle}_e(t, p) \to \mathtt{handle}_e(u, p)}\ (\mathtt{c1}) \qquad \frac{t \to u}{vt \to vu}\ (\mathtt{c2})$$

We write $\twoheadrightarrow$ for the **reflexive-transitive closure** of $\to$, and $t \twoheadrightarrow^{(b,h,p)} u$ if t reduces to u in b b-steps, h h1-steps, and p r/h2-steps. Rule (r) propagates exceptions through applications. The successful handling of exceptions is captured by (h1), since it takes place when there is a raise in the evaluation position of the handler and both are indexed by the same exception. Rule (h2) captures the unsuccessful handling of exceptions: when there is either a value or a raise indexed by a different exception in the evaluation position of the handler.

Our operational semantics respects Haskell's behavior when handling exceptions. However, while Haskell continuations are attached to exceptions, in this language continuations are attached to handlers.

Example 2.1. In the following Haskell example, the `catch` function will catch the following exception `Exception (print "div by zero")` thrown when a division by zero occurs, and apply `id` to continuation `print "div by zero"`:

```
catch (((λx.λy.if y == 0 then (throw (print "div by zero")) else (x/y))1)0) id
→⁴ catch (Exception (print "div by zero")) id →² print "div by zero"
```

Using `handle` and `raise`, we can capture the whole behavior by simply parameterizing the operations by an exception handling division by zero `div0`:

```
handle_div0(((λx.λy.if y == 0 then raise_div0() else (x/y))1)0, id (print "division by zero"))
→³ handle_div0(raise_div0(), id (print "division by zero")) →² print "division by zero"
```

Definition 2.2. *The size of normal forms is defined as follows:*

$$|\mathtt{raise}_e()| = 1 \quad |x| = 0 \quad |\lambda x.t| = 0 \quad |vt| = 1 + |t| \quad |\mathtt{handle}_e(t, u)| = 1 + |t|$$

As rule (c2) specify, our reduction relation is a weak strategy, *i.e.*, a deterministic reduction relation that does not reduce in the body of abstractions, as customary in effectful languages. Being so, it makes sense to assign size zero to abstractions that are either not applied

The **set of normal forms** can be characterized by the following grammar:

$$\begin{array}{rrcl} \text{(Neutral)} & \mathtt{ne} & ::= & x\ \mathtt{n}\phi \mid (\lambda x.t)\ \mathtt{ne} \mid \mathtt{handle}_e(\mathtt{ne}, t) \\ \text{(Pseudo-Normal)} & \mathtt{n}\phi & ::= & x \mid \lambda x.t \mid \mathtt{ne} \\ \text{(Normal)} & \mathtt{no} & ::= & \mathtt{raise}_e() \mid \mathtt{n}\phi \end{array}$$

If we ignore the operations that handle exceptions, we can see that these normal forms are precisely the ones expected for Plotkin's open call-by-value λ-calculus, when we restrict applications to only have values on the left. Since we consider $\mathtt{raise}_e()$ to be a valid result of a computation, we have to add it to the set of normal forms. The behavior of $\mathtt{handle}_e(t, u)$, however, depends on the form of t (in fact, this is the reason why these operations are not algebraic as stated in the Section 1). Since the behavior of a

handling operator depends on the subterm t in the evaluation position, when t is a normal form that is neither a value, nor a raise, the wrapping handling should be considered as "stuck".

Finally, since a stuck $\mathtt{handle}_e(t,u)$ is also neither a value, nor a raise, it is a neutral term. Also, note that, even though $\mathtt{raise}_e()$ is a normal form, it does not produce a normal form when appearing at the right of an application. For that reason, we separate $\mathtt{raise}_e()$ from the rest of the normal forms.

The following proposition states that the set of normal forms is characterized by the above grammar.

Proposition 2.3 (Syntactic Characterization of Normal Forms)**.** *Let t be a term. Then, $t \not\to$ iff $t \in \mathtt{no}$.*

Example 2.4. Catching exceptions:

$$\begin{array}{lll} (\lambda x.\mathtt{handle}_e(xy,x))\ (\lambda z.\mathtt{raise}_e()) & \to_{\mathtt{b}} & \mathtt{handle}_e((\lambda z.\mathtt{raise}_e())\ y, (\lambda z.\mathtt{raise}_e())) \\ \to_{\mathtt{b}} \mathtt{handle}_e(\mathtt{raise}_e(), \lambda z.\mathtt{raise}_e()) & \to_{\mathtt{h1}} & \lambda z.\mathtt{raise}_e() \end{array}$$

And notice that $|\lambda z.\mathtt{raise}_e()| = 0$.

3 Type System

Like in monadic idempotent intersection types [GVT23], and in quantitative global memory with non-idempotent types [AKR23], we design our type system by having in mind that both terms and (non-idempotent) intersection types become monadic. For this, we combine Girard's boring CBV translation of intuitionistic logic into linear logic [Gir87] with Moggi's CBV translation [Mog89], to obtain the translation $A \Rightarrow B = !A \multimap T(!B)$, where T depends on the choice of monad, that, in turn, will depend on the effect that is being considered. For the purposes of this paper, we will consider the exceptions monad $T_X = X \to \mathcal{E} \oplus X$, where $\oplus$ is the disjoint union and $\mathcal{E}$ is a set of names of exceptions. To distinguish between persistent and consuming constructors, we follow [AGK20] and [KV20] by introducing tight constant and persistent typing rules.

Let $\star$ represent the **unit type**. The **set of types** is given by the following grammar:

$$\begin{array}{rrcl} \text{(Tight Constants)} & \mathtt{tt} & ::= & \mathtt{v} \mid \mathtt{a} \mid \mathtt{r} \mid \mathtt{n} \\ \text{(Substitutable Types)} & \sigma & ::= & \mathtt{v} \mid \mathtt{a} \mid \mathcal{M} \Rightarrow \delta \\ \text{(Multi-Types)} & \mathcal{M} & ::= & [\sigma_i]_{i \in I} \text{ where } I \text{ is a finite set} \\ \text{(Value Types)} & \mu & ::= & \mathtt{v} \mid \mathtt{a} \mid \mathcal{M} \\ \text{(Types)} & \tau & ::= & \mathtt{tt} \mid \mathcal{M} \\ \text{(Monadic Types)} & \delta & ::= & \gamma \oplus \tau \text{ where } \gamma \in \mathcal{E} \cup \{\star\} \end{array}$$

Constant types are used to deal with persistent symbols, *i.e.*, symbols that are not going to be consumed during evaluation, and the choice of constant types comes from the need to be able to type every normal form with a constant type: for persistent variables, we use $\mathtt{v}$; for abstractions and variables that can be replaced by abstractions, we use $\mathtt{a}$; for $\mathtt{raise}_e()$ we use $\mathtt{r}$; and for neutral terms, we use $\mathtt{n}$. Of course, neutral terms will result from the interaction between normal terms, and this interaction is captured by the type system by the persistent rules for applications and handles in normal form.

The **type system** is given by the following rules:

$$\frac{}{x : [\sigma] \vdash^{(0,0,0,0)} x : \sigma}\ (\mathtt{ax}) \qquad \frac{\Gamma \vdash^{(b,h,p,s)} t : \delta}{\Gamma \setminus\!\setminus x \vdash^{(b,h,p,s)} \lambda x.t : \Gamma(x) \Rightarrow \delta}\ (\mathtt{abs}) \qquad \frac{}{\vdash^{(0,1,0,0)} \mathtt{raise}_e() : e \oplus [\,]}\ (\mathtt{raise}_c)$$

$$\frac{(\Gamma_i \vdash^{(b_i,h_i,p_i,s_i)} v : \sigma_i)_{i\in I}}{+_{i\in I}\ \Gamma_i \vdash^{(+_{i\in I} b_i, +_{i\in I} h_i, +_{i\in I} p_i, +_{i\in I} s_i)} v : [\sigma_i]_{i\in I}}\ (\mathtt{many}) \qquad \frac{\Gamma \vdash^{(b,h,p,s)} v : \mu}{\Gamma \vdash^{(b,h,p,s)} v : \star \oplus \mu}\ (\mathtt{unit})$$

$$\frac{\Gamma \vdash^{(b,h,p,s)} v : \mathcal{M} \Rightarrow \delta \qquad \Delta \vdash^{(b',h',p',s')} t : \star \oplus \mathcal{M}}{\Gamma + \Delta \vdash^{(1+b+b', h+h', p+p', s+s')} vt : \delta}\ (\mathtt{app}) \qquad \frac{\Gamma \vdash^{(b,h,p,s)} t : e \oplus \tau}{\Gamma \vdash^{(b,h,1+p,s)} vt : e \oplus \tau}\ (\mathtt{propag})$$

$$\frac{\Gamma \vdash^{(b,h,p,s)} t : e \oplus [\,] \qquad \Delta \vdash^{(b',h',p',s')} u : \delta}{\Gamma + \Delta \vdash^{(b+b', 1+h+h', p+p', s+s')} \mathtt{handle}_e(t,u) : \delta}\ (\mathtt{handle1})$$

$$\frac{\Gamma \vdash^{(b,h,p,s)} t : \delta \qquad [\delta = \star \oplus \mu] \text{ or } [\delta = e' \oplus \tau\ (e \neq e')]}{\Gamma \vdash^{(b,h,1+p,s)} \mathtt{handle}_e(t,u) : \delta}\ (\mathtt{handle2})$$

$$\frac{}{\vdash^{(0,0,0,0)} \lambda x.t : \mathtt{a}}\ (\mathtt{abs}_p) \qquad \frac{}{\vdash^{(0,0,0,1)} \mathtt{raise}_e() : e \oplus \mathtt{r}}\ (\mathtt{raise}_p)$$

$$\frac{\Gamma \vdash^{(b,h,p,s)} t : \star \oplus \overline{\mathtt{r}}}{x : [\mathtt{v}] + \Gamma \vdash^{(b,h,p,1+s)} xt : \star \oplus \mathtt{n}}\ (\mathtt{app1}_p) \qquad \frac{\Gamma \vdash^{(b,h,p,s)} t : \star \oplus \mathtt{n}}{\Gamma \vdash^{(b,h,p,1+s)} (\lambda x.u)t : \star \oplus \mathtt{n}}\ (\mathtt{app2}_p)$$

$$\frac{\Gamma \vdash^{(b,h,p,s)} t : \star \oplus \mathtt{n}}{\Gamma \vdash^{(b,h,p,1+s)} \mathtt{handle}_e(t,u) : \star \oplus \mathtt{n}}\ (\mathtt{handle}_p)$$

Rule (`propag`) is used to type vt, when t reduces to $\mathtt{raise}_e()$. In this case, the whole application will reduce to $\mathtt{raise}_e()$ (cf. reduction rule (`r`)), which is reflected in the typing rule. Rule (`handle2`) is used to type $\mathtt{handle}_e(t,u)$, when t reduces to $\mathtt{raise}_e()$ (cf. reduction rule (`s`)). For this case, it is important to clarify the following. Even though it is possible to type $\mathtt{raise}_e()$ with either rule ($\mathtt{raise}_c$) or ($\mathtt{raise}_p$), the former should be used to type $\mathtt{raise}_e()$ when it is going to be *consumed*, and the latter when it is going to *persist* in the normal form. We distinguish between these two cases by pivoting on the right-injection of the type: we use $[\,]$ when typing consumable exceptions, and `r` to type persistent exceptions. Rule (`handle1`) is used to type $\mathtt{handle}_e(t,u)$, when t reduces to a value or a $\mathtt{raise}_{e'}()$, such that $e \neq e'$. These two possibilities are reflected in the typing rule by using conditions: $\delta = \star \oplus \mu$ for t reducing to a value; and $\delta = e' \oplus \tau\ (e \neq e')$ for t reducing to $\mathtt{raise}_{e'}()$.

A multi-type $\mathcal{M}$ is **tight**, if all the types in $\mathcal{M}$ are tight. An environment Γ is **tight**, if it assigns tight multi-types to all variables. A type δ is **tight**, if it is of the form $\gamma \oplus \mathtt{tt}$. A type derivation Φ is **tight** if the type environment and the type of the conclusion are both tight.

Example 3.1 (Tight type derivation for Example 2.4)**.** Let Φ_0 be the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : [[\,] \Rightarrow (e \oplus [\,])] \vdash^{(0,0,0,0)} x : [\,] \Rightarrow (e \oplus [\,])}\ (\mathtt{ax}) \qquad \dfrac{}{\vdash^{(0,0,0,0)} y : [\,]}\ (\mathtt{many})}{x : [[\,] \Rightarrow (e \oplus [\,])] \vdash^{(1,0,0,0)} xy : e \oplus [\,]}\ (\mathtt{app}) \qquad \dfrac{\dfrac{}{x : [\mathtt{a}] \vdash^{(0,0,0,0)} x : \mathtt{a}}\ (\mathtt{ax})}{x : [\mathtt{a}] \vdash^{(0,0,0,0)} x : \star \oplus \mathtt{a}}\ (\mathtt{unit})}{x : [[\,] \Rightarrow (e \oplus [\,])], \mathtt{a}] \vdash^{(1,0,0,0)} \mathtt{handle}_e(xy, x) : \star \oplus \mathtt{a}}\ (\mathtt{handle1})}{\vdash^{(1,0,0,0)} \lambda x.\mathtt{handle}_e(xy, x) : [[\,] \Rightarrow (e' \oplus [\,]), \mathtt{a}] \Rightarrow (\star \oplus \mathtt{a})}\ (\mathtt{abs})$$

And Ψ_0 be the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash^{(0,1,0,0)} \mathtt{raise}_e() : e \oplus [\,]}\,(\mathtt{raise}_c)}{\vdash^{(0,1,0,0)} \lambda z.\mathtt{raise}_e() : [\,] \Rightarrow (e \oplus [\,])}\,(\mathtt{abs}) \qquad \dfrac{}{\vdash^{(0,0,0,0)} \lambda z.\mathtt{raise}_e() : \mathtt{a}}\,(\mathtt{abs}_p)}{\vdash^{(0,1,0,0)} \lambda z.\mathtt{raise}_e() : [[\,] \Rightarrow (e \oplus [\,]), \mathtt{a}]}\,(\mathtt{many})}{\vdash^{(0,1,0,0)} \lambda z.\mathtt{raise}_e() : \star \oplus [[\,] \Rightarrow (e \oplus [\,]), \mathtt{a}]}\,(\mathtt{unit})$$

Then we can build the following derivation:

$$\dfrac{\Phi_0 \qquad \Psi_0}{\vdash^{(2,1,0,0)} (\lambda x.\mathtt{handle}_e(xy, x))(\lambda z.\mathtt{raise}_{e'}()) : \star \oplus \mathtt{a}}\,(\mathtt{app})$$

The following theorem is the main result of this paper and reveals the expressiveness of our type system. For any tight type derivation Φ of a program t with counters b, h, p, and s, we are able to show that t evaluates to a normal form of size s in exactly b `b`-steps, h `h1`-steps, and p `r`/`h2`-steps, where h is the number of exception that were handled, and p is the number times exceptions were propagated. Therefore, the type system is not only sound, *i.e.*, able to guess the number of steps to normal form as well as the size of this normal form, but also complete with respect to the operational semantics.

Theorem 3.2 (Soundness and Completeness)**.**

- *(Soundness) If* $\Phi \triangleright \Gamma \vdash^{(b,h,p,s)} t : \delta$ *tight, then there exists* $u \in \mathtt{no}$*, such that* $t \twoheadrightarrow^{(b,h,p)} u$*, with* b `b`*-steps,* h `h1`*-steps,* p `r`/`h2`*-steps, and* $|u| = s$*.*
- *(Completeness) If* $t \twoheadrightarrow^{(b,h,p)} u$*, with* b `b`*-steps,* h `h1`*-steps, and* p `r`/`h2`*-steps, then there exists* $\Phi \triangleright \Gamma \vdash^{(b,h,p,|u|)} t : \delta$ *tight.*

Example 2.4 illustrates that $(\lambda x.\mathtt{handle}_e(xy, x))\ (\lambda z.\mathtt{raise}_e())$ evaluates to $\lambda z.\mathtt{raise}_e()$ in 2 `b`-steps and 1 `h1`-steps, and that $|\lambda z.\mathtt{raise}_e()| = 0$. Example 3.1 illustrates that the type system agrees with these measures, since there is a tight type derivation with counters $b = 2$, $h = 1$, $p = 0$, and $s = 0$.

4 Conclusion

In this work, we propose a tight type system for a calculus with exception handlers. Neither the calculus nor the type system can fit inside a general treatment as for example the one in [GVT23], where a monadic idempotent intersection type system is defined for a computational λ-calculus with *algebraic* operators à la Plotkin and Power [PP03]. In fact, exception handlers (like continuations) cannot be modelled as algebraic effects [PP13]. However, more general theories of algebraic effects have recently emerged, they combine algebraic operations with specific algebraic morphisms to manipulate program flow: the resulting theory is referred to as the theory of effects and handlers [BP15], [PP13] (and see [Pre15] for a gentle introduction). Exception handlers can be viewed as a preliminary step to a general treatment of tight types including arbitrary effects, instead of algebraic effects only.

References

[AGK20] Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds, fully developed. J. Funct. Program., 30(e14):1–101, 2020. doi:10.1017/S095679682000012X.

[AKR23] Sandra Alves, Delia Kesner, and Miguel Ramos. Quantitative global memory. In Logic, Language, Information, and Computation - 29th International Workshop, WoLLIC 2023, July 11-14, 2023, Proceedings, 2023. To appear.

[BP15] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. J. Log. Algebraic Methods Program., 84(1):108–123, 2015. doi:10.1016/j.jlamp.2014.02.001.

[Gir87] Jean-Yves Girard. Linear logic. Theor. Comput. Sci., 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

[GVT23] Francesco Gavazzo, Gabriele Vanoni, and Riccardo Treglia. On monadic intersection types, 2023. Submitted.

[KV20] Delia Kesner and Pierre Vial. Consuming and persistent types for classical logic. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020, pages 619–632. ACM, 2020. doi:10.1145/3373718.3394774.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In 4th Annual Symposium on Logic in Computer Science, (LICS), 1989, Pacific Grove, California, USA, pages 14–23. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39155.

[PP03] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. Appl. Categorical Struct., 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

[PP13] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. Log. Methods Comput. Sci., 9(4), 2013. doi:10.2168/LMCS-9(4:23)2013.

[Pre15] Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. In Dan R. Ghica, editor, The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015, volume 319 of Electronic Notes in Theoretical Computer Science, pages 19–35. Elsevier, 2015. doi:10.1016/j.entcs.2015.12.003.

[Wad95] Philip Wadler. Monads for functional programming. In 1st International Spring School on Advanced Functional Programming Techniques on Advanced Functional Programming, (AFP), 1995, Båstad, Sweden, Tutorial Text, volume 925 of Lecture Notes in Computer Science, pages 24–52. Springer, 1995. doi:10.1007/3-540-59451-5_2.