Exception Handling and Classical Logic

Steffen van Bakel
Department of Computing
Imperial College London
London, UK

ABSTRACT

We present λ^{try}, an extension of the λ-calculus with named exception handling, via try, throw and catch, and present a basic notion of type assignment expressing recoverable exception handling and show that it is sound. We define an interpretation for λ^{try} to Parigot's $\lambda\mu$-calculus, and show that reduction (both lazy and call by value) is preserved by the interpretation. We will show that also types assignable in the basic system are preserved by the interpretation.

We will then add a notion of total failure through halt that escapes applicative contexts without being caught by a handler, and show that we can interpret this in $\lambda\mu$ when adding *top* as destination. We will argue that introducing handlers for halt will break the relation with $\lambda\mu$.

We will conclude the paper by showing that it is possible to add handlers for program failure by introducing panic and dedicated handlers to λ^{try}. We will need to extend the language with a conditional construct that is typed in a non-traditional way, that cannot be expressed in $\lambda\mu$ or logic. This will allow both recoverable exceptions and total failure, dealt with by handlers; we will show a non-standard soundness result for this system.

KEYWORDS

exception handling, abort, classical logic, lambda calculus

ACM Reference Format:
Steffen van Bakel. 2019. Exception Handling and Classical Logic. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3354166.3354186

INTRODUCTION

In this paper we will investigate the relation between exception handling and Classical Logic [9, 20], but will tread a path different to that usually taken over the last 20 years or so. Where the normal approach is to start from Classical Logic and to seek computational content in proofs, here we will do the reverse: we will define a λ-calculus enriched with a primitive form of named (recoverable) exception handling, and investigate if its natural notions of type assignment can be represented in Parigot's $\lambda\mu$ [18, 19], a calculus that represents minimal classical logic [1].

https://doi.org/10.1145/3354166.3354186

We will also add non-recoverable exceptions; then the corresponding calculus is $\lambda\mu$-*top* [1], but only if we do not 'catch' these exceptions. When trying to add handlers for failing exceptions, the correspondence with $\lambda\mu$ or $\lambda\mu$-*top* breaks down, highlighting that type theories based on classical logic do not fully cover exception handling. To stress that point even further, we will enrich λ^{try} with a non-conventionally typed conditional structure, and the type constant fail that is reserved for failing computations; for this system, we will show that computations either run preserving the type, or run to a term that has type fail, so fail, as can be expected from the character of failing exceptions. This thereby constitutes a language for which the standard subject-reduction result does not hold, and which therefore cannot be represented in calculi based on classical logic.[1]

For a long time it has been thought that only intuitionistic logic had a computational meaning, given its strong relation with types in programming; this is known as the Curry-Howard isomorphism, and is most manifest in the simply typed lambda calculus. However, since it is not possible to comfortably express notions like control or context manipulation in the pure λ-calculus, it is clear that the λ-calculus by itself, although Turing complete, is perhaps not expressive enough. Most of these control features, such as direct returns, coroutines, or exception handling, usually exhibit a form of non-local exit, which, albeit specifiable (and therefore realisable) in the pure calculus, are not easily represented, and certainly not using meaningful types. As such, these additions required different formalisms for behaviour specification - e.g. translation to continuation passing style (CPS) or abstract machines.

That situation changed when Griffin [11] observed that the C-operator of Felleisen's λC-calculus [8] can be typed with $\neg\neg A \to A$ (or $((A \to \bot) \to \bot) \to A$), thus highlighting the first link between classical logic and sequential control in computer science. This led to work by Parigot, who introduced a candidate for describing continuations in the form of the $\lambda\mu$-calculus.

The study of the relation between exception handling and classical logic goes back a few decades. Fundamental work has been done by Nakano [15, 16], followed by Crolard [5], in building intuitive systems for analysing throw/catch structures in a functional context. Crolard's intuition with respect to the representation of throw and catch as terms in $\lambda\mu$ is an essential development, and is also used in Bierman's [3] interpretation into $\lambda\mu$ of de Groote's [13] calculus $\lambda^{\rightarrow}_{exn}$, and in a certain sense also by [17], albeit for call-by-value languages. However, in both approaches the argument of the throw-term is the actual exception handler, which is different from the usual perception of what the information encapsulated in a thrown exception should be.

[1] All such calculi are designed to satisfy preservation of provable statements under the operation of *cut-elimination*, which translates to the property of subject reduction on the level of the calculi.

Here we will present the λ^{try}-calculus, a λ-calculus extended with a try/throw/catch syntax which is more similar to the constructions found in common programming languages. In our view, shared by many in the literature, exceptions should exclusively only be thrown when reached during the execution of a program; we therefore accept the (almost) generic approach (an exception is that of [15]) and define *reduction strategies* that do not permit reduction inside an abstraction; unlike in other papers, here we will we consider both call-by-name (lazy) and call-by-value.

Rather than selecting the exception handler *through its type*, as is the common practice in languages like `java` [10], in λ^{try} the handlers are called *by name*, giving exception handling a more functional 'feel'. This calculus can be implemented in $\lambda\mu$ in that we will present an interpretation that preserves both lazy and call-by-value reduction in λ^{try}; as was the case in [4, 5, 17], the 'context erasing' capability of μ-reduction is used to model the functionality of throw.

To investigate if all natural notions of type assignment for this calculus can correspond to the one for $\lambda\mu$, we will present three variants of λ^{try}, with different notions of type assignment. The first comprises a 'basic' theory, based on the approach of recoverable exceptions currently used [7] for example in `java`; it assumes that all exception handlers return the same type as that of the main term in a try-construct, effectively hiding the occurrence of the exception and allowing for execution to continue normally even after an exception has been thrown. We will show that assignable types are preserved under CBN and CBV-reduction and under the interpretation into $\lambda\mu$.

The second notion of type assignment we will present represents 'failure'; we add the construct halt, which corresponds to an exception that cannot be caught so has no possibility of recovery. This induces a notion of type assignment, for which we will show soundness for both CBN and CBV. We will modify the interpretation into on mapping onto *$\lambda\mu$-top*, a variant of $\lambda\mu$ that represents full classical logic, and show that assignable types are preserved.

Since both these notions are presented for a small extension of the λ-calculus, the notions are sound but not really expressive: for example, both throw and halt can have all types. Although the type $\bot$ is used when mapping the calculus into *$\lambda\mu$-top*, it is not used for the calculus itself, so we cannot tell by the assignable types if a program will fail, an arguably desirable property.

It is fair to state that type assignment for exception handling that marks failing computations is only really relevant in the presence of the conditional construct, where, depending on the evaluation of the boolean expression, the program continues normally or raises an exception. We will therefore extend λ^{try} further, add a conditional construct together with term constants and their types, and add a handling mechanism to deal with occurrences of halt, so achieve both recoverable and fatal exceptions. Also for this extension we will show a soundness result, which states that a computation either runs preserving the type, or fails. The key difference for this system is that we have to allow for the conditional construct to be typed in a non-conventional way. A direct consequence of this choice is that no longer can we preserve assignable types under the interpretation into $\lambda\mu$ or *$\lambda\mu$-top*.

These result put into evidence that exception handling can be either recoverable or failing, characterised through assignable types, and that named exception handling is perfectly feasible in the context of functional programming. Moreover, type assignment systems for exception handling need not all be based on classical logic.

$$(Ax):\ \frac{}{\Gamma, x{:}A \vdash x : A}$$

$$(\to I):\ \frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ (x \notin \Gamma)$$

$$(\to E):\ \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

Figure 1: Curry type assignment system for the λ-calculus.

1 RELATED SYSTEMS

In this section we will revise some formal languages and their type assignment systems that are of interest to this paper. We revisit Curry's λ-calculus [2, 6], and Parigot's $\lambda\mu$ [18].

1.1 The λ-calculus

We quickly revise some basic notions for the λ-calculus, to better set the context of this paper.

Definition 1.1 (Lambda terms, call-by-name and call-by-value).

(1) *λ-terms* are defined by the grammar:

$$\begin{array}{rcll} M,N & ::= & V \mid MN & \\ V & ::= & x \mid \lambda x.M & (\textit{values}) \end{array}$$

(2) (One-step) β-reduction is defined using the β-rule

$$(\beta):\ (\lambda x.M)N \to M\{N/x\}$$

and evaluation contexts that are defined as terms with a single hole by:

$$\mathrm{C} ::= [\,] \mid \mathrm{C}M \mid M\mathrm{C} \mid \lambda x.\mathrm{C}$$

We write $\mathrm{C}[M]$ for the term obtained from the context C by replacing its hole $[\,]$ with M, allowing variables to be captured. One-step β reduction is defined as the compatible closure of the β-rule through:

$$(\beta):\ \mathrm{C}[(\lambda x.M)N] \ \to\ \mathrm{C}[M\{N/x\}]$$

for any evaluation context. We write $\to_\beta^*$ for the transitive closure of $\to_\beta$, and use that notation for all the notions of reduction we consider in this paper.

(3) *Call-by-name evaluation contexts* are defined through:

$$\mathrm{C_N} ::= [\,] \mid \mathrm{C_N}M$$

Call-by-name (CBN) reduction $\to_{\beta\mu}^{\mathrm{N}}$ (also known as *lazy* reduction) is defined through:

$$(\beta):\ \mathrm{C_N}[(\lambda x.M)N] \ \to\ \mathrm{C_N}[M\{N/x\}]$$

(4) Call-by-value *evaluation contexts* are defined through:

$$\mathrm{C_V} ::= [\,] \mid \mathrm{C_V}M \mid V\mathrm{C_V}$$

Call-by-value (CBV) reduction $\to_{\beta\mu}^{\mathrm{V}}$ is defined through:

$$(\beta_{\mathrm{V}}):\ \mathrm{C_V}[(\lambda x.M)V] \ \to\ \mathrm{C_V}[M\{V/x\}]$$

Curry type assignment for the λ-calculus is defined by:

Definition 1.2 (Curry type assignment for the λ-calculus).

(1) Let φ range over a countable (infinite) set of type-variables. The set of *Curry types* is defined by the grammar:

$$A, B \ ::= \ \varphi \mid A \rightarrow B$$

(2) A *context of variables* Γ is a partial mapping from term variables to types, denoted as a finite set of *statements* $x{:}A$, such that the *subjects* of the statements (x) are distinct. We write Γ_1, Γ_2 for the *compatible* union of Γ_1 and Γ_2 (if $x{:}A_1 \in \Gamma_1$ and $x{:}A_2 \in \Gamma_2$, then $A_1 = A_2$), and write $\Gamma, x{:}A$ for $\Gamma, \{x{:}A\}$, and $x \notin \Gamma$ if there exists no A such that $x{:}A \in \Gamma$.

(3) *Curry type assignment* is defined by the inference system in Fig. 1.

1.2 On adding exception handling to the λ-calculus

The main topic of this paper is to define an extension of the λ-calculus with exception handling, modelled through try, catch and throw, and investigate notions of type assignment for it and their relation to classical logic. Before coming to that, perhaps we should point out some of the inevitable limitations of extending the λ-calculus with exception handling.

• From the point of view of programming, throwing of exceptions from inside an abstraction, as modelled by the reduction rule

$$\lambda x.\mathsf{throw}\ \alpha\ N \ \ \rightarrow \ \ \mathsf{throw}\ \alpha\ N$$

should not be allowed.[2] One reason is that subject reduction will then fail (the variable x might appear in N; see Ex. 3.4), but, perhaps more importantly, it would correspond to letting a program raise an exception just because it occurs in a function definition, regardless of whether or not evaluation of the program has led to the exception.

• In call-by-value or call-by-name functional programming languages, reductions never take place underneath an abstraction, so exceptions defined inside a function are only ever thrown when the function has been called (a redex involving the abstraction has been contracted). This restriction seems to have been applied to almost all proposals for λ-calculi with control in the past (an exception is [15]).

• A common approach to typeing the throw action is to base its rule on the rule for $\bot$-elimination from Classical Logic [9],

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

(*ex falso quodlibet sequitur*). which allows any type to be assigned to the expression, as through the rule

$$(\mathsf{throw}) : \ \frac{\Gamma \vdash M : A \mid \Delta}{\Gamma \vdash \mathsf{throw}\ \alpha\ M : B \mid \alpha{:}A, \Delta}$$

This, however, is only ever useful in languages that have a conditional construct, when one of the two alternatives throws a recoverable exception whereas the other executes normally (see also Sect. 5). We do not know, *a priori*, if a boolean will evaluate to true or false, so if the exception handler is only thrown in the else-part, the type assignment system should be able to express both that the computation will continue normally (in case the boolean evaluates to true), or fails (in case the boolean evaluates to false).

In order to successfully type this with the normal type assignment rule for the conditional

$$(\mathsf{cond}) : \ \frac{\Gamma \vdash M : \mathsf{bool} \mid \Delta \quad \Gamma \vdash P : B \mid \Delta \quad \Gamma \vdash Q : B \mid \Delta}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q : B \mid \Delta}$$

we need to be able to 'warp' the type of the throw to B, for any B. Type assignment thereby then 'hides' the fact that an exception was thrown. This last point will be relevant in Sect. 5 where we present a notion of type assignment that allows for *failing* exceptions, for which this hiding feature is no longer present, and the type assignment can (in certain cases) predict failure of a program. When adding the conditional construct, as we will do in the final part of the paper, and allowing for both recoverable and failing exceptions, this apparent shortcoming disappears, and part of a program can fail without that affecting the type for the whole.

• Normal programming hygiene would demand that exceptions can only be thrown towards an existing and corresponding catch (in our case, the one with the right name). Our approach here, where we use a try-construct

$$\mathsf{try}\ M;\ \mathsf{catch}\ \mathsf{n}_1(x) = N_1;\ \ldots;\ \mathsf{catch}\ \mathsf{n}_n(x) = N_n$$

that contains a number of catch expressions that deal with the exceptions that might be thrown inside M, demands that the result of a normal execution, which would exit from the try-construct, cannot contain a throw towards one of the exception handlers inside the try-construct, but can only refer to exception handlers that are defined *outside* the try-construct. In fact, the names for the exception handlers are bound in the construct, and we do not want reduction 'to free' bound names or variables.

If this seems restrictive, dropping this restriction for names is easily dealt with using dynamic scoping, and involves checking if a handler for that name is also defined 'one level up', or assuming that all locally defined exception handlers are otherwise redefined on the outermost level where they generate an *undefined* message, with reduction rules like

$$\begin{array}{l}(\mathsf{try}\ V;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i})P \ \ \rightarrow \\ \qquad \mathsf{try} VP;\ \overrightarrow{\mathsf{catch}\ \mathsf{m}(x) = \mathsf{Error}(\text{"Message not understood"})} \\ \hfill (\overrightarrow{m_i} = fn(V))\end{array}$$

etc. Because dynamic scoping cannot be directly represented in $\lambda\mu$, we choose here to syntactically restrict the terms; this leads to more elegant and tractable solutions to the various theoretical results we achieve, where we can focus on the essential properties without overly complicating the system.

1.3 The calculus $\lambda\mu$

Parigot's $\lambda\mu$-calculus is a proof-term syntax for classical logic, expressed in Natural Deduction, defined as an extension of the Curry type assignment system for the λ-calculus. With $\lambda\mu$ Parigot created a multi-conclusion typing system which corresponds to classical logic; the derivable statements have the shape $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$, where A is the main conclusion of the statement, expressed as the *active* conclusion, and Δ contains the alternative conclusions; the left-hand context Γ represents the types of the free term variables of M. As with Implicative Intuitionistic Logic, the reduction rules

[2] This rule is implied in systems where throw is allowed to escape from *any* context, as in [15].

for the terms that represent the proofs correspond to proof contractions; the difference is that the reduction rules for the λ-calculus are the *logical* reductions, *i.e.* deal with the elimination of a type construct that has been introduced directly above. In addition to these, Parigot expresses also the *structural* rules that change the focus of a proof.

We now present the variant of $\lambda\mu$ we consider in this paper, as defined by Parigot in [19]:

Definition 1.3 (Syntax of $\lambda\mu$). The *$\lambda\mu$-terms* we consider are defined by the grammar:

$$\begin{array}{rcll} M,N & ::= & V \mid MN \mid \mu\alpha.[\beta]M & \\ V & ::= & x \mid \lambda x.M & (\textit{values}) \end{array}$$

Recognising both λ and μ as binders, the notion of free and bound names and variables is defined as usual, and we accept Barendregt's convention to keep free and bound names and variables distinct, using (silent) α-conversion whenever necessary.We write $x \in M$ ($\alpha \in M$) if x (α) occurs in M, either free of bound. As in Haskell [14], we will use '_' as a special name: when we write $\mu\_.[\alpha]M$, the name '_' does not occur (free) in $[\alpha]M$. We will call the pseudo-terms of the shape $[\alpha]M$ *commands*, written $\mathtt{C}$, and treat them as terms for reasons of brevity, whenever convenient.

In $\lambda\mu$, reduction of terms is expressed via implicit substitution, and as usual, $M\{N/x\}$ stands for the substitution of all occurrences of x in M by N.

We define two kinds of structural substitution: the first is the standard one, where $M\{N{\cdot}\gamma/\alpha\}$ stands for the term obtained from M in which every command of the form $[\alpha]P$ is replaced by $[\gamma]PN$ (γ is a fresh name). The second will be of use for call-by-value reduction; here $\{N{\cdot}\gamma/\alpha\}M$ stands for the term obtained from M in which every $[\alpha]P$ is replaced by $[\gamma]NP$.

They are formally defined by:

Definition 1.4 (Structural substitution). (1) *Right-structural substitution*, $M\{N{\cdot}\gamma/\alpha\}$, is defined inductively by:

$$\begin{array}{rcll} x\,\{N{\cdot}\gamma/\alpha\} & \triangleq & x & \\ (\lambda x.M)\,\{N{\cdot}\gamma/\alpha\} & \triangleq & \lambda x.(M\{N{\cdot}\gamma/\alpha\}) & \\ (M_1M_2)\,\{N{\cdot}\gamma/\alpha\} & \triangleq & M_1\{N{\cdot}\gamma/\alpha\}\,M_2\{N{\cdot}\gamma/\alpha\} & \\ [\alpha]M\,\{N{\cdot}\gamma/\alpha\} & \triangleq & [\gamma](M\{N{\cdot}\gamma/\alpha\}N) & \\ [\beta]M\,\{N{\cdot}\gamma/\alpha\} & \triangleq & [\beta](M\{N{\cdot}\gamma/\alpha\}) & (\beta \neq \alpha) \\ (\mu\delta.\mathtt{C})\,\{N{\cdot}\gamma/\alpha\} & \triangleq & \mu\delta.(\mathtt{C}\{N{\cdot}\gamma/\alpha\}) & \end{array}$$

(2) *Left-structural substitution*, $\{N{\cdot}\gamma/\alpha\}M$, is defined by:

$$\begin{array}{lcll} \{N{\cdot}\gamma/\alpha\}\,x & \triangleq & x & \\ \{N{\cdot}\gamma/\alpha\}\,(\lambda x.M) & \triangleq & \lambda x.(\{N{\cdot}\gamma/\alpha\}M) & \\ \{N{\cdot}\gamma/\alpha\}\,(M_1M_2) & \triangleq & \{N{\cdot}\gamma/\alpha\}M_1\;\{N{\cdot}\gamma/\alpha\}M_2 & \\ \{N{\cdot}\gamma/\alpha\}\,[\alpha]M & \triangleq & [\gamma]N(\{N{\cdot}\gamma/\alpha\}M) & \\ \{N{\cdot}\gamma/\alpha\}\,[\beta]M & \triangleq & [\beta]\{N{\cdot}\gamma/\alpha\}M & (\beta \neq \alpha) \\ \{N{\cdot}\gamma/\alpha\}\,\mu\delta.\mathtt{C} & \triangleq & \mu\delta.\{N{\cdot}\gamma/\alpha\}\mathtt{C} & \end{array}$$

[18] only defines the first variant of these notions of structural substitutions (so does not use the prefix 'right'); the two notions are defined together, but rather informally, using a notion for contexts in [17].

We have the following notions of reduction on $\lambda\mu$. For the third, call by value, different variants exists in the literature; we adopt the one from [17].

$$\begin{array}{rc} (Ax): & \dfrac{}{\Gamma, x{:}A \vdash x : A \mid \Delta} \\[2ex] (\mu): & \dfrac{\Gamma \vdash M : B \mid \alpha{:}A, \beta{:}B, \Delta}{\Gamma \vdash \mu\alpha.[\beta]M : A \mid \beta{:}B, \Delta}\;(\alpha \notin \Delta) \\[2ex] & \dfrac{\Gamma \vdash M : A \mid \alpha{:}A, \Delta}{\Gamma \vdash \mu\alpha.[\alpha]M : A \mid \Delta}\;(\alpha \notin \Delta) \\[2ex] (\to I): & \dfrac{\Gamma, x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A \to B \mid \Delta}\;(x \notin \Gamma) \\[2ex] (\to E): & \dfrac{\Gamma \vdash M : A \to B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta} \end{array}$$

Figure 2: Type assignment rules for $\lambda\mu$

Definition 1.5 ($\lambda\mu$ reduction). (1) The reduction rules of $\lambda\mu$ are:

$$\begin{array}{rrcll} \textit{logical}\,(\beta): & (\lambda x.M)N & \to & M\{N/x\} & \\ \textit{structural}\,(\mu): & (\mu\alpha.\mathtt{C})N & \to & \mu\gamma.\mathtt{C}\{N{\cdot}\gamma/\alpha\} & (\gamma\ \textit{fresh}) \\ \textit{erasing}\,(E): & \mu\alpha.[\alpha]M & \to & M & (\alpha \notin M) \\ \textit{renaming}\,(R): & [\beta]\mu\gamma.\mathtt{C} & \to & \mathtt{C}\{\beta/\gamma\} & \end{array}$$

Evaluation contexts are defined by:

$$\mathrm{C} \;::=\; [\,] \mid \mathrm{C}M \mid M\mathrm{C} \mid \lambda x.\mathrm{C} \mid \mu\alpha.[\beta]\mathrm{C}$$

(Free, unconstrained) reduction $\to_{\beta\mu}$ on $\lambda\mu$-terms is defined through $\mathrm{C}[M] \to^{\mathrm{N}}_{\beta\mu} \mathrm{C}[N]$ if $M \to N$ using either the β, μ, E, or R-reductions rule.

(2) *Call-by-name evaluation contexts* are defined as:

$$\mathrm{C_N} \;::=\; [\,] \mid \mathrm{C_N}M \mid \mu\alpha.[\beta]\mathrm{C_N}$$

Call-by-name reduction is defined through: $\mathrm{C_N}[M] \to^{\mathrm{N}}_{\beta\mu} \mathrm{C_N}[N]$ if $M \to N$ using either the β, μ, E, or R rule.

(3) *Call-by-value evaluation contexts* are defined through:

$$\mathrm{C_V} \;::=\; [\,] \mid \mathrm{C_V}M \mid V\mathrm{C_V} \mid \mu\alpha.[\beta]\mathrm{C_V}$$

Call-by-value is defined through: $\mathrm{C_V}[M] \to^{\mathrm{V}}_{\beta\mu} \mathrm{C_V}[N]$ if $M \to N$ using either μ, E, R, or:

$$\begin{array}{rrcll} (\beta_{\mathrm{V}}): & (\lambda x.M)V & \to^{\mathrm{N}}_{\beta\mu} & M\{V/x\} & \\ (\mu_{\mathrm{V}}): & V(\mu\alpha.\mathtt{C}) & \to^{\mathrm{N}}_{\beta\mu} & \mu\gamma.\{V{\cdot}\gamma/\alpha\}\,\mathtt{C} & (\gamma\ \textit{fresh}) \end{array}$$

(4) Call-by-name *applicative* contexts are defined as:

$$\mathrm{C^A_N} \;::=\; [\,] \mid \mathrm{C^A_N}M$$

whereas call-by-value applicative contexts are defined as:

$$\mathrm{C^A_V} \;::=\; [\,] \mid \mathrm{C^A_V}M \mid V\mathrm{C^A_V}$$

Remark that, for rule (μ_{V}), $\mu\alpha.[\beta]N$ is not a value. Also, unlike for the λ-calculus, call-by-value reduction is not a sub-reduction system of $\to_{\beta\mu}$: the rule (μ_{V}) (and left-structural substitution) are not part of $\to_{\beta\mu}$.

Notice that a term might be in either CBN or CBV-normal form (*i.e.* reduction has stopped), but not need be that for $\to_{\beta\mu}$.

Type assignment for $\lambda\mu$ is defined below; there is a *main*, or *active*, conclusion, labelled by a term, and the *alternative* conclusions are labelled by names α, β, *etc.*

Definition 1.6 (Typing rules for $\lambda\mu$). (1) Types and contexts are those of Def. 1.2.

(2) A *context of names* Δ is a partial mapping from *names* to types, denoted as a finite set of *statements* $\alpha{:}A$, such that the *subjects* of the statements (α) are distinct. Notions Δ_1, Δ_2, as well as $\Delta, \alpha{:}A$ and $\alpha \notin \Delta$ are defined as for Γ.

(3) The type assignment rules for $\lambda\mu$ are presented in Fig. 2; we will write $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$ for statements derivable in this system.

We can think of $[\alpha]M$ as storing the type of M amongst the alternative conclusions by giving it the name α.

Example 1.7. Take the term $\mu\alpha.[\alpha]\mu\beta.[\gamma]M$, such that M does not contain α or β, and $\alpha \neq \gamma$. Then by renaming,

$$\mu\alpha.[\alpha]\mu\beta.[\gamma]M \rightarrow_{\beta\mu} \mu\alpha.[\gamma]M\{\alpha/\beta\} = \mu\alpha.[\gamma]M$$

but also, by erasure, $\mu\alpha.[\alpha]\mu\beta.[\gamma]M \rightarrow_{\beta\mu} \mu\beta.[\gamma]M$. Notice that $\mu\alpha.[\gamma]M =_\alpha \mu\_.[\gamma]M =_\alpha \mu\beta.[\gamma]M$.

We can show that type assignment is closed under reduction for both call-by-name and call-by-value reduction. This result might itself be as expected, and is presented here mostly for completeness. First we show results for the three notions of substitution.

Lemma 1.8 (Substitution lemma). (1) *If* $\Gamma, x{:}B \vdash_{\lambda\mu} M : A \mid \Delta$ *and* $\Gamma \vdash_{\lambda\mu} L : B \mid \Delta$, *then* $\Gamma \vdash_{\lambda\mu} M\{L/x\} : A \mid \Delta$.

(2) *If* $\Gamma \vdash_{\lambda\mu} M : A \mid \alpha{:}B \rightarrow C, \Delta$ *and* $\Gamma \vdash_{\lambda\mu} L : B \mid \Delta$, *then* $\Gamma \vdash_{\lambda\mu} M\{L{\cdot}\gamma/\alpha\} : A \mid \gamma{:}C, \Delta$.

(3) *If* $\Gamma \vdash_{\lambda\mu} L : B \rightarrow C \mid \Delta$ *and* $\Gamma \vdash_{\lambda\mu} M : A \mid \alpha{:}B, \Delta$, *then* $\Gamma \vdash_{\lambda\mu} \{L{\cdot}\gamma/\alpha\}M : A \mid \gamma{:}C, \Delta$.

With this lemma we can now show:

Theorem 1.9. (1) *If* $M \rightarrow^{\text{N}}_{\beta\mu} N$, *and* $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$, *then* $\Gamma \vdash_{\lambda\mu} N : A \mid \Delta$.

(2) *If* $M \rightarrow^{\text{V}}_{\beta\mu} N$, *and* $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$, *then* $\Gamma \vdash_{\lambda\mu} N : A \mid \Delta$.

[1] also present a variant of $\lambda\mu$, called $\lambda\mu$-*top*, where *top* is a special name that cannot occur bound and denotes the top-level. Their motivation for this extension is: "*On the programming calculi side, the presence of the continuation top makes it possible to distinguish between aborting a computation and throwing to a continuation (as aborting corresponds to throwing to the special top-level continuation). This distinction can be used to develop more refined programming calculi for languages with control operators.*" We will follow this suggestion below, when we look to model aborting computations in λ^{try} in Sect. 4.

Definition 1.10 ($\lambda\mu$-top). (1) Terms of the $\lambda\mu$-*top*-calculus are defined as in Def. 1.3, extended with the case $\mu\alpha.[top]M$, where *top* is a name that cannot occur bound.

(2) The notion of type assignment for $\lambda\mu$-*top*, $\vdash_t$, is defined extending the set of types with the constant $\bot$ and the type assignment rules of Fig. 2, extended with the rule

$$(top):\ \frac{\Gamma \vdash M : \bot \mid \alpha{:}A, \Delta}{\Gamma \vdash \mu\alpha.[top]M : A \mid \Delta}$$

The appropriate variants of Lem. 1.8 and Thm. 1.9 can much in the same way be shown to hold for $\vdash_t$.

The following lemma is needed below when encoding throwing exceptions.

Lemma 1.11. (1) *The reduction rule* $\text{C}^{\text{A}}_{\text{N}}[\mu\_.[\beta]N] \rightarrow \mu\_.[\beta]N$ *is admissible in* $\rightarrow^{\text{N}}_{\beta\mu}$.

(2) *The reduction rule* $\text{C}^{\text{A}}_{\text{V}}[\mu\_.[\beta]N] \rightarrow \mu\_.[\beta]N$ *is admissible in* $\rightarrow^{\text{V}}_{\beta\mu}$.

Proof. (1) By induction on the structure of contexts:

($\text{C}^{\text{A}}_{\text{N}} = [\,]$): Immediate.

($\text{C}^{\text{A}}_{\text{N}} = \text{C}^{\text{A}\prime}_{\text{N}}M$): Notice that $\_ \neq \beta$ and that there is no sub-term called $\_$ in N; then

$$\begin{array}{rcl}\text{C}^{\text{A}\prime}_{\text{N}}[\mu\_.[\beta]N]M & \rightarrow (IH) & (\mu\_.[\beta]N)M \\ & \rightarrow^{\text{N}}_{\beta\mu} & \mu\gamma.[\beta]N\{M{\cdot}\gamma/\_\} \\ & = & \mu\gamma.[\beta]N \\ & =_\alpha & \mu\_.[\beta]N\end{array}$$

Notice that γ is fresh; since there is no sub-term called $\_$ in M, the structural substitution $\{M{\cdot}\gamma/\_\}$ has no effect, so, in particular, γ does not appear in $[\beta]M$.

(2) By induction on the structure of contexts. The first two cases are similar to above; the third one is:

($\text{C}^{\text{A}}_{\text{V}} = V\text{C}^{\text{A}\prime}_{\text{V}}$):

$$\begin{array}{rcl}V\text{C}^{\text{A}\prime}_{\text{V}}[\mu\_.[\beta]M] & \rightarrow (IH) & \\ V(\mu\_.[\beta]M) & \rightarrow^{\text{V}}_{\beta\mu} & \\ \mu\gamma.[\beta]\{V{\cdot}\gamma/\_\}M & = & \\ \mu\gamma.[\beta]M & =_\alpha & \mu\_.[\beta]M\end{array}$$ □

Notice that this result also holds for $\beta = top$.

1.4 On modelling the catch/throw mechanism in $\lambda\mu$

Throwing an exception involves an interruption of normal execution, and a jump out of the current context; those jumps can be modelled by rules that eradicate the context, like for example $\text{C}[\mathcal{A}(M)] \rightarrow M$ as used in $\lambda\mathcal{C}$ [8] (using the *abort* constructor $\mathcal{A}$), or similarly as in [11, 15].

The kind of contexts that can be aborted differ from paper to paper. The most common approach, as used in [4, 8] and here, is to allow aborting applicative contexts only, but, for example, [15] allows aborting executions inside abstractions as well. Allowing the latter kind of comes with obstacles, in that type assignment and in particular soundness (preservation of assignable types under reduction) becomes more difficult to achieve, since in general we cannot preserve types when aborting from an abstraction (see Ex. 3.4), which leaves that we can only safely abort from applicative contexts.

In terms of provable properties it is preferable to model eradication of applicative contexts 'one syntactic construction at the time', rather than use the $\lambda\mathcal{C}$-approach, which aborts entire contexts via $\text{C}[\mathcal{A}(M)] \rightarrow M$, where the context is assumed to be as large as possible. This is exactly what can be modelled in $\lambda\mu$ (using the result of Lem. 1.11), where the functionality of throwing an exception M to name n can be represented by $\mu\_.[\mathsf{n}]M$ (where n does not occur in M), i.e. a context switch that can be used to erase (only) an applicative context. We can now implement the functionality of 'escaping from the context' via the mechanism of consuming it via the reduction steps:

$$\begin{array}{rcl}(\mu\_.[\mathsf{n}]M)PQ & \rightarrow & (\mu\_.[\mathsf{n}]M)QR \ \rightarrow \\ & & (\mu\_.[\mathsf{n}]M)R \ \rightarrow \ \mu\_.[\mathsf{n}]M\end{array}$$

Notice that this will always leave the prefix $\mu\_.[\mathsf{n}]$, which therefore has to be removed through the encoding of the catch functionality.

We can achieve this using $\lambda\mu$'s renaming and erasing reduction steps: we model catching on name n through $\mu\mathsf{n}.[\alpha]M$:

$$\begin{array}{llll} \mu\mathsf{n}.[\alpha](\mu\_.[\mathsf{n}]M)PQ & \to^*_{\beta\mu} (1.11) & \mu\mathsf{n}.[\alpha]\mu\_.[\mathsf{n}]M & \to_{\beta\mu} (R) \\ & & \mu\mathsf{n}.[\mathsf{n}]M & \to_{\beta\mu} (E) \quad M \end{array}$$

However, this is not enough; we also want the catch-mechanism to disappear when computation terminates normally, as in

$$\mathsf{try}\ V;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i} \ \to\ V$$

which is then modelled through, as a last step $\mu\mathsf{n}.[\alpha]V \ \to_{\beta\mu}\ V$ but this is only possible when $\alpha = \mathsf{n}$.

In conclusion, throwing to the name n has to be modelled through $\mu\_.[\mathsf{n}]$, whereas catching on the name n has to be modelled through $\mu\mathsf{n}.[\mathsf{n}]$. This is the approach of all historic interpretations into $\lambda\mu$, as, for example, the one presented in [17].

2 THE CALCULUS λ^{try}

The calculus λ^{try} we will present in this section will use the C++/java-like syntax of try, throw, and catch, but will discern the exception handlers by name rather than by type. We will see the term 'catch n(x) = M' as an *exception handler named* n that can receive a parameter on x after which it runs M with the parameter taking the position of x in M, and 'throw n(N)' a call to the exception handler with name n, passing it the argument N.

Terms of λ^{try} are defined as follows:

Definition 2.1 (Syntax of λ^{try}). (1) The set of *pre-terms* of λ^{try} is defined by the grammar:

$$\begin{array}{rcll} \mathsf{Catch\_Block} & ::= & \mathsf{catch}\ \mathsf{m}(x) = M \mid \mathsf{Catch\_Block};\ \mathsf{catch}\ \mathsf{n}(x) = N \\ M,N & ::= & V \mid MN \mid \mathsf{try}\ M;\ \mathsf{Catch\_Block} \mid \mathsf{throw}\ \mathsf{n}(M) \\ V & ::= & x \mid \lambda x.M & (\textit{Values}) \end{array}$$

(2) We will call n in 'catch n$(x) = N$' a *declared name* and will write $\overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}$ for the catch-block

$$\mathsf{catch}\ \mathsf{n}_1(x) = N_1;\ \ldots;\ \mathsf{catch}\ \mathsf{n}_n(x) = N_n.$$

Since exceptions are called using their name, the order in which they appear in the catch-block is not important.

(3) The set of *terms* are pre-terms that satisfy the following restrictions:
   (a) In $\overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i}$ the names n_i do not occur in the exception handler M_j, for any $i,j \in \underline{n}$ (where $i \in \underline{n}$ stands for $i \in \{1,\ldots,n\}$), and all declared names $\mathsf{n}_1,\cdots,\mathsf{n}_n$ are distinct;
   (b) for each throw $\mathsf{n}_l(N)$ that occurs inside M in the term $\mathsf{try}\ M;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}$, none of the names n_i occur in N.

(4) We define the notion of *bound variables* and of *bound names* of M (respectively $bv(M)$ and $bn(M)$) as usual, extended with:

$$\begin{array}{rcl} bv(\mathsf{try}\ M;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}) & = & bv(M) \cup \\ && bv(N_1) \cup \cdots \cup bv(N_n) \cup \{x\} \\ bv(\mathsf{throw}\ \mathsf{n}(M)) & = & bv(M) \\ bn(\mathsf{try}\ M;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}) & = & \{\mathsf{n}_1,\ldots,\mathsf{n}_n\} \cup \\ && bn(M) \cup bn(N_1) \cup \cdots \cup bn(N_n) \\ bn(\mathsf{throw}\ \mathsf{n}(M)) & = & bn(M) \end{array}$$

where the occurrences of x in the terms N_i are bound by catch in the try-construct, and, by Barendregt's convention, x does not occur free in M. A variable or name is *free* in M if it occurs in M and is not bound; we write $fv(M)$ for the set of free variables in M, and $fn(M)$ for its free names.

We accept Barendregt's convention, so all free and bound variables and names are distinct, using renaming of bound variables or names (α-conversion) when necessary.

To control the throwing of exceptions, we define a notion of *call by name* (*lazy*) and *call by value* reduction; these define an evaluation strategy, where an exception is only ever thrown when needed to continue reduction.

Definition 2.2 (λ^{try}-reduction). (1) The notion of *call-by-name reduction* $\to^{\mathrm{N}}_{\mathrm{TRY}}$ on λ^{try} is defined as an extension of lazy reduction on λ-terms. The main reduction rules are:

$$\begin{array}{rlcl} (\beta): & (\lambda x.M)N & \to & M\{N/x\} \\ (\mathsf{throw}): & (\mathsf{throw}\ \mathsf{n}(N))M & \to & \mathsf{throw}\ \mathsf{n}(N) \\ (\mathsf{try\text{-}throw}): & \multicolumn{3}{l}{\mathsf{try}\ \mathsf{throw}\ \mathsf{n}_l(N);\ \mathsf{Catch\_Block};\ \mathsf{catch}\ \mathsf{n}_l(x) = M_l} \\ & & \to & M_l\{N/x\} \\ (\mathsf{try\text{-}normal}): & \mathsf{try}\ N;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i} & \to & N \quad (\overrightarrow{\mathsf{n}_i \notin N}) \end{array}$$

Call-by-name *applicative* contexts are defined as:

$$\mathrm{C}^{\mathrm{A}}_{\mathrm{N}} \ ::=\ [\,] \mid \mathrm{C}^{\mathrm{A}}_{\mathrm{N}}M \mid \mathsf{try}\ \mathrm{C}^{\mathrm{A}}_{\mathrm{N}};\ \mathsf{Catch\_Block}$$

(2) The notion of *call-by-value reduction* $\to^{\mathrm{V}}_{\mathrm{TRY}}$ on λ^{try} is defined using the reduction rules from call-by-name, with the exception of (β) which gets replaced by:

$$(\beta_{\mathrm{V}}):\quad (\lambda x.M)V \ \to\ M\{V/x\}$$

It adds the rule:

$$(\mathsf{throw}_{\mathrm{V}}):\ V(\mathsf{throw}\ \mathsf{n}(N)) \ \to\ \mathsf{throw}\ \mathsf{n}(N)$$

Call-by-value applicative contexts are defined as:

$$\mathrm{C}^{\mathrm{A}}_{\mathrm{V}} \ ::=\ [\,] \mid \mathrm{C}^{\mathrm{A}}_{\mathrm{V}}M \mid V\mathrm{C}^{\mathrm{A}}_{\mathrm{V}} \mid \mathsf{try}\ \mathrm{C}^{\mathrm{A}}_{\mathrm{V}};\ \mathsf{Catch\_Block}$$

Notice that, as in all (call by name, or call by value) functional languages, reduction does not allow for the evaluation of the body of an abstraction; this implies that throws inside the body are not 'triggered' until at least the surrounding abstraction has disappeared as the result of the contraction of a redex. If execution inside a try-block leads to a term N that does not contain throws to the declared names, then the result of the try-block is just that N; it is not necessarily the case that reduction of N has terminated.

We will now define an interpretation of λ^{try}-terms into $\lambda\mu$, using the approach we discussed above. Notice that, by the very nature of $\lambda\mu$, when encoding throw using a context switch, the body of the throw is not the information that something has gone wrong that gets passed to the exception handler, but in fact the *entire* exception handler. This implies that, when dealing with the term 'try M; $\overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}$', we need to bring the exception handlers catch n$(x) = N$ *inside* the interpretation of M; this is done using substitution,[3] introducing variables c_{n_i} that are placed in front of the argument that is passed to the exception handler in throw $\mathsf{n}_i(M)$.

[3]A perhaps more elegant approach is to encode a try-block using a redex, rather than term substitution, but that implies that we can no longer model lazy (cbv) reduction in λ^{try} by lazy (cbv) reduction in $\lambda\mu$, in particular when modelling the step $M \to N \ \Rightarrow\ \mathsf{try}\ M;\ \mathsf{Catch\_Block} \to \mathsf{try}\ N;\ \mathsf{Catch\_Block}$.

Definition 2.3 (Interpretation of λ^{try} into $\lambda\mu$). We extend the set of names in $\lambda\mu$ with n, m, …, and define the interpretation of terms in λ^{try} into $\lambda\mu$-terms as follows:

$$\begin{array}{lcl}
\ulcorner x \rrbracket_{\lambda\mu} & \triangleq & x \\
\ulcorner \lambda x.M \rrbracket_{\lambda\mu} & \triangleq & \lambda x.\ulcorner M \rrbracket_{\lambda\mu} \\
\ulcorner MN \rrbracket_{\lambda\mu} & \triangleq & \ulcorner M \rrbracket_{\lambda\mu} \ulcorner N \rrbracket_{\lambda\mu} \\
\ulcorner \mathsf{throw}\,\mathsf{n}(M) \rrbracket_{\lambda\mu} & \triangleq & \mu\_.[\mathsf{n}]\,c_{\mathsf{n}}\ulcorner M \rrbracket_{\lambda\mu} \\
\ulcorner \mathsf{try}\ M;\ \mathsf{catch}\ \mathsf{n}(x) = N \rrbracket_{\lambda\mu} & \triangleq & (\mu\mathsf{n}.[\mathsf{n}]\ulcorner M \rrbracket_{\lambda\mu})\,\{\lambda x.\ulcorner N \rrbracket_{\lambda\mu}/c_{\mathsf{n}}\} \\
\multicolumn{3}{l}{\ulcorner \mathsf{try}\ M; \mathsf{Catch\_Block};\ \mathsf{catch}\ \mathsf{n}(x) = N \rrbracket_{\lambda\mu} \ \triangleq} \\
\multicolumn{3}{r}{(\mu\mathsf{n}.[\mathsf{n}]\ulcorner M; \mathsf{Catch\_Block} \rrbracket_{\lambda\mu})\,\{\lambda x.\ulcorner N \rrbracket_{\lambda\mu}/c_{\mathsf{n}}\}}
\end{array}$$

Remark 2.4. Although many names can be used in a λ^{try}-term, when interpreting into $\lambda\mu$ all collapse onto the outermost one. To illustrate this, take the term

$$\mathsf{try}\ M(\mathsf{throw}\,\mathsf{m}(N))(\mathsf{throw}\,\mathsf{n}(L));\ \mathsf{catch}\ \mathsf{n}(x) = P;\ \mathsf{catch}\ \mathsf{m}(x) = Q$$

The interpretation of this term is

$$\begin{array}{l}
(\mu\mathsf{m}.[\mathsf{m}](\mu\mathsf{n}.[\mathsf{n}]\ulcorner M \rrbracket(\mu\_.[\mathsf{m}]\,c_{\mathsf{m}}\ulcorner N \rrbracket)(\mu\_.[\mathsf{n}]\,c_{\mathsf{n}}\ulcorner L \rrbracket)) \\
\qquad\qquad \{\lambda x.\ulcorner P \rrbracket_{\lambda\mu}/c_{\mathsf{n}}\})\,\{\lambda x.\ulcorner Q \rrbracket_{\lambda\mu}/c_{\mathsf{m}}\} \quad = \\
\mu\mathsf{m}.[\mathsf{m}]\ \mu\mathsf{n}.[\mathsf{n}]\ \ulcorner M \rrbracket_{\lambda\mu}(\mu\_.[\mathsf{m}](\lambda x.\ulcorner Q \rrbracket_{\lambda\mu})\ulcorner N \rrbracket_{\lambda\mu}) \\
\qquad\qquad (\mu\_.[\mathsf{n}](\lambda x.\ulcorner P \rrbracket_{\lambda\mu})\ulcorner L \rrbracket_{\lambda\mu}) \ \rightarrow_{\beta\mu} \ (R) \\
\mu\mathsf{m}.[\mathsf{m}]\ \ulcorner M \rrbracket_{\lambda\mu}(\mu\_.[\mathsf{m}](\lambda x.\ulcorner Q \rrbracket_{\lambda\mu})\ulcorner N \rrbracket_{\lambda\mu}) \\
\qquad\qquad (\mu\_.[\mathsf{m}](\lambda x.\ulcorner P \rrbracket_{\lambda\mu})\ulcorner L \rrbracket_{\lambda\mu})
\end{array}$$

We will show that both reduction and assignable types (under the basic system, see Sect. 3) are preserved under this interpretation. First we show that term-substitution is preserved under the interpretation.

Lemma 2.5 ($\ulcorner\cdot\rrbracket_{\lambda\mu}$ preserves term substitution).

$$\ulcorner M \rrbracket_{\lambda\mu}\{\ulcorner N \rrbracket_{\lambda\mu}/x\} = \ulcorner M\{N/x\} \rrbracket_{\lambda\mu}.$$

We can show that CBN-reduction and CBV-reduction on λ^{try}-terms is preserved under the interpretation:

Theorem 2.6 (Soundness of $\ulcorner\cdot\rrbracket_{\lambda\mu}$ with respect to $\rightarrow^{\mathrm{N}}_{\mathrm{TRY}}$). *If $P \rightarrow^{\mathrm{N}}_{\mathrm{TRY}} Q$, then $\ulcorner P \rrbracket_{\lambda\mu} \rightarrow^{\mathrm{N}*}_{\beta\mu} \ulcorner Q \rrbracket_{\lambda\mu}$.*

So it seems that interpreting into $\lambda\mu$ is the natural thing to do.

Similarly, we can verify that the interpretation respects call by value reduction $\rightarrow^{\mathrm{V}}_{\mathrm{TRY}}$.

Theorem 2.7 (Soundness of $\ulcorner\cdot\rrbracket_{\lambda\mu}$ with respect to $\rightarrow^{\mathrm{V}}_{\mathrm{TRY}}$). *If $P \rightarrow^{\mathrm{V}}_{\mathrm{TRY}} Q$, then $\ulcorner P \rrbracket_{\lambda\mu} \rightarrow^{\mathrm{V}*}_{\beta\mu} \ulcorner Q \rrbracket_{\lambda\mu}$.*

The only non-β-reduction steps for the $\lambda\mu$-calculus used in these two encoding results are *renaming*, *erasing*, and μ (or μ_{V}) towards _, *i.e.* a non-occurring name.

3 BASIC TYPE ASSIGNMENT

In this section we will define a notion of basic type assignment for terms in λ^{try} in the traditional way; in particular, in rule (try), we will demand that the type of the main term is exactly that returned by all exception handlers.

Definition 3.1 (Basic type assignment for λ^{try}). (1) Types and contexts of variables Γ and names Δ are those of Def. 1.6.

(2) Basic type assignment for terms in λ^{try} is defined through the inference system in Fig. 3. We write $\Gamma \vdash_{\mathrm{B}} M : A \mid \Delta$ for statements derivable using these rules.

$$(Ax):\ \frac{}{\Gamma, x{:}A \vdash x : A \mid \Delta}$$

$$(\rightarrow I):\ \frac{\Gamma, x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A \rightarrow B \mid \Delta}\ (x \notin \Gamma)$$

$$(\rightarrow E):\ \frac{\Gamma \vdash M : A \rightarrow B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta}$$

$$(\mathsf{throw}):\ \frac{\Gamma \vdash N : A \mid \mathsf{n}{:}A \rightarrow B, \Delta}{\Gamma \vdash \mathsf{throw}\,\mathsf{n}(N) : C \mid \mathsf{n}{:}A \rightarrow B, \Delta}$$

$$(\mathsf{try}):\ \frac{\Gamma \vdash M : B \mid \overrightarrow{\mathsf{n}_i{:}A_i{\rightarrow}B}, \Delta \quad \Gamma, x{:}A_i \vdash N_i : B \mid \Delta\ (\forall i \in \underline{n})}{\Gamma \vdash \mathsf{try}\ M;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i} : B \mid \Delta}\ (\overrightarrow{\mathsf{n}_i \notin \Delta})$$

Figure 3: Basic type assignment for λ^{try}

Notice that our (throw) rule allows to derive *any* type for the term throw n(N), but provided there is an exception handler with name n capable of accepting arguments of the type of N, as represented by the context of names.

Explaning rule (try), notice that, if we have derivations for

$$\frac{\boxed{}}{\Gamma \vdash M : C \mid \overrightarrow{\mathsf{n}_i{:}A_i \rightarrow B_i}, \Delta} \qquad \text{and} \qquad \frac{\boxed{}}{\Gamma, x{:}A_i \vdash N_i : B_i \mid \Delta\ (\forall i \in \underline{n})}$$

then we cannot predict, *a priori*, if running M to normal form M' will throw an exception or not. If it does not, then running the term try M; $\overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}$ will result in M' (assuming M' is free of throws) and to achieve subject reduction, M' should be of type C. If it does, running M will produce throw n(L) and (assuming $\mathsf{n} = \mathsf{n}_l \in \overrightarrow{\mathsf{n}_i}$), try M; $\overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}$ will run to $N_i\{L/x\}$, which has type B_i. So in order to achieve a subject reduction result also for this case, there is no choice but to demand that $C = B_1 = \cdots = B_n$.

We can show:

Lemma 3.2 (Substitution lemma for $\vdash_{\mathrm{B}}$). *If $\Gamma, x{:}C \vdash_{\mathrm{B}} M : A \mid \Delta$ and $\Gamma \vdash_{\mathrm{B}} N : C \mid \Delta$, then $\Gamma \vdash_{\mathrm{B}} M\{N/x\} : A \mid \Delta$.*

It is relatively straightforward to show that this notion of type assignment is closed under CBN and CBV-reduction:

Theorem 3.3 (Subject reduction for $\vdash_{\mathrm{B}}$). (1) *If $\Gamma \vdash_{\mathrm{B}} P : A \mid \Delta$ and $P \rightarrow^{\mathrm{N}}_{\mathrm{TRY}} Q$, then $\Gamma \vdash_{\mathrm{B}} Q : A \mid \Delta$.*
(2) *If $\Gamma \vdash_{\mathrm{B}} P : B \mid \Delta$ and $P \rightarrow^{\mathrm{V}}_{\mathrm{TRY}} Q$, then $\Gamma \vdash_{\mathrm{B}} Q : B \mid \Delta$.*

Proof. (1) By induction on the definition of $\rightarrow^{\mathrm{N}}_{\mathrm{TRY}}$; we only show the interesting parts.

(β): Standard, using Lemma 3.2.

(throw): Then $\Delta = \mathsf{n}{:}A \rightarrow C, \Delta'$, $P = (\mathsf{throw}\,\mathsf{n}(N))\,M \rightarrow \mathsf{throw}\,\mathsf{n}(N) = Q$; the derivation for P is constructed as:

$$\frac{\dfrac{\dfrac{\boxed{\ \mathcal{D}\ }}{\Gamma \vdash N : A \mid \mathsf{n}{:}A \rightarrow C, \Delta'}}{\Gamma \vdash \mathsf{throw}\,\mathsf{n}(N) : D \rightarrow B \mid \mathsf{n}{:}A \rightarrow C, \Delta'}\,(\mathsf{throw}) \qquad \dfrac{\boxed{}}{\Gamma \vdash M : D \mid \mathsf{n}{:}A \rightarrow C, \Delta'} \atop \vdots}{\Gamma \vdash (\mathsf{throw}\,\mathsf{n}(N))\,M : B \mid \mathsf{n}{:}A \rightarrow C, \Delta'}\,(\rightarrow E)$$

We can construct the derivation for Q:[4]

[4]Notice that throw n (N) changes type; this corresponds to a feature of reduction in $\lambda\mu$, where in some presentations the structural rule is written as (using the notation of Definition 1.5) $(\mu\alpha.[\beta]M)N \rightarrow \mu\alpha.([\beta]M\{N{\cdot}\alpha/\alpha\})$; before the reduction, α has type $A \rightarrow B$, say, and after it has type B.

$$\dfrac{\begin{array}{c}\boxed{\mathcal{D}}\\ \Gamma \vdash N : A \mid \mathsf{n}{:}A\to C,\Delta'\end{array}}{\Gamma\vdash \mathsf{throw}\,\mathsf{n}(N) : B \mid \mathsf{n}{:}A\to C,\Delta'}\ (\mathsf{throw})$$

(try-throw): Then $P = \mathsf{try}\ \mathsf{throw}\,\mathsf{n}_l(M);\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i} \to N_l\{M/x\} = Q$; the derivation for P is constructed as follows:

$$\dfrac{\dfrac{\begin{array}{c}\boxed{}\\ \Gamma\vdash M:A_l \mid \overrightarrow{\mathsf{n}_i{:}A_i\to B},\Delta\end{array}}{\Gamma\vdash \mathsf{throw}\,\mathsf{n}_l(M):B\mid \overrightarrow{\mathsf{n}_i{:}A_i\to B},\Delta}\ (\mathsf{throw}) \qquad \begin{array}{c}\boxed{}\\ \Gamma,x{:}A_i\vdash N_i:B\mid\Delta\\ \vdots\ (\forall i\in\underline{n})\end{array}}{\Gamma\vdash \mathsf{try}\ \mathsf{throw}\,\mathsf{n}_l(M);\ \mathsf{catch}\ \mathsf{n}(x)=N:B\mid\Delta}\ (\mathsf{try})$$

In particular, we have derivations for both $\Gamma \vdash_{\mathrm{B}} M : A_l \mid \overrightarrow{\mathsf{n}_i{:}A_i \to C},\Delta$ and $\Gamma,x{:}A_l \vdash_{\mathrm{B}} N_l : B \mid \Delta$. By the definition of λ^{try}-terms, we know that $\mathsf{n}_i \notin \mathit{fn}(M)$, for all $i\in\underline{n}$, so by thinning we can remove $\overrightarrow{\mathsf{n}_i{:}A_i\to B}$ from the co-context for the first to obtain $\Gamma\vdash_{\mathrm{B}} M:A_l \mid \Delta$. Then, by Lem. 3.2, we obtain $\Gamma\vdash_{\mathrm{B}} N_l\{M/x\} : B\mid\Delta$.

(try-normal): Then $P = \mathsf{try}\ Q;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i} \to Q$, and $\overrightarrow{\mathsf{n}_i} \notin Q$; the derivation for P is constructed as follows:

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma\vdash Q:B\mid\overrightarrow{\mathsf{n}_i{:}A_i\to B},\Delta\end{array} \qquad \begin{array}{c}\boxed{}\\ \Gamma,x{:}A_i\vdash N_i:B\mid\Delta\ (\forall i\in\underline{n})\end{array}}{\Gamma\vdash\mathsf{try}\ Q;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x)=N_i}:B\mid\Delta}\ (\mathsf{try})$$

In particular, we have $\Gamma\vdash_{\mathrm{B}} Q:B\mid\overrightarrow{\mathsf{n}_i{:}A_i\to B},\Delta$; as above we can remove $\overrightarrow{\mathsf{n}_i{:}A_i\to B}$ from the co-context to obtain $\Gamma\vdash_{\mathrm{B}} Q:B\mid\Delta$.

(2) The proof is much like that for the previous part, but with the addition of:

(throw$_{\mathrm{V}}$): Then $\Delta = \mathsf{n}{:}A\to C,\Delta'$, $P = V(\mathsf{throw}\,\mathsf{n}(N)) \to \mathsf{throw}\,\mathsf{n}(N) = Q$; the derivation for P is constructed as:

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma\vdash V:E\to F\mid\mathsf{n}{:}A\to C,\Delta'\end{array} \qquad \dfrac{\begin{array}{c}\boxed{\mathcal{D}}\\ \Gamma\vdash N:A\mid\mathsf{n}{:}A\to C,\Delta'\end{array}}{\Gamma\vdash\mathsf{throw}\,\mathsf{n}(N):E\mid\mathsf{n}{:}A\to C,\Delta'}\ (\mathsf{throw})}{\Gamma\vdash V(\mathsf{throw}\,\mathsf{n}(N)):F\mid\mathsf{n}{:}A\to C,\Delta'}\ (\to E)$$

We can construct the derivation for Q:

$$\dfrac{\begin{array}{c}\boxed{\mathcal{D}}\\ \Gamma \vdash N : A \mid \mathsf{n}{:}A\to C,\Delta'\end{array}}{\Gamma\vdash \mathsf{throw}\,\mathsf{n}(N) : F \mid \mathsf{n}{:}A\to C,\Delta'}\ (\mathsf{throw})$$

□

Although restricting throwing an exception to applicative contexts might seem too limiting, it is in fact not possible to extend it to full reduction whilst preserving soundness.

Example 3.4. Assume we would have tried to model throwing exceptions from inside an abstraction as well, by adding the rule:

$$(\mathsf{throw\text{-}abstr}):\quad \lambda x.\mathsf{throw}\,\mathsf{n}(N) \;\to\; \mathsf{throw}\,\mathsf{n}(N)$$

Apart from that this is undesirable within programming languages (it would correspond to throwing an exception simply because it occurs in a function definition), or the fact that we cannot model this reduction in pure $\lambda\mu$, also subject reduction would fail instantly. Suppose we can derive

$$\dfrac{\dfrac{\begin{array}{c}\boxed{}\\ \Gamma,x{:}A\vdash N:D\mid\mathsf{n}{:}D\to C,\Delta\end{array}}{\Gamma,x{:}A\vdash\mathsf{throw}\,\mathsf{n}(N):B\mid\mathsf{n}{:}D\to C,\Delta}\ (\mathsf{throw})}{\Gamma\vdash\lambda x.\mathsf{throw}\,\mathsf{n}(N):A\to B\mid\mathsf{n}{:}D\to C,\Delta}\ (\to I)$$

We can construct

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma,x{:}A\vdash N:D\mid\mathsf{n}{:}D\to C,\Delta\end{array}}{\Gamma,x{:}A\vdash\mathsf{throw}\,\mathsf{n}(N):A\to B\mid\mathsf{n}{:}D\to C,\Delta}\ (\mathsf{throw})$$

but cannot, in general, derive $\Gamma\vdash_{\mathrm{B}}\mathsf{throw}\,\mathsf{n}(N):A\to B\mid\mathsf{n}{:}D\to C,\Delta$: notice that x might be free in N, so then would need a type in any derivation for N. This problem was observed by [15, 16], who solved it by not allowing an abstraction to be typeable if the bound variable occurs in a thrown term, and avoided by many others who do not allow throwing an exception from within an abstraction, as we do here.

We will now show that our encoding into $\lambda\mu$ preserves types assignable in the basic system:

Theorem 3.5 (Preservation of assignable types). *If* $\Gamma\vdash_{\mathrm{B}} M : B\mid\overrightarrow{\mathsf{n}_i{:}A_i\to C_i}$, *then* $\Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i}\vdash_{\lambda\mu}\llbracket M\rrbracket_{\lambda\mu}:B\mid\overrightarrow{\mathsf{n}_i{:}C_i}$.

Proof. By induction on the definition of $\vdash_{\mathrm{B}}$; we will only show the interesting cases.

(throw): Then the derivation looks like

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma\vdash M:A_i\mid\overrightarrow{\mathsf{n}_i{:}A_i\to C_i}\end{array}}{\Gamma\vdash\mathsf{throw}\,\mathsf{n}_l(M):B\mid\overrightarrow{\mathsf{n}_i{:}A_i\to C_i}}\ (\mathsf{throw})$$

By induction we have $\Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i}\vdash\llbracket M\rrbracket_{\lambda\mu}:A_i\mid\overrightarrow{\mathsf{n}_i{:}C_i},\Delta$, and we can construct:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\overline{\Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i}\vdash c_{\mathsf{n}_l}:A\to C_i\mid\overrightarrow{\mathsf{n}_i{:}C_i}}\ (Ax)\\ \vdots\end{array} \qquad \begin{array}{c}\boxed{}\\ \Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i}\vdash\llbracket M\rrbracket_{\lambda\mu}:A_i\mid\overrightarrow{\mathsf{n}_i{:}C_i}\end{array}}{\Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i}\vdash c_{\mathsf{n}_l}\llbracket M\rrbracket_{\lambda\mu}:C_i\mid\overrightarrow{\mathsf{n}_i{:}C_i}}\ (\to E)}{\Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i}\vdash c_{\mathsf{n}_l}\llbracket M\rrbracket_{\lambda\mu}:C_i\mid \_{:}B,\overrightarrow{\mathsf{n}_i{:}C_i}}\ (Wk)}{\Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i}\vdash\mu\_.[\mathsf{n}_l]c_{\mathsf{n}_l}\llbracket M\rrbracket_{\lambda\mu}:B\mid\overrightarrow{\mathsf{n}_i{:}C_i}}\ (\mu)$$

and $\llbracket\mathsf{throw}\,\mathsf{n}_l(M)\rrbracket_{\lambda\mu} = \mu\_.[\mathsf{n}_l]c_{\mathsf{n}_l}\llbracket M\rrbracket_{\lambda\mu}$. Notice that the weakening step is correct, in that the names n and $\_$ do not occur (free) in $\llbracket M\rrbracket_{\lambda\mu}$, so (perhaps using thinning) can be assumed to not occur in Δ.

(try): Then the derivation ends like (assuming there are m exception handlers defined):

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma\vdash M:B\mid\overrightarrow{\mathsf{m}_j{:}D_j\to B},\overrightarrow{\mathsf{n}_i{:}A_i\to C_i}\\ \vdots\end{array} \qquad \begin{array}{c}\boxed{}\\ \Gamma,x{:}D_j\vdash N_j:B\mid\overrightarrow{\mathsf{n}_i{:}A_i\to C_i}\ (\forall j\in\underline{m})\end{array}}{\Gamma\vdash\mathsf{try}\ M;\ \mathsf{catch}\ \mathsf{m}_i(x)=N_i:B\mid\overrightarrow{\mathsf{n}_i{:}A_i\to C_i}}\ (\mathsf{try})$$

We can now construct derivations for the two alternatives of the interpretation of a try-expression; for clarity, we only present the second, the first is almost identical.

So the case we deal with here is:

$$\begin{array}{rl}\llbracket\mathsf{try}\ M;\ \mathsf{Catch\_Block};\ \mathsf{catch}\ \mathsf{m}(x) = N\rrbracket_{\lambda\mu} & \triangleq\\ & (\mu\mathsf{m}.[\mathsf{m}]\llbracket\mathsf{try}\ M;\ \mathsf{Catch\_Block}\rrbracket_{\lambda\mu})\,\{\lambda x.\llbracket N\rrbracket_{\lambda\mu}/c_{\mathsf{m}}\}\end{array}$$

Let $M' = \mathsf{try}\,M;\ \mathsf{Catch\_Block}$; then $\Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i},c_{\mathsf{m}}{:}D\to B\vdash\llbracket M'\rrbracket_{\lambda\mu}:B\mid\mathsf{m}{:}B,\overrightarrow{\mathsf{n}_i{:}C_i}$ and $\Gamma,\overrightarrow{c_{\mathsf{n}_i}{:}A_i\to C_i},x{:}D\vdash\llbracket N\rrbracket_{\lambda\mu}:B\mid\overrightarrow{\mathsf{n}_i{:}C_i}$ follow by induction. We can construct:

$$\dfrac{\dfrac{\dfrac{\boxed{}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i}, x{:}D \vdash \ulcorner L \rrbracket_{\lambda\mu} : B \mid \overrightarrow{\mathsf{n}_i{:}C_i}}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash \lambda x . \ulcorner L \rrbracket_{\lambda\mu} : D \to B \mid \overrightarrow{\mathsf{n}_i{:}C_i}}\,(\to I)}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash \lambda x . \ulcorner L \rrbracket_{\lambda\mu} : D \to B \mid \mathsf{m}{:}B, \overrightarrow{\mathsf{n}_i{:}C_i}}\,(Wk)$$

Then $\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash \ulcorner M' \rrbracket_{\lambda\mu} \{\lambda x . \ulcorner L \rrbracket_{\lambda\mu} / c_{\mathsf{m}}\} : B \mid \mathsf{m}{:}B\overrightarrow{\mathsf{n}_i{:}C_i}$ follows by Lem. 1.8, and we can construct:

$$\dfrac{\dfrac{\boxed{}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash \ulcorner M' \rrbracket_{\lambda\mu} \{\lambda x . \ulcorner L \rrbracket_{\lambda\mu} / c_{\mathsf{m}}\} : B \mid \mathsf{m}{:}B, \overrightarrow{\mathsf{n}_i{:}C_i}}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash \mu \mathsf{m} . [\mathsf{m}] \ulcorner M \rrbracket_{\lambda\mu} \{\lambda x . \ulcorner N_1 \rrbracket_{\lambda\mu} / c_{\mathsf{m}}\} : B \mid \overrightarrow{\mathsf{n}_i{:}C_i}}\,(\mu) \qquad \square$$

So λ^{try} with basic type assignment is fully representable in $\lambda\mu$.

4 ADDING halt TO λ^{try}

We will now define a notion of type assignment that extends the system we defined above, by allowing for both recoverable and unrecoverable failure; to distinguish raising the latter kind of exception from the former, throw, we considered above, we use the keyword halt. The idea is that halt gets propagated through the system and becomes the end result. Therefore, we need to add reduction rules that consume applicative contexts, as for throw, and make sure to not 'catch' the halt, as that would localise the event and limit its range (see also Rem. 4.9).

Not catching halt is done also for technical reasons. We will argue below that raising a halt is different from throw: when aiming for a representation in $\lambda\mu$, we cannot use handlers and parameter passing for halt. In Rem. 4.9 we will discuss an alternative approach, and indicate why that does not satisfy the purpose.

As mentioned above, following the suggestion of [1], we will aim to map our calculus onto *$\lambda\mu$-top*, where *top* is a special name that cannot occur bound and denotes the top-level. We would therefore want to define a notion of type assignment that, for example, deals with halt by assigning it the type $\bot$, but that would not be possible, as argued in Ex. 4.8.

We extend the calculus λ^{try} from Def. 2.1, by extending the set of pre-terms through adding the construct halt; the notion of reduction is defined as in Def. 2.2, by adding the rule that expresses that also halt consumes an applicative context.

Definition 4.1 (Syntax of $\lambda_{\mathsf{h}}^{\text{TRY}}$). (1) The set of *pre-terms* of $\lambda_{\mathsf{h}}^{\text{TRY}}$ is defined by the grammar:

$$\begin{array}{rcl} \mathsf{Catch\_Block} & ::= & \epsilon \mid \mathsf{Catch\_Block};\ \mathsf{catch}\ \mathsf{n}(x) = M \\ M, N & ::= & V \mid MN \mid \mathsf{try}\ M;\ \mathsf{Catch\_Block} \\ & \mid & \mathsf{throw}\ \mathsf{n}(M) \mid \mathsf{halt} \\ V & ::= & x \mid \lambda x . M \end{array}$$

(2) The CBN-reduction system for $\lambda_{\mathsf{h}}^{\text{TRY}}$ is like that for λ^{try} from Def. 2.2, defined by the rules:

$$\begin{array}{rcll} (\beta) : (\lambda x . M) N & \to & M\{N/x\} & \\ (\mathsf{throw}) : (\mathsf{throw}\ \mathsf{n}(N)) M & \to & \mathsf{throw}\ \mathsf{n}(N) & \\ (\mathsf{halt}) : \mathsf{halt}\ M & \to & \mathsf{halt} & \\ (\mathsf{try\text{-}throw}) : \mathsf{try}\ \mathsf{throw}\ \mathsf{n}_l(N);\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i} & & & \\ & \to & M_l\{N/x\} & (\mathsf{n}_l \in \overrightarrow{\mathsf{n}_i}) \\ (\mathsf{try\text{-}normal}) : \mathsf{try}\ N;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i} & \to & N & (\overrightarrow{\mathsf{n}_i} \notin N) \end{array}$$

CBN *applicative* contexts are defined as in Def. 2.2.

(3) The CBV-reduction system for $\lambda_{\mathsf{h}}^{\text{TRY}}$ is that of CBN, replacing rule (β) by the first reduction rule below, and adding the second and third:

$$\begin{array}{rlcl} (\beta_{\mathsf{V}}) : & (\lambda x . M) V & \to & M\{V/x\} \\ (\mathsf{throw_V}) : & V(\mathsf{throw}\ \mathsf{n}(N)) & \to & \mathsf{throw}\ \mathsf{n}(N) \\ (\mathsf{halt_V}) : & V\ \mathsf{halt} & \to & \mathsf{halt} \end{array}$$

CBV *applicative* contexts are defined as in Def. 2.2.

$$\begin{array}{rc} (Ax) : & \dfrac{}{\Gamma, x{:}A \vdash x : A \mid \Delta} \\[2ex] (\to I) : & \dfrac{\Gamma, x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x . M : A \to B \mid \Delta} \\[2ex] (\to E) : & \dfrac{\Gamma \vdash M : A \to B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta} \\[2ex] (\mathsf{throw}) : & \dfrac{\Gamma \vdash M : A \mid \mathsf{n}{:}A \to B, \Delta}{\Gamma \vdash \mathsf{throw}\ \mathsf{n}(M) : C \mid \mathsf{n}{:}A \to B, \Delta} \\[2ex] (\mathsf{halt}) : & \dfrac{}{\Gamma \vdash \mathsf{halt} : A \mid \Delta} \\[2ex] (\mathsf{try}) : & \dfrac{\Gamma \vdash M : C \mid \overrightarrow{\mathsf{n}{:}A_i \to C}, \Delta \quad \Gamma, x{:}A_i \vdash N_i : C \mid \Delta\ (\forall i \in \underline{n})}{\Gamma \vdash \mathsf{try}\ M;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i} : C \mid \Delta} \end{array}$$

Figure 4: Type assignment for λ^{try} with failure

Notice that the system only handles throws; halt is just propagated through the reduction system until it is the remaining term, as in

$$\mathsf{try}\ \mathsf{halt};\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i} \ \to\ \mathsf{halt}$$

through reduction rule try-normal. So when a halt occurs, no parameter passing takes place, and the event is not handled.

We define failure type assignment for terms in $\lambda_{\mathsf{h}}^{\text{TRY}}$ as follows:

Definition 4.2 (Type assignment for $\lambda_{\mathsf{h}}^{\text{TRY}}$). Type assignment for terms in $\lambda_{\mathsf{h}}^{\text{TRY}}$, $\vdash_{\mathsf{h}}$, is defined through the inference system in Fig 4. We write $\Gamma \vdash_{\mathsf{h}} M : A \mid \Delta$ if this judgement is derivable using these rules.

Notice that we use the same set of types as before, so are not using the type constant $\bot$ that is used in *$\lambda\mu$-top*. Also, the way halt is treated in the type assignment system is the same as throw, in that it allows halt to have any type at all, essentially following the logic rule *ex falso quodlibet sequitur*. Here we do not inhabit this rule with a term construct, as is done for example, in $\Lambda\mu$ [12] and [1]. Rather, we limit its use to just (halt).

So although aborting a computation is successfully modelled in the calculus itself, there is no representation of that in the type system.

We can now show the following soundness result for CBN reduction.

THEOREM 4.3 (SUBJECT REDUCTION FOR $\vdash_{\mathsf{h}}$ WRT $\to^{\text{N}}_{\text{TRY}}$). *If* $\Gamma \vdash_{\mathsf{h}} P : C \mid \Delta$ *and* $P \to^{\text{N}}_{\text{TRY}} Q$, *then* $\Gamma \vdash_{\mathsf{h}} Q : C \mid \Delta$.

We can also show a similar result for CBV:

THEOREM 4.4 (SUBJECT REDUCTION FOR $\vdash_{\mathsf{h}}$ WRT $\to^{\text{V}}_{\text{TRY}}$). *If* $\Gamma \vdash_{\mathsf{h}} P : C \mid \Delta$ *and* $P \to^{\text{V}}_{\text{TRY}} Q$, *then* $\Gamma \vdash_{\mathsf{h}} Q : C \mid \Delta$.

So, in terms of type assignment for a λ-calculus with exceptions, the failure system satisfies the basic requirement with respect to call-by-name and call-by-value reduction.

We can interpret $\lambda_{\mathsf{h}}^{\text{TRY}}$ in *$\lambda\mu$-top* as follows:

Definition 4.5 (Interpretation of $\lambda^{\text{TRY}}_{\mathsf{h}}$ into $\lambda\mu$-top). (1) We add the term constant *halt*[5] to $\lambda\mu$-*top* that can only be assigned $\perp$ by adding the inference rule:

$$(halt):\ \frac{}{\Gamma \vdash_t halt : \perp \mid \Delta}$$

(2) The interpretation of $\lambda^{\text{TRY}}_{\mathsf{h}}$ in $\lambda\mu$-*top* is defined as follows:

$$\begin{array}{lcl}
\ulcorner x \lrcorner_t & \triangleq & x \\
\ulcorner \lambda x.M \lrcorner_t & \triangleq & \lambda x.\ulcorner M \lrcorner_t \\
\ulcorner MN \lrcorner_t & \triangleq & \ulcorner M \lrcorner_t \ulcorner N \lrcorner_t \\
\ulcorner \mathsf{throw\ n}(M) \lrcorner_t & \triangleq & \mu\_.[\mathsf{n}]c_{\mathsf{n}}\ulcorner M \lrcorner_t \\
\ulcorner \mathsf{try}\ M; \epsilon \lrcorner_t & \triangleq & \ulcorner M \lrcorner_t \\
\multicolumn{3}{l}{\ulcorner \mathsf{try}\ M;\ \mathsf{Catch\_Block};\ \mathsf{catch\ n}(x) = L \lrcorner_t \triangleq} \\
\multicolumn{3}{l}{\qquad (\mu\mathsf{n}.[\mathsf{n}]\ulcorner \mathsf{try}\ M;\ \mathsf{Catch\_Block} \lrcorner_t)\,\{\lambda x.\ulcorner L \lrcorner_t / c_{\mathsf{n}}\}} \\
\ulcorner \mathsf{halt} \lrcorner_t & \triangleq & \mu\_.[top]\ halt
\end{array}$$

Notice that, in order to achieve that $\ulcorner \mathsf{halt} \lrcorner_t$ consumes applicative contexts, we need to use the prefix '$\mu\_$', as we have done also for $\ulcorner \mathsf{throw\ n}(M) \lrcorner_t$.

We can now show:

Theorem 4.6 (Soundness of the interpretation for $\lambda^{\text{TRY}}_{\mathsf{h}}$).
(1) *If $P \to^{\text{N}}_{\mathsf{h}} Q$, then $\ulcorner P \lrcorner_t \to^{\text{N}\,*}_{\beta\mu} \ulcorner Q \lrcorner_t$.*
(2) *If $P \to^{\text{V}}_{\mathsf{h}} Q$, then $\ulcorner P \lrcorner_t \to^{\text{N}\,*}_{\beta\mu} \ulcorner Q \lrcorner_t$.*

We can also show that assignable types are preserved.

Theorem 4.7 (Preservation of assignable types). *If $\Gamma \vdash_{\mathsf{h}} M : B \mid \overrightarrow{\mathsf{n}_i{:}A_i \to C_i}$, then $\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_t \ulcorner M \lrcorner : B \mid \overrightarrow{\mathsf{n}_i{:}C_i}$.*

Proof. By induction on the definition of $\vdash_{\mathsf{h}}$, very similar to that of Thm. 3.5, but with an added case.

(halt): Then $M = \mathsf{halt}$. We can construct

$$\dfrac{\dfrac{}{\Gamma \vdash halt : \perp \mid \_{:}C, \Delta}\ (halt)}{\Gamma \vdash \mu\_.[top]\,halt : C \mid \Delta}\ (top)$$

and $\ulcorner \mathsf{halt} \lrcorner = \mu\_.[top]\ halt.$ □

Remark 4.8. In this paper we are mainly looking at the relation between notions of exception handling and classical logic; in that setting, it would be reasonable to add the type constant $\perp$ to the type language, and use it to type halt, as also suggested in the proof of the previous theorem.

This is, on its own, perfectly feasible, and works well on the level of λ^{try} itself, but we would not be able to establish a relation with $\lambda\mu$. For example, we can add the rules

$$(\mathsf{halt}):\ \frac{}{\Gamma \vdash \mathsf{halt} : \perp \mid \Delta}$$

$$(\to E_{\text{N}}):\ \frac{\Gamma \vdash M : \perp \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : \perp \mid \Delta}$$

for a notion of type assignment towards CBN, and add the rule

$$(\to E_{\text{V}}):\ \frac{\Gamma \vdash M : A \mid \Delta \quad \Gamma \vdash N : \perp \mid \Delta}{\Gamma \vdash MN : \perp \mid \Delta}$$

for CBV.[6]

The problem appears in the proof of Thm. 4.7, where we would have the case

$(\to E_{\text{N}})$: Then $M = PQ$, the derivation looks like

$$\dfrac{\dfrac{\boxed{\qquad\qquad}}{\Gamma \vdash_{\mathsf{h}} P : \perp \mid \overrightarrow{\mathsf{n}_i{:}A_i \to C_i}} \quad \dfrac{\boxed{\qquad\qquad}}{\Gamma \vdash_{\mathsf{h}} Q : D \mid \overrightarrow{\mathsf{n}_i{:}A_i \to C_i}}}{\Gamma \vdash_{\mathsf{h}} PQ : \perp \mid \overrightarrow{\mathsf{n}_i{:}A_i \to C_i}}\ (\to E)$$

and by induction we have $\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_t \ulcorner P \lrcorner{:}\perp \mid \overrightarrow{\mathsf{n}_i{:}C_i}$ and $\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_t \ulcorner Q \lrcorner : D \mid \overrightarrow{\mathsf{n}_i{:}C_i}$. In order for us to be able to combine these two derivations in $\vdash_t$, we need to create the arrow type $D{\to}\perp$ from $\perp$. The only way to do that, in $\vdash_t$, is to apply rule (top):

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{\qquad\qquad}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_{\lambda\mu} \ulcorner P \lrcorner : \perp \mid \overrightarrow{\mathsf{n}_i{:}C_i}}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_{\lambda\mu} \ulcorner P \lrcorner : \perp \mid \_{:}D{\to}\perp, \overrightarrow{\mathsf{n}_i{:}C_i}}\ (Wk)}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_{\lambda\mu} \mu\_.[top]\ulcorner P \lrcorner : D \to \perp \mid \overrightarrow{\mathsf{n}_i{:}C_i}}\ (top) \quad \begin{array}{c}\dfrac{\boxed{\qquad\qquad}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_{\lambda\mu} \ulcorner Q \lrcorner : D \mid \overrightarrow{\mathsf{n}_i{:}C_i}} \\ \vdots \end{array}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_{\lambda\mu} (\mu\_.[top]\ulcorner P \lrcorner)\ulcorner Q \lrcorner : \perp \mid \overrightarrow{\mathsf{n}_i{:}C_i}}\ (\to E)$$

but $(\mu\_.[top]\ulcorner P \lrcorner)\ulcorner Q \lrcorner \neq \ulcorner PQ \lrcorner$. In fact, these terms are computationally incompatible.

So we cannot give the type $\perp$ the role it should have.

In the next section, we will introduce a notion of type assignment that uses the type constant fail (which can be seen as $\perp$) for calls to panic, which are essentially catchable halts; as suggested here, we will not be able to establish a relation with $\lambda\mu$ or $\lambda\mu$-*top* for that notion.

Remark 4.9. It might seem natural to define failing through halt in much the same way as throw (and that is basically what is suggested in [1]). This would lead to, contrary to what we have done above, adding $\mathsf{panic\ n}(N)$ as a construct, together with dedicated exception handlers $\mathsf{abort\ n}(x) = L$, so using, for example, the grammar:

$$\begin{array}{rcl}
\mathsf{Catch\_Block} & ::= & \epsilon \mid \mathsf{Catch\_Block};\ \mathsf{catch\ n}(x) = M \\
\mathsf{Abort\_Block} & ::= & \epsilon \mid \mathsf{Abort\_Block};\ \mathsf{abort\ n}(x) = M \\
M, N & ::= & V \mid MN \mid \mathsf{try}\ M;\ \mathsf{Catch\_Block} \mid \mathsf{throw\ n}(N) \\
 & \mid & \mathsf{try}\ M;\ \mathsf{Abort\_Block} \mid \mathsf{panic\ n}(N) \\
V & ::= & x \mid \lambda x.M
\end{array}$$

and the (additional) reduction rules

$$\begin{array}{rcll}
(\mathsf{panic}) : (\mathsf{panic\ n}(N))M & \to & \mathsf{panic\ n}(N) & \\
(\mathsf{try\text{-}panic}) : \mathsf{try\ panic\ n}_l(N);\ \overrightarrow{\mathsf{abort\ n}_i(x) = M_i} & & & \\
 & \to & M_l\{N/x\} & (\mathsf{n}_l \in \overrightarrow{\mathsf{n}_i}) \\
(\mathsf{try\text{-}normal}) : \mathsf{try}\ N;\ \overrightarrow{\mathsf{abort\ n}_i(x) = M_i} & \to & N & (\mathsf{n}_i \notin N)
\end{array}$$

for CBN, and extending the interpretation $\ulcorner\cdot\lrcorner_t$ with the cases:

$$\begin{array}{rcl}
\ulcorner \mathsf{panic\ n}(M) \lrcorner_t & \triangleq & \mu\_.[top]c_{\mathsf{n}}\ulcorner M \lrcorner_t \\
\ulcorner \mathsf{try}\ M;\ \mathsf{Abort\_Block};\ \mathsf{abort\ n}(x) = L \lrcorner_t & \triangleq & \\
\multicolumn{3}{r}{(\mu\mathsf{n}.[\mathsf{n}]\ulcorner \mathsf{try}\ M;\ \mathsf{Abort\_Block} \lrcorner_t)\,\{\lambda x.\ulcorner L \lrcorner_t / c_{\mathsf{n}}\}}
\end{array}$$

This would behave well on the level of λ^{try} (see also the next section), but not when we aim to show that

$$\text{if } P \to_{\mathsf{h}} Q, \text{ then } \ulcorner P \lrcorner_t \to^{\text{N}\,*}_{\beta\mu} \ulcorner Q \lrcorner_t,$$

for either CBN or CBV-reduction. Although that property follows straightforwardly from the proof of Thm. 2.6, and for the additional case panic, the reduction rule try-panic throws a spanner in the works. Then the interpretation of the term

$$\mathsf{try\ panic\ n}(N);\ \mathsf{abort\ n}(x) = M$$

[5]We use *halt* just as a place holder, it is not active in the reduction relation.
[6]Notice that these two variants of $(\to E)$ need to be added to achieve subject reduction.

does not reduce to that of $M\{N/x\}$:

$$\begin{array}{lcl}
\ulcorner \mathsf{try\ panic\ n}(N);\ \mathsf{abort\ n}(x) = M \lrcorner_{\lambda\mu} & \triangleq & \\
(\mu \mathsf{n}.[\mathsf{n}] \ulcorner \mathsf{panic\ n}(N) \lrcorner_{\lambda\mu})\,\{\lambda z.\ulcorner M \lrcorner_t / c_{\mathsf{n}}\} & \triangleq & \\
(\mu \mathsf{n}.[\mathsf{n}] (\mu\_.[top] c_{\mathsf{n}} \ulcorner N \lrcorner_{\lambda\mu}))\,\{\lambda z.\ulcorner M \lrcorner_t / c_{\mathsf{n}}\} & = & \\
\mu \mathsf{n}.[\mathsf{n}] \mu\_.[top]\,(\lambda x.\ulcorner M \lrcorner_{\lambda\mu}) \ulcorner N \lrcorner_{\lambda\mu} & \rightarrow^{\mathrm{N}}_{\beta\mu} & (E) \\
\mu\_.[top]\,(\lambda x.\ulcorner M \lrcorner_{\lambda\mu}) \ulcorner N \lrcorner_{\lambda\mu} & \rightarrow^{\mathrm{N}}_{\beta\mu} & \\
\mu\_.[top] \ulcorner M \lrcorner_{\lambda\mu} \{\ulcorner N \lrcorner_{\lambda\mu} / x\} & = & (2.5) \\
\mu\_.[top] \ulcorner M\{N/x\} \lrcorner_{\lambda\mu} & &
\end{array}$$

So here we have $\ulcorner P \lrcorner_t \rightarrow^{\mathrm{N}*}_{\beta\mu} \mu\_.[top] \ulcorner Q \lrcorner_t$, not $\ulcorner P \lrcorner_t \rightarrow^{\mathrm{N}*}_{\beta\mu} \ulcorner Q \lrcorner_t$, as desired, and $\mu\_.[top] \ulcorner Q \lrcorner_t$ and $\ulcorner Q \lrcorner_t$ are also computationally incompatible.[7]

In a certain sense, the encoding expects $\ulcorner M\{N/x\} \lrcorner_{\lambda\mu}$ to be 'thrown again', which suggests the reduction rule

$$\begin{array}{lll}
(\mathsf{try\text{-}panic}): & \mathsf{try\ panic\ n}_l(N);\ \overrightarrow{\mathsf{abort\ n}_i(x) = M_i} & \\
& \rightarrow\ \ \mathsf{panic\ n}_l(M_l\{N/x\}) & (\mathsf{n}_l \in \overrightarrow{\mathsf{n}_i})
\end{array}$$

where handlers for panics should be redefined consistently, violating Barendregt's convention. Alternatively, we could invoke a handler for all aborts, as in

$$\begin{array}{lll}
(\mathsf{try\text{-}panic}): & \mathsf{try\ panic\ n}_l(N);\ \overrightarrow{\mathsf{abort\ n}_i(x) = M_i} & \\
& \rightarrow\ \ \mathsf{panic\text{-}top}(M_l\{N/x\}) & (\mathsf{n}_l \in \overrightarrow{\mathsf{n}_i})
\end{array}$$

which gets dealt with at the 'outermost level'.

Otherwise, we can assume that there is no handler named n_l, and that the panic escapes the try-block without being processed. But that would constitute the solution we presented above, by just using the keyword halt.

So, in order to define a notion of aborting exceptions for our language λ^{try} that is strongly related to classical logic (*i.e.* mappable into $\lambda\mu$ or variants thereof), we cannot opt to 'handle' these events, nor explicitly use the type $\perp$ to type them, but are forced to add simply a constant halt to the language that consumes all applicative contexts.

5 HANDLING FAILING COMPUTATION

In this section, we will generalise the approach of the previous section, and add the construct panic that is dealt with by handlers. As explained in Rem. 4.8 and 4.9, this is not straightforward, and we will have to forgo on establishing a direct relation with $\lambda\mu$ or *$\lambda\mu$-top*.

Our approach will be to construct a system that adds a type constant fail to the type language, and is set up in such a way that, essentially, only calls to panic can be typed with fail. Our aim is to define a calculus that is close to 'normal' programming: programs can raise exceptions and panic from within the same try-statement. As we argued above, for reasons of subject reduction, we have to demand that the return type of the handlers is equal to that of the main term, which would mean that we cannot return fail for a failing program without having to demand that all handlers return fail. That clearly goes agains intuition, since (1) *we cannot expect the type checker to decide if a program will fail; (2) failure can depend on input, which need not be part of the code; (3) the programmer should have the liberty to cater for the event of a successful computation and a total failure in a different way.* We therefore introduce a new feature: handlers for exceptions, called catch, all return the type of the main term, whereas handlers for panic call, called abort, all return fail.

The system we will present thereby is unconventional in that the standard subject reduction result does not hold as such. Our aim is to show that, as usual, types are preserved under normal reduction (is sound), but that the type fail is only used when a panic is raised; as a result of this duplicity we will not be able to show the normal subject reduction result. Since in standard notions of type assignment for the λ-calculi this property holds, for both the reduction strategies CBN and CBV we need to explicitly insert the duplicity of keeping the type under reduction or running to a term with type fail. Note that this will not be decidable, since modelling the concept that predicts how a program will run, *i.e.* if a panic will be triggered, through assignable types, is impossible.

To introduce the duplicity, we enrich the language with a conditional construct; then depending on the result of running the boolean expression, either the then or else part will be deployed. Assuming the boolean expression tests if the execution is running normally (like a test for division by zero), we can call panic in one part, and continue normal execution in the other. Our aim is that in the first case a term is returned of type fail, whereas the second one will return a normal type, int in our case. We will type the whole term then with int, which then is the type for the result produced by normal reduction.

Remark 5.1. Since the conditional is encodable in the pure λ-calculus through $\lambda btf.btf$, with the boolean constant true through $\lambda ab.a$, and false through $\lambda ab.b$, there is no need to add the conditional construct explicitly for reasons of expressivity. The type bool then necessarily is a type suitable for both $\lambda ab.a$ and $\lambda ab.b$, so has to correspond to $A \rightarrow A \rightarrow A$, for any A (or $\forall \varphi.\varphi \rightarrow \varphi \rightarrow \varphi$). This is found also in the standard way of typeing the conditional construct, which demands that the then and else part have the same type as the expression itself, as normally expressed through the rule:

$$(\mathsf{cond}): \frac{\Gamma \vdash M : \mathsf{bool} \mid \Delta \quad \Gamma \vdash P : A \mid \Delta \quad \Gamma \vdash Q : A \mid \Delta}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q : A \mid \Delta}$$

But this standard approach would not allow us the characterisation of failing computations through assignable types we aim for. So, rather, we deviate from that standard approach and, essentially, let bool correspond to

$$A \rightarrow A \rightarrow A \vee A \rightarrow \mathsf{fail} \rightarrow A \vee \mathsf{fail} \rightarrow A \rightarrow A.$$

If we would allow that to be a type for both $\lambda ab.a$ and $\lambda ab.b$, we would be forced to set $A = \mathsf{fail}$ and we would be forced to allow for fail to be treated as any type, rather than just the type for panic.

So to be able to express the characteristic we aim for, we are forced to add the conditional construct explicitly, which allows us to use non-standard type assignment rule(s) for it that allow the two branches to have different types, provided that one of them is typed with fail. This is achieved by adding the rules of Fig. 5.

We define $\lambda^{\mathrm{TRY}}_{\mathsf{fail}}$ by extending the calculus λ^{try} from Def. 2.1, by adding panic and abort, a conditional construct and term constants (ranged over by c, and including err, true, false, numbers, (prefix)

[7] Notice the similarity with the problem spotted in Rem. 4.8.

$$(Ax): \frac{}{\Gamma, x{:}A \vdash x : A \mid \Delta} \qquad (\mathsf{throw}): \frac{\Gamma \vdash M : A \mid \mathsf{n}{:}A \to B, \Delta}{\Gamma \vdash \mathsf{throw}\ \mathsf{n}(M) : C \mid \mathsf{n}{:}A \to B, \Delta} \qquad (\mathsf{panic}): \frac{\Gamma \vdash M : A \mid \mathsf{n}{:}A \to \mathsf{fail}, \Delta}{\Gamma \vdash \mathsf{panic}\ \mathsf{n}(M) : \mathsf{fail} \mid \mathsf{n}{:}A \to \mathsf{fail}, \Delta}$$

$$(\mathsf{c}): \frac{}{\Gamma \vdash \mathsf{c} : \sigma(\mathsf{c}) \mid \Delta} \qquad (\mathsf{try}): \frac{\Gamma \vdash M : C \mid \overrightarrow{\mathsf{n}{:}A_i \to B_i}, \Delta \quad \overrightarrow{\Gamma, x{:}A_i \vdash N_i : B_i \mid \Delta}\ (\forall i \in \underline{n})}{\Gamma \vdash \mathsf{try}\ M;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i} : C \mid \Delta}\ (\forall i\ [B_i = C \vee B_i = \mathsf{fail}])$$

$$(\to I): \frac{\Gamma, x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A \to B \mid \Delta} \qquad (\mathsf{try}_{\mathsf{fail}}): \frac{\Gamma \vdash M : \mathsf{fail} \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B_i}, \Delta \quad \overrightarrow{\Gamma, x{:}A_i \vdash N_i : B_i \mid \Delta}\ (\forall i \in \underline{n})}{\Gamma \vdash \mathsf{try}\ M;\ \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i} : \mathsf{fail} \mid \Delta}\ (\forall i\ [B_i = C \vee B_i = \mathsf{fail}])$$

$$(\to E): \frac{\Gamma \vdash M : A \to B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta} \qquad (\mathsf{cond}): \frac{\Gamma \vdash M : \mathsf{bool} \mid \Delta \quad \Gamma \vdash P : A \mid \Delta \quad \Gamma \vdash Q : B \mid \Delta}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q : C \mid \Delta}\ \begin{array}{l}(A = B = C \vee (A = C\ \&\ B = \mathsf{fail}) \vee \\ (A = \mathsf{fail}\ \&\ B = C))\end{array}$$

$$(\to E_{\mathsf{fail}}): \frac{\Gamma \vdash M : \mathsf{fail} \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : \mathsf{fail} \mid \Delta} \qquad (\mathsf{cond}_{\mathsf{fail}}): \frac{\Gamma \vdash M : \mathsf{fail} \mid \Delta \quad \Gamma \vdash P : A \mid \Delta \quad \Gamma \vdash Q : B \mid \Delta}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q : \mathsf{fail} \mid \Delta}$$

Figure 5: The system $\vdash_{\mathsf{fail}}$.

addition and multiplication, boolean operators, etc, etc.) to the set of pre-terms.

Definition 5.2 ($\lambda^{\text{TRY}}_{\mathsf{fail}}$). (1) The set of pre-terms of $\lambda^{\text{TRY}}_{\mathsf{fail}}$ is defined through the grammar:

$$\begin{array}{rcl} \mathsf{handlers} & ::= & \mathsf{catch}\ \mathsf{m}(x) = M \mid \mathsf{handlers};\ \mathsf{catch}\ \mathsf{m}(x) = M \mid \\ & & \mathsf{abort}\ \mathsf{n}(x) = N \mid \mathsf{handlers};\ \mathsf{abort}\ \mathsf{n}(x) = N \\ M, N & ::= & V \mid MN \mid \mathsf{try}\ M;\ \mathsf{handlers} \mid \mathsf{throw}\ \mathsf{n}(M) \mid \\ & & \mathsf{panic}\ \mathsf{n}(M) \mid \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q \\ V & ::= & x \mid \mathsf{c} \mid \lambda x.M \end{array}$$

The order in which the handlers are listed is not important; we will reorganise them whenever convenient, and will use handle for either catch or abort.

(2) Call-by-name reduction is defined as in Def. 2.2 by the rules

$$\begin{array}{rll} (\beta): (\lambda x.M)N & \to & M\{N/x\} \\ (\mathsf{throw}): (\mathsf{throw}\ \mathsf{n}(N))M & \to & \mathsf{throw}\ \mathsf{n}(N) \\ (\mathsf{try\text{-}throw}): & & \\ \mathsf{try}\ \mathsf{throw}\ \mathsf{n}_l(N);\ \overrightarrow{\mathsf{handle}\ \mathsf{n}_i(x) = M_i};\ \mathsf{catch}\ \mathsf{n}_l(x) = M_l & & \\ & \to & M_l\{N/x\} \\ (\mathsf{try\text{-}normal}): \mathsf{try}\ N;\ \overrightarrow{\mathsf{handle}\ \mathsf{n}_i(x) = M_i} & \to & N \quad (\overrightarrow{\mathsf{n}_i \notin N}) \\ (\mathsf{panic}): (\mathsf{panic}\ \mathsf{n}(N))M & \to & \mathsf{panic}\ \mathsf{n}(N) \\ (\mathsf{try\text{-}panic}): & & \\ \mathsf{try}\ \mathsf{panic}\ \mathsf{n}_l(N);\ \overrightarrow{\mathsf{handle}\ \mathsf{n}_i(x) = M_i};\ \mathsf{abort}\ \mathsf{n}_l(x) = M_l & & \\ & \to & M_l\{N/x\} \\ (\mathsf{cond\text{-}true}): \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ P\ \mathsf{else}\ Q & \to & P \\ (\mathsf{cond\text{-}false}): \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ P\ \mathsf{else}\ Q & \to & Q \\ (\mathsf{cond\text{-}throw}): \mathsf{if}\ \mathsf{throw}\ \mathsf{n}(N)\ \mathsf{then}\ P\ \mathsf{else}\ Q & \to & \mathsf{throw}\ \mathsf{n}(N) \\ (\mathsf{cond\text{-}panic}): \mathsf{if}\ \mathsf{panic}\ \mathsf{n}(N)\ \mathsf{then}\ P\ \mathsf{else}\ Q & \to & \mathsf{panic}\ \mathsf{n}(N) \end{array}$$

CBN *applicative* contexts are defined as:

$$\mathrm{C}^{\mathrm{A}}_{\mathrm{N}} ::= [\,] \mid \mathrm{C}^{\mathrm{A}}_{\mathrm{N}} M \mid \mathsf{try}\ \mathrm{C}^{\mathrm{A}}_{\mathrm{N}};\ \mathsf{Catch\_Block} \mid \mathsf{if}\ \mathrm{C}^{\mathrm{A}}_{\mathrm{N}}\ \mathsf{then}\ P\ \mathsf{else}\ Q$$

(3) Call-by-value reduction is defined as in the previous part by also changing/adding the rule

$$\begin{array}{rcl} (\beta_{\mathrm{V}}): & (\lambda x.M)V & \to\ M\{V/x\} \\ (\mathsf{panic}_{\mathrm{V}}): & V(\mathsf{panic}\ \mathsf{n}(N)) & \to\ \mathsf{panic}\ \mathsf{n}(N) \end{array}$$

CBV applicative contexts are defined as:

$$\begin{array}{rcl} \mathrm{C}^{\mathrm{A}}_{\mathrm{V}} & ::= & [\,] \mid \mathrm{C}^{\mathrm{A}}_{\mathrm{V}} M \mid V\mathrm{C}^{\mathrm{A}}_{\mathrm{V}} \mid \mathsf{try}\ \mathrm{C}^{\mathrm{A}}_{\mathrm{V}};\ \mathsf{Catch\_Block} \\ & \mid & \mathsf{if}\ \mathrm{C}^{\mathrm{A}}_{\mathrm{V}}\ \mathsf{then}\ P\ \mathsf{else}\ Q \end{array}$$

We will now define a notion of type assignment that characterises *unrecoverable failure.* The idea is that the exception handlers that deal with panic return terms that are typed fail and have to return a panic call, so panic gets propagated through the system and fail becomes the type of the whole program.[8] In order to deal with this properly, we need to extend our notion of type assignment.

Definition 5.3 (Type assignment with throw *and* panic*).*

(1) We extend the set of types by adding the type constant fail and normal type constants, ranged over by c:

$$\begin{array}{rcl} c & ::= & \mathsf{bool} \mid \mathsf{int} \mid \ldots \\ A, B & ::= & \varphi \mid \mathsf{fail} \mid c \mid A \to B. \end{array}$$

(2) Type assignment (with failure) $\vdash^{\mathrm{N}}_{\mathsf{fail}}$ for terms in $\lambda^{\text{TRY}}_{\mathsf{fail}}$ is defined through the inference system presented in Fig. 5, where all types are not equal to fail unless explicitly mentioned, and σ assigns the appropriate ground type to each constant.

(3) The notion $\vdash^{\mathrm{V}}_{\mathsf{fail}}$ is defined using the rules of Fig. 5, extended with the rule

$$(\to E_{\mathrm{V}}): \frac{\Gamma \vdash M : A \mid \Delta \qquad \Gamma \vdash N : \mathsf{fail} \mid \Delta}{\Gamma \vdash MN : \mathsf{fail} \mid \Delta}$$

Note that we no longer require that the handlers return the same type as the main term in a try-expression, but allow them to either return that type, or fail; moreover, each panic $\mathsf{n}(M)$ is typed with fail (fails), and rules $(\to E_{\mathsf{fail}})$ and $(\to E_{\mathrm{V}})$ propagate the type fail in applicative contexts. Also, an abstraction can never fail; the only rule that is allowed for abstractions is $(\to I)$, so the type for an abstraction is of the shape $A \to B$, and both $A \neq \mathsf{fail}$ and $B \neq \mathsf{fail}$.

Example 5.4. We have (essentially) restricted the use of fail to panic only. For example, the term

$$\begin{array}{l} \mathsf{try}\ (\lambda xy.x)\,(\mathsf{panic}\ \mathsf{m}(N))\,(\mathsf{throw}\ \mathsf{n}(L)); \\ \qquad \mathsf{catch}\ \mathsf{n}(x) = P;\ \mathsf{abort}\ \mathsf{m}(x) = Q \end{array}$$

is not typeable in $\vdash_{\mathsf{fail}}$, since it would demand that the type for $\lambda xy.x$ contains fail. It would be typeable if we relax the restriction, allowing fail as a normal type. Take the sub-term

$$M = (\lambda xy.x)\,(\mathsf{panic}\ \mathsf{m}(N))(\mathsf{throw}\ \mathsf{n}(L))$$

which will panic. We can allow the throw and panic to return different types inside M, as in Fig. 6. When we place this term inside the context of dealing with the catch on n and abort on m, the special character of the rule (try) in $\vdash_{\mathsf{fail}}$ becomes evident; it allows the return type of exception handlers to *differ* from the type of the main term in case the latter is fail.

[8]We could even add the term halt with type $\perp$ for this purpose, similar to the previous section.

Let $\Delta' = \mathsf{m};\, B \to \mathsf{fail}, \mathsf{n}{:}C \to A, \Delta$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, x{:}\mathsf{fail}, y{:}A \vdash x : \mathsf{fail} \mid \Delta'}\,(Ax)}{\Gamma, x{:}\mathsf{fail} \vdash \lambda y.x : A \to \mathsf{fail} \mid \Delta'}\,(\to I)}{\Gamma \vdash \lambda xy.x : \mathsf{fail} \to A \to \mathsf{fail} \mid \Delta'}\,(\to I) \quad \dfrac{\dfrac{\boxed{}}{\Gamma \vdash N : C \mid \Delta'}}{\Gamma \vdash \mathsf{panic\ m}(N) : \mathsf{fail} \mid \Delta'}\,(\mathsf{panic})}{\Gamma \vdash (\lambda xy.x)\,(\mathsf{panic\ m}(N)) : A \to \mathsf{fail} \mid \Delta'}\,(\to E) \quad \dfrac{\dfrac{\boxed{}}{\Gamma \vdash L : B \mid \Delta'}}{\Gamma \vdash \mathsf{throw\ n}(L) : A \mid \Delta'}\,(\mathsf{throw})}{\Gamma \vdash (\lambda xy.x)\,(\mathsf{panic\ m}(N))(\mathsf{throw\ n}(L)) : \mathsf{fail} \mid \Delta'}\,(\to E)$$

Figure 6: A derivation for $(\lambda xy.x)\,(\mathsf{panic\,m}(N))(\mathsf{throw\ n}(L))$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \mid \Delta}\,(\mathsf{true}) \quad \dfrac{\dfrac{\boxed{}}{\Gamma \vdash N : C \mid \Delta}}{\Gamma \vdash \mathsf{panic\ m}(N) : \mathsf{fail} \mid \Delta}\,(\mathsf{panic}) \quad \dfrac{\dfrac{\boxed{}}{\Gamma \vdash L : B \mid \Delta}}{\Gamma \vdash \mathsf{throw\ n}(L) : A \mid \Delta}\,(\mathsf{throw})}{\Gamma \vdash \mathsf{if\ true\ then\ panic\ m}(N)\ \mathsf{else\ throw\ n}(L) : A \mid \Delta}\,(\mathsf{cond}) \quad \dfrac{\boxed{}}{\Gamma, x{:}B \vdash P : A \mid \emptyset} \quad \dfrac{\boxed{}}{\Gamma, x{:}C \vdash Q : \mathsf{fail} \mid \emptyset}}{\Gamma \vdash \mathsf{try\ if\ true\ then\ panic\ m}(N)\ \mathsf{else\ throw\ n}(L);\ \mathsf{catch\ n}(x) = P;\ \mathsf{abort\ m}(x) = Q : A \mid \emptyset}\,(\mathsf{try})$$

Figure 7: A derivation for $\mathsf{try\ if\ true\ then\ panic\,m}(N)\ \mathsf{else\ throw\,n}(L);\ \mathsf{catch\ n}(x) = P;\ \mathsf{abort\ m}(x) = Q$

$$\dfrac{\dfrac{\boxed{}}{\Gamma \vdash M : \mathsf{fail} \mid \Delta_2} \quad \dfrac{\boxed{}}{\Gamma, x{:}C \vdash P : A \mid \Delta} \quad \dfrac{\boxed{}}{\Gamma, x{:}C \vdash Q : \mathsf{fail} \mid \emptyset}}{\Gamma \vdash \mathsf{try}\ M;\ \mathsf{catch\ n}(x) = P;\ \mathsf{abort\ m}(x) = Q : \mathsf{fail} \mid \Delta}\,(\mathsf{try})$$

But relaxing the restriction would take away the characteristic that the type fail indicates a failing execution; we therefore opt to have fewer typeable terms.

Using the conditional structure, the similar term

$$\mathsf{try\ if\ true\ then\ panic\ m}(N)\ \mathsf{else\ throw\ n}(L);\quad \mathsf{catch\ n}(x) = P;\ \mathsf{abort\ m}(x) = Q$$

is typeable under the restriction; see Fig. 7.

Our notion of type assignment is predictive in that we can show that terms typed with fail will raise a panic.

Lemma 5.5. (1) *If* $\Gamma \vdash_{\mathsf{fail}} M : \mathsf{fail} \mid \Delta$, *then* $M \to^{\mathrm{N}}_{\mathrm{P}} \mathsf{panic\ n}(N)$.
(2) *If* $\Gamma \vdash^{\mathrm{V}}_{\mathsf{fail}} M : \mathsf{fail} \mid \Delta$, *then* $M \to^{\mathrm{V}}_{\mathrm{P}} \mathsf{panic\ n}(N)$.

Notice that something similar also holds for type assignment in the λ-calculus (extended with type constants): if $\Gamma \vdash M : \mathsf{int}$, then M will run to an integer. Note that, because of the presence of throw, this property does not hold for $\vdash_{\mathsf{fail}}$.

So failing (having type fail) is now exclusively the domain of panic, as we intended from the outset; in particular, the type assignment system forces the type of the body of an abort to have type fail as well, running the body of each abort has to result in a panic as well.

We can also show that type assignment is closed under term substitution.

Lemma 5.6 (Substitution lemma for $\vdash_{\mathsf{fail}}$). *If* $\Gamma, x{:}C \vdash_{\mathsf{fail}} M : A \mid \Delta$ *and* $\Gamma \vdash_{\mathsf{fail}} N : C \mid \Delta$, *then* $\Gamma \vdash_{\mathsf{fail}} M\{N/x\} : A \mid \Delta$.

The main result we show for this system is the following soundness result. It states that running a program will either run normally, preserving the assigned type, or will run to a term that has type fail, so throws a panic.

Theorem 5.7 (Soundness for $\vdash_{\mathsf{fail}}$ with respect to $\to^{\mathrm{N}}_{\mathsf{fail}}$). *If* $\Gamma \vdash_{\mathsf{fail}} P{:}C \mid \Delta$ *and* $P \to^{\mathrm{N}}_{\mathsf{fail}}{}^{*}\, Q$, *then either* $\Gamma \vdash_{\mathsf{fail}} Q{:}C \mid \Delta$, *or* $\Gamma \vdash_{\mathsf{fail}} Q{:}\mathsf{fail} \mid \Delta$.

Proof. By induction on the definition $\to^{\mathrm{N}}_{\mathsf{fail}}{}^{*}$; we focus on the single step reduction, and only show the interesting cases, that were not already included in the proof of Thm. 3.3.

(panic): Then $P = (\mathsf{panic\ n}(N))\,M \to \mathsf{panic\ n}(N) = Q$, and the return type for the exception handler n is fail; then the derivation for P looks like:

$$\dfrac{\dfrac{\dfrac{\boxed{\mathcal{D}}}{\Gamma \vdash N : A \mid \mathsf{n}{:}A \to \mathsf{fail}, \Delta}}{\Gamma \vdash \mathsf{panic\ n}(N) : \mathsf{fail} \mid \mathsf{n}{:}A \to \mathsf{fail}, \Delta}\,(\mathsf{panic}) \quad \begin{array}{c}\dfrac{\boxed{}}{\Gamma \vdash M : B \mid \mathsf{n}{:}A \to \mathsf{fail}, \Delta}\\ \vdots\end{array}}{\Gamma \vdash (\mathsf{panic\ n}(N))\,M : \mathsf{fail} \mid \mathsf{n}{:}A \to \mathsf{fail}, \Delta}\,(\to E_{\mathsf{f}})$$

Notice that we have a sub-derivation for $\Gamma \vdash_{\mathsf{fail}} \mathsf{panic\ n}(N) : \mathsf{fail} \mid \mathsf{n}{:}A \to \mathsf{fail}, \Delta$.

(try-panic): Then $P = \mathsf{try\ panic\ n}_l(M);\ \overrightarrow{\mathsf{abort\ n}_i(x) = N_i} \to N_l\{M/x\} = Q$ and the derivation for P is shaped as follows:

$$\dfrac{\dfrac{\dfrac{\boxed{}}{\Gamma \vdash M : A_l \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B_i}, \Delta}}{\Gamma \vdash \mathsf{panic\ n}_l(M) : \mathsf{fail} \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B_i}, \Delta}\,(\mathsf{panic}) \quad \begin{array}{c}\dfrac{\boxed{}}{\Gamma, x{:}A_i \vdash N_i : B_i \mid \Delta}\\ \vdots\ (l \in \underline{n}, \forall i \in \underline{n})\end{array}}{\Gamma \vdash \mathsf{try\ panic\ n}_l(M);\ \overrightarrow{\mathsf{abort\ n}_i(x) = N_i} : \mathsf{fail} \mid \Delta}\,(\mathsf{try})$$

so $B_l = \mathsf{fail}$. In particular, we have derivations for both $\Gamma \vdash_{\mathsf{fail}} M : A_l \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B_i}, \Delta$ and $\Gamma, x{:}A_l \vdash_{\mathsf{fail}} N_l : \mathsf{fail} \mid \Delta$.[9] By thinning, we can remove $\overrightarrow{\mathsf{n}_i{:}A_i \to B_i}$ from the co-context for the first to obtain $\Gamma \vdash_{\mathsf{fail}} M : A_l \mid \Delta$. Then, by Lem. 5.6, we obtain $\Gamma \vdash_{\mathsf{fail}} N_l\{M/x\} : \mathsf{fail} \mid \Delta$.

(cond-true): Then $P = \mathsf{if\ true\ then}\ M\ \mathsf{else}\ N \to M = Q$. Since true can only be assigned bool, the derivation is constructed as follows:

$$\dfrac{\dfrac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool} \mid \Delta}\,(\sigma) \quad \dfrac{\boxed{}}{\Gamma \vdash Q : A \mid \Delta} \quad \dfrac{\boxed{}}{\Gamma \vdash N : B \mid \Delta}}{\Gamma \vdash \mathsf{if\ true\ then}\ Q\ \mathsf{else}\ N : C \mid \Delta}\,(\mathsf{cond})$$

and either:

$((A = B = C) \vee (A = C \;\&\; B = \mathsf{fail}))$: Then, in particular, $\Gamma \vdash_{\mathsf{fail}} Q : C \mid \Delta$.

$(A = \mathsf{fail} \;\&\; B = C)$: Then, in particular, $\Gamma \vdash_{\mathsf{fail}} Q : \mathsf{fail} \mid \Delta$.

[9] Remark that we cannot apply $(\to I)$ to the latter result.

(cond-throw): Then $P = \mathsf{if\ throw\ n}(R)\ \mathsf{then}\ M\ \mathsf{else}\ N \rightarrow \mathsf{throw\ n}(P) = Q$, and the derivation for P is constructed as:

$$\dfrac{\dfrac{\begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ \Gamma \vdash R : E \mid \mathsf{n}{:}E \rightarrow F, \Delta'\end{array}}{\Gamma \vdash \mathsf{throw\ n}(R) : \mathsf{bool} \mid \mathsf{n}{:}E \rightarrow F, \Delta'}\,(\mathsf{throw}) \qquad \begin{array}{c}\boxed{\quad\quad}\\ \Gamma \vdash M : B \mid \Delta\\ \vdots\end{array} \qquad \begin{array}{c}\boxed{\quad\quad}\\ \Gamma \vdash N : C \mid \Delta\end{array}}{\Gamma \vdash \mathsf{if\ throw\ n}(P)\ \mathsf{then}\ M\ \mathsf{else}\ N : D \mid \Delta}\,(\mathsf{cond})$$

for certain B, C, and D. Then we can construct the derivation:

$$\dfrac{\begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ \Gamma \vdash R : E \mid \mathsf{n}{:}E \rightarrow F, \Delta'\end{array}}{\Gamma \vdash \mathsf{throw\ n}(R) : D \mid \mathsf{n}{:}E \rightarrow F, \Delta'}\,(\mathsf{throw})$$

(cond-panic): Then $P = \mathsf{if\ panic\ n}(R)\ \mathsf{then}\ M\ \mathsf{else}\ N \rightarrow \mathsf{panic\ n}(R) = Q$, and the derivation for P is constructed as:

$$\dfrac{\begin{array}{c}\boxed{\quad\quad\quad}\\ \Gamma \vdash \mathsf{panic\ n}(R) : \mathsf{fail} \mid \Delta\end{array} \qquad \begin{array}{c}\boxed{\quad\quad}\\ \Gamma \vdash M : B \mid \Delta\end{array} \qquad \begin{array}{c}\boxed{\quad\quad}\\ \Gamma \vdash N : C \mid \Delta\end{array}}{\Gamma \vdash \mathsf{if\ panic\ n}(R)\ \mathsf{then}\ M\ \mathsf{else}\ N : \mathsf{fail} \mid \Delta}\,(\mathsf{cond})$$

for certain B and C. Notice that we have a sub-derivation for $\Gamma \vdash_{\mathsf{fail}} \mathsf{panic\ n}(R) : \mathsf{fail} \mid \Delta$. □

For the call-by-value system, we can show:

Theorem 5.8 (Soundness for $\vdash_{\mathsf{fail}}$ with respect to $\rightarrow^{\mathrm{v}}_{\mathrm{TRY}}$). *If $\Gamma \vdash^{\mathrm{v}}_{\mathsf{fail}} P{:}C \mid \Delta$ and $P \rightarrow^{\mathrm{v}}_{\mathrm{TRY}}{}^{*}\ Q$, then either $\Gamma \vdash^{\mathrm{v}}_{\mathsf{fail}} Q{:}C \mid \Delta$, or $\Gamma \vdash^{\mathrm{v}}_{\mathsf{fail}} Q{:}\mathsf{fail} \mid \Delta$.*

Proof. The proof is much like that for the previous result, with the addition of:

($\mathsf{throw}_{\mathrm{V}}$): Then $\Delta = \mathsf{n}{:}A \rightarrow C, \Delta'$, $P = V(\mathsf{throw\ n}(N)) \rightarrow \mathsf{throw\ n}(N) = Q$, and the derivation for P is constructed as:

$$\dfrac{\begin{array}{c}\boxed{\quad\quad\quad}\\ \Gamma \vdash_{\mathsf{fail}} V : E \rightarrow F \mid \mathsf{n}{:}A \rightarrow C, \Delta'\end{array} \qquad \dfrac{\begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ \Gamma \vdash_{\mathsf{fail}} N : A \mid \mathsf{n}{:}A \rightarrow C, \Delta'\end{array}}{\Gamma \vdash_{\mathsf{fail}} \mathsf{throw\ n}(N) : E \mid \mathsf{n}{:}A \rightarrow C, \Delta'}\,(\mathsf{throw})}{\Gamma \vdash_{\mathsf{fail}} V(\mathsf{throw\ n}(N)) : F \mid \mathsf{n}{:}A \rightarrow C, \Delta'}\,(\rightarrow E)$$

We can construct the derivation for Q:

$$\dfrac{\begin{array}{c}\boxed{\quad\mathcal{D}\quad}\\ \Gamma \vdash_{\mathsf{fail}} N : A \mid \mathsf{n}{:}A \rightarrow C, \Delta'\end{array}}{\Gamma \vdash_{\mathsf{fail}} \mathsf{throw\ n}(N) : F \mid \mathsf{n}{:}A \rightarrow C, \Delta'}\,(\mathsf{throw})$$

($\mathsf{panic}_{\mathrm{V}}$): Then $P = V(\mathsf{panic\ n}(N)) \rightarrow \mathsf{panic\ n}(N) = Q$, and the derivation for P is constructed like:

$$\dfrac{\begin{array}{c}\boxed{\quad\quad}\\ \Gamma \vdash V : A \mid \Delta\end{array} \qquad \begin{array}{c}\boxed{\quad\quad\quad}\\ \Gamma \vdash \mathsf{panic\ n}(N) : \mathsf{fail} \mid \Delta\end{array}}{\Gamma \vdash V(\mathsf{panic\ n}(N)) : \mathsf{fail} \mid \Delta}\,(\rightarrow E_{\mathrm{V}})$$

We have $\Gamma \vdash_{\mathsf{fail}} \mathsf{panic\ n}(N) : \mathsf{fail} \mid \Delta$ in a sub-derivation. □

Conclusion

We have defined λ^{try}, a natural extension to the λ-calculus by adding exception handling, and shown that it can be embedded into $\lambda\mu$, preserving both cbn and cbv reduction. The normal notion of type assignment for λ^{try}, here called the basic system, is also preserved by our mapping onto $\lambda\mu$. Type assignment is not preserved, however, for the notion of type assignment that captures total program failure using exception handling. We also have presented a notion of handling of exception and panic calls, together with a natural notion of type assignment, that cannot be represented in $\lambda\mu$ or $\lambda\mu$-*top*.

We thus have shown that, although a strong link between typeable exception handling and double negation elimination is evident, exception handling *itself* is a feature that is not naturally a part of calculi based on classical logic, since it is possible to define notions of type assignment that are natural for λ^{try}, but are not founded on classical logic.

By letting go of the link between programming and logic, we have shown that it is possible to define handling of exception and panic calls for formal calculi in a computationally meaningful way.

Acknowledgements

I am greatly indebted to the students who, over the years, have worked on this and other problems related to classical logic and functional programming with me. A special mention is for Mihai-Radu-Niculae Popescu, Wil Fisher, Adela Baciu, James Griffiths, and Isaac van Bakel.

REFERENCES

[1] Z.M. Ariola, H. Herbelin, and A. Sabry. 2007. A Proof-Theoretic Foundation of Abortive Continuations. In *Proceedings of Higher-Order and Symbolic Computation, 2007.* 403–429.
[2] H. Barendregt. 1984. *The Lambda Calculus: its Syntax and Semantics* (revised ed.). North-Holland, Amsterdam.
[3] G.M. Bierman. 1998. A Computational Interpretation of the $\lambda\mu$-calculus. In *MFCS'98 (LNCS)* 1450, 336–345.
[4] G.M. Bierman. 1998. *A Computational Interpretation of the $\lambda\mu$-calculus.* Technical Report. University of Cambridge. Expanded version of [3].
[5] T. Crolard. 1999. A confluent lambda-calculus with a catch/throw mechanism. *Journal of Functional Programming* 9, 6 (1999), 625–647.
[6] H.B. Curry. 1934. Functionality in Combinatory Logic. In *Proc. Nat. Acad. Sci. U.S.A*, Vol. 20. 584–590.
[7] S. Drossopoulou and T. Valkevych. 2000. Java Exceptions Throw No Surprises. (March 2000). http://pubs.doc.ic.ac.uk/nosurprises-00/ Unpublished.
[8] M. Felleisen and R Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103, 2 (1992).
[9] G. Gentzen. 1935. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift* 39, 2 (1935), 176–210 and 405–431.
[10] J. Gosling, W.N. Joy, and G.L. Steele Jr. 1996. *The Java Language Specification.* Addison-Wesley.
[11] T. Griffin. 1990. A formulae-as-types notion of control. In *POPL'90*, 47–58.
[12] Ph. de Groote. 1994. On the Relation between the $\lambda\mu$-Calculus and the Syntactic Theory of Sequential Control. In *LPAR'94. LNCS* 822, 31–43.
[13] Ph. de Groote. 1995. A Simple Calculus of Exception Handling. In *TLCA '95, 1995, Proceedings LNCS* 902, 201–215.
[14] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. 1992. Report on the Programming Language Haskell. *ACM SIGPLAN Notices* 27, 5 (1992), 1–64.
[15] H. Nakano. 1994. The Non-deterministic Catch and Throw Mechanism and Its Subject Reduction Property. In *Logic, Language and Computation, Festschrift in Honor of Satoru Takasu LNCS* 792, 61–72.
[16] H. Nakano. 1995. *Logical Structures of the Catch and Throw Mechanism.* PhD thesis. University of Tokyo.
[17] C.-H.L. Ong and C.A. Stewart. 1997. A Curry-Howard foundation for functional computation with control. In *Proceedings of the 24th Annual ACM Symposium on Principles Of Programming Languages.* 215–227.
[18] M. Parigot. 1992. An algorithmic interpretation of classical natural deduction. In *LPAR'92, LNCS* 624, 190–201.
[19] M. Parigot. 1993. Classical Proofs as Programs. In *Kurt Gödel Colloquium.* 263–276. Presented at TYPES Workshop, at Bǎstad, June 1992.
[20] M.E. Szabo (Ed.). 1969. *The Collected Papers of Gerhard Gentzen.* North-Holland.