Exception Handling with and without Classical Logic

Steffen van Bakel

Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2BZ, UK

s.vanbakel@imperial.ac.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 call-by-name 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 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 tp as destination. We will argue that introducing handlers for halt will break the relation with $\lambda\mu$. We will show 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. We conclude the paper by showing that the system enjoys the principal typing property.

Introduction

For a long time it has been thought that only Intuitionistic Logic [9, 10, 11] had any computational meaning, given its strong relation with types in functional programming and the λ-calculus [13, 5] through the Curry-Howard isomorphism [21]. However, since it is not possible to comfortably express notions like control or context manipulation in the pure λ-calculus, it is clear that that by itself, although Turing complete, it is perhaps not expressive enough. Most of the 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 require different formalisms for behaviour specification - e.g. translation to continuation-passing style (CPS) or abstract machines.

That situation changed when Griffin [18] observed that the C-operator of Felleisen's λC-calculus [15] can be typed with $\neg\neg A \to A$ (or better, $((A\to\bot)\to\bot)\to A$), thus highlighting the first link between Classical Logic [16, 34] and sequential control in computer science. This led to work by Parigot, who introduced a candidate for describing delimited continuations in the form of the $\lambda\mu$-calculus [28, 29], a calculus that represents minimal classical logic [1].

In this paper we will investigate the relation between exception handling and Classical Logic, 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 named (recoverable) exception handling, and investigate if its natural notions of type assignment can be represented in Classical Logic through mapping those to $\lambda\mu$.

We will also add non-recoverable exceptions (through halt and panic); then the corresponding calculus is $\lambda\mu$-tp [1], but only if we do not 'catch' these exceptions. When trying to add handlers for failing exceptions, the correspondence with $\lambda\mu$ or $\lambda\mu$-tp 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 fail (run to a term that has type 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]

The study of the relation between exception handling and classical logic goes back a few decades. Fundamental work has been done by Nakano [25, 26], followed by Crolard [12], 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 [7] interpretation into $\lambda\mu$ of de Groote's [20] calculus $\lambda^{\rightarrow}_{exn}$, which has an exception handling mechanism *à la* ML [24], and in a certain sense also by [27], albeit for call-by-value (CBV) languages. However, in both approaches the argument of the throw-term is the actual exception handler, rather than information that gets passed to the handler, as is usual.

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 [25]) and define *reduction strategies* that do not permit reduction inside an abstraction; unlike in other papers, we will we consider both call-by-name (CBN) and CBV.

Rather than selecting the exception handler *through a type constructor* or *through a type*, as is the common practice in languages like `java` [17], 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 CBN and CBV reduction in λ^{try}; as was the case in previous work [27, 12, 8], 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 [14] for example in `java` and ML; 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 on the (abstract interpretation) level of types, and allowing for execution to continue normally even after an exception has been thrown: those exceptions are *recoverable*. 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 represent 'failure'; we add the construct halt, which corresponds to an exception that cannot be caught so has no possibility of recovery. This induces a non-standard notion of type assignment, where failure is propagated 'outwards', for which we will show soundness for both CBN and CBV. We will modify the interpretation into on mapping onto $\lambda\mu$-tp, 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

[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.

are sound but not really expressive: for example, both throw and halt can have all types. Although the type $\perp$ is used when mapping the calculus into $\lambda\mu$-tp, 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, but now equipped with parameter passing and called panic, so achieve a notion of 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 $\lambda\mu$-tp.

These results 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.

Contents of this paper

We start the paper by giving an overview of past related research and results in Sect. 1; in particular we will look at CBN and CBV-reduction in Parigot's $\lambda\mu$ and give formal proofs for subject reduction results for both those notions. In Sect. 2 we will look at systems dealing with exception handing, like ML, Nakano's system and Crolard's interpretation thereof in to $\lambda\mu$. We will also discuss de Groote's calculus $\lambda^{\rightarrow}_{exn}$, and Bierman's interpretation into CBV $\lambda\mu$.

In Sect. 3 we will present λ^{try}, our λ-calculus extended with recoverable exception handling through try-catch and throw, for which we will define CBN and CBV reduction strategies. We will present an interpretation $\llbracket\cdot\rrbracket^{\lambda\mu}$ from λ^{try} into $\lambda\mu$, and show that it preserves both CBN and CBV-reductions. In Sect. 4 we will define a basic notion of type assignment for λ^{try}, for which we show subject reduction and preservation of types with respect to $\llbracket\cdot\rrbracket^{\lambda\mu}$.

This is followed by Sect. 5, where we start our investigation into adding handling of unrecoverable exceptions to λ^{try}. We start by defining $\lambda^{\mathsf{try}}_{\mathsf{H}}$, adding the constant halt that basically escapes all contexts, and extend type assignment by treating halt like throw and show that to be sound for both CBN and CBV reduction. We will extend our interpretation to one mapping onto $\lambda\mu$-tp, and show that both CBN and CBV reductions are preserved. However, we will argue in Sect. 5.2 that it is not possible to add handlers for halt whilst retaining those properties.

Then in Sect. 6 we extend λ^{try} to $\lambda^{\mathsf{try}}_{\mathsf{F}}$ by adding catchable, but unrecoverable exceptions (panic). To make this relevant, we also add a conditional construct that we type in a non-conventional manner. We add the type constant fail to the type language, and define a CBN and CBV notion of type assignment for which we show soundness. We conclude in Sect. 7, where we show the practicality of $\lambda^{\mathsf{try}}_{\mathsf{F}}$ by extending the notion of principal typings from the λ-calculus to $\lambda^{\mathsf{try}}_{\mathsf{F}}$, and show soundness and completeness results for that.

*: Note. An extended abstract of this paper appeared as [3]; in particular, this paper adds a discussion on past papers on its topic, gives detailed proofs, gives a more refined presentation of the system with failure, addresses a problem with call-by-value reduction on that system, and shows the principal typing property for it. A version of this paper with all proofs given in full can be found at `www.doc.ic.ac.uk/~svb/Research`.

1 Context and Background

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 [13, 5], and Parigot's $\lambda\mu$ [28].

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, β, CBN, AND CBV REDUCTION [5]) *i*) *λ-terms* are defined by the grammar:

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

Seeing λ as a binder, the notion of free and bound variables is defined as usual, and we accept Barendregt's convention to keep free and bound variables distinct, using (silent) α-conversion whenever necessary. 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, $x \in M$ if x occurs in M, either free of bound, and call a term *closed* if it has no free variables.

ii) Contexts are defined as terms with a single hole $[\,]$ by:

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

We write $\mathtt{C}[M]$ for the term obtained from the context $\mathtt{C}$ 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:

$$\mathtt{C}[(\lambda x.M)N] \rightarrow \mathtt{C}[M\{N/x\}]$$

for any context. We write $\rightarrow_\beta^*$ for the reflexive, transitive closure of $\rightarrow_\beta$.

iii) The *Call-by-name evaluation contexts* are defined through:

$$\mathtt{C}_\mathrm{N} ::= [\,] \mid \mathtt{C}_\mathrm{N}M$$

The Call-by-name (CBN) reduction strategy $\rightarrow_\beta^\mathrm{N}$ is defined through:

$$\mathtt{C}_\mathrm{N}[(\lambda x.M)N] \rightarrow \mathtt{C}_\mathrm{N}[M\{N/x\}]$$

iv) The.*Call-by-value evaluation contexts* are defined through:

$$\mathtt{C}_\mathrm{V} ::= [\,] \mid \mathtt{C}_\mathrm{V}M \mid V\mathtt{C}_\mathrm{V}$$

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

$$\mathtt{C}_\mathrm{V}[(\lambda x.M)V] \rightarrow \mathtt{C}_\mathrm{V}[M\{V/x\}]$$

Remark that CBN-evaluation contexts do not allow the left-hand side to be an abstraction (as is allowed in normal evaluation contexts), so do not allow the contraction of redexes inside an abstraction; neither is it allowed to reduce inside the right-hand side, so certainly it is not allowed to reduce inside a redex. For CBV-evaluation contexts, we are allowed to reduce inside the right-hand side, but only if that is not an abstraction (nor a variable), so the application is not a CBV-redex. Thereby $\rightarrow_\beta^\mathrm{N}$ and $\rightarrow_\beta^\mathrm{V}$ are called strategies since in both there can only ever be one redex to contract; this is not the case for $\rightarrow_\beta$.

The notions and notations of this definition are also used for the other calculi defined in this paper.

Curry (or simple) type assignment for the λ-calculus is defined as follows:

Definition 1.2 (CURRY TYPE ASSIGNMENT FOR THE λ-CALCULUS) *i*) Let φ range over a countable (infinite) set of type-variables. The set of *Curry types* is defined by the grammar:

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

ii) A *context* (of term 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 $\Gamma,\{x{:}A\}$, $x \notin \Gamma$ if there exists no A such that $x{:}A \in \Gamma$, and $\Gamma \backslash x$ for $\Gamma \backslash \{x{:}A\}$.

iii) *Curry type assignment* is defined by the following inference system:

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

$$(\rightarrow I): \frac{\Gamma,x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \rightarrow B}\,(x \notin \Gamma) \qquad (\rightarrow E): \frac{\Gamma \vdash M : A \rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

We write $\Gamma \vdash_{\mathrm{C}} M : A$ for judgements derivable using these rules.

The main properties for $\vdash_{\mathrm{C}}$ are that type assignment is decidable, and every typeable term is strongly normalisable.

Decidability of type assignment is established through the definition of an algorithm that defines the principal typing for each term, using Robinson's unification algorithm [32].

Definition 1.3 (Substitution and unification) *i*) *a*) The *substitution* $(\varphi \mapsto C)$, where φ is a type variable and C a Curry type, is inductively defined[2] by:

$$\begin{array}{lcl} (\varphi \mapsto C)\ \varphi & = & C \\ (\varphi \mapsto C)\ \varphi' & = & \varphi' \\ (\varphi \mapsto C)\ A \rightarrow B & = & ((\varphi \mapsto C)\ A) \rightarrow ((\varphi \mapsto C)\ B) \end{array}$$

b) If S_1, S_2 are substitutions, then so is $S_1 \circ S_2$, where $S_1 \circ S_2\, A = S_1\,(S_2\, A)$.

c) $S\Gamma = \{\, x{:}SB \mid x{:}B \in \Gamma \,\}$.

d) $S\langle \Gamma; A \rangle = \langle S\Gamma; SA \rangle$.

e) If there exists a substitution S such that $SA = B$, then B is a *(substitution) instance* of A.

f) Id_S is the identity substitution that replaces all type variables by themselves.

ii) (Robinson's unification algorithm) Unification of Curry types is defined by:

$$\begin{array}{llll} \mathit{unify}\ \varphi & \varphi & = & (\varphi \mapsto \varphi) \\ \mathit{unify}\ \varphi & B & = & (\varphi \mapsto B) \quad (\varphi \textit{ does not occur in } B) \\ \mathit{unify}\ A & \varphi & = & \mathit{unify}\ \varphi\ A \\ \mathit{unify}\ (A \rightarrow B) & (C \rightarrow D) & = & S_2 \circ S_1 \\ & \text{where} & & S_1 = \mathit{unify}\ A\ C \\ & & & S_2 = \mathit{unify}\ (S_1\, B)\ (S_1\, D) \end{array}$$

iii) The operation *unifyC* generalises *unify* to contexts:

$$\begin{array}{llll} \mathit{unifyC}\ (\Gamma_1,x{:}A) & (\Gamma_2,x{:}B) & = & S_2 \circ S_1, \\ & \text{where} & & S_1 = \mathit{unify}\ A\ B \\ & & & S_2 = \mathit{unifyC}\ (S_1\, \Gamma_1)\ (S_1\, \Gamma_2) \\ \mathit{unifyC}\ (\Gamma_1,x{:}A) & \Gamma_2 & = & \mathit{unifyC}\ \Gamma_1\ \Gamma_2 \quad (x \notin \Gamma_2) \\ \mathit{unifyC}\ \emptyset & \Gamma_2 & = & Id_S \end{array}$$

This definition specifies *unify* as a partial function; if the side condition '*φ does not occur in B*' fails, no result is returned. So, for example, *unify* $\varphi\ \varphi \rightarrow \varphi$ does not return a substitution.

If successful, unification returns the most general unifier:

[2] All algorithmic definitions in this paper are presented in 'functional style', where calls are matched against the alternatives 'top-down', the first match is taken, and the result is undefined in case there is no match.

Lemma 1.4 ([32]) *For all A, B: if S_1 is a substitution such that $S_1\,A = S_1\,B$ (so then S_1 is a unifier of A and B), then there exist substitutions S_2 and S_3 such that $S_2 = \mathit{unify}\ A\ B$ and $S_1 = S_3 \circ S_2$.*

Unification is associative and commutative: we can show that, for all types A, B, and C

$$\begin{array}{rcl} \mathit{unify}\ ((\mathit{unify}\ A\ B)\ A)\ C & = & \mathit{unify}\ ((\mathit{unify}\ A\ C)\ A)\ B \\ & = & \mathit{unify}\ ((\mathit{unify}\ B\ C)\ B)\ A \\ & = & \mathit{unify}\ A\ ((\mathit{unify}\ B\ C)\ B) \quad \mathit{etc.} \end{array}$$

which justifies a 'higher-order notation'; we will write $\mathit{unify}\ A\ B\ C\ \ldots$ for the unification of any number of types and will use the same higher-order notation for *unifyC*.

In the literature, Milner's algorithm $\mathcal{W}$ [23] is the usual point of reference for the principal typing property; however, mainly since designed to deal with the polymorphic `let` construct,[3] there contexts are supposed to have types for all term variables which might get updated by the running of $\mathcal{W}$. In a system without `let`, we can use the more intuitive 'divide and conquer' approach; we type subterms in an application separately, and freshly construct the contexts needed; we then need to unify those, making sure they map each term variable to at most one type.

Definition 1.5 (pt_{C}, PRINCIPAL TYPING ALGORITHM FOR $\vdash_{\mathrm{C}}$) The principal typing algorithm for Curry's system is given by:

$$\begin{array}{lcl} pt_{\mathrm{C}}\,x & = & \langle x{:}\varphi; \varphi \rangle \\ & & \text{where } \varphi \textit{ is fresh} \\ pt_{\mathrm{C}}\,(\lambda x.M) & = & \langle \Gamma; C \rangle \\ & & \text{where } \langle \Gamma'; B \rangle = pt_{\mathrm{C}}\,M \\ & & \Gamma, C = \begin{cases} \Gamma' \backslash x,\ A \to B & (x{:}A \in \Gamma') \\ \Gamma',\ \varphi \to B & (x \notin \Gamma') \end{cases} \\ & & \varphi \textit{ is fresh} \end{array}$$

$$\begin{array}{lcl} pt_{\mathrm{C}}\,(MN) & = & S_2 \circ S_1\, \langle \Gamma_1 \cup \Gamma_2; \varphi \rangle \\ & & \text{where } \langle \Gamma_1; A \rangle = pt_{\mathrm{C}}\,M \\ & & \langle \Gamma_2; B \rangle = pt_{\mathrm{C}}\,N \\ & & S_1 = \mathit{unify}\ A\ B \to \varphi \\ & & S_2 = \mathit{unifyC}\ (S_1\,\Gamma_1)\ (S_1\,\Gamma_2) \\ & & \varphi \textit{ is fresh} \end{array}$$

The algorithm as presented here is not purely functional. The 0-ary function *fresh* is supposed to return a new, unused type variable. It is obvious that such a function is not referential transparent, but for the sake of readability, we prefer not to be explicit on the handling of type variables.

The following property can be shown to hold.

Theorem 1.6 (PRINCIPAL TYPING PROPERTY) *If $\Gamma \vdash_{\mathrm{C}} M : A$, then there are context Γ', type B and a substitution S such that: $pt_{\mathrm{C}}\,M = \langle \Gamma'; B \rangle$, and both $S\Gamma' \subseteq \Gamma$ and $SB = A$.*

Proof: By induction on the structure of terms, much like that of Thm. 7.4. □

1.2 The $\lambda\mu$-calculus

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. We present the variant of $\lambda\mu$ we consider in this paper, as defined by Parigot in [29].

Definition 1.7 (SYNTAX OF $\lambda\mu$) The *$\lambda\mu$-terms* we consider are defined by the grammar:

[3] When typing ML's let$x := E_1$inE_2, 'divide and conquer' would not work: in order to type E_2, we need to have access to the (polymorphic) type for x, and cannot construct it as done in Def. 1.5. In order to correctly instantiate the polymorphic type found for E_1 when typing E_2, $\mathcal{W}$ first types E_1, quantifies the type found as much as possible and assigns that type to x when typing E_2. This implies that, when encountering a term variable, $\mathcal{W}$ uses its type from the context rather than creating it fresh. Unification is not defined on quantified types.

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

Recognising both λ and μ as binders, the notion of free and bound names and variables is defined as usual. As in Haskell [22], we will use '_' as a special name: when we write $\mu\_.[\alpha]M$, the name '_' cannot occur (free) in $[\alpha]M$. We will call the pseudo-terms of the shape $[\alpha]M$ *commands*, written 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 (instantaneous) substitution of all occurrences of x in M by N. Two kinds of structural substitution are defined: 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 CBV 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 1.8 (STRUCTURAL SUBSTITUTION) *Right-structural substitution*, $M\{N{\cdot}\gamma/\alpha\}$, and *left-structural substitution*, $\{N{\cdot}\gamma/\alpha\}M$, are defined inductively over pseudo terms by:

$$\begin{array}{rl@{\qquad}ll}
x\{N{\cdot}\gamma/\alpha\} & \triangleq x & \{N{\cdot}\gamma/\alpha\}x & \triangleq x \\
(\lambda x.M)\{N{\cdot}\gamma/\alpha\} & \triangleq \lambda x.(M\{N{\cdot}\gamma/\alpha\}) & \{N{\cdot}\gamma/\alpha\}(\lambda x.M) & \triangleq \lambda x.(\{N{\cdot}\gamma/\alpha\}M) \\
(PQ)\{N{\cdot}\gamma/\alpha\} & \triangleq P\{N{\cdot}\gamma/\alpha\}\ Q\{N{\cdot}\gamma/\alpha\} & \{N{\cdot}\gamma/\alpha\}(PQ) & \triangleq \{N{\cdot}\gamma/\alpha\}P\ \{N{\cdot}\gamma/\alpha\}Q \\
[\alpha]M\{N{\cdot}\gamma/\alpha\} & \triangleq [\gamma](M\{N{\cdot}\gamma/\alpha\}N) & \{N{\cdot}\gamma/\alpha\}[\alpha]M & \triangleq [\gamma]N(\{N{\cdot}\gamma/\alpha\}M) \\
[\beta]M\{N{\cdot}\gamma/\alpha\} & \triangleq [\beta](M\{N{\cdot}\gamma/\alpha\})\ (\beta \neq \alpha) & \{N{\cdot}\gamma/\alpha\}[\beta]M & \triangleq [\beta]\{N{\cdot}\gamma/\alpha\}M\ (\beta \neq \alpha) \\
(\mu\delta.\mathrm{C})\{N{\cdot}\gamma/\alpha\} & \triangleq \mu\delta.(\mathrm{C}\{N{\cdot}\gamma/\alpha\}) & \{N{\cdot}\gamma/\alpha\}\mu\delta.\mathrm{C} & \triangleq \mu\delta.\{N{\cdot}\gamma/\alpha\}\mathrm{C}
\end{array}$$

[28] 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 of contexts in [27].

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 [27].

Definition 1.9 ($\lambda\mu$ REDUCTION) *i*) The reduction rules of $\lambda\mu$ are:

$$\begin{array}{rrcll}
\textit{logical } (\beta): & (\lambda x.M)N & \to & M\{N/x\} & \\
\textit{structural } (\mu): & (\mu\alpha.\mathrm{C})N & \to & \mu\gamma.\mathrm{C}\{N{\cdot}\gamma/\alpha\} & (\gamma \textit{ fresh})^4 \\
\textit{erasing } (E): & \mu\alpha.[\alpha]M & \to & M & (\alpha \notin M) \\
\textit{renaming } (R): & \mu\alpha.[\beta]\mu\gamma.[\delta]M & \to & \mu\alpha.([\delta]M)\{\beta/\gamma\} &
\end{array}$$

Contexts are defined by:

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

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

ii) CBN *evaluation contexts* are defined as:

$$\mathsf{C}_{\mathbf{N}} ::= [\,] \mid \mathsf{C}_{\mathbf{N}}M \mid \mu\alpha.[\beta]\mathsf{C}_{\mathbf{N}}$$

and each $\mathsf{C}_{\mathbf{N}}$ is redex-free. CBN reduction $\to^{\mathbf{N}}_{\beta\mu}$ is defined through: $\mathsf{C}_{\mathbf{N}}[M] \to^{\mathbf{N}}_{\beta\mu} \mathsf{C}_{\mathbf{N}}[N]$ if $M \to N$ using either the β, μ, E, or R-reduction rule.

iii) CBV *evaluation contexts* are defined through:

^{4} In the literature, it is common to write $(\mu\alpha.\mathrm{C})N \to \mu\alpha.\mathrm{C}\{N/\alpha\}$ for the structural rule. This notation not only violates the spirit of Barendregt's convention, it also hides the fact that the bound α on the left and right are different: when dealing with typeable terms, their types differ (respectively $A \to B$ and B).

$$\mathsf{C}_{\mathsf{V}} ::= [\,] \mid \mathsf{C}_{\mathsf{V}} M \mid V \mathsf{C}_{\mathsf{V}} \mid \mu\alpha.[\beta]\mathsf{C}_{\mathsf{V}}$$

and each C_{V} is redex-free. CBV reduction $\rightarrow^{\mathrm{v}}_{\beta\mu}$ is defined through: $\mathsf{C}_{\mathsf{V}}[M] \rightarrow^{\mathrm{v}}_{\beta\mu} \mathsf{C}_{\mathsf{V}}[N]$ if $M \rightarrow N$ using either rule μ, E, R, or one of the rules:

$$\begin{array}{rrcll}
(\beta_{\mathrm{V}}): & (\lambda x.M)V & \rightarrow & M\{V/x\} & \\
(\mu_{\mathrm{V}}): & V(\mu\alpha.\mathrm{C}) & \rightarrow & \mu\gamma.\{V{\cdot}\gamma/\alpha\}\mathrm{C} & (\gamma \textit{ fresh})
\end{array}$$

iv) CBN *applicative* contexts are defined as:

$$\mathsf{C}^{\mathsf{A}}_{\mathsf{N}} ::= [\,] \mid \mathsf{C}^{\mathsf{A}}_{\mathsf{N}} M$$

whereas CBV applicative contexts are defined as:

$$\mathsf{C}^{\mathsf{A}}_{\mathsf{V}} ::= [\,] \mid \mathsf{C}^{\mathsf{A}}_{\mathsf{V}} M \mid V \mathsf{C}^{\mathsf{A}}_{\mathsf{V}}$$

Notice that we now need to demand that a CBN or CBV evaluation context is redex-free; otherwise, $(\mu\alpha.[\beta](\lambda x.M)N)P$ is ambiguous. This is not needed for CBN or CBV applicative contexts, as argued above in Def. 1.1.

Remark that, for rule (μ_{V}), $\mu\alpha.[\beta]N$ is not a value. Also, unlike for the λ-calculus, CBV reduction is not a sub-reduction system of $\rightarrow_{\beta\mu}$: the rule (μ_{V}) (and left-structural substitution) are not part of $\rightarrow_{\beta\mu}$. Both CBN and CBV constitute *reduction strategies* in that they pick exactly one free $\beta\mu$-redex to contract; notice that a term might be in either CBN or CBV-normal form (*i.e.* reduction has stopped), but not need be that for $\rightarrow_{\beta\mu}$.

It is possible to add the CBV-rules to $\lambda\mu$, and define

$$\begin{array}{rrcll}
(\mu_L): & (\mu\alpha.\mathrm{C})N & \rightarrow_{\beta\mu} & \mu\gamma.\mathrm{C}\{N{\cdot}\gamma/\alpha\} & (\gamma \textit{ fresh}) \\
(\mu_R): & M(\mu\alpha.\mathrm{C}) & \rightarrow_{\beta\mu} & \mu\gamma.\{M{\cdot}\gamma/\alpha\}\mathrm{C} & (\gamma \textit{ fresh})
\end{array}$$

but then reduction would no longer be confluent: it would then contain the critical pair $(\mu\alpha.\mathrm{C})\,(\mu\beta.\mathrm{C}')$, which can be contracted in two ways, with perhaps different outcomes.

As shown in [33], it is possible to define a domain theoretic model of $\lambda\mu$, based on the solution of the domain equations $D = C \rightarrow R$ and $C = D \times C$, where R is an arbitrary domain of 'results'. The domain C is set of what are called 'continuations' in [33], which are infinite tuples of elements in D, where D is the domain of continuous functions from C to R and is the set of 'denotations' of terms. Using the strong relation between intersection type theories and domain constructions, based on those results, [4] defined an intersection type assignment system for $\lambda\mu$ that induces a filter model [6] for $\lambda\mu$ where continuations are typed with sequence of term types. This approach was then used in [2], but limiting the set of results R to just a singleton set, thus effectively obtaining 'negated intersection types'. For that system, strong normalisation of cut elimination is shown, from which a number of other normalisation and characterisation results follow.

1.3 Type assignment for $\lambda\mu$ and soundness

With $\lambda\mu$ Parigot created a multi-conclusion typing system which corresponds to implicative classical logic with *focus*; there derivable statements have the 'sequent' shape $\Gamma \vdash M : A \mid \Delta$, where A is the main conclusion of the statement, expressed as the *active* conclusion, Γ is the set of assumptions and Δ is the set of alternative conclusions, or have the shape $\Gamma \vdash \Delta$ is there is no formula under focus. The formulas and inference rules for this system are:

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

and a context Γ is a set of formulas,[5] where $\Gamma, A = \Gamma \cup \{A\}$ and

[5] Treating a contexts as a list would require additional inference rules for context manipulation.

$$(Ax):\ \frac{}{\Gamma,A \vdash A \mid \Delta} \qquad (Act):\ \frac{\Gamma \vdash \ \mid A,\Delta}{\Gamma \vdash A \mid \Delta} \qquad (Pass):\ \frac{\Gamma \vdash A \mid A,\Delta}{\Gamma \vdash \ \mid A,\Delta}$$

$$(\to I):\ \frac{\Gamma,A \vdash B \mid \Delta}{\Gamma \vdash A\to B \mid \Delta} \qquad (\to E):\ \frac{\Gamma \vdash A\to B \mid \Delta \quad \Gamma \vdash A \mid \Delta}{\Gamma \vdash B \mid \Delta}$$

Since formulas in Δ can be seen as negated, the rule (Act) represents double negation elimination.

$\lambda\mu$ gives a Curry-Howard interpretation to the above inference rules, as will be clear from the definition of type assignment for $\lambda\mu$ as below; there is a *main*, or *active*, conclusion, labelled by a term, and the *alternative* conclusions are labelled by names α, β, *etc.*

Judgements in $\lambda\mu$ are of the shape $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$, where Δ consists of pairs of Greek characters (the *names*) and types; the left-hand context Γ, as for the λ-calculus, contains pairs of Roman characters and types, and represents the types of the free term variables of M.

Definition 1.10 (Typing rules for $\lambda\mu$) *i*) Types and variable contexts Γ are those of Def. 1.2.

ii) 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.

iii) The type assignment rules for $\lambda\mu$, adapted to our notation, are:

$$(Ax):\ \frac{}{\Gamma,x{:}A \vdash x : A \mid \Delta} \quad (\mu):\ \frac{\Gamma \vdash M : B \mid \alpha{:}A,\beta{:}B,\Delta}{\Gamma \vdash \mu\alpha.[\beta]M : A \mid \beta{:}B,\Delta}\ (\alpha \notin \Delta) \quad \frac{\Gamma \vdash M : A \mid \alpha{:}A,\Delta}{\Gamma \vdash \mu\alpha.[\alpha]M : A \mid \Delta}\ (\alpha \notin \Delta)$$

$$(\to I):\ \frac{\Gamma,x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A\to B \mid \Delta}\ (x \notin \Gamma) \qquad (\to E):\ \frac{\Gamma \vdash M : A\to B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta}$$

We will write $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$ for statements derivable in this system.

iv) We extend Barendregt's convention on free and bound variables and names to judgements (for all the notions of type assignment we define here), so in $\Gamma,x{:}A \vdash_{\lambda\mu} M : B \mid \alpha{:}C,\Delta$, both x and α cannot appear bound in M.

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

As with Implicative Intuitionistic Logic and the Lambda Calculus, the reduction rules 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, where elimination takes place for a type constructor that appears in one of the alternative conclusions (the Greek variable is the name given to a subterm): he therefore needs to express that the focus of the derivation (proof) changes (see the rules in Def. 1.10), and this is achieved by extending the syntax with two new constructs $[\alpha]M$ and $\mu\alpha.M$ that act as witness to *deactivation* and *activation*, which together move the focus of the derivation, and together are called a *context switch*.

The intuition behind the structural rule is given by de Groote [19]: *"in a $\lambda\mu$-term $\mu\alpha.M$ of type $A\to B$, only the subterms named by α are* really *of type $A\to B$ (...); hence, when such a μ-abstraction is applied to an argument, this argument must be passed over to the sub-terms named by α."* This can be illustrated by the derivations for the reduction $(\mu\alpha.[\beta]\mathsf{C}[\mu\gamma.[\alpha]M])N \to_{\beta\mu} \mu\delta.[\beta]\mathsf{C}[\mu\gamma.[\delta]MN]$ in Fig. 1 (where $\beta{:}C \in \Delta$).

We have the following standard result.

Lemma 1.11 (Weakening and thinning for $\vdash_{\lambda\mu}$) *The following rules are admissible for $\vdash_{\lambda\mu}$:*

$$(Wk):\ \frac{\Gamma \vdash M : A \mid \Delta}{\Gamma' \vdash M : A \mid \Delta'}\ (\Gamma \subseteq \Gamma', \Delta \subseteq \Delta') \qquad (Th):\ \frac{\Gamma \vdash M : A \mid \Delta}{\Gamma' \vdash M : A \mid \Delta'}\ \begin{array}{l}(\Gamma' = \{x{:}B \in \Gamma \mid x \in fv(M)\},\\ \ \Delta' = \{\mathsf{n}{:}B \in \Delta \mid \mathsf{n} \in fn(M)\})\end{array}$$

$$\dfrac{\dfrac{\boxed{}}{\Gamma \vdash M : A{\to}B \mid \alpha{:}A{\to}B,\gamma{:}D,\Delta}}{\Gamma \vdash \mu\gamma.[\alpha]M : D \mid \alpha{:}A{\to}B,\Delta}\,(\mu)
\qquad
\dfrac{\dfrac{\dfrac{\boxed{}}{\Gamma \vdash M : A{\to}B \mid \delta{:}B,\gamma{:}D,\Delta} \quad \dfrac{\dfrac{\boxed{}}{\Gamma \vdash N : A \mid \Delta}}{\Gamma \vdash N : A \mid \delta{:}B,\gamma{:}D,\Delta}\,(Wk)}{\Gamma \vdash MN : B \mid \delta{:}B,\gamma{:}D,\Delta}\,(\to E)}{\Gamma \vdash \mu\gamma.[\delta]MN : D \mid \delta{:}B,\Delta}\,(\mu)$$

$$\dfrac{\dfrac{\dfrac{\boxed{}}{\Gamma \vdash \mathsf{C}[\mu\gamma.[\alpha]M] : C \mid \alpha{:}A{\to}B,\Delta}}{\Gamma \vdash \mu\alpha.[\beta]\mathsf{C}[\mu\gamma.[\alpha]M] : A{\to}B \mid \Delta}\,(\mu) \quad \dfrac{\boxed{}}{\Gamma \vdash N : A \mid \Delta}}{\Gamma \vdash (\mu\alpha.[\beta]\mathsf{C}[\mu\gamma.[\alpha]M])N : B \mid \Delta}\,(\to E)
\qquad
\dfrac{\dfrac{\boxed{}}{\Gamma \vdash \mathsf{C}[\mu\gamma.[\delta]MN] : C \mid \delta{:}B,\Delta}}{\Gamma \vdash \mu\delta.[\beta]\mathsf{C}[\mu\gamma.[\delta]MN] : B \mid \Delta}\,(\mu)$$

Figure 1. An illustration of structural reduction in $\lambda\mu$.

Proof: Standard. □

Notice that, by our extension of Barendregt's convention in Def. 1.10, Γ' and Δ' cannot contain statements for the bound names and variables in M.

Example 1.12 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, $\mu\alpha.[\alpha]\mu\beta.[\gamma]M \to_{\beta\mu} \mu\alpha.[\gamma]M\{\alpha/\beta\} = \mu\alpha.[\gamma]M$ but also, by erasure, $\mu\alpha.[\alpha]\mu\beta.[\gamma]M \to_{\beta\mu} \mu\beta.[\gamma]M$. Notice that $\mu\alpha.[\gamma]M =_\alpha \mu\_.[\gamma]M =_\alpha \mu\beta.[\gamma]M$.

We will now show that type assignment is closed under reduction for both CBN and CBV reduction. This result might itself be as expected, and is presented here mostly for completeness. First we show results for the three notions of term substitution.

Lemma 1.13 (SUBSTITUTION LEMMA) *i) If $\Gamma,x{:}B \vdash_{\lambda\mu} M : A \mid \Delta$ and $\Gamma \vdash_{\lambda\mu} L : B \mid \Delta$, then $\Gamma \vdash_{\lambda\mu} M\{L/x\} : A \mid \Delta$.*

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

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

Proof: *i*) By induction on the definition of term substitution.

ii) By induction on the definition of right-structural substitution.

($\mu\delta.[\alpha]N\{L{\cdot}\gamma/\alpha\} \triangleq \mu\delta.[\gamma](N\{L{\cdot}\gamma/\alpha\}L)$): Then by rule (μ) $\Gamma \vdash_{\lambda\mu} N : B\to C \mid \delta{:}A,\alpha{:}B\to C,\Delta$, and by induction $\Gamma \vdash_{\lambda\mu} N\{L{\cdot}\gamma/\alpha\} : B\to C \mid \delta{:}A,\gamma{:}C,\Delta$. Since α, δ and γ all do not occur (free) in N, we can construct

$$\dfrac{\dfrac{\dfrac{\boxed{}}{\Gamma \vdash N\{L{\cdot}\gamma/\alpha\} : B\to C \mid \delta{:}A,\gamma{:}C,\Delta} \quad \dfrac{\dfrac{\boxed{}}{\Gamma \vdash L : B \mid \Delta}}{\Gamma \vdash L : B \mid \delta{:}A,\gamma{:}C,\Delta}\,(Wk)}{\Gamma \vdash (N\{L{\cdot}\gamma/\alpha\})L : C \mid \delta{:}A,\gamma{:}C,\Delta}\,(\to E)}{\Gamma \vdash \mu\delta.[\gamma](N\{L{\cdot}\gamma/\alpha\})L : A \mid \gamma{:}C,\Delta}\,(\mu)$$

($(\mu\delta.[\beta]N)\{L{\cdot}\gamma/\alpha\} \triangleq \mu\delta.[\beta](N\{L{\cdot}\gamma/\alpha\})$ $(\beta \neq \alpha)$): Then by rule (μ) there exists D such that $\Delta = \beta{:}D,\Delta'$, and $\Gamma \vdash_{\lambda\mu} N : D \mid \delta{:}A,\beta{:}D,\alpha{:}B\to C,\Delta'$, and by induction $\Gamma \vdash_{\lambda\mu} N\{L{\cdot}\gamma/\alpha\} : D \mid \delta{:}A,\beta{:}D,\gamma{:}C,\Delta'$. But then, by rule (μ), also $\mu\delta.[\beta]N\{L{\cdot}\gamma/\alpha\} : A : \Gamma \vdash_{\lambda\mu} \beta{:}D,\gamma{:}C,\Delta'$.

iii) By induction on the definition of left-structural substitution.

($\{L{\cdot}\gamma/\alpha\}\mu\delta.[\alpha]N \triangleq \mu\delta.[\gamma]L(\{L{\cdot}\gamma/\alpha\}N)$): Then by rule (μ) $\Gamma \vdash_{\lambda\mu} N : B \mid \delta{:}A,\alpha{:}B,\Delta$, and by induction $\Gamma \vdash_{\lambda\mu} \{L{\cdot}\gamma/\alpha\}N : B \mid \delta{:}A,\gamma{:}C,\Delta$. Since δ and γ do not occur (free) in L, we can construct

$$\dfrac{\dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash L : B \to C \mid \Delta\end{array}}{\Gamma \vdash L : B \to C \mid \delta{:}A,\gamma{:}C,\Delta}\,(Wk) \qquad \begin{array}{c}\boxed{\phantom{\Gamma \vdash \{L\cdot\gamma/\alpha\}N : B \mid \delta}}\\ \Gamma \vdash \{L\cdot\gamma/\alpha\}N : B \mid \delta{:}A,\gamma{:}C,\Delta\end{array}}{\dfrac{\Gamma \vdash L\{L\cdot\gamma/\alpha\}N : C \mid \delta{:}A,\gamma{:}C,\Delta}{\Gamma \vdash \mu\delta.[\gamma]L\{L\cdot\gamma/\alpha\}N : A \mid \gamma{:}C,\Delta}\,(\mu)}\,(\to E)$$

($\{L\cdot\gamma/\alpha\}(\mu\delta.[\beta]N) \triangleq \mu\delta.[\beta](\{L\cdot\gamma/\alpha\}N)$ $(\beta \neq \alpha)$): Then by rule (μ) there exists D such that $\beta{:}D,\Delta' = \Delta$, and $\Gamma \vdash_{\lambda\mu} N : D \mid \delta{:}A,\alpha{:}B,\beta{:}D,\Delta'$. Then by induction we have $\Gamma \vdash_{\lambda\mu} \{L\cdot\gamma/\alpha\}N : D \mid \delta{:}A,\gamma{:}C,\beta{:}D,\Delta'$. But then, by rule (μ), also $\Gamma \vdash_{\lambda\mu} \mu\delta.[\beta]\{L\cdot\gamma/\alpha\}N : A \mid \gamma{:}C,\beta{:}D,\Delta'$. □

We will now show that type assignment respects CBN and CBV reduction:

Theorem 1.14 *i) If $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$, and $M \to^{\mathrm{N}}_{\beta\mu} N$, then $\Gamma \vdash_{\lambda\mu} N : A \mid \Delta$.*
ii) If $M \to^{\mathrm{V}}_{\beta\mu} N$, and $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$, then $\Gamma \vdash_{\lambda\mu} N : A \mid \Delta$.

Proof: *i*) By induction on the definition of $\to^{\mathrm{N}}_{\beta\mu}$.

($(\lambda x.M)N \to^{\mathrm{N}}_{\beta\mu} M\{N/x\}$): The derivation for $\Gamma \vdash_{\lambda\mu} (\lambda x.M)N : A \mid \Delta$ is shaped like

$$\dfrac{\dfrac{\begin{array}{c}\boxed{\phantom{\Gamma,x{:}B \vdash M : A}}\\ \Gamma,x{:}B \vdash M : A \mid \Delta\end{array}}{\Gamma \vdash \lambda x.M : B \to A \mid \Delta}\,(\to I) \qquad \begin{array}{c}\boxed{}\\ \Gamma \vdash N : B \mid \Delta\end{array}}{\Gamma \vdash (\lambda x.M)N : A \mid \Delta}\,(\to E)$$

Then, by Lem. 1.13, we have $\Gamma \vdash_{\lambda\mu} M\{N/x\} : A \mid \Delta$.

($(\mu\alpha.[\alpha]M)N \to^{\mathrm{N}}_{\beta\mu} \mu\gamma.[\gamma]M\{N\cdot\gamma/\alpha\}N$): The derivation for $(\mu\alpha.[\alpha]M)N$ is shaped like

$$\dfrac{\dfrac{\begin{array}{c}\boxed{\phantom{\Gamma \vdash M : B \to A \mid \alpha{:}B \to A,\Delta}}\\ \Gamma \vdash M : B \to A \mid \alpha{:}B \to A,\Delta\end{array}}{\Gamma \vdash \mu\alpha.[\alpha]M : B \to A \mid \Delta}\,(\mu) \qquad \begin{array}{c}\boxed{}\\ \Gamma \vdash N : B \mid \Delta\end{array}}{\Gamma \vdash (\mu\alpha.[\alpha]M)N : A \mid \Delta}\,(\to E)$$

Then by Lem. 1.13, we have $\Gamma \vdash_{\lambda\mu} M\{N\cdot\gamma/\alpha\} : B \to A \mid \gamma{:}A,\Delta$. Since γ is fresh, by weakening also $\Gamma \vdash_{\lambda\mu} N : B \mid \gamma{:}A,\Delta$, and we can construct

$$\dfrac{\begin{array}{c}\boxed{\phantom{\Gamma \vdash M\{N\cdot\gamma/\alpha\} : B \to A \mid \gamma}}\\ \Gamma \vdash M\{N\cdot\gamma/\alpha\} : B \to A \mid \gamma{:}A,\Delta\end{array} \qquad \begin{array}{c}\boxed{}\\ \Gamma \vdash N : B \mid \gamma{:}A,\Delta\end{array}}{\dfrac{\Gamma \vdash M\{N\cdot\gamma/\alpha\}N : A \mid \gamma{:}A,\Delta}{\Gamma \vdash \mu\gamma.[\gamma]M\{N\cdot\gamma/\alpha\}N : A \mid \Delta}\,(\mu)}\,(\to E)$$

($(\mu\alpha.[\delta]M)N \to^{\mathrm{N}}_{\beta\mu} \mu\gamma.[\delta]M\{N\cdot\gamma/\alpha\}$, *with* $\alpha \neq \delta$): The derivation for $(\mu\alpha.[\delta]M)N$ is shaped like

$$\dfrac{\dfrac{\begin{array}{c}\boxed{\phantom{\Gamma \vdash M : C \mid \alpha{:}B \to A,\delta{:}C}}\\ \Gamma \vdash M : C \mid \alpha{:}B \to A,\delta{:}C,\Delta'\end{array}}{\Gamma \vdash \mu\alpha.[\delta]M : B \to A \mid \delta{:}C,\Delta'}\,(\mu) \qquad \begin{array}{c}\boxed{\phantom{\Gamma \vdash N : B \mid \delta{:}C}}\\ \Gamma \vdash N : B \mid \delta{:}C,\Delta'\end{array}}{\Gamma \vdash (\mu\alpha.[\delta]M)N : A \mid \delta{:}C,\Delta'}\,(\to E)$$

with $\Delta = \delta{:}C,\Delta'$. Then by Lem. 1.13, we have $\Gamma \vdash_{\lambda\mu} M\{N\cdot\gamma/\alpha\} : C \mid \gamma{:}A,\delta{:}C,\Delta'$, and we can construct

$$\dfrac{\begin{array}{c}\boxed{\phantom{\Gamma \vdash M\{N\cdot\gamma/\alpha\} : C \mid \gamma{:}A,\delta}}\\ \Gamma \vdash M\{N\cdot\gamma/\alpha\} : C \mid \gamma{:}A,\delta{:}C,\Delta'\end{array}}{\Gamma \vdash \mu\gamma.[\delta]M\{N\cdot\gamma/\alpha\} : A \mid \delta{:}C,\Delta'}\,(\mu)$$

($\mu\alpha.[\alpha]M \to^{\mathrm{N}}_{\beta\mu} M$): The derivation for $\mu\alpha.[\alpha]M$ is shaped like

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash M : A \mid \alpha{:}A,\Delta\end{array}}{\Gamma \vdash \mu\alpha.[\alpha]M : A \mid \Delta}\,(\mu)$$

Since α does not occur in M, we can thin $\alpha{:}A,\Delta$ and obtain $\Gamma \vdash_{\lambda\mu} M : A \mid \Delta$.

$(\mu\alpha.[\beta]\mu\gamma.[\delta]M \rightarrow^{\mathrm{N}}_{\beta\mu} \mu\alpha.([\delta]M)\{\beta/\gamma\})$: The derivation for $(\mu\alpha.[\delta]M)N$ is shaped like

$$\dfrac{\dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash M : D \mid \alpha{:}A,\beta{:}B,\gamma{:}B,\delta{:}D,\Delta'\end{array}}{\Gamma \vdash \mu\gamma.[\delta]M : B \mid \alpha{:}A,\beta{:}B,\delta{:}D,\Delta'}\,(\mu)}{\Gamma \vdash \mu\alpha.[\beta]\mu\gamma.[\delta]M : A \mid \beta{:}B,\delta{:}D,\Delta'}\,(\mu)$$

So in particular, replacing all occurrences of γ by β, we obtain a derivation for $\Gamma \vdash_{\lambda\mu} M\{\beta/\gamma\} : D \mid \alpha{:}A,\beta{:}B,\delta{:}D,\Delta'$. Now either:

$(\delta \neq \gamma)$: Then we can construct:

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash M\{\beta/\gamma\} : D \mid \alpha{:}A,\beta{:}B,\delta{:}D,\Delta'\end{array}}{\Gamma \vdash \mu\alpha.[\delta]M\{\beta/\gamma\} : A \mid \beta{:}B,\delta{:}D,\Delta'}\,(\mu)$$

$(\delta = \gamma)$: Then $D = B$ as well, and we can construct:

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash M\{\beta/\gamma\} : B \mid \alpha{:}A,\beta{:}B,\Delta'\end{array}}{\Gamma \vdash \mu\alpha.[\beta]M\{\beta/\gamma\} : A \mid \beta{:}B,\Delta'}\,(\mu)$$

$(M \rightarrow^{\mathrm{N}}_{\beta\mu} N \Rightarrow MP \rightarrow^{\mathrm{N}}_{\beta\mu} NP,\ \mu\alpha.[\beta]M \rightarrow^{\mathrm{N}}_{\beta\mu} \mu\alpha.[\beta]N)$: By induction. $\square$

ii) By induction on the definition of $\rightarrow^{\mathrm{V}}_{\beta\mu}$.

$(V(\mu\alpha.[\alpha]M) \rightarrow^{\mathrm{V}}_{\beta\mu} \mu\gamma.[\gamma]V\{V{\cdot}\gamma/\alpha\}M)$: The derivation for $V(\mu\alpha.[\alpha]M)$ is shaped like

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash V : B \rightarrow A \mid \Delta\end{array} \qquad \dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash M : B \mid \alpha{:}B,\Delta\end{array}}{\Gamma \vdash \mu\alpha.[\alpha]M : B \mid \Delta}\,(\mu)}{\Gamma \vdash V(\mu\alpha.[\alpha]M) : A \mid \Delta}\,(\rightarrow E)$$

Then by Lem. 1.13, we have $\Gamma \vdash_{\lambda\mu} \{V{\cdot}\gamma/\alpha\}M : B \mid \gamma{:}A,\Delta$, and we can construct

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash V : B \rightarrow A \mid \Delta\end{array}}{\Gamma \vdash V : B \rightarrow A \mid \gamma{:}A,\Delta}\,(Wk) \qquad \begin{array}{c}\boxed{}\\ \Gamma \vdash \{V{\cdot}\gamma/\alpha\}M : B \mid \gamma{:}A,\Delta\end{array}}{\Gamma \vdash V\{V{\cdot}\gamma/\alpha\}M : A \mid \gamma{:}A,\Delta}\,(\rightarrow E)}{\Gamma \vdash \mu\gamma.[\gamma]V\{V{\cdot}\gamma/\alpha\}M : A \mid \Delta}\,(\mu)$$

$(V(\mu\alpha.[\delta]M) \rightarrow^{\mathrm{V}}_{\beta\mu} \mu\gamma.[\delta]\{V{\cdot}\gamma/\alpha\}M$, *with* $\alpha \neq \delta)$: The derivation for $V(\mu\alpha.[\delta]M)$ is shaped like

$$\dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash V : B \rightarrow A \mid \delta{:}C,\Delta'\end{array} \qquad \dfrac{\begin{array}{c}\boxed{}\\ \Gamma \vdash M : C \mid \alpha{:}B,\delta{:}C,\Delta'\end{array}}{\Gamma \vdash \mu\alpha.[\delta]M : B \mid \delta{:}C,\Delta'}\,(\mu)}{\Gamma \vdash V(\mu\alpha.[\delta]M) : A \mid \delta{:}C,\Delta'}\,(\rightarrow E)$$

with $\Delta = \delta{:}C,\Delta'$. Then by Lem. 1.13, we have $\Gamma \vdash_{\lambda\mu} \{V{\cdot}\gamma/\alpha\}M : C \mid \gamma{:}A,\delta{:}C,\Delta'$, and by rule (μ) we have $\mu\gamma.[\delta]\{V{\cdot}\gamma/\alpha\}M : A : \Gamma \vdash_{\lambda\mu} \delta{:}C,\Delta'$. $\square$

Example 1.15 In [28], Parigot represents 'double negation elimination' in $\lambda\mu$ through the term $\lambda y.\mu\alpha.[\gamma]y(\lambda x.\mu\delta.[\alpha]x)$; notice that this term is not closed as it has a free name γ. To type it, rather than adding negation, $\perp$ is added as a pseudo-type to express negation as well as

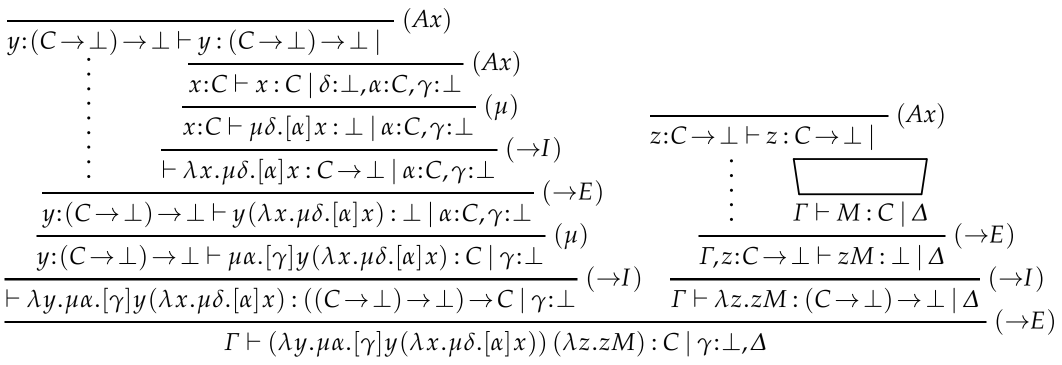
(Ax)
y:(C→⊥)→⊥ ⊢ y : (C→⊥)→⊥ |
(Ax)
x:C ⊢ x : C | δ:⊥,α:C,γ:⊥
(μ)
x:C ⊢ μδ.[α]x : ⊥ | α:C,γ:⊥
(→I)
⊢ λx.μδ.[α]x : C→⊥ | α:C,γ:⊥
(→E)
y:(C→⊥)→⊥ ⊢ y(λx.μδ.[α]x) : ⊥ | α:C,γ:⊥
(μ)
y:(C→⊥)→⊥ ⊢ μα.[γ]y(λx.μδ.[α]x) : C | γ:⊥
(→I)
⊢ λy.μα.[γ]y(λx.μδ.[α]x) : ((C→⊥)→⊥)→C | γ:⊥
(Ax)
z:C→⊥ ⊢ z : C→⊥ |
Γ ⊢ M : C | Δ
(→E)
Γ,z:C→⊥ ⊢ zM : ⊥ | Δ
(→I)
Γ ⊢ λz.zM : (C→⊥)→⊥ | Δ
(→E)
Γ ⊢ (λy.μα.[γ]y(λx.μδ.[α]x)) (λz.zM) : C | γ:⊥,Δ

Figure 2. Modeling double negation elimination in $\lambda\mu$

contradiction. Assuming $z \notin M$, we can construct the derivation in Fig. 2 (using multiplicative style); notice that γ is of type $\bot$. Since $\delta \notin [\alpha]M$ and $\alpha \notin M$ by Barendregt's convention, we have:

$$\begin{array}{lll}(\lambda y.\mu\alpha.[\gamma]y(\lambda x.\mu\delta.[\alpha]x))\,(\lambda z.zM) & \rightarrow^{\mathrm{N}}_{\beta\mu} & \mu\alpha.[\gamma]\,(\lambda z.zM)\,(\lambda x.\mu\delta.[\alpha]x) \;\rightarrow^{\mathrm{N}}_{\beta\mu} \\ \mu\alpha.[\gamma]\,(\lambda x.\mu\delta.[\alpha]x)M & \rightarrow^{\mathrm{N}}_{\beta\mu} & \mu\alpha.[\gamma]\mu\delta.[\alpha]M \;\rightarrow^{\mathrm{N}}_{\beta\mu} \\ \mu\alpha.[\alpha]M & \rightarrow^{\mathrm{N}}_{\beta\mu} & M\end{array}$$

which illustrates Thm. 1.14.

The following lemma is needed below when encoding throwing exceptions.

Lemma 1.16 i) The reduction rule $\mathsf{C}^{\mathrm{A}}_{\mathrm{N}}[\mu\_.[\beta]N] \rightarrow \mu\_.[\beta]N$ *is admissible in* $\rightarrow^{\mathrm{N}}_{\beta\mu}$.
ii) The reduction rule $\mathsf{C}^{\mathrm{A}}_{\mathrm{V}}[\mu\_.[\beta]N] \rightarrow \mu\_.[\beta]N$ *is admissible in* $\rightarrow^{\mathrm{V}}_{\beta\mu}$.

Proof: i) By induction on the structure of contexts:

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

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

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

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

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

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

□

1.4 The $\lambda\mu$-tp-calculus

Following [33], [1] also presents a variant of $\lambda\mu$, called $\lambda\mu$-tp, where tp is a name that cannot occur bound and denotes the top-level. It does that by adding tp as a name-constant, and the type assignment rule:

$$(\mathsf{tp}): \; \frac{\Gamma \vdash M : \bot \mid \Delta}{\Gamma \vdash [\mathsf{tp}]M : \bot \mid \Delta}$$

Notice that, implicitly, the assumption is that tp has type $\bot$, but that this connection is not registered in the co-context.

Their motivation for this extension is: *"On the programming calculi side, the presence of the continuation* tp *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 distinc-*

tion can be used to develop more refined programming calculi for languages with control operators." AHS'07. We will follow this suggestion below, when we look to model aborting computations in λ^{try} in Sect. 5.

Definition 1.17 ($\lambda\mu$-tp [1]) *i)* Terms of the $\lambda\mu$-tp-calculus are defined as in Def. 1.7, extended with the case $\mu\alpha.[\mathsf{tp}]M$, where tp is a name that cannot occur bound.

ii) The notion of type assignment for $\lambda\mu$-tp, $\vdash_{\mathsf{tp}}$, is defined using the types defined by the grammar

$$A,B \ ::= \ \varphi \mid \bot \mid A \to B$$

and the type assignment rules of Def. 1.10, extended with the inference rule

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

Notice that now $\bot$ is a type. The appropriate variants of Lem. 1.13 and Thm. 1.14 can much in the same way be shown to hold for $\vdash_{\mathsf{tp}}$. Notice that Lem. 1.16 also holds for $\beta = \mathsf{tp}$.

Example 1.18 In $\lambda\mu$-tp, 'double negation elimination' can be expressed using the closed term $\lambda y.\mu\alpha.[\mathsf{tp}](y\lambda x.\mu\delta.[\alpha]x)$; the structure of the derivation is exactly the same as that in Ex. 1.15, but for the fact that γ has been replaced by tp, and $\gamma{:}\bot$ has been removed from the co-context.

2 Calculi with exception handling and $\lambda\mu$

In this section we will look at the intricacies of adding exceptions to a λ-calculus, and how exceptions are dealt with in ML. We then look at Nakano's calculus λ^{N} [25] that tries to do that, and Crolard's interpretation of λ^{N} into $\lambda\mu$ [12], and de Groote's calculus $\lambda^{\rightarrow}_{exn}$ [20] and Bierman's interpretation of $\lambda^{\rightarrow}_{exn}$ in CBV-$\lambda\mu$ [8].

Exception handling is a common feature of quite a few programming languages. Exceptions are raised when a program reaches an undesirable state, and basically involves abandoning a computation, aborting the current context, and passing control to a dedicated handler, that is written to deal with the anomaly.

The kind of contexts that can be aborted differ from paper to paper. The most common approach, as used in [15, 8] and here, is to allow aborting *applicative* contexts only. For example, Nakano'94b allows aborting executions inside abstractions as well; allowing this kind of computational exit 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. 4.5 and 6.6). This leaves that (in the context of the λ-calculus with types) we can only safely abort from applicative contexts.

2.1 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 equipping 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

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

should not be allowed.[6] One reason is that subject reduction will then fail (the variable x might appear in N; see Ex. 4.5), 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 CBV or CBN 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 [25]).

• A common approach to typeing the throw action is to base its rule on the rule for $\bot$-elimination from Classical Logic [16],

$$(EFQ) : \frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

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

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

(See Def. 2.1). 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. 6). 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. 6 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, in the sense that when the type fail has been inferred, it is certain that the program will fail. When adding the conditional construct and allowing a more liberal way of typing it, 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 dy-

[6] This rule is implied in systems where throw is allowed to escape from *any* context, as in [25].

namic scoping,[7] 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

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

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.

2.2 Exceptions in ML

Exceptions were introduced for ML in [24], and the approach taken there forms the basis for that used in Haskell and OCaml [31], to name but two more recently developed functional programming languages. Exception handling for ML is defined in a rather ad-hoc way, since not really representable in a functional, λ-calculus way. This is caused by the very nature of exceptions that, when raised, propagate "outwards and upwards". In a formal calculus, this requires a reduction rule that is capable of eradicating contexts, like for example $\mathtt{C}[\mathcal{A}(M)] \to M$ as used in λC [15] (using the *abort* constructor $\mathcal{A}$), or similarly as in [25, 18]. Normally, contexts that are discarded are *applicative*, as in $(\mathcal{A}(M))N_1 \cdots N_n \to M$, and the applicative context has to be assumed to be maximal.

Then using raise as a keyword that models an exception occurring, we can say that we need a reduction rule of the shape 'raise $M \to$ raise' (discarding the context 'one term at the time'), which cannot be expressed in a pure functional language, not even using a fixed-point construction. Rather, exception handling in ML is dealt with as an implementation issue *outside* of the normal reduction strategy, by placing markers with the handlers, and unwinding the stack until the correct handler for the exception is reached.

Focussing for the moment on ML, exceptions have to be declared in ML through a statement of the form:

```
exception exceptionName of argumentsType
```

(`of` *`argumentsType`* is optional) where *`exceptionName`* is the declared identifier, a type constructor, and the added *`argumentsType`* is the type of the arguments that can be passed. Raising has the syntax

```
raise expression
```

where *`expression`* is (some program term that evaluates to) the exception to be raised.

Handling an exception is dealt with through `handle` expressions, that have the form

```
expression handle matches
```

where *`matches`* is one or more pattern rules of the shape "*`name => body`*" that match against declared exception names and executes *`body`* when successful, and *`expression`* is some program term that might raise an exception. This matching feature highlights that exception names are treated as *constructors*, that create objects of ML's built-in extensible data type `exn`.[8]

Raising an exception *`name`* involves an interruption of normal execution, and a jump out of the current context, unfolding the stack until a handler for *`name`* has been reached, or the outermost level is reached after which the run-time system deals with the event through

[7] In dynamic scoping, the compiler maintains a mapping from identifiers to values for each 'environment'. Then referencing an identifier produces what is associated to it in the most recent environment; the compiler first searches the current environment and then successively the surrounding ones, in order.

[8] Haskell has no extensible data type; rather user-defined exceptions are instances of the `Exception` type class.

printing an error message or the program terminates. This reduction behaviour is left rather unspecified in ML, but can be defined formally as done by de Groote (see Sect. 2.5) and as we will do in Definition 3.2 for our calculus λ^{try}; as mentioned above, it is not expressible in a pure λ-calculus.

The return type of an exception is always `exn`, and when the *`exceptionName`* is declared with *`argumentsType`*, then the type of *`exceptionName`* is *`argumentsType`* $\rightarrow$ `exn`. Therefore the argument passed to `raise` has type `exn`; the term `raise` *`expression`* given above has *any* type, since a `raise` expression should be useable in any context. Also, to guarantee subject reduction, the bodies of the handlers should all have the same type as the main expression.

A declaration like

```
exception exceptionName of argumentsType
```

is dealt with in a different way than a normal function declaration (which gets modelled through a `let`-construct) because *`exceptionName`* is not an identifier, but a type constructor. Informally, the declaration adds the statement

$$\mathit{exceptionName} : \mathit{argumentsType} \rightarrow \texttt{exn}$$

to an exceptions environment (called E here), mapping type constructors to types, which gets built up before typing the main term in the program.

This suggests inference rules like:

$$\frac{\Gamma; E \vdash \mathit{params}_i : \mathit{argType}_i}{\Gamma; E \vdash \mathit{name}_i(\mathit{params}_i) : \texttt{exn}} \qquad \frac{\Gamma; E \vdash \mathit{expr} : \texttt{exn}}{\Gamma; E \vdash \texttt{raise}\ \mathit{expr} : A}$$

$$\frac{\Gamma; E \vdash \mathit{expr} : A \quad \Gamma, x{:}\texttt{argType}_i; E \vdash \mathit{body}_i : A \ (\forall i \in \underline{n})}{\Gamma; E \vdash \mathit{expr}\ \texttt{handle}\ \overrightarrow{\mathit{name}_i\ (x)\ \texttt{=>}\ \mathit{body}_i} : A}$$

where $E = \mathit{name}_1{:}\mathit{argType}_1 \rightarrow \texttt{exn}, \ldots, \mathit{name}_n{:}\mathit{argType}_n \rightarrow \texttt{exn}$, and of course the parameters need not be present. Notice that, in the second rule, `raise` *`expr`* can have any type, and that in the last rule, the statements for the name_i in E link the type for the x in $\Gamma, x{:}\texttt{argType}_i; E \vdash \mathit{body}_i : A$ to name_i.

We will see in Sect. 4 that this interpretation of the way exceptions are treated in ML is similar to the basic type assignment system for λ^{try}, where we would write $\Gamma \vdash \mathit{expr} : A \mid E$ in closer correspondence to type judgements in $\lambda\mu$. There throwing an exception has any type as well. The type `exn` is not used there; since in λ^{try} handler names are *not* first-class citizens, so cannot be used outside of throwing or catching, there is no need for a separate type assignment rule for them.

Formal systems dealing with exception analyses for ML are described in the literature; a good summary of various systems is presented in Section 6 of [30]. Since the purpose of those systems does not correspond to ours, and neither of those approaches is related to classical logic, so not really related to our results, we will not present those here. In Sect. 2.5 we discuss de Groote's calculus $\lambda^{\rightarrow}_{exn}$ [20] that has features inspired by ML's exception handling, and how Bierman [8] has successfully linked $\lambda^{\rightarrow}_{exn}$ to CBV-$\lambda\mu$.

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

In terms of provable properties it is preferable to model eradication of applicative contexts 'one syntactic construction at the time', rather than use the λC-approach, which aborts entire contexts via $\mathtt{C}[\mathcal{A}(M)] \rightarrow M$, where the context is implicitly assumed to be as large as possible. This is exactly what can be modelled in $\lambda\mu$ (using the result of Lem. 1.16), 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.

This gives that we can now implement the functionality of 'escaping from the context' via the mechanism of consuming it via the reduction steps:

$$(\mu\_.[\mathsf{n}]M)PQR \;\rightarrow\; (\mu\_.[\mathsf{n}]M)QR \;\rightarrow\; (\mu\_.[\mathsf{n}]M)R \;\rightarrow\; \mu\_.[\mathsf{n}]M$$

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 $\mu\mathsf{n}.[\alpha]M$, essentially allowing for:

$$\mu\mathsf{n}.[\alpha](\mu\_.[\mathsf{n}]M)PQR \;\rightarrow^*_{\beta\mu} (1.16)\;\; \mu\mathsf{n}.[\alpha]\mu\_.[\mathsf{n}]M \;\rightarrow_{\beta\mu} (R)\;\; \mu\mathsf{n}.[\mathsf{n}]M \;\rightarrow_{\beta\mu} (E)\;\; M$$

However, this is not enough; we also want the catch-mechanism to disappear when computation terminates normally, as in $\mathsf{try}\ V; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i} \rightarrow V$. This asks for a last step $\mu\mathsf{n}.[\alpha]V \rightarrow V$ but this is in $\lambda\mu$ only possible when $\alpha = \mathsf{n}$ and α does not appear in V.

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 interpretations into $\lambda\mu$ we discuss here, even the historic ones, as, for example, the one presented in [27].

2.4 Nakano's system and Crolard's interpretation

In [25], Nakano presented an unnamed programming language with catch and throw, together with a notion of type assignment. It is presented as extension of a λ-calculus with pairing and disjunctive choice, and by adding *tag* variables that are used to mark destinations for the throw instructions. Crolard [12] later established a relation between Nakano's calculus and Parigot's $\lambda\mu$ (see below).

We briefly summarise Nakano's system as appeared in [25]; to facilitate the comparison with λ^{try}, we will not consider pairing, disjoint union, nor constants, and use Crolard's syntax; we will call it λ^{N} here.

Definition 2.1 (Nakano's λ^{N}) *i*) The terms of λ^{N} are defined through the grammar:

$$M,N \;::=\; x \mid \lambda x.M \mid MN \mid \mathsf{catch}\ \alpha\ M \mid \mathsf{throw}\ \alpha\ M$$

ii) Reduction on terms in λ^{N} is defined through the rules:

$$\begin{array}{lll} M\{\mathsf{throw}\ \alpha\ N/x\} & \rightarrow\ \mathsf{throw}\ \alpha\ N & (x \in M,\ M \neq x,\ M \neq \mathrm{C}[\mathsf{catch}\ \alpha\ \mathrm{C}'[x]])^{9} \\ \mathsf{catch}\ \alpha\ M & \rightarrow\ M & (\alpha \notin M) \\ \mathsf{catch}\ \alpha\ (\mathsf{throw}\ \alpha\ M) & \rightarrow\ M & (\alpha \notin M) \\ (\lambda x.M)N & \rightarrow\ M\{N/x\} & \end{array}$$

iii) The notion of types and context of variables Γ is the same as that of Def. 1.10; as usual, we use $Dom(\Gamma) \triangleq \{\, x \mid \exists A\ \ (x{:}A \in \Gamma)\,\}$. The notion of name context maps names to pairs of types and sets of term variables, $\alpha{:}\langle A, S\rangle$. The definition of compatible union of contexts of names Δ_1,Δ_2 then requires the type for a name to be the same in both Δ_1 and Δ_2, and takes the union of the associated sets.

iv) Type assignment for λ^{N} is defined through the following inference rules (modified here to our syntax and limitations of the language).

$$(Ax):\ \frac{}{\Gamma,x{:}A \vdash x : A \mid \Delta} \qquad (\rightarrow I):\ \frac{\Gamma,x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A\rightarrow B \mid \Delta}\,(*) \qquad (\rightarrow E):\ \frac{\Gamma_1 \vdash M : A\rightarrow B \mid \Delta_1 \quad \Gamma_2 \vdash N : A \mid \Delta_2}{\Gamma_1,\Gamma_2 \vdash MN : B \mid \Delta_1,\Delta_2}$$

$$(\mathsf{catch}):\ \frac{\Gamma \vdash M : A \mid \alpha{:}\langle A, S\rangle,\Delta}{\Gamma \vdash \mathsf{catch}\ \alpha\ M : A \mid \Delta} \qquad (\mathsf{throw}):\ \frac{\Gamma \vdash M : A \mid \Delta}{\Gamma \vdash \mathsf{throw}\ \alpha\ M : B \mid \alpha{:}\langle A, Dom(\Gamma)\rangle,\Delta}$$

$(*)$: $x \notin S$ for all $\alpha{:}\langle A, S\rangle \in \Delta$.

[9] The third restriction is missing in [25], but this seems to be in error.

With the extra feature $\langle A, Dom(\Gamma)\rangle$ the system registers in rule (throw) which are the term variables in the context of inputs used to type the term to be thrown. Then in rule $(\to I)$, to avoid binding of a free variable that occurs in a term that gets thrown, abstraction is only typed if the variable to be bound does not appear in any such term. Notice that rule (throw) adds $\alpha{:}\langle A, S\rangle$ to the context of names; it is not clear if Nakano intends this to be a true addition, *i.e.* if α is supposed to not already occur in Δ; since the third reduction rule demands that the name towards which the expression M is thrown does not appear in M, this seems to be the case.

Nakano proves that the system satisfies subject reduction; it avoids the problem highlighted in Ex. 4.5 through not allowing abstraction over variables that occur in thrown terms. However, this restriction is quite strong. For example, the term

$$\lambda x.(\lambda ab.b)(\mathsf{throw}\ \alpha\ x)\,x$$

would be considered untypeable in Nakano's system, since we have the λ^{N}-reduction

$$\lambda x.(\lambda ab.b)\,(\mathsf{throw}\ \alpha\ x)\,x \ \to\ \mathsf{throw}\ \alpha\ x$$

even though its type would be $A \to A$, and it also safely runs to $\lambda x.x$ under head-reduction, so reduction is not confluent. In our approach (as detailed below), we will type the term, but will not allow an exception to be thrown from inside an abstraction. Moreover, the system in not 'multiplicative', so contexts might be substantially larger than needed to type the term, so $Dom(\Gamma)$ might be unnecessarily large.

Notice that the first reduction rule states that *any* context can be eradicated by a throw that occurs inside it; thereby, this yields a highly non-confluent calculus; this was already observed by Nakano, who gives the following example[10]: take

$$M \ = \ \mathsf{catch}\ \alpha\ ((\lambda xy.I)\ (\mathsf{throw}\ \alpha\ K)\ (\mathsf{throw}\ \alpha\ Z))$$

(where $I = \lambda x.x$, $K = \lambda ab.a$, and $Z = \lambda ab.b$), then we have the following three reduction results:

$$\begin{array}{lclcl}
M & \to & \mathsf{catch}\ \alpha\ ((\lambda y.I)\,(\mathsf{throw}\ \alpha\ Z)) & \to & \mathsf{catch}\ \alpha\ I \ \to\ I \\
M & \to & \mathsf{catch}\ \alpha\ (\mathsf{throw}\ \alpha\ K) & \to & K \\
M & \to & \mathsf{catch}\ \alpha\ (\mathsf{throw}\ \alpha\ Z) & \to & Z
\end{array}$$

So reduction is manifestly non-confluent.

In part to address this, Crolard [12] defines an interpretation of terms in λ^{N} into $\lambda\mu$. Observe that, since reduction in $\lambda\mu$ is confluent, this interpretation cannot preserve (all) λ^{N}-reductions.

Definition 2.2 Crolard's interpretation $\ulcorner\cdot\lrcorner^{\mathrm{C}}$ is (adapted to our notation) defined by:

$$\begin{array}{rcl}
\ulcorner x\lrcorner^{\mathrm{C}} & = & x \\
\ulcorner \lambda x.N\lrcorner^{\mathrm{C}} & = & \lambda x.\ulcorner N\lrcorner^{\mathrm{C}} \\
\ulcorner MN\lrcorner^{\mathrm{C}} & = & \ulcorner M\lrcorner^{\mathrm{C}}\ulcorner N\lrcorner^{\mathrm{C}}
\end{array}
\qquad
\begin{array}{rcl}
\ulcorner \mathsf{catch}\ \alpha\ M\lrcorner^{\mathrm{C}} & = & \mu\alpha.[\alpha]\ulcorner M\lrcorner^{\mathrm{C}} \\
\ulcorner \mathsf{throw}\ \alpha\ N\lrcorner^{\mathrm{C}} & = & \mu\_.[\alpha]\ulcorner N\lrcorner^{\mathrm{C}}
\end{array}$$

Notice that this interpretation follows the observations made in Sect. 2.3. Then, for example, we get:

$$\ulcorner \mathsf{catch}\ \alpha\ (\mathsf{throw}\ \alpha\ M)\lrcorner^{\mathrm{C}} \ \triangleq\ \mu\alpha.[\alpha](\mu\_.[\alpha]\ulcorner M\lrcorner^{\mathrm{C}}) \ \to\ \mu\alpha.[\alpha]\ulcorner M\lrcorner^{\mathrm{C}} \ \to\ \ulcorner M\lrcorner^{\mathrm{C}}$$

The last step is only possible if α does not occur in $\ulcorner M\lrcorner^{\mathrm{C}}$, so respects the restriction imposed by the λ^{N}-reduction rule.

Nakano's example term translates as:

[10] Nakano uses numbers rather than I, K, and Z to get a typeable term.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}A,b{:}B,a{:}A \vdash a : A \mid \alpha{:}A}\,(Ax)}{x{:}A,b{:}B \vdash \lambda a.a : A \to A \mid \alpha{:}A}\,(\to I)}{x{:}A \vdash \lambda ba.a : B \to A \to A \mid \alpha{:}A}\,(\to I) \quad \dfrac{\dfrac{}{x{:}A \vdash x : A \mid \_{:}B,\alpha{:}A}\,(Ax)}{\mu\_.[\alpha]x : B : x{:}A \vdash \alpha{:}A}\,(\mu)}{x{:}A \vdash (\lambda ba.a)\,(\mu\_.[\alpha]x) : A \to A \mid \alpha{:}A}\,(\to E) \quad \dfrac{}{x{:}A \vdash x : A \mid \alpha{:}A}\,(Ax)}{\dfrac{x{:}A \vdash (\lambda ba.a)\,(\mu\_.[\alpha]x)x : A \mid \alpha{:}A}{\vdash \lambda x.(\lambda ba.a)\,(\mu\_.[\alpha]x)x : A \to A \mid \alpha{:}A}\,(\to I)}\,(\to E)$$

Figure 3. A derivation for $\vdash_{\lambda\mu} \lambda x.(\lambda ba.a)\,(\mu\_.[\alpha]x)x : A \to A \mid \alpha{:}A$.

$$⟦\text{catch } \alpha \ ((\lambda xy.I) \ (\text{throw } \alpha \ K) \ (\text{throw } \alpha \ Z))⟧^{\mathsf{C}} \ \triangleq \ \mu\alpha.[\alpha]((\lambda xy.I)\,(\mu\_.[\alpha]K)\,(\mu\_.[\alpha]Z))$$

which in $\lambda\mu$ *only* reduces as follows:

$$\mu\alpha.[\alpha]((\lambda xy.I) \ (\mu\_.[\alpha]K) \ (\mu\_.[\alpha]Z)) \ \to_{\beta\mu} \ \mu\alpha.[\alpha]((\lambda y.I) \ (\mu\_.[\alpha]Z)) \ \to_{\beta\mu} \ \mu\alpha.[\alpha]I \ \to_{\beta\mu} \ I$$

Moreover,

$$⟦\lambda x.(\lambda ab.b)\,(\text{throw } \alpha \ x)x⟧^{\mathsf{C}} \ \triangleq \ \lambda x.(\lambda ab.b)\,(\mu\_.[\alpha]x)x \ \to^{*}_{\beta\mu} \ \lambda x.x$$

and we can show $\vdash_{\lambda\mu} \lambda x.(\lambda ba.a)\,(\mu\_.[\alpha]x)x : A \to A \mid \alpha{:}A$, as in Fig. 3, so non-typeability in Nakano's system is not preserved under Crolard's interpretation.

2.5 De Groote and Bierman's approach

In [8], Bierman studies the interpretation of de Groote's simply-typed calculus $\lambda^{\to}_{exn}$, as presented in [20] into an abstract machine that evaluates $\lambda\mu$-terms using a CBV strategy; De Groote's calculus is based on ML's [24] handling of exceptions. We will follow Bierman's notation here, updated to ours.

Definition 2.3 $\lambda^{\to}_{exn}$-terms are defined through the grammar:[11]

$$\begin{array}{rcl} M,N & ::= & V \mid MN \mid \text{raise } (\mathsf{n},M) \mid \text{let n in } M \text{ handle n } x \ \Rightarrow \ N \text{ end} \\ V & ::= & c \mid x \mid \lambda x.M \end{array}$$

It uses the set of types defined as $A,B ::= c \mid \bot \mid A \to B$ (where c ranges over a set of ground types and $\bot$ is a distinguished ground type) and the type assignment rules:

$$\dfrac{}{\Gamma,x{:}A \vdash x : A \mid \Delta} \qquad \dfrac{\Gamma,x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A \to B \mid \Delta} \qquad \dfrac{\Gamma \vdash M : B \mid \Delta}{\Gamma \vdash \text{raise } (\mathsf{n},M) : A \mid \mathsf{n}{:}\neg B,\Delta}$$

$$\dfrac{}{\Gamma \vdash \mathrm{c} : \sigma\mathrm{c} \mid \Delta} \qquad \dfrac{\Gamma \vdash M : A \to B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta} \qquad \dfrac{\Gamma \vdash M : B \mid \mathsf{n}{:}\neg A,\Delta \qquad \Gamma,x{:}A \vdash N : B \mid \Delta}{\Gamma \vdash \text{let n in } M \text{ handle n } x \ \Rightarrow \ N \text{ end} : B \mid \Delta}$$

Here $\neg A = A \to \mathtt{exn}$, with exn ML's type of exceptions as discussed above, represented as $\bot$; σ assigns the appropriate ground type to each constant.

This system is, like Nakano's, developed out of classical logic, and de Groote argues that it is *complete, i.e.* fully represents classical proofs.[12]

Definition 2.4 The (call by value) reduction relation on $\lambda^{\to}_{exn}$ is defined through the rules:

[11] In [20] de Groote represents the syntax of $\lambda^{\to}_{exn}$ differently and thereby also the inference rules, by allowing for n to be a separate term, rather than only in the appropriate context, like raise (n M), as we do here in λ^{try}. Moreover, de Groote uses x for term variables, and y for names, which he calls 'exception variables', and keeps their types in the left-hand context; de Groote also adds n and n V to values, but these are not proper terms in Bierman's approach; they are in ML.

[12] Apart from the fact that we do not aim for completeness in this sense for λ^{try}, it seems also likely that, with the restrictions present in Def. 3.1 (*iii*), this would not be possible to show for any of the notions of type assignment we define here for λ^{try}.

$$\begin{array}{lcll}
(\lambda x.M)V & \to & M\{V/x\} & \\
V'(\mathsf{raise}\ (\mathsf{n},V)) & \to & \mathsf{raise}\ (\mathsf{n},V) & \\
(\mathsf{raise}\ (\mathsf{n},V))M & \to & \mathsf{raise}\ (\mathsf{n},V) & \\
\mathsf{raise}\ (\mathsf{m},\mathsf{raise}\ (\mathsf{n},V)) & \to & \mathsf{raise}\ (\mathsf{n},V) & \\
\mathsf{let\ n\ in}\ V\ \mathsf{handle\ n}\ x \Rightarrow N\ \mathsf{end} & \to & V & (\mathsf{n} \notin \mathit{fn}(V)) \\
\mathsf{let\ n\ in\ raise}\ (\mathsf{n},V)\ \mathsf{handle\ n}\ x \Rightarrow N\ \mathsf{end} & \to & N\{V/x\} & (\mathsf{n} \notin \mathit{fn}(V,N)) \\
\mathsf{let\ n\ in\ raise}\ (\mathsf{m},V)\ \mathsf{handle\ n}\ x \Rightarrow N\ \mathsf{end} & \to & \mathsf{raise}\ (\mathsf{m},V) & (\mathsf{m} \neq \mathsf{n}, \mathsf{n} \notin \mathit{fn}(V))
\end{array}$$

De Groote makes no attempt to link this reduction relation to how exceptions are dealt with in ML.

Notice that there are no rules permitting raising an exception from within an abstraction, thereby avoiding the subject reduction problem mentioned in Ex. 4.5. However, de Groote does not put the side-condition on the last three rules, opening the system to another kind of subject reduction problem; Bierman adds the restrictions in his presentation. Operationally, the $\lambda^{\to}_{exn}$-term 'let n in M handle n $x \Rightarrow N$ end' corresponds to the λ^{try}-term 'try M; catch $\mathsf{n}(x) = N$' (see Def. 3.1).

Definition 2.5 (Interpretation of $\lambda^{\to}_{exn}$ into $\lambda\mu$) Bierman defines the interpretation of $\lambda^{\to}_{exn}$-terms into $\lambda\mu$-terms as follows:

$$\begin{array}{rcl}
\ulcorner x \lrcorner^{\mathrm{B}} & \triangleq & x \\
\ulcorner \lambda x.M \lrcorner^{\mathrm{B}} & \triangleq & \lambda x.\ulcorner M \lrcorner^{\mathrm{B}} \\
\ulcorner MN \lrcorner^{\mathrm{B}} & \triangleq & \ulcorner M \lrcorner^{\mathrm{B}} \ulcorner N \lrcorner^{\mathrm{B}} \\
\ulcorner \mathsf{raise}\ (\mathsf{n},M) \lrcorner^{\mathrm{B}} & \triangleq & (\lambda x.\mu\beta.[\mathsf{n}]x)\ulcorner M \lrcorner^{\mathrm{B}}\,{}^{13} \\
\ulcorner \mathsf{let\ n\ in}\ M\ \mathsf{handle\ n}\ x \Rightarrow N\ \mathsf{end} \lrcorner^{\mathrm{B}} & \triangleq & \mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\lambda x.\ulcorner N \lrcorner^{\mathrm{B}})\,(\mu\mathsf{n}.[\beta_{\mathsf{n}}]\ulcorner M \lrcorner^{\mathrm{B}})
\end{array}$$

and states "*It is quite easy to verify that this translation preserves the expected operational behaviour.*", intended through his interpretation into the abstract machine which essentially runs CBV reduction, but is not shown. We illustrate Bierman's claim.

Example 2.6 Take 'let n in raise n V handle n $z \Rightarrow N$ end'. Then we have:

$$\begin{array}{ll}
\ulcorner \mathsf{let\ n\ in\ raise}\ (\mathsf{n}\ V)\ \mathsf{handle\ n}\ z \Rightarrow N\ \mathsf{end} \lrcorner^{\mathrm{B}} & \triangleq \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\lambda z.\ulcorner N \lrcorner^{\mathrm{B}})\,(\mu\mathsf{n}.[\beta_{\mathsf{n}}]\ulcorner \mathsf{raise}\ (\mathsf{n},V) \lrcorner^{\mathrm{B}}) & \triangleq \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\lambda z.\ulcorner N \lrcorner^{\mathrm{B}})\,(\mu\mathsf{n}.[\beta_{\mathsf{n}}](\lambda x.\mu\_.[\mathsf{n}]x)\ulcorner V \lrcorner^{\mathrm{B}}) & \to^{\mathrm{v}}_{\beta\mu} \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\lambda z.\ulcorner N \lrcorner^{\mathrm{B}})\,(\mu\mathsf{n}.[\beta_{\mathsf{n}}](\mu\_.[\mathsf{n}]\,\ulcorner V \lrcorner^{\mathrm{B}})) & \to^{\mathrm{v}}_{\beta\mu} \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\lambda z.\ulcorner N \lrcorner^{\mathrm{B}})\,(\mu\mathsf{n}.[\mathsf{n}]\ulcorner V \lrcorner^{\mathrm{B}}) & \to^{\mathrm{v}}_{\beta\mu} \\
(\lambda z.\ulcorner N \lrcorner^{\mathrm{B}})\,(\mu\mathsf{n}.[\mathsf{n}]\ulcorner V \lrcorner^{\mathrm{B}}) & \to^{\mathrm{v}}_{\beta\mu} \\
(\lambda z.\ulcorner N \lrcorner^{\mathrm{B}})\ulcorner V \lrcorner^{\mathrm{B}} \ \to^{\mathrm{v}}_{\beta\mu}\ \ulcorner N\{V/x\} \lrcorner^{\mathrm{B}} &
\end{array}$$

Also, for 'let n in V handle n $z \Rightarrow N$ end' with $\mathsf{n} \notin \mathit{fn}(V)$ we get:

$$\begin{array}{lll}
\ulcorner \mathsf{let\ n\ in}\ V\ \mathsf{handle\ n}\ z \Rightarrow N\ \mathsf{end} \lrcorner^{\mathrm{B}} & \triangleq & \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\lambda z.\ulcorner N \lrcorner^{\mathrm{B}})\,(\mu\mathsf{n}.[\beta_{\mathsf{n}}]\ulcorner V \lrcorner^{\mathrm{B}}) & \to^{\mathrm{v}}_{\beta\mu} & \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\mu\gamma.\{(\lambda z.N)\cdot\gamma/\mathsf{n}\}\,[\beta_{\mathsf{n}}]\ulcorner V \lrcorner^{\mathrm{B}}) & = & (\mathsf{n} \neq \beta_{\mathsf{n}}, \gamma\ \mathit{fresh}) \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\mu\gamma.[\beta_{\mathsf{n}}]\{(\lambda z.N)\cdot\gamma/\mathsf{n}\}\ulcorner V \lrcorner^{\mathrm{B}}) & = & (\mathsf{n} \notin \mathit{fn}(V)) \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}](\mu\gamma.[\beta_{\mathsf{n}}]\ulcorner V \lrcorner^{\mathrm{B}}) & \to^{\mathrm{v}}_{\beta\mu} & \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}]\ulcorner V \lrcorner^{\mathrm{B}}\{\beta_{\mathsf{n}}/\gamma\} & = & \\
\mu\beta_{\mathsf{n}}.[\beta_{\mathsf{n}}]\ulcorner V \lrcorner^{\mathrm{B}} \ \to^{\mathrm{v}}_{\beta\mu}\ \ulcorner V \lrcorner^{\mathrm{B}} & &
\end{array}$$

13 [7] essentially uses $(\lambda x.\mu\beta.[a]x)\ulcorner M \lrcorner^{\mathrm{B}}$, but the use of a rather than n seems to be in error; also, we assume β to be fresh, so can be replaced by '_'.

both as intended.

3 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 '$\text{catch } \mathsf{n}(x) = M$' 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 '$\text{throw } \mathsf{n}(N)$' a call to the exception handler with name n, passing it the argument N. By the very nature of exception handling, this implies that then N itself is a term that does not call on n (so exception handling is non-recursive), but can call on other exception handlers, defined outside the scope of the present try-term.

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

Definition 3.1 (Syntax of λ^{try}) *i*) The set of *pre-terms* of λ^{try} is defined by the grammar:

$$\begin{array}{rcll} \text{Catch\_Block} & ::= & \epsilon \mid \text{Catch\_Block catch } \mathsf{n}(x) = N;\,^{14} & \\ M,N & ::= & V \mid MN \mid \text{try } M; \text{Catch\_Block} \mid \text{throw } \mathsf{n}(M) & \\ V & ::= & x \mid \lambda x.M & (\mathit{Values}) \end{array}$$

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

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

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

iii) The set of *terms* are pre-terms that satisfy the following restrictions:

a) In a try-term, the catch-block is never empty (ϵ), and in the catch-block $\overrightarrow{\text{catch } \mathsf{n}_i(x) = M_i}$ the names n_i do not occur in the exception handlers M_j, for any $i,j \in \underline{n}$,[15] and all declared names $\mathsf{n}_1,\cdots,\mathsf{n}_n$ are distinct;

b) For each $\text{throw } \mathsf{n}_l(N)$ that occurs inside M in the term $\text{try } M; \overrightarrow{\text{catch } \mathsf{n}_i(x) = N_i}$, none of the names n_i occur in N.

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

$$\begin{array}{lcl} bv(x) & = & \emptyset \\ bv(\lambda x.M) & = & bv(M) \cup \{x\} \\ bv(MN) & = & bv(M) \cup bv(N) \\ bv(\text{try } M; \overrightarrow{\text{catch } \mathsf{n}_i(x) = N_i}) & = & bv(M) \cup \\ & & \quad bv(N_1) \cup \cdots \cup bv(N_n) \cup \{x\} \\ bv(\text{throw } \mathsf{n}(M)) & = & bv(M) \end{array}
\qquad
\begin{array}{lcl} bn(x) & = & \emptyset \\ bn(\lambda x.M) & = & bn(M) \\ bn(MN) & = & bn(M) \cup bn(N) \\ bn(\text{try } M; \overrightarrow{\text{catch } \mathsf{n}_i(x) = N_i}) & = & \{\mathsf{n}_1,\ldots,\mathsf{n}_n\} \cup \\ & & \quad bn(M) \cup bn(N_1) \cup \cdots \cup bn(N_n) \\ bn(\text{throw } \mathsf{n}(M)) & = & bn(M) \end{array}$$

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. We write $fv(M)$ for the set of free variables in M, and $fn(M)$ for its free names.

Remark that the conditions of part (*iii*) serve to maintain a naming 'hygiene', in that we

[14] Notice that each Catch_Block does finish with ';', but that this semicolon is included *in* the block; for readability, we will add the semicolon below, so write $\text{try } M; \text{Catch\_Block}; \text{catch } \mathsf{n}(x) = N$ rather than $\text{try } M; \text{Catch\_Block catch } \mathsf{n}(x) = N$.

[15] We use $i \in \underline{n}$ for $i \in \{1,\ldots,n\}$.

do not want terms that contain occurrences of bound names to reduce to terms where those names are free when throwing an exception.

To control the throwing of exceptions, we define a notion of CBN and CBV reduction; these define an evaluation strategy, where only ever one sub-expression can execute and an exception is only ever thrown when needed to continue reduction.

Definition 3.2 (λ^{try}-REDUCTION) *i*) The notion of CBN *reduction* $\rightarrow^{\mathrm{N}}_{\mathsf{try}}$ on λ^{try} is defined as an extension of that reduction on λ-terms. The main reduction rules are:

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

CBN *applicative evaluation contexts* are defined as:

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

ii) The notion of CBV *reduction* $\rightarrow^{\mathrm{v}}_{\mathsf{try}}$ on λ^{try} is defined as an extension of CBV reduction on λ-terms using the main reduction rules from CBN, with the exception of (β) which gets replaced by:

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

It adds the rule:

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

CBV *applicative evaluation contexts* are defined as:

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

Notice that, as in all (CBN, or CBV) 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, but it is natural to change the rule so that this would be required.

Moreover, a term like

$$\mathsf{try}\ \lambda x.\mathsf{throw}\,\mathsf{n}(x); \mathsf{catch}\ \mathsf{n}(y) = N$$

is stuck; computation has finished on the main term, which is a value, but is not allowed to emerge from the block because it would free the name n.

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 (interpretation of the) exception handlers catch $\mathsf{n}(x) = N$ *inside* the interpretation of M; this is done using substitution,[16] introducing variables c_{n_i} that are placed in front of the argument that is passed to the exception handler in throw $\mathsf{n}_i(M)$. As an illustration, we aim to interpret

[16] 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 CBN (CBV) reduction in λ^{try} by CBN (CBV) reduction in $\lambda\mu$, in particular when modelling the step $M \rightarrow N \ \Rightarrow\ \mathsf{try}\ M; \mathsf{Catch\_Block} \rightarrow \mathsf{try}\ N; \mathsf{Catch\_Block}$.

$$\mathsf{try}\ \mathsf{C}[\mathsf{throw}\,\mathsf{n}(N)];\mathsf{catch}\ \mathsf{n}(x) = P;$$

by

$$\mu\mathsf{n}.[\mathsf{n}]\,\mathsf{C}[\mu\_.[\mathsf{n}]\,(\lambda x.\llbracket P\rrbracket^{\lambda\mu})\llbracket N\rrbracket^{\lambda\mu}]$$

Definition 3.3 (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:

$$\begin{array}{rcl}
\llbracket x\rrbracket^{\lambda\mu} & \triangleq & x \\
\llbracket \lambda x.M\rrbracket^{\lambda\mu} & \triangleq & \lambda x.\llbracket M\rrbracket^{\lambda\mu} \\
\llbracket MN\rrbracket^{\lambda\mu} & \triangleq & \llbracket M\rrbracket^{\lambda\mu}\llbracket N\rrbracket^{\lambda\mu} \\
\llbracket \mathsf{throw}\,\mathsf{n}(M)\rrbracket^{\lambda\mu} & \triangleq & \mu\_.[\mathsf{n}]\,c_{\mathsf{n}}\llbracket M\rrbracket^{\lambda\mu} \\
\llbracket \mathsf{try}\ M;\epsilon\rrbracket^{\lambda\mu} & \triangleq & \llbracket M\rrbracket^{\lambda\mu} \\
\llbracket \mathsf{try}\ M;\mathsf{Catch\_Block};\mathsf{catch}\ \mathsf{n}(x) = N\rrbracket^{\lambda\mu} & \triangleq & (\mu\mathsf{n}.[\mathsf{n}]\llbracket \mathsf{try}\ M;\mathsf{Catch\_Block}\rrbracket^{\lambda\mu})\,\{\lambda x.\llbracket N\rrbracket^{\lambda\mu}/c_{\mathsf{n}}\}
\end{array}$$

Remark 3.4 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 (where we drop the superscript on $\llbracket\cdot\rrbracket^{\lambda\mu}$).

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

Once the named exception handlers have been translated to anonymous abstractions and inserted inside the term, the names no longer play a role, and the prefix $\mu\mathsf{m}.[\mathsf{m}]$ just serves as a delimiter for the two throws.

We will show that both reduction and assignable types (under the basic system, see Sect. 4) are preserved under this interpretation. First we show that term-substitution is preserved.

Lemma 3.5 ($\llbracket\cdot\rrbracket^{\lambda\mu}$ preserves term substitution) $\llbracket M\rrbracket^{\lambda\mu}\{\llbracket N\rrbracket^{\lambda\mu}/x\} = \llbracket M\{N/x\}\rrbracket^{\lambda\mu}$.

Proof: By induction on the definition of term substitution.

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

Theorem 3.6 (Soundness of $\llbracket\cdot\rrbracket^{\lambda\mu}$ with respect to $\rightarrow^{\mathrm{N}}_{\mathsf{try}}$) *If $P \rightarrow^{\mathrm{N}}_{\mathsf{try}} Q$, then $\llbracket P\rrbracket^{\lambda\mu} \rightarrow^{\mathrm{N}*}_{\beta\mu} \llbracket Q\rrbracket^{\lambda\mu}$.*

Proof: By induction on the definition of $\rightarrow^{\mathrm{N}}_{\mathsf{try}}$. We show the non-trivial cases (and drop the superscript on $\llbracket\cdot\rrbracket^{\lambda\mu}$).

(throw): Then $P = (\mathsf{throw}\,\mathsf{n}(N))M \rightarrow \mathsf{throw}\,\mathsf{n}(N) = Q$, and

$$\llbracket(\mathsf{throw}\,\mathsf{n}(N))M\rrbracket \ \triangleq\ (\mu\_.[\mathsf{n}]\,c_{\mathsf{n}}\llbracket N\rrbracket)\llbracket M\rrbracket \ \rightarrow^{\mathrm{N}}_{\beta\mu}(1.16)\ \ \mu\_.[\mathsf{n}]\,c_{\mathsf{n}}\llbracket N\rrbracket \ \triangleq\ \llbracket\mathsf{throw}\,\mathsf{n}(N)\rrbracket$$

(try-throw): Then $P = \mathsf{try}\ \mathsf{throw}\,\mathsf{n}_l(N);\overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = M_i} \rightarrow M_l\{N/x\} = Q$, with $l \in \{1,\ldots,n\}$, and

$$\begin{array}{lr}
[\![\mathsf{try\ throw\ n}_l(N);\overrightarrow{\mathsf{catch\ n}_i(x) = M_i}]\!] & \triangleq \\
(\mu\mathsf{n}_n.[\mathsf{n}_n][\![\mathsf{try\ throw\ n}_l(N);\overrightarrow{\mathsf{catch\ n}_i(x) = M_i}]\!]\,)\ \{\lambda x.[\![M_n]\!]/c_{\mathsf{n}_n}\} & \triangleq \\
(\mu\mathsf{n}_n.[\mathsf{n}_n]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![\mathsf{throw\ n}_l(N)]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)\ \{\lambda x.[\![M_n]\!]/c_{\mathsf{n}_n}\} & = \\
\qquad (c_{\mathsf{n}_n} \notin (\mu\mathsf{n}_n.[\mathsf{n}_n]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![\mathsf{throw\ n}_l(N)]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)\ \{\lambda x.[\![M_{n-1}]\!]/c_{\mathsf{n}_{n-1}}\}) & \\
(\mu\mathsf{n}_n.[\mathsf{n}_n]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![\mathsf{throw\ n}_l(N)]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots) & \rightarrow_{\beta\mu}^{\mathrm{N}} (E) \\
\qquad\qquad (\mathsf{n}_n \notin (\mu\mathsf{n}_{n-1}.[\mathsf{n}_{n-1}]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![\mathsf{throw\ n}_l(N)]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots) & \\
(\mu\mathsf{n}_{n-1}.[\mathsf{n}_{n-1}]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![\mathsf{throw\ n}_l(N)]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)\ \{\lambda x.[\![M_{n-1}]\!]/c_{\mathsf{n}_{n-1}}\} & \rightarrow_{\beta\mu}^{\mathrm{N}*} (E) \\
(\mu\mathsf{n}_l.[\mathsf{n}_l]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![\mathsf{throw\ n}_l(N)]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)\ \{\lambda x.[\![M_l]\!]/c_{\mathsf{n}_l}\} & \triangleq \\
(\mu\mathsf{n}_l.[\mathsf{n}_l]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1]\mu\_.[\mathsf{n}_l]c_{\mathsf{n}_l}[\![N]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)\ \{\lambda x.[\![M_l]\!]/c_{\mathsf{n}_l}\} & = \\
\mu\mathsf{n}_l.[\mathsf{n}_l]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1]\mu\_.[\mathsf{n}_l](\lambda x.[\![M_l]\!]\,)[\![N]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots) & \rightarrow_{\beta\mu}^{\mathrm{N}*} (R) \\
\mu\mathsf{n}_l.[\mathsf{n}_l]\mu\_.[\mathsf{n}_l]((\lambda x.[\![M_l]\!]\,)[\![N]\!]\,) \quad \rightarrow_{\beta\mu}^{\mathrm{N}} (R) \quad \mu\mathsf{n}_l.[\mathsf{n}_l]((\lambda x.[\![M_l]\!]\,)[\![N]\!]\,) & \rightarrow_{\beta\mu}^{\mathrm{N}} (E) \\
(\lambda x.[\![M_l]\!]\,)[\![N]\!] \quad \rightarrow_{\beta\mu}^{\mathrm{N}} \quad [\![M_l]\!]\,\{[\![N]\!]/x\} \quad = (3.5) \quad [\![M_l\{N/x\}]\!] &
\end{array}$$

(try-normal): Then $P = \mathsf{try}\ N;\overrightarrow{\mathsf{catch\ n}_i(x) = M_i} \rightarrow N = Q$, with $\mathsf{n}_i \notin N$ for all $i \in \underline{n}$. Notice that, since $\mathsf{n}_i \notin N$ for all $i \in \underline{n}$, also n_i and c_{n_i} do not occur in $[\![N]\!]$. Then:

$$\begin{array}{lr}
[\![\mathsf{try}\ N;\overrightarrow{\mathsf{catch\ n}_i(x) = M_i}]\!] \ \triangleq\ (\mu\mathsf{n}_n.[\mathsf{n}_n][\![\mathsf{try}\ N;\overrightarrow{\mathsf{catch\ n}_i(x) = M_i}]\!]\,)\ \{\lambda x.[\![M_n]\!]/c_{\mathsf{n}_n}\} & \triangleq \\
(\mu\mathsf{n}_n.[\mathsf{n}_n]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![N]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)\ \{\lambda x.[\![M_n]\!]/c_{\mathsf{n}_n}\} & = \\
\qquad\qquad (c_{\mathsf{n}_n} \notin (\mu\mathsf{n}_n.[\mathsf{n}_n]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![N]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)\ \{\lambda x.[\![M_{n-1}]\!]/c_{\mathsf{n}_{n-1}}\}) & \\
(\mu\mathsf{n}_n.[\mathsf{n}_n]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![N]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots) & \rightarrow_{\beta\mu}^{\mathrm{N}} (E) \\
\qquad\qquad\qquad (\mathsf{n}_n \notin (\mu\mathsf{n}_{n-1}.[\mathsf{n}_{n-1}]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![N]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)) & \\
(\mu\mathsf{n}_{n-1}.[\mathsf{n}_{n-1}]\cdots(\mu\mathsf{n}_1.[\mathsf{n}_1][\![N]\!]\,)\ \{\lambda x.[\![M_1]\!]/c_{\mathsf{n}_1}\}\cdots)\ \{\lambda x.[\![M_{n-1}]\!]/c_{\mathsf{n}_{n-1}}\} & \rightarrow_{\beta\mu}^{\mathrm{N}*} \quad [\![N]\!]
\end{array}$$

$(M \rightarrow N \Rightarrow ML \rightarrow NL)$: $[\![ML]\!] \ \triangleq\ [\![M]\!]\,[\![L]\!] \ \rightarrow_{\beta\mu}^{\mathrm{N}*} (IH)\ [\![N]\!]\,[\![L]\!] \ \triangleq\ [\![NL]\!]$

$(M \rightarrow N \Rightarrow \mathsf{try}\ M;\overrightarrow{\mathsf{catch\ n}_i(x) = L_i} \rightarrow \mathsf{try}\ N;\overrightarrow{\mathsf{catch\ n}_i(x) = L_i})$: $[\![\mathsf{try}\ M;\overrightarrow{\mathsf{catch\ n}_i(x) = L_i}]\!] \ \triangleq$

$$\begin{array}{lll}
(\mu\mathsf{n}_n.[\mathsf{n}_n]\cdots\mu\mathsf{n}_1.[\mathsf{n}_1]\,[\![M]\!]\,)\ \overrightarrow{\{\lambda x.[\![L_i]\!]/c_{\mathsf{n}_i}\}} & \rightarrow_{\beta\mu}^{\mathrm{N}*} (IH) & \\
(\mu\mathsf{n}_n.[\mathsf{n}_n]\cdots\mu\mathsf{n}_1.[\mathsf{n}_1]\,[\![N]\!]\,)\ \overrightarrow{\{\lambda x.[\![L_i]\!]/c_{\mathsf{n}_i}\}} & \triangleq & [\![\mathsf{try}\ N;\overrightarrow{\mathsf{catch\ n}_i(x) = L_i}]\!]
\end{array}$$ □

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

Similarly, we can verify that the interpretation respects CBV reduction $\rightarrow_{\mathsf{try}}^{\mathrm{V}}$.

Theorem 3.7 (SOUNDNESS OF $[\![\cdot]\!]^{\lambda\mu}$ WITH RESPECT TO $\rightarrow_{\mathsf{try}}^{\mathrm{V}}$) *If $P \rightarrow_{\mathsf{try}}^{\mathrm{V}} Q$, then $[\![P]\!]^{\lambda\mu} \rightarrow_{\beta\mu}^{\mathrm{V}*} [\![Q]\!]^{\lambda\mu}$.*

Proof: By induction on the definition of $\rightarrow_{\mathsf{try}}^{\mathrm{V}}$. Most cases correspond to those of the previous proof, except for:

$$\begin{array}{lll}
(\mathsf{throw}_{\mathrm{V}}):\ [\![V(\mathsf{throw\ n}(N))]\!]^{\lambda\mu} & \triangleq & [\![V]\!]^{\lambda\mu}(\mu\_.[\mathsf{n}]c_{\mathsf{n}}.[\![N]\!]^{\lambda\mu}) \ \rightarrow_{\beta\mu}^{\mathrm{V}} (\mu_{\mathrm{V}}) \\
\quad \mu\gamma.\{[\![V]\!]^{\lambda\mu}\cdot\gamma/\_\}([\mathsf{n}].c_{\mathsf{n}}[\![N]\!]^{\lambda\mu}) & = & \mu\gamma.[\mathsf{n}].c_{\mathsf{n}}[\![N]\!]^{\lambda\mu} \qquad\qquad =_{\alpha} \\
\quad \mu\_.[\mathsf{n}]c_{\mathsf{n}}[\![N]\!]^{\lambda\mu} & \triangleq & [\![\mathsf{throw\ n}(N)]\!]^{\lambda\mu}
\end{array}$$

Notice that 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.

4 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; this is, in principle, also the approach chosen for java [14], and all the notions of type assignment presented above in Sect. 2.

Definition 4.1 (BASIC TYPE ASSIGNMENT FOR λ^{try}) *i*) Types and contexts of variables Γ and of names Δ are those of Def. 1.10, but using n, m rather than Greek characters.

ii) Basic type assignment for terms in λ^{try} is defined through the following inference system:

$$(Ax): \frac{}{\Gamma,x{:}A \vdash x : A \mid \Delta} \qquad (\to E): \frac{\Gamma \vdash M : A \to B \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : B \mid \Delta}$$

$$(\to I): \frac{\Gamma,x{:}A \vdash M : B \mid \Delta}{\Gamma \vdash \lambda x.M : A \to B \mid \Delta}\,(x \notin \Gamma) \qquad (\mathsf{throw}): \frac{\Gamma \vdash N : A \mid \Delta}{\Gamma \vdash \mathsf{throw}\,\mathsf{n}(N) : C \mid \mathsf{n}{:}A \to B,\Delta}$$

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

We write $\Gamma \vdash_{\mathsf{B}} M : A \mid \Delta$ for statements derivable using these rules.

Notice that our (throw) rule is almost identical to Nakano's in that it allows to derive *any* type for the term $\mathsf{throw}\,\mathsf{n}(N)$, 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.

Comparing this notion to how ML types exceptions, we can see that in rule (throw) we do not implicitly type n with $A \to \mathtt{exn}$. Instead, the return type of names needs to be that of the main term in the try-block.

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

$$\begin{array}{c}\boxed{\phantom{\Gamma \vdash M : C \mid \mathsf{n}_i{:}A_i \to B_i,\Delta}}\\ \Gamma \vdash M : C \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B_i},\Delta\end{array} \quad \text{and} \quad \begin{array}{c}\boxed{\phantom{\Gamma,x{:}A_i \vdash N_i : B_i \mid \Delta}}\\ \Gamma,x{:}A_i \vdash N_i : B_i \mid \Delta\end{array}\ (\forall i \in \underline{n})$$

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 $\mathsf{try}\ M; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}$ will result in M' (assuming M' is free of throws) and in order to achieve subject reduction, M' should be of type C. If it does, running M will produce $\mathsf{throw}\,\mathsf{n}(L)$ and (assuming $\mathsf{n} = \mathsf{n}_l \in \vec{\mathsf{n}_i}$), $\mathsf{try}\ M; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}$ will run to $N_l\{L/x\}$, which has type B_l. So in order to achieve a subject reduction result also for this case, there is no choice but to demand that $C = B_1 = \cdots = B_n$.

Lemma 4.2 (Weakening and thinning for $\vdash_{\mathsf{B}}$) *The following rules are admissible for* $\vdash_{\mathsf{B}}$:

$$(Wk): \frac{\Gamma \vdash M : A \mid \Delta}{\Gamma' \vdash M : A \mid \Delta'}\,(\Gamma \subseteq \Gamma', \Delta \subseteq \Delta')$$

$$(Th): \frac{\Gamma \vdash M : A \mid \Delta}{\Gamma' \vdash M : A \mid \Delta'}\,(\Gamma' = \{\, x{:}B \in \Gamma \mid x \in fv(M) \,\},\ \Delta' = \{\, \mathsf{n}{:}B \in \Delta \mid \mathsf{n} \in fn(M) \,\})$$

Proof: Standard. □

We can show:

Lemma 4.3 (Substitution lemma for $\vdash_{\mathsf{B}}$) *If* $\Gamma,x{:}C \vdash_{\mathsf{B}} M : A \mid \Delta$ *and* $\Gamma \vdash_{\mathsf{B}} N : C \mid \Delta$, *then* $\Gamma \vdash_{\mathsf{B}} M\{N/x\} : A \mid \Delta$.

Proof: By induction on the definition of term substitution.

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

Theorem 4.4 (Subject reduction for $\vdash_{\mathsf{B}}$) *i) If* $\Gamma \vdash_{\mathsf{B}} P : A \mid \Delta$ *and* $P \to^{\mathsf{N}}_{\mathsf{try}} Q$, *then* $\Gamma \vdash_{\mathsf{B}} Q : A \mid \Delta$.
ii) If $\Gamma \vdash_{\mathsf{B}} P : B \mid \Delta$ *and* $P \to^{\mathsf{V}}_{\mathsf{try}} Q$, *then* $\Gamma \vdash_{\mathsf{B}} Q : B \mid \Delta$.

Proof: *i*) By induction on the definition of $\to^{\mathsf{N}}_{\mathsf{try}}$.

(β): Standard, using Lem. 4.3.

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

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

We can construct the derivation for Q:[17]

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

(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:

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

In particular, we have derivations for both $\Gamma \vdash_{\mathsf{B}} M : A_l \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B}, \Delta$ and $\Gamma, x{:}A_l \vdash_{\mathsf{B}} N_l : B \mid \Delta$. By the definition of λ^{try}-terms, we know that $\mathsf{n}_i \notin fn(M)$, for all $i \in \underline{n}$, so by thinning (Lem. 4.2) we can remove $\overrightarrow{\mathsf{n}_i{:}A_i \to B}$ from the co-context for the first to obtain $\Gamma \vdash_{\mathsf{B}} M : A_l \mid \Delta$. Then, by Lem. 4.3, we obtain $\Gamma \vdash_{\mathsf{B}} N_l\{M/x\} : B \mid \Delta$.

(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:

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

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

($M \to N \Rightarrow ML \to NL$): Then the derivation for P is constructed as follows:

$$\dfrac{\begin{array}{c}\boxed{\phantom{\mathcal{D}\mathcal{D}\mathcal{D}}}\\ \Gamma \vdash_{\mathsf{B}} M : A \to B \mid \Delta\end{array} \quad \begin{array}{c}\boxed{\mathcal{D}}\\ \Gamma \vdash_{\mathsf{B}} L : A \mid \Delta\end{array}}{\Gamma \vdash_{\mathsf{B}} ML : B \mid \Delta}\,(\to E)$$

By induction, $\Gamma \vdash_{\mathsf{B}} N : A \to B \mid \Delta$, and we can construct a derivation for Q:

$$\dfrac{\begin{array}{c}\boxed{\phantom{\mathcal{D}\mathcal{D}\mathcal{D}}}\\ \Gamma \vdash_{\mathsf{B}} N : A \to B \mid \Delta\end{array} \quad \begin{array}{c}\boxed{\mathcal{D}}\\ \Gamma \vdash_{\mathsf{B}} L : A \mid \Delta\end{array}}{\Gamma \vdash_{\mathsf{B}} NL : B \mid \Delta}\,(\to E)$$

($M \to N \Rightarrow \mathsf{try}\ M; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = L_i} \to \mathsf{try}\ N; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = L_i}$): Then the derivation for P is constructed as follows:

$$\dfrac{\begin{array}{c}\boxed{\phantom{\mathcal{D}\mathcal{D}\mathcal{D}}}\\ \Gamma \vdash_{\mathsf{B}} M : B \mid \overrightarrow{\mathsf{n}_i{:}A_i \to C}, \Delta\end{array} \quad \begin{array}{c}\boxed{\mathcal{D}}\\ \Gamma, x{:}A_i \vdash_{\mathsf{B}} L_i : C \mid \Delta \ \ (\forall i \in \underline{n})\end{array}}{\Gamma \vdash_{\mathsf{B}} \mathsf{try}\ M; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = L_i} : B \mid \Delta}\,(\mathsf{try})$$

By induction, $\Gamma \vdash_{\mathsf{B}} N : B \mid \Delta$, and we can construct a derivation for Q:

[17] 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 1.9) $(\mu\alpha.[\beta]M)N \to \mu\alpha.([\beta]M\{N{\cdot}\alpha/\alpha\})$; before the reduction, α has type $A \to B$, say, and after it has type B.

$$\dfrac{\begin{array}{c}\boxed{\qquad\qquad\qquad}\\ \Gamma \vdash_{\mathsf{B}} N : B \mid \overrightarrow{\mathsf{n}_i{:}A_i \to C}, \Delta\end{array} \qquad \begin{array}{c}\boxed{\qquad\mathcal{D}\qquad}\\ \Gamma, x{:}A_i \vdash_{\mathsf{B}} L_i : C \mid \Delta \quad (\forall i \in \underline{n})\end{array}}{\Gamma \vdash_{\mathsf{B}} \mathsf{try}\ N; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = L_i} : B \mid \Delta}\ (\mathsf{try})$$

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

($\mathsf{throw}_{\mathrm{V}}$): 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 follows:

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

We can construct the derivation for Q:

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

□

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, as we will argue now.

Example 4.5 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 the fact 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

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

We can construct

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

but cannot, in general, derive $\Gamma \vdash_{\mathsf{B}} \mathsf{throw}\,\mathsf{n}(N) : A \to B \mid \mathsf{n}{:}D \to C, \Delta$: notice that x might be free in N, so then would need a type in any derivation for N. This problem was observed by Nakano [25, 26], 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 by not allowing the throwing of 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 4.6 (Preservation of assignable types under $[\![\cdot]\!]^{\lambda\mu}$) *If* $\Gamma \vdash_{\mathsf{B}} M : B \mid \overrightarrow{\mathsf{n}_i{:}A_i \to C_i}$, *then* $\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash_{\lambda\mu} [\![M]\!]^{\lambda\mu} : B \mid \overrightarrow{\mathsf{n}_i{:}C_i}$.

Proof: By induction on the definition of $\vdash_{\mathsf{B}}$.

(throw): Then the derivation looks like

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

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

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

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

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

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

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 trivial. . First:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\boxed{\qquad\qquad}\\ \Gamma \vdash [\![\mathsf{try}\ M; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}]\!]^{\lambda\mu} : C_i \mid \mathsf{n}_n{:}C_i, \overrightarrow{\mathsf{m}_j{:}B}\end{array}}{\Gamma \vdash \mu\mathsf{n}_n.[\mathsf{n}_n][\![\mathsf{try}\ M; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}]\!]^{\lambda\mu} : C_i \mid \overrightarrow{\mathsf{m}_j{:}B}}\,(\mu)}{\Gamma \vdash \mu\mathsf{n}_n.[\mathsf{n}_n][\![\mathsf{try}\ M; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}]\!]^{\lambda\mu} : (A_n \to C_i) \to C_i \mid \overrightarrow{\mathsf{m}_j{:}B}}\,(\to I) \qquad \dfrac{\begin{array}{c}\boxed{\qquad\qquad}\\ \Gamma, x{:}A_n \vdash [\![N_n]\!]^{\lambda\mu} : C_i \mid \overrightarrow{\mathsf{m}_j{:}B}\end{array}}{\Gamma \vdash \lambda x.[\![N_n]\!]^{\lambda\mu} : A_n \to C_i \mid \overrightarrow{\mathsf{m}_j{:}B}}\,(\to I)}{\Gamma \vdash (\mu\mathsf{n}_n.[\mathsf{n}_n][\![\mathsf{try}\ M; \overrightarrow{\mathsf{catch}\ \mathsf{n}_i(x) = N_i}]\!]^{\lambda\mu})\,(\lambda x.[\![N_n]\!]^{\lambda\mu}) : C_i \mid \overrightarrow{\mathsf{m}_j{:}B}}\,(\to E)$$

and second:

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

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

$$\dfrac{\dfrac{\begin{array}{c}\boxed{\qquad\qquad}\\ \Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i}, x{:}D \vdash [\![N]\!]^{\lambda\mu} : B \mid \overrightarrow{\mathsf{n}_i{:}C_i}\end{array}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash \lambda x.[\![N]\!]^{\lambda\mu} : D \to B \mid \overrightarrow{\mathsf{n}_i{:}C_i}}\,(\to I)}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash \lambda x.[\![N]\!]^{\lambda\mu} : D \to B \mid \mathsf{m}{:}B, \overrightarrow{\mathsf{n}_i{:}C_i}}\,(Wk)$$

Then $\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash [\![M']\!]^{\lambda\mu}\ \{\lambda x.[\![N]\!]^{\lambda\mu}/c_{\mathsf{m}}\} : B \mid \mathsf{m}{:}B, \overrightarrow{\mathsf{n}_i{:}C_i}$ follows by Lem. 1.13, and we can construct:

$$\dfrac{\begin{array}{c}\boxed{\qquad\qquad}\\ \Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash [\![M']\!]^{\lambda\mu}\{\lambda x.[\![N]\!]^{\lambda\mu}/c_{\mathsf{m}}\} : B \mid \mathsf{m}{:}B, \overrightarrow{\mathsf{n}_i{:}C_i}\end{array}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \to C_i} \vdash \mu\mathsf{m}.[\mathsf{m}][\![M']\!]^{\lambda\mu}\{\lambda x.[\![N]\!]^{\lambda\mu}/c_{\mathsf{m}}\} : B \mid \overrightarrow{\mathsf{n}_i{:}C_i}}\ (\mu) \qquad \square$$

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

5 Adding halt to λ^{try}

We will now define $\lambda^{\mathsf{try}}_{\mathsf{H}}$, a variant of λ^{try}, and 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 Sect. 5.2).

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 Sect. 5.2 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 $\lambda^{\mathsf{try}}_{\mathsf{H}}$ onto $\lambda\mu$-tp, where tp 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 $\perp$, but that would not be possible, as argued in Rem. 5.7.

We extend the calculus λ^{try} from Def. 3.1, by extending the set of pre-terms through adding the construct halt; the notion of reduction is defined as in Def. 3.2, by adding the rule that expresses that also halt consumes an applicative context.

Definition 5.1 (Syntax of $\lambda^{\mathsf{try}}_{\mathsf{H}}$) *i*) The set of *pre-terms* of $\lambda^{\mathsf{try}}_{\mathsf{H}}$ is defined by the grammar:

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

ii) The CBN-reduction system for $\lambda^{\mathsf{try}}_{\mathsf{H}}$ is like that for λ^{try} from Def. 3.2, defined by the rules:

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

CBN applicative evaluation contexts are defined as in Def. 3.2.

iii) The CBV-reduction system for $\lambda^{\mathsf{try}}_{\mathsf{H}}$ is that of CBN, replacing rule (β) by the first reduction rule below, and adding the second and third:

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

CBV applicative evaluation contexts are defined as in Def. 3.2.

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 type assignment for terms in $\lambda^{\mathsf{try}}_{\mathsf{H}}$ as follows.

Definition 5.2 (Type assignment for $\lambda^{\mathsf{try}}_{\mathsf{H}}$) Type assignment for terms in $\lambda^{\mathsf{try}}_{\mathsf{H}}$, $\vdash_{\mathsf{H}}$, is defined through the inference system:

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

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 $\perp$ that is used in $\lambda\mu$-tp. 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 (EFQ). Here we do not inhabit this rule with a term construct, as is done for example, in $\Lambda\mu$ [19] 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 5.3 (Subject reduction for $\vdash_{\mathsf{H}}$) *i) If $\Gamma \vdash_{\mathsf{H}} P : C \mid \Delta$ and $P \to^{\mathsf{N}}_{\mathsf{try}} Q$, then $\Gamma \vdash_{\mathsf{H}} Q : C \mid \Delta$.*
ii) If $\Gamma \vdash_{\mathsf{H}} P : C \mid \Delta$ and $P \to^{\mathsf{V}}_{\mathsf{try}} Q$, then $\Gamma \vdash_{\mathsf{H}} Q : C \mid \Delta$.

Proof: *i*) By induction on the definition $\to^{\mathsf{N}}_{\mathsf{try}}$. The proof is very similar to that of Thm. 4.4; we only show here the added case.

(halt): Then $P = \mathsf{halt}\, M \to \mathsf{halt} = Q$, and the derivation for P is shaped like:

$$
\dfrac{\dfrac{}{\Gamma \vdash \mathsf{halt} : A \to C \mid \Delta}\,(\mathsf{halt}) \qquad \dfrac{\boxed{\quad}}{\Gamma \vdash M : A \mid \Delta}}{\Gamma \vdash \mathsf{halt}\, M : C \mid \Delta}\,(\to E)
$$

We have $\Gamma \vdash_{\mathsf{H}} \mathsf{halt} : C \mid \Delta$ by rule (halt).

ii) Similar to that of the previous case and Thm. 4.4, but with the added case:

$(\mathsf{halt}_{\mathsf{V}})$: Then $P = V\, \mathsf{halt} \to \mathsf{halt} = Q$, and the derivation for P is constructed as:

$$
\dfrac{\dfrac{\boxed{\quad}}{\Gamma \vdash_{\mathsf{H}} V : A \to C \mid \Delta} \qquad \dfrac{}{\Gamma \vdash_{\mathsf{H}} \mathsf{halt} : A \mid \Delta}\,(\mathsf{halt})}{\Gamma \vdash_{\mathsf{H}} V\, \mathsf{halt} : C \mid \Delta}\,(\to E)
$$

Notice that we have $\Gamma \vdash_{\mathsf{H}} \mathsf{halt} : C \mid \Delta$ by rule (halt). □

So, in terms of type assignment for a λ-calculus with exceptions, the $\vdash_{\mathsf{H}}$ system satisfies the basic requirement with respect to CBN and CBV reduction.

We can interpret $\lambda^{\mathsf{try}}_{\mathsf{H}}$ in $\lambda\mu$-tp as follows:

Definition 5.4 (Interpretation of $\lambda^{\mathsf{try}}_{\mathsf{H}}$ into $\lambda\mu$-tp) *i*) We add the term constant *halt*[18] to $\lambda\mu$-tp that can only be assigned $\perp$ by adding the inference rule:

$$
(halt): \ \dfrac{}{\Gamma \vdash_{\mathsf{tp}} halt : \perp \mid \Delta}
$$

ii) The interpretation of $\lambda^{\mathsf{try}}_{\mathsf{H}}$ in $\lambda\mu$-tp is defined as follows:

[18] We use *halt* just as a place holder, it is not active in the reduction relation; in fact, any $\lambda\mu$-tp-term of type $\perp$ will do here.

$$\begin{array}{rcl}
\ulcorner x \lrcorner^{\mathsf{tp}} & \triangleq & x \\
\ulcorner \lambda x.M \lrcorner^{\mathsf{tp}} & \triangleq & \lambda x.\ulcorner M \lrcorner^{\mathsf{tp}} \\
\ulcorner MN \lrcorner^{\mathsf{tp}} & \triangleq & \ulcorner M \lrcorner^{\mathsf{tp}} \ulcorner N \lrcorner^{\mathsf{tp}} \\
\ulcorner \mathsf{throw\ n}(M) \lrcorner^{\mathsf{tp}} & \triangleq & \mu\_.[\mathsf{n}]\, c_{\mathsf{n}} \ulcorner M \lrcorner^{\mathsf{tp}} \\
\ulcorner \mathsf{try}\ M; \epsilon \lrcorner^{\mathsf{tp}} & \triangleq & \ulcorner M \lrcorner^{\mathsf{tp}} \\
\ulcorner \mathsf{try}\ M; \mathsf{Catch\_Block}; \mathsf{catch\ n}(x) = L \lrcorner^{\mathsf{tp}} & \triangleq & (\mu \mathsf{n}.[\mathsf{n}] \ulcorner \mathsf{try}\ M; \mathsf{Catch\_Block} \lrcorner^{\mathsf{tp}})\ \{\lambda x.\ulcorner L \lrcorner^{\mathsf{tp}} / c_{\mathsf{n}}\} \\
\ulcorner \mathsf{halt} \lrcorner^{\mathsf{tp}} & \triangleq & \mu\_.[\mathsf{tp}]\, \mathit{halt}
\end{array}$$

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

We can now show:

Theorem 5.5 (Soundness of the interpretation for $\lambda^{\mathsf{try}}_{\mathsf{H}}$)

i) If $P \rightarrow^{\mathrm{N}}_{\mathsf{H}} Q$, *then* $\ulcorner P \lrcorner^{\mathsf{tp}} \rightarrow^{\mathrm{N}*}_{\beta\mu} \ulcorner Q \lrcorner^{\mathsf{tp}}$.

ii) If $P \rightarrow^{\mathrm{V}}_{\mathsf{H}} Q$, *then* $\ulcorner P \lrcorner^{\mathsf{tp}} \rightarrow^{\mathrm{V}*}_{\beta\mu} \ulcorner Q \lrcorner^{\mathsf{tp}}$.

Proof: *i*) As in the proof of Thm. 3.6 (we drop the superscript on $\ulcorner \cdot \lrcorner^{\mathsf{tp}}$):

(halt): Then $P = \mathsf{halt}\, M \rightarrow^{\mathrm{N}}_{\mathsf{H}} \mathsf{halt} = Q$, and

$$\ulcorner \mathsf{halt}\, M \lrcorner^{\mathsf{tp}} \ \triangleq\ (\mu\_.[\mathsf{tp}]\, \mathit{halt}) \ulcorner M \lrcorner^{\mathsf{tp}} \ \rightarrow^{\mathrm{N}}_{\beta\mu} (1.16)\ \ \mu\_.[\mathsf{tp}]\, \mathit{halt} \ \triangleq\ \ulcorner \mathsf{halt} \lrcorner^{\mathsf{tp}}$$

ii) As the proof for the previous case, except for the rules:

(β_{V}): As in case (β) in the previous part.

($\mathsf{halt}_{\mathrm{V}}$): Then $P = V\, \mathsf{halt} \rightarrow \mathsf{halt} = Q$.

$$\ulcorner V\, \mathsf{halt} \lrcorner^{\mathsf{tp}} \ \triangleq\ \ulcorner V \lrcorner^{\mathsf{tp}} (\mu\_.[\mathsf{tp}]\, \mathit{halt}) \ \rightarrow^{\mathrm{V}}_{\beta\mu} (1.16)\ \ \mu\_.[\mathsf{tp}]\, \mathit{halt} \ \triangleq\ \ulcorner \mathsf{halt} \lrcorner^{\mathsf{tp}} \qquad \square$$

We can also show that assignable types are preserved.

Theorem 5.6 (Preservation of assignable types) *If* $\Gamma \vdash_{\mathsf{H}} M : B \mid \overrightarrow{\mathsf{n}_i{:}A_i \rightarrow C_i}$, *then we also have* $\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \rightarrow C_i} \vdash_{\mathsf{tp}} \ulcorner M \lrcorner : B \mid \overrightarrow{\mathsf{n}_i{:}C_i}$.

Proof: By induction on the definition of $\vdash_{\mathsf{H}}$, similar to that of Thm. 4.6, but with an added case.

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

$$\dfrac{\dfrac{}{\Gamma \vdash \mathit{halt} : \bot \mid \_{:}C, \Delta}\ (\mathit{halt})}{\Gamma \vdash \mu\_.[\mathsf{tp}]\, \mathit{halt} : C \mid \Delta}\ (\mathsf{tp})$$

for any pair of contexts, and $\ulcorner \mathsf{halt} \lrcorner = \mu\_.[\mathsf{tp}]\, \mathit{halt}$. $\square$

5.1 When typing halt with $\bot$

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 $\bot$ 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 $\lambda\mu$ or $\lambda\mu$-tp.

Remark 5.7 We can add the rules

$$(\mathsf{halt}) : \dfrac{}{\Gamma \vdash \mathsf{halt} : \bot \mid \Delta} \qquad (\rightarrow E_{\mathrm{N}}) : \dfrac{\Gamma \vdash M : \bot \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : \bot \mid \Delta}$$

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

$$(\rightarrow E_{\mathrm{V}}) : \dfrac{\Gamma \vdash M : A \mid \Delta \quad \Gamma \vdash N : \bot \mid \Delta}{\Gamma \vdash MN : \bot \mid \Delta}$$

for CBV.[19]

The problem appears in the proof of Thm. 5.6, where we would have the case

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

$$\dfrac{\dfrac{\boxed{}}{\Gamma \vdash_{\mathsf{H}} P : \bot \mid \overrightarrow{\mathsf{n}_i{:}A_i \rightarrow C_i}} \qquad \dfrac{\boxed{}}{\Gamma \vdash_{\mathsf{H}} Q : D \mid \overrightarrow{\mathsf{n}_i{:}A_i \rightarrow C_i}}}{\Gamma \vdash_{\mathsf{H}} PQ : \bot \mid \overrightarrow{\mathsf{n}_i{:}A_i \rightarrow C_i}}\;(\rightarrow E)$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \rightarrow C_i} \vdash_{\lambda\mu} \ulcorner P \lrcorner : \bot \mid \overrightarrow{\mathsf{n}_i{:}C_i}}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \rightarrow C_i} \vdash_{\lambda\mu} \ulcorner P \lrcorner : \bot \mid \_{:}D \rightarrow \bot, \overrightarrow{\mathsf{n}_i{:}C_i}}\;(Wk)}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \rightarrow C_i} \vdash_{\lambda\mu} \mu\_.[\mathsf{tp}]\ulcorner P \lrcorner : D \rightarrow \bot \mid \overrightarrow{\mathsf{n}_i{:}C_i}}\;(\mathsf{tp}) \qquad \dfrac{\boxed{}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \rightarrow C_i} \vdash_{\lambda\mu} \ulcorner Q \lrcorner : D \mid \overrightarrow{\mathsf{n}_i{:}C_i}}}{\Gamma, \overrightarrow{c_{\mathsf{n}_i}{:}A_i \rightarrow C_i} \vdash_{\lambda\mu} (\mu\_.[\mathsf{tp}]\ulcorner P \lrcorner)\ulcorner Q \lrcorner : \bot \mid \overrightarrow{\mathsf{n}_i{:}C_i}}\;(\rightarrow E)$$

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

So we cannot give the type $\bot$ the role it should have, which seems to contradict the motivation of [1].

In the next section, we will introduce a notion of type assignment that uses the type constant fail (which could be seen as $\bot$) 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 $\lambda\mu$-tp for that notion.

5.2 On catching halt

It might seem natural to define a failing computation through halt in much the same way as throw (and that is basically what is suggested in [1]). We will argue here that this cannot be expressed in $\lambda\mu$-tp.

Attempting this would lead to, contrary to what we have done above, adding panic $\mathsf{n}(N)$ as a construct, together with dedicated exception handlers abort $\mathsf{n}(x) = L$, so using, for example, the grammar:

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

and the (additional) reduction rules

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

for CBN, and the inference rules

[19] Notice that these two variants of $(\rightarrow E)$ would need to be added to achieve subject reduction.

$$(\mathsf{panic}):\ \frac{\Gamma \vdash M : A \mid \Delta}{\Gamma \vdash \mathsf{panic}\ \mathsf{n}(M) : \bot \mid \mathsf{n}{:}A \to \bot, \Delta}$$

$$(\to E_{\mathsf{P}}):\ \frac{\Gamma \vdash M : \bot \mid \Delta \quad \Gamma \vdash N : A \mid \Delta}{\Gamma \vdash MN : \bot \mid \Delta}$$

$$(\mathsf{try}_{\mathsf{P}}):\ \frac{\Gamma \vdash M : \bot \mid \overrightarrow{\mathsf{n}_i{:}A_i \to \bot}, \Delta \quad \Gamma, x{:}A_i \vdash N_i : \bot \mid \Delta\ (\forall i \in \underline{n})}{\Gamma \vdash \mathsf{try}\ M;\ \overrightarrow{\mathsf{abort}\ \mathsf{n}_i(x) = N_i} : \bot \mid \Delta}$$

As before, 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 means that we cannot return $\bot$ for a failing program without having to demand that all handlers return $\bot$, as we do in rule $(\mathsf{try}_{\mathsf{P}})$; this is why the control blocks are either a Catch_Block or an Abort_Block.

We extend the interpretation $\ulcorner \cdot \lrcorner^{\mathsf{tp}}$ with the cases:

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

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

$$\text{if } P \to_{\mathsf{H}} Q, \text{ then } \ulcorner P \lrcorner^{\mathsf{tp}} \to^*_{\beta\mu} \ulcorner Q \lrcorner^{\mathsf{tp}},$$

for either CBN or CBV-reduction. Although that property follows straightforwardly from the proof of Thm. 3.6, and for the additional case panic, the reduction rule try-panic throws a spanner in the works. Then the interpretation of the term

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

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

$$\begin{array}{ll}\ulcorner \mathsf{try}\ \mathsf{panic}\ \mathsf{n}(N); \mathsf{abort}\ \mathsf{n}(x) = M \lrcorner^{\mathsf{tp}} & \triangleq \\ (\mu \mathsf{n}.[\mathsf{n}] \ulcorner \mathsf{panic}\ \mathsf{n}(N) \lrcorner^{\mathsf{tp}})\ \{\lambda x.\ulcorner M \lrcorner^{\mathsf{tp}}/c_{\mathsf{n}}\} & \triangleq \\ (\mu \mathsf{n}.[\mathsf{n}]\, (\mu\_.[\mathsf{tp}]\, c_{\mathsf{n}} \ulcorner N \lrcorner^{\mathsf{tp}}))\ \{\lambda x.\ulcorner M \lrcorner^{\mathsf{tp}}/c_{\mathsf{n}}\} & = \\ \mu \mathsf{n}.[\mathsf{n}]\, \mu\_.[\mathsf{tp}]\, (\lambda x.\ulcorner M \lrcorner^{\mathsf{tp}}) \ulcorner N \lrcorner^{\mathsf{tp}} & \to^{\mathrm{N}}_{\beta\mu} \quad (E) \\ \mu\_.[\mathsf{tp}]\, (\lambda x.\ulcorner M \lrcorner^{\mathsf{tp}}) \ulcorner N \lrcorner^{\mathsf{tp}} & \to^{\mathrm{N}}_{\beta\mu} \\ \mu\_.[\mathsf{tp}]\, \ulcorner M \lrcorner^{\mathsf{tp}} \{\ulcorner N \lrcorner^{\mathsf{tp}}/x\} & = (3.5)\ \ \mu\_.[\mathsf{tp}]\, \ulcorner M\{N/x\} \lrcorner^{\mathsf{tp}}\end{array}$$

So here we have $\ulcorner P \lrcorner^{\mathsf{tp}} \to^{\mathrm{N}*}_{\beta\mu} \mu\_.[\mathsf{tp}]\, \ulcorner Q \lrcorner^{\mathsf{tp}}$, not $\ulcorner P \lrcorner^{\mathsf{tp}} \to^{\mathrm{N}*}_{\beta\mu} \ulcorner Q \lrcorner^{\mathsf{tp}}$ as we desired. Moreover, the terms $\mu\_.[\mathsf{tp}]\, \ulcorner Q \lrcorner^{\mathsf{tp}}$ and $\ulcorner Q \lrcorner^{\mathsf{tp}}$ are also computationally incompatible.[20]

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

$$(\mathsf{try\text{-}panic}):\quad \mathsf{try}\ \mathsf{panic}\ \mathsf{n}_l(N); \overrightarrow{\mathsf{abort}\ \mathsf{n}_i(x) = M_i}\ \to\ \mathsf{panic}\ \mathsf{n}_l(M_l\{N/x\}) \quad (\mathsf{n}_l \in \vec{\mathsf{n}_i})$$

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

$$(\mathsf{try\text{-}panic}):\quad \mathsf{try}\ \mathsf{panic}\ \mathsf{n}_l(N); \overrightarrow{\mathsf{abort}\ \mathsf{n}_i(x) = M_i}\ \to\ \mathsf{panic\text{-}top}(M_l\{N/x\}) \quad (\mathsf{n}_l \in \vec{\mathsf{n}_i})$$

which gets dealt with at the 'outermost level'. In fact, the interaction between the prefixes $\mu \mathsf{n}.[\mathsf{m}]$ as discussed in Sect. 2.3 gets disturbed; when mapping to $\lambda\mu$-tp we would need the prefix $\mu \mathsf{tp}.[\mathsf{tp}]$ at the 'outside', but are not allowed to bind tp.

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.

[20] Notice the similarity with the problem spotted in Rem. 5.7.

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 that can be assigned all types to the language that consumes all applicative contexts.

6 Handling failing computations

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. 5.7 and Sect. 5.2, this is not straightforward, and we will have to forgo on establishing a direct relation with $\lambda\mu$ or $\lambda\mu$-tp.

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 practice: programs can raise exceptions and panic from within the same try-statement. As we argued above, to satisfy 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 against 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 throws, called catch, all return the type of the main term, whereas handlers for panic calls, called abort, all return fail. It would, in principle, be possible to generalise this to more 'failure' types, but at the moment one will do.

The system we will present is thereby 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; we therefore lose the connection with logic for this system, since proof contraction (the equivalent of subject reduction in logic) does not change the proven formula. 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 whether reduction returns a term of type fail 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. On the other hand, the type system can predict if a panic is guaranteed to happen.

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 6.1 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 $A \to A \to A$, for any A (or $\forall \varphi\, .\varphi \to \varphi \to \varphi$). 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 in the present setting expressed through the rule:

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

Figure 4. Basic reduction rules for $\rightarrow_{\mathsf{F}}^{\mathsf{N}}$.

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

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, which would diminish expressivity of the system.

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 the conditional 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

$$\frac{\Gamma \vdash M : \mathsf{fail} \mid \Delta \quad \Gamma \vdash P : A \mid \Delta \quad \Gamma \vdash Q : B \mid \Delta}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q : \mathsf{fail} \mid \Delta}$$

(if running the boolean expression fails, the whole computation will fail)

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

(if M runs to false, the computation will fail, otherwise it runs to a term of type A)

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

(if M runs to true, the computation will fail, otherwise it runs to a term of type A).

We define $\lambda_{\mathsf{F}}^{\mathsf{try}}$ by extending the calculus λ^{try} from Def. 3.1, by adding panic and abort, a conditional construct and term constants to the set of pre-terms.

Definition 6.2 ($\lambda_{\mathsf{F}}^{\mathsf{try}}$) *i)* The set of pre-terms of $\lambda_{\mathsf{F}}^{\mathsf{try}}$ is defined through the grammar:

$$\begin{array}{rcl}
\mathsf{Handlers} & ::= & \epsilon \mid \mathsf{Handlers}; \mathsf{catch}\ \mathsf{m}(x) = M \mid \mathsf{Handlers}; \mathsf{abort}\ \mathsf{n}(x) = N \\
M, N & ::= & V \mid MN \mid \mathsf{try}\ M; \mathsf{Handlers} \mid \mathsf{throw}\,\mathsf{n}(M) \mid \mathsf{panic}\,\mathsf{n}(M) \\
 & \mid & \mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q \\
V & ::= & x \mid \mathsf{c} \mid \lambda x.M \\
\mathsf{c} & ::= & \mathsf{err} \mid \mathsf{true} \mid \mathsf{false} \mid 0 \mid 1 \mid \ldots \mid + \mid \times \mid \wedge \mid \vee \mid \ldots
\end{array}$$

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.

ii) CBN reduction $\rightarrow_{\mathsf{F}}^{\mathsf{N}}$ is defined as in Def. 3.2 by the rules in Fig. 4, and CBN *applicative*

$$(Ax):\ \frac{}{\Gamma,x{:}A \vdash x:A \mid \Delta} \qquad (\mathrm{c}):\ \frac{}{\Gamma \vdash \mathrm{c} : \sigma\mathrm{c} \mid \Delta} \qquad (\to I):\ \frac{\Gamma,x{:}A \vdash M:B \mid \Delta}{\Gamma \vdash \lambda x.M : A\to B \mid \Delta}$$

$$(\to E):\ \frac{\Gamma \vdash M:A\to B \mid \Delta \quad \Gamma \vdash N:A \mid \Delta}{\Gamma \vdash MN:B \mid \Delta} \qquad (\to E_{\mathsf{F}}):\ \frac{\Gamma \vdash M:\mathsf{fail} \mid \Delta \quad \Gamma \vdash N:A \mid \Delta}{\Gamma \vdash MN:\mathsf{fail} \mid \Delta}$$

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

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

$$(\mathsf{try}_{\mathsf{F}}):\ \frac{\Gamma \vdash M:\mathsf{fail} \mid \overrightarrow{\mathsf{n}_i{:}A_i\to B_i},\Delta \quad \Gamma,x{:}A_i \vdash N_i:B_i \mid \Delta\ (\forall i\in\underline{n})}{\Gamma \vdash \mathsf{try}\ M;\overrightarrow{\mathsf{handle}_i\ \mathsf{n}_i(x)=N_i}:\mathsf{fail} \mid \Delta}\ (\forall i\in\underline{n}\ (B_i=C \vee B_i=\mathsf{fail})\ )$$

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

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

Figure 5. The system $\vdash^{\mathrm{N}}_{\mathsf{F}}$.

evaluation contexts are defined as:

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

The operators $+$, $\times$, $\ldots$ are all pre-fix.

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.[21] In order to deal with this properly, we need to extend our notion of type assignment.

Definition 6.3 (Type assignment with throw and panic) *i*) We extend the set of types by adding (normal) ground types, ranged over by c:

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

and assume the function σ which assigns the appropriate ground type to each constant.

We also add the constant fail, but do not consider that a type as normal, since we do not want to let fail occur as a subtype; we will only allow the notation $A\to\mathsf{fail}$ as types for abort routines.

ii) CBN type assignment (with failure) $\vdash^{\mathrm{N}}_{\mathsf{F}}$ for terms in $\lambda^{\mathsf{try}}_{\mathsf{F}}$ is defined through the inference system in Fig. 5.

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 $\mathsf{panic\,n}(M)$ is typed with fail (so fails), and the rule $(\to E_{\mathsf{F}})$ propagates 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 $A \neq \mathsf{fail} \neq B$.

Remark 6.4 Although the derivation rule $(\to E_{\mathsf{F}})$ is clearly inspired by $\mathsf{fail} \leq A\to\mathsf{fail}$, for all A, or more generally by the logic rule (*EFQ*), we explicitly do not inhabit this rule with a term

[21] We could even add the term halt with type $\bot$ for this purpose, similar to the previous section, but refrain from doing so since we cannot assign any other type to a term that has type fail.

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

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

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

construct, as is implicitly done for the systems above in rules (throw) and (halt). Rather, we limit its use to just $(\to E_{\mathsf{F}})$. Our treatment thereby better corresponds to the characteristic of aborting computations. If we would allow, as above, the rule

$$(\mathsf{panic}) : \ \dfrac{\Gamma \vdash M : A \mid \Delta}{\Gamma \vdash \mathsf{panic}\,\mathsf{n}(M) : C \mid \mathsf{n}{:}A \to B, \Delta}$$

then it would be possible to assign an abortive computation any type, rather than just the one indicating that computation has failed, and we would no longer be able to distinguish between exceptions and panic through assignable types.

Example 6.5 We have (essentially) restricted the use of fail to panic only. For example, the term

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

is not typeable, since it would demand that the type for $\lambda xy.x$ contains fail. It would be typeable if we relax this restriction, and allow 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 $\vdash_{\mathsf{F}}$ 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.

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

But relaxing the restriction would take away the characteristic that the type fail indicates a failing execution, since we would have to allow $x{:}\mathsf{fail} \vdash x : \mathsf{fail}$. We therefore opt to have fewer typeable terms, thereby also enforcing a perhaps better programming style, in that the conditional structure cannot be simulated.

Using the conditional structure, the similar term

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

is typeable under the restriction, as shown in Fig. 7.

Notice that, in particular, the type assignment system forces the type of the body of an abort to have type fail as well, so running the body of each abort has to result in a panic as well.

Example 6.6 Remark that, for reasons discussed above, we explicitly do not consider rules like

$$\dfrac{\Gamma,x{:}A \vdash M : \mathsf{fail} \mid \Delta}{\Gamma \vdash \lambda x.M : \mathsf{fail} \mid \Delta} \qquad \dfrac{\Gamma,x{:}A \vdash M : \mathsf{fail} \mid \Delta}{\Gamma \vdash \lambda x.M : A \to \mathsf{fail} \mid \Delta} \qquad \dfrac{\Gamma \vdash M : \mathsf{fail} \mid \Delta}{\Gamma \vdash M : B \mid \Delta}$$

Moreover, assume we would add the first of the above rules and assume we can derive:

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

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

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

$$\dfrac{\dfrac{\dfrac{\boxed{}}{\Gamma, x{:}A \vdash M : \mathsf{fail} \mid \Delta}}{\Gamma \vdash \lambda x.M : \mathsf{fail} \mid \Delta}\,(\lambda\text{-}\mathsf{fail}) \quad \dfrac{\boxed{}}{\Gamma \vdash N : B \mid \Delta}}{\Gamma \vdash (\lambda x.M)N : \mathsf{fail} \mid \Delta}\,({\to}E_{\mathsf{F}})$$

Remark that now we cannot apply Lem. 6.8 to conclude $\Gamma \vdash_{\mathsf{F}} M\{N/x\} : \mathsf{fail} \mid \Delta$, since we cannot be sure that $A = B$; therefore we would not be able to show soundness.

We can show that weakening and thinning are both admissible.

Lemma 6.7 The following rules are admissible in $\vdash^{\mathsf{N}}_{\mathsf{F}}$:

$$(Wk):\ \dfrac{\Gamma \vdash M : A \mid \Delta}{\Gamma' \vdash M : A \mid \Delta'}\ (\Gamma \subseteq \Gamma', \Delta \subseteq \Delta')$$
$$(Th):\ \dfrac{\Gamma \vdash M : A \mid \Delta}{\Gamma' \vdash M : A \mid \Delta'}\ (\Gamma' = \{\, x{:}B \in \Gamma \mid x \in fv(M) \,\},\ \Delta' = \{\, \mathsf{n}{:}B \in \Delta \mid \mathsf{n} \in fn(M) \,\})$$

Proof: Standard. □

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

Lemma 6.8 (Substitution lemma for $\vdash^{\mathsf{N}}_{\mathsf{F}}$) *If* $\Gamma, x{:}C \vdash^{\mathsf{N}}_{\mathsf{F}} M : A \mid \Delta$ *and* $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} N : C \mid \Delta$, *then* $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M\{N/x\} : A \mid \Delta$.

Proof: By induction on the structure of terms.

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

Proof: The result follows by induction on the definition $\to^{\mathsf{N}*}_{\mathsf{F}}$; we focus on the single step reduction, and only show the interesting cases

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

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

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

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

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

so $B_l = \mathsf{fail}$. In particular, we have derivations for both $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M : A_l \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B_i},\Delta$ and $\Gamma,x{:}A_l \vdash^{\mathsf{N}}_{\mathsf{F}} N_l : \mathsf{fail} \mid \Delta$.[22] By Lem. 6.7, we can remove $\overrightarrow{\mathsf{n}_i{:}A_i \to B_i}$ from the co-context for the first to obtain $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M : A_l \mid \Delta$. Then, by Lem. 6.8, we obtain $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} N_l\{M/x\} : \mathsf{fail} \mid \Delta$.

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

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

and either:

$((A = B = C) \vee (A = C \wedge B = \mathsf{fail}))$: Then, in particular, $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} Q : C \mid \Delta$.

$(A = \mathsf{fail} \wedge B = C)$: Then, in particular, $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} Q : \mathsf{fail} \mid \Delta$.

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

$$\dfrac{\dfrac{\dfrac{\boxed{\;\mathcal{D}\;}}{\Gamma \vdash R : E \mid \Delta'}}{\Gamma \vdash \mathsf{throw}\,\mathsf{n}(R) : \mathsf{bool} \mid \mathsf{n}{:}E \to F,\Delta'}\,(\mathsf{throw}) \quad \dfrac{\boxed{}}{\Gamma \vdash M : B \mid \Delta} \quad \dfrac{\boxed{}}{\Gamma \vdash N : C \mid \Delta}}{\Gamma \vdash \mathsf{if}\;\mathsf{throw}\,\mathsf{n}(R)\;\mathsf{then}\;M\;\mathsf{else}\;N : D \mid \Delta}\,(\mathsf{cond})$$

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

$$\dfrac{\dfrac{\boxed{\;\mathcal{D}\;}}{\Gamma \vdash R : E \mid \Delta'}}{\Gamma \vdash \mathsf{throw}\,\mathsf{n}(R) : D \mid \mathsf{n}{:}E \to F,\Delta'}\,(\mathsf{throw})$$

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

$$\dfrac{\dfrac{\boxed{}}{\Gamma \vdash \mathsf{panic}\,\mathsf{n}(R) : \mathsf{fail} \mid \Delta} \quad \dfrac{\boxed{}}{\Gamma \vdash M : B \mid \Delta} \quad \dfrac{\boxed{}}{\Gamma \vdash N : C \mid \Delta}}{\Gamma \vdash \mathsf{if}\;\mathsf{panic}\,\mathsf{n}(R)\;\mathsf{then}\;M\;\mathsf{else}\;N : \mathsf{fail} \mid \Delta}\,(\mathsf{cond}_{\mathsf{F}})$$

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

6.1 On CBV-reduction for $\lambda^{\mathsf{try}}_{\mathsf{F}}$

As above, we can define a notion of CBV-reduction for $\lambda^{\mathsf{try}}_{\mathsf{F}}$, through:

Definition 6.10 CBV reduction $\to^{\mathsf{V}}_{\mathsf{F}}$ is defined as in Def. 6.2 by also changing/adding the rule

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

CBV applicative contexts are defined as:

$$\mathsf{C}^{\mathsf{A}}_{\mathsf{V}} \;::=\; [\,] \mid \mathsf{C}^{\mathsf{A}}_{\mathsf{V}} M \mid V \mathsf{C}^{\mathsf{A}}_{\mathsf{V}} \mid \mathsf{try}\;\mathsf{C}^{\mathsf{A}}_{\mathsf{V}};\mathsf{Handlers} \mid \mathsf{if}\;\mathsf{C}^{\mathsf{A}}_{\mathsf{V}}\;\mathsf{then}\;P\;\mathsf{else}\;Q$$

If we now look at type assignment, a particular feature pops up, in that we need to add an extra type assignment rule (as we did for CBV-$\lambda\mu$).

Definition 6.11 The CBV notion $\vdash^{\mathsf{V}}_{\mathsf{F}}$ is for $\lambda^{\mathsf{try}}_{\mathsf{F}}$ defined using the rules of Def. 6.3, extended with

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

the rule

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

This was not necessary for $\vdash_{\mathsf{B}}$ nor $\vdash_{\mathsf{H}}$, since there the use of either throw of halt is hidden; since our intention is that the use of panic is not, as in Rem. 5.7 we have to add this type assignment rule. Notice that we need to formulate rule $(\to E_{\mathrm{V}})$ using M rather than a value, since we can have:

$$(\lambda x.x)\,(\lambda x.x)(\mathsf{panic\,n}(N)) \;\to^{\mathsf{v}}_{\mathsf{F}}\; (\lambda x.x)\,(\mathsf{panic\,n}(N)) \;\to^{\mathsf{v}}_{\mathsf{F}}\; \mathsf{panic\,n}(N)$$

This gives a well-behaved system, in the sense that it is straightforward to show soundness.

Lemma 6.12 (Substitution lemma for $\vdash^{\mathsf{v}}_{\mathsf{F}}$) *If $\Gamma,x{:}C \vdash^{\mathsf{v}}_{\mathsf{F}} M : A \mid \Delta$ and $\Gamma \vdash^{\mathsf{v}}_{\mathsf{F}} N : C \mid \Delta$, then $\Gamma \vdash^{\mathsf{v}}_{\mathsf{F}} M\{N/x\} : A \mid \Delta$.*

Proof: By induction on the structure of terms. The proof is much the same as for Thm. 6.8, except for one case:

$(M \equiv PQ)$: Then $\Gamma,x{:}C \vdash^{\mathsf{v}}_{\mathsf{F}} M : A \mid \Delta$ is typed using either $(\to E)$, $(\to E_{\mathsf{F}})$, or $(\to E_{\mathrm{V}})$, so we have the same cases as before, extended by:

$(\to E_{\mathrm{V}})$: Then $A = \mathsf{fail}$, and $\Gamma,x{:}C \vdash^{\mathsf{N}}_{\mathsf{F}} P : B \mid \Delta$ for some B, and $\Gamma,x{:}C \vdash^{\mathsf{N}}_{\mathsf{F}} Q : \mathsf{fail} \mid \Delta$. By induction both $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} P\{N/x\} : B \mid \Delta$ and $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} Q\{N/x\} : \mathsf{fail} \mid \Delta$; then, by $(\to E_{\mathrm{V}})$, we have $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} P\{N/x\}\; Q\{N/x\} : \mathsf{fail} \mid \Delta$. $\square$

We can now show:

Theorem 6.13 (Soundness for $\vdash^{\mathsf{v}}_{\mathsf{F}}$ with respect to $\to^{\mathsf{v}}_{\mathsf{F}}$) *If $\Gamma \vdash^{\mathsf{v}}_{\mathsf{F}} P : C \mid \Delta$ and $P \to^{\mathsf{v}*}_{\mathsf{F}} Q$, then either $\Gamma \vdash^{\mathsf{v}}_{\mathsf{F}} Q : C \mid \Delta$, or $\Gamma \vdash^{\mathsf{v}}_{\mathsf{F}} Q : \mathsf{fail} \mid \Delta$.*

Proof: The proof is much like that for Thm. 6.9, with the addition of:

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

$$\dfrac{\dfrac{\boxed{}}{\Gamma \vdash_{\mathsf{F}} V : E \to F \mid \mathsf{n}{:}A \to C,\Delta'} \qquad \dfrac{\dfrac{\boxed{\;\mathcal{D}\;}}{\Gamma \vdash_{\mathsf{F}} N : A \mid \Delta'}}{\Gamma \vdash_{\mathsf{F}} \mathsf{throw\,n}(N) : E \mid \mathsf{n}{:}A \to C,\Delta'}\,(\mathsf{throw})}{\Gamma \vdash_{\mathsf{F}} V(\mathsf{throw\,n}(N)) : F \mid \mathsf{n}{:}A \to C,\Delta'}\,(\to E)$$

We can construct the derivation for Q:

$$\dfrac{\dfrac{\boxed{\;\mathcal{D}\;}}{\Gamma \vdash_{\mathsf{F}} N : A \mid \Delta'}}{\Gamma \vdash_{\mathsf{F}} \mathsf{throw\,n}(N) : F \mid \mathsf{n}{:}A \to C,\Delta'}\,(\mathsf{throw})$$

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

$$\dfrac{\dfrac{\boxed{}}{\Gamma \vdash V : A \mid \Delta} \qquad \dfrac{\boxed{}}{\Gamma \vdash \mathsf{panic\,n}(N) : \mathsf{fail} \mid \Delta}}{\Gamma \vdash V(\mathsf{panic\,n}(N)) : \mathsf{fail} \mid \Delta}\,(\to E_{\mathrm{V}})$$

We have $\Gamma \vdash^{\mathsf{v}}_{\mathsf{F}} \mathsf{panic\,n}(N) : \mathsf{fail} \mid \Delta$ in a sub-derivation.

However, we lose the property of the predictive character of the type fail. For example, we can have a derivation like (where $x{:}B \to A, y{:}B \in \Gamma$)

$$\dfrac{\dfrac{}{\Gamma \vdash x : B{\to}A \mid \Delta}\,(Ax) \quad \dfrac{}{\Gamma \vdash y : B \mid \Delta}\,(Ax)}{\Gamma \vdash xy : A \mid \Delta}\,({\to}E) \qquad \dfrac{\boxed{\phantom{\Gamma \vdash \mathsf{panic\,n}(N) : \mathsf{fail}}}}{\Gamma \vdash \mathsf{panic\,n}(N) : \mathsf{fail} \mid \Delta}$$
$$\overline{\qquad\qquad \Gamma \vdash xy(\mathsf{panic\,n}(N)) : \mathsf{fail} \mid \Delta \qquad\qquad}\,({\to}E_{\mathrm{V}})$$

but this term is in CBV-normal form, so will never reduce to $\mathsf{panic\,n}(N)$ as suggested by the type fail. Notice that it is not typeable in $\vdash^{\mathsf{N}}_{\mathsf{F}}$.

Moreover, it is not clear how to define a notion of principal typeing for this system, as we do in the next section for $\vdash^{\mathsf{N}}_{\mathsf{F}}$, because using the standard approach, through unification, we would reject to type $xy(\mathsf{panic\,n}(N))$.

7 Principal typing for $\vdash^{\mathsf{N}}_{\mathsf{F}}$

In this section, we will show that we can extend the notion of principal typing from Sect. 1.1 in a natural way to $\vdash^{\mathsf{N}}_{\mathsf{F}}$. We will define the algorithm pt_{F} that calculates the principal typing for each term typeable in $\vdash^{\mathsf{N}}_{\mathsf{F}}$; as for pt_{C} from Def. 1.5, it is defined using a notion of unification and substitution of type variables by types.

We first extend the notions of substitution and unification to our notion of types:

Definition 7.1 *i*) The *substitution* $(\varphi \mapsto C)$, where φ is a type variable and C a type (not equal to fail), is inductively defined over the structure of types as in Def. 1.3 by adding:

$$(\varphi \mapsto C)\ \mathrm{c} \;=\; \mathrm{c}$$

ii) Unification on λ^{try}-types with fail is defined as as in Def. 1.3, by adding:

$$\mathit{unify}\ \ c\ \ c \;=\; \mathit{Id}_S$$

As above, this definition implies that all non-specified cases, like unify int bool, or $\mathit{unify}\ \varphi$ fail, do not return a substitution.

Lemma 7.2 (Soundness of substitution) *If $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M : A \mid \Delta$, then $S\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M : SA \mid S\Delta$.*

Proof: By straightforward induction on the structure of derivations. □

We now define a notion of principal typing for terms of $\lambda^{\mathsf{try}}_{\mathsf{F}}$.

Definition 7.3 The principal typing algorithm for $\vdash^{\mathsf{N}}_{\mathsf{F}}$ is given by:

$$\begin{array}{l}
pt_{\mathsf{F}}\,\mathrm{c} \;=\; \langle \emptyset; \sigma\,\mathrm{c}; \emptyset \rangle \\
pt_{\mathsf{F}}\,x \;=\; \langle x{:}\varphi; \varphi; \emptyset \rangle \\
\qquad \text{where } \varphi \ \ \mathit{is}\ \ \mathit{fresh} \\
pt_{\mathsf{F}}\,(\lambda x.M) \;=\; \langle \Gamma; C; \Delta \rangle \\
\qquad \text{where } \langle \Gamma'; P; \Delta \rangle \;=\; pt_{\mathsf{F}}\,M^{\ (i)} \\
\qquad\qquad\qquad \Gamma; C \;=\; \begin{cases} \Gamma' \backslash x; A \to P & (x{:}A \in \Gamma') \\ \Gamma'; \varphi \to P & (x \notin \Gamma') \end{cases} \\
\qquad\qquad\qquad \varphi \ \ \mathit{is}\ \ \mathit{fresh} \\
pt_{\mathsf{F}}\,MN \;=\; S_3 \circ S_2 \circ S_1\, \langle \Gamma_1 \cup \Gamma_2; \varphi; \Delta_1 \cup \Delta_2 \rangle \\
\qquad \text{where } \langle \Gamma_1; P_1; \Delta_1 \rangle \;=\; pt_{\mathsf{F}}\,M \\
\qquad\qquad\quad\ \langle \Gamma_2; P_2; \Delta_2 \rangle \;=\; pt_{\mathsf{F}}\,N \\
\qquad\qquad\qquad\qquad S_1 \;=\; \begin{cases} \mathit{unify}\ P_1\ P_2 \to \varphi & (P_1 \neq \mathsf{fail}) \\ (\varphi \mapsto \mathsf{fail})\ {}^{P}{}_{(ii)} & (P_1 = \mathsf{fail}) \end{cases} \\
\qquad\qquad\qquad\qquad S_2 \;=\; \mathit{unifyC}\ (S_1\,\Gamma_1)\ (S_1\,\Gamma_2) \\
\qquad\qquad\qquad\qquad S_3 \;=\; \mathit{unifyC}\ (S_2 \circ S_1\,\Delta_1)\ (S_2 \circ S_1\,\Delta_2) \\
\qquad\qquad\qquad\qquad \varphi \ \ \mathit{is}\ \ \mathit{fresh}
\end{array}$$

$$pt_{\mathsf{F}}\,(\mathsf{try}\ M;\epsilon) \;=\; pt_{\mathsf{F}}\,M$$

$$\begin{array}{l}
pt_{\mathsf{F}}\,(\mathsf{try}\ M;\mathsf{Handlers};\mathsf{catch}\ \mathsf{n}(x) = N) \;=\; S_3{\circ}S_2{\circ}S_1\,\langle \Gamma_1 \cup \Gamma_2; A; \Delta_1 \backslash \mathsf{n} \cup \Delta_2\rangle \\
\quad \text{where } \begin{array}[t]{rcl}
\langle \Gamma_1; P_1; \Delta_1\rangle &=& pt_{\mathsf{F}}\,(\mathsf{try}\ M;\mathsf{Handlers}) \\
\langle \Gamma_2'; P_2; \Delta_2\rangle &=& pt_{\mathsf{F}}\,N \qquad\qquad\qquad\qquad (P_2 \neq \mathsf{fail}) \\
\Gamma_2; C &=& \begin{cases} \Gamma_2' \backslash x; D \rightarrow P_2 & (x{:}D \in \Gamma_2') \\ \Gamma_2'; \varphi \rightarrow P_2 & (x \notin \Gamma_2') \end{cases} \\
S_1 &=& \begin{cases} \mathit{unify}\ P_1\ P_2 & (P_1 \neq \mathsf{fail}) \\ \mathit{Id}_S & (P_1 = \mathsf{fail}) \end{cases} \\
S_2 &=& \mathit{unifyC}\ (S_1\,\Gamma_1)\ (S_1\,\Gamma_2) \\
S_3 &=& \mathit{unifyC}\ (S_2{\circ}S_1\,\Delta_1)\ (S_2{\circ}S_1(\mathsf{n}{:}C,\Delta_2))\ ^{(iii)} \quad (\mathsf{n} \notin \Delta_2) \\
\varphi && \mathit{is\ fresh}
\end{array}
\end{array}$$

$$\begin{array}{l}
pt_{\mathsf{F}}\,(\mathsf{try}\ M;\mathsf{Handlers};\mathsf{abort}\ \mathsf{n}(x) = N) \;=\; S_2{\circ}S_1\,\langle \Gamma_1 \cup \Gamma_2; P; \Delta_1 \backslash \mathsf{n} \cup \Delta_2\rangle \\
\quad \text{where } \begin{array}[t]{rcl}
\langle \Gamma_1; P; \Delta_1\rangle &=& pt_{\mathsf{F}}\,(\mathsf{try}\ M;\mathsf{Handlers}) \\
\langle \Gamma_2'; \mathsf{fail}; \Delta_2\rangle &=& pt_{\mathsf{F}}\,N\ ^{(iv)} \\
\Gamma_2; C &=& \begin{cases} \Gamma_2' \backslash x; D \rightarrow \mathsf{fail} & (x{:}D \in \Gamma_2') \\ \Gamma_2'; \varphi \rightarrow \mathsf{fail} & (x \notin \Gamma_2') \end{cases} \\
S_1 &=& \mathit{unifyC}\ \Gamma_1\ \Gamma_2 \\
S_2 &=& \mathit{unifyC}\ (S_1\,\Delta_1)\ (S_1(\mathsf{n}{:}C,\Delta_2)) \quad (\mathsf{n} \notin \Delta_2) \\
\varphi && \mathit{is\ fresh}
\end{array}
\end{array}$$

$$\begin{array}{l}
pt_{\mathsf{F}}\,(\mathsf{throw}\ \mathsf{n}(N)) \;=\; \langle \Gamma; \varphi; \mathsf{n}{:}P \rightarrow \varphi', \Delta\rangle \\
\quad \text{where } \begin{array}[t]{rcl}
\langle \Gamma; P; \Delta\rangle &=& pt_{\mathsf{F}}\,N \\
\varphi, \varphi' && \mathit{are\ fresh}
\end{array}
\end{array}$$

$$\begin{array}{l}
pt_{\mathsf{F}}\,(\mathsf{panic}\ \mathsf{n}(N)) \;=\; \langle \Gamma; \mathsf{fail}; \mathsf{n}{:}P \rightarrow \mathsf{fail}, \Delta\rangle \\
\quad \text{where } \langle \Gamma; P; \Delta\rangle \;=\; pt_{\mathsf{F}}\,N
\end{array}$$

$$\begin{array}{l}
pt_{\mathsf{F}}\,(\mathsf{if}\ Q\ \mathsf{then}\ R\ \mathsf{else}\ S) \;=\; S_4{\circ}S_3{\circ}S_2{\circ}S_1\,\langle \Gamma_1 \cup \Gamma_2 \cup \Gamma_3; \varphi; \Delta_1 \cup \Delta_2 \cup \Delta_3\rangle \\
\quad \text{where } \begin{array}[t]{rcl}
\langle \Gamma_1; P_1; \Delta_1\rangle &=& pt_{\mathsf{F}}\,Q \\
\langle \Gamma_2; P_2; \Delta_2\rangle &=& pt_{\mathsf{F}}\,R \\
\langle \Gamma_3; P_3; \Delta_3\rangle &=& pt_{\mathsf{F}}\,S \\
S_1 &=& \begin{cases} \mathit{unify}\ P_1\ \mathsf{bool} & (P_1 \neq \mathsf{fail}) \\ (\varphi \mapsto \mathsf{fail}) & (P_1 = \mathsf{fail}) \end{cases} \\
S_2 &=& \begin{cases} \left.\begin{array}{ll} (\mathit{unify}\ (S_1\,P_2)\ (S_1\,P_3)){\circ}(\varphi \mapsto S_1\,P_2) & \\ & (P_2 \neq \mathsf{fail} \neq P_3) \\ (\varphi \mapsto S_1\,P_2) & (P_2 \neq \mathsf{fail} = P_3) \\ (\varphi \mapsto S_1\,P_3) & (P_2 = \mathsf{fail} \neq P_3) \\ (\varphi \mapsto \mathsf{fail}) & (P_2 = \mathsf{fail} = P_3) \end{array}\right\} & (P_1 \neq \mathsf{fail}) \\ \mathit{Id}_S & (P_1 = \mathsf{fail}) \end{cases} \\
S_3 &=& \mathit{unifyC}\ (S_2{\circ}S_1\,\Gamma_1)\ (S_2{\circ}S_1\,\Gamma_2)\ (S_2{\circ}S_1\,\Gamma_3) \\
S_4 &=& \mathit{unifyC}\ (S_3{\circ}S_2{\circ}S_1\,\Delta_1)\ (S_3{\circ}S_2{\circ}S_1\,\Delta_2)\ (S_3{\circ}S_2{\circ}S_1\,\Delta_3) \\
\varphi && \mathit{is\ fresh}
\end{array}
\end{array}$$

Notes:

i) If $P = \mathsf{fail}$, then the algorithm returns no result.

ii) Officially, $(\varphi \mapsto \mathsf{fail})$ is not a substitution; but since φ occurs only as the placeholder for the return type, we can safely abuse our notation here.

iii) We do not have to consider $\mathsf{n} \in \Delta_1$ or not separately; this call to *unifyC* will create the correct type for n.

iv) If $pt_{\mathsf{F}}\,N$ does not produce fail, the algorithm fails.

We can show that the algorithm creates valid judgements:

Lemma 7.4 (Soundness of pt_{F}) *If* $pt_{\mathsf{F}}\,M = \langle \Gamma; P; \Delta\rangle$, *then* $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M : P \mid \Delta$.

Proof: By induction on the structure of terms, using Lem. 7.2.

We will now show the main result for pt_{F}, which states that it calculates the most general typeing with respect to type substitution for all terms typeable in $\vdash^{\mathsf{N}}_{\mathsf{F}}$.

Theorem 7.5 (Completeness of substitution.) *If $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M : A \mid \Delta$, then there are contexts Γ' and Δ', type P and a substitution S such that: $pt_{\mathsf{F}}\, M = \langle \Gamma' ; P ; \Delta' \rangle$, and $S\Gamma' \subseteq \Gamma$, $S\Delta' \subseteq \Delta$, and $SP = A$.*

This last result shows the practicality of our notion of type assignment.

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 $\lambda\mu$-tp. 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 distinct handling of exception and panic calls for formal calculi in a computationally meaningful way. This was emphasised by showing that our system has the principal typeing property, a prerequisite for its use in programming.

Acknowledgements

I am greatly indebted to the students who, over the years, have worked with me on this and other problems related to classical logic and functional programming. 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. A Proof-Theoretic Foundation of Abortive Continuations. In *Proceedings of Higher-Order and Symbolic Computation, 2007*, pages 403–429, 2007.

[2] S. van Bakel. Characterisation of Normalisation Properties for $\lambda\mu$ using Strict Negated Intersection Types. *ACM Transactions on Computational Logic*, 19, 2018.

[3] S. van Bakel. Exception Handling and Classical Logic. In E. Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 21:1–21:14. ACM, 2019.

[4] S. van Bakel, F. Barbanera, and U. de'Liguoro. Intersection Types for the $\lambda\mu$-calculus. *Logical Methods in Computer Science*, 141(1), 2018.

[5] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[6] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[7] G.M. Bierman. A Computational Interpretation of the $\lambda\mu$-calculus. In *Proceedings of* Symposium on Mathematical Foundations of Computer Science, volume 1450 of *Lecture Notes in Computer Science*, pages 336–345. Springer Verlag, 1998.

[8] G.M. Bierman. A Computational Interpretation of the $\lambda\mu$-calculus. Technical report, University of Cambridge, 1998. Expanded version of [7].

[9] L.E.J. Brouwer. *Over de Grondslagen der Wiskunde*. PhD thesis, Faculteit der Wiskunde en Informatica, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, 1907.

[10] L.E.J. Brouwer. De onbetrouwbaarheid der logische principes. *Tijdschrift voor Wijsbegeerte*, 2:152–158, 1908.

[11] L.E.J. Brouwer. Unreliability of the Logical Principles. In A. Heyting, editor, *Collected Works 1. Philosophy and Foundations of Mathematics*. North-Holland, Amsterdam, 1975.

[12] T. Crolard. A confluent lambda-calculus with a catch/throw mechanism. *Journal of Functional Programming*, 9(6):625–647, 1999.

[13] H.B. Curry. Functionality in Combinatory Logic. In *Proc. Nat. Acad. Sci. U.S.A*, volume 20, pages 584–590, 1934.

[14] S. Drossopoulou and T. Valkevych. Java Exceptions Throw No Surprises. Unpublished, March 2000.

[15] M. Felleisen and R Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2), 1992.

[16] G. Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39(2):176–210 and 405–431, 1935.

[17] J. Gosling, W.N. Joy, and G.L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[18] T. Griffin. A formulae-as-types notion of control. In *Proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, Orlando (Fla., USA)*, pages 47–58, 1990.

[19] Ph. de Groote. On the Relation between the $\lambda\mu$-Calculus and the Syntactic Theory of Sequential Control. In *Proceedings of 5th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'94)*, volume 822 of *Lecture Notes in Computer Science*, pages 31–43. Springer Verlag, 1994.

[20] Ph. de Groote. A Simple Calculus of Exception Handling. In M. Dezani-Ciancaglini and G.D. Plotkin, editors, *Second International Conference on* Typed Lambda Calculi and Applications*, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, volume 902 of *Lecture Notes in Computer Science*, pages 201–215. Springer Verlag, 1995.

[21] W.A. Howard. The Formula-as-Types Notion of Construction. In J.P. Seldin and J.R. Hindley, editors, *To H. B. Curry, Essays in Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic press, New York, 1980, 1980.

[22] 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. Report on the Programming Language Haskell. *ACM SIGPLAN Notices*, 27(5):1–64, 1992.

[23] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[24] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[25] H. Nakano. The Non-deterministic Catch and Throw Mechanism and Its Subject Reduction Property. In N.D. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language and Computation, Festschrift in Honor of Satoru Takasu*, volume 792 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 1994.

[26] H. Nakano. *Logical Structures of the Catch and Throw Mechanism*. PhD thesis, University of Tokyo, 1995.

[27] C.-H.L. Ong and C.A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proceedings of the 24th Annual ACM Symposium on Principles Of Programming Languages*, pages 215–227, 1997.

[28] M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proceedings of 3rd International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'92)*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Verlag, 1992.

[29] M. Parigot. Classical Proofs as Programs. In *Kurt Gödel Colloquium*, pages 263–276, 1993. Presented at TYPES Workshop, at Băstad, June 1992.

[30] F. Pessaux and X. Leroy. Type-Based Analysis of Uncaught Exceptions. In A.W. Appel and A. Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 276–290. ACM, 1999.

[31] D. Rémy. Using, Understanding, and Unraveling the OCaml Language. From Practice To Theory and Vice Versa. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science*, pages 413–536. Springer, 2000.

[32] J.A. Robinson. A Machine-Oriented Logic Based on Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[33] Th. Streicher and B. Reus. Classical logic: Continuation Semantics and Abstract Machines. *Journal of Functional Programming*, 11(6):543–572, 1998.

[34] M.E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1969.

Appendix A Proofs

Proof of 7.5. By induction on the structure of terms in Λ.

$(M \equiv x)$: Then, by rule (Ax), $x{:}A \in \Gamma$, and $pt_{\mathsf{F}}\, x = \langle \{x{:}\varphi\}; \varphi; \emptyset \rangle$. Take $S = (\varphi \mapsto A)$.

$(M \equiv \mathrm{c})$: Then, by rule (c), $A = \sigma\mathrm{c}$, and $pt_{\mathsf{F}}\, \mathrm{c} = \langle \emptyset; \sigma\mathrm{c}; \emptyset \rangle$. Take $S = Id_S$.

$(M \equiv \lambda x.N)$: Then, by rule $(\to I)$, there are C,D such that $A = C \to D$, and $\Gamma, x{:}C \vdash^{\mathsf{N}}_{\mathsf{F}} N : D \mid \Delta$. Then, by induction, there are Γ'', Δ'', P' and S' such that $pt_{\mathsf{F}}\, N = \langle \Gamma''; P'; \Delta'' \rangle$, and $S'\,\Gamma'' \subseteq \Gamma, x{:}C$, $S'\,\Delta'' \subseteq \Delta$, $S\,P' = D$. Then either:

$(x \in fv(N))$: Then $x{:}A' \in \Gamma''$, and $pt_{\mathsf{F}}\,(\lambda x.N) = \langle \Gamma'' \backslash x; A' \to P'; \Delta \rangle$. Since $S'\,\Gamma'' \subseteq \Gamma, x{:}C$, in particular $S'\,A' = C$, $S'\,(\Gamma'' \backslash x) \subseteq \Gamma$, and $S'\,(A' \to P') = C \to D$. Take $\Gamma' = \Gamma'' \backslash x$, $\Delta' = \Delta''$, $P = A' \to P'$, and $S = S'$.

$(x \notin fv(N))$: Then $pt_{\mathsf{F}}\,(\lambda x.N) = \langle \Gamma''; \varphi \to P'; \Delta \rangle$, x does not occur in Γ'', and φ does not occur in $\langle \Gamma''; P'; \Delta \rangle$. Since $S'\,\Gamma'' \subseteq \Gamma, x{:}C$, in particular $S'\,\Gamma'' \subseteq \Gamma$. Take $S = S' \circ (\varphi \mapsto C)$, then, since φ does not occur in Γ'', also $S\Gamma'' \subseteq \Gamma$. Notice that $S(\varphi \to P') = C \to D$; take $\Gamma' = \Gamma''$, $\Delta' = \Delta''$, and $P = \varphi \to P'$.

$(M \equiv QR)$: Then, by rule $(\to E)$, there exists a B such that $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} Q : B \to A \mid \Delta$ and $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} R : B \mid \Delta$. By induction, there are S_1, S_2, $\langle \Gamma'_1; P_1; \Delta'_1 \rangle = pt_{\mathsf{F}}\, Q$ and $\langle \Gamma'_2; P_2; \Delta'_2 \rangle = pt_{\mathsf{F}}\, R$ (no type variables shared) such that $S_1\,\Gamma'_1 \subseteq \Gamma$, $S_1\,\Delta'_1 \subseteq \Delta$, $S_2\,\Gamma'_2 \subseteq \Gamma$, $S_2\,\Delta'_2 \subseteq \Delta$, $S_1\,P_1 = B \to A$ and $S_2\,P_2 = B$. Notice that S_1, S_2 do not interfere. Let φ be a fresh type variable.

Now either:

$(P_1 \neq \mathsf{fail})$: Let
$$\begin{array}{rcl} S_u &=& \mathit{unify}\ P_1\ (P_2 \to \varphi) \\ S_\Gamma &=& \mathit{unifyC}\ (S_u\,\Gamma'_1)\ (S_u\,\Gamma'_2) \\ S_\Delta &=& \mathit{unifyC}\ (S_\Gamma \circ S_u\,\Delta'_1)\ (S_\Gamma \circ S_u\,\Delta'_2) \\ pt_{\mathsf{F}}\, QR &=& S_\Delta \circ S_\Gamma \circ S_u\, \langle \Gamma'_1 \cup \Gamma'_2; \varphi; \Delta'_1 \cup \Delta'_2 \rangle \end{array}$$

We need to argue that $pt_{\mathsf{F}}\, QR$ is successful: since this can only fail on unification (of P_1 and $P_2 \to \varphi$, or in the unification of the contexts), we need to argue that these are successful. Take $S_3 = S_2 \circ S_1 \circ (\varphi \mapsto A)$, then
$$\begin{array}{rcl} S_3\,P_1 &=& B \to A, \text{ and} \\ S_3(P_2 \to \varphi) &=& B \to A. \end{array}$$
so P_1 and $P_2 \to \varphi$ have a common instance $B \to A$, and by Lem. 1.4, S_u exists.

Notice that we have
$$\begin{array}{rclcrcl} S_3\,\Gamma'_1 &\subseteq& \Gamma, \text{ and} && S_3\,\Delta'_1 &\subseteq& \Delta, \text{ and} \\ S_3\,\Gamma'_2 &\subseteq& \Gamma && S_3\,\Delta'_2 &\subseteq& \Delta \end{array}$$
since Γ'_1 and Γ'_2 share no type-variables. Since Γ is a context, each term variable has only one type, and therefore S_3 is a unifier for Γ'_1 and Γ'_2 (and similar for Δ), so we know that an S_4 exists which extends the substitution that unifies the contexts, even after being changed with S_u, so such that
$$\begin{array}{rclcrcl} S_4(S_u\,\Gamma'_1) &\subseteq& \Gamma, \text{ and} && S_4(S_\Gamma \circ S_u\,\Delta'_1) &\subseteq& \Delta, \text{ and} \\ S_4(S_u\,\Gamma'_2) &\subseteq& \Gamma. && S_4(S_\Gamma \circ S_u\,\Delta'_2) &\subseteq& \Delta. \end{array}$$
So S_4 also unifies $S_u\,\Gamma'_1$ and $S_u\,\Gamma'_2$ and $S_\Gamma \circ S_u\,\Delta'_1$ and $S_\Gamma \circ S_u\,\Delta'_2$, so by Lem. 1.4 there exists a substitution S_5 such that $S_4 = S_5 \circ S_\Delta \circ S_\Gamma \circ S_u$. Take $S = S_5$.

$(P_1 = \mathsf{fail})$: Let
$$\begin{array}{rcl} S_\Gamma &=& \mathit{unifyC}\ \Gamma'_1\ \Gamma'_2 \\ S_\Delta &=& \mathit{unifyC}\ (S_\Gamma\,\Delta'_1)\ (S_\Gamma\,\Delta'_2) \\ pt_{\mathsf{F}}\, QR &=& S_\Delta \circ S_\Gamma\, \langle \Gamma'_1 \cup \Gamma'_2; \mathsf{fail}; \Delta'_1 \cup \Delta'_2 \rangle \end{array}$$

Take $S_3 = S_2 \circ S_1 \circ (\varphi \mapsto A)$, then

$$\begin{array}{rcl@{\qquad}rcl} S_3\,\Gamma'_1 &\subseteq& \Gamma\text{, and} & S_3\,\Delta'_1 &\subseteq& \Delta\text{, and}\\ S_3\,\Gamma'_2 &\subseteq& \Gamma & S_3\,\Delta'_2 &\subseteq& \Delta \end{array}$$

As above an S_4 exists such that

$$\begin{array}{rcl@{\qquad}rcl} S_4\,\Gamma'_1 &\subseteq& \Gamma\text{, and} & S_4(S_\Gamma\,\Delta'_1) &\subseteq& \Delta\text{, and}\\ S_4\,\Gamma'_2 &\subseteq& \Gamma. & S_4(S_\Gamma\,\Delta'_2) &\subseteq& \Delta. \end{array}$$

Again by Lem. 1.4 there exists a substitution S_5 such that $S_4 = S_5 \circ S_\Delta \circ S_\Gamma$. Take $S = S_5$.

$(M \equiv \mathsf{try}\ L; \overrightarrow{\mathsf{handle}_i\ \mathsf{n}_i(x) = N_i};; \mathsf{catch}\ \mathsf{n}_{n+1}(x) = N_{n+1})$: By rule (try) there exist A_i, B_i ($i \in \underline{n+1}$), such that $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} L : A \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B_i}, \mathsf{n}_{n+1}{:}A_{n+1} \to B_{n+1}, \Delta$ and $\Gamma, x{:}A_i \vdash^{\mathsf{N}}_{\mathsf{F}} N_i : B_i \mid \Delta$ where handle_i is either catch and $B_i = A$, or abort and $B_i = \mathsf{fail}$ for every $i \in \underline{n}$, and $B_{n+1} = A$.

Let $M' = \mathsf{try}\ L; \overrightarrow{\mathsf{handle}_i\ \mathsf{n}_i(x) = N_i}$ and notice that $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M' : A \mid \mathsf{n}_{n+1}{:}A_{n+1} \to B_{n+1}, \Delta$ holds as well. Then by induction there exist S_1, S_2, $\langle \Gamma'_1; P_1; \Delta'_1 \rangle = pt_{\mathsf{F}}\, M'$ and $\langle \Gamma'_2; P_2; \Delta'_2 \rangle = pt_{\mathsf{F}}\, N_{n+1}$ (no type variables shared and $P_2 \neq \mathsf{fail}$) such that

$$\begin{array}{rcl} S_1\,\Gamma'_1 &\subseteq& \Gamma\\ S_1\,\Delta'_1 &\subseteq& \mathsf{n}_{n+1}{:}A_{n+1} \to B_{n+1}, \Delta\\ S_1\,P_1 &=& A \end{array} \quad\text{and}\quad \begin{array}{rcl} S_2\,\Gamma'_2 &\subseteq& \Gamma, x{:}A_{n+1}\\ S_2\,\Delta'_2 &\subseteq& \Delta\\ S_2\,P_2 &=& B_{n+1} \end{array}$$

Take $\Gamma_1 = \Gamma'_1$, and $\Delta_2 = \Delta'_2$. Now either $x{:}D \in \Gamma'_2$ and we take $C = D \to P_2$, and $\Gamma_2 = \Gamma'_2 \setminus x{:}D$, or $x \notin \Gamma'_2$ and $C = \varphi \to P_2$, $\Gamma_2 = \Gamma'_2$, with φ fresh. Likewise, either $\mathsf{n}_{n+1}{:}E \in \Delta'_1$ and we take $\Delta_1 = \Delta'_1 \setminus \mathsf{n}$ or $\mathsf{n}_{n+1} \notin \Delta'_1$ and $E = \varphi'$ with φ' fresh, $\Delta_1 = \Delta'_1$. Take $S_u = \mathit{unify}\ E\ C$, and $S_3 = S_2 \circ S_1 \circ S_u$, then we have $S_3\,\Gamma_1 \subseteq \Gamma$, $S_3\,\Delta_1 \subseteq \Delta$, $S_3\,P_1 = A$, $S_3\,\Gamma_2 \subseteq \Gamma$, and $S_3\,\Delta_2 \subseteq \Delta$.

Since S_3 unifies all these, the unifications called when calculating

$$pt_{\mathsf{F}}(\mathsf{try}\ M; \mathsf{Handlers}; \mathsf{catch}\ \mathsf{n}(x) = N) \;=\; S'\,\langle \Gamma_1 \cup \Gamma_2; P_1; \Delta_1 \cup \Delta_2 \rangle$$

are successful, and using Lem. 1.4 S_3 can now be decomposed into $S \circ S'$.

$(M \equiv \mathsf{try}\ L; \overrightarrow{\mathsf{handle}_i\ \mathsf{n}_i(x) = N_i};; \mathsf{abort}\ \mathsf{n}_{n+1}(x) = N_{n+1})$: By rule (try) there exist A_i, B_i ($i \in \underline{n+1}$), such that $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} L : A \mid \overrightarrow{\mathsf{n}_i{:}A_i \to B_i}, \mathsf{n}_{n+1}{:}A_{n+1} \to B_{n+1}, \Delta$ and $\Gamma, x{:}A_i \vdash^{\mathsf{N}}_{\mathsf{F}} N_i : B_i \mid \Delta$ where handle_i is either catch and $B_i = A$, or abort and $B_i = \mathsf{fail}$ for every $i \in \underline{n}$, and $B_{n+1} = \mathsf{fail}$.

Let $M' = \mathsf{try}\ L; \overrightarrow{\mathsf{handle}_i\ \mathsf{n}_i(x) = N_i}$ and notice that we have $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} M' : A \mid \mathsf{n}_{n+1}{:}A_{n+1} \to \mathsf{fail}, \Delta$ as well. Then by induction there exist S_1, S_2, $\langle \Gamma'_1; P_1; \Delta'_1 \rangle = pt_{\mathsf{F}}\, M'$ and $\langle \Gamma'_2; \mathsf{fail}; \Delta'_2 \rangle = pt_{\mathsf{F}}\, N_{n+1}$ (no type variables shared) such that

$$\begin{array}{rcl} S_1\,\Gamma'_1 &\subseteq& \Gamma\\ S_1\,\Delta'_1 &\subseteq& \mathsf{n}_{n+1}{:}A_{n+1} \to \mathsf{fail}, \Delta\\ S_1\,P_1 &=& A \end{array} \quad\text{and}\quad \begin{array}{rcl} S_2\,\Gamma'_2 &\subseteq& \Gamma, x{:}A_{n+1}\\ S_2\,\Delta'_2 &\subseteq& \Delta \end{array}$$

Take $\Gamma_1 = \Gamma'_1$, and $\Delta_2 = \Delta'_2$. Now either $x{:}D \in \Gamma'_2$ and we take $C = D \to \mathsf{fail}$, and $\Gamma_2 = \Gamma'_2 \setminus x{:}D$, or $x \notin \Gamma'_2$ and $C = \varphi \to \mathsf{fail}$, $\Gamma_2 = \Gamma'_2$, with φ fresh. Likewise, either $\mathsf{n}_{n+1}{:}E \in \Delta'_1$ and we take $\Delta_1 = \Delta'_1 \setminus \mathsf{n}$ or $\mathsf{n}_{n+1} \notin \Delta'_1$ and $E = \varphi'$ with φ' fresh, $\Gamma_2 = \Gamma'_2$. Take $S_u = \mathit{unify}\ E\ C$, and $S_3 = S_u \circ S_2 \circ S_1$, then we have $S_3\,\Gamma_1 \subseteq \Gamma$, $S_3\,\Delta_1 \subseteq \Delta$, $S_3\,P_1 = A$, $S_3\,\Gamma_2 \subseteq \Gamma$, and $S_3\,\Delta_2 \subseteq \Delta$. As above, unification of the contexts is successful, and S_3 can now be decomposed into a unifying substitution, and the substitution we are looking for.

$(M \equiv \mathsf{throw}\,\mathsf{n}(N))$: By rule (throw) there exist C, D such that $\mathsf{n}{:}C \to D \in \Delta$ and $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} N : C \mid \Delta \setminus \mathsf{n}$. By induction there exists S', $\langle \Gamma'; P; \Delta' \rangle = pt_{\mathsf{F}}\, N$, such that $S'\,\Gamma' \subseteq \Gamma$, $S'\,\Delta' \subseteq \Delta \setminus \mathsf{n}$, and $S'\,P = C$. Notice that $pt_{\mathsf{F}}(\mathsf{throw}\,\mathsf{n}(N)) = \langle \Gamma; \varphi; \mathsf{n}{:}P \to \varphi', \Delta \rangle$, where φ, φ' fresh; define $S = (\varphi \mapsto A) \circ (\varphi' \mapsto D) \circ S'$, then $S\Gamma' \subseteq \Gamma$, $S(\mathsf{n}{:}P \to \varphi', \Delta') \subseteq \Delta$, and $S\,\varphi = A$.

$(M \equiv \mathsf{panic}\,\mathsf{n}(N))$: By rule (panic), $A = \mathsf{fail}$ and there exists C such that $\mathsf{n}{:}C \to \mathsf{fail} \in \Delta$ and $\Gamma \vdash^{\mathsf{N}}_{\mathsf{F}} N : C \mid \Delta \setminus \mathsf{n}$. By induction there exists S', $\langle \Gamma'; P; \Delta' \rangle = pt_{\mathsf{F}}\, N$, such that $S'\,\Gamma' \subseteq \Gamma$, $S'\,\Delta' \subseteq \Delta \setminus \mathsf{n}$, and $S'\,P = C$. Notice that $pt_{\mathsf{F}}(\mathsf{throw}\,\mathsf{n}(N)) = \langle \Gamma; \mathsf{fail}; \mathsf{n}{:}P \to \mathsf{fail}, \Delta \rangle$; define $S = S'$, then $S\Gamma' \subseteq \Gamma$, and $S(\mathsf{n}{:}P \to \mathsf{fail}, \Delta') \subseteq \Delta$.

($M \equiv \mathsf{if}\ Q\ \mathsf{then}\ R\ \mathsf{else}\ S$): We distinguish the cases:

(cond): Then there are B,C such that $\Gamma \vdash^{\mathrm{N}}_{\mathsf{F}} Q : \mathsf{bool} \mid \Delta$, $\Gamma \vdash^{\mathrm{N}}_{\mathsf{F}} R : B \mid \Delta$, and $\Gamma \vdash^{\mathrm{N}}_{\mathsf{F}} S : C \mid \Delta$, and $A = B = C$, or $A = B$ and $C = \mathsf{fail}$, or $A = C$ and $B = \mathsf{fail}$. By induction there are S_1, S_2, S_3 and $\langle \Gamma_1; P_1; \Delta_1 \rangle = pt_{\mathsf{F}}\, Q$, $\langle \Gamma_2; P_2; \Delta_2 \rangle = pt_{\mathsf{F}}\, R$, and $\langle \Gamma_3; P_3; \Delta_3 \rangle = pt_{\mathsf{F}}\, S$, such that:

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
S_1\,\Gamma_1 & \subseteq & \Gamma & S_2\,\Gamma_2 & \subseteq & \Gamma & S_3\,\Gamma_3 & \subseteq & \Gamma \\
S_1\,\Delta_1 & \subseteq & \Delta & S_2\,\Delta_2 & \subseteq & \Delta & S_3\,\Delta_3 & \subseteq & \Delta \\
S_1\,P_1 & = & \mathsf{bool} & S_2\,P_2 & = & B & S_3\,P_3 & = & C
\end{array}$$

Now take φ fresh, and

$$\begin{array}{lcl}
S_4 & = & \mathit{unify}\ P_1\ \mathsf{bool} \\
S_5 & = & \begin{cases} (\mathit{unify}\ (S_4\,P_2)\ (S_4\,P_3)) \circ (\varphi \mapsto S_4\,P_2) & (P_2 \neq \mathsf{fail} \neq P_3) \\ (\varphi \mapsto S_4\,P_2) & (P_2 \neq \mathsf{fail} = P_3) \\ (\varphi \mapsto S_4\,P_3) & (P_2 = \mathsf{fail} \neq P_3) \\ (\varphi \mapsto \mathsf{fail}) & (P_2 = \mathsf{fail} = P_3) \end{cases} \\
S_\Gamma & = & \mathit{unifyC}\ (S_5 \circ S_4\,\Gamma_1)\ (S_5 \circ S_4\,\Gamma_2)\ (S_5 \circ S_4\,\Gamma_3) \\
S_\Delta & = & \mathit{unifyC}\ (S_\Gamma \circ S_5 \circ S_4\,\Delta_1)\ (S_\Gamma \circ S_5 \circ S_4\,\Delta_2)\ (S_\Gamma \circ S_5 \circ S_4\,\Delta_3)
\end{array}$$

As above, since S_1, S_2 and S_3 create common instances, the unifications in S_4, S_5, S_Γ, and S_Δ are successful, and

$$pt_{\mathsf{F}}\,(\mathsf{if}\ Q\ \mathsf{then}\ R\ \mathsf{else}\ S) \ = \ S_\Delta \circ S_\Gamma \circ S_5 \circ S_4\, \langle \Gamma_1 \cup \Gamma_2 \cup \Gamma_3; \varphi; \Delta_1 \cup \Delta_2 \cup \Delta_3 \rangle$$

is well defined, and by Lem. 1.4 there exists S such that $S_3 \circ S_2 \circ S_1 = S \circ S_\Delta \circ S_\Gamma \circ S_5 \circ S_4$.

(cond$_{\mathsf{F}}$): Similar to the previous case, where now $A = \mathsf{fail} = P_1$. □