A TYPE SYSTEM FOR CALL-BY-NAME EXCEPTIONS

SYLVAIN LEBRESNE

Purdue University
e-mail address: slebresn@purdue.edu

Abstract. We present an extension of System F with call-by-name exceptions. The type system is enriched with two syntactic constructs: a union type for programs whose execution may raise an exception at top level, and a *corruption type* for programs that may raise an exception in any evaluation context (not necessarily at top level). We present the syntax and reduction rules of the system, as well as its typing and subtyping rules. We then study its properties, such as confluence. Finally, we construct a realizability model using orthogonality techniques, from which we deduce that well-typed programs are weakly normalizing and that the ones who have the type of natural numbers really compute a natural number, without raising exceptions.

1. Introduction

Exceptions are a convenient mechanism for handling errors in programming languages. Most modern languages use them: Java, ML, C++, ... The main computational features of exceptions are:

(1) One can raise an exception instead of any other expression (or instruction);
(2) It propagates automatically by default;
(3) One can catch it only when they need to.

Exceptions have long been confined to call-by-value languages and are usually presented as a mechanism which "cuts through" the normal control flow of a program when raised. This is viewing the raising of an exception as an effect of the calculus. Unfortunately, this view makes exceptions hard to transpose to call-by-name calculi since those do not cope well with effects.

This is a well-known problem. While in call-by-value the effect of a term $t\ u$ can be simply predicted based solely on the effects of t and u, in call-by-name it also depends on the actual term t. Indeed, in call-by-name u may well not be evaluated thus not producing its effect (or evaluated many times, producing the effect many times). For exceptions, this means that in call-by-name, the fact that u raises an exception does not necessarily imply that $t\ u$ will. Hence in call-by-name, as summarized by S. Peyton Jones *et al.* [14], "(...) the only productive way to think about an expression is to consider the *value it computes*, not the *way in which the value is computed*". Based on this observation, they proposed the

idea of exceptions-as-values: a value is either a "normal" value, or an "exceptional" one. In their framework, exceptions are not effects anymore. And while they present this idea in the context of the Haskell programming language, this is a very general idea for exceptions in call-by-name calculi.

From a typing perspective, exceptions are no simple beasts. Indeed, the type system should allow the use of exceptions in any part of a program. What should then be the type of the operation for raising an exception[1]? A solution, used in ML for instance, is to allow the operation of raising an exception to have any type. In [14], S. Peyton Jones *et al.* chose a similar solution, making exceptional values inhabitants of all types. While simple, this solution comes at a price, the loss of type safety with respect to exceptions. The type of an expression never ensures that no exceptions can be raised during evaluation.

If we want the type of an expression to reflect which exception it may raise, a more precise typing is in order. For call-by-value languages, since exceptions are effects, a convenient and efficient solution is to add to the type system an effect system [13, 7]. Unfortunately and unsurprisingly, this solution is unadapted to the typing of call-by-name exceptional values. Indeed, tracking values with types is much more difficult than tracking effects.

The call-by-name evaluation is well represented amongst type theoretical calculi which are at the core of many proof assistants (COQ [4], LEGO [15], . . .). We believe this reinforce the case for studying exceptions in call-by-name and their precise typing. Indeed, the solution of having exceptional values inhabiting all types would be inconsistent in these settings.

This paper proposes a type system for exceptions in call-by-name calculi. By introducing the new notion of *corruption*, this type system is able to track which exceptions may escape from a term during evaluation. By using subtyping, this notion is able to cope with the automatic propagation of exceptions and to respect the modularity of typing. This type system is presented in the context of an extension of System F with exceptions. System F is used here as a first step towards more elaborate type theoretical frameworks.

Meta-theoretical properties of the resulting calculus are proved in particular by exhibiting a realizability model. While parts of the proofs are given in this paper, more detailed proofs of the results presented[2] are available in the Ph.D thesis of the author [8].

The remaining of the paper is organized as follows. We explain our design in Section 2: we justify the kind of exception-as-values we use and describe the three levels of corruption our type system distinguishes. We formally present our calculus in Section 3 and state the properties it enjoys and Section 4 provides some examples. Then in Section 5 we design a realizability model of our calculus that gives some insight on the meaning of corruption and we prove its soundness. Finally, we present in Section 6 some related works before concluding in Section 7 with future works.

2. Design of the system

2.1. **Which exceptions-as-values?** As stressed above, exceptions in call-by-name calculi should be values. But there are essentially two designs for exceptions as values: either we

---

[1]Remark that imperative language alleviate this problem by making the operation of raising an exception an instruction and not an expression.

[2]A notable difference is the presence of lists in the language described in this document, while in [8] the data type of lists is only presented for a first-order typed version of the language.

encode them explicitly in the language, or we make them primitives. The first option is a well-known one and let us first present its drawbacks in order to justify the need for the primitive solution.

Encoding explicitly exceptions is an old idea [21, 16]: to each type A is associated a type Maybe A which is either values of A tagged as correct values or exceptional values (this idea is nicely explained, for the Haskell programming language, in [14]). It has later been realized that the Maybe type constructor forms a *monad* [11, 22]. And P. Wadler and P. Thiemann proposed in [23] to add effects to monads, allowing for the detection of uncaught exceptions in such monadic encoding. However this approach has some drawbacks, namely:

- Terms using exceptions are crippled by extra clutter. For example, in Haskell, to apply a function f :: Int -> Int to a value x :: Maybe Int we are forced to write:

    ```
    do a <- x
       return (f a)
    ```

  Using exceptions is not as transparent for the programmer as it is in call-by-value languages;
- As remarked in [14], modularity and code re-use are compromised, especially for higher order functions. Consider the following sorting function:

    ```
    sort :: (a -> a -> Bool) -> [a] -> [a]
    ```

  This function cannot be applied to a comparison function that may raise exceptions such as:

    ```
    cmp :: a -> a -> Maybe Bool
    ```

  Indeed, with monads, we need to know where the sort function uses the comparison in order to add the monad's operations;
- Monads force the evaluation of arguments (in the example above, the evaluation of x is forced before the application to f). One could not see that as an inconvenience, and this is indeed desirable for most uses of monads (IO, states, . . .). Nonetheless, this is a constraint and it makes exceptions not usable in non monadic call-by-name code. We think that this can be avoided for exceptions.

This leads us to the second design choice: making exceptions primitives. This has been first proposed by S. Peyton Jones *et al.* [14] with *imprecise exceptions*. The idea is that a value of *any* type is either a "normal" value, or an "exceptional" one. The resulting mechanism allows exceptions to be used in place of any other term (as for more traditional "call-by-value" exceptions and contrarily to monadic ones). Note that since values may be exceptional, we can have for instance, a list, which is fully defined but for which some elements are exceptional values (see Section 4). These exceptions are raised only when (and if) the list is evaluated. A main difference with the call-by-value mechanism of exceptions is for example that a term like $(\lambda x. 0)$ (raise ε) (where 0 is simply the constant zero and ε some exception) will reduce to 0 and not to raise ε.

Our system, named *Fx*, adapts this idea to System F, adding it two new term constructions: raise and try. But while the exceptions of [14] are not precisely typed (the raising operation is in all types), we propose a type system where the type of an expression indicates which exceptions the expression may raise.

2.2. **Expected properties.** The type system we will present enjoys the following properties:

- If a term can raise an exception, its type indicates it. In particular, programs of type $\mathbb{N}$ are not able to raise exceptions;
- Programmers can use a term $\mathtt{raise}\,\varepsilon$ in place of any other term. In particular, $\mathtt{raise}\,\varepsilon$ type as a function;
- Exceptions and their typing discipline do not jeopardize modularity and code re-use. A function defined without exceptions in mind still accepts exceptional arguments and behave in a sensible way. Moreover, this is done without knowing the actual code of the function.

2.3. **Three levels of corruption.** We call *corrupted*, a term that may mention exceptions. Given a type A (say the type $\mathbb{N}$ of natural numbers), we distinguish three levels of *corruptions* for the terms related with this type:

- Terms of A. They are not corrupted, either they do not mention exceptions or the ones they mention are caught or erased during reduction;
- Terms of $A \uplus \{\varepsilon\}$. They are terms of A or terms that reduce to the exception ε, i.e. reduce to $\mathtt{raise}\,\varepsilon$ (we then say that they raise ε).
- Terms of $A^{\{\varepsilon\}}$. They are terms of A that may mention the exception ε but do not necessarily reduce to it (for instance, if S is the successor function, $S\,(\mathtt{raise}\,\varepsilon)$ has type $\mathbb{N}^{\{\varepsilon\}}$, but not type $\mathbb{N} \uplus \{\varepsilon\}$ since it has not type $\mathbb{N}$ nor does it reduce to $\mathtt{raise}\,\varepsilon$).

Moreover, to handle the properties of corruption, we use a subtyping relation. And in particular we have the subtyping: $A \leq A \uplus \{\varepsilon\} \leq A^{\{\varepsilon\}}$.

The following section explains why the need to distinguish at least those three levels. But one might wonder why we do not distinguish more levels. Like the terms containing exceptions but *not* at top level. Or terms having an exception at a depth of *at most* 2 (like $\mathtt{raise}\,\varepsilon$ or $\lambda x.\,\mathtt{raise}\,\varepsilon$ but not $\lambda x.\,\lambda y.\,\mathtt{raise}\,\varepsilon$), etc. As of now, while such more precise notion may well be sound, we have not study them. The main reason is that they would complicate and clutter the type system while we are not convinced they would prove useful in practice.

2.4. **Why we need to distinguish these three levels.** The construction $A \uplus \{\varepsilon\}$ is really needed because of the typing of the $\mathtt{try}$ operation, since for a $\mathtt{try}$ to catch an exception in its body, this body has to reduce to the exception.

But because we do not want to change the typing rule of application, the construction $A \uplus \{\varepsilon\}$ clearly does not fulfill all our needs. Firstly, we cannot use it to type $S\,(\mathtt{raise}\,\varepsilon)$. Secondly, given a function M of type $A \to B$, we cannot apply it to a term N of type $A \uplus \{\varepsilon\}$. Indeed, $M\,N$ is generally not of type $B \uplus \{\varepsilon\}$ (note however that it would be the case in a call-by-value calculus). Consider for instance $M = \lambda x.\,\lambda y.\,x$ (of type $A \to (C \to A)$) and $N = \mathtt{raise}\,\varepsilon$, then $M\,N$ reduces to $\lambda y.\,\mathtt{raise}\,\varepsilon$ which is not of type $(C \to A) \uplus \{\varepsilon\}$ (since it is neither a function of type $C \to A$ nor the exception ε)[3].

---

[3]Note that this is typically this example, of the typing of a term like $(\lambda x.\,\lambda y.\,x)\,N$ when N may raises an exception, that makes effect system [13] unsuited to call-by-name exceptions.

To solve these problems, we use a second type construction, the *corruption* of a type A by an exception of name ε, denoted $A^{\{\varepsilon\}}$. The main property the corruption enjoys is a good behavior with respect to arrow types:

$$(A \to B)^{\{\varepsilon\}} \quad \doteq \quad A^{\{\varepsilon\}} \to B^{\{\varepsilon\}}$$

This subtyping equality[4] may seem paradoxical with the usual subtyping rule of arrow (contra-variance to the left, co-variance to the right). This is however justified by the realizability model of Section 5.

Intuitively, terms of type $A^{\{\varepsilon\}}$ should be seen as terms of type A where some sub-terms may have been replaced by $\mathtt{raise}\,\varepsilon$ (hence, programmers can use $\mathtt{raise}\,\varepsilon$ wherever they want, which, in turns, corrupts the resulting type). Equivalently, while terms of $A \uplus \{\varepsilon\}$ are terms that may reduce to $\mathtt{raise}\,\varepsilon$ at top-level, terms of $A^{\{\varepsilon\}}$ are the ones that may reduce to $\mathtt{raise}\,\varepsilon$ in *any* evaluation context.

Now, with corruption, we can apply a function $f : A \to B$ to a potentially exceptional term. Indeed, we have that

$$A \to B \;\leq\; (A \to B) \uplus \{\varepsilon\} \;\leq\; (A \to B)^{\{\varepsilon\}} \;\doteq\; A^{\{\varepsilon\}} \to B^{\{\varepsilon\}}.$$

Remark that since we use subtyping, there is no need to actually know the term f. This allows for modularity: to type the application of some (external) function f to a term u, it is enough to know the type of f, and this even when u may raise exceptions but the exported type of f does not mention exceptions. This is in particular convenient for primitive functions like the successor function S, allowing to type-check $S\,(\mathtt{raise}\,\varepsilon)$ with the type $\mathbb{N}^{\{\varepsilon\}}$ without the need to give S a complicated type (the type of S is simply $\mathbb{N} \to \mathbb{N}$).

2.5. **Exceptions by the millions.** While we have only used one exception names ε in the above section, it is useful to be able to handle more than one exception at a time. To that end, the general type constructions are $A \uplus \Delta$ and A^{Δ} where Δ is a set of exception names.

Using sets of exceptions requires some type identification using the following subtyping rules:

$$
\begin{aligned}
(A \uplus \Delta') \uplus \Delta &\;\doteq\; A \uplus (\Delta \cup \Delta') \\
(A^{\Delta'})^{\Delta} &\;\doteq\; A^{(\Delta \cup \Delta')} \\
A \uplus \emptyset &\;\doteq\; A \\
A^{\emptyset} &\;\doteq\; A
\end{aligned}
$$

---

[4]The subtyping equality $A \doteq B$ is simply defined as shorthand for $A \leq B$ and $B \leq A$.

3. Formal presentation

We present the *Fx* calculus, an extension of System F with typed exceptions, natural numbers and lists.

3.1. **Syntax, reductions and associated properties.**

3.1.1. *Syntax of terms.* We consider a countable set $\mathcal{E}$ of names of exceptions and a distinguished set of variables $\mathcal{V}$.

Definition 3.1 (Terms). A *term* of *Fx* is a term generated by the following grammar:

$$
\begin{aligned}
M, N \quad ::= \quad & x \quad | \quad \lambda x.\, M \quad | \quad M\ N \\
& | \quad \mathtt{raise}\,\varepsilon \quad | \quad \mathtt{try}\, M \, \mathtt{with}\, \varepsilon \mapsto N \\
& | \quad 0 \quad | \quad S \quad | \quad \mathtt{rec} \quad | \quad [\,] \quad | \quad \mathtt{cons} \quad | \quad \mathtt{list\_rec}
\end{aligned}
$$

In this definition, variables are ranged over by $x, y, \ldots$ while exception names are ranged over by $\varepsilon, \varepsilon', \ldots$. Notions of free and bound variables are defined as usual, as well as the external operation of substitution (written $M\{x := N\}$). The set of all closed terms is denoted $\mathcal{T}$ and terms are considered up to α-equivalence. Note that the construction $\mathtt{try}\, M \, \mathtt{with}\, \varepsilon \mapsto N$ does not bind the occurrences of ε. The term $\mathtt{raise}\,\varepsilon$ is called an *exception*, ε being its name, but, as an abuse of terminology, we also call ε an exception. In the term $\mathtt{try}\, M \, \mathtt{with}\, \varepsilon \mapsto N$ we will sometimes call M the body and N the handler of the $\mathtt{try}$ construction.

To the terms of the lambda calculus, we add the constructions to raise and catch exceptions as well as two usual structured data types: the natural numbers and the lists.

3.1.2. *Computation in Fx.*

Definition 3.2 (Regular values). A *regular value* is a (closed) term of *Fx* having one of the following form:

$$
\begin{aligned}
RV \quad ::= \quad & \lambda x.\, M \mid 0 \mid S \mid S\ N \mid \mathtt{rec} \mid \mathtt{rec}\ M \mid \mathtt{rec}\ M\ N \\
& | \; [\,] \mid \mathtt{cons} \mid \mathtt{cons}\ M \mid \mathtt{cons}\ M\ N \mid \mathtt{list\_rec} \mid \mathtt{list\_rec}\ M \mid \mathtt{list\_rec}\ M\ N
\end{aligned}
$$

Note that $S\ N$ is a regular value for any term N and hence $S\,(\mathtt{raise}\,\varepsilon)$ is a regular value as well.

Definition 3.3 (Values). A *values* is a (closed) term of *Fx* having one of the following form:

$$
V ::= RV \mid \mathtt{raise}\,\varepsilon
$$

where RV is a regular value and ε any exception name.

For well-typed term, a value corresponds to a weak head normal form.

Definition 3.4 (Computation). The *notion of reduction* $>$ for the calculus is defined by the rules of Figure 1. Computation in *Fx* is defined from the notion of reduction by the relation of *reduction* $\succ$ whose rules are given in Figure 2. We note $\succ^*$ the reflexive and transitive closure of $\succ$ and we note $=$ its reflexive, transitive and symmetric closure. Moreover, if $M = N$, we will say that M is *equivalent* to N.

$$
\begin{array}{rcll}
(\lambda x.\, M)\ N & > & M\{x := N\} \\
(\mathtt{raise}\,\varepsilon)\ M & > & \mathtt{raise}\,\varepsilon \\[4pt]
\mathtt{try}\,(\mathtt{raise}\,\varepsilon)\,\mathtt{with}\,\varepsilon \mapsto N & > & N \\
\mathtt{try}\,(\mathtt{raise}\ \varepsilon')\,\mathtt{with}\,\varepsilon \mapsto N & > & \mathtt{raise}\ \varepsilon' & (\text{if } \varepsilon \neq \varepsilon') \\
\mathtt{try}\,V\,\mathtt{with}\,\varepsilon \mapsto N & > & V & (\text{if } V \text{ is a regular value}) \\[4pt]
\mathtt{rec}\ X\ Y\ 0 & > & X \\
\mathtt{rec}\ X\ Y\ (S\ N) & > & Y\ N\ (\mathtt{rec}\ X\ Y\ N) \\
\mathtt{rec}\ X\ Y\ (\mathtt{raise}\,\varepsilon) & > & \mathtt{raise}\,\varepsilon \\[4pt]
\mathtt{list\_rec}\ X\ Y\ [\,] & > & X \\
\mathtt{list\_rec}\ X\ Y\ (\mathtt{cons}\ E\ L) & > & Y\ E\ L\ (\mathtt{list\_rec}\ X\ Y\ L) \\
\mathtt{list\_rec}\ X\ Y\ (\mathtt{raise}\,\varepsilon) & > & \mathtt{raise}\,\varepsilon
\end{array}
$$

Figure 1: Notion of reduction for *Fx*

$$
\frac{M > M'}{M \succ M'} \qquad \frac{M \succ M'}{M\ N \succ M'\ N} \qquad \frac{M \succ M'}{\mathtt{try}\,M\,\mathtt{with}\,\varepsilon \mapsto N \succ \mathtt{try}\,M'\,\mathtt{with}\,\varepsilon \mapsto N}
$$

$$
\frac{M \succ M'}{\lambda x.\, M \succ \lambda x.\, M'} \qquad \frac{N \succ N'}{M\ N \succ M\ N'} \qquad \frac{N \succ N'}{\mathtt{try}\,M\,\mathtt{with}\,\varepsilon \mapsto N \succ \mathtt{try}\,M\,\mathtt{with}\,\varepsilon \mapsto N'}
$$

Figure 2: Relation of reduction for *Fx*

Note that, as usual, the scope of capture of the $\mathtt{try}$ construction is dynamic: in the term $(\lambda x.\, \mathtt{try}\,x\,\mathtt{with}\,\varepsilon \mapsto 0)\ (\mathtt{raise}\,\varepsilon)$, the exception is caught during reduction and the whole term reduces to 0. We say that a term M *raises the exception* ε if $M \succ^* \mathtt{raise}\,\varepsilon$ (that is, if M reduces to the exception named ε).

Definition 3.5 (to have a value). We will say that a term M *has a value* if and only if it reduces to a value, that is if there exists a value V such that $M \succ^* V$.

It can be proved [8] that this notion is equivalent to the one of having a normal form for the weak head reduction of the calculus.

We now show that adding $\mathtt{raise}$ and $\mathtt{try}$ does not break the confluence of the calculus:

Theorem 3.6 (Confluence). *If M, N and N' are terms such that $M \succ^* N$ and $M \succ^* N'$, then there exists a term P such that $N \succ^* P$ and $N' \succ^* P$.*

Proof. We adapt the proof originated by Tait and Martin-Löf for the confluence of pure lambda-calculus that can be found in [1] for example. We define the notion of parallel reduction $\gg$ for *Fx*, we show that it satisfies the diamond property and conclude since $\succ^* = \gg^*$. Proofs of these properties are easy to tackle inductions we leave to the interested reader. We however give in appendix A the definition of the parallel reduction for *Fx*. $\square$

3.2. **The type system.** As stressed in Section 2.3, Fx uses a subtyping relation $\leq$. Thus, Fx is in fact an extension of the second-order lambda calculus with subtyping introduced by Mitchell [10, 24] (and we will call this calculus System $F\eta$ in the following). Note that we will however use a presentation of this calculus that differs from the original one and that can be found for example in [19].

Definition 3.7 (Types). The syntax of types for Fx is built upon the one of System F. *Type* of Fx are generated by the following grammar:

$$A, B ::= \alpha \mid \mathbb{N} \mid A \, \mathtt{list} \mid A \to B \mid \forall \alpha.\, A \mid A \uplus \Delta \mid A^\Delta$$

In $A \uplus \Delta$ and A^Δ, Δ is a finite set of exceptions names ($\Delta \subseteq \mathcal{E}$). Moreover, α stands for a type variable taken from the set of type variables $\mathcal{A}$. Notions of free and bound type variable are defined as usual, as well as the external operation of substitution (written $A\{\alpha := B\}$). We denote by $FV(A)$ the set of all the free type variables of the type A. Types are considered up to α-equivalence. Precedences for the arrow construction and the universal quantifier are the usual ones; the precedences of $A \uplus \Delta$ and A^Δ being higher. Moreover, we will often write $A \, \mathtt{list}^\Delta$ for $(A \, \mathtt{list})^\Delta$.

3.2.1. *Typing.*

Definition 3.8 (Typing context). A *typing context* Γ is a finite set of declarations having the form $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$ where $x_1, \ldots, x_n$ are pairwise distinct term variables and where $A_1, \ldots, A_n$ are arbitrary types.

The set $FV(\Gamma)$ of free variables of Γ denotes the union of the sets of free type variables for the types used in Γ, that is to say:

$$FV(x_1 : A_1, \ldots, x_n : A_n) \;=\; \bigcup_{i \in \{1 \ldots n\}} FV(A_i)$$

Definition 3.9 (Typing). The type system of Fx is defined from the *typing judgment*

$$\Gamma \vdash M : A$$

that reads 'in the typing context Γ, the term M has type A'. This judgment is inductively defined by the rules of Figure 3.

Remark that the typing rules from System $F\eta$ are unchanged, we simply add rules. Also note that the usual typing rules for the recursion operators can be retrieved from *(rec)* and *(fold)* by taking $\Delta = \emptyset$ (theses rules are in fact typing schemes).

3.2.2. *Subtyping.*

Definition 3.10 (Subtyping). The *subtyping* relation between two types A and B, written $A \leq B$, is inductively defined by the rules of Figure 4.

The equality $A \doteq B$ is defined as short for "$A \leq B$ and $A \geq B$". In the inference rules, when the equality $A \doteq B$ appears as a premise, it figures for the two premises $A \leq B$ and $A \geq B$. And when it appears as a conclusion, it figures for two inference rules, one having $A \leq B$ as a conclusion, the other one having $A \geq B$. The subtyping rules from $F\eta$ are unchanged. The rules *(ex-noexc)*, *(eq-uu)* and *(eq-cc)* dealt with sets of exceptions. The hierarchy of corruption (see 2.3) is implemented by

---

System Fη typing rules:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \ (ax) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B} \ (abs) \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B} \ (app)$$

$$\frac{\Gamma \vdash M : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha.\, A} \ (gen) \qquad \frac{\Gamma \vdash M : A \quad A \leq B}{\Gamma \vdash M : B} \ (subs)$$

Natural numbers typing rules:

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \ (zero) \qquad \frac{}{\Gamma \vdash S : \mathbb{N} \to \mathbb{N}} \ (succ)$$

$$\frac{}{\Gamma \vdash \mathtt{rec} : \forall \alpha.\, \alpha \uplus \Delta \to (\mathbb{N}^\Delta \to \alpha \uplus \Delta \to \alpha \uplus \Delta) \to \mathbb{N}^\Delta \uplus \Delta' \to \alpha \uplus (\Delta \cup \Delta')} \ (rec)$$

List typing rules:

$$\frac{}{\Gamma \vdash [\,] : \forall \alpha.\, \alpha \ \mathtt{list}} \ (nil) \qquad \frac{}{\Gamma \vdash \mathtt{cons} : \forall \alpha.\, \alpha \to \alpha \ \mathtt{list} \to \alpha \ \mathtt{list}} \ (cons)$$

$$\frac{}{\begin{array}{c} \Gamma \vdash \mathtt{list\_rec} : \forall \alpha.\, \forall \beta.\, \alpha \uplus \Delta \to (\beta^\Delta \to \beta \ \mathtt{list}^\Delta \to \alpha \uplus \Delta \to \alpha \uplus \Delta) \\ \to \beta \ \mathtt{list}^\Delta \uplus \Delta' \to \alpha \uplus (\Delta \cup \Delta') \end{array}} \ (fold)$$

Exceptions handling typing rules:

$$\frac{}{\Gamma \vdash \mathtt{raise}\, \varepsilon : \forall \alpha.\, \alpha \uplus \{\varepsilon\}} \ (raise) \qquad \frac{\Gamma \vdash M : A \uplus \{\varepsilon\} \quad \Gamma \vdash N : A}{\Gamma \vdash \mathtt{try}\, M \, \mathtt{with}\, \varepsilon \mapsto N : A} \ (try)$$

Figure 3: Typing judgments

(ex-uni) and *(ex-corrupt)*. The rules *(ex-fallc)* and *(ex-fallu)* are justified by the absence of computational content of the universal quantification. Moreover, corruption and union commutes *(eq-uc)*.

The subtyping is stable by union *(ex-ctx)*, but also by corruption (this is proved by Theorem 3.11). Rule *(ex-arru)* simply says that, since a term M of type $(A \to B) \uplus \Delta$ is either a term of type $A \to B$ or an exception of Δ, it can always be applied to a term of type A, resulting in a term of type B (if M is a true function) or an exception of Δ (if so is M).

As discussed in Section 2.4, the rule *(eq-arrc)* is the main rule of corruption and allows exceptions to be used anywhere. Note that we really need an equality here on pain of losing the subject-reduction property.

Finally, the list construction is monotonic (rule *(ex-lcor)*) and a list of corrupted elements is in particular a corrupted list (rule *ex-lctx*).

The subtyping associated to the notion of corruption is a quite flexible one, especially with respect to arrows. As noted in Section 2.4, it allows to derive

$$A_1 \to A_2 \to \ldots \to A_n \leq A_1^\Delta \to A_2^\Delta \to \ldots \to A_n^\Delta$$

System Fη rules :

$$\frac{}{A \leq A} \; \text{(st-id)} \qquad \frac{A \leq B \quad B \leq C}{A \leq C} \; \text{(st-trans)} \qquad \frac{A' \leq A \quad B \leq B'}{A \to B \leq A' \to B'} \; \text{(st-arrow)}$$

$$\frac{A \leq B \quad \alpha \notin FV(A)}{A \leq \forall \alpha. \, B} \; \text{(f-gen)} \qquad \frac{}{\forall \alpha. \, A \leq A\{\alpha := B\}} \; \text{(f-inst)}$$

$$\frac{\alpha \notin FV(A)}{\forall \alpha. \, (A \to B) \leq A \to \forall \alpha. \, B} \; \text{(f-distr)}$$

Exception related rules :

$$\frac{}{A \leq A \uplus \Delta} \; \text{(ex-uni)} \qquad\qquad \frac{}{A \uplus \Delta \leq A^\Delta} \; \text{(ex-corrupt)}$$

$$\frac{}{A^\emptyset \leq A} \; \text{(ex-noexc)}$$

$$\frac{A \leq B}{A \uplus \Delta \leq B \uplus \Delta} \; \text{(ex-ctx)} \qquad\qquad \frac{}{(A \to B) \uplus \Delta \leq A \to B \uplus \Delta} \; \text{(ex-arru)}$$

$$\frac{}{\forall \alpha. \, A^\Delta \leq (\forall \alpha. \, A)^\Delta} \; \text{(ex-fallc)} \qquad\qquad \frac{}{\forall \alpha. \, (A \uplus \Delta) \leq (\forall \alpha. \, A) \uplus \Delta} \; \text{(ex-fallu)}$$

$$\frac{}{A^\Delta \; \texttt{list} \leq A \; \texttt{list}^\Delta} \; \text{(ex-lcor)} \qquad\qquad \frac{A \leq B}{A \; \texttt{list} \leq B \; \texttt{list}} \; \text{(ex-lctx)}$$

Exception related equality rules :

$$\frac{}{(A \uplus \Delta) \uplus \Delta' \; \doteq \; A \uplus (\Delta \cup \Delta')} \; \text{(eq-uu)} \qquad\qquad \frac{}{(A^\Delta)^{\Delta'} \; \doteq \; A^{(\Delta \cup \Delta')}} \; \text{(eq-cc)}$$

$$\frac{}{(A \uplus \Delta)^{\Delta'} \; \doteq \; A^{\Delta'} \uplus \Delta} \; \text{(eq-uc)} \qquad\qquad \frac{}{(A \to B)^\Delta \; \doteq \; A^\Delta \to B^\Delta} \; \text{(eq-arrc)}$$

Figure 4: The subtyping relation

but also that $A^\Delta \to B \leq A^\Delta \to B^\Delta$ or that $A \to B^\Delta \leq A^\Delta \to B^\Delta$ for instance[5]. However, what the subtyping of corruption forbids is the removable of corruption in covariant position. That is, corruption allows the use of functions with exceptions they do not handle themselves, but it then always ensure that the return type mentions those exceptions.

3.2.3. *Typing the recursion operations. Fx* uses natural numbers and lists. To work with these data types, we have equipped the calculus with recursion operators (`rec` and `list_rec`) of Gdel's System T[6]. However, in the presence of exceptions, the usual typing of these

---

[5]Proofs for all those relations follow the same pattern. Corruption is introduced on the right with *(ex-uni)* and *(ex-corrupt)* and is then distributed over the operands of the arrow with *(eq-arrc)*. Lastly, double corruption $((A^\Delta)^\Delta)$ is eliminated with *(eq-cc)* if needed.

operators is not precise enough. Indeed, consider the case of the natural numbers. The usual typing rule of the recursion operator `rec` is

$$\forall \alpha. \, \alpha \to (\mathbb{N} \to \alpha \to \alpha) \to \alpha \to \alpha,$$

and hence, using the corruption type and its associated subtyping rules, it also has the type

$$\forall \alpha. \, \alpha^{\Delta} \to (\mathbb{N}^{\Delta} \to \alpha^{\Delta} \to \alpha^{\Delta}) \to \alpha^{\Delta} \to \alpha^{\Delta}$$

for any set of exception names Δ. However, this last type is not precise enough and for instance, it does not precisely account for the reduction rule

$$\texttt{rec } X \ Y \ (\texttt{raise}\,\varepsilon) \ \succ \ \texttt{raise}\,\varepsilon.$$

Dealing with this imprecision is the reason of the addition of the set Δ' in the typing rule of `rec` (rule *(rec)* of Figure 3). Moreover, the function *eval* which will be introduced in section 4 reveals another imprecision. Given a corrupted natural number, this function returns either a well formed natural number or an exception at top-level. But to give this function the type we want, that is to say the type $\mathbb{N}^{\Delta} \to \mathbb{N} \uplus \Delta$, we need the addition of the set Δ in the typing rule of the recursion operator (rule *(rec)* of Figure 3). The typing rule of the recursion operator `list_rec` follows the same modifications.

3.3. **Properties of typing.** The subtyping relation is stable by corruption:

Theorem 3.11. *If A and B are two types such that $A \leq B$, then for any set of exception names Δ, $A^{\Delta} \leq B^{\Delta}$.*

Proof. We proceed by induction on the derivation of $A \leq B$. All the cases are easily resolved since corruption commutes with all type constructions. For example, taking the case of rule *(ex-arru)*, we have to show that $((A \to B) \uplus \Delta')^{\Delta} \leq (A \to B \uplus \Delta')^{\Delta}$. But using rule *(ex-arru)*, $(A^{\Delta} \to B^{\Delta}) \uplus \Delta' \leq A^{\Delta} \to B^{\Delta} \uplus \Delta'$ and we conclude using the fact that $(A^{\Delta} \to B^{\Delta}) \uplus \Delta' \ \dot{=} \ ((A \to B) \uplus \Delta')^{\Delta}$ and $A^{\Delta} \to B^{\Delta} \uplus \Delta' \ \dot{=} \ (A \to B \uplus \Delta')^{\Delta}$. $\square$

A few remarkable subtyping rules are also easily derivable from the ones of Figure 4:

Theorem 3.12. *The following subtyping relations hold:*

$$
\begin{aligned}
A \uplus \emptyset &\ \leq\ A \\
(\forall \alpha. \, A) \uplus \Delta &\ \leq\ \forall \alpha. \, (A \uplus \Delta) \\
(\forall \alpha. \, A)^{\Delta} &\ \leq\ \forall \alpha. \, (A^{\Delta})
\end{aligned}
$$

Proof. Proof of $A \uplus \emptyset \leq A$ comes from rules *(ex-corrupt)* and *(ex-noexc)*. The proofs for $(\forall \alpha. \, A) \uplus \Delta \leq \forall \alpha. \, (A \uplus \Delta)$ and $(\forall \alpha. \, A)^{\Delta} \leq \forall \alpha. \, (A^{\Delta})$ are similar. For instance, for the former one, we use *(f-inst)* and *(ex-uni)* to show that $\forall \alpha. \, A \leq A \uplus \Delta$. Then, using *(ex-ctx)* and *(eq-uu)*, we show that $(\forall \alpha. \, A) \uplus \Delta \leq A \uplus \Delta$. And we conclude with *(f-gen)*. $\square$

$$\frac{}{M \sqsubseteq_\Delta M} \; \textit{(c-id)} \qquad \frac{\varepsilon \in \Delta}{M \sqsubseteq_\Delta \mathtt{raise}\,\varepsilon} \; \textit{(c-rai)} \qquad \frac{M \sqsubseteq_\Delta M'}{\lambda x.\, M \sqsubseteq_\Delta \lambda x.\, t'} \; \textit{(c-lam)}$$

$$\frac{M \sqsubseteq_\Delta M' \quad N \sqsubseteq_\Delta N'}{M\ N \sqsubseteq_\Delta M'\ N'} \; \textit{(c-app)} \qquad \frac{M \sqsubseteq_\Delta M' \quad N \sqsubseteq_\Delta N'}{\mathtt{try}\,M\,\mathtt{with}\,\varepsilon \mapsto N \sqsubseteq_\Delta \mathtt{try}\,M'\,\mathtt{with}\,\varepsilon \mapsto N'} \; \textit{(c-try)}$$

Figure 5: Corruption relation

Definition 3.13. The *corruption relation* $\sqsubseteq_\Delta$ between terms is inductively defined Figure 5. To have $M \sqsubseteq_\Delta N$ means that N is obtained from M by replacing some sub-terms in any position by $\mathtt{raise}\,\varepsilon$, ε belonging to Δ.

Thus, Theorem 3.14 formally states that, in term of programming, exceptions can be used in any place, but with the added cost of corrupting the type.

Theorem 3.14 (corruption). *If M and N are two terms, A a type and Δ a set of exceptions such that $\Gamma \vdash M : A$ and $M \sqsubseteq_\Delta N$, then $\Gamma \vdash N : A^\Delta$.*

Proof. This theorem is proved by induction on the statement $M \sqsubseteq_\Delta N$. The proof presents no major difficulty as long as we first prove the three following "inversion" results :

(1) If M is a term, A a type and Γ a typing context such that

$$\Gamma \vdash \lambda x.\, M : A,$$

then there exists a set of type variable $\overrightarrow{\alpha} \notin FV(\Gamma)$ and two terms B and C such that $\forall \overrightarrow{\alpha}.\,(B \to C) \le A$ and $\Gamma, x : B \vdash M : C$.

(2) If M and N are two terms, A a type and Γ a typing context such that

$$\Gamma \vdash M\ N : A\ ,$$

then there exists a term C such that $\Gamma \vdash M : C \to A$ and $\Gamma \vdash N : C$.

(3) If M and N are two terms, A is a type and Γ is a typing context such that

$$\Gamma \vdash \mathtt{try}\,M\,\mathtt{with}\,\varepsilon \mapsto N : A,$$

then $\Gamma \vdash M : A \uplus \{\varepsilon\}$ and $\Gamma \vdash N : A$.

Proofs of these three results are straightforward inductions on the derivation of the initial typing judgment. □

4. Examples

A simple yet classical function on natural numbers which can raise an exception is the predecessor function. In Fx, we can define:

$$\mathtt{pred} \quad \equiv \quad \mathtt{rec}\,(\mathtt{raise}\,\varepsilon)\,(\lambda x.\,\lambda y.\,x) \quad : \quad \mathbb{N} \to \mathbb{N} \uplus \{pred\_err\}$$

It has the expected reductions, i.e. $\mathtt{pred}\,0 \succ^* \mathtt{raise}\,\varepsilon$ and $\mathtt{pred}\,(S\ N) \succ^* N$. We can then define a "safe" predecessor $\mathtt{pred}'$ from $\mathtt{pred}$ which returns 0 when applied to 0:

$$\mathtt{pred}' \quad \equiv \quad \lambda n.\,\mathtt{try}\,(\mathtt{pred}\,n)\,\mathtt{with}\,pred\_err \mapsto 0 \quad : \quad \mathbb{N} \to \mathbb{N}$$

Having exceptions, it is possible to define the functions that return the head and the tail of a list:

$$hd \quad : \quad A \; \texttt{list} \to A \uplus \{hd\_fail\}$$
$$\equiv \quad \texttt{list\_rec} \; (\texttt{raise} \; hd\_fail) \; (\lambda e. \, \lambda l. \, \lambda\_. \, e)$$

$$tl \quad : \quad A \; \texttt{list} \to A \; \texttt{list} \uplus \{tl\_fail\}$$
$$\equiv \quad \texttt{list\_rec} \; (\texttt{raise} \; tl\_fail) \; (\lambda e. \, \lambda l. \, \lambda\_. \, l)$$

We can also define the Euclidean division $(div : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \uplus \{div\_by\_0\})$ and the mapping of a function to a list $(map : \forall\alpha. \forall\beta. (\alpha \to \beta) \to \alpha \; \texttt{list} \to \beta \; \texttt{list})$. The type system being modular, we can (and will) use these two functions without having to exhibit a particular implementation. But using them allows us to define the following function that maps the function $n \mapsto \frac{2}{n-1}$ to a list:

$$f \quad : \quad \mathbb{N} \; \texttt{list} \to (\mathbb{N}^{\{pred\_err\}} \uplus \{div\_by\_0\}) \; \texttt{list}$$
$$\equiv \quad map \; (\lambda n. \, div \; 10 \; (pred \; n))$$

Remark that the result is always a list but that can contain exceptional values. For instance, $f \; [2; 1; 5]$ computes the list $[5; \texttt{raise} \; div\_by\_0; 2]$ which does not reduces to $\texttt{raise} \; div\_by\_0$. Again, exceptions are values that propagate only when used. Now we can get the first element of the result of this function with:

$$g \quad : \quad \mathbb{N} \; \texttt{list} \to \mathbb{N}^{\{pred\_err, div\_by\_0\}} \uplus \{hd\_fail\}$$
$$\equiv \quad \lambda l. \, hd \; (f \; l)$$

We can apply g to some argument and catch the exception hd_fail with a $\texttt{try}$, but we cannot catch the two other exceptions since these ones are not necessarily at top-level. If we want to catch them, we need a function that evaluates a natural number potentially corrupted and returns either a well formed natural number or an exception. It is the purpose of the following function:

$$eval \quad : \quad \mathbb{N}^{\Delta} \to \mathbb{N} \uplus \Delta$$
$$\equiv \quad \lambda n. \, (\texttt{rec} \; (\lambda a. \, a) \; (\lambda m. \, \lambda r. \, \lambda a. \, r \; (S \; a)) \; n) \; 0$$

With this function, we can now capture the exceptions that can appear in the result of the function g above. That is what the following function does (where we use a straightforward shortcut allowing the $\texttt{try}$ to catch all the exceptions):

$$h \quad : \quad \mathbb{N} \; \texttt{list} \to \mathbb{N}$$
$$\equiv \quad \lambda l. \, \texttt{try} \; (eval \; (g \; l))$$
$$\texttt{with} \; pred\_err, div\_by\_0, hd\_fail \mapsto 0$$

Note that for instance $h \; [2; 1; 5]$ will return 5 in our system since the part of the list that would yield an exception (the second element after the mapping) is never used (we only use the head of the list). In contrast, a similar function in say Caml would have yielded 0.

5. Realizability model

We will define a realizability model for Fx using techniques of orthogonality (see [12, 20] for examples of use of such techniques). The choice of those orthogonality techniques is mainly motivated by two reasons: we believe that it offers a nice way to handle second order and it will come in handy for the definition of the interpretation of corruption, allowing a much more simple definition than a "direct" model would allow. We start by introducing a few definitions necessary to the construction of the model.

5.1. **Daimon and contexts.** We add a new and distinguished term, the *daimon* (denoted ✠) similar to the one of [5]. This term computationally behaves like an uncatchable exception. We also introduce the new term construction $M; N$. This construction tests if M is the daimon and if so, return N. Otherwise, it does not reduce. The reduction rules for these two additions are given Figure 6. Moreover, ✠ is added to the definition of value. Note that none of these constructs have typing rules and as such, they cannot be used in well-typed terms. It can also easily be proved that they do not break the confluence property of the language. In those respects, they are only convenient technical addition for the model and should not be considered as inherent part of the language.

$$
\begin{array}{rcl}
✠\, N & > & ✠ \\
\mathtt{try}\,✠\,\mathtt{with}\,\varepsilon \mapsto N & > & ✠ \\
\mathtt{rec}\,X\,Y\,✠ & > & ✠ \\
\mathtt{list\_rec}\,X\,Y\,✠ & > & ✠ \\
✠; N & > & N
\end{array}
$$

Figure 6: Reduction rules for ✠ and ;.

The daimon has two purposes in the model. First, it will inhabit all type interpretation, property that will be used to show that all the terms of the interpretation are weakly normalizing (see Lemma 5.13). Secondly, our model is a realizability one, types will be interpreted by sets of terms. But the principle of our orthogonality model is to not define those sets directly, but instead to first define the interpretation of types as sets of evaluation contexts. Then, to each such set S of evaluation contexts is associated the set of all the terms that "behave correctly" for all the contexts of S. This notion of a term M "behaving correctly" in a context C is the orthogonality relation. For our model we chose it to be that M put in the context C reduces to this distinguished term ✠. But to define formally this orthogonality relation, let us first define formally the evaluation contexts we will use:

Definition 5.1 (Context). A *context* is a term with a hole (denoted by $[\,]$) defined by:

$$C ::= [\,]\mid C\,N \mid \mathtt{try}\,C\,\mathtt{with}\,\varepsilon \mapsto ✠ \mid \mathtt{rec}\,M\,N\,C \mid \mathtt{list\_rec}\,X\,Y\,C$$

The set of all contexts is noted $\mathcal{C}$ and the term obtained by filling the hole of a context C with the term M is noted $C[M]$.

Note that our definition of context is more restrictive than the usual one (where a context is *any* term with a hole). Actually, save the restriction in the handler of $\mathtt{try}$ to ✠ (which will allow for a simpler interpretation of corruption), our contexts are the evaluation contexts of call-by-name evaluation.

Moreover, we will not care about the order of two adjacent $\mathtt{try}$ in a context. Since the set of all exception names $\mathcal{E}$ is countable, we can fix *a priori* a bijection $\phi : \mathcal{E} \to \mathcal{N}$ and we define the following notation:

Notation 5.2. If Δ is a (finite) set of exception names, then $\mathtt{try}\,[\,]\,\mathtt{with}\,\Delta$ is a notation for

- the context $\mathtt{try}\,(\ldots(\mathtt{try}\,[\,]\,\mathtt{with}\,\varepsilon_1 \mapsto ✠)\ldots)\,\mathtt{with}\,\varepsilon_n \mapsto ✠$ if $\Delta \neq \emptyset$ and if $\varepsilon_1, \ldots \varepsilon_n$ are the elements of Δ arranged according to ϕ (that is, $\phi(\varepsilon_1) < \ldots < \phi(\varepsilon_n)$).

• the empty context otherwise (if $\Delta = \emptyset$).

Contexts have the following property :

Lemma 5.3. *If C is a context and M is a term such that $C[M]$ has a value, then M has a value.*

Proof. By case on the form of the context C and by induction on the length of the reduction of $C[M]$ to the value. No case raises specific difficulties. □

5.2. **Orthogonality relation.**

Definition 5.4 (Orthogonality relation). If M is a term and C a context, then $M \perp C$ (and we say that M and C are orthogonal) if and only if $C[M] \succ^* ✠$.

Moreover, if S is a set of contexts, we define the set of terms $S^\perp$ by

$$S^\perp \;\; = \;\; \{\, M \mid \forall C \in S,\ M \perp C \,\}$$

Note that as with any orthogonality relation, we can easily check that $\perp$ verifies the following properties:

Lemma 5.5. *If S and T are two context sets such that $S \subseteq T$, then $T^\perp \subseteq S^\perp$.*

Lemma 5.6. *If I is any set and $(S_i)_{i \in I}$ is a family of set of contexts indexed by i, then $(\bigcup_{i \in I} S_i)^\perp \;=\; \bigcap_{i \in I} S_i{}^\perp$.*

5.3. **Operations on sets.** We recall the two standard definition of concatenation · (of a set of terms and a set of contexts) and composition ∘ (of two sets of contexts):

$$
\begin{aligned}
A \cdot S &= \{\, C\big[[\,]\,N\big] \mid C \in S,\ N \in A \,\} \\
S \circ T &= \{\, C[D[\,]] \mid C \in S,\ D \in T \,\}
\end{aligned}
$$

For instance,

$$\{0,\ 1\} \cdot \{\mathtt{try}\,[\,]\,\mathtt{with}\,\varepsilon \mapsto ✠\} \;=\; \{\mathtt{try}\,([\,]\,0)\,\mathtt{with}\,\varepsilon \mapsto ✠,\ \mathtt{try}\,([\,]\,1)\,\mathtt{with}\,\varepsilon \mapsto ✠\}$$
$$\{\mathtt{try}\,[\,]\,\mathtt{with}\,\varepsilon \mapsto ✠,\ [\,]\,(\lambda x.\,x)\} \circ \{[\,]\,1\} \;=\; \{\mathtt{try}\,([\,]\,1)\,\mathtt{with}\,\varepsilon \mapsto ✠,\ ([\,]\,1)\,(\lambda x.\,x)\}$$

We then define two operations on sets of contexts:

$$
\begin{aligned}
\downarrow_\Delta S &= S \circ \{\, \mathtt{try}\,[\,]\,\mathtt{with}\,\Delta \,\} \\
\uparrow_\Delta S &= \{\, \mathtt{try}\,[\,]\,\mathtt{with}\,\Delta \,\} \circ S
\end{aligned}
$$

and thus, for instance,

$$
\begin{aligned}
C_1 &\equiv \downarrow_\Delta (\mathtt{rec}\ 0\ (\lambda x.\,\lambda y.\,y)\ [\,]) = \{\, \mathtt{rec}\ 0\ (\lambda x.\,\lambda y.\,y)\ (\mathtt{try}\,[\,]\,\mathtt{with}\,\Delta) \,\} \\
C_2 &\equiv \uparrow_\Delta (\mathtt{rec}\ 0\ (\lambda x.\,\lambda y.\,y)\ [\,]) = \{\, \mathtt{try}\,(\mathtt{rec}\ 0\ (\lambda x.\,\lambda y.\,y)\ [\,])\,\mathtt{with}\,\Delta \,\}
\end{aligned}
$$

and if $\varepsilon \in \Delta$,

$$
\begin{aligned}
C_1[\mathtt{raise}\,\varepsilon] &= \mathtt{rec}\ 0\ (\lambda x.\,\lambda y.\,y)\ (\mathtt{try}\,\mathtt{raise}\,\varepsilon\,\mathtt{with}\,\Delta) & \succ^* ✠ \\
C_1[S\ (\mathtt{raise}\,\varepsilon)] &= \mathtt{rec}\ 0\ (\lambda x.\,\lambda y.\,y)\ (\mathtt{try}\,S\ (\mathtt{raise}\,\varepsilon)\,\mathtt{with}\,\Delta) & \succ^* \mathtt{raise}\,\varepsilon \\
C_2[\mathtt{raise}\,\varepsilon] &= \mathtt{try}\,(\mathtt{rec}\ 0\ (\lambda x.\,\lambda y.\,y)\ (\mathtt{raise}\,\varepsilon))\,\mathtt{with}\,\Delta & \succ^* ✠ \\
C_2[S\ (\mathtt{raise}\,\varepsilon)] &= \mathtt{try}\,(\mathtt{rec}\ 0\ (\lambda x.\,\lambda y.\,y)\ (S\ (\mathtt{raise}\,\varepsilon)))\,\mathtt{with}\,\Delta & \succ^* ✠
\end{aligned}
$$

It can be checked that by definition we have the following equalities:

$$\begin{aligned}
\uparrow_\Delta(A \cdot S) &= A \cdot \uparrow_\Delta S \\
\uparrow_\Delta(\downarrow_{\Delta'} S) &= \downarrow_{\Delta'}(\uparrow_\Delta S)
\end{aligned}$$

Moreover, we have the following lemma:

Lemma 5.7. *If S is a set of contexts and Δ and Δ' are sets of exception names, then*

$$\begin{aligned}
(\uparrow_\Delta(\uparrow_{\Delta'} S))^\perp &= (\uparrow_{\Delta \cup \Delta'} S)^\perp \\
(\downarrow_\Delta(\downarrow_{\Delta'} S))^\perp &= (\downarrow_{\Delta \cup \Delta'} S)^\perp
\end{aligned}$$

Proof. We only give the proof for $(\uparrow_\Delta(\uparrow_{\Delta'} S))^\perp = (\uparrow_{\Delta \cup \Delta'} S)^\perp$ since the other one is similar. We show two inclusions:

- We show that $(\uparrow_\Delta(\uparrow_{\Delta'} S))^\perp \subseteq (\uparrow_{\Delta \cup \Delta'} S)^\perp$:
  Let $t \in (\uparrow_\Delta(\uparrow_{\Delta'} S))^\perp$ and $C \in \uparrow_{\Delta \cup \Delta'} S$, by definition $C = \mathtt{try}\, D\, \mathtt{with}\, (\Delta \cup \Delta')$ where $D \in S$. Hence $\mathtt{try}\,(\mathtt{try}\, D[t]\, \mathtt{with}\, \Delta')\, \mathtt{with}\, \Delta \succ^* \maltese$ and
  - either $D[t] \succ^* \maltese$, but then $C[t] \succ^* \maltese$.
  - or $D[t] \succ^* \mathtt{raise}\, \varepsilon$ when $\varepsilon \in \Delta'$ or $\varepsilon \in \Delta$. But then again, $C[t] \succ^* \maltese$.
- We show $(\uparrow_{\Delta \cup \Delta'} S)^\perp \subseteq (\uparrow_\Delta(\uparrow_{\Delta'} S))^\perp$:
  Let $t \in (\uparrow_{\Delta \cup \Delta'} S)^\perp$ and $C \in \uparrow_\Delta(\uparrow_{\Delta'} S)$, by definition, $C = \mathtt{try}\,(\mathtt{try}\, D\, \mathtt{with}\, \Delta')\, \mathtt{with}\, \Delta$ where $D \in S$. Hence $\mathtt{try}\, D[t]\, \mathtt{with}\, (\Delta \cup \Delta') \succ^* \maltese$ and
  - either $D[t] \succ^* \maltese$, but then $C[t] \succ^* \maltese$.
  - or $D[t] \succ^* \mathtt{raise}\, \varepsilon$ when $\varepsilon \in \Delta \cup \Delta'$. But then again, $C[t] \succ^* \maltese$. $\qquad\square$

Along with the definition of $\uparrow_\Delta$, this lemma implies $(A \cdot \uparrow_\Delta(\uparrow_{\Delta'} S))^\perp = (A \cdot \uparrow_{\Delta \cup \Delta'} S)^\perp$.

5.4. **Model definition.** We call valuation function any function ρ from type variables to the power set of $\mathcal{C}$ minus the empty set ($\rho : \mathcal{A} \to (\mathcal{P}(\mathcal{C}))^+$). To each type A we associate two sets:

A set of contexts	$\mid A \mid_\rho \subseteq \mathcal{C}$
A set of terms	$[\![A]\!]_\rho \subseteq \mathcal{T}$

The set $[\![A]\!]_\rho$ is uniformly defined from $\mid A \mid_\rho$ by

$$[\![A]\!]_\rho = \mid A \mid_\rho{}^\perp = \{\, M \mid \forall C \in \mid A \mid_\rho,\ M \perp C \,\}.$$

The set $\mid A \mid_\rho$ is defined by induction on A. Its definition is given Figure 7.

Note that the interpretation in the model of the construction $A \uplus \Delta$ and A^Δ follows, to some extends, the idea that terms of type $A \uplus \Delta$ are terms that may raise an exception only at top level, where terms of A^Δ are those that may raise an exception in any evaluation context. This is emphasized by the "opposition" of the operations $\downarrow_\Delta$ and $\uparrow_\Delta$. Remark that it is only the restriction to $\maltese$ in the handlers of $\mathtt{try}$ contexts that allows for such a simple definition of the interpretation of corruption. Indeed, thanks to this restriction we ensure that for any context C, $C[\mathtt{raise}\, \varepsilon]$ will always reduce to $\mathtt{raise}\, \varepsilon$ or $\maltese$.

The other interesting point of the model is the interpretation of arrow types. In Fx, a function f which has type $A \to B$ has also all the types $A^\Delta \to B^\Delta$ for any Δ. Our arrow type is thus smaller than the usual realizability one and so, functions of Fx are in particular realizability functions. More formally,

$$
\begin{array}{rcl}
|\,\alpha\,|_\rho & = & \rho(\alpha) \\[4pt]
|\,\mathbb{N}\,|_\rho & = & \{\,\mathtt{rec}\ \maltese\ (\lambda y.\,\lambda x.\,x)\,[\,]\,\} \\[4pt]
|\,A\ \mathtt{list}\,|_\rho & = & \{\,\mathtt{list\_rec}\ \maltese\ (\lambda e.\,\lambda l.\,\lambda r.\,(|\,A\,|_\rho)[e]; r)\,[\,]\,\} \\[4pt]
|\,A \uplus \Delta\,|_\rho & = & \downarrow_\Delta |\,A\,|_\rho \\[4pt]
|\,A^\Delta\,|_\rho & = & \uparrow_\Delta |\,A\,|_\rho \\[4pt]
|\,A \to B\,|_\rho & = & \displaystyle\bigcup_{\Delta \subseteq \mathcal{E}} |\,A^\Delta\,|_\rho{}^{\perp}\cdot|\,B^\Delta\,|_\rho \\[10pt]
|\,\forall \alpha.\,A\,|_\rho & = & \displaystyle\bigcup_{S \subseteq \mathcal{C}^+} |\,A\,|_{\rho;\,\alpha \leftarrow S}
\end{array}
$$

Figure 7: Definition of the $|\,A\,|_\rho$ set of contexts

Lemma 5.8. *If A and B are two types and ρ a valuation function, then*

$$
[\![\,A \to B\,]\!]_\rho \;=\; \bigcap_{\Delta \subseteq \mathcal{E}} \{\,M \mid \forall N \in [\![\,A^\Delta\,]\!]_\rho,\ M\,N \in [\![\,B^\Delta\,]\!]_\rho\,\}.
$$

Proof. We prove the two inclusions forming the equality separately, both being simple consequences of definitions. $\square$

We can moreover show that the interpretation of $A \uplus \Delta$ is a union and that the interpretations of the natural numbers and the lists are standards:

Lemma 5.9. *If A is a type, Δ a set of exception names and ρ a valuation function, then*

$$
[\![\,A \uplus \Delta\,]\!]_\rho = [\![\,A\,]\!]_\rho \cup \{\,M \mid M \succ^* \mathtt{raise}\,\varepsilon,\ \varepsilon \in \Delta\,\}.
$$

Proof. By definition,

$$
[\![\,A \uplus \Delta\,]\!]_\rho = (\downarrow_\Delta |\,A\,|_\rho)^{\perp} = \{\,M \mid \forall C \in |\,A\,|_\rho,\ C[\mathtt{try}\,M\,\mathtt{with}\,\Delta] \succ^* \maltese\,\}\,.
$$

We show each side of the inclusion separately:

- For $M \in \{\,t \mid \forall C \in |\,A\,|_\rho,\ C[\mathtt{try}\,M\,\mathtt{with}\,\Delta] \succ^* \maltese\,\}$ we have $\mathtt{try}\,M\,\mathtt{with}\,\Delta \in [\![\,A\,]\!]_\rho$.
  Hence $\mathtt{try}\,M\,\mathtt{with}\,\Delta$ has a value (Lemma 5.13) and M as well (Lemma 5.3) :
  - Either $M \succ^* \mathtt{raise}\,\varepsilon$ for $\varepsilon \in \Delta$ and we conclude directly.
  - Or $M \succ^* V$ where V is a regular value and then $\mathtt{try}\,M\,\mathtt{with}\,\Delta \succ^* V$. But the interpretation being closed by equivalence (Lemma 5.12), $V \in [\![\,A\,]\!]_\rho$ and $M \in [\![\,A\,]\!]_\rho$.
- Let $M \in [\![\,A\,]\!]_\rho \cup \{\,M \mid M \succ^* \mathtt{raise}\,\varepsilon,\ \varepsilon \in \Delta\,\}$ and let $C \in |\,A\,|_\rho$. We have to show $C[\mathtt{try}\,M\,\mathtt{with}\,\Delta] \succ^* \maltese$:
  - If $M \succ^* \mathtt{raise}\,\varepsilon$ for $\varepsilon \in \Delta$, then $\mathtt{try}\,M\,\mathtt{with}\,\Delta \succ^* \maltese$ and $C[\mathtt{try}\,M\,\mathtt{with}\,\Delta] \succ^* \maltese$.
  - Otherwise, $M \in [\![\,A\,]\!]_\rho$ and then $\mathtt{try}\,M\,\mathtt{with}\,\Delta \in [\![\,A\,]\!]_\rho$ (because, M having a value (Lemma 5.13), there exists V such that $M \succ^* V$. But then $\mathtt{try}\,M\,\mathtt{with}\,\Delta \succ^* V$ and we use the closure by equivalence of the interpretation (Lemma 5.12)). Finally, using the definition of orthogonality, $C[\mathtt{try}\,M\,\mathtt{with}\,\Delta] \succ^* \maltese$. $\square$

Lemma 5.10. *If ρ is a valuation function, Δ a set of exception names and if Φ represents one of 0, ✠ or $\mathtt{raise}\,\varepsilon$ for some $\varepsilon \in \Delta$, then*

$$[\![\,\mathbb{N}^\Delta\,]\!]_\rho = \{\ M \mid M \succ^* S^n\ \Phi, n \in \mathbb{N}\ \}.$$

Proof. In the following, Φ will always represent one of 0, ✠ or $\mathtt{raise}\,\varepsilon$ for some $\varepsilon \in \Delta$.

- If for some integer n, $M \succ^* S^n\ \Phi$, then by induction on n it is easy to show that $\mathtt{try}\,(\mathtt{rec}\ ✠\ (\lambda y.\,\lambda x.\,x)\ M)\,\mathtt{with}\,\Delta \succ^* ✠$.
- If $M \in [\![\,\mathbb{N}^\Delta\,]\!]_\rho$, then there exists k such that $\mathtt{try}\,(\mathtt{rec}\ ✠\ (\lambda y.\,\lambda x.\,x)\ M)\,\mathtt{with}\,\Delta \succ^k ✠$. Hence, we show by induction on k that for any $k' \leq k$ and for any term M:

$$\text{if}\quad \mathtt{try}\,(\mathtt{rec}\ ✠\ (\lambda y.\,\lambda x.\,x)\ M)\,\mathtt{with}\,\Delta \succ^{k'} ✠\quad \text{then}\quad M \succ^* S^n\ \Phi$$

  - We cannot have $k = 0$ since $\mathtt{try}\,(\mathtt{rec}\ ✠\ (\lambda y.\,\lambda x.\,x)\ M)\,\mathtt{with}\,\Delta$ is not ✠.
  - We have $\mathtt{try}\,(\mathtt{rec}\ ✠\ (\lambda y.\,\lambda x.\,x)\ M)\,\mathtt{with}\,\Delta \succ N \succ^k ✠$. But the first reduction can only occurs either if M is 0, ✠ or $\mathtt{raise}\,\varepsilon$ (and in the last case we have $\varepsilon \in \Delta$), or if M is $S\ M'$. In this last case, we easily conclude using the reduction of $\mathtt{rec}$ and the induction hypothesis. □

Lemma 5.11. *If ρ is a valuation function, A a type, Δ a set of exception names and if Φ represents one of 0, ✠ or $\mathtt{raise}\,\varepsilon$ for some $\varepsilon \in \Delta$, then*

$$[\![\,(A\ \mathtt{list})^\Delta\,]\!]_\rho$$
$$= \{\ M \mid M \succ^* \mathtt{cons}\ a_0\ (\ldots(\mathtt{cons}\ a_n\ \Phi)\ldots), n \in \mathbb{N}, \forall 0 \leq i \leq n,\ a_i \in [\![\,A^\Delta\,]\!]_\rho\ \}\ .$$

Proof. The proof follows the same structure as the one of Lemma 5.10. □

5.5. **Model properties.**

Lemma 5.12 (closure by equivalence)**.** *If M and N are two terms, A is a type and ρ is a valuation function such that $M \in [\![\,A\,]\!]_\rho$ and $M = N$, then*

$$N \in [\![\,A\,]\!]_\rho.$$

Proof. Let $M \in [\![\,A\,]\!]_\rho$ and $M = N$. Let $C \in |\,A\,|_\rho$, by definition $C[M] \succ^* ✠$. But since $M = N$, $C[M] = C[N]$. Thus, by confluence of the reduction (theorem 3.6), $C[N] \succ^* ✠$ and $N \in [\![\,A\,]\!]_\rho$. □

In particular, the interpretation is closed by reduction and anti-reduction.

Lemma 5.13. *If M is a term, A a type and ρ a valuation function such that $M \in [\![\,A\,]\!]_\rho$, then M has a value.*

Proof. By definition $M \in [\![\,A\,]\!]_\rho$ yields $M \in \{\ M \mid \forall C \in |\,A\,|_\rho,\ C[M] \succ^* ✠\ \}$. Thus, if $C \in |\,A\,|_\rho$, $C[M]$ has a value and, using Lemma 5.3, we have that M has a value too. We simply have to make sure that there always exists such a context C, that is $|\,A\,|_\rho$ is never empty. But for any type A, it can be easily proved by induction on A that $|\,A\,|_\rho \neq \emptyset$ and $✠ \in [\![\,A\,]\!]_\rho$ (remark that both properties have to be proved simultaneously since the non-emptiness of $|\,A \to B\,|_\rho$ depends upon the non-emptiness of $[\![\,A\,]\!]_\rho$ which comes (by induction hypothesis) of the non-emptiness of $|\,A\,|_\rho$). □

Lemma 5.14. *If A is a type, ρ a valuation function and Δ a set of exception names, then for all $\varepsilon \in \Delta$, $\mathtt{raise}\,\varepsilon \in [\![\, A^\Delta \,]\!]_\rho$.*

Proof. Let $C \in |\, A^\Delta \,|_\rho$, by definition $C = \{\mathtt{try}\,[\,]\,\mathtt{with}\,\Delta\} \circ D$ where $D \in |\, A \,|_\rho$. We can easily show that either $D[\mathtt{raise}\,\varepsilon] \succ^* \maltese$ or $D[\mathtt{raise}\,\varepsilon] \succ^* \mathtt{raise}\,\varepsilon$. In any case, $C[\mathtt{raise}\,\varepsilon] \succ^* \maltese$. $\qquad\square$

An important and essential property of the model is that it validates the subtyping rule *(eq-arrc)*:

Lemma 5.15. *If A and B are two types, ρ is valuation function and Δ is a set of exception names, then*
$$[\![\, (A \to B)^\Delta \,]\!]_\rho \quad = \quad [\![\, A^\Delta \to B^\Delta \,]\!]_\rho$$

Proof. Using Lemmas 5.6 and 5.7, we have that
$$[\![\, (A \to B)^\Delta \,]\!]_\rho = \bigcap_{\Delta' \subseteq \mathcal{E}} ((\uparrow_{\Delta'} |\, A \,|_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} |\, B \,|_\rho)^\perp$$
$$[\![\, A^\Delta \to B^\Delta \,]\!]_\rho = \bigcap_{\Delta' \subseteq \mathcal{E}} ((\uparrow_{\Delta \cup \Delta'} |\, A \,|_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} |\, B \,|_\rho)^\perp$$

It directly follows that $[\![\, (A \to B)^\Delta \,]\!]_\rho \subseteq [\![\, A^\Delta \to B^\Delta \,]\!]_\rho$.

For the other inclusion, let $M \in [\![\, A^\Delta \to B^\Delta \,]\!]_\rho$. If Δ' is a set of exception names and if $C \in (\uparrow_{\Delta'} |\, A \,|_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} |\, B \,|_\rho$, since we can show that $(\uparrow_{\Delta'} |\, A \,|_\rho)^\perp \subseteq (\uparrow_{\Delta \cup \Delta'} |\, A \,|_\rho)^\perp$ we have $C \in (\uparrow_{\Delta \cup \Delta'} |\, A \,|_\rho)^\perp \cdot \uparrow_{\Delta \cup \Delta'} |\, B \,|_\rho$ and we conclude. $\qquad\square$

Lemma 5.16. *The interpretations validate the following equalities:*
$$\begin{array}{rcl} |\, (\forall \alpha.\, A)^\Delta \,|_\rho & = & |\, \forall \alpha.\, A^\Delta \,|_\rho \\ |\, (\forall \alpha.\, A) \uplus \Delta \,|_\rho & = & |\, \forall \alpha.\, A \uplus \Delta \,|_\rho \\ |\, (A \uplus \Delta)^{\Delta'} \,|_\rho & = & |\, (A^{\Delta'}) \uplus \Delta \,|_\rho \\ [\![\, (A^\Delta)^{\Delta'} \,]\!]_\rho & = & [\![\, A^{\Delta \cup \Delta'} \,]\!]_\rho \\ [\![\, (A \uplus \Delta) \uplus \Delta' \,]\!]_\rho & = & [\![\, A \uplus (\Delta \cup \Delta') \,]\!]_\rho \end{array}$$

Proof. The three first equalities are direct consequence of the definitions, the two last are direct consequence of Lemma 5.7. $\qquad\square$

5.6. Model soundness.
We first show that subtyping is sound with respect to the interpretation we have defined.

Lemma 5.17 (Subtyping soudness). *If A and B are two types and ρ a valuation function such that $A \leq B$, then for any set of exception names Δ,*
$$[\![\, A^\Delta \,]\!]_\rho \subseteq [\![\, B^\Delta \,]\!]_\rho$$

Proof. We reason by induction on the derivation of $A \leq B$. Many cases are either trivial (*(st-id)* and *(st-trans)*) or direct consequences of the lemmas we have defined so far. We only give in the following the cases that do not belong to one of these categories:

(st-arrow): Consider $\Delta \subseteq \mathcal{E}$ and $M \in [\![\, (A \to B)^\Delta \,]\!]_\rho = [\![\, A^\Delta \to B^\Delta \,]\!]_\rho$. We now have to show that $M \in [\![\, (A' \to B')^\Delta \,]\!]_\rho = [\![\, A'^\Delta \to B'^\Delta \,]\!]_\rho$. For $C \in |\, A'^\Delta \to B'^\Delta \,|_\rho$ we will establish $C[M] \succ^* \maltese$. By definition of $|\, A'^\Delta \to B'^\Delta \,|_\rho$, there exist Δ', $N \in [\![\, A'^{\Delta \cup \Delta'} \,]\!]_\rho$ and $D \in |\, B'^{\Delta \cup \Delta'} \,|_\rho$ such that $C = D[[\,]\,N]$. By induction hypothesis, $[\![\, A'^{\Delta \cup \Delta'} \,]\!]_\rho \subseteq [\![\, A^{\Delta \cup \Delta'} \,]\!]_\rho$.

But since $M \in [\![A^\Delta \to B^\Delta]\!]_\rho$, using Lemma 5.8, we have $M\ N \in [\![B^{\Delta \cup \Delta'}]\!]_\rho$. Then by induction hypothesis, $[\![B^{\Delta \cup \Delta'}]\!]_\rho \subseteq [\![B'^{\Delta \cup \Delta'}]\!]_\rho$ and finally, $C[M] = D[M\ N] \succ^* \maltese$.

(f-gen): For $\Delta \subseteq \mathcal{E}$ and $M \in [\![A^\Delta]\!]_\rho$ we will show that $M \in [\![(\forall \alpha.\, B)^\Delta]\!]_\rho = [\![\forall \alpha.\, B^\Delta]\!]_\rho$. Let $C \in |\,\forall \alpha.\, B^\Delta\,|_\rho$, there exists S such that $C \in |\,B^\Delta\,|_{\rho;\, \alpha \leftarrow S}$. Moreover, since $\alpha \notin \mathrm{FV}(A)$, we have $[\![A^\Delta]\!]_\rho = [\![A^\Delta]\!]_{\rho;\, \alpha \leftarrow S}$. And since $[\![A^\Delta]\!]_{\rho;\, \alpha \leftarrow S} \subseteq [\![B^\Delta]\!]_{\rho;\, \alpha \leftarrow S}$ by induction hypothesis, $M \in [\![B^\Delta]\!]_{\rho;\, \alpha \leftarrow S}$ and finally, $C[M] \succ^* \maltese$.

(f-inst): Given $\Delta \subseteq \mathcal{E}$, we will show that $|\,(A\{\alpha := B\})^\Delta\,|_\rho \subseteq |\,(\forall \alpha.\, A)^\Delta\,|_\rho$ and then conclude by orthogonality (Lemma 5.5). Let $C \in |\,(A\{\alpha := B\})^\Delta\,|_\rho$, we show by a straightforward induction on A that $|\,(A\{\alpha := B\})^\Delta\,|_\rho = |\,A^\Delta\,|_{\rho;\, \alpha \leftarrow |\,B\,|_\rho}$. Moreover, by definition of $|\,\forall \alpha.\, A^\Delta\,|_\rho$, we have $|\,A^\Delta\,|_{\rho;\, \alpha \leftarrow |\,B\,|_\rho} \subseteq |\,\forall \alpha.\, A^\Delta\,|_\rho$, from which if follows that $C \in |\,\forall \alpha.\, A^\Delta\,|_\rho = |\,(\forall \alpha.\, A)^\Delta\,|_\rho$.

(f-distr): Consider $\Delta \subseteq \mathcal{E}$ and $t \in [\![(\forall \alpha.\, (A \to B))^\Delta]\!]_\rho = [\![\forall \alpha.\, (A^\Delta \to B^\Delta)]\!]_\rho$, we will show that $t \in [\![(A \to \forall \alpha.\, B)^\Delta]\!]_\rho = [\![A^\Delta \to \forall \alpha.\, B^\Delta]\!]_\rho$. Let $C \in |\,A^\Delta \to \forall \alpha.\, B^\Delta\,|_\rho$, by definition there exists Δ', $u \in [\![A^{\Delta \cup \Delta'}]\!]_\rho$ and $D \in |\,(\forall \alpha.\, B^\Delta)^{\Delta'}\,|_\rho = |\,\forall \alpha.\, B^{\Delta \cup \Delta'}\,|_\rho$ such that $C = D[[\ ]\ u]$. Then there exists S such that $D \in |\,B^{\Delta \cup \Delta'}\,|_{\rho;\, \alpha \leftarrow S}$ and since $\alpha \notin \mathrm{FV}(A)$, $u \in [\![A^{\Delta \cup \Delta'}]\!]_{\rho;\, \alpha \leftarrow S}$. Thus by definition of $[\![\forall \alpha.\, (A^\Delta \to B^\Delta)]\!]_\rho$ and using Lemma 5.8, we have $t\ u \in [\![B^{\Delta \cup \Delta'}]\!]_{\rho;\, \alpha \leftarrow S}$ and finally $C[t] = D[t\ u] \succ^* \maltese$.

(ex-arru): Consider $\Delta \subseteq \mathcal{E}$ and $t \in [\![((A \to B) \uplus \Delta')^\Delta]\!]_\rho = [\![(A^\Delta \to B^\Delta) \uplus \Delta']\!]_\rho$, we have to show that $t \in [\![(A \to B \uplus \Delta')^\Delta]\!]_\rho = [\![A^\Delta \to B^\Delta \uplus \Delta']\!]_\rho$. Consider $\Delta'' \subseteq \mathcal{E}$ and $u \in [\![A^{\Delta \cup \Delta''}]\!]_\rho$, using Lemma 5.8, we must show that $t\ u \in [\![B^{\Delta \cup \Delta''} \uplus \Delta']\!]_\rho$. But using Lemma 5.9 we have:
- Either $t \succ^* \mathtt{raise}\,\varepsilon$ for $\varepsilon \in \Delta'$, but then $t\ u \succ^* \mathtt{raise}\,\varepsilon$ and hence using Lemma 5.9, $t\ u \in [\![B^{\Delta \cup \Delta''} \uplus \Delta']\!]_\rho$.
- Or $t \in [\![A^\Delta \to B^\Delta]\!]_\rho$ and using Lemma 5.8, $t\ u \in [\![B^{\Delta \cup \Delta''}]\!]_\rho$ which in turn gives $t\ u \in [\![B^{\Delta \cup \Delta''} \uplus \Delta']\!]_\rho$ (Lemma 5.9).

(ex-ctx): This case is trivial with the use of Lemma 5.9.

(ex-uni): This case is trivial with the use of Lemma 5.9.

(ex-corrupt): We need to show $[\![(A \uplus \Delta')^\Delta]\!]_\rho = [\![A^\Delta \uplus \Delta']\!]_\rho \subseteq [\![(A^{\Delta'})^\Delta]\!]_\rho = [\![A^{\Delta \cup \Delta'}]\!]_\rho$. But $[\![A^\Delta \uplus \Delta']\!]_\rho = [\![A^\Delta]\!]_\rho \cup \{\ t \mid t \succ^* \mathtt{raise}\,\varepsilon,\ \varepsilon \in \Delta'\ \}$ and it can be easily shown that if $\Delta \subseteq \Delta'$, then $[\![A^\Delta]\!]_\rho \subseteq [\![A^{\Delta'}]\!]_\rho$ and that if $\varepsilon \in \Delta$, then $\mathtt{raise}\,\varepsilon \in [\![A^\Delta]\!]_\rho$ (Lemma 5.14). $\qquad\square$

We define the interpretation corrupted by some set of exception names Δ (eventually empty) of a typing context Γ by:

$$[\![\Gamma^\Delta]\!]_\rho \quad = \quad \{\ \sigma \mid \forall\, (x : A) \in \Gamma,\ \sigma(x) \in [\![A^\Delta]\!]_\rho\ \}$$

Moreover, if σ is a substitution of term variables and M is a term, we use the notation $M[\sigma]$ for the *parallel substitution* of M by σ, which consists in applying σ to all free variables of M in parallel. We can now show that our interpretation is sound with respect to typing:

Theorem 5.18 (Model soundness). *If M is a term, A a type and Γ a typing context such that $\Gamma \vdash M : A$, then for all valuation function ρ, for all set of exception names Δ and for all substitution $\sigma \in [\![\Gamma^\Delta]\!]_\rho$, we have $M[\sigma] \in [\![A^\Delta]\!]_\rho$.*

Proof. We use induction on the derivation of $\Gamma \vdash t : A$. Note that since $[\![A]\!]_\rho \subseteq [\![A^\Delta]\!]_\rho$ (Lemma 5.17), we will only show that $t[\sigma] \in [\![A]\!]_\rho$ when possible. We give here only the

interesting cases. The other cases are either simple (*(ax)*, *(subs)*, *(zero)*, *(succ)*, *(nil)*, *(cons)*) or, for *(fold)*, follows closely the structure of the proof for *(rec)*.

(abs): Let $C \in \, |\, (A \to B)^\Delta \,|_\rho = |\, A^\Delta \to B^\Delta \,|_\rho$, we need to show that $C[\lambda x.\, t[\sigma]] \succ^* \maltese$. By definition, there exists Δ', $u \in [\![\, A^{\Delta'} \,]\!]_\rho$ and $D \in |\, B^{(\Delta \cup \Delta')} \,|_\rho$ such that $C = D[[\,]\, u]$. Then, if $\delta = \sigma + \{x \to u\}$, we have $\delta \in [\![\, (\Gamma, x : A)^{(\Delta \cup \Delta')} \,]\!]_\rho$ ($\sigma \in [\![\, \Gamma^{\Delta'} \,]\!]_\rho \subseteq [\![\, \Gamma^{(\Delta \cup \Delta')} \,]\!]_\rho$ and $u \in [\![\, A^{\Delta'} \,]\!]_\rho \subseteq [\![\, A^{\Delta \cup \Delta'} \,]\!]_\rho$), and also by induction hypothesis, $t[\delta] \in [\![\, B^{(\Delta \cup \Delta')} \,]\!]_\rho$. However, $(\lambda x.\, t[\sigma])\, u \succ t[\delta]$ and $[\![\, B^{(\Delta \cup \Delta')} \,]\!]_\rho$ is closed by anti-reduction (Lemma 5.12), and thus $(\lambda x.\, M[\sigma])\, u \in [\![\, B^{(\Delta \cup \Delta')} \,]\!]_\rho$ and finally $C[\lambda x.\, M[\sigma]] = D[(\lambda x.\, M[\sigma])\, u] \succ^* \maltese$.

(app): We easily conclude using Lemma 5.8.

(gen): Let $C \in \, |\, (\forall \alpha.\, A)^\Delta \,|_\rho = |\, \forall \alpha.\, A^\Delta \,|_\rho$, by definition there exists S non empty such that $C \in \, |\, A^\Delta \,|_{\rho; \, \alpha \leftarrow S}$. Moreover, since $\alpha \notin \mathrm{FV}(\Gamma)$, $[\![\, \Gamma^\Delta \,]\!]_\rho = [\![\, \Gamma^\Delta \,]\!]_{\rho; \, \alpha \leftarrow S}$. It follows that by induction hypothesis, $t[\sigma] \in [\![\, A^\Delta \,]\!]_{\rho; \, \alpha \leftarrow S}$. Finally, $C[t[\sigma]] \succ^* \maltese$ and $t[\sigma] \in [\![\, (\forall \alpha.\, A)^\Delta \,]\!]_\rho$.

(rec): We have to show that

$$\mathtt{rec} \in [\![\, \forall \alpha.\, \alpha \uplus \Delta \to (\mathbb{N}^\Delta \to \alpha \uplus \Delta \to \alpha \uplus \Delta) \to \mathbb{N}^\Delta \uplus \Delta' \to \alpha \uplus (\Delta \cup \Delta') \,]\!]_\rho.$$

Using Lemma 5.8, we have to show that for any non empty set of contexts S, for any Δ_1, Δ_2 and Δ_3 and for

$$
\begin{aligned}
z &\in & [\![\, \alpha^{\Delta_1} \uplus \Delta \,]\!]_{\rho; \, \alpha \leftarrow S} \\
f &\in & [\![\, \mathbb{N}^{\Delta_1 \cup \Delta_2 \cup \Delta} \to \alpha^{\Delta_1 \cup \Delta_2} \uplus \Delta \to \alpha^{\Delta_1 \cup \Delta_2} \uplus \Delta \,]\!]_{\rho; \, \alpha \leftarrow S} \\
n &\in & [\![\, \mathbb{N}^{\Delta_1 \cup \Delta_2 \cup \Delta_3} \uplus \Delta' \,]\!]_{\rho; \, \alpha \leftarrow S}
\end{aligned}
$$

we have $\mathtt{rec}\ z\ f\ n \in [\![\, \alpha^{\Delta_1 \cup \Delta_2 \cup \Delta_3} \uplus (\Delta \cup \Delta') \,]\!]_{\rho; \, \alpha \leftarrow S}$. With Lemma 5.9, we have either $n \succ^* \mathtt{raise}\, \varepsilon$ for $\varepsilon \in \Delta'$ (and we easily conclude), or $n \in [\![\, \mathbb{N}^{\Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \Delta} \,]\!]_{\rho; \, \alpha \leftarrow S}$. In this last case, there exists some k such that $n \succ^* S^k\ \Phi$ where Φ is one of 0, $\maltese$ or $\mathtt{raise}\, \varepsilon$ for $\varepsilon \in \Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \Delta$ (Lemma 5.10). We then proceed by induction on k. If $k = 0$ ($n \succ^* \Phi$), we easily conclude in each case of Φ. Otherwise, we must show that $\mathtt{rec}\ z\ f\ (S\ (S^k\ \Phi)) \in [\![\, \alpha^{\Delta_1 \cup \Delta_2 \cup \Delta_3} \uplus (\Delta \cup \Delta') \,]\!]_{\rho; \, \alpha \leftarrow S}$. But $\mathtt{rec}\ z\ f\ (S\ (S^k\ \Phi)) \succ f\ (S^k\ \Phi)\ (\mathtt{rec}\ z\ f\ (S^k\ \Phi))$. We then conclude using Lemma 5.8 with f, the fact that $S^k\ \Phi \in [\![\, \mathbb{N}^{\Delta_1 \cup \Delta_2 \cup \Delta_3 \cup \Delta} \,]\!]_\rho$ (Lemma 5.10) and the induction hypothesis.

(raise): It is a direct using Lemma 5.9.

(try): By induction hypothesis, $t[\sigma] \in [\![\, (A \uplus \Delta)^\Delta \,]\!]_\rho = [\![\, A^\Delta \uplus \Delta \,]\!]_\rho$ and $u[\sigma] \in [\![\, A^\Delta \,]\!]_\rho$. Using Lemma 5.9, we have either that $t[\sigma] \in [\![\, A^\Delta \,]\!]_\rho$ or that $t[\sigma] \succ^* \mathtt{raise}\, \varepsilon$. Since terms inhabiting the interpretation have values (Lemma 5.13), in both case we can show that either $\mathtt{try}\, t[\sigma]\, \mathtt{with}\, \varepsilon \mapsto u[\sigma]$ reduces to $u[\sigma]$ or, if $t[\sigma] \in [\![\, A^\Delta \,]\!]_\rho$, it reduces to some t' such that $t[\sigma] \succ^* t'$. In both case, we can conclude. $\qquad \square$

Note that in this model, we only consider closed terms by construction. For this very reason, we cannot establish a strong normalization theorem using this model. But, from the model, we obtain a form a weak head normalization theorem (let us recall that values corresponds to weak head normal form):

Theorem 5.19 (Weak head normalization). *If M is a closed term, A a type and Γ a typing context such that $\Gamma \vdash M : A$, then M has a value.*

Proof. This comes directly from the model soundness theorem and Lemma 5.13. $\qquad \square$

The model allows us to prove for instance that our typing of exceptions is safe for the primitive data types of the natural numbers:

Lemma 5.20 (type safety for natural numbers)**.** *If M is a term such that $\vdash M : \mathbb{N}$, then $M \succ^* S^n\ 0$ for some $n \geq 0$.*

Proof. If $\vdash M : \mathbb{N}$, then with Theorem 5.18, $M \in [\![\mathbb{N}]\!]_\rho$. We conclude using Lemma 5.10 and the fact that M is well typed and there is no typing rule for ✠.   □

Hence, if a program is of the type of the natural numbers, we assure that it will compute a true natural number without producing errors.

6. RELATED WORKS

The static detection of uncaught exceptions has been studied in many works, based on typing or not. For instance, for the OCaml languages, J.C. Guzmn and A. Surez [7] have proposed an extension of the type system where arrows are annotated by the exceptions a function can raise. Later, X. Leroy and F. Pessaux [13] have proposed a similar system but have added polymorphism over these annotations. Their solution is efficient and covers all the Ocaml language, including modularity. However, all these works consider exceptions in call-by-value languages and rely heavily on the exceptions-as-control-flow paradigm.

In call-by-name, it is standard to use monads to encode exceptions [21, 16]. We have however already explained in section 2.1 the drawbacks of such approach. As already stressed, from a computational point of view, the exception mechanism described in this paper is very similar to the imprecise exceptions of S. Peyton Jones *et al.* [14] who are implemented in the ghc Haskell compiler [17]. The novelty of this paper is to provide a precise type system for this exception mechanism while in [14] exceptional values inhabit all types. The "imprecision" of imprecise exceptions comes from the willingness to not force a particular reduction strategy for primitive binary operators. For instance, with imprecise exception the term $(\mathtt{raise}\ \varepsilon) + (\mathtt{raise}\ \varepsilon')$ evaluates to the set $\{\mathtt{raise}\ \varepsilon, \mathtt{raise}\ \varepsilon'\}$ (hence exceptional values are sets). Since in Fx we do not have binary primitive operators, we have no need for such so-called imprecision. However, in Fx, the addition should be coded using the $\mathtt{rec}$ operator, such coding being bound to be non commutative for exceptions (the coding have to choose on which operand of the addition the recursion should be performed). We however believe that if needed, the typing of exceptions presented in this paper could be adapted with almost no changes to the case of imprecise exceptions since our type notions already deal with sets of exceptions.

In the literature, exceptions are often considered as control operators. Note however that exceptions have a dynamic semantic, and as such, cannot be compared to static control operators like first-class continuations [18]. In particular, the typing of exceptions does not necessarily lift the logic to a classical one. Besides, in this paper, we address the problem of the static detection of uncaught exceptions. We do not know of previous works on control operators dealing with this particular problem.

Exceptions in type theoretical settings have been less studied. However, R. David and G. Mounier [3] have designed a typed mechanism of exceptions for the language AF2. However, as with monads, the propagation of exceptions in their system has to be forced by means of Krivine's storage operators. Besides, their exceptions are restricted in the sense that only data types can carry exceptions and for example, exceptions cannot be used as functions.

7. Conclusion and future works

We have presented the Fx calculus, an extension of System F with typed exceptions. We have presented a mechanism of exceptions that does not force a particular β-reduction strategy for the calculus. We have also provided a type system for this mechanism that performs static detection of uncaught exceptions. This type system is modular and allows the use and propagation of exceptions to be transparent for the programmer. Finally, we have justified the semantic of our calculus by exhibiting a realizability model.

This calculus can be improved in a certain number of ways. First, by proving more meta-theoretical properties. Our realizability model only allows to prove weak head normalization but it could probably be modified in order to prove strong normalization. In fact, we believe that the simple change of the definition of the orthogonality relation (definition 5.4) to "$M \perp C$ if and only if $C[M] \succ^* \maltese$ *and* $C[M]$ is strongly normalizing", would yield a strong normalization model (but with this new notion the interpretation will not be closed by anti-reduction anymore and proofs will have to be adapted). Moreover, we have not completed yet the proof of subject-reduction for Fx. However, a detailed proof of subject-reduction for the restriction of the calculus to first-order can be found in [8] (showing that corruption does not break intrinsically the subject-reduction property). Adapting this proof to second order (and thus to Fx) is however not trivial, not because of corruption, but because of the subtyping rules of quantification. Besides, the realizability model already proves a form a type safety for the calculus.

Type inference for Fx is obviously undecidable [25]. But type inference for restrictions of Fx, to first-order for instance, remains to be studied, and we have good hopes since we know that in such a restriction, the subtyping relation is decidable.

Exceptions in Fx are simple names. We would like to extend the calculus so that they carry arguments. However, we will then need to account in the type system for the types of these arguments, which will complicates notably the type system.

As mentioned in the introduction, we think that corruption is a promising notion for the addition of exceptions to proof assistants based on type theoretical calculi. To that end, we think that a natural extension would be to add dependent product to our calculus. As our type system is heavily based on subtyping, we would build on previous works on subtyping in dependent calculus [2, 9]. Moreover, we already know how to extend our realizability model to handle the dependent product: if T is a type and U_x a type family indexed by x, we can take

$$| \Pi x : T . U |_\rho = \bigcup_{\Delta \subseteq \mathcal{E}} \{ M \cdot C \mid M \in [\![T^\Delta]\!]_\rho \wedge C \in | U_M^\Delta |_\rho \}$$

References

[1] H.P. Barendregt. *The lambda calculus.* North-Holland, 1984.
[2] G. Chen. Subtyping calculus of construction, extended abstract. In *The 22nd International Symposium on Mathematical Foundation of Computer Science*, volume 1295. Springer.
[3] R. David and G. Mounier. An intuitionistic λ-calculus with exceptions. *Journal of Functional Programming*, 15(01):33–52, 2004.
[4] The Coq development team. The Coq Proof Assistant Reference Manual v8.1, 2006.
[5] J.Y. Girard. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(03):301–506, 2001.
[6] J.Y. Girard et al. *Proofs and types.* Cambridge University Press New York, 1989.

[7] J. Guzman and A. Suarez. An extended type system for exceptions. *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, pages 127–135, 1994.

[8] Sylvain Lebresne. *Une approche de la détection statique d'exceptions non rattrapées en appel par nom.* PhD thesis, Université Paris Diderot – Paris 7, 2008.

[9] A. Miquel. The implicit calculus of constructions. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications (TLCA 2001)*, volume 2044, pages 344–359, 2001.

[10] J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2-3):211–249, 1988.

[11] E. Moggi. Notions of computation and monads. *INF. COMPUT.*, 93(1):55–92, 1991.

[12] M. Parigot. Strong normalization for second order classical natural deduction. *Logic in Computer Science, 1993. LICS'93., Proceedings of Eighth Annual IEEE Symposium on*, pages 39–46, 1993.

[13] F. Pessaux and X. Leroy. Type-based analysis of uncaught exceptions. *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 276–290, 1999.

[14] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. *ACM SIGPLAN Notices*, 34(5):25–36, 1999.

[15] Randy Pollack. The LEGO Proof Assistant, 1998.

[16] M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, 1990.

[17] The GHC Team. GHC, the `control.exception` module, 2009. `http://www.haskell.org/ghc/docs/latest/html/libraries/base/Control-Exception.html`.

[18] H. Thielecke. Comparing Control Constructs by Double-Barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2):141–160, 2002.

[19] J. Tiuryn and P. Urzyczyn. The subtyping problem for second-order types is undecidable. *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 74–85, 1996.

[20] J. Vouillon and P.A. Melliès. Semantic types: a fresh look at the ideal model for types. *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 52–63, 2004.

[21] P. Wadler. How to Replace Failure by a List of Successes A method for exception handling, backtracking, and pattern matching. *Functional Programming Languages and Computer Architecture*, 1985.

[22] P. Wadler. Comprehending monads. *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, 1990.

[23] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)*, 4(1):1–32, 2003.

[24] J.B. Wells. The undecidability of Mitchells subtyping relation. *Technical Report 95-019, Boston University, Boston, Massachusetts*, 1995.

[25] J.B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.

Appendix A. Parallel reduction for *Fx*

$$\frac{}{M \gg M} \qquad \frac{M \gg M'}{\lambda x.\, M \gg \lambda x.\, M'} \qquad \frac{M \gg M' \quad N \gg N'}{M\ N \gg M'\ N'}$$

$$\frac{M \gg M' \quad N \gg N'}{(\lambda x.\, M)\ N \gg M'\{x := N'\}} \qquad \frac{}{(\mathtt{raise}\,\varepsilon)\ M \gg \mathtt{raise}\,\varepsilon}$$

$$\frac{N \gg N'}{\mathtt{try}\,(\mathtt{raise}\,\varepsilon)\,\mathtt{with}\,\varepsilon \mapsto N \gg N'} \qquad \frac{}{\mathtt{try}\,(\mathtt{raise}\,\varepsilon')\,\mathtt{with}\,\varepsilon \mapsto N \gg \mathtt{raise}\,\varepsilon'}$$

$$\frac{M \gg M' \quad N \gg N'}{\mathtt{try}\,M\,\mathtt{with}\,\varepsilon \mapsto N \gg \mathtt{try}\,M'\,\mathtt{with}\,\varepsilon \mapsto N'}$$

$$\frac{V \gg V' \quad V \text{ is a regular value}}{\text{try } V \text{ with } \varepsilon \mapsto N \gg V'}$$

$$\frac{X \gg X'}{\text{rec } X \ Y \ 0 \gg X'} \qquad \frac{X \gg X' \quad Y \gg Y' \quad N \gg N'}{\text{rec } X \ Y \ (S \ N) \gg Y' \ N' \ (\text{rec } X' \ Y' \ N')}$$

$$\frac{}{\text{rec } X \ Y \ (\text{raise} \, \varepsilon) \gg \text{raise} \, \varepsilon}$$

$$\frac{X \gg X'}{\text{list\_rec } X \ Y \ [\,] \gg X'} \qquad \frac{X \gg X' \quad Y \gg Y' \quad E \gg E' \quad L \gg L'}{\text{list\_rec } X \ Y \ (\text{cons } E \ L) \gg Y' \ E' \ L' \ (\text{list\_rec } X' \ Y' \ L')}$$

$$\frac{}{\text{list\_rec } X \ Y \ (\text{raise} \, \varepsilon) \gg \text{raise} \, \varepsilon}$$