

# A System F with call-by-name exceptions

Sylvain Lebesne

Preuves, Programmes et Systèmes (PPS), CNRS, Université Paris 7, Paris, France,  
Projet Logical, LIX, École Polytechnique, Palaiseau, France

**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  $A \uplus \{\varepsilon\}$  for programs of type  $A$  whose execution may raise the exception  $\varepsilon$  at top level, and a *corruption type*  $A^{\{\varepsilon\}}$  for programs that may raise the exception  $\varepsilon$  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, Objective Caml, C++, . . . . The main computational features of exceptions are:

1. You can raise an exception instead of any other expression (or instruction);
2. It propagates automatically by default;
3. Programmers can catch it only when (s)he needs 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. Adding exceptions to lazy languages is more difficult since an expression is evaluated only when needed and thus, programs do not have a readily-predictable control flow. As noted by S. Peyton Jones *et al.* [9], “(. . .) 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*”. That is why they proposed the idea of exceptions-as-values: a value (of any type) is either a “normal” value, or it is an “exceptional” one.

Exceptions have been less studied in type theoretical settings (in a broad sense). Indeed, exceptions are more a practical facility than a theoretically useful tool. However, with the development of proof assistants, there is a higher demand of users for an exceptions mechanism in these tools.

But adding exceptions to type theoretical frameworks raises new difficulties, at least for two reasons: first, because these languages are independent from any reduction strategy (thus a notion of control-flow has no sense), and second, it is undesirable in such frameworks to give all the possible types to an exception.

The independence towards reduction strategy (see Coq [3] for example) is a consequence of the fact that type theoretical languages are usually “pure” (they do not allow side effects). Hence, the idea of exceptions-as-values seems also well fitted to these languages.

Moreover, in type theoretical languages, exceptions cannot be in all types for consistency reasons. More generally, these languages are expected to capture more properties than usual functional languages. In particular, the exceptions raised by an expression should be reflected in the type of this expression.

In this paper, we propose an extension of System F with exceptions-as-values and a type system that allows static detection of uncaught exceptions using a notion of *corruption*. This notion, by using subtyping, avoids any extra clutter for the programmer and allows for modularity. Here, we use System F as a first step towards more elaborate type theoretical frameworks.

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. Then, the Section 4 provides some examples. We design in Section 5 a realizability model of our calculus that gives some insight on the meaning of corruption. 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?

There are essentially two designs for exceptions as values: either we encode explicitly exceptions in the language, or we make them primitives.

Encoding explicitly exceptions is an old idea [13, 10]: 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 [9]). It has later been realized that the `Maybe` type constructor forms a *monad* [7, 14]. And P. Wadler and P. Thiemann proposed in [15] to add effects to monads, allowing for the detection of uncaught exceptions in such monadic encoding. However, this approach has some drawbacks:

- 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 [9], modularity and code re-use are compromised, especially for higher order functions. Consider a sorting function taking a comparison function (returning a boolean) as argument. Then, the sorting function cannot be applied to a comparison function which raises exceptions;

- Monads force the evaluation of arguments (in the example above, the evaluation of  $\mathbf{x}$  is forced before the application to  $\mathbf{f}$ ). One could not see that as an inconvenience, and this is indeed desirable for most uses of monads, but nonetheless, we think that it 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.* [9] 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). 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.

Our system, named *Fx*, adapts this idea to System F, adding it two new term constructions: **raise** and **try**. But while the exceptions of [9] 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 **raise**  $\varepsilon$  in place of any other term. In particular, **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 catch them all;
- Terms of  $A \uplus \{\varepsilon\}$ . They are terms of  $A$  or terms that reduce to the exception  $\varepsilon$  (they raise it).
- Terms of  $A^{\{\varepsilon\}}$ . They are terms of  $A$  that may mention the exception  $\varepsilon$  but do not necessarily reduce to it (for instance, if  $S$  is the successor function,  $S(\mathbf{raise} \ \varepsilon)$  has type  $\mathbb{N}^{\{\varepsilon\}}$ , but not type  $\mathbb{N} \uplus \{\varepsilon\}$  since it has not type  $\mathbb{N}$  nor does it reduce to **raise**  $\varepsilon$ ).

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

## 2.4 Why we need to distinguish these three levels.

The construction  $A \uplus \{\varepsilon\}$  is really needed because of the typing of the `try` operation, since for a `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  $S (\mathbf{raise} \varepsilon)$ . And secondly, there remains terms that we cannot substitute by an exception: we can write  $(\lambda x. S x) 0$  but not  $(\lambda x. S x) (\mathbf{raise} \varepsilon)$ .

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

$$(A \rightarrow B)^{\{\varepsilon\}} = A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}}$$

This subtyping equality 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 `raise  $\varepsilon$`  (hence, programmers can use `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 `raise  $\varepsilon$`  at top-level, terms of  $A^{\{\varepsilon\}}$  are the ones that may reduce to `raise  $\varepsilon$`  in *any* evaluation context.

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

$$A \rightarrow B \leq (A \rightarrow B) \uplus \{\varepsilon\} \leq (A \rightarrow B)^{\{\varepsilon\}} = A^{\{\varepsilon\}} \rightarrow B^{\{\varepsilon\}}.$$

Remark that since we use subtyping, there is no need to actually know the term  $f$ . This allows for modularity and, in particular, this is convenient for primitive functions like  $S$ , allowing to type-check  $S (\mathbf{raise} \varepsilon)$  with the type  $\mathbb{N}^{\{\varepsilon\}}$ .

## 2.5 Typing the recursion operator

So,  $Fx$  uses primitive natural numbers and hence, provides the usual recursion operator (denoted `rec`). Moreover, a corrupted natural number is an integer where some sub-terms may have been replaced by an exception. Hence, computationally, it is not difficult to write, using `rec`, a function that, given a corrupted integer, return either its argument if it is a well formed natural number, or the corrupting exception otherwise (we will give such a function in Section 4). But to give it the type we expect, *i.e.*  $\mathbb{N}^\Delta \rightarrow \mathbb{N} \uplus \Delta$ , we will have to give to the recursion operator a type more precise than the one it is usually given.

# 3 Formal presentation

## 3.1 Syntax, reductions and associated properties

**Syntax of terms** We consider a countable set  $\mathcal{E}$  of names of exceptions (we can use more than one exception in  $Fx$ ) and a distinguished set of variables  $\mathcal{V}$ . Terms of  $Fx$  are defined by:

$$M, N ::= x \mid \lambda x. M \mid M N \mid \mathbf{raise} \varepsilon \mid \mathbf{try} M \mathbf{with} \varepsilon \mapsto N \mid 0 \mid S \mid \mathbf{rec}$$

In this definition, variables are ranged over by  $x, y, \dots$  while exception names are ranged over by  $\varepsilon, \varepsilon', \dots$ . 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  $\alpha$ -equivalence. Note that the construction  $\mathbf{try} M \mathbf{with} \varepsilon \mapsto N$  does not bind the occurrences of  $\varepsilon$ . The term  $\mathbf{raise} \varepsilon$  is called an *exception*,  $\varepsilon$  being its name, but, as an abuse of terminology, we also call  $\varepsilon$  an exception. In the term  $\mathbf{try} M \mathbf{with} \varepsilon \mapsto N$  we will sometimes call  $M$  the body and  $N$  the handler of the  $\mathbf{try}$  construction.

**Computation in  $Fx$**  Values are the terms of  $Fx$  defined by

$$V ::= \lambda x. M \mid 0 \mid S \mid S N \mid \mathbf{rec} \mid \mathbf{rec} M \mid \mathbf{rec} M N$$

The notion of reduction for the calculus is given by the rules of Figure 1.

$\begin{array}{l} (\lambda x. M) N \succ M\{x := N\} \\ \mathbf{try} (\mathbf{raise} \varepsilon) \mathbf{with} \varepsilon \mapsto N \succ N \\ \mathbf{try} (\mathbf{raise} \varepsilon') \mathbf{with} \varepsilon \mapsto N \succ \mathbf{raise} \varepsilon' \\ \mathbf{try} V \mathbf{with} \varepsilon \mapsto N \succ V \end{array}$	$\begin{array}{l} (\mathbf{raise} \varepsilon) M \succ \mathbf{raise} \varepsilon \\ \mathbf{rec} X Y 0 \succ X \\ \mathbf{rec} X Y (S N) \succ Y N (\mathbf{rec} X Y N) \\ \mathbf{rec} X Y (\mathbf{raise} \varepsilon) \succ \mathbf{raise} \varepsilon \end{array}$
--	---

**Fig. 1.** Notion of reduction for  $Fx$

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 transitive and reflexive closure of  $\succ$ .

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

**Fig. 2.** Relation of reduction for  $Fx$

Note that, as usual, the scope of capture of the  $\mathbf{try}$  construction is dynamic: in the term  $(\lambda x. \mathbf{try} x \mathbf{with} \varepsilon \mapsto 0) (\mathbf{raise} \varepsilon)$ , the exception is caught during reduction and the whole term reduces to 0.

We say that a term  $M$  *raises the exception*  $\varepsilon$  if  $M \succ^* \mathbf{raise} \varepsilon$  (that is, if  $M$  reduces to the exception named  $\varepsilon$ ).

Adding  $\mathbf{raise}$  and  $\mathbf{try}$  does not break the confluence of the calculus:

**Theorem 1 (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 straightforwardly 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  $\gg$ , show that it satisfies the diamond property and conclude since  $\succ^* = \gg^*$ . We give in appendix A the definition of the parallel reduction for  $Fx$ .

### 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 System  $F\eta$  (System F with subtyping [6, 16]). Besides, we can use more than one exception so that type constructions handle sets of exceptions names.

The syntax of types for  $Fx$  is built upon the one of System F:

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

In  $A \uplus \Delta$  and  $A^\Delta$ ,  $\Delta$  is a finite set of exceptions names ( $\Delta \subseteq \mathcal{E}$ ). Moreover,  $\alpha$  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  $\alpha$ -equivalence. Precedences for the arrow construction and the universal quantifier are the usual ones; the precedences of  $A \uplus \Delta$  and  $A^\Delta$  being higher.

**Typing** A *typing context*  $\Gamma$  is a finite set of declarations having the form  $\Gamma \equiv x_1 : A_1, \dots, x_n : A_n$  where  $x_1, \dots, x_n$  are pairwise distinct term variables and where  $A_1, \dots, A_n$  are arbitrary types. The set  $FV(\Gamma)$  denotes the union of the sets of free type variables for the types used in  $\Gamma$ . The type system of  $Fx$  is defined from the *typing judgment*

$$\Gamma \vdash M : A$$

that reads ‘in the typing context  $\Gamma$ , the term  $M$  has type  $A$ ’. This judgment is inductively defined by the rules of Figure 3. Remark that the typing rules for System  $F\eta$  are unchanged, we simply add rules. Also note that the usual typing rule for the recursion operator can be retrieved from  $(rec)$  by taking  $\Delta = \emptyset$  (and the  $(rec)$  rule is in fact a typing scheme).

**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  $A = B$  is defined as short for “ $A \leq B$  and  $A \geq B$ ”, and the inference rules with an equality on conclusion is a notation for the two expected inference rules.

The subtyping rules of  $F\eta$  are unchanged. The rules  $(ex-noexu)$ ,  $(ex-noexc)$ ,  $(eq-uu)$  and  $(eq-cc)$  dealt with sets of exceptions. The hierarchy of corruption

<b>System <math>F\eta</math> typing rules:</b>		
$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \text{ (ax)}$	$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B} \text{ (abs)}$	$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \text{ (app)}$
$\frac{\Gamma \vdash M : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha. A} \text{ (gen)}$	$\frac{\Gamma \vdash M : A \quad A \leq B}{\Gamma \vdash M : B} \text{ (subs)}$	
<b>Natural numbers typing rules:</b>		
$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \text{ (zero)}$	$\frac{}{\Gamma \vdash S : \mathbb{N} \rightarrow \mathbb{N}} \text{ (succ)}$	
$\frac{}{\Gamma \vdash \text{rec} : \forall \alpha. \alpha \multimap \Delta \rightarrow (\mathbb{N}^\Delta \rightarrow \alpha \multimap \Delta \rightarrow \alpha \multimap \Delta) \rightarrow \mathbb{N}^\Delta \rightarrow \alpha \multimap \Delta} \text{ (rec)}$		
<b>Exceptions handling typing rules:</b>		
$\frac{}{\Gamma \vdash \text{raise } \varepsilon : \forall \alpha. \alpha \multimap \{\varepsilon\}} \text{ (raise)}$	$\frac{\Gamma \vdash M : A \multimap \{\varepsilon\} \quad \Gamma \vdash N : A}{\Gamma \vdash \text{try } M \text{ with } \varepsilon \mapsto N : A} \text{ (try)}$	

**Fig. 3.** Typing judgments

<b>System <math>F\eta</math> rules :</b>		
$\frac{}{A \leq A} \text{ (st-id)}$	$\frac{A \leq B \quad B \leq C}{A \leq C} \text{ (st-trans)}$	$\frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'} \text{ (st-arrow)}$
$\frac{A \leq B \quad \alpha \notin FV(A)}{A \leq \forall \alpha. B} \text{ (f-gen)}$	$\frac{}{\forall \alpha. A \leq A\{\alpha := B\}} \text{ (f-inst)}$	
$\frac{\alpha \notin FV(A)}{\forall \alpha. (A \rightarrow B) \leq A \rightarrow \forall \alpha. B} \text{ (f-arr)}$		
<b>Exception related rules :</b>		
$\frac{}{A \multimap \emptyset \leq A} \text{ (ex-noexu)}$	$\frac{}{A^\emptyset \leq A} \text{ (ex-noexc)}$	$\frac{}{(A \rightarrow B) \multimap \Delta \leq A \rightarrow B \multimap \Delta} \text{ (ex-arru)}$
$\frac{A \leq B}{A \multimap \Delta \leq B \multimap \Delta} \text{ (ex-ctx)}$	$\frac{}{A \leq A \multimap \Delta} \text{ (ex-uni)}$	$\frac{}{A \multimap \Delta \leq A^\Delta} \text{ (ex-corrupt)}$
$\frac{}{\forall \alpha. A^\Delta \leq (\forall \alpha. A)^\Delta} \text{ (ex-fallc)}$		$\frac{}{\forall \alpha. (A \multimap \Delta) \leq (\forall \alpha. A) \multimap \Delta} \text{ (ex-fallu)}$
<b>Exception related equality rules :</b>		
$\frac{}{(A \multimap \Delta) \multimap \Delta' = A \multimap (\Delta \cup \Delta')} \text{ (eq-uu)}$		$\frac{}{(A^\Delta)^{\Delta'} = A^{(\Delta \cup \Delta')}} \text{ (eq-cc)}$
$\frac{}{(A \multimap \Delta)^{\Delta'} = A^{\Delta'} \multimap (\Delta - \Delta')} \text{ (eq-uc)}$		$\frac{}{(A \rightarrow B)^\Delta = A^\Delta \rightarrow B^\Delta} \text{ (eq-arrc)}$

**Fig. 4.** The subtyping relation

(see 2.3) is implemented by *(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 can be proved by simultaneous structural inductions on  $A$  and  $B$ ). Rule *(ex-arru)* simply says that, since a term  $M$  of type  $(A \rightarrow B) \uplus \Delta$  is either a term of type  $A \rightarrow B$  or an exception of  $\Delta$ , 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  $\Delta$  (if so is  $M$ ).

Finally, 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.

### 3.3 Properties of typing

We define the relation  $\sqsubseteq_{\Delta}$  between terms by  $M \sqsubseteq_{\Delta} N$  if and only if  $N$  is obtained from  $M$  by replacing some sub-terms in any position by  $\mathbf{raise} \varepsilon$ ,  $\varepsilon$  belonging to  $\Delta$ . Then, Theorem 2 formally states that, in term of programming, exceptions can be used in any place, but with the added cost of corrupting the type.

**Theorem 2 (corruption).** *If  $M$  and  $N$  are two terms,  $A$  a type and  $\Delta$  a set of exceptions such that  $\Gamma \vdash M : A$  and  $M \sqsubseteq_{\Delta} N$ , then  $\Gamma \vdash N : A^{\Delta}$ .*

*Proof.* We prove this theorem 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  $\Gamma$  a typing context such that  $\Gamma \vdash \lambda x. M : A$ , then there exists a set of type variable  $\vec{\alpha} \notin FV(\Gamma)$  and two terms  $B$  and  $C$  such that  $\forall \vec{\alpha}. (B \rightarrow C) \leq A$  and  $\Gamma, x : B \vdash M : C$ .
2. If  $M$  and  $N$  are two terms,  $A$  a type and  $\Gamma$  a typing context such that  $\Gamma \vdash M N : A$ , then there exists a term  $C$  such that  $\Gamma \vdash M : C \rightarrow A$  and  $\Gamma \vdash N : C$ .
3. If  $M$  and  $N$  are two terms,  $A$  a type and  $\Gamma$  a typing context such that  $\Gamma \vdash \mathbf{try} M \mathbf{with} \varepsilon \mapsto N : A$ , then  $\Gamma \vdash M : A \uplus \{\varepsilon\}$  and  $\Gamma \vdash N : A$ .

Proofs of these three results are straightforward induction 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:

$$\mathbf{pred} \equiv \mathbf{rec} (\mathbf{raise} \varepsilon) (\lambda x. \lambda y. x) \quad : \quad \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\}$$

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

$$\mathbf{pred}' \equiv \lambda n. \mathbf{try} (\mathbf{pred} n) \mathbf{with} \varepsilon \mapsto 0 \quad : \quad \mathbb{N} \rightarrow \mathbb{N}$$

As  $Fx$  is an extension of System F, we can define lists using second-order encodings. Let us recall such encodings of `list`:

$$\begin{aligned} \mathbf{list} &\equiv \forall\beta. \forall\alpha. (\alpha \rightarrow (\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha) \\ \mathbf{nil} &\equiv \lambda n. \lambda c. n : \mathbf{list} \\ \mathbf{cons} &\equiv \lambda i. \lambda l. \lambda n. \lambda c. c \ i \ (l \ z \ c) : \forall\beta. \beta \rightarrow \mathbf{list}(\beta) \rightarrow \mathbf{list}(\beta) \end{aligned}$$

where we use the shortcut notation  $\mathbf{list}(A) \equiv \forall\alpha. (\alpha \rightarrow (A \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha)$ .

We can now define `head` and `tail` functions that raise an exception when applied to the empty list. Notice that the code of the `tail` function relies on the same “trick” than the one of the predecessor for natural numbers in their second-order encoding version:

$$\begin{aligned} \mathbf{head} &\equiv \lambda l. l \ (\mathbf{raise} \ \varepsilon) \ (\lambda i. \lambda r. i) : \mathbf{list} \rightarrow \forall\beta. \beta \uplus \{\varepsilon\} \\ \mathbf{tail}' &\equiv \lambda l. \lambda n. \lambda c. (l \ (\lambda x. n) \ (\lambda e. \lambda x. \lambda y. y \ i \ (x \ c))) \ (\lambda x. \lambda y. y) \\ \mathbf{tail} &\equiv \lambda l. l \ (\mathbf{raise} \ \varepsilon) \ (\lambda n. \lambda c. \mathbf{tail}' \ l) : \mathbf{list} \rightarrow \mathbf{list} \uplus \{\varepsilon\} \end{aligned}$$

We now define the mapping of a function to a list of integers:

$$\mathbf{map} \equiv \lambda f. \lambda l. \lambda n. \lambda c. l \ n \ (\lambda i. \lambda r. c \ (f \ i) \ r) : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbf{list}(\mathbb{N}) \rightarrow \mathbf{list}(\mathbb{N})$$

Then we can define a (not very useful) function mapping to a list a function that take the successor of the predecessor of the elements:

$$\mathbf{foo} \equiv \mathbf{map} \ (\lambda e. S \ (\mathbf{pred} \ e)) : \mathbf{list}(\mathbb{N}) \rightarrow \mathbf{list}(\mathbb{N}^{\{\varepsilon\}})$$

Now, if given a list  $l$ ,  $\mathbf{foo} \ l$  has type  $\mathbf{list}(\mathbb{N}^{\{\varepsilon\}})$  and we can still get the head of the list with  $\mathbf{head} \ (\mathbf{foo} \ l)$ . However, this natural number can be corrupted (if the first element of  $l$  is 0) and if we want to check for the corruption, we can use the following function, that ‘uncorrupts’ integers :

$$\mathbf{eval} \equiv \lambda n. (\mathbf{rec} \ (\lambda a. a) \ (\lambda m. \lambda r. \lambda a. r \ (S \ a)) \ n) \ 0 : \mathbb{N}^\Delta \rightarrow \mathbb{N} \uplus \Delta$$

To type this function, we instantiate the type of the recursion operator by the type  $(\mathbb{N} \rightarrow \mathbb{N} \uplus \Delta) \uplus \Delta$ .

## 5 Realizability model

### 5.1 Daimon, weak head reduction and contexts

We add a *daimon*,  $\blackstar$ , similar to the one of [4]. It has no typing rules and computationally behaves like an uncatchable exception.

A (closed) term is in *weak head normal form*, if it is in one of the following forms (where  $V$  is a value):

$$\mathbf{whnf} ::= V \mid \mathbf{raise} \ \varepsilon \mid \blackstar$$

Rules for *weak head reduction* ( $\succ_h$ ) are given in Figure 5. The transitive and reflexive closure of  $\succ_h$  is noted  $\succ_h^*$ .

$$\boxed{
\begin{array}{c}
\frac{M > M'}{M \succ_h M'} \qquad \frac{M \succ_h M'}{M N \succ_h M' N} \\
\frac{M \succ_h M' \quad N \succ_h N'}{\text{try } M \text{ with } \varepsilon \mapsto N \succ_h \text{try } M' \text{ with } \varepsilon \mapsto N'} \qquad \frac{M \succ_h M'}{\text{rec } X Y M \succ_h \text{rec } X Y M'}
\end{array}
}$$

**Fig. 5.** Weak head reduction

Moreover, we say that  $t$  has a *weak head normal form* if there exists a weak head normal form  $t'$  such that  $t \succ t'$ . It should be noted that we can prove that  $t$  has a weak head normal form if and only if there exists a weak head normal form  $t''$  such that  $t \succ_h^* t''$ .

A *context* is a term with a hole (denoted by  $[]$ ) and is defined by:

$$C ::= [] \mid C N \mid \text{try } C \text{ with } \varepsilon \mapsto \blackbox \mid \text{rec } M N C$$

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 the restriction in the handler of **try** to  $\blackbox$ . In fact, we consider contexts up to the following equivalence relation:

$$\text{try}(\text{try}[] \text{ with } \varepsilon_1 \mapsto \blackbox) \text{ with } \varepsilon_2 \mapsto \blackbox \equiv \text{try}(\text{try}[] \text{ with } \varepsilon_2 \mapsto \blackbox) \text{ with } \varepsilon_1 \mapsto \blackbox$$

Then,  $\Delta$  being the set of exception  $\{\varepsilon_0, \dots, \varepsilon_n\}$ , we denote by  $\text{try}[] \text{ with } \Delta \mapsto \blackbox$  the context  $\text{try} \dots \text{try}[] \text{ with } \varepsilon_0 \mapsto \blackbox \dots \text{ with } \varepsilon_n \mapsto \blackbox$ .

Contexts have the following property :

**Lemma 1.** *If  $C$  is a context and  $M$  is a term such that  $C[M]$  has a weak head normal form, then  $M$  has a weak head normal form.*

*Proof.* By case on the form of the context  $C$  and by induction on the length of the reduction of  $C[M]$  to its weak head normal form.

## 5.2 Operations on sets

We define some operations on sets of contexts :

$$\begin{array}{l}
S^\perp = \{ M \mid \forall C \in S, C[M] \succ^* \blackbox \} \\
A \cdot S = \{ C [[] N] \mid C \in S, N \in A \} \quad \left| \quad \downarrow_\Delta S = S \circ \{ \text{try}[] \text{ with } \Delta \mapsto \blackbox \} \right. \\
S \circ T = \{ C[D[]] \mid C \in S, D \in T \} \quad \left| \quad \uparrow_\Delta S = \{ \text{try}[] \text{ with } \Delta \mapsto \blackbox \} \circ S \right.
\end{array}$$

## 5.3 A model for $Fx$

We define a realizability model for  $Fx$  using techniques of orthogonality (see [12] for examples of use of such techniques). We call valuation function any function  $\rho$  from type variables to the powerset of  $\mathcal{C}$  minus the empty set ( $\rho : \mathcal{A} \rightarrow (\mathcal{P}(\mathcal{C}))^+$ ). To each type  $A$  we associate two sets:

$$\begin{array}{ll}
\text{A set of contexts} & |A|_\rho \subseteq \mathcal{C} \\
\text{A set of terms} & \llbracket A \rrbracket_\rho \subseteq \mathcal{T}
\end{array}$$

The set  $\llbracket A \rrbracket_\rho$  is uniformly defined from  $|A|_\rho$  by

$$\llbracket A \rrbracket_\rho = |A|_\rho^\perp = \{ M \mid \forall C \in |A|_\rho, C[M] \succ^* \boxtimes \}.$$

The set  $|A|_\rho$  is defined by induction on  $A$  by:

$$\left. \begin{array}{l} |\alpha|_\rho = \rho(\alpha) \\ |\mathbb{N}|_\rho = \{ \mathbf{rec} \boxtimes (\lambda y. \lambda x. x) [] \} \\ |A \uplus \Delta|_\rho = \downarrow_\Delta |A|_\rho \\ |A^\Delta|_\rho = \uparrow_\Delta |A|_\rho \end{array} \right| \begin{array}{l} |A \rightarrow B|_\rho = \bigcup_{\Delta \subseteq \mathcal{E}} (|A^\Delta|_\rho)^\perp \cdot |B^\Delta|_\rho \\ |\forall \alpha. A|_\rho = \bigcup_{S \subseteq \mathcal{C}^+} |A|_{\rho; \alpha \leftarrow S} \end{array}$$

Note that the interpretation in the model of the construction  $A \uplus \Delta$  and  $A^\Delta$  follows, to some extent, the idea that terms of type  $A \uplus \Delta$  are terms that may raise an exception only at top level, where terms of  $A^\Delta$  are those that may raise an exception in any evaluation context. This is emphasized by the ‘‘opposition’’ of the operations  $\downarrow_\Delta$  and  $\uparrow_\Delta$ .

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

**Lemma 2.** *If  $A$  and  $B$  are two types and  $\rho$  a valuation function, then*

$$\llbracket A \rightarrow B \rrbracket_\rho = \bigcap_{\Delta \subseteq \mathcal{E}} \{ M \mid \forall N \in \llbracket A^\Delta \rrbracket_\rho, M N \in \llbracket B^\Delta \rrbracket_\rho \}.$$

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

We define the interpretation of a typing context  $\Gamma$  by:

$$\llbracket \Gamma \rrbracket_\rho = \{ \sigma \mid \forall (x : A) \in \Gamma, \sigma(x) \in \llbracket A \rrbracket_\rho \}$$

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

**Theorem 3 (Model soundness).** *If  $M$  is a term,  $A$  a type and  $\Gamma$  a typing context such that  $\Gamma \vdash M : A$ , then for all valuation function  $\rho$  and for all substitution  $\sigma \in \llbracket \Gamma \rrbracket_\rho$  we have  $M[\sigma] \in \llbracket A \rrbracket_\rho$ .*

*Proof.* We have seen that arrow types of  $Fx$  is smaller than the realizability one. Thus forming functions requires stronger assumptions and to prove the soundness of the interpretation for the typing rule of abstraction (*abs*), we need to prove the more general soundness result stating that if  $\Gamma \vdash M : A$ , then for all valuation function  $\rho$ , for all set of exceptions name  $\Delta$  and for all substitution  $\sigma \in \llbracket \Gamma^\Delta \rrbracket_\rho$  we have  $M[\sigma] \in \llbracket A^\Delta \rrbracket_\rho$  (where  $\llbracket \Gamma^\Delta \rrbracket_\rho = \{ \sigma \mid \forall (x : A) \in \Gamma, \sigma(x) \in \llbracket A^\Delta \rrbracket_\rho \}$ ).

The proof of this more general theorem proceeds by induction on the derivation of  $\Gamma \vdash M : A$  and to prove the case of the subsumption rule (*subs*), we first show that if  $A \leq B$ , then for all set of exceptions names  $\Delta$ , we have  $\llbracket A^\Delta \rrbracket_\rho \subseteq \llbracket B^\Delta \rrbracket_\rho$ . We prove this last result by induction on the derivation of  $A \leq B$  and no case presents major difficulty.

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 weak head normalization theorem:

**Theorem 4 (Weak head normalization).** *If  $M$  is a closed term,  $A$  a type and  $\Gamma$  a typing context such that  $\Gamma \vdash M : A$ , then  $M$  has a weak head normal form.*

*Proof.* This comes directly from the model soundness theorem and the fact that if  $M \in \llbracket A \rrbracket_\rho$ , then  $M$  has a weak head normal form. Indeed, if  $M \in \llbracket A \rrbracket_\rho$  and if  $C \in |A|_\rho$ , then  $C[M] \succ^* \blackbox$  and hence  $C[M]$  has a weak head normal form. We then conclude with Lemma 1. We just have to be sure that there always exists such a context  $C$ , that is  $|A|_\rho$  is never empty but this is a direct consequence of the definition of the interpretation.

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

**Lemma 3 (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 3,  $M \in \llbracket \mathbb{N} \rrbracket_\rho$ . But we can easily show that  $\llbracket \mathbb{N} \rrbracket_\rho = \{ M \mid M \succ^* S^n 0 \} \cup \{ M \mid M \succ^* S^n \blackbox \}$ .

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

Many works about the static detection of uncaught exceptions have been done, based on typing or not. For instance, for the OCaml languages, J.C. Guzmán and A. Suárez [5] have proposed an extension of the type system where arrows are annotated by which exceptions a function can raise. Later, X. Leroy and F. Pessaux [8] proposed a similar system but add 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 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 [11]. 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 [2] have designed a typed mechanism of exceptions for the language AF2. But their exceptions are restricted in the sense that only data types can carry exceptions and for example, exceptions cannot be used as functions in their system.

## 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  $\beta$ -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.

We believe that our calculus can be extended in the following ways:

- We conjecture, but have not proved yet, the subject-reduction property for  $Fx$ . The difficulty lies in the interaction between subtyping and implicit polymorphism. However, we do have proved that the restriction of  $Fx$  to first-order types have the subject-reduction property, and we believe that in adapting our exceptions to languages with explicit polymorphism (like the dependent product), we will not encounter this problem. Besides, for  $Fx$ , we have shown that our model allows us to state a type safety lemma that makes the proof of the subject-reduction property less urgent.
- Our realizability model only allows to state a weak normalization theorem. To turn it into a strong normalization one, we need to find a suitable notion of saturated sets (that can handle open terms).
- We can extend the calculus to allow exceptions to carry informations.
- A natural extension would be to add dependent product to our calculus, and we have good hopes that such an extension can be done. At least, 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 will take
 
$$| \Pi x : T. U |_\rho = \bigcup_{\Delta \subseteq \varepsilon} \{ M \cdot C \mid M \in \llbracket T^\Delta \rrbracket_\rho \wedge C \in | U_M^\Delta |_\rho \}$$
- Type inference in  $Fx$  is obviously undecidable [17]. However, type inference for the restriction of  $Fx$  to first-order types remains to be studied, and we have good hopes that it is decidable, since we know that in such a restriction, the subtyping relation is decidable.

## References

1. H.P. Barendregt. *The lambda calculus*. North-Holland, 1984.

2. R. David and G. Mounier. An intuitionistic  $\lambda$ -calculus with exceptions. *Journal of Functional Programming*, 15(01):33–52, 2004.
3. The Coq development team. The Coq Proof Assistant Reference Manual v8.1, 2006.
4. 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.
5. 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.
6. J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2-3):211–249, 1988.
7. E. Moggi. Notions of computation and monads. *INF. COMPUT.*, 93(1):55–92, 1991.
8. 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.
9. 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.
10. M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–42, 1990.
11. H. Thielecke. Comparing Control Constructs by Double-Barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2):141–160, 2002.
12. 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.
13. 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.
14. P. Wadler. Comprehending monads. *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78, 1990.
15. P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)*, 4(1):1–32, 2003.
16. J.B. Wells. The undecidability of Mitchell’s subtyping relation. *Technical Report 95-019, Boston University, Boston, Massachusetts*, 1995.
17. 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.

## A Parallel reduction for $Fx$

$$\begin{array}{c}
\frac{}{\overline{M \gg M}} \qquad \frac{M \gg M'}{\lambda x. M \gg \lambda x. M'} \\
\frac{M \gg M' \quad N \gg N'}{M N \gg M' N'} \qquad \frac{M \gg M' \quad N \gg N'}{(\lambda x. M) N \gg M' \{x := N'\}} \\
\frac{N \gg N'}{\text{try (raise } \varepsilon) \text{ with } \varepsilon \mapsto N \gg N'} \qquad \frac{}{\text{try (raise } \varepsilon') \text{ with } \varepsilon \mapsto N \gg \text{raise } \varepsilon'} \\
\frac{M \gg M' \quad N \gg N'}{\text{try } M \text{ with } \varepsilon \mapsto N \gg \text{try } M' \text{ with } \varepsilon \mapsto N'}
\end{array}$$

$$\begin{array}{c}
\frac{V \gg_v V'}{\text{try } V \text{ with } \varepsilon \mapsto N \gg V'} \\
\frac{X \gg X'}{\text{rec } X Y 0 \gg X'} \quad \frac{X \gg X' \quad Y \gg Y' \quad N \gg N'}{\text{rec } X Y (S N) \gg Y' N' (\text{rec } X' Y' N')} \\
\frac{}{0 \gg_v 0} \quad \frac{}{S \gg_v S} \quad \frac{}{\text{rec} \gg_v \text{rec}} \quad \frac{M \gg M'}{\lambda x. M \gg_v \lambda x. M'} \\
\frac{N \gg N'}{S N \gg_v S N'} \quad \frac{M \gg M'}{\text{rec } M \gg_v \text{rec } M'} \quad \frac{M \gg M' \quad N \gg N'}{\text{rec } M N \gg_v \text{rec } M' N'}
\end{array}$$