

# System $F$ with exceptions

Sylvain Lebresne

Logical, INRIA and PPS, Université Paris 7

March 17, 2009

## 1 Introduction

## 2 Presenting $F_X$

- The terms side
- The types side

## 3 A realisability model of $F_X$

## 4 Conclusion and perspectives

# 1 Introduction

## 2 Presenting $F_X$

- The terms side
- The types side

## 3 A realisability model of $F_X$

## 4 Conclusion and perspectives

# Exceptions

Exceptions are a convenient mechanism to handle exceptional cases. The main features we expect of an exception system are:

# Exceptions

Exceptions are a convenient mechanism to handle exceptional cases. The main features we expect of an exception system are:

- An exception can be raised anywhere;

# Exceptions

Exceptions are a convenient mechanism to handle exceptional cases. The main features we expect of an exception system are:

- An exception can be raised anywhere;
- Propagation is automatic by default;

# Exceptions

Exceptions are a convenient mechanism to handle exceptional cases. The main features we expect of an exception system are:

- An exception can be raised anywhere;
- Propagation is automatic by default;
- It can be catch only when needed.

# Exceptions

Exceptions are a convenient mechanism to handle exceptional cases. The main features we expect of an exception system are:

- An exception can be raised anywhere;
- Propagation is automatic by default;
- It can be catch only when needed.

Usually,

# Exceptions

Exceptions are a convenient mechanism to handle exceptional cases. The main features we expect of an exception system are:

- An exception can be raised anywhere;
- Propagation is automatic by default;
- It can be catch only when needed.

Usually,

- Exceptions = modification of the control flow;

# Exceptions

Exceptions are a convenient mechanism to handle exceptional cases. The main features we expect of an exception system are:

- An exception can be raised anywhere;
- Propagation is automatic by default;
- It can be catch only when needed.

Usually,

- Exceptions = modification of the control flow;
- Exceptions have long been confined to call-by-value languages.

# What about not call-by-value languages?

Lazy languages (say Haskell) have no readily readable notion of control flow.

# What about not call-by-value languages?

Lazy languages (say Haskell) have no readily readable notion of control flow.

However, it has been long known that exceptions can be encoded into values:

# What about not call-by-value languages?

Lazy languages (say Haskell) have no readily readable notion of control flow.

However, it has been long known that exceptions can be encoded into values:

```
data ExVal a = OK a
             | Bad
```

# What about not call-by-value languages?

Lazy languages (say Haskell) have no readily readable notion of control flow.

However, it has been long known that exceptions can be encoded into values:

```
data ExVal a = OK a
             | Bad
```

```
f :: Int → ExVal Int
f x = ...
```

# What about not call-by-value languages?

Lazy languages (say Haskell) have no readily readable notion of control flow.

However, it has been long known that exceptions can be encoded into values:

$$\text{data ExVal } a = \text{OK } a$$

$$\quad \quad \quad | \text{Bad}$$

$$f :: \text{Int} \rightarrow \text{ExVal Int}$$

$$f \ x = \dots$$

It was then realized that the `ExVal` type constructor forms a *monads*.

# The Haskell case

Still, this monadic approach has some drawbacks:

# The Haskell case

Still, this monadic approach has some drawbacks:

- Using exceptions requires extra clutter in the code. To apply a function  $f$  of type  $\text{Int} \rightarrow \text{Int}$  to a value  $v$  of type  $\text{ExVal Int}$ , we have to write

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

# The Haskell case

Still, this monadic approach has some drawbacks:

- Using exceptions requires extra clutter in the code. To apply a function  $f$  of type  $\text{Int} \rightarrow \text{Int}$  to a value  $v$  of type  $\text{ExVal Int}$ , we have to write

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

- It forces the evaluation of arguments.

# The Haskell case

Still, this monadic approach has some drawbacks:

- Using exceptions requires extra clutter in the code. To apply a function  $f$  of type  $\text{Int} \rightarrow \text{Int}$  to a value  $v$  of type  $\text{ExVal Int}$ , we have to write
$$\begin{array}{l} \text{do } a \leftarrow v \\ \text{return } (f \ a) \end{array}$$
- It forces the evaluation of arguments.
- Modularity and code re-use for higher order functions is somehow compromised.

# The “imprecise exception” idea

Simon Peyton Jones et al. proposed the idea of imprecise exceptions.

# The “imprecise exception” idea

Simon Peyton Jones et al. proposed the idea of imprecise exceptions.

A value of any type is either a normal value, or an exceptional one. Hence raising an exception is in all types.

# The “imprecise exception” idea

Simon Peyton Jones et al. proposed the idea of imprecise exceptions.

A value of any type is either a normal value, or an exceptional one. Hence raising an exception is in all types.

From the reduction side, it amounts to refusing the rule:

$$M \text{ raise } \succ \text{ raise}$$

# More on these Imprecise exceptions

Exceptions are values and act as so. Thus, in Haskell, they are lazy. Consider the following function:

```
zipWith f []      []      = []  
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys  
zipWith f xs      ys      = raise
```

# More on these Imprecise exceptions

Exceptions are values and act as so. Thus, in Haskell, they are lazy. Consider the following function:

```
zipWith f []      []      = []  
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys  
zipWith f xs      ys      = raise
```

A call to `zipWith` may return:

# More on these Imprecise exceptions

Exceptions are values and act as so. Thus, in Haskell, they are lazy. Consider the following function:

```
zipWith f []      []      = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs      ys      = raise
```

A call to `zipWith` may return:

- An exception value directly (`zipWith (+) [] [1]`);

# More on these Imprecise exceptions

Exceptions are values and act as so. Thus, in Haskell, they are lazy. Consider the following function:

```
zipWith f []      []      = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs      ys      = raise
```

A call to `zipWith` may return:

- An exception value directly (`zipWith (+) [] [1]`);
- A list with an exception value at the end (`zipWith (+) [1] [1,2]`);

# More on these Imprecise exceptions

Exceptions are values and act as so. Thus, in Haskell, they are lazy. Consider the following function:

```
zipWith f []      []      = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f xs      ys      = raise
```

A call to `zipWith` may return:

- An exception value directly (`zipWith (+) [] [1]`);
- A list with an exception value at the end (`zipWith (+) [1] [1,2]`);
- A list fully defined but where some elements are exceptional values (`zipWith (/) [1,2] [1,0]`).

# More on these Imprecise exceptions

Exceptional values can hide inside lazy data structures.

# More on these Imprecise exceptions

Exceptional values can hide inside lazy data structures.

To ensure that a data structure contains no exceptional values one must force evaluation of all the elements of that structure.

# Exceptions and type theory

Languages of type theory (in the broad sense), because they are pure, do not enforce a particular reduction strategy. For this very reason, a notion of control flow has no sense.

# Exceptions and type theory

Languages of type theory (in the broad sense), because they are pure, do not enforce a particular reduction strategy. For this very reason, a notion of control flow has no sense.

We will use the “imprecise exception” idea to add exceptions in type theory.

# Exceptions and type theory

Languages of type theory (in the broad sense), because they are pure, do not enforce a particular reduction strategy. For this very reason, a notion of control flow has no sense.

We will use the “imprecise exception” idea to add exceptions in type theory.

Exceptions can't be in all type (for consistency reason) and we will **type it** precisely (static detection of uncaught exceptions).

# Exceptions and type theory

Languages of type theory (in the broad sense), because they are pure, do not enforce a particular reduction strategy. For this very reason, a notion of control flow has no sense.

We will use the “imprecise exception” idea to add exceptions in type theory.

Exceptions can't be in all type (for consistency reason) and we will **type it** precisely (static detection of uncaught exceptions).

We use here System F as a first step towards more elaborate type theoretical frameworks.

## 1 Introduction

## 2 Presenting $F_X$

- The terms side
- The types side

## 3 A realisability model of $F_X$

## 4 Conclusion and perspectives

# What is $F_X$ ?

An extension of System  $F_\eta$  with typed exceptions and primitive natural numbers.

# What is $F_X$ ?

An extension of System  $F_\eta$  with typed exceptions and primitive natural numbers.

- With raise and try constructions.

# What is $F_X$ ?

An extension of System  $F_\eta$  with typed exceptions and primitive natural numbers.

- With raise and try constructions.
- A subtyping relation, noted  $\leq$ .

# What is $F_\lambda$ ?

An extension of System  $F_\eta$  with typed exceptions and primitive natural numbers.

- With raise and try constructions.
- A subtyping relation, noted  $\leq$ .
- With new type constructions.

# What is $F_X$ ?

An extension of System  $F_\eta$  with typed exceptions and primitive natural numbers.

- With raise and try constructions.
- A subtyping relation, noted  $\leq$ .
- With new type constructions.
- All the typing and subtyping rules of  $F_\eta$  are unchanged. We only add rules.

# What is $F_X$ ?

An extension of System  $F_\eta$  with typed exceptions and primitive natural numbers.

- With raise and try constructions.
- A subtyping relation, noted  $\leq$ .
- With new type constructions.
- All the typing and subtyping rules of  $F_\eta$  are unchanged. We only add rules.
- Justified by a realizability model.

# $F_X$ syntax

$\mathcal{E}$  : a countable set of names of exceptions.

# *F<sub>X</sub>* syntax

$\mathcal{E}$  : a countable set of names of exceptions.

The syntax of *F<sub>X</sub>* is defined by:

$$M, N ::=$$

# *F<sub>X</sub>* syntax

$\mathcal{E}$  : a countable set of names of exceptions.

The syntax of *F<sub>X</sub>* is defined by:

$$M, N ::= x \mid \lambda x. M \mid M N$$

# *F<sub>X</sub>* syntax

$\mathcal{E}$  : a countable set of names of exceptions.

The syntax of *F<sub>X</sub>* is defined by:

$$M, N ::= x \mid \lambda x. M \mid M N \mid \text{raise } \varepsilon \mid \text{try } M \text{ with } \varepsilon \mapsto N$$

where  $\varepsilon \in \mathcal{E}$ .

# *F<sub>X</sub>* syntax

$\mathcal{E}$  : a countable set of names of exceptions.

The syntax of *F<sub>X</sub>* is defined by:

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

where  $\varepsilon \in \mathcal{E}$ .

# *F<sub>X</sub>* syntax

$\mathcal{E}$  : a countable set of names of exceptions.

The syntax of *F<sub>X</sub>* is defined by:

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

where  $\varepsilon \in \mathcal{E}$ .

We also define a notion of value:

$$V ::= \lambda x. M \mid 0 \mid S \mid S N \mid \text{Rec} \mid \text{Rec } M \mid \text{Rec } M N \\ \mid \text{TRec} \mid \text{TRec } M \mid \text{TRec } M N$$

# Reduction in $F_X$

The notion of reduction is defined by the following rules:

# Reduction in $F_\lambda$

The notion of reduction is defined by the following rules:

$$(\lambda x. M) N \quad \rightarrow \quad M\{x := N\}$$

# Reduction in $F_X$

The notion of reduction is defined by the following rules:

$$\begin{array}{ll}
 (\lambda x. M) N & \succ M\{x := N\} \\
 (\text{raise } \varepsilon) M & \succ \text{raise } \varepsilon \\
 \text{try}(\text{raise } \varepsilon) \text{with } \varepsilon \mapsto N & \succ N \\
 \text{try}(\text{raise } \varepsilon') \text{with } \varepsilon \mapsto N & \succ \text{raise } \varepsilon' \\
 \text{try } V \text{with } \varepsilon \mapsto N & \succ V
 \end{array}$$

# Reduction in $F_X$

The notion of reduction is defined by the following rules:

|                                                                              |               |                                   |
|------------------------------------------------------------------------------|---------------|-----------------------------------|
| $(\lambda x. M) N$                                                           | $\rightarrow$ | $M\{x := N\}$                     |
| $(\text{raise } \varepsilon) M$                                              | $\rightarrow$ | $\text{raise } \varepsilon$       |
| $\text{try}(\text{raise } \varepsilon) \text{ with } \varepsilon \mapsto N$  | $\rightarrow$ | $N$                               |
| $\text{try}(\text{raise } \varepsilon') \text{ with } \varepsilon \mapsto N$ | $\rightarrow$ | $\text{raise } \varepsilon'$      |
| $\text{try } V \text{ with } \varepsilon \mapsto N$                          | $\rightarrow$ | $V$                               |
| $\text{Rec } X \ Y \ 0$                                                      | $\rightarrow$ | $X$                               |
| $\text{Rec } X \ Y \ (S \ N)$                                                | $\rightarrow$ | $Y \ N \ (\text{Rec } X \ Y \ N)$ |
| $\text{Rec } X \ Y \ (\text{raise } \varepsilon)$                            | $\rightarrow$ | $\text{raise } \varepsilon$       |

# Reduction in $F_X$

The notion of reduction is defined by the following rules:

|                                                                              |                                               |
|------------------------------------------------------------------------------|-----------------------------------------------|
| $(\lambda x. M) N$                                                           | $\rightarrow M\{x := N\}$                     |
| $(\text{raise } \varepsilon) M$                                              | $\rightarrow \text{raise } \varepsilon$       |
| $\text{try}(\text{raise } \varepsilon) \text{ with } \varepsilon \mapsto N$  | $\rightarrow N$                               |
| $\text{try}(\text{raise } \varepsilon') \text{ with } \varepsilon \mapsto N$ | $\rightarrow \text{raise } \varepsilon'$      |
| $\text{try } V \text{ with } \varepsilon \mapsto N$                          | $\rightarrow V$                               |
| $\text{Rec } X \ Y \ 0$                                                      | $\rightarrow X$                               |
| $\text{Rec } X \ Y \ (S \ N)$                                                | $\rightarrow Y \ N \ (\text{Rec } X \ Y \ N)$ |
| $\text{Rec } X \ Y \ (\text{raise } \varepsilon)$                            | $\rightarrow \text{raise } \varepsilon$       |
| $\text{TRec } X \ Y \ 0$                                                     | $\rightarrow X$                               |
| $\text{TRec } X \ Y \ (S \ N)$                                               | $\rightarrow \text{TRec } (Y \ X) \ Y \ N$    |
| $\text{TRec } X \ Y \ (\text{raise } \varepsilon)$                           | $\rightarrow \text{raise } \varepsilon$       |

# Types of $F_\lambda$

The syntax for the types of  $F_\lambda$  is given by:

$$A, B ::=$$

# Types of *F<sub>x</sub>*

The syntax for the types of *F<sub>x</sub>* is given by:

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

# Types of $F_x$

The syntax for the types of  $F_x$  is given by:

$$A, B ::= \alpha \mid \mathbb{N} \mid A \rightarrow B \mid \forall \alpha. A \\ \mid A \uplus \Delta \quad (\text{terms from } A \text{ or exceptions of } \Delta)$$

Où  $\Delta \subseteq \mathcal{E}$

# Types of $F_x$

The syntax for the types of  $F_x$  is given by:

$$\begin{aligned}
 A, B ::= & \alpha \mid \mathbb{N} \mid A \rightarrow B \mid \forall \alpha. A \\
 & \mid A \uplus \Delta \quad (\text{terms from } A \text{ or exceptions of } \Delta) \\
 & \mid A^\Delta \quad (\text{terms corrupted by exceptions of } \Delta)
 \end{aligned}$$

Où  $\Delta \subseteq \mathcal{E}$



$A \star \Delta$  : terms of type  $A$  or exceptions of names in  $\Delta$ .



$A \uplus \Delta$  : terms of type  $A$  or exceptions of names in  $\Delta$ .

■  $0 : \mathbb{N} \uplus \{\varepsilon\}$



$A \uplus \Delta$  : terms of type  $A$  or exceptions of names in  $\Delta$ .

- $0 : \mathbb{N} \uplus \{\varepsilon\}$
- $\text{raise } \varepsilon : \mathbb{N} \uplus \{\varepsilon\}$



$A \uplus \Delta$  : terms of type  $A$  or exceptions of names in  $\Delta$ .

- $0 : \mathbb{N} \uplus \{\varepsilon\}$
- $\text{raise } \varepsilon : \mathbb{N} \uplus \{\varepsilon\}$

Terms of  $A \uplus \Delta$  are the ones caught by a try.

# Why it's not sufficient

# Why it's not sufficient

- Typing of data types :  $S \text{ (raise } \varepsilon) \text{ (not } \mathbb{N} \uplus \{\varepsilon\})$ ;

# Why it's not sufficient

- Typing of data types :  $S$  (raise  $\varepsilon$ ) (not  $\mathbb{N} \uplus \{\varepsilon\}$ );
- How to apply a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  to a potentially exceptional term?

# Why it's not sufficient

- Typing of data types :  $S$  (raise  $\varepsilon$ ) (not  $\mathbb{N} \uplus \{\varepsilon\}$ );
- How to apply a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  to a potentially exceptional term? A new typing rule for application?

# Why it's not sufficient

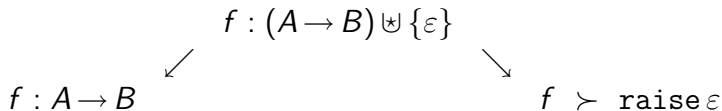
- Typing of data types :  $S$  (raise  $\varepsilon$ ) (not  $\mathbb{N} \uplus \{\varepsilon\}$ );
- How to apply a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  to a potentially exceptional term? A new typing rule for application? Which one?

# Why it's not sufficient

- Typing of data types :  $S \text{ (raise } \varepsilon) \text{ (not } \mathbb{N} \uplus \{\varepsilon\})$ ;
- How to apply a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  to a potentially exceptional term? A new typing rule for application? Which one?
- What to do of a term of type  $(A \rightarrow B) \uplus \Delta$ ?

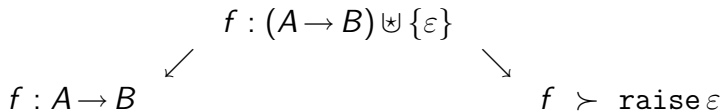
# Why it's not sufficient

- Typing of data types :  $S(\text{raise } \varepsilon)$  (not  $\mathbb{N} \uplus \{\varepsilon\}$ );
- How to apply a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  to a potentially exceptional term? A new typing rule for application? Which one?
- What to do of a term of type  $(A \rightarrow B) \uplus \Delta$ ?



# Why it's not sufficient

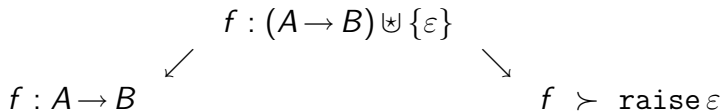
- Typing of data types :  $S(\text{raise } \varepsilon)$  (not  $\mathbb{N} \uplus \{\varepsilon\}$ );
- How to apply a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  to a potentially exceptional term? A new typing rule for application? Which one?
- What to do of a term of type  $(A \rightarrow B) \uplus \Delta$ ?



But  $f$  is not allowed (yet) to be applied to a term of  $A$ .

# Why it's not sufficient

- Typing of data types :  $S(\text{raise } \varepsilon)$  (not  $\mathbb{N} \uplus \{\varepsilon\}$ );
- How to apply a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  to a potentially exceptional term? A new typing rule for application? Which one?
- What to do of a term of type  $(A \rightarrow B) \uplus \Delta$ ?



But  $f$  is not allowed (yet) to be applied to a term of  $A$ .

Type construction  $A \uplus \Delta$  doesn't deal well with arrow types.

# The corruption

A term of  $A^\Delta$  is a term of  $A$  for which some sub-terms could have been replaced by an exception.

# The corruption

A term of  $A^\Delta$  is a term of  $A$  for which some sub-terms could have been replaced by an exception.

Another (equivalent) way to see it,

# The corruption

A term of  $A^{\Delta}$  is a term of  $A$  for which some sub-terms could have been replaced by an exception.

Another (equivalent) way to see it,

- a term of  $A \uplus \Delta$  may raise an exception at top-level.

# The corruption

A term of  $A^\Delta$  is a term of  $A$  for which some sub-terms could have been replaced by an exception.

Another (equivalent) way to see it,

- a term of  $A \uplus \Delta$  may raise an exception at top-level.
- a term of  $A^\Delta$  may raise an exception in some evaluation context (of  $A$ ).

# The corruption

A term of  $A^\Delta$  is a term of  $A$  for which some sub-terms could have been replaced by an exception.

Another (equivalent) way to see it,

- a term of  $A \uplus \Delta$  may raise an exception at top-level.
- a term of  $A^\Delta$  may raise an exception in some evaluation context (of  $A$ ).

Remarks:

# The corruption

A term of  $A^{\Delta}$  is a term of  $A$  for which some sub-terms could have been replaced by an exception.

Another (equivalent) way to see it,

- a term of  $A \uplus \Delta$  may raise an exception at top-level.
- a term of  $A^{\Delta}$  may raise an exception in some evaluation context (of  $A$ ).

Remarks:

- $A \uplus \Delta$  is a broader type than  $A$  ( $A \leq A \uplus \Delta$ ).

# The corruption

A term of  $A^\Delta$  is a term of  $A$  for which some sub-terms could have been replaced by an exception.

Another (equivalent) way to see it,

- a term of  $A \uplus \Delta$  may raise an exception at top-level.
- a term of  $A^\Delta$  may raise an exception in some evaluation context (of  $A$ ).

Remarks:

- $A \uplus \Delta$  is a broader type than  $A$  ( $A \leq A \uplus \Delta$ ).
- $A^\Delta$  is a broader type than  $A \uplus \Delta$  ( $A \uplus \Delta \leq A^\Delta$ ).

# The corruption

A term of  $A^\Delta$  is a term of  $A$  for which some sub-terms could have been replaced by an exception.

Another (equivalent) way to see it,

- a term of  $A \uplus \Delta$  may raise an exception at top-level.
- a term of  $A^\Delta$  may raise an exception in some evaluation context (of  $A$ ).

Remarks:

- $A \uplus \Delta$  is a broader type than  $A$  ( $A \leq A \uplus \Delta$ ).
- $A^\Delta$  is a broader type than  $A \uplus \Delta$  ( $A \uplus \Delta \leq A^\Delta$ ).

We have 3 levels of corruptions.

# Corruption main property

The main property that corruption enjoys is

# Corruption main property

The main property that corruption enjoys is

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

# Corruption main property

The main property that corruption enjoys is

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

with this equality rule we have:

$$S : \mathbb{N} \rightarrow \mathbb{N}$$

# Corruption main property

The main property that corruption enjoys is

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

with this equality rule we have:

$$S : \mathbb{N} \rightarrow \mathbb{N} \leq (\mathbb{N} \rightarrow \mathbb{N}) \uplus \Delta$$

# Corruption main property

The main property that corruption enjoys is

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

with this equality rule we have:

$$S : \mathbb{N} \rightarrow \mathbb{N} \leq (\mathbb{N} \rightarrow \mathbb{N}) \uplus \Delta \leq (\mathbb{N} \rightarrow \mathbb{N})^{\Delta}$$

# Corruption main property

The main property that corruption enjoys is

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

with this equality rule we have:

$$S : \mathbb{N} \rightarrow \mathbb{N} \leq (\mathbb{N} \rightarrow \mathbb{N}) \uplus \Delta \leq (\mathbb{N} \rightarrow \mathbb{N})^{\Delta} = \mathbb{N}^{\Delta} \rightarrow \mathbb{N}^{\Delta}$$

# Corruption main property

The main property that corruption enjoys is

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

with this equality rule we have:

$$S : \mathbb{N} \rightarrow \mathbb{N} \leq (\mathbb{N} \rightarrow \mathbb{N}) \uplus \Delta \leq (\mathbb{N} \rightarrow \mathbb{N})^{\Delta} = \mathbb{N}^{\Delta} \rightarrow \mathbb{N}^{\Delta}$$

and finally:

$$S(\text{raise } \varepsilon) : \mathbb{N}^{\{\varepsilon\}}$$

# Typing rules of $F_\lambda$

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

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B}$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

# Typing rules of $F_x$

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

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B}$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

$$\frac{\Gamma \vdash M : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha. A}$$

# Typing rules of *F<sub>x</sub>*

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

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B}$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

$$\frac{\Gamma \vdash M : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha. A}$$

$$\frac{\Gamma \vdash M : A \quad A \leq B}{\Gamma \vdash M : B}$$

# Typing rules of *F<sub>x</sub>*

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

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x. M : A \rightarrow B}$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$$

$$\frac{\Gamma \vdash M : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash M : \forall \alpha. A}$$

$$\frac{\Gamma \vdash M : A \quad A \leq B}{\Gamma \vdash M : B}$$

$$\overline{\Gamma \vdash 0 : \mathbb{N}}$$

$$\overline{\Gamma \vdash S : \mathbb{N} \rightarrow \mathbb{N}}$$

$$\overline{\Gamma \vdash \text{Rec} : \forall \alpha. \alpha \rightarrow (\mathbb{N} \rightarrow \alpha \rightarrow \alpha) \rightarrow \mathbb{N} \rightarrow \alpha}$$

# Typing rules of *F<sub>X</sub>* continued

$$\overline{\Gamma \vdash \text{raise } \varepsilon : \emptyset \uplus \{\varepsilon\}}$$

$$\emptyset \equiv \forall \alpha. \alpha$$

# Typing rules of *F<sub>X</sub>* continued

$$\overline{\Gamma \vdash \text{raise } \varepsilon : \emptyset \uplus \{\varepsilon\}} \quad \emptyset \equiv \forall \alpha. \alpha$$

$$\frac{\Gamma \vdash M : A \uplus \{\varepsilon\} \quad \Gamma \vdash N : A}{\Gamma \vdash \text{try } M \text{ with } \varepsilon \mapsto N : A}$$

# Typing rules of *F<sub>X</sub>* continued

$$\overline{\Gamma \vdash \text{raise } \varepsilon : \emptyset \uplus \{\varepsilon\}} \quad \emptyset \equiv \forall \alpha. \alpha$$

$$\frac{\Gamma \vdash M : A \uplus \{\varepsilon\} \quad \Gamma \vdash N : A}{\Gamma \vdash \text{try } M \text{ with } \varepsilon \mapsto N : A}$$

$$\overline{\Gamma \vdash \text{TRec} : \forall \alpha. \alpha \rightarrow (\mathbb{N} \rightarrow \alpha) \rightarrow \mathbb{N}^\Delta \rightarrow \alpha \uplus \Delta}$$

# First subtyping rules: $F_\eta$

$$\frac{}{A \leq A}$$

$$\frac{A \leq B \quad B \leq C}{A \leq C}$$

# First subtyping rules: $F_\eta$

$$\frac{}{A \leq A} \qquad \frac{A \leq B \quad B \leq C}{A \leq C}$$

$$\frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'}$$

# First subtyping rules: $F_\eta$

$$\frac{}{A \leq A} \qquad \frac{A \leq B \quad B \leq C}{A \leq C}$$

$$\frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'}$$

$$\frac{A \leq B \quad \alpha \notin FV(A)}{A \leq \forall \alpha. B}$$

$$\frac{}{\forall \alpha. A \leq A\{\alpha := B\}}$$

$$\frac{\alpha \notin FV(A)}{\forall \alpha. (A \rightarrow B) \leq A \rightarrow \forall \alpha. B}$$

# Subtyping rules for exceptions (1/2)

$$\overline{A \uplus \emptyset} \leq A$$

$$\overline{A^\emptyset} \leq A$$

# Subtyping rules for exceptions (1/2)

$$\overline{A \uplus \emptyset} \leq A$$

$$\overline{A^\emptyset} \leq A$$

$$\overline{A \leq A \uplus \Delta}$$

$$\overline{A \uplus \Delta \leq A^\Delta}$$

# Subtyping rules for exceptions (1/2)

$$\overline{A \uplus \emptyset} \leq A$$

$$\overline{A^\emptyset} \leq A$$

$$\overline{A} \leq \overline{A \uplus \Delta}$$

$$\overline{A \uplus \Delta} \leq \overline{A^\Delta}$$

$$\overline{\emptyset^\Delta} \leq \overline{\emptyset \uplus \Delta}$$

# Subtyping rules for exceptions (1/2)

$$\overline{A \uplus \emptyset} \leq A$$

$$\overline{A^\emptyset} \leq A$$

$$\overline{A \leq A \uplus \Delta}$$

$$\overline{A \uplus \Delta \leq A^\Delta}$$

$$\overline{\emptyset^\Delta \leq \emptyset \uplus \Delta}$$

$$\frac{A \leq B}{A \uplus \Delta \leq B \uplus \Delta}$$

# Subtyping rules for exceptions (2/2)

$$\overline{\forall \alpha. (A \uplus \Delta) \leq (\forall \alpha. A) \uplus \Delta}$$

$$\overline{\forall \alpha. A^\Delta \leq (\forall \alpha. A)^\Delta}$$

# Subtyping rules for exceptions (2/2)

$$\overline{\forall \alpha. (A \star \Delta) \leq (\forall \alpha. A) \star \Delta}$$

$$\overline{\forall \alpha. A^\Delta \leq (\forall \alpha. A)^\Delta}$$

$$\overline{(A \star \Delta) \star \Delta' = A \star (\Delta \cup \Delta')}$$

$$\overline{(A^\Delta)^{\Delta'} = A^{(\Delta \cup \Delta')}}$$

# Subtyping rules for exceptions (2/2)

$$\overline{\forall \alpha. (A \star \Delta) \leq (\forall \alpha. A) \star \Delta}$$

$$\overline{\forall \alpha. A^\Delta \leq (\forall \alpha. A)^\Delta}$$

$$\overline{(A \star \Delta) \star \Delta' = A \star (\Delta \cup \Delta')}$$

$$\overline{(A^\Delta)^{\Delta'} = A^{(\Delta \cup \Delta')}}}$$

$$\overline{(A \star \Delta)^{\Delta'} = A^{\Delta'} \star \Delta}$$

# Subtyping rules for exceptions (2/2)

$$\overline{\forall \alpha. (A \uplus \Delta) \leq (\forall \alpha. A) \uplus \Delta}$$

$$\overline{\forall \alpha. A^\Delta \leq (\forall \alpha. A)^\Delta}$$

$$\overline{(A \uplus \Delta) \uplus \Delta' = A \uplus (\Delta \cup \Delta')}$$

$$\overline{(A^\Delta)^{\Delta'} = A^{(\Delta \cup \Delta')}}}$$

$$\overline{(A \uplus \Delta)^{\Delta'} = A^{\Delta'} \uplus \Delta}$$

$$\overline{(A \rightarrow B)^\Delta = A^\Delta \rightarrow B^\Delta}$$

# Examples

$$\text{pred} \equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) \quad :$$

# Examples

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

# Examples

$$\begin{aligned}\text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) & : & \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 & : & \mathbb{N} \rightarrow \mathbb{N}\end{aligned}$$

# Examples

$$\begin{aligned}\text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) & : & \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 & : & \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n)\end{aligned}$$

# Examples

$$\begin{aligned}\text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) &: \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 &: \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n)\end{aligned}$$

We can define second order list and then:

# Examples

$$\begin{aligned} \text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) & : & \quad \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 & : & \quad \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n) \end{aligned}$$

We can define second order list and then:

`map (λn. add2 (pred n)) [3; 0; 4]`

# Examples

$$\begin{aligned} \text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) & : & \quad \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 & : & \quad \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n) \end{aligned}$$

We can define second order list and then:

$$\text{map } (\lambda n. \text{add2 } (\text{pred } n)) [3; 0; 4] \succ [4; S (S (\text{raise } \varepsilon)); 5]$$

# Examples

$$\begin{aligned} \text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) &: \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 &: \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n) \end{aligned}$$

We can define second order list and then:

$$\text{map } (\lambda n. \text{add2 } (\text{pred } n)) [3; 0; 4] \succ [4; S (S (\text{raise } \varepsilon)); 5] : \text{list}(\mathbb{N}^{\{\varepsilon\}})$$

# Examples

$$\begin{aligned} \text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) & : & \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 & : & \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n) \end{aligned}$$

We can define second order list and then:

$$\text{map } (\lambda n. \text{add2 } (\text{pred } n)) [3; 0; 4] \succ [4; S (S (\text{raise } \varepsilon)); 5] : \text{list}(\mathbb{N}^{\{\varepsilon\}})$$

$$\text{head } [4; S (S (\text{raise } \varepsilon)); 5]$$

# Examples

$$\begin{aligned} \text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) & : & \quad \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 & : & \quad \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n) \end{aligned}$$

We can define second order list and then:

$$\text{map } (\lambda n. \text{add2 } (\text{pred } n)) [3; 0; 4] \succ [4; S (S (\text{raise } \varepsilon)); 5] : \text{list}(\mathbb{N}^{\{\varepsilon\}})$$

$$\text{head } [4; S (S (\text{raise } \varepsilon)); 5] \succ 4$$

# Examples

$$\begin{aligned} \text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) & : & \quad \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 & : & \quad \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n) \end{aligned}$$

We can define second order list and then:

$$\text{map } (\lambda n. \text{add2 } (\text{pred } n)) [3; 0; 4] \succ [4; S (S (\text{raise } \varepsilon)); 5] : \text{list}(\mathbb{N}^{\{\varepsilon\}})$$

$$\text{head } [4; S (S (\text{raise } \varepsilon)); 5] \succ 4 : \mathbb{N}^{\{\varepsilon\}}$$

# Examples

$$\begin{aligned} \text{pred} &\equiv \text{Rec } (\text{raise } \varepsilon) (\lambda x. \lambda y. x) & : & \quad \mathbb{N} \rightarrow \mathbb{N} \uplus \{\varepsilon\} \\ \text{pred}' &\equiv \lambda n. \text{try } (\text{pred } n) \text{ with } \varepsilon \mapsto 0 & : & \quad \mathbb{N} \rightarrow \mathbb{N} \\ \text{add2} &\equiv \lambda n. S (S \ n) \end{aligned}$$

We can define second order list and then:

$$\text{map } (\lambda n. \text{add2 } (\text{pred } n)) [3; 0; 4] \succ [4; S (S (\text{raise } \varepsilon)); 5] : \text{list}(\mathbb{N}^{\{\varepsilon\}})$$

$$\text{head } [4; S (S (\text{raise } \varepsilon)); 5] \succ 4 : \mathbb{N}^{\{\varepsilon\}}$$

$$\text{try } (\text{TRec } 0 \ S \ (\text{head } [4; S (S (\text{raise } \varepsilon)); 5])) \text{ with } \varepsilon \mapsto 0 : \mathbb{N}$$

## 1 Introduction

## 2 Presenting $F_X$

- The terms side
- The types side

## 3 A realisability model of $F_X$

## 4 Conclusion and perspectives

# A realizability model

We have developed a realizability model of  $F_x$  designed using orthogonality techniques.

# A realizability model

We have developed a realizability model of  $F_\lambda$  designed using orthogonality techniques.

- we define interpretation of types as sets of contexts;

# A realizability model

We have developed a realizability model of  $F_\lambda$  designed using orthogonality techniques.

- we define interpretation of types as sets of contexts;
- interpretation as sets of terms is then defined by orthogonality.

# A realizability model

We have developed a realizability model of  $F_x$  designed using orthogonality techniques.

- we define interpretation of types as sets of contexts;
- interpretation as sets of terms is then defined by orthogonality.

Interpretation of arrow types is non-standard.

# A realizability model

We have developed a realizability model of *Fx* designed using orthogonality techniques.

- we define interpretation of types as sets of contexts;
- interpretation as sets of terms is then defined by orthogonality.

Interpretation of arrow types is non-standard.

$$A \rightarrow B \quad = \quad \bigcap_{\Delta \subseteq \mathcal{E}} (A^\Delta \xrightarrow{r} B^\Delta)$$

# A realizability model

We have developed a realizability model of *Fx* designed using orthogonality techniques.

- we define interpretation of types as sets of contexts;
- interpretation as sets of terms is then defined by orthogonality.

Interpretation of arrow types is non-standard.

$$A \rightarrow B = \bigcap_{\Delta \subseteq \mathcal{E}} (A^\Delta \xrightarrow{r} B^\Delta)$$

Interpretations of  $A \uplus \Delta$  and  $A^\Delta$  follow the idea that

- $A \uplus \Delta$  may raise exceptions at top-level.
- $A^\Delta$  may raise exceptions in some evaluation context.

# Technicalities

We add a daimon (denoted by  $\blacklozenge$ ). It is an uncatchable exception.

# Technicalities

We add a daimon (denoted by  $\bot$ ). It is an uncatchable exception.

The orthogonality relation is defined by:

$$S^{\perp} = \{ M \mid \forall C \in S, C[M] \multimap^* \bot \}$$

# Technicalities

We add a daimon (denoted by  $\bot$ ). It is an uncatchable exception.

The orthogonality relation is defined by:

$$S^\perp = \{ M \mid \forall C \in S, C[M] \multimap^* \bot \}$$

Contexts are:

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

# Technicalities

We add a daimon (denoted by  $\boxtimes$ ). It is an uncatchable exception.

The orthogonality relation is defined by:

$$S^\perp = \{ M \mid \forall C \in S, C[M] \succ^* \boxtimes \}$$

Contexts are:

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

and we use the following operations:

$$A \cdot S = \{ C[[\ ] \ M] \mid C \in S, M \in A \}$$

$$S \circ T = \{ C[D[\ ]] \mid C \in S, D \in T \}$$

# Model definition

$$\begin{aligned}
 |\alpha|_\rho &= \rho(\alpha) \\
 |\mathbb{N}|_\rho &= \{ \text{Rec } \boxtimes (\lambda y. \lambda x. x) [] \} \\
 |\forall \alpha. A|_\rho &= \bigcup_{S \subseteq \mathcal{C}^+} |A|_{\rho; \alpha \leftarrow S}
 \end{aligned}$$

# Model definition

$$\begin{aligned}
 |\alpha|_\rho &= \rho(\alpha) \\
 |\mathbb{N}|_\rho &= \{ \text{Rec } \blacklozenge (\lambda y. \lambda x. x) [] \} \\
 |\forall \alpha. A|_\rho &= \bigcup_{S \subseteq \mathcal{C}^+} |A|_{\rho; \alpha \leftarrow S} \\
 |A \uplus \Delta|_\rho &= |A|_\rho \circ \{ \text{try} [] \text{ with } \Delta \mapsto \blacklozenge \} \\
 |A^\Delta|_\rho &= \{ \text{try} [] \text{ with } \Delta \mapsto \blacklozenge \} \circ |A|_\rho
 \end{aligned}$$

# Model definition

$$\begin{aligned}
 |\alpha|_\rho &= \rho(\alpha) \\
 |\mathbb{N}|_\rho &= \{ \text{Rec } \blacklozenge (\lambda y. \lambda x. x) [] \} \\
 |\forall \alpha. A|_\rho &= \bigcup_{S \subseteq \mathcal{C}^+} |A|_{\rho; \alpha \leftarrow S} \\
 |A \uplus \Delta|_\rho &= |A|_\rho \circ \{ \text{try}[] \text{ with } \Delta \mapsto \blacklozenge \} \\
 |A^\Delta|_\rho &= \{ \text{try}[] \text{ with } \Delta \mapsto \blacklozenge \} \circ |A|_\rho \\
 |A \rightarrow B|_\rho &= \bigcup_{\Delta \subseteq \mathcal{E}} (|A^\Delta|_\rho)^\perp \cdot |B^\Delta|_\rho
 \end{aligned}$$

# Model definition

$$\begin{aligned}
 |\alpha|_\rho &= \rho(\alpha) \\
 |\mathbb{N}|_\rho &= \{ \text{Rec } \boxtimes (\lambda y. \lambda x. x) [] \} \\
 |\forall \alpha. A|_\rho &= \bigcup_{S \subseteq \mathcal{C}^+} |A|_{\rho; \alpha \leftarrow S} \\
 |A \boxplus \Delta|_\rho &= |A|_\rho \circ \{ \text{try}[] \text{ with } \Delta \mapsto \boxtimes \} \\
 |A^\Delta|_\rho &= \{ \text{try}[] \text{ with } \Delta \mapsto \boxtimes \} \circ |A|_\rho \\
 |A \rightarrow B|_\rho &= \bigcup_{\Delta \subseteq \mathcal{E}} (|A^\Delta|_\rho)^\perp \cdot |B^\Delta|_\rho
 \end{aligned}$$

We then define the interpretation of a type  $A$ :

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

# Model properties

## Theorem (Model soundness)

*The model is sound.*

# Model properties

## Theorem (Model soundness)

*The model is sound.*

## Corollary (Type safety)

*If  $M$  is a term such that  $\vdash M : \mathbb{N}$ , then  $M \rightarrow^* S^n 0$ .*

# Model properties

## Theorem (Model soundness)

*The model is sound.*

## Corollary (Type safety)

*If  $M$  is a term such that  $\vdash M : \mathbb{N}$ , then  $M \succ^* S^n 0$ .*

## Corollary (Weak normalization)

*Terms of  $F_x$  are weakly normalizing.*

## 1 Introduction

## 2 Presenting $F_X$

- The terms side
- The types side

## 3 A realisability model of $F_X$

## 4 Conclusion and perspectives

# Conclusion

A mechanism of exceptions adapted to call-by-name evaluation with type based detection of uncaught exceptions and which respect modularity.

# Conclusion

A mechanism of exceptions adapted to call-by-name evaluation with type based detection of uncaught exceptions and which respect modularity.

What's next?

# Conclusion

A mechanism of exceptions adapted to call-by-name evaluation with type based detection of uncaught exceptions and which respect modularity.

What's next?

- Subject-reduction;

# Conclusion

A mechanism of exceptions adapted to call-by-name evaluation with type based detection of uncaught exceptions and which respect modularity.

What's next?

- Subject-reduction;
- Add more data-types;

# Conclusion

A mechanism of exceptions adapted to call-by-name evaluation with type based detection of uncaught exceptions and which respect modularity.

What's next?

- Subject-reduction;
- Add more data-types;
- Allow exceptions to carry arguments;

# Conclusion

A mechanism of exceptions adapted to call-by-name evaluation with type based detection of uncaught exceptions and which respect modularity.

What's next?

- Subject-reduction;
- Add more data-types;
- Allow exceptions to carry arguments;
- Adapt this mechanism to dependent types (PTS for example).