A Simple Calculus of Exception Handling

Philippe de Groote

INRIA-Lorraine – CRIN – CNRS
615, rue du Jardin Botanique - B.P. 101
54602 Villers lès Nancy Cedex – FRANCE
e-mail: degroote@loria.fr

Abstract. We introduce a simply-typed λ-calculus ($\boldsymbol{\lambda}^{\rightarrow}_{exn}$) featuring an ML-like exception handling mechanism. This calculus, whose type system corresponds to classical logic through the Curry-Howard isomorphism, satifies several interesting properties: among other, Church-Rosser, subject reduction, and strong-normalisation. Moreover, its typing system ensures that the reduction of well-typed expressions cannot give rise to uncaught exceptions.

1 Introduction

During the last four years, several authors have introduced various calculi that extend the Curry-Howard isomorphism to classical logic and therefore provide a computational interpretation of classical proofs [3, 4, 10, 12, 14, 16, 21]. In this paper we propose yet another such calculus that we call $\boldsymbol{\lambda}^{\rightarrow}_{exn}$.

The originality of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ derives from its basis on an exception handling mechanism à la ML. Moreover, from a programming point of view, $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ satifies the interesting property that well-typed programs cannot give rise to uncaught exceptions. This result is achieved by introducing a conservative extention of ML-like operational semantics.

The paper is organised as follows.

Section 2 performs an informal type-theoretic analysis of ML-like exception handling. This analysis results in an interpretation of the type of exceptions as the absurdity type, and yields a typing system that corresponds to classical logic.

In Section 3, we define formally the syntax of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ and its typing relation.

The ML-like operational semantics of exceptions does not completely match the typing system of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$. We discuss this issue, which is related to the subject reduction property, in Section 4.

In Section 5, we provide $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ with a modified operational semantics. This modified semantics is such that $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ satisfies the Church-Rosser and subject reduction property. Consequently, when evaluating a well-typed program, any raised exception is eventually handled.

In Section 6, we investigate the relation between the ML-like and the modified semantics. We prove that, for non-exceptional values, the modified semantics is equivalent to the ML-like semantics.

Section 7 introduces a CPS-interpretation of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ and shows that there exists a logical embedding of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ into the simply-typed λ-calculus.

In Section 8, we establish the strong normalisation of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$.

We discuss related work and conclusions in Section 9.

2 Informal Analysis of ML-like Exception handling

The notion of exception and the one of data type constructor in Standard ML are unified. This unification, which follows a proposal by MacQueen [1], is based on the special datatype *exn* whose values are exceptions. Indeed the following standard ML declaration:

In:
M. Dezani-Ciancaglini and G. Plotkin (Eds)
Proceedings of the Second International Conference on Typed Lambda Calculi and Applications,
Lecture Notes in Computer Science, Vol. 902, Springer-Verlag (1995), pp. 201-215.

$$\texttt{exception}\ \mathit{foo}\ \texttt{of}\ \mathit{int};$$

amounts to the declaration of the constructor *foo* whose type is $\mathit{int} \to \mathit{exn}$.

Values of type *exn* are first-class citizens, they may be stored, be passed as parameters, returned as results, etc. In addition, and in contrast to other values, they may also be turned into *packets* by being *raised* (see [13] for details).

The typing and reduction rules of the operator `raise` are the following:

$$\frac{\Gamma \vdash M : \mathit{exn}}{\Gamma \vdash (\texttt{raise}\ M) : \alpha}$$

$$\begin{array}{rcll} V\,(\texttt{raise}\ M) & \to & (\texttt{raise}\ M) & \text{(for } V \text{ a value)} \\ (\texttt{raise}\ M)\,N & \to & (\texttt{raise}\ M) & \text{(for } N \text{ any expression)} \end{array}$$

These rules correspond respectively to the deduction rule and the proof reduction rules that are used in natural deduction for *falsity* [9].

$$\frac{\bot}{\alpha} \qquad \frac{\overset{\vdots}{\alpha \to \beta} \quad \dfrac{\overset{\vdots}{\bot}}{\alpha}}{\beta} \;\to\; \frac{\overset{\vdots}{\bot}}{\beta} \qquad \frac{\dfrac{\overset{\vdots}{\bot}}{\alpha \to \beta} \quad \overset{\vdots}{\alpha}}{\beta} \;\to\; \frac{\overset{\vdots}{\bot}}{\beta}$$

Therefore it makes sense to identify, through the Curry-Howard isomorphism, the type of exceptions (exn) with the logical notion of falsity ($\bot$). Let us accept this identification and proceed further with the type-theoretic analysis of exception handling.

Packets, i.e. raised exceptions, are propagated and then possibly handled. The typing rule for exception handlers is akin to the following:

$$\frac{\Gamma \vdash M : \alpha \qquad \Gamma \vdash N : \mathit{exn} \to \alpha}{\Gamma \vdash M\ \texttt{handle}\ N : \alpha}$$

This rule is certainly sound, but not satisfactory. On the one hand, we would like to have a rule that allows exception declarations to be discarded. This is mandatory if we want to preserve the logical consistency of the type system because of the identifification of the type of exception with false. On the other hand, as it is stated, this rule does not properly reflect the standard ML exception handling mechanism. Indeed in standard ML the right hand side of the operator `handle` is not an expression but a *match*.

A solution to these two issues is to use the `let` construct to declare exception constructors locally and to consider the following typing rule:

$$\frac{\Gamma, y : \alpha \to \mathit{exn} \vdash M : \beta \qquad \Gamma, x : \alpha \vdash N : \beta}{\Gamma \vdash \texttt{let exception}\ y\ \texttt{of}\ \alpha\ \texttt{in}\ M\ \texttt{handle}\ (y\,x) \Rightarrow N\ \texttt{end} : \beta}$$

This rule, which is consistent with the definition of standard ML, corresponds to the elimination of the disjunction for the particular case of the excluded middle. Therefore it is sound with respect to classical logic. This is not too surprising because it is known, since Griffin's work [12], that there is a strong connection between classical logic and sequential control.

The fact that the above rule is classical allows us to write *classical programs* in the sense of [12]. This is illustrated by the following example.

Example 2.1 In classical logic, conjunction can be defined in terms of implication and negation as follows: $\alpha \wedge \beta = \neg(\alpha \to \neg\beta)$. This definition allows a *classical* pairing operator, together with the associated projections, to be defined:

```
fun pair (x:int) (y:int) = fn (f:(int->int->exn)) => (f x y);

fun proj1 (p:(int -> int -> exn) -> exn) =
              let exception y of int
              in
              (raise (p (fn x => (raise y x))))
              handle
              (y x) => x
              end;

fun proj2 (p:(int -> int -> exn) -> exn) =
              let exception y of int
              in
              (raise (p (fn x => y)))
              handle
              (y x) => x
              end;
```

The types that a compiler infers for these three pieces of code are the following:

```
val pair = fn : int -> int -> (int -> int -> exn) -> exn
val proj1 = fn : ((int -> int -> exn) -> exn) -> int
val proj2 = fn : ((int -> int -> exn) -> exn) -> int
```

Finally, the execution of the programs yields the expected results:

```
- proj1 (pair 1 2);
val it = 1 : int
- proj2  (pair 1 2);
val it = 2 : int
```

3 Definition of a Simple Calculus of Exception Handling

We define a simple calculus ($\boldsymbol{\lambda}^{\rightarrow}_{exn}$) of exception handling based on the ideas discussed in the previous section.

Definition 3.1 *The* types *of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ are given by the following grammar:*

$$\tau \ ::= \ a \mid \bot \mid (\tau \rightarrow \tau)$$

where "a" range over a finite set of ground types, and $\bot$ is a distinguished ground type.

We let the lowercase Greek letters $(\alpha, \beta, \gamma, \ldots)$ range over types, and we write $\neg\alpha$ as an abbreviation for $(\alpha \rightarrow \bot)$.

Definition 3.2 *The* expressions *of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ are built upon two distinct alphabets of variables: the λ-variables and the exception variables. The raw syntax of the expressions is the following.*

$$\begin{array}{rcl} T & ::= & c \mid x \mid y \mid \lambda x.T \mid (T\,T) \mid \\ & & (\texttt{raise}\ T) \mid \texttt{let}\ y : \neg\alpha\ \texttt{in}\ T\ \texttt{handle}\ (y\,x) \Rightarrow T\ \texttt{end} \end{array}$$

where c ranges over possible constants, x ranges over λ-variables, and y over exception variables.

We use uppercase Roman letters $(A, B, C, \ldots)$ to denote expressions, and we adopt the usual notational conventions [5, pp. 22–23]. The notions of free and bound (occurrences of) variables are defined as usual. In particular, the scoping rule for the `handle` construct is the following: in an expression of the form

$$\texttt{let}\ y : \neg\alpha\ \texttt{in}\ A\ \texttt{handle}\ (y\,x) \Rightarrow B\ \texttt{end},$$

the exception variable y is bound in the subexpression A, and the λ-variable x is bound in the subexpression B. We also assume that some implicit convention (e.g. [5, p. 26]) prevents clashes between free and bound variables, and we let $A[x{:=}B]$ denote the usual capture-avoiding substitution.

When writing proofs, we use a more compact notation for the `raise` and the `handle` constructs. We write

$$\mathcal{R}\,(A) \quad \text{for} \quad (\texttt{raise}\ A).$$

Similarly, we write

$$\langle\, x.\, A \mid y.\, B \,\rangle \quad \text{for} \quad \texttt{let}\ x : \neg\alpha\ \texttt{in}\ A\ \texttt{handle}\ (x\,y) \Rightarrow B\ \texttt{end},$$

α being left implicit. We also write

$$\langle\, (x_i).\, A \mid (y_i.\, B_i) \,\rangle \quad \text{for} \quad \langle\, x_1.\, \langle\, x_2.\, \cdots \langle\, x_n.\, A \mid y_n.\, B_n \,\rangle \cdots \mid y_2.\, B_2 \,\rangle \mid y_1.\, B_1 \,\rangle,$$

the number n of nested constructs being left implicit.

$\boldsymbol{\lambda}^{\rightarrow}_{exn}$ is a call-by-value language. We thus have to define the notion of value.

Definition 3.3 *The notion of* value *is defined as follows:*

$$V \ ::= \ c \mid x \mid y \mid \lambda x.\, T \mid (y\, V)$$

Note that there is a significant difference between λ-variables and exception variables. The latter act as datatype constructors. Therefore, the application of an exception variable to a value is a value. From now on, the uppercase Roman letter V (with possible subscripts) will range over values.

Finally the typing rules of our calculus are the ones of the simply typed λ-calculus together with the rules discussed in the previous section for the `raise` and the `handle` constructs.

Definition 3.4 *Define a* typing environment *to be a function, undefined almost everywhere, that assigns types to λ-variables and that assigns types of the form $\neg\alpha$ to exception variables. Let $\Gamma, \Delta, \ldots$ range over typing environments, and let σ be a given function that assigns a type to each constant.*

The typing rules *of the calculus are the following:*

$$\Gamma \vdash c : \sigma(c)$$

$$\Gamma \vdash x : \Gamma(x) \qquad\qquad \Gamma \vdash y : \Gamma(y)$$

$$\frac{\Gamma,\, x : \alpha \vdash B : \beta}{\Gamma \vdash \lambda x.\, B : \alpha \to \beta} \qquad\qquad \frac{\Gamma \vdash A : \alpha \to \beta \qquad \Gamma \vdash B : \alpha}{\Gamma \vdash A\, B : \beta}$$

$$\frac{\Gamma \vdash A : \bot}{\Gamma \vdash (\texttt{raise}\ A) : \alpha} \qquad \frac{\Gamma,\, y : \neg\alpha \vdash A : \beta \qquad \Gamma,\, x : \alpha \vdash B : \beta}{\Gamma \vdash \texttt{let}\ y : \neg\alpha\ \texttt{in}\ A\ \texttt{handle}\ (y\,x) \Rightarrow B\ \texttt{end} : \beta}$$

As we noted, the typing rules of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$, as a logical system, are consistent with respect to classical logic. It is quite easy to show that the system is also complete. We thus have the following proposition.

Proposition 3.5 *Consider the calculus $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ without constants. Given a type α, there exists an expression A such that $\vdash A : \alpha$ if and only if α, seen as a proposition, is a classical tautology.* $\square$

4 Consistency Problems with ML-like Operational Semantics

Let the function σ that assigns types to constants be consistent, in the sense that its range is a consistent set of propositions. Then Proposition 3.5 ensures that closed expressions of type $\bot$ cannot exist. This property, in turn, ensures that the execution of a program cannot give rise to an uncaught exception, provided that the calculus satisfies the subject reduction property.

Unfortunately this is not the case with an ML-like semantics. Indeed, consider the following example:

Example 4.1 The following piece of code defines another *classical* pairing operator:

```
fun var_pair x y = let exception P of (int -> int -> exn)
                   in P handle (P g) => raise (g x y)
                   end;
```

It is easy to check that the expression

```
proj_1 (var_pair 1 2)
```

is a well-typed expression of type *int*. The execution of this expression, however, gives rise to an uncaught exception.

The dynamic semantics of standard ML is given in a natural semantics style [13]. Table 1 adapts this semantics to $\boldsymbol{\lambda}^{\to}_{exn}$, in terms of reduction rules. The problem with subject

$(\lambda x.\, M)\, V \;\to\; M[x{:=}V]$	(β_V)
$V_1\, (\texttt{raise}\; V) \;\to\; (\texttt{raise}\; V)$	$(\texttt{raise}_{\,left})$
$(\texttt{raise}\; V)\, M \;\to\; (\texttt{raise}\; V)$	$(\texttt{raise}_{\,right})$
$(\texttt{raise}\,(\texttt{raise}\; V)) \;\to\; (\texttt{raise}\; V)$	$(\texttt{raise}_{\,idem})$
$\texttt{let}\; y : \neg\alpha\; \texttt{in}\; V\; \texttt{handle}\; (y\, x) \Rightarrow N\; \texttt{end} \;\to\; V$	$(\texttt{handle}_{\,simp})$
$\texttt{let}\; y : \neg\alpha\; \texttt{in}\; (\texttt{raise}\; y\, V)\; \texttt{handle}\; (y\, x) \Rightarrow N\; \texttt{end} \;\to\; N[x{:=}V]$	$(\texttt{handle/raise}_{\,1})$
$\texttt{let}\; y : \neg\alpha\; \texttt{in}\; (\texttt{raise}\; z\, V)\; \texttt{handle}\; (y\, x) \Rightarrow N\; \texttt{end} \;\to\; (\texttt{raise}\; z\, V) \quad \text{if } z \neq y$	$(\texttt{handle/raise}_{\,2})$

Table 1. ML-like Reduction Rules

reduction is related to the last three rules: bound variables may become free by reduction.[1] Take for instance Rule $(\texttt{handle}_{\,simp})$: any free occurrence of y in V is bound in the redex but becomes free in the contractum.

5 Modified Operational Semantics

To circumvent the problem related to the ML-like semantics, we must modify the three last rules of Table 1. This idea gives rise to the modified operational semantics specified by

$$(\lambda x.\,M)\,V \;\;\to\;\; M[x{:=}V] \qquad\qquad (\beta_V)$$

$$V_1\,(\mathtt{raise}\;V) \;\;\to\;\; (\mathtt{raise}\;V) \qquad\qquad (\mathtt{raise}_{\,\mathit{left}})$$

$$(\mathtt{raise}\;V)\,M \;\;\to\;\; (\mathtt{raise}\;V) \qquad\qquad (\mathtt{raise}_{\,\mathit{right}})$$

$$(\mathtt{raise}\;(\mathtt{raise}\;V)) \;\;\to\;\; (\mathtt{raise}\;V) \qquad\qquad (\mathtt{raise}_{\,\mathit{idem}})$$

$$\begin{array}{l}\mathtt{let}\;y : \neg\alpha\;\mathtt{in}\;V\;\mathtt{handle}\;(y\,x) \Rightarrow N\;\mathtt{end}\\ \qquad\qquad\qquad \to\;\; V \quad \text{if } y \notin FV(V)\end{array} \qquad\qquad (\mathtt{handle}_{\,\mathit{simp}})$$

$$\begin{array}{l}\mathtt{let}\;y_1 : \neg\alpha_1;\; y_2 : \neg\alpha_2;\; \ldots;\; y_n : \neg\alpha_n\\ \mathtt{in}\;\;(\mathtt{raise}\;y_i\;V)\,\mathtt{handle}\;(y_n\,x) \Rightarrow N_n\\ \qquad\qquad\qquad\qquad \vdots\\ \qquad\qquad\qquad \mid \quad (y_2\,x) \Rightarrow N_2\\ \qquad\qquad\qquad \mid \quad (y_1\,x) \Rightarrow N_1\\ \mathtt{end}\\ \qquad\qquad\qquad \to\;\; \mathtt{let}\;y_1 : \neg\alpha_1;\; y_2 : \neg\alpha_2;\; \ldots;\; y_n : \neg\alpha_n\\ \qquad\qquad\qquad\qquad \mathtt{in}\;\;N_i[x{:=}V]\,\mathtt{handle}\;(y_n\,x) \Rightarrow N_n\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots\\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \quad (y_2\,x) \Rightarrow N_2\\ \qquad\qquad\qquad\qquad\qquad\qquad \mid \quad (y_1\,x) \Rightarrow N_1\\ \qquad\qquad\qquad\qquad \mathtt{end}\end{array} \qquad (\mathtt{handle/raise})$$

$$\begin{array}{l}V\,(\mathtt{let}\;y : \neg\alpha\;\mathtt{in}\;M\;\mathtt{handle}\;(y\,x) \Rightarrow N\;\mathtt{end})\\ \qquad\qquad \to\;\; \mathtt{let}\;y : \neg\alpha\;\mathtt{in}\;V\,M\;\mathtt{handle}\;(y\,x) \Rightarrow V\,N\;\mathtt{end}\end{array} \qquad (\mathtt{handle}_{\,\mathit{left}})$$

$$\begin{array}{l}(\mathtt{let}\;y : \neg\alpha\;\mathtt{in}\;M\;\mathtt{handle}\;(y\,x) \Rightarrow N\;\mathtt{end})\;\;O\\ \qquad\qquad \to\;\; \mathtt{let}\;y : \neg\alpha\;\mathtt{in}\;M\,O\;\mathtt{handle}\;(y\,x) \Rightarrow N\,O\;\mathtt{end}\end{array} \qquad (\mathtt{handle}_{\,\mathit{right}})$$

$$\begin{array}{l}(\mathtt{raise}\;\mathtt{let}\;y : \neg\alpha\;\mathtt{in}\;M\;\mathtt{handle}\;(y\,x) \Rightarrow N\;\mathtt{end})\\ \;\to\;\; \mathtt{let}\;y : \neg\alpha\;\mathtt{in}\;(\mathtt{raise}\;M)\;\mathtt{handle}\;(y\,x) \Rightarrow (\mathtt{raise}\;N)\;\mathtt{end}\end{array} \qquad (\mathtt{raise/handle})$$

Table 2. Modified Reduction Rules

the reduction rules of Table 2. The three first rules of the modified semantics correspond exactly to the ML-like rules. The fourth rule comes with a proviso that ensures that bound variables may not become free by reduction. Rule (`handle/raise`) is more general that the corresponding rules in the ML-like semantics. In particular, this new rule allows handlers to be used more than once. Finally, the three last rules are necessary to ensure that the execution of programs will not be stuck. These three last rules, which may seem intricate, are nothing but the commuting conversions of disjunction that are used in natural deduction [11]. For instance, Rule ($\mathtt{handle}_{\,\mathit{left}}$) corresponds to the following proof reduction:

$$\dfrac{\begin{array}{c}\vdots\\ B \to C\end{array} \qquad \dfrac{\begin{array}{c}[\neg A]\\ \vdots\\ B\end{array} \qquad \begin{array}{c}A\\ \vdots\\ B\end{array}}{B}}{C} \quad\to\quad \dfrac{\dfrac{\begin{array}{c}\vdots\\ B \to C\end{array} \qquad \begin{array}{c}[\neg A]\\ \vdots\\ B\end{array}}{C} \qquad \dfrac{\begin{array}{c}\vdots\\ B \to C\end{array} \qquad \begin{array}{c}[A]\\ \vdots\\ B\end{array}}{C}}{C}$$

Before investigating further the properties of the modified semantics, we must first

[1] This subject reduction problem is not actually related to the exception handling mechanism of standard ML but rather to the possibility of declaring locally a datatype constructor. Take, for instance the ML expression: `let datatype foo = c in c end`. The New-Jersey SML compiler evaluates this expression to the constructor `c` of type `?.foo`. Here, the question mark suggests that the datatype `foo` has been declared in some unknown module, that is in a lost environment.

determine in what sense the reduction rules of Table 2 specify an operational semantics for $\boldsymbol{\lambda}^{\rightarrow}_{exn}$. The problem is that we have only introduced some notions of reduction without defining any reduction strategy. To settle this, we simply define the result of the evaluation of an expression to be its normal form (if any). Then it remains to prove that this normal form, when it exists, is unique. This is the purpose of the next proposition.

Proposition 5.1 (Church-Rosser Property) *Let $\twoheadrightarrow$ be the reduction relation induced by the notions of reduction of Table 2 (that is the least reflexive, transitive relation containing $\rightarrow$ and compatible with the expression formation rules). If A, B, C are expressions such that $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$ then there exists an expression D such that $B \twoheadrightarrow D$ and $C \twoheadrightarrow D$.*

Proof. The property can be established by using the standard technique due to Tait and Martin-Löf. The different cases are numerous because we are dealing with nine notions of reduction. Nevertheless the proof is not difficult because there are only a few critical pairs. $\square$

To better understand how the modified semantics works, let us see how example 4.1, which gave rise to an uncaught exception, may now be reduced. (We use the compact syntax.)

$$\begin{aligned}
&(\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,((\lambda x.\, \lambda y.\, \langle\, P.\, P \mid g.\, \mathcal{R}\,(g\,x\,y)\,\rangle)\,1\,2)\\
&\rightarrow_{\beta_V}\ (\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,((\lambda y.\, \langle\, P.\, P \mid g.\, \mathcal{R}\,(g\,1\,y)\,\rangle)\,2)\\
&\rightarrow_{\beta_V}\ (\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\langle\, P.\, P \mid g.\, \mathcal{R}\,((g\,1\,2))\,\rangle\\
&\rightarrow_{\mathtt{h}_{\mathit{left}}}\ \langle\, P.(\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,P\\
&\qquad\qquad\mid g.(\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\mathcal{R}\,((g\,1\,2))\,\rangle\\
&\rightarrow_{\beta_V}\ \langle\, P.\langle\, y.\, \mathcal{R}\,(P\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle\\
&\qquad\qquad\mid g.(\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\mathcal{R}\,((g\,1\,2))\,\rangle\\
&\rightarrow_{\mathtt{h/r}}\ \langle\, P.\langle\, y.\, (\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\mathcal{R}\,(((\lambda x.\, \mathcal{R}\,(y\,x))\,1\,2))\mid x.\, x\,\rangle\\
&\qquad\qquad\mid g.(\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\mathcal{R}\,((g\,1\,2))\,\rangle\\
&\rightarrow_{\mathtt{h}_{\mathit{simp}}}\ \langle\, y.\, (\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\mathcal{R}\,(((\lambda x.\, \mathcal{R}\,(y\,x))\,1\,2))\mid x.\, x\,\rangle\\
&\rightarrow_{\beta_V}\ \langle\, y.\, (\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\mathcal{R}\,((\mathcal{R}\,(y\,1)\,2))\mid x.\, x\,\rangle\\
&\rightarrow_{\mathtt{r}_{\mathit{right}}}\ \langle\, y.\, (\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\mathcal{R}\,(\mathcal{R}\,((y\,1)))\mid x.\, x\,\rangle\\
&\rightarrow_{\mathtt{r}_{\mathit{idem}}}\ \langle\, y.\, (\lambda p.\, \langle\, y.\, \mathcal{R}\,(p\,(\lambda x.\, \mathcal{R}\,(y\,x)))\mid x.\, x\,\rangle)\,\mathcal{R}\,((y\,1))\mid x.\, x\,\rangle\\
&\rightarrow_{\mathtt{r}_{\mathit{left}}}\ \langle\, y.\, \mathcal{R}\,(y\,1)\mid x.\, x\,\rangle\\
&\rightarrow_{\mathtt{h/r}}\ \langle\, y.\, 1 \mid x.\, x\,\rangle\\
&\rightarrow_{\mathtt{h}_{\mathit{simp}}}\ 1
\end{aligned}$$

As expected, $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ with the modified semantics satifies the subject reduction property.

Proposition 5.2 (Subject Reduction Property) *Let Γ be a typing environment, A be an expression, and α be a type such that $\Gamma \vdash A : \alpha$. If $A \rightarrow B$, according to the one-step reduction relation induced by the notions of reduction of Table 2, then $\Gamma \vdash B : \alpha$.*

Proof. As usual, the proof is done by induction on the derivation of $A \rightarrow B$. A substituiton lemma and a few easy lemmas concerning the typing relation are needed. $\square$

The above proposition immediately generalises to the reduction relation $\twoheadrightarrow$. By Proposition 3.5, we know that there cannot exist closed expressions of type $\perp$. This implies that there also cannot exist closed expressions of the form ($\mathtt{raise}\ M$) that are well-typed. Hence, as a corollary of Proposition 5.2, any closed expression that is well-typed cannot be reduced to an expression of the form ($\mathtt{raise}\ M$). Therefore, if one sees $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ as an idealised programming language, we get the interesting property that the evaluation of well-typed programs never gives rise to uncaught exceptions.

6 Relation between the two Operational Semantics

The results so far are encouraging. We have introduced a simple calculus of exception handling whose type system amounts, through the Curry-Howard isomorphism, to classical logic. We have also provided this calculus with notions of reduction that make sense from a proof-theoretical point of view. The resulting system satisfies properties of interest such as Church-Rosser and subject reduction. Nevertheless, we achieved this last result by modifying the ML-like semantics, while our prime motivation was to analyse ML-like exception handling. This raises several questions. How far did we modify the ML-like semantics? Can the new semantics be seen as an extension of the ML-like ones, or did we obtain something completely different?

In this section, we answer the above questions by proving that, as far as non-exceptional values are concerned, the modified semantics is a conservative extention of the ML-like one. This means that whenever a program P evaluates to a non-exceptional value V according to the ML-like semantics, this program P evaluates to the same value V according to the modified semantics, where programs are defined to be closed expressions of ground type.

First we define the notion of non-exceptional value.

Definition 6.1 Non-exceptional values *are defined by the following grammar:*

$$U \ ::= \ c \mid x \mid \lambda x.\, T$$

Note that, as far as ground types different from $\perp$ are concerned, the notion of non-exceptional value coincides exactly with the one of value. From now on, the uppercase Roman letter U (with possible subscripts) will range over non-exceptional values.

In order to state the property that we intend to prove in this section, we must define some standard reduction strategies. To this end, we introduce the notion of applicative context.

Definition 6.2 Applicative contexts *are defined by the following grammar, where* [ ] *stands for the empty context:*

$$\mathcal{C} \ ::= \ [\ ] \mid V\,\mathcal{C} \mid \mathcal{C}\,T \mid (\texttt{raise}\,\mathcal{C}) \mid \texttt{let}\, y : \neg\tau \,\texttt{in}\, \mathcal{C}\, \texttt{handle}\, (y\,x) \Rightarrow T\, \texttt{end}$$

We denote by $\mathcal{C}[A]$ the expression obtained by plugging an expression A into an applicative context $\mathcal{C}$.

The notion of applicative context allows the one of standard reduction to be defined.

Definition 6.3 *Let A and B be expressions. We say that A* reduces standardly in one step to *B, according to the ML-like notions of reduction if and only if there exist expressions A', B' and an applicative context $\mathcal{C}$ such that*

(i) $A \equiv \mathcal{C}[A']$,

(ii) $B \equiv \mathcal{C}[B']$,

(iii) *A', B' is one of the redex/contractum pairs specified by Table 1,*

where $\equiv$ stands for the syntactic identity (modulo α-conversion).

The ML-like one-step standard reduction relation from A to B is denoted by $A \to_{ML} B$. The refexive, transitive closure of this relation is written $\twoheadrightarrow_{ML}$.

The modified one-step standard reduction ($\to_{exn}$) and its reflexive, transitive closure ($\twoheadrightarrow_{exn}$) are defined similarly.

In the case of programs, i.e. closed terms of ground types, it can be shown that the standard reduction is a normalizing strategy. We are now ready to state and prove that the modified semantics is conservative over the ML-like one.

Proposition 6.4 (Conservation Property) *Let P be a program and U be a non-exceptional value. If $P \twoheadrightarrow_{ML} U$ then $P \twoheadrightarrow_{exn} U$.*

Proof. The complete proof requires an induction on the number of `handle` constructs that are nested into the applicative contexts. We give here the proof for the basic case and leave the inductive case to the reader.

We proceed by induction on the length of the standard reduction sequence $P \twoheadrightarrow_{ML} U$.

Since $P \twoheadrightarrow_{ML} U$, we know that there exist a redex/contractum pair A, B and an applicative context $\mathcal{C}$ such that:

$$P \;\equiv\; \mathcal{C}[A] \;\rightarrow_{ML}\; \mathcal{C}[B] \;\twoheadrightarrow_{ML}\; U$$

There are seven cases, according to the notion of reduction to which the redex/contractum pair A, B belongs.

In the case of β_V, $\texttt{raise}_{\mathit{left}}$, $\texttt{raise}_{\mathit{right}}$, and $\texttt{raise}_{\mathit{idem}}$ the proof is immediate because these four notions of reduction are kept unchanged in the modified semantics.

In the case of $\texttt{handle}_{\mathit{simp}}$, we have $A \equiv \langle\, y.\, V \mid x.\, N \,\rangle$ and $B \equiv V$. Then, using $\texttt{handle}_{\mathit{left}}$, $\texttt{handle}_{\mathit{right}}$, and `raise/handle`, we have:

$$\mathcal{C}[\langle\, y.\, V \mid x.\, N \,\rangle] \;\twoheadrightarrow_{exn}\; \langle\, y.\, \mathcal{C}[V] \mid x.\, \mathcal{C}[N] \,\rangle.$$

Then, by induction hypothesis,

$$\langle\, y.\, \mathcal{C}[V] \mid x.\, \mathcal{C}[N] \,\rangle \;\twoheadrightarrow_{exn}\; \langle\, y.\, U \mid x.\, \mathcal{C}[N] \,\rangle.$$

Since U is non-exceptional, $y \notin FV(U)$. Therefore, by modified $\texttt{handle}_{\mathit{simp}}$, we get

$$\langle\, y.\, U \mid x.\, \mathcal{C}[N] \,\rangle \;\rightarrow_{exn}\; U.$$

The proof in the last two cases ($\texttt{handle/raise}_1$ and $\texttt{handle/raise}_2$) is similar. □

7 CPS-Interpretation

The modified semantics appears now as a conservative extension of the ML-like ones. Nevertheless, one may still wonder if Table 2 is just an ad-hoc adaptation of the ML-like reduction rules, designed to make Proposition 6.4 hold, or if there is anything canonical in the modified semantics. We already answered partially this question by suggesting that the modified reduction rules correspond to (well-known) proof-theoretic conversions.

In this section, we justify further the modified operational semantics by showing that there is a logical embedding (in the sense of [12]) of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ into the simply typed λ-calculus. To this end we introduce a *continuation-passing-style* translation of the expressions of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$. This CPS-translation is an adaptation of Plotkin's [19].

Definition 7.1 (CPS-translation) *The* CPS-translation $\overline{M}$ *of an expression M is inductively defined as follows:*

(i) $\overline{c} \;=\; \lambda k.\, k\, c;$

(ii) $\overline{x} \;=\; \lambda k.\, k\, x;$

(iii) $\overline{y} \;=\; \lambda k.\, k\, (\lambda v.\, \lambda k.\, k\, (y\, v));$

(iv) $\overline{\lambda x.\, M} \;=\; \lambda k.\, k\, (\lambda x.\, \overline{M});$

(v) $\overline{M\, N} \;=\; \lambda k.\, \overline{M}\, (\lambda m.\, \overline{N}\, (\lambda n.\, m\, n\, k));$

(vi) $\overline{\mathcal{R}\,(M)} \;=\; \lambda k.\, \overline{M}\, (\lambda x.\, x);$

(vii) $\overline{\langle\, y.\, M \mid x.\, N \,\rangle} \;=\; \lambda k.\, (\lambda y.\, \overline{M} k\,)\, (\lambda x.\, \overline{N}\, k).$

This CPS-translation is correct with respect to the modified reduction rules. To prove this, we need one auxiliary definition and three technical lemmas.

Definition 7.2 *The auxiliary function $\mathbf{\Psi}$, sending values to values, is defined as follows:*

(i) $\mathbf{\Psi}(c) \ = \ c$;

(ii) $\mathbf{\Psi}(x) \ = \ x$;

(iii) $\mathbf{\Psi}(y) \ = \ \lambda v.\, \lambda k.\, k\,(y\,v)$;

(iv) $\mathbf{\Psi}(\lambda x.\, M) \ = \ \lambda x.\, \overline{M}$;

(v) $\mathbf{\Psi}(y\,V) \ = \ y\,\mathbf{\Psi}(V)$.

Lemma 7.3 $\overline{V} \twoheadrightarrow_\beta \lambda k.\, k\,\mathbf{\Psi}(V)$, *for any value* V.

Proof. By induction on the structure of V. □

Lemma 7.4 $\overline{M[x{:=}V]} \twoheadrightarrow_\beta \overline{M}[x{:=}\mathbf{\Psi}(V)]$, *for any expression* M *and any value* V.

Proof. By induction on the structure of M, using Lemma 7.3 for the base case. □

Lemma 7.5 *Let* M *be an expression. If* $k \notin FV(M)$ *then* $\lambda k.\, \overline{M}\,k \rightarrow_\beta \overline{M}$.

Proof. Because $\overline{M}$ is an abstraction for any M. □

We are now in the position of proving that the CPS-translation of Definition 7.1 is compatible with the modified semantics. More precisely, we intend to prove that the translation preserves conversion between terms.

Proposition 7.6 (Correctness of the CPS-Translation) *Let* A, B *be expressions. If* $A \rightarrow B$, *according to the modified semantics, then* $\overline{A} =_\beta \overline{B}$.

Proof. The proof is done by induction on the derivation of $A \rightarrow B$. The inductive steps are straightforward. As for the basic cases, we focus on three of them, leaving the others, which are similar, to the reader.
(`handle`$_{simp}$)

$$\begin{array}{rll} \overline{\langle\, y.\, V \mid x.\, N\,\rangle} = & \lambda k.\, (\lambda y.\, \overline{V}\,k)\,(\lambda x.\, \overline{N}\,k) & \\ =_\beta & \lambda k.\, \overline{V}\,k & y \notin FV(\overline{V}) \\ =_\beta & \overline{V} & \text{by Lemma 7.5} \end{array}$$

(`handle/raise`)

$$\begin{array}{ll} \overline{\langle\, (y_i).\, \mathcal{R}\,(y_j\,V) \mid (x_i.\, N_i)\,\rangle} & \\ =_\beta \lambda k.\, (\lambda y_1.\, \ldots (\lambda y_n.\, (\lambda k.\, \overline{y_j\,V}\,(\lambda x.\, x))\,k)\,(\lambda x_n.\, \overline{N_n}\,k) \cdots)\,(\lambda x_1.\, \overline{N_1}\,k) & \\ =_\beta \lambda k.\, (\lambda y_1.\, \ldots (\lambda y_n.\, \overline{y_j\,V}\,(\lambda x.\, x))\,(\lambda x_n.\, \overline{N_n}\,k) \cdots)\,(\lambda x_1.\, \overline{N_1}\,k) & \\ =_\beta \lambda k.\, (\lambda y_1.\, \ldots (\lambda y_n.\, (\lambda k.\, k\,(y_j\,\mathbf{\Psi}(V)))\,(\lambda x.\, x))\,(\lambda x_n.\, \overline{N_n}\,k) \cdots)\,(\lambda x_1.\, \overline{N_1}\,k) & \\ & \text{by Lemma 7.3} \\ =_\beta \lambda k.\, (\lambda y_1.\, \ldots (\lambda y_n.\, y_j\,\mathbf{\Psi}(V))\,(\lambda x_n.\, \overline{N_n}\,k) \cdots)\,(\lambda x_1.\, \overline{N_1}\,k) & \\ =_\beta \lambda k.\, (\lambda y_1.\, \ldots (\lambda y_n.\, (\lambda x_j.\, \overline{N_j}\,k)\,\mathbf{\Psi}(V))\,(\lambda x_n.\, \overline{N_n}\,k) \cdots)\,(\lambda x_1.\, \overline{N_1}\,k) & \\ =_\beta \lambda k.\, (\lambda y_1.\, \ldots (\lambda y_n.\, \overline{N_j}[x_j{:=}\mathbf{\Psi}(V)]\,k)\,(\lambda x_n.\, \overline{N_n}\,k) \cdots)\,(\lambda x_1.\, \overline{N_1}\,k) & \\ =_\beta \lambda k.\, (\lambda y_1.\, \ldots (\lambda y_n.\, \overline{N_j[x_j{:=}V]}\,k)\,(\lambda x_n.\, \overline{N_n}\,k) \cdots)\,(\lambda x_1.\, \overline{N_1}\,k) & \text{by Lemma 7.4} \\ =_\beta \overline{\langle\, (y_i).\, N_j[x_j{:=}V] \mid (x_i.\, N_i)\,\rangle} & \end{array}$$

($\texttt{handle}_{\,left}$)

$$\begin{array}{ll}
\multicolumn{2}{l}{\overline{V\,\langle\, y.\,M \mid x.\,N\,\rangle}} \\
= & \lambda k.\,\overline{V}\,(\lambda m.\,(\lambda k.\,(\lambda y.\,\overline{M}\,k)\,(\lambda x.\,\overline{N}\,k))\,(\lambda n.\,m\,n\,k)) \\
=_\beta & \lambda k.\,(\lambda k.\,k\,\mathbf{\Psi}(V))\,(\lambda m.\,(\lambda k.\,(\lambda y.\,\overline{M}\,k)\,(\lambda x.\,\overline{N}\,k))\,(\lambda n.\,m\,n\,k)) \quad \text{by Lemma 7.3} \\
=_\beta & \lambda k.\,(\lambda k.\,(\lambda y.\,\overline{M}\,k)\,(\lambda x.\,\overline{N}\,k))\,(\lambda n.\,\mathbf{\Psi}(V)\,n\,k) \\
=_\beta & \lambda k.\,(\lambda y.\,\overline{M}\,(\lambda n.\,\mathbf{\Psi}(V)\,n\,k))\,(\lambda x.\,\overline{N}\,(\lambda n.\,\mathbf{\Psi}(V)\,n\,k)) \\
=_\beta & \lambda k.\,(\lambda y.\,(\lambda k.\,(\lambda k.\,k\,\mathbf{\Psi}(V))\,(\lambda m.\,\overline{M}\,(\lambda n.\,m\,n\,k)))\,k) \\
 & \qquad\qquad\qquad\qquad (\lambda x.\,(\lambda k.\,(\lambda k.\,k\,\mathbf{\Psi}(V))\,(\lambda m.\,\overline{N}\,(\lambda n.\,m\,n\,k)))\,k) \\
=_\beta & \lambda k.\,(\lambda y.\,(\lambda k.\,\overline{V}\,(\lambda m.\,\overline{M}\,(\lambda n.\,m\,n\,k)))\,k)\,(\lambda x.\,(\lambda k.\,\overline{V}\,(\lambda m.\,\overline{N}\,(\lambda n.\,m\,n\,k)))\,k) \\
 & \hfill \text{by Lemma 7.3} \\
= & \lambda k.\,\overline{V\,M}\,k[y{:=}\lambda x.\,\overline{V\,N}\,k] \\
= & \overline{\langle\, y.\,V\,M \mid x.\,V\,N\,\rangle}
\end{array}$$

□

Griffin has shown that Plotkin's CPS-translation induces, at the level of the types, a negative translation of classical logic into intuitionistic one [12].

Definition 7.7 *Griffin's* negative translation *is defined as follows:*
$\overline{\alpha} = \neg\neg\alpha^*$, *where:*

(i) $\bot^* \;=\; \bot$*;*

(ii) $a^* \;=\; a$, *for a atomic;*

(iii) $(\alpha \to \beta)^* \;=\; \alpha^* \to \overline{\beta}$

Our CPS-translation and Griffin's negative translation commute with the typing relation of $\boldsymbol{\lambda}^{\to}_{exn}$ and the one of the simply-typed λ-calculus. This is stated by the next propostion, which shows that there is a logical embedding of $\boldsymbol{\lambda}^{\to}_{exn}$ into the simply-typed λ-calculus.

Proposition 7.8 (Logical Interpretation) *Let* $\vdash_{exn}$ *and* $\vdash_\lambda$ *stand respectively for the typing relation of* $\boldsymbol{\lambda}^{\to}_{exn}$ *and the one of the simply-typed* λ*-calculus. If* $\vdash_{exn}\ M : \alpha$ *then* $\vdash_\lambda\ \overline{M} : \overline{\alpha}$*, for any expression* M *and any type* α*.*

Proof. We prove that, whenever the sequent

$$(x_i : \alpha_i),\ (y_j : \neg\beta_j)\ \vdash_{exn}\ M : \alpha$$

is derivable, so is the sequent

$$(x_i : \alpha_i^*),\ (y_j : \neg\beta_j^*)\ \vdash_\lambda\ \overline{M} : \overline{\alpha}.$$

We proceed by induction on the derivation of the typing judgement. We give the proof for the `handle` construct and leave the other cases to the reader.

$$\dfrac{\dfrac{\dfrac{\left\{\begin{array}{l} y : \neg\beta^* \vdash_\lambda \overline{M} : \neg\neg\alpha^* \\ k : \neg\alpha^* \vdash_\lambda k : \neg\alpha^* \end{array}\right.}{k : \neg\alpha^*,\, y : \neg\beta^* \vdash_\lambda \overline{M}\,k : \bot}}{k : \neg\alpha^* \vdash_\lambda \lambda y.\,\overline{M}\,k : \neg\neg\beta^*} \qquad \dfrac{\dfrac{\left\{\begin{array}{l} x : \beta^* \vdash_\lambda \overline{N} : \neg\neg\alpha^* \\ k : \neg\alpha^* \vdash_\lambda k : \neg\alpha^* \end{array}\right.}{k : \neg\alpha^*,\, x : \beta^* \vdash_\lambda \overline{N}\,k : \bot}}{k : \neg\alpha^* \vdash_\lambda \lambda x.\,\overline{N}\,k : \neg\beta^*}}{\dfrac{k : \neg\alpha^* \vdash_\lambda (\lambda y.\,\overline{M}\,k)\,(\lambda x.\,\overline{N}\,k) : \bot}{\vdash_\lambda \lambda k.\,(\lambda y.\,\overline{M}\,k)\,(\lambda x.\,\overline{N}\,k) : \overline{\alpha}}}$$

□

8 Strong Normalisation

By Proposition 6.4, we know that the ML-like and the modified operational semantics are equivalent for programs that yield non exceptional results. This means that whenever a program terminates and yields some value according to the ML-like semantics, it will terminate and yield the same value according to the modified semantics. On the other hand, by Propositions 3.5 and 5.2, we know that the modified semantics ensures that programs may not yield exceptional results (i.e. uncaught exceptions). Therefore, it seems that these three propositions together allows us to conclude that the modified semantics should be preferred to the original one.

This conclusion, however, is premature because Proposition 6.4 concerns only programs yielding non-exceptional results. Indeed a program raising an uncaught exception according to the ML-like semantics could possibly loop for ever according to the modified semantics. We would then have replaced a property that is observable (the production of an uncaught exception) by some other property that is not observable (the non-termination of a program).

In order to eliminate this last possible objection, we establish the strong normalisation of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$. Technically, we show that any infinite sequence of reductions in $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ induces an infinite sequence of β-reductions in the simply-typed λ-calculus. To this end, we need a syntactic translation of the expressions of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ into simply-typed λ-terms. The CPS-translation of Definition 7.1 is such a syntactic translation. Nevertheless, we cannot use it as such because it does not properly simulate the reduction relation of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$. Therefore, we introduce a modified CPS-translation.

Definition 8.1 *The* modified CPS-translation $\overline{\overline{M}}$ *of an expression* M *is defined as follows:* $\overline{\overline{M}} = \lambda k.\, M : k$, *where*

(i) $V : K \;=\; K\,\boldsymbol{\Phi}(V)$;
(ii) $(V_1\,V_2) : K \;=\; \boldsymbol{\Phi}(V_1)\,\boldsymbol{\Phi}(V_2)\,K$;
(iii) $(V_1\,N) : K \;=\; N : \lambda n.\, \boldsymbol{\Phi}(V_1)\,n\,K$;
(iv) $(M\,V_2) : K \;=\; M : \lambda m.\, m\,\boldsymbol{\Phi}(V_2)\,K$;
(v) $(M\,N) : K \;=\; M : (\lambda m.\, N : (\lambda n.\, m\,n\,K))$;
(vi) $\mathcal{R}\,(M) : K \;=\; M : (\lambda x.\, x)$;
(vii) $(\langle\, y.\, M \mid x.\, N\,\rangle) : K \;=\; M : K\;[y{:=}\lambda x.\,(N : K)\,]$;

(viii) $\boldsymbol{\Phi}(c) \;=\; c$;
(ix) $\boldsymbol{\Phi}(x) \;=\; x$;
(x) $\boldsymbol{\Phi}(y) \;=\; \lambda v.\, \lambda k.\, k\,(y\,v)$;
(xi) $\boldsymbol{\Phi}(\lambda x.\, M) \;=\; \lambda x.\, \overline{\overline{M}}$;
(xii) $\boldsymbol{\Phi}(y\,V) \;=\; y\,\boldsymbol{\Phi}(V)$.

The modified CPS-translation is compatible with the CPS-translation of the previous section in the sense of the following proposition:

Proposition 8.2 *Let* A *be an expression. Then:*

(a) $\boldsymbol{\Psi}(A) \twoheadrightarrow_\beta \boldsymbol{\Phi}(A)$, *whenever* A *is a value,*
(b) $\overline{A}\,K \twoheadrightarrow_\beta A : K$, *for any expression* K,
(c) $\overline{A} \twoheadrightarrow_\beta \overline{\overline{A}}$.

Proof. The proof is by induction on the structure of A. Property (c) is the property of interest, while Properties (a) and (b) are needed to make the induction work. □

Proposition 8.2.(c), together with the subject reduction property of the simply-typed λ-calculus, implies that Proposition 7.8 still holds with the modifed CPS-translation.

As expected, the modified CPS-translation allows the notions of reduction of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ to be simulated by β-reduction. This property is stated by the two next propositions.

Proposition 8.3 *Let R stand for the notion of reduction β_V or* `handle/raise`. *Let A and B be two expressions such that $A \rightarrow_R B$. Then:*

(a) $A : K \overset{+}{\rightarrow}_\beta B : K$, *for any expression K,*

(b) $\overline{\overline{A}} \overset{+}{\rightarrow}_\beta \overline{\overline{B}}$,

where $\overset{+}{\rightarrow}_\beta$ stands for the transitive (but not reflexive) closure of $\rightarrow_\beta$.

Proof. Property (b), which is the property of interest, is a direct consequence of (a). The proof of (a) is akin to the one of proposition 7.6. Lemmas similar to Lemmas 7.3, 7.4, and 7.5 are needed. □

The other notions of reduction of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ are invariants of the translation.

Proposition 8.4 *Let R stand for one of the notions of reduction* $\texttt{raise}_{\textit{left}}$, $\texttt{raise}_{\textit{right}}$, $\texttt{raise}_{\textit{idem}}$, $\texttt{handle}_{\textit{simp}}$, $\texttt{handle}_{\textit{left}}$, $\texttt{handle}_{\textit{right}}$, *or* `raise/handle`. *Let A and B be two expressions. If $A \rightarrow_R B$ then $\overline{\overline{A}} \equiv \overline{\overline{B}}$.*

Proof. The proof is similar to that of proposition 7.6. □

We may not yet conclude that $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ is strongly normalisable because there is still the possibility of infinite reduction sequences due to the notions of reduction of Proposition 8.4. For these notions of reduction, we must establish strong normalisation independently. To this end, we introduce a norm on the expressions of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$.

Definition 8.5 *The* norm $|A|$ *of an expression A is inductively defined as follows:*

(i) $|c| \;=\; 1$;
(ii) $|x| \;=\; 1$;
(iii) $|y| \;=\; 1$;
(iv) $|\lambda x.\, M| \;=\; |M|$;
(v) $|M\,N| \;=\; (\#N \times |M|) + (\#M \times |N|)$;
(vi) $|\mathcal{R}\,(M)| \;=\; |M| + \#M$;
(vii) $|\langle\, y.\, M \mid x.\, N\,\rangle| \;=\; |M| + |N|$;
(viii) $\#c \;=\; 1$;
(ix) $\#x \;=\; 1$;
(x) $\#y \;=\; 1$;
(xi) $\#\lambda x.\, M \;=\; \#M$;
(xii) $\#M\,N \;=\; \#M \times \#N$;
(xiii) $\#\mathcal{R}\,(M) \;=\; \#M$;
(xiv) $\#\langle\, y.\, M \mid x.\, N\,\rangle \;=\; \#M + \#N + 1$.

Proposition 8.6 *Let R stand for one of the notions of reduction* $\texttt{raise}_{\textit{left}}$, $\texttt{raise}_{\textit{right}}$, $\texttt{raise}_{\textit{idem}}$, $\texttt{handle}_{\textit{simp}}$, $\texttt{handle}_{\textit{left}}$, $\texttt{handle}_{\textit{right}}$, *or* `raise/handle`. *Let A and B be two expressions. If $A \rightarrow_R B$ then $|A| > |B|$.*

Proof. The proof is a straightforward induction on the derivation of $A \rightarrow B$. □

We may now state the main result of this section.

Proposition 8.7 (Strong Normalisation) *Any well-typed term of $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ is strongly normalisable.*

Proof. Consequence of Propositions 7.8, 8.2, 8.3, 8.4, and 8.6. □

9 Related Work and Conclusions

Griffin, in 1990, was the first to stress the relation between sequential control and classical logic [12]. His work is based on Felleisen's syntactic theory of sequential control, which provides an idealisation of Scheme `call/cc`.

Around the same time, Murthy studied the computational content of classical proofs [6, 14]. His work is based mainly on negative translations of classical logic, and CPS-transforms.

The work of Griffin was extended by Barbanera and Berardi [2, 3], who noted that Felleinsen's reduction rules are similar to Prawitz's handling of double negation [20]. They use a control operator akin to Felleisen's $\mathcal{C}$ to extract the computational content of classical proofs of Σ_1^0-sentences.

On the proof-theoretic side, Parigot has introduced the $\lambda\mu$-calculus, an algorithmic interpretation of cut elimination in classical natural deduction [16, 17, 18]. From a computer science point of view, the "classical" constructs of the $\lambda\mu$-calculus may be interpreted in terms of labels and jumps [7].

Independently of Parigot, Rehof and Sørensen have developped a calculus (λ_Δ) reminiscent of the $\lambda\mu$-calculus [21]. They use applications of the form (xM) instead of named terms. Consequently they may simulate Parigot's structural reduction with usual substitution and β-reduction.

In yet another direction, there is Girard's work on classical logic, based on the notion of polarity [10]. Girard does not provide his system with expressions encoding proofs. Nevertheless, Murthy has shown how to extract λ-terms from Girard's system, using Felleisen's operator [15].

More recently, Barbanera and Berardi have introduced an intriguing symmetric λ-calculus based on a syntactic identification between a type and its double negation [4].

All the above systems are based on computational interpretations of double negations. In the case of λ_Δ and of the systems based on Felleisen's theory of control, the treatment of double negations, which is explicit, is based on Prawitz-like reduction rules. In the case of Parigot's $\lambda\mu$-calculus, Girard's LC, and Berardi's symmetric calculus, the treatment of double negations is left implicit (it is hidden somehow in the formal system). Nevertheless, we have shown that first-order $\lambda\mu$-claculus is isomorphic to a subtheory of the Felleisen-Griffin system [8].

To the best of our knowledge, $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ is the only "classical" λ-calculus based on a computational interpretation of the elimination of the excluded-middle law, consequently modelling ML-like exception handling. Nevertheless this difference, which is not as deep as it could seem at first sight, vanishes from a denotational point of view. Indeed, Definitions 7.1 and 7.7 do not differ from the ones given by Griffin for Felleisen's calculus. Consequently, it should be possible to simulate $\boldsymbol{\lambda}^{\rightarrow}_{exn}$ in one of the other calculi by using translations akin to the ones used in [8, 21].

References

1. A. Appel, D.B. MacQueen, R. Milner, and M. Tofte. Unifying exceptions with constructors in standard ML. Technical Report ECS-LFCS-88-55, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.
2. F. Barbanera and S. Berardi. Continuations and simple types: a strong normalization result. In *Proceedings of the ACM SIGPLAN Workshop on Continuations*. Report STAN-CS-92-1426, Stanford University, 1992.
3. F. Barbanera and S. Berardi. Extracting constructive content from classical logic via control-like reductions. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 45–59. Lecture Notes in Computer Science, 664, Springer Verlag, 1993.

4. F. Barbanera and S. Berardi. A symmetric lambda-calculus for “classical” program extraction. In M. Hagiya and J.C. Mitchell, editors, *Proceedings of the International Symposium on Theretical Aspects of Computer Software*, pages 494–515. Lecture Notes in Computer Science, 789, Springer Verlag, 1994.
5. H.P. Barendregt. *The lambda calculus, its syntax and semantics.* North-Holland, revised edition, 1984.
6. R. Constable and C. Murthy. Finding computational content in classical proofs. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 341–362. Cambridge University Press, 1991.
7. Ph. de Groote. A CPS-translation of the $\lambda\mu$-calculus. In S. Tison, editor, *Proceedings of the 19th International Colloquium on Trees in Algebra and Programming (CAAP'94)*, pages 85–99. Lecture Notes in Computer Science, 787, Springer Verlag, 1994.
8. Ph. de Groote. On the relation between the $\lambda\mu$-calculus and the syntactic theory of sequential control. In *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning–LPAR'94*, pages 31–43. Lecture Notes in Computer Science, 822, Springer Verlag, 1994.
9. J.H. Gallier. On the correspondence between proofs and λ-terms. In Ph. de Groote, editor, *Cahiers du Centre de Logique (Université Catholique de Louvain), Volume 8*, pages 55–138. Academia, Louvain-la-Neuve, 1995.
10. J.-Y. Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1:255–296, 1991.
11. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.
12. T. G. Griffin. A formulae-as-types notion of control. In *Conference record of the seventeenth annual ACM symposium on Principles of Programming Languages*, pages 47–58, 1990.
13. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.
14. C. R. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the sixth annual IEEE symposium on logic in computer science*, pages 96–107, 1991.
15. C. R. Murthy. A computational analysis of Girard's translation and LC. In *Proceedings of the seventh annual IEEE symposium on logic in computer science*, pages 90–101, 1992.
16. M. Parigot. $\lambda\mu$-Calculus: an algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 190–201. Lecture Notes in Artificial Intelligence, 624, Springer Verlag, 1992.
17. M. Parigot. Classical proofs as programs. In G. Gottlod, A. Leitsch, and D. Mundici, editors, *Proceedings of the third Kurt Gödel colloquium – KGC'93*, pages 263–276. Lecture Notes in Computer Science, 713, Springer Verlag, 1993.
18. M. Parigot. Strong normalization for second order classical natural deduction. In *Proceedings of the eighth annual IEEE symposium on logic in computer science*, pages 39–46, 1993.
19. G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theretical Computer Science*, 1:125–159, 1975.
20. D. Prawitz. *Natural Deduction, A Proof-Theoretical Study.* Almqvist & Wiksell, Stockholm, 1965.
21. N.J. Rehof and M.H. Sørensen. The λ_Δ-calculus. In M. Hagiya and J.C. Mitchell, editors, *Proceedings of the International Symposium on Theoretical Aspects of Computer Software – TACS'94*, pages 516–542. Lecture Notes in Computer Science, 789, Springer Verlag, 1994.