JFP **15** (1): 33–52, 2005.
DOI: 10.1017/S0956796804005362 Printed in the United Kingdom

An intuitionistic λ-calculus with exceptions

R. DAVID and G. MOUNIER

Laboratoire de Mathématiques, Université de Savoie Campus Scientifique, 73376 Le Bourget du Lac cedex, France
(*e-mail:* david@univ-savoie.fr, Georges.Mounier@ac-lyon.fr)

Abstract

We introduce a typed λ-calculus which allows the use of exceptions in the ML style. It is an extension of the system AF_2 of Krivine & Leivant (Krivine, 1990; Leivant, 1983). We show its main properties: confluence, strong normalization and weak subject reduction. The system satisfies the "the proof as program" paradigm as in AF_2. Moreover, the underlined logic of our system is *intuitionistic* logic.

1 Introduction

The major benefit of the "proof as program" paradigm (also known as Curry–Howard isomorphism) is that the proof itself ensures that the program extracted is correct. For a long time it has been restricted to intuitionistic logic. This had a major drawback: the programs extracted from intuitionistic proofs have no mechanisms for non local exit. However, in practical programming, these instructions play an important role to handle exceptional situations: `Catch and throw` in Lisp, `raise`, `handle` in ML or `raise, try with` in CAML (a variant of the ML family) are some examples of such instructions. The `call-cc` (call with current continuation) of Scheme is an even more powerful facility.

Since Griffin (1990) has shown that Felleisen's operator $\mathscr{C}$ (Friedman *et al.*, 1987) may be typed by using classical logic, and thus has opened the proof as program paradigm to classical logic, many type systems were proposed to extend the Curry–Howard isomorphism: De Groote's $\lambda_{\overrightarrow{exn}}$ calculus (de Groote, 1995), Krivine's λ_c-calculus (Krivine, 1994), Parigot's $\lambda\mu$-calculus (Parigot, 1992) or Nakano's catch and throw mechanism (Nakano, 1994) are some of them.

The logic "behind" these systems is classical logic, in the sense that the system allows to prove formulas that are not provable in intuitionistic logic. The programs extracted provide an *unlimited* access to the current continuation. Unlike the `call-cc` of Scheme, `try with` in CAML or `catch and throw` in Lisp provides only a *restricted* access to the current continuation.

None of the systems mentioned above are totally satisfactory because first order either is missing or does not allow a correct treatment of exceptions: the proof that gives a program satisfying the specifications is too far from intuition.

- De Groote's $\lambda_{\overrightarrow{exn}}$ calculus uses a ML-like mechanism to capture exceptions, but it uses propositional logic, and it is not clear how to extend it to second order logic.
- Nakano's system (we have not looked at it in detail) seems to have the same drawbacks as $\lambda_{\overrightarrow{exn}}$.
- Krivine's λ_c-calculus uses a term C of type $\forall X(\neg\neg X \to X)$ that looks like the `call-cc` of Scheme, but this calculus is not confluent, and thus a particular reduction strategy (the head reduction) is needed. Moreover, the only type preserved by reduction is $\bot$.
- Parigot's $\lambda\mu$-calculus is based on a proof system with several conclusions, and thus is classic, but it is not clear how to extract programs from a proof of the totality of a function.

The systems above have the full power of classical logic. We do not care *a priori* about which logic we use. We would like a language having an exception mechanism similar to that of CAML and such that programs can be extracted from proofs. We started from AF_2, the second order type system introduced by Krivine (1990a) Leivant (1983), where the specification of a program is given by equations which correspond to a particular algorithm, and we tried to add exception mechanisms to this system while preserving its basic principle: a term computing a function is extracted from any proof of the totality of this function.

This paper introduces a typed λ-calculus called EX_2 which is an extension of AF_2 and satisfies the following properties:

- There is an exception mechanism "à la CAML".
- The program extracted from *any* proof of the totality of some function computes this function.
- A well typed program will never raise an uncaught exception. Note that this is not the case in CAML!
- The (untyped) λ-calculus is confluent, and typed λ-terms are strongly normalizable.
- The system satisfies the subject reduction property. Actually, it satisfies only a weak form of this property, but this is enough for safe programming.
- The induced logic is intuitionistic logic. This shows that intuitionistic logic is enough to get restricted access to the current continuation.

The paper is organized as follows. In section 2 we define the syntax of EX_2 terms and the typing system. Section 3 gives some examples: we show that, by using exceptions, we may get more efficient programs. In section 4 we state the main properties of EX_2: confluence of the reduction, strong normalization and parallel subject reduction. We also show that the induced logic is intuitionistic logic.

The proofs of the various results are not given in this paper. The appendix gives the main ideas, and points out the main difficulties. Complete proofs can be found in Mounier (1999), which is available on the web page of the first author.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

2 The system

2.1 The untyped calculus

Notations

1. A *data type* is given by a list of typed constructors. For example, the data type of
   (a) natural numbers is given by $N = \{0{:}N,\ S{:}N \to N\}$;
   (b) booleans is given by $Bool = \{true{:}Bool, false{:}Bool\}$;
   (c) lists of elements of type N is given by $L = \{nil{:}L,\ cons{:}N \times L \to L\}$.
2. An exception is related to a data type, but we want to be able to distinguish several exceptions on the same data type. We use indices for that: an *exception constructor* (or simply an exception) is the name of a data type, possibly indexed by a natural number. Data types indexed by distinct natural numbers are considered as distinct exceptions: for an example, see section 3.3. In the rest of the paper, if α is an exception and D is a data type, "α is an exception of type D" means $\alpha = D$ or $\alpha = D_i$.
3. In the rest of the paper, D, F represents data types, α, β exceptions and L a set (possibly empty) of exceptions. $L + \alpha$ represents the set $L \cup \{\alpha\}$ in case $\alpha \notin L$.

Definition 2.1
The set of EX_2 terms (also called λ-terms) is given by the following grammar:

$$T = x|\ \lambda x\ T|\ (T\ \ T)|\ \varepsilon_\alpha\langle T\rangle|\ \tau_\alpha\langle T, T\rangle$$

where x (resp. α) ranges over variables (resp. exceptions).

Comments and notations

- To help understand the meaning of the new symbols, we give the same example written both in EX_2 (section 3.1) and CAML (section 3.2). To be short $\varepsilon_\alpha\langle T\rangle$ is the way to raise an exception and $\tau_\alpha\langle T_1, T_2\rangle$ catches the exception in T_1 (if any) using the filter T_2.
- As usual, $(u\ v)$ will often be written $u\ v$ and $((u\ t_1)\cdots t_n)$ as $u\ t_1 \ldots t_n$.

Definition 2.2

1. We define the reduction δ as the least congruence satisfying the following reduction rules (the left member of a rule is called a redex and the right member the reduct):

$$\begin{array}{lll} (\lambda x\ u\ v) & \to_\delta & u[x{:=}v] \\ (\varepsilon_\alpha\langle u\rangle\ v) & \to_\delta & \varepsilon_\alpha\langle u\rangle \\ \tau_\alpha\langle \lambda x\, u, v\rangle & \to_\delta & \lambda x\, u \\ \tau_\alpha\langle \varepsilon_\alpha\langle u\rangle, v\rangle & \to_\delta & (v\ u) \\ \tau_\alpha\langle \varepsilon_\beta\langle u\rangle, v\rangle & \to_\delta & \varepsilon_\beta\langle u\rangle \qquad \text{for } \alpha \neq \beta \end{array}$$

2. $t \to_\delta^* t'$ if t' is obtained from t by reducing some (possibly zero) δ-redexes.
3. A λ-term is normal if it contains no δ-redex.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

2.2 The type system

As already mentioned, the typed calculus is an extension of AF_2. For self-containment, we recall the types of AF_2.

Definition 2.3 (Types of AF_2)

1. First order terms are built from variables, constant and function symbols.
2. The set $\mathscr{F}$ of formulas is defined by the following rules.
   - If X is a n-ary predicate variable and $t_1, \cdots, t_n$ are first order terms then $X(t_1, \cdots, t_n) \in \mathscr{F}$.
   - If $F, G \in \mathscr{F}$, then $(F \to G) \in \mathscr{F}$.
   - If $F \in \mathscr{F}$ and x (resp. X) is a first order variable (resp. predicate variable) $\forall x F \in \mathscr{F}$ (resp. $\forall X F \in \mathscr{F}$).

Definition 2.4 (Types of EX_2)

1. For each exception α, let r_α (resp. t_α, cas_α) be a new unary (resp. binary, ternary) function symbol. Actually, t_α also is a kind of binder (see the comment below).
2. For first order terms, we also allow the new function symbols.
3. For formulas in $\mathscr{F}$, we also allow the following atomic formulas:
   - If t is a first order term and α is a exception, then $E^\alpha[t] \in \mathscr{F}$.
   - If t is a first order term, L is a set of exceptions and D a data type, then $D^L[t] \in \mathscr{F}$.
4. The set of EX_2 types is the quotient of $\mathscr{F}$ by $\simeq$ where $\simeq$ is the smallest congruence by which, for each data type D, $D^\emptyset[t]$ is related to the formula which, in AF_2, defines the data type D.

Comments and notations

- The function symbols r_α, t_α and cas_α deal with exceptions at the level of first order terms: r_α to build exceptions, t_α to catch exceptions and cas_α to give names to terms which may depend on the value – exception or not – of another term. See also Definition 2.5.
- The fact that types are quotient of formulas means, for example, that $N^\emptyset[t]$ and $\forall X(X\mathbf{0}, \forall y(Xy \to X\boldsymbol{s}(y)) \to Xt)$ are considered as the same type.
- Note that t_α is not, strictly speaking, a binary symbol because it binds a variable of the second argument: this will be denoted by $t_\alpha(a, x \to b)$ instead of $t_\alpha(a, b)$. Intuitively, $x \to b$ represents the function that maps x to b. This is, however, not a problem because such a function cannot be applied to a term to build another term. Note that we consider $t_\alpha(a, x \to b)$ and $t_\alpha(a, y \to b[x := y])$ represent the same term.
- The intuition for $E^\alpha[t]$ and $D^L[t]$ is the following: if α is an exception of type D, $E^\alpha[t]$ means that t is an exceptional value of type D. From the logical point of view, $D^{L+\alpha}[t]$ can be understood as the disjunction $D^L[t] \vee E^\alpha[t]$ and $D^\emptyset[t]$ by $D[t]$. From the computational point of view, this is a union type that can

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

Table 1. *Typing rules*

Name	Conditions	Conclusion
ax	Γ contains $x : A$	$\Gamma \vdash x : A$
$\rightarrow_i$	$\Gamma, x : B \vdash u : C$	$\Gamma \vdash \lambda x\, u : B \rightarrow C$
$\rightarrow_e$	$\Gamma \vdash u : B \rightarrow C$ and $\Gamma \vdash v : B$	$\Gamma \vdash (u\ v) : C$
$\forall_{i,1}$	$\Gamma \vdash u : A$	
	and x does not occur free in Γ	$\Gamma \vdash u : \forall x A$
$\forall_{e,1}$	$\Gamma \vdash u : \forall x A$ and a is a term	$\Gamma \vdash u : A[x{:=}a]$
$\forall_{i,2}$	$\Gamma \vdash u : A$	
	and X does not occur free in Γ	$\Gamma \vdash u : \forall X A$
$\forall_{e,2}$	$\Gamma \vdash u : \forall X A$ and F is a formula	$\Gamma \vdash u : A[X{:=}F]$
eq	$\Gamma \vdash u : A[x{:=}a_1]$	
	and $l(\Gamma) \Vdash a_1 = a_2$	$\Gamma \vdash u : A[x{:=}a_2]$
exc	$\Gamma \vdash u : D[a]$ and α is of type D	$\Gamma \vdash \varepsilon_\alpha\langle u\rangle : E^\alpha[r_\alpha(a)]$
$prop$	$\Gamma \vdash u : E^\alpha[a]$ and T is a type	$\Gamma \vdash u : T \rightarrow E^\alpha[a]$
$\vee_{i,1}$	$\Gamma \vdash u : E^\alpha[a]$	$\Gamma \vdash u : D^\alpha[a]$
$\vee_{i,2}$	$\Gamma \vdash u : D^L[a]$	$\Gamma \vdash u : D^{L+\alpha}[a]$
$\vee_e$	$\Gamma, x : D^L[a] \vdash u : T$	
	and $\Gamma, x : E^\alpha[a] \vdash u : T$	
	and $\Gamma \vdash v : D^{L+\alpha}[a]$	$\Gamma \vdash u[x{:=}v] : T$
try	$\Gamma \vdash u : D^{L+\alpha}[a]$	
	and $\Gamma \vdash v : \forall x(F[x] \rightarrow D^L[b])$	
	and α is of type F	$\Gamma \vdash \tau_\alpha\langle u, v\rangle : D^L[t_\alpha(a, x \rightarrow b)]$

be seen as the locative form of disjunction according to Girard's terminology, and our work points out an interesting use of union types. Note that union types are more less frequent in the literature than intersection types. See the typing rules (Table 1), and the remarks after Definition 2.6.

- Though a type is the equivalence class of a formula, in the rest of this paper we use indifferently the words formula and type.
- As usual, $A_1 \rightarrow (A_2 \rightarrow (\cdots (A_k \rightarrow B)\cdots))$ will also be written $A_1, A_2, \cdots, A_k \rightarrow B$. $\bot$ in an abreviation for $\forall X X$ and $\neg F$ for $F \rightarrow \bot$.

Definition 2.5

- An equation is (the universal closure of) a formula of the form $u = v$ i.e. $\forall X(Xu \rightarrow Xv)$. A specification is a set of equations.
- Let $\mathscr{A}x$ be the set of the following (conditional) equations:

$$
\begin{array}{lll}
cas_E : E^\alpha[x] \rightarrow cas_\alpha(x, y, z) = y & & \\
cas_D : D^L[x] \rightarrow cas_\alpha(x, y, z) = z & & (\text{for } \alpha \notin L). \\
trap_E : D^\emptyset[x] \rightarrow t_\alpha(r_\alpha(x), y \rightarrow b) = b[y := x] & & (\alpha \text{ is of type } D) \\
trap_D : D^L[x] \rightarrow t_\alpha(x, y \rightarrow b) = x & & (\text{for } \alpha \notin L).
\end{array}
$$

Comments

In these equations, the symbol $=$ denotes the usual second-order encoding of equality, i.e. $\forall X(Xa \rightarrow Xb)$. These equations simply say that, whether we know if x is an exception or not, we know the value of $cas_\alpha(x, y, z)$ and $t_\alpha(x, y \rightarrow b)$.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

Notations

- A context Γ is a function that assigns types to λ-variables. It will be denoted by $x_1{:}A_1, \cdots, x_n{:}A_n$.
- If $\Gamma = x_1{:}A_1, \cdots, x_n{:}A_n$ is a context, the set of types $A_1, A_2, \cdots, A_n$ will be denoted $l(\Gamma)$ and called the logical content of Γ.
- Let r be a typing rule of Table 1. The logical rule $l(r)$ is obtained from r by "forgetting" the algorithmic content. For example, $l(\rightarrow_i)$ is the rule: if $\Gamma, B \Vdash C$ then $\Gamma \Vdash B \rightarrow C$.

Definition 2.6
Let $\mathscr{E}$ be a specification.

1. Let t be a λ-term and Γ be a context. t is typable of type A in the context Γ (with respect to $\mathscr{E}$) if the judgement $\Gamma \vdash_{\mathscr{E}} t{:}A$ can be obtained by using the typing rules of Table 1. Note that the rule (eq) uses the logical notion of consequence $l(\Gamma) \Vdash u = v$ defined in the next point.
2. Let F (resp. $\mathscr{A}$) be a formula (resp. a finite set of formulas). F is a logical consequence of $\mathscr{A}$ (with respect to $\mathscr{E}$) if F is obtained from $\mathscr{A} \cup \mathscr{A}\, x \cup \mathscr{E}$ by using the logical rules obtained from all the typing rules except the rule eq. This is denoted by $\mathscr{A} \Vdash_{\mathscr{E}} F$.

Remarks

1. *Typing rules*
   - For simplicity of notation, we forget the subscript $\mathscr{E}$ when it is clear from the context.
   - The typing rules from (ax) to $(\forall_{e,2})$ are those of AF_2 (Krivine, 1990).
   - The rule (eq) of AF_2 has been slightly modified to allow equalities with typing conditions.
   - The rule (exc) builds an exceptional value from a λ-term whose type is a data type.
   - The rule $(prop)$ ensures the propagation of exceptions in head position: if $\Gamma \vdash u : E^{\alpha}[a]$ and $\Gamma \vdash v : T$, we have $\Gamma \vdash (u\, v) : E^{\alpha}[a]$, where T is any type. We could have given a general version of this rule: $\Gamma \vdash u : T_1, \ldots, T_n \rightarrow E^{\alpha}[a]$. It seems that this n-ary propagation rule is not a consequence of the given rule (although it can be mimicked by performing η-expansions). Even though this more liberal rule is aesthetically more pleasant, we choose not to do so both for simplicity and because we do not really need it.
   - The rules $(\vee_{i,1})$, $(\vee_{i,2})$ and $(\vee_e)$ show that $D^{L+\alpha}[t]$ is more a union of $D^L[t]$ and $E^{\alpha}[t]$ than a disjunction since, in the premises, the same proof-term must appear.
   - These typing rules have the following consequence: if $\Gamma, x : D^L[a] \vdash u : T$ and $\Gamma, x : E^{\alpha}[a] \vdash u : T$ then $\Gamma, x : D^{L+\alpha}[a] \vdash u : T$.
   - The rule (try) also is an elimination rule for $\vee$, but rather unusual.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

2. *The use of specifications*

- First note that the rule $l(eq)$ has been ommited in the logical rules because it is useless: this rule follows immediately from the definition of $u = v$.
- $\mathscr{A}x$ is a schemata of axioms: the free variables can be instantiated by any first order terms.
- Finally, note that in the rule of AF_2 corresponding to eq, only the instances of the equations in $\mathscr{E}$ are allowed, whereas in the eq rule of EX_2, we allow any equation that can be proved in the system.

2.3 Reduction rules and cut elimination

The reduction rules on typed λ-terms correspond to cut elimination on the logical side. For example, the reduction of $\tau_\alpha\langle\varepsilon_\alpha\langle u\rangle, v\rangle$ into $(v\ u)$ corresponds to the transformation of the proof (where α is an exception of type F):

$$\dfrac{\dfrac{\dfrac{\Gamma\vdash u : F[a]}{\Gamma\vdash\varepsilon_\alpha\langle u\rangle : E^\alpha[r_\alpha(a)]}\,(exc)}{\Gamma\vdash\varepsilon_\alpha\langle u\rangle : D^{L+\alpha}[r_\alpha(a)]}\,(\vee_i) \qquad \Gamma\vdash v : \forall x(F[x]\to D^L[f(x)]}{\Gamma\vdash\tau_\alpha\langle\varepsilon_\alpha\langle u\rangle, v\rangle : D^L[t_\alpha(r_\alpha(a), f)]}\,(try)$$

into the proof

$$\dfrac{\dfrac{\Gamma\vdash v : F[a]\to D^L[f(a)] \qquad \Gamma\vdash u : F[a]}{\Gamma\vdash(v\ u) : D^L[f(a)]}\,(app) \qquad F[a]\to f(a) = t_\alpha(r_\alpha(a), f)\ (trap_E)}{\Gamma\vdash(v\ u) : D^L[t_\alpha(r_\alpha(a), f)]}\,(eq)$$

3 Examples

In this section we first give a typical example of the use of exceptions. Let BT be the data type of binary trees whose leaves contain natural numbers. We want to compute the product of the leaves in such a way that, when a zero is found, the answer is given without looking at the remaining leaves. Note that in a lazy language such as Haskell it is possible to use the same shortcut without exceptions.

We then give a more elaborate example which shows that, to import a procedure, it is enough to know its specification, i.e. its type.

We denote by $N[x]$ the formula $\forall X(X\mathbf{0}, \forall y(Xy\to X\boldsymbol{s}(y))\to Xx)$ and by $Bool[x]$ the formula $\forall X(Xtrue\to Xfalse\to Xx)$.

We use the storage operators introduced by Krivine (1990a) to simulate the "call by value" in the "call by name" strategy. For example, let $\delta = \lambda f\,(f\ \underline{0})$ and $G = \lambda x\,\lambda y\,(x\ \lambda z\,(y\ (\underline{S}\ z)))$, where $\underline{0}$ (resp. $\underline{S}$) is any λ-term of type $N[\mathbf{0}]$ (resp. $\forall x(N[x]\to N[\boldsymbol{s}(x)])$). Then $T = \lambda n(n\ \delta\ G)$ is a storage operator for N. The storage operators are used here to force the propagation of exceptions.

Since, in the following examples, we only use terms such as $cas_\alpha(x, x, t)$, to simplify notations $Cas_\alpha(x, t)$ will denote $cas_\alpha(x, x, t)$.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

3.1 The product of the leaves in a binary tree

BT is defined by the formula

$$BT[x] : \forall X(\forall l \forall r(Xl, Xr \to Xtree(l,r)), \forall n(N[n] \to Xleaf(n)) \to Xx).$$

The product h is defined by the specification

$$h(leaf(n)) = n \text{ and } h(tree(l,r)) = mult(h(l), h(r))$$

where *mult* computes the product of two natural numbers.

It is easy to give a typed λ-term computing h. However, by using exceptions, we can get a more efficient term: if a recursive call finds a leaf which contains 0, the other parts of the tree do not have to be examined, since we know the result is 0. The idea of the faster algorithm is thus: if a leaf contains zero, return an exception. This exception will be propagated, and finally caught (to give 0). This new function h' is defined by the following specification:

$$f(leaf(n)) = if(test(n), r_\alpha(\mathbf{0}), n)$$
$$f(tree(l,r)) = prod(f(l), f(r))$$
$$h'(a) = t_\alpha(f(a), x \to x)$$

where the auxiliary functions are defined by the following specification:

$$prod(x,y) = Cas_\alpha(x, Cas_\alpha(y, mult(x,y))))$$
$$if(true,a,b) = a \text{ and } if(false,a,b) = b$$
$$test(\mathbf{0}) = true \text{ and } test(\boldsymbol{s}(x)) = false.$$

Note that we wrote $f(tree(l,r)) = prod(f(l), f(r))$, instead of $f(tree(l,r)) = mult(f(l), f(r))$ as would be expected (and is actually done in CAML). This is because EX_2 is governed by call by name evaluation (unlike CAML). Instead of using the evaluation mode of CAML, this makes the definition more explicit, even though it is a bit less aesthetic.

Proposition 3.1
Assume $\vdash P : \forall x \forall y(N[x], N[y] \to N[mult(x,y)])$ and $\vdash Z : \forall n(N[n] \to Bool[test(n)])$.
Let $V = \lambda l\, \lambda r\, (T\ l\ (T\ r\ \lambda x\, \lambda y\, (P\ y\ x)))$,
$U = \lambda n\, ((Z\ n)\ \varepsilon_\alpha \langle \underline{0} \rangle\ n)$ and $Prod = \lambda a\, \tau_\alpha \langle (a\ V\ U), \lambda z\, z \rangle$.
Then $\vdash Prod : \forall a(BT[a] \to N[h(a)])$.

The proof that $\vdash Prod : \forall a(BT[a] \to N[h'(a)])$ is a rather standard typing exercice (see the appendix). The result follows from (1) $\Vdash \forall a(BT[a] \to h'(a) = h(a))$ using (eq).

To prove (1), it seems to be necessary to prove (2) $\Vdash \forall x(E^\alpha[x] \to \neg D[x])$ but this is impossible in our logic, the intuitive semantic argument is the following: it is easy to have a model with one single point and, in this model, the result is not true!

To prove (2), an additional axiom is necessary, that we call *the plurality axiom*: $\neg(\mathbf{0} = \boldsymbol{s}(\mathbf{0}))$. This axiom (which is also necessary in AF_2), simply ensures that there are at least two points in the model. Note that, if there is only one point in the model, (1) is trivial. Thus, if we could distinguish between the two cases (either there is only one point or not) we would not need this new axiom, but this needs the axiom $A \vee \neg A$, i.e. classical logic, and we shall see in section 4 that the logic of our system is... intuitionistic logic.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

3.2 The same example in CAML

To help readers familiar with the use of exceptions in computer science, we give below the corresponding program written in CAML, a functional language (Leroy & Weis, 1993) of the ML family with exception mechanisms, and where types are automatically generated. Readers will note that the CAML program and the EX_2 program are very similar.

We first recall briefly the exception mechanism in CAML.

Declare an exception. There is a predefined type `exn` which behaves essentially as any type. The terms of type `exn` are called exceptions.

It is possible to add new constructors for the type `exn`: for example, the instruction `exception alpha of int` creates a constructor `alpha` of type `int -> exn`.

Raise an exception. The predefined function `raise` has type `exn -> a` (for any type a). It is used to raise exceptions:

```
1 + raise (alpha (2+3));;
# Uncaught exception: alpha 5
```

When the expression `raise (alpha 5)` is evaluated, the exception is propagated. If it is not caught (see below), the computation stops at the global level and the exceptional value is printed.

Catch an exception. The predefined function `try e with filter` is used to catch an exception: if `e` is evaluated without giving an exception, the value of `e` is returned. Otherwise, the exceptional value is filtered through clauses of `filter`.

Example. The type of binary trees is defined by:

```
type tree = LEAF of int | NODE of tree * tree ;;
# Type tree defined.
```

The product is defined by:

```
let rec prod = function
  (LEAF n) -> n | NODE (l,r) -> (prod l) * (prod r) ;;
# prod : tree -> int = <fun>
prod (NODE(NODE((LEAF 1), (LEAF 0)),(LEAF 2))) ;;
# - : int = 0
```

The faster algorithm requires two functions: `fastprod1` returns an exception if a leaf contains 0. `fastprod` catches the exception and returns 0.

```
exception alpha of int;;
# Exception alpha defined
let rec fastprod1 = function
 (LEAF n) -> if (n = 0) then (raise (alpha 0)) else n
| NODE (l, r) -> (fastprod1 l) * (fastprod1 r) ;;
 # fastprod1 : tree -> int = <fun>
```

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

Note that, despite the type computed by Caml for the function `fastprod1`, this function may raise an uncaught exception:

```
fastprod1 (NODE(NODE((LEAF 1), (LEAF 0)), (LEAF 2))) ;;
 # Uncaught exception: alpha 0

let fastprod a = try (fastprod1 a) with (alpha x) -> x ;;
# fastprod : tree -> int = <fun>

fastprod (NODE(NODE((LEAF 1), (LEAF 0)), (LEAF 2))) ;;
#- : int = 0
```

3.3 Search in diaries

This example shows an important property of EX_2: to use an imported procedure, it is enough to know its specification, i.e. its type.

Assume we have a diary, i.e. a list of address books each one being a list of pairs (name, phone number). We have a program (call it g) which, given a name, returns the corresponding phone number, if the name is present in the address book and an exception otherwise. We want a program which searches in the diary by examining successively each of the address books.

Assume, for simplicity, that names and phone numbers are natural numbers. Let g be a function that returns a natural number (the phone number) if the name is found and $r_\beta(\mathbf{0})$ otherwise (where β is an exception of type N). The natural number 0 returned by g in case the name is not found will not be used, and is thus arbitrary.

The desired function is defined by the following specification (where cons represents the addition of an element at the beginning of a list both for address books and diaries and α is an exception of type N).

$$f(nil, n) = r_\alpha(\mathbf{0}) \text{ and } f(cons(l, q), n) = t_\beta(g(l, n), x \to f(q, n))$$

It returns the phone number if the name is found in the diary and an exception otherwise. Note that we use two exceptions: the first one β for the exception raised by the function g and the second one α for the exception raised by the function f.

Let C denote the type of pairs of natural numbers. The type $AdB[x]$ (resp. $Di[x]$) of address books (resp. diaries) is given by

$$AdB[x] : \forall X(\forall y \forall z(C[y], Xz \to Xcons(y, z)), Xnil \to Xx)$$
$$Di[x] : \forall X(\forall y \forall z(AdB[y], Xz \to Xcons(y, z)), Xnil \to Xx)$$

These types are the usual codings of term algebras: for example, AdB represents the least set that contains the empty list *nil* and that is closed by the *cons* operation: if y is in C, i.e. y is a pair of natural numbers and z is in Adb, then $cons(y, z)$ is in Adb.

Proposition 3.2
Assume $\vdash G : \forall l \forall n(AdB[l], N[n] \to N^\beta[g(l, n)])$.
Let $t = \lambda a\, \lambda n\, (a\ V\ U)$ where $V = \lambda l\, \lambda q\, \tau_\beta \langle (G\ l\ n), \lambda x\, q \rangle$ and $U = \varepsilon_\alpha \langle \underline{0} \rangle$.
Then $\vdash t : \forall a \forall n(Di[a], N[n] \to N^\alpha[f(a, n)])$.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

4 The main properties of the calculus

4.1 Confluence

Theorem 4.1
The δ-reduction is confluent (even in the untyped system).

Proof
This is proved by using the standard method of parallel reduction. It can also be shown that the δ-reduction is an orthogonal combinatory reduction system (see van Oostrom *et al.*, 1993), and this implies confluence. □

4.2 Strong normalization

Theorem 4.2
EX_2 is strongly normalizing.

The proof uses the standard method of reducibility candidates. We only give here the main ideas. More details are given in the appendix.

We first define a new system which consists in forgetting the first order in EX_2: this system, that we call FX_2 is thus system F (Lafont *et al.*, 1989) with the addition of types for second order exceptions. The strong normalization of EX_2 follows immediately from the one of FX_2.

For FX_2, the proof follows the one of system F: we first give the definition of the reducibility candidates and the notion of interpretation. Finally, we prove the adequation theorem from which the strong normalization follows immediately. As usual, a candidate of reducibility is a saturated set A such that $\mathcal{N}_0 \subset A \subset \mathcal{N}$ where $\mathcal{N}$ is the set of strongly normalizing λ-terms and $\mathcal{N}_0$ is the set of λ-terms of the form $(x\ t_1 \ldots t_n)$ where x is a variable and $t_1, \cdots, t_n \in \mathcal{N}$. The main difficulty is the definition of the notion of saturation: this is not immediate because it seems to need a loop.

4.3 Subject reduction

Usually, subject reduction is easy to prove. Here the unusual rule $(\vee_e)$ causes some problems. A redex occurring in v may be duplicated in $u[x:=v]$. Since these occurrences come from the same proof, subject reduction seems to need that these redexes are reduced in the same way.

Look at the following example. Let $v = (z\ (\lambda t\, t\ \varepsilon_\alpha\langle\underline{0}\rangle)\ \underline{1})$ and $v' = (z\ \varepsilon_\alpha\langle\underline{0}\rangle\ \underline{1})$. Let f be the function $x \to x^x$ and $t = \mathit{If}(z, r_\alpha(\mathbf{0}), s(\mathbf{0}))$ where the function If is defined by $\mathit{If}(true, a, b) = a$ and $\mathit{If}(false, a, b) = b$.

The terms $(v\ v)$ and $(v'\ v')$ are typable of type $N^\alpha[Cas_\alpha(t, f(t))]$ in the context $\Gamma = z : Bool[z]$:

- $\Gamma, y : N[x] \vdash (y\ y) : N[f(x)]$ and thus (by $\vee_{i,1}$ and cas_D) $\Gamma, y : N[x] \vdash (y\ y) : N^\alpha[Cas_\alpha(x, f(x))]$.
- $\Gamma, y : E^\alpha[x] \vdash (y\ y) : E^\alpha[x]$, and thus (by $\vee_{i,2}$ and cas_E) $\Gamma, y : E^\alpha[x] \vdash (y\ y) : N^\alpha[Cas_\alpha(x, f(x))]$.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

- It is easy to check that $\Gamma \vdash v : N^{\alpha}[t]$ and $\Gamma \vdash v' : N^{\alpha}[t]$ and thus the rule $\vee_e$ gives the result.

It is clear that $(v\ v) \rightarrow_{\delta}^{*} (v\ v')$ and $(v\ v) \rightarrow_{\delta}^{*} (v'\ v)$. However, we do not know how to type $(v\ v')$ or $(v'\ v)$.

We only show subject reduction for a parallel reduction, which consists in reducing simultaneously the redexes of v duplicated in $u[x:=v]$. The same kind of problem appears in Barbanera & Berardi (1993) and Pierce (1990), and was solved similarly in Barbanera & Berardi (1993).

Open question
We do not know whether the (full) subject reduction holds for the δ-reduction.

Definition 4.3
Let t be a λ-term.

1. Let E be a set of sub-terms of t. E is *primary* if all the elements of E are syntactically identical.
2. $t \rightarrow_{\|} t'$ if t' is obtained from t by δ-reducing each element of some primary set of redexes of t.
3. $t \rightarrow_{\|}^{*} t'$ if t' is obtained from t by some (possibly zero) steps of $\|$-reduction.

Note that, since a term is only defined up to α-equivalence, the syntactical identity mentioned above also is up to α-equivalence. Also note that two sub-terms in a primary set, since they are identical, are disjoint, and thus the $\|$-reduction is well defined and the order in which the δ-reductions are done does not matter.

Theorem 4.4

1. Assume $t \rightarrow_{\|}^{*} t'$ and $\Gamma \vdash t : A$. Then $\Gamma \vdash t' : A$.
2. Assume $\Gamma \vdash t : A$ and $\bar{t}$ is the δ-normal form of t. Then $\Gamma \vdash \bar{t} : A$.

Remark
The previous weak subject reduction property is enough for us to use EX_2 as a proof system for programming: it ensures that the result of a program, once completely reduced, will have the right type.

4.4 Programming with EX_2

The following theorem (which characterizes λ-terms whose type is N or an exception in the empty context) implies that the main property of AF_2 is preserved: if $\vdash_{\mathscr{E}} t : \forall x(N[x] \rightarrow N[f(x)])$ and the specification $\mathscr{E}$ is consistent, then t computes the function f (*i.e.* for all natural number n, $(t\ \underline{n}) \rightarrow_{\delta}^{*} \underline{f(n)}$). The result is given for N but the same holds for any data type. To prove the consistency we have to build a model (in a sense near from the use of this term in AF_2) but, in fact, it is enough to prove that the equations are consistent on the concerned data types (see Mounier (1999) for more details).

Note that this also implies that, if the type of a program is N, then its execution cannot raise an uncaught exception.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

The use of the plurality axiom (see section 3.1) introduces a problem. We cannot *realize* it by, typically, the identity, because then we would have to prove the normalization for the extended system but our proof does not work, since when we erase first order terms this axiom becomes: $\forall X(X \to X) \to \bot$, and this is false. We thus introduce a new constant to type it. The programs we get are not closed, but it is not difficult to see that, because of its type, this constant cannot appear in head position of a term whose type is a data type, and thus it is useless in the computation.

Theorem 4.5
Assume α is an exception of type N.

1. If $\vdash t : N[a]$, there is a natural number n such that $t \to_\delta^* \underline{n} = \lambda x\, \lambda f\, (f^n\ x)$ and $\Vdash a = \boldsymbol{s}^n(\mathbf{0})$.
2. If $\vdash t : E^\alpha[a]$ then $t \to_\delta^* \varepsilon_\alpha\langle u\rangle$ for some u such that $\vdash u : N[b]$ and $\Vdash a = r_\alpha(b)$.
3. If $\vdash t : N^\alpha[a]$ then:
   - either $t \to_\delta^* \underline{n}$ for some n such that $\Vdash a = \boldsymbol{s}^n(\mathbf{0})$.
   - or $t \to_\delta^* \varepsilon_\alpha\langle\underline{n}\rangle$ for some n such that $\Vdash a = r_\alpha(\boldsymbol{s}^n(\mathbf{0}))$.

4.5 The logic of EX_2

The next theorem shows that the logic of EX_2 is intuitionistic logic. Denote by $\Vdash_F$ (resp. $\Vdash_{EX_2}$, $\Vdash_{FX_2}$) the notion of logical consequence associated to system F (resp. EX_2 and FX_2).

Theorem 4.6
Let A (resp. $\mathscr{A}$) be a formula (resp. a finite set of formulas) of system F. $\mathscr{A} \Vdash_{EX_2} A$ if and only if $\mathscr{A} \Vdash_F A$.

This is proved by giving a translation from formulas of EX_2 into formulas of Girard's system F (Lafont *et al.*, 1989).

5 Conclusion and future work

We have introduced a new typed λ-calculus EX_2 by adding to AF_2 a mechanism for handling exceptions. We have proved the expected properties of the system: confluence, strong normalization and preservation of the type by reduction to the normal form. We have shown that the logic of EX_2 is intuitionistic logic.

The fundamental paradigm of AF_2 ("a proof of the totality of a function is an implementation of that function") still holds for exceptions ("a proof that a function raises an exception is... an exception"). Moreover, if a program is well typed, we have the guarantee that its execution will not raise an uncaught exception. This is not the case in CAML. Nevertheless, some variants of the type system of Caml have been proposed to solve this problem (see, for instance, Pessaux (1999)). It would be interesting to study the connections between these pure "typing" approaches with ours, which is more oriented towards proofs.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

The proof terms we obtain with our system are much more efficient than what would be obtained by simply performing some kind of monadic transformation (using the exception monad). This is due to the fact that (a) we use exceptions even at the level of proofs, and (b) that $D^L[t]$ works like an union type rather than like a disjunction (that would be more or less the result of a monadic transformation).

The program *Propre* introduced by Manoury & Simonot (1993) is able to construct automatically a proof of totality of a very large class of functions defined by equations in AF_2. This program is already implemented in the proof assistant CoQ. It will be very interesting to extend this program to EX_2.

6 Appendix

6.1 Proof of propositions 3.1 and 3.2

Lemma 6.1 gives the main properties of the storage operators. Lemma 6.3 shows how, by using a storage operator, we can extend a function from $N \times N$ into N to exceptional values.

Definition 6.1

- Let O be a new 0-ary predicate symbol and, for each formula F, denote $F \to O$ by $\neg F$.
- Let $N^*[x] = \forall X(\neg X\mathbf{0}, \forall y(\neg Xy \to \neg X\boldsymbol{s}(y)) \to \neg Xx)$.

Lemma 6.2
Let α be a exception.

1. $\vdash T : \forall O \forall x(N^*[x] \to \neg\neg N[x])$.
2. If $\Gamma \vdash t : N[n]$ and $\Gamma \vdash F : \forall x(N[x] \to N[f(x)])$ then $\Gamma \vdash (T\ t\ F) : N[f(n)]$.
3. $\vdash T : \forall x(E^\alpha[x] \to E^\alpha[x])$.
4. If $\Gamma \vdash t_1 : N[a]$ and $\Gamma \vdash t_2 : E^\alpha[b]$ then $\Gamma \vdash (T\ t_1\ t_2) : E^\alpha[b]$.

Lemma 6.3
Let f be a function from $N \times N$ into N and assume that F is a λ-term such that $\vdash F : \forall x \forall y(N[y], N[x] \to N[f(x, y)])$.

Let g be the function defined by $g(x, y) = Cas_\alpha(x, Cas_\beta(y, f(x, y)))$ and let $t = \lambda x\, \lambda y\, (T\ x\ (T\ y\ F))$. Then $\vdash t : \forall x \forall y(N^\alpha[x], N^\beta[y] \to N^{\alpha,\beta}[g(x, y)])$.

Proof
First note that the order of the arguments in the type of t has been reversed with respect to the one of F. This is due to the use of T and could be easily repaired by changing slightly T. The same change appears in Proposition 3.1.

1. $\Gamma \vdash u : N^{\alpha,\beta}[g(x, y)]$ where $\Gamma = x : N[x], y : N^\beta[y]$ and $u = (T\ x\ (T\ y\ F))$. This is proved in the following way: let $\Gamma_1 = x : N[x], y : N[y]$ and $\Gamma_2 = x : N[x], y : E^\beta[y]$.

   (a) $\Gamma_1 \vdash u : N^\beta[Cas_\beta(y, f(x, y))]$. From

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

- $\vdash T : N^*[y] \to ((N[y] \to A) \to A)$ with $A = N[x] \to N[f(x,y)]$,
- $x : N[x] \vdash x : N^*[x]$. Use $N[x]$ with $\neg X$.
- $\vdash T : N^*[x] \to ((N[x] \to B) \to B)$ with $B = N[f(x,y)]$.

we get $\Gamma_1 \vdash u : N[f(x,y)]$. By $(\vee_{i,1})$ we get $\Gamma_1 \vdash u : N^\beta[f(x,y)]$ and by (eq), (cas_D) $\Gamma_1 \vdash u : N^\beta[Cas_\beta(y, f(x,y))]$

(b) $\Gamma_2 \vdash u : N^\beta[Cas_\beta(y, f(x,y))]$.
$\Gamma_2 \vdash u : E^\beta[y]$ follows immediately from Lemma 6.1. Again, by $(\vee_{i,2})$ we get $\Gamma_2 \vdash u : N^\beta[y]$ and by (eq), (cas_E) $\Gamma_2 \vdash u : N^\beta[Cas_\beta(y, f(x,y))]$

(c) From (a) and (b) we get $\Gamma \vdash u : N^\beta[Cas_\beta(y, f(x,y))]$, then we get the result by using again $(\vee_{i,2})$, (eq) and (cas_D).

2. $\Delta \vdash u : N^{\alpha,\beta}[g(x,y)]$ where $\Delta = x : E^\alpha[x], y : N^\beta[y]$. This is proved in the following way: it follows easily from Lemma 6.2 by using $(prop)$ and (app) that $\Delta \vdash u : E^\alpha[x]$. The result follows then by using $(\vee_{i,1})$, $(\vee_{i,2})$, (eq) and (cas_E).
3. The result follows from (1) and (2). □

6.1.1 Proof of Proposition 3.1

1. We first prove $a : BT[a] \vdash (a\ V\ U) : N^\alpha[f(a)]$.
   - $\vdash V : \forall x \forall y (N^\alpha[x], N^\alpha[y] \to N^\alpha[prod(x,y)]$ is an immediate consequence of lemma 6.1 with P and $mult$.
   - We get $\vdash V : \forall l \forall r (N^\alpha[f(l)], N^\alpha[f(r)] \to N^\alpha[f(tree(l,r))]$ by using (eq).
   - $\vdash U : \forall n (N[n] \to N^\alpha[if(test(n), r_\alpha(\mathbf{0}), n)]$ is easy and thus again by (eq) $\vdash U : \forall n (N[n] \to N^\alpha[f(leaf(n))]$.
   - We get the desired result by replacing, in $BT[a]$, X(.) by $N^\alpha[f(.)]$.
2. We get $a : BT[a] \vdash \tau_\alpha \langle (a\ V\ U), \lambda z\, z \rangle : N[t_\alpha(f(a), x \to x)]$ by applying (try) to $a : BT[a] \vdash (a\ V\ U) : N[f(a)]$ and we conclude $a : BT[a] \vdash \tau_\alpha \langle (a\ V\ U), \lambda z\, z \rangle : N[h'(a)]$ by (eq).

Lemma 6.4
Let α be a exception and assume $\mathscr{E}$ contains the plurality axiom. Then, $\Vdash \forall a (BT[a] \to h'(a) = h(a))$

Proof
By induction on a, using a case analysis. Actually, a stronger induction hypothesis is necessary. We prove the following, by simultaneous induction on a: $N[f(a)] \to f(a) = h(a)$, $E^\alpha[f(a)] \to f(a) = r_\alpha(\mathbf{0})$ and $E^\alpha[f(a)] \to h(a) = \mathbf{0}$. We will not detail this proof, which is straightforward, but uses at many points Lemma 6.1.1 below. The result follows then easily. □

Lemma 6.5
Let α be a exception and assume $\mathscr{E}$ contains the plurality axiom. Then, $\Vdash \forall x (E^\alpha[x] \to \neg D[x])$.

Proof
We know (cas_E) that $E^\alpha[x] \Vdash cas_\alpha(x, \mathbf{0}, \boldsymbol{s}(\mathbf{0})) = \mathbf{0}$, and (cas_D), and that $D[x] \Vdash cas_\alpha(x, \mathbf{0}, \boldsymbol{s}(\mathbf{0})) = \boldsymbol{s}(\mathbf{0})$. Since the equality is transitive, $E^\alpha[x], D[x] \Vdash \boldsymbol{s}(\mathbf{0}) = \mathbf{0}$ and the plurality axiom gives $E^\alpha[x], D[x] \Vdash \bot$. □

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

6.1.2 Proof of Proposition 3.2

- Let $\Gamma = l : AdB[l], n : N[n]$. Since $\Gamma \vdash (G\ l\ n) : N^{\beta}[g(l,n)]$, using $(\vee_{i,2})$ we get, $\Gamma \vdash (G\ l\ n) : N^{\alpha,\beta}[g(l,n)]$ and it follows that $\Gamma, z : N^{\alpha}[z] \vdash \tau_{\beta}\langle (G\ l\ n), \lambda x\, z\rangle : N^{\alpha}[t_{\beta}(g(l,n), x \to z)]$ and thus $n : N[n] \vdash V : \forall l \forall q (AdB[l], N^{\alpha}[f(q,n)] \to N^{\alpha}[f(cons(l,q),n)])$.
- It is easy to check that $n : N[n] \vdash U : N^{\alpha}[f(\emptyset, n)]$ and, by induction on a, that $a : Di[a], n : N[n] \vdash (a\ V\ U) : N^{\alpha}[f(a)]$)
- The result follows immediately.

6.2 Proof of Theorem 4.2

6.2.1 The system FX_2

The λ-terms are those of EX_2. The formulas are those of system F with the addition, for each exception α (resp. data type D and finite set L of exceptions) of a constant predicate E^{α} (resp. D^{L}).

The rules are those of system F plus the following rules – we give them the name of the corresponding rules of EX_2 (where D, F are data types, α is a exception and L is a set of exceptions such that $\alpha \notin L$).

(*exc*) If $\Gamma \vdash_F u : D$ then $\Gamma \vdash_F \varepsilon_{\alpha}\langle u\rangle : E^{\alpha}$ where α is an exception of type D.
(*prop*) If $\Gamma \vdash_F u : E^{\alpha}$ then $\Gamma \vdash_F u : T \to E^{\alpha}$ where T is any type.
(*try*) If $\Gamma \vdash_F u : D^{L+\alpha}$ and $\Gamma \vdash_F v : F \to D^{L}$
then $\Gamma \vdash_F \tau_{\alpha}\langle u, v\rangle : D^{L}$ where α is an exception of type F.
$(\vee_{i,1})$ If $\Gamma \vdash_F u : E^{\alpha}$ then $\Gamma \vdash_F u : D^{\alpha}$.
$(\vee_{i,2})$ If $\Gamma \vdash_F u : D^{L}$ then $\Gamma \vdash_F u : D^{L+\alpha}$.
$(\vee_{e})$ If $\Gamma, x : D^{L} \vdash_F u : T$ and $\Gamma, x : E^{\alpha} \vdash_F u : T$ and $\Gamma \vdash_F v : D^{L+\alpha}$ then $\Gamma \vdash_F u[x{:=}v] : T$.

Note that these rules are those of EX_2 without first order and $(\forall_{i,1})$, $(\forall_{e,1})$, (eq) are useless.

Definition 6.6
Let A be a formula of EX_2. Its translation A^0 is the formula of FX_2 defined by

- $X(t_1 \ldots t_n)^0 = X$, $E^{\alpha}[t]^0 = E^{\alpha}$ and $D^{L}[t]^0 = D^{L}$.
- $(\forall x F)^0 = F^0$, $(F \to G)^0 = F^0 \to G^0$ and $(\forall X F)^0 = \forall X F^0$

Lemma 6.7
Assume $\Gamma \vdash t : A$. Then, $\Gamma^0 \vdash_F t : A^0$

6.2.2 Reducibility candidates and interpretations

Definition 6.8

1. Let $\mathscr{N}$ be the set of strongly normalizing λ-terms and $\mathscr{N}_0$ be the set of λ-terms of the form $(x\ t_1 \ldots t_n)$ where x is a variable and $t_1, \cdots, t_n \in \mathscr{N}$.
2. Let $\mathscr{E}^{\alpha}$ be the set of terms in $\mathscr{N}$ that reduce to a term of the form $\varepsilon_{\alpha}\langle u\rangle$.
3. A set A of λ-terms is saturated if it satisfies the following properties:

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

(sat_1) $\forall u, t_1 \ldots t_n, \forall t \in \mathcal{N}$ if $(u[x:=t]\ t_1 \ldots t_n) \in A$ then $(\lambda x\, u\ t\ t_1 \ldots t_n) \in A$.
(sat_2) For each type denotation α, $\forall t_1 \ldots t_n, \forall t, v \in \mathcal{N}$:
if $(t\ t_1 \ldots t_n) \in A$ and $t \notin \mathscr{E}^\alpha$ then $(\tau_\alpha\langle t, v\rangle\ t_1 \ldots t_n) \in A$.
(sat_3) For each type denotation α, $\forall v, t_1 \ldots t_n, \forall t \in \mathscr{E}^\alpha$:
if, for all w such that $t \to_\delta^* \varepsilon_\alpha\langle w\rangle$, $(v\ w\ t_1 \ldots t_n) \in A$
then $(\tau_\alpha\langle t, v\rangle\ t_1 \ldots t_n) \in A$.
(sat_4) $\forall t \in A, \forall t'$ if $t \to_\delta^* t'$ then $t' \in A$.

4. The set $\mathscr{R}$ of reducibility candidates is the set of saturated A such that $\mathcal{N}_0 \subset A \subset \mathcal{N}$.

Remark
The choice of the saturation properties is the difficult point of the proof. In particular, we have to mention the exceptional λ-terms but we cannot use the interpretation $|E^\alpha|$ since the definition of $|E^\alpha|$ needs the notion of reducibility candidates. $\mathscr{E}^\alpha$ may be seen as a first approximation of $|E^\alpha|$.

Proposition 6.9
$\mathcal{N}$ is a reducibility candidate.

We now define the interpretation of a formula: the interpretation of D^α will be the union of the interpretations of D and E^α. We define first the sets used in the definition of the interpretation of E^α.

Definition 6.10

1. Let α be an exception of type D. $\mathscr{F}^\alpha$ is the set of λ-terms $t \in \mathcal{N}$ such that:
   (a) t reduces to a λ-term of the form $\varepsilon_\alpha\langle w\rangle$ where $w \in |D|$
   (b) t does not reduce to a λ-term of the form $\varepsilon_\alpha\langle w\rangle$ where $w \notin |D|$
2. Let $\mathcal{N}_1$ be the least saturated set such that $\mathcal{N}_0 \subset \mathcal{N}_1$.
3. A set A is E-closed if $\forall t \in A, \forall v \in \mathcal{N} : (t\ v) \in A$.

Remark
We believe that condition (a) above implies condition (b) but we have not been able to prove that.

See the remark below for the definition of $|D|$.

Definition 6.11
An interpretation I is a function which assigns to each second order variable X a set $|X|_I$ of λ-terms. Let I be an interpretation. I is extended to a function $T \mapsto |T|_I$ from types to sets of λ-terms in the following way:

1. $|A \to B|_I = |A|_I \to |B|_I$
2. $|\forall X B|_I = \cap_{F \in \mathscr{R}} |A|_{I[X:=F]}$ where $I[X:=F]$ is the interpretation such that $|X|_{I[X:=F]} = F$ and $|Y|_{I[X:=F]} = |Y|_I$ for $Y \neq X$.
3. $|E^\alpha| = \mathcal{N}_1 \cup \mathscr{F}^\alpha$
4. $|D^{L+\alpha}| = |D^L| \cup |E^\alpha|$ where D is a data type, L is a set of exceptions, and α is a exception such that $\alpha \notin L$.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

Remark
The reader might think there is a loop in the previous definitions: the interpretation of D is needed to define $\mathscr{F}^\alpha$ (cf. Definition 6.10) and $\mathscr{F}^\alpha$ is needed to define the interpretation of D (cf. Definition 6.11). This is not a loop because, for a data type D, the definition of $|D|$ uses only the first two rules of Definition 6.11. For example, the interpretation of the data type $N = \forall X(X \to (X \to X) \to X)$ is $\cap_{F \in \mathscr{R}}(F \to (F \to F) \to F)$.

Proposition 6.12
The interpretation of any formula is a candidate of reducibility.

Proof
We first show that $\mathscr{F}^\alpha$ is saturated and E-closed. We then show that $|E^\alpha|$ has the same properties. □

Proposition 6.13
Let I be an interpretation. Assume $\Gamma = x_1{:}A_1, \cdots, x_n{:}A_n \vdash_{\mathscr{F}} u : A$ and $t_i \in |A_i|_I$ $(1 \leqslant i \leqslant n)$. Then $u[x_1{:=}t_1, \cdots, x_n{:=}t_n] \in |A|_I$.

Proof
By induction on $\Gamma \vdash_{\mathscr{F}} u : A$. The saturation properties of the candidates of reducibility have been chosen for that. We only give the cases where the last rule is (*prop*) or (*try*). The other cases are similar. For any term t, t' will represent $t[x_1{:=}t_1, \cdots, x_n{:=}t_n]$:

1. (*prop*) Assume $\Gamma \vdash_{\mathscr{F}} u : E^\alpha$. We have to show that $u' \in |T \to E^\alpha|_I$. By induction hypothesis , $u' \in |E^\alpha|_I$. Since $|E^\alpha|_I$ is E-closed and $|T|_I \subset \mathscr{N}$ we have $|E^\alpha|_I \subset |T|_I \to |E^\alpha|_I = |T \to E^\alpha|_I$.
2. (*try*) Assume $\Gamma \vdash_{\mathscr{F}} u : D^{\beta,\alpha}$ and $\Gamma \vdash_{\mathscr{F}} v : G \to D^\beta$ where α is an exception of type G. By induction hypothesis we have $v' \in |G|_I \to |D^\beta|_I$ and $u' \in |D^{\beta,\alpha}|_I = |D|_I \cup |E^\beta|_I \cup |E^\alpha|_I$.
   - If $u' \in |D|_I$ then $u' \notin \mathscr{E}^\alpha$, and thus $\tau_\alpha\langle u', v'\rangle \in |D|_I$ since v' is strongly normalizable and $|D|_I$ satisfies sat_2.
   - If $u' \in |E^\beta|_I$ the proof is the same as in the previous case, since $|E^\beta|_I$ satisfies sat_2.
   - If $u' \in |E^\alpha|_I$:
     — If $u' \in \mathscr{N}_1$. Since $\mathscr{N}_1$ satisfies sat_2 we have $\tau_\alpha\langle u', v'\rangle \in \mathscr{N}_1 \subset |D|_I \subset |D^\beta|_I$.
     — If $u' \in \mathscr{F}^\alpha$. By definition of $\mathscr{F}^\alpha$, we know that if $u' \to_\delta^* \varepsilon_\alpha\langle a\rangle$ then $a \in |G|_I$ and thus $(v\ a) \in |D^\beta|_I$. Since $|D^\beta|_I$ satisfies sat_3, it follows that $\tau_\alpha\langle u', v'\rangle \in |D^\beta|_I$. □

6.2.3 *Proof of the theorem*

Let I be the interpretation defined by $|X|_I = \mathscr{N}$ for each variable. Since $x_i \in \mathscr{N}_0 \subset |A_i|_I$ Proposition 6.13 shows that if $x_1{:}A_1, \cdots, x_n{:}A_n \vdash_{\mathscr{F}} u : A$, then $u = u[x_1{:=}x_1 \cdots x_n{:=}x_n] \in |A|_I \subset \mathscr{N}$.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

6.3 Proof of Theorem 4.3

It follows easily from the following lemmas. The crucial point is Lemma 6.14:

Lemma 6.14
Assume $u[x:=v] \to_{\parallel}^{*} t'$ and v is not redex-creating (i.e. v is neither $\lambda x.u$ nor $\varepsilon_\alpha\langle u\rangle$ for some u). Then $t' = u'[x:=v']$ with $u \to_{\parallel}^{*} u'$ and $v \to_{\parallel}^{*} v'$.

Lemma 6.15
Let t be a λ-term.

1. If $t \to_\delta t_1$, there is a λ-term t_2 such that $t \to_{\parallel} t_2$ and $t_1 \to_{\parallel} t_2$;
2. If t is strongly normalizable and $\bar{t}$ is the δ-normal form of t then $t \to_{\parallel}^{*} \bar{t}$.

6.4 Proof of Theorem 4.4

We only give a sketch of the proof of the first item of the theorem. The other points are similar.

Let $\vdash t : N[a]$. Then t is strongly normalizable. Its normal form $\bar{t}$ is closed and, by subject reduction, $\vdash \bar{t} : N[a]$. The proof follows the usual one: we show that $\bar{t}$ must be of the form $\lambda x \lambda f\ v$ where $f : \forall y(Xy \to X\boldsymbol{s}(y))$ and $x : X\mathbf{0}$. For that, some lemmas are needed: for example, $\bar{t}$ cannot be $\varepsilon_\alpha\langle u\rangle$.

We then prove, by induction on the complexity of v, that, if $x : X\mathbf{0}, f : \forall y(Xy \to X\boldsymbol{s}(y)) \vdash v : Xa$, then there exists a natural number n such that $v = (f^n\ x)$ and $\Vdash a = \boldsymbol{s}^n(\mathbf{0})$.

6.5 Proof of Theorem 4.5

A formula A of EX_2 is first translated into a formula A^0 of FX_2 by forgetting the first order (as in the proof of strong normalization). Then a formula A of FX_2 is translated into a formula $\overline{A}$ of system F in the following way.

Definition 6.16

1. Let A be a formula of FX_2. $\overline{A}$ is defined by:
   - $\overline{X} = X$ if X is a second order variable, $\overline{A \to B} = \overline{A} \to \overline{B}$ and $\overline{\forall XA} = \forall X\overline{A}$.
   - $\overline{E^\alpha} = \overline{D^\emptyset} = D^0$ if α is an exception of type D.
   - $\overline{D^{L+\alpha}} = \overline{D^L} \vee \overline{E^\alpha}$ where $A \vee B$ is the abbreviation of: $\forall X((A \to X) \to (B \to X) \to X)$.
2. Let A be a formula of EX_2. Denote by A' the formula $\overline{A^0}$.

Since, for a formula A of system F, $A' = A$ the theorem follows immediately from Lemma 6.17 below.

Lemma 6.17

1. Let A (resp. $\mathscr{A}$) be a formula (resp. a finite set of formulas) of EX_2. If $\mathscr{A} \Vdash_{EX_2} A$ then $\mathscr{A}' \Vdash_F A'$.
2. Let A (resp. $\mathscr{A}$) be a formula (resp. a finite set of formulas) of system F. If $\mathscr{A} \Vdash_F A$ then $\mathscr{A} \Vdash_{EX_2} A$.

https://doi.org/10.1017/S0956796804005362 Published online by Cambridge University Press

Proof

The second point is immediate. For the first one, since $\mathscr{A} \Vdash_{EX_2} A$ easily implies $\mathscr{A}^0 \Vdash_{FX_2} A^0$, it is enough to show the result for formulas of FX_2, i.e. it is enough to show that, if $\mathscr{A} \Vdash_{FX_2} A$, then $\overline{\mathscr{A}} \Vdash_F \overline{A}$. This is done by induction on the derivation. The only non-trivial case is when the last rule is (*try*). From $\mathscr{A} \Vdash_{FX_2} D^{L+\alpha}$ and $\mathscr{A} \Vdash_{FX_2} G \to D^L$, we have deduced $\mathscr{A} \Vdash_{FX_2} D^L$ (where α is an exception of type G). Since $\overline{G} = G$ and $\overline{D^{L+\alpha}} = \overline{D^L} \vee \overline{E^\alpha} = \overline{D^L} \vee G$, the induction hypothesis gives $\overline{\mathscr{A}} \Vdash_F \overline{D^L} \vee G$ and $\overline{\mathscr{A}} \Vdash_F G \to \overline{D^L}$ from which we get $\overline{\mathscr{A}} \Vdash_F \overline{D^L}$. □

References

Friedman, D. P., Duba, B., Felleisen, M. and Kohlbecker, E. (1987) A syntactic theory of sequential control. *Theor. Comput. Sci.* **52**, 205–237.

Barbanera, F. and Berardi, S. (1993) Extracting constructive content from classical logic via control-like reductions. *Proc. Int. Conf. Typed Lambda Calculi and Applications: LNCS 664*, pp. 45–59. Springer-Verlag.

de Groote, P. (1995) A simple calculus of exception handling. *Second Int. Conf. on Typed Lambda Calculi and Applications: LNCS 902*, pp. 201–215.

Dezani-Ciancaglini, M., Cardone, F. and de'Liguoro, U. (1994) Combining type disciplines. *Ann. Pure Appl. Logic*, **66**, 197–230.

Griffin, T. (1990) A formulae-as-types notion of control. *Proc. ACM Conf. Principle of Programming Languages*, pp. 47–58. ACM Press.

Lafont, Y., Girard, J. L. and Taylor, P. (1989) *Proofs and types*. Cambridge University Press.

Krivine, J.-L. (1990) *Lambda-Calcul, Types et Modèles*. Masson.

Krivine, J.-L. (1990a) Opérateurs de mise en mémoire et traduction de Gödel. *Archive Math. Logic*, **30**, 241–267.

Krivine, J.-L. (1994) Classical logic, storage operators and second order λ-calculus. *Ann. Pure Appl. Logic*, **68**, 53–78.

Krivine, J. L. and Parigot, M. (1990) Programming with proofs. *Inf. Process. Cybern.* **26**(3), 149–167.

Leivant, D. (1983) Reasoning about functional programs and complexity classes associated with type disciplines. *24th Annual Symp. on Found. of Comp. Sc.*, pp. 460–469.

Leroy, X. and Weis, P. (1993) *Le Langage Caml*. InterEditions.

Manoury, P. and Simonot, M. (1993) *Des Preuves de Totalité de Fonctions comme Synthèse de Programmes*. PhD thesis, Université de Paris VII.

Mounier, G. (1999) *Un λ-calcul intuitionniste avec exceptions*. PhD thesis, Université de Savoie.

Nakano, H. (1994) A constructive logic behind the catch and throw mechanism. *Ann. Pure Appl. Logic*, **69**, 269–301.

Nour, K. (2000) Mixed logic and storage operators. *Archive Math. Logic*, **39**, 261–280.

Parigot, M. (1992) $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. *Proc. Int. Conf. Logic Programming and Automated Reasoning: LNCS 624*, pp. 190–201. Springer-Verlag.

Pessaux, F. (1999) *Détection statique d'exceptions non rattrapées en Objective Caml*. PhD thesis, Université de Paris VI.

Pierce, B. (1990) Preliminary investigation of a calculus with intersection and union types. Internal report, Carnegie Mellon University.

van Oostrom, V., Klop, J. W. and van Raamsdonk, F. (1993) Combinatory reduction systems: introduction and survey. *Theor. Comput. Sci.* **121**, 279–308.