Exceptions considered harmless

Jean-François Monin

France Télécom CNET, LAA/EIA/EVP
Technopôle Anticipa, 2 avenue Pierre Marzin
F-22307 Lannion Cedex, France
monin@lannion.cnet.fr

Program extraction is a well known technique for developing correct functional programs from a constructive proof of their specification. This paper shows how to deal with exceptions in such a framework. We propose a modular (and impredicative) formalization in the calculus of constructions and we illustrate the technique on three examples.

Key words: Exceptions. Type system. Impredicativity. Program extraction. Continuations. Double negation translation.

1 Introduction

In both the imperative and the functional world, control-flow escape mechanisms – typically goto statements and exceptions – are problematic from a verification point of view. The problem is perhaps more important in functional programming: during the design of ML, which was originally the tactics language of LCF, exceptions were considered as an essential feature. Nowadays, ML is used as a general purpose language, and it is current practice to use exceptions: not only in exceptional situations, and not only for efficiency reasons.

We want to show how an existing framework basically devoted to the construction of purely functional programs, namely program extraction in the calculus of inductive constructions, can handle exceptions in a modular way. By modular we mean that:

(i) Only the parts of a program affected by exceptions need special treatment.
(ii) A component that may raise exceptions can be used without change in different environments.

Our solution is based on a continuation passing style (CPS) translation as in [3], but uses impredicative types in order to decrease the complexity of the translation while keeping modularity (a detailed discussion of this point is beyond the scope of this paper). The low level details of this translation are hidden in a small number of primitives. Readers acquainted with the monadic style of programming [21] will not be surprised to recognize a monad in these primitives.

The technique is illustrated on three examples. The first two are very simple and allow us to present the basics. The third one is an adaptation of a bigger algorithm independently developed in Coq by J. Rouyer [20], namely first order unification. Only small changes were needed in order to get a more efficient program from the original one.

The basic solution presented here is slightly more general than the previous one in [15], in order to make the treatment of example 1 possible. On the other hand, this paper concentrates on the case of a single exception carrying no value. The extension to the general case presented in [15] can be transposed without difficulty.

Note that, in the framework of conventional imperative programming, escape mechanisms are often considered as an optimization trick, whereas researchers have concentrated their efforts on block based control structures, with one input and one output. Typically, exceptions are extraneous to program calculation [7] and specification refinement [1,14].

1.1 Functional programming and formal specifications

Strongly typed functional programming has for a long time been advocated as a good framework for developing programs easy to reason about. Pure functional programs are mathematical expressions representing values which can be manipulated as easily as ordinary mathematical expressions. In particular, the result of a computation does not depend on the order of evaluation of subexpressions. Hence the tenet:

$$\textit{specification} \quad = \quad \textit{program}$$

Clearly, the straightforward recursive definition of factorial is as good as any other mathematical definition of this function. However, most of us would not be inclined to admit the tail recursive definition of factorial as its specification, though it is still a functional program. Things get quickly worse with slightly more complex problems like sorting. A good specification states that the result should be an ordered permutation, and takes the form of a very convincing

and very inefficient function. Note that, the nice mathematical properties of functional languages make possible the transformation of an inefficient program into an efficient one using algebraic manipulations (see for instance the Bird-Meertens formalism).

There is another way of developing correct functional programs, *program extraction* [11,17]: one tries to build a *constructive* proof of a specification $\forall x.\ P(x) \to \exists y.\ Q(x,y)$, where x is the input, y the output, P the precondition and Q the relation between input and output. Such a proof can be considered a functional program through a correspondence studied by Curry, Howard, Martin-Löf and others (see e.g. [9]):

$$\begin{aligned} \textit{formula} &= \textit{type}, \\ \textit{proof} &= \textit{program}. \end{aligned}$$

For example, a proof of $A \to B$ gives a proof of B from any proof of A, and then can be considered as a function from A to B: hence $\to$ denotes implication as well as the function space constructor. More generally, a formula A is considered as a type corresponding to the set of the proofs of A. Using a suitable realizability interpretation, it is also possible to remove irrelevant (from an algorithmic point of view) parts of the proof. A general result of the related meta-theory ensures that the extracted program f satisfies its specification, i.e. $\forall x\ P(x) \to Q(x, f(x))$. Such a mechanism is implemented in Coq, a general proof assistant devoted to the *Calculus of Inductive Constructions* [6].

In this framework, one simultaneously develops a program with its proof. Here are the main steps:

(i) State the specification, a logical formula, as a goal to be proved.
(ii) Prove it, typically by induction on one or several variables.
(iii) Ask the system to extract the algorithmic content of the proof.

Note that only step (ii) has an effect upon efficiency of the extracted program f. In order to make the specification as clear as possible, one is free to use any function, including inefficient ones. For instance $Q(x,y)$ may have the shape $y = g(x)$, making program g a specification of f.

1.2 Introducing exceptions

In practice "impure" features like exceptions (also state and input/output, but they are not considered in this paper) prove very useful. Let us consider the computation of the product of the leaves of a binary tree. We know that the result must be zero as soon as a zero leaf is met. The natural way of

```
function leavemult (t : tree) : nat =
  letrec mulrec(t : tree) : nat =
    match t with
      leaf(n) → if n=0 then raise nul else n
    | node(t1,t2) → (mulrec t1) × (mulrec t2)

  in try mulrec t with nul → 0
```

Fig. 0. Simple example 0

expressing this is to raise an exception caught by the calling function (see figure 0). Attempts to simulate this behaviour in a purely functional setting are possible but lead the programmer to error-prone manipulation of additional parameters.

In order to extend the formulae-as-types setting to exceptions, we need to understand their type as well as their logical meaning. Unfortunately, just *typing* an expression raising an exception is not a trivial matter, and a language like ML assigns an indeterminate type to **raise** v, which can occur in an expression of any type [1]. The only constraint is that E_1 and E_2 must have the same type in **try** E_1 **with** $\langle \text{pattern} \rangle \to E_2$. The problem has already been studied for more general *control operators* such as **Callcc** in the early 90's [10,16]. There are deep connections with constructive interpretations of classical logic [9,8,12] but we will follow a somewhat different path here.

Let us just remark that there is no hope of introducing exceptions without breaking the original simplicity of functional programming for a simple reason: the result of such computations is sensitive to the order of evaluation. For instance, the following expression returns $\langle 1, 1 \rangle$ if the pair is computed from left to right and $\langle 2, 2 \rangle$ in the other case:

try $\langle$**raise** exc(1), **raise** exc(2)$\rangle$ **with** exc(n) $\to \langle n, n \rangle$.

The general trick is then to *translate* the types and their associated functions into more complicated types and functions, in a way that takes into account some evaluation order. Such a translation can be extended to exceptions.

1.3 The rôle of continuations

The notation $\{x{:}A \mid (P\ x)\}$ is used for a type inhabited by ordered pairs $\langle x, p \rangle$, where p is a proof of $(P\ x)$. During program extraction, p is removed and this type becomes just A. The specification $\forall x{:}S.\ P(x) \to \exists y{:}T.\ Q(x, y)$

[1] As a consequence, the type of mulrec in example 0 forgets the fact that exception nul could be raised, though it should be considered as a possible "result" of mulrec.

given above can be restated as $\{x{:}S \mid (P\,x)\} \to \{y{:}T \mid (Q\,x\,y)\}$, which becomes $S \to T$ at extraction time.

Castéran remarked about example 0 that stating a goal of the right form very naturally leads the user to an algorithm in continuation passing style [3]. More specifically, instead of proving the goal $\forall t$:tree. RESU(t) by induction on t, where RESU(t) = $\{n$:nat $\mid$ (Prod $t\,n)\}$ and where Prod is the obvious predicate – this is called *direct style* – he considered a goal equivalent to:

$$\forall t', t{:}\text{tree}.\ (\text{RESU}(t') \to \text{RESU}(t)) \to \text{RESU}(t).$$

Intuitively, the idea is to search an object r of type RESU(t') and to apply a function k of type RESU(t') $\to$ RESU(t) to r in order to get the final result. The function k is called a *continuation*. We will see how to hide continuations thanks to a suitable generalization of the remark of Castéran.

The rest of this paper is organized as follows. Section 2 is a very quick and informal introduction to the calculus of constructions with inductive definitions, as used in the Coq system. Section 3 introduces general definitions enabling the development of programs with exceptions. Section 4 illustrates their use on some examples, which have been completely and mechanically verified.

2 General Framework and Notations

2.1 The Calculus of Constructions

The Calculus of Constructions is a typed λ-calculus. Objects of the first level are constants like 0 or the successor function. They inhabit objects of the second level, which are propositions seen as set of proofs, themselves of type Prop. Typed abstraction is denoted by $\lambda x^A.\ B$ or $\lambda x{:}A.\ B$. Application is denoted by juxtaposition fa or $(f\,a)$. Products are denoted by $\forall x{:}A.\ B$; when B does not depend on A, the simpler notation $A \to B$ is generally used. Application associates to the left and arrow to the right. The reader is refered to [4,2] for a detailed presentation.

Examples. Logical operations such as $\vee$ are of type Prop $\to$ Prop $\to$ Prop. Predicates on natural numbers, are objects of type nat $\to$ Prop. The type of

iterators can be represented in system F style by

$$\mathrm{iter} = \forall X{:}\mathrm{Prop}.\ X \to (X \to X) \to X.$$

Inhabitants of iter are polymorphic higher order functions which, given a type X, a object x of X and a function f from X to X, return $f^n(x)$ for some integer n. For instance for $n = 2$ we have

$$\mathrm{it2} = \lambda X^{\mathrm{Prop}}.\ \lambda x^X.\ \lambda f^{X \to X}.\ f(fx).$$

The type iter is itself of type Prop, hence iterators may be applied to iterators. Church used a similar encoding for natural numbers.

For program extraction purposes, Coq in fact distinguishes two sorts of props: Prop (properties) and Set (real objects). Only data and functions of sort Set are kept by program extraction. Objects of sort Prop handle logical information on data and functions, which is useful for reasoning during program construction, but useless at run time. This feature is used below, in order to capture the piece of information carried by an exception with no additional cost at run time.

Data structures like nat (the natural numbers), binary trees and so on are of type Set. Then given a predicate P of type $\mathrm{nat} \to \mathrm{Prop}$ and a function f of type $\forall x{:}\mathrm{nat}.\ (P\ x) \to \mathrm{nat}$, the corresponding extracted function f_e is of type $\mathrm{nat} \to \mathrm{nat}$; moreover if x is a nat such that Px, that is, if we have a proof p of type Px, $f_e x$ and fxp denote the same value.

2.2 The Calculus of Inductive Constructions

For theoretical and practical reasons, the Calculus of Constructions has been extended to inductive types. The user can give his own *inductive definitions* in a secure way. The simpler ones correspond to concrete data types of ML. For example, the definition of nat is

Inductive nat : Set :=
  O : nat | S : nat → nat.

The binary trees we use in this paper are defined by:

Inductive tree : Set :=
  leaf : nat → tree | node : tree → tree → tree.

Predicates and n-ary relations can also be inductively defined à la Prolog, for instance:

Inductive even : nat → Prop :=
  ev0 : (even O) | evSS : ∀n:nat. (even n) → (even (S (S n))).

It is then possible to define the type of even numbers as

Inductive even_nat : Set :=
  en_intro : ∀n:nat. (even n) → even_nat.

The type $\{x{:}A \mid (P\ x)\}$ introduced above is in fact a general purpose inductive type. A type like even_nat can also be defined by $\{n{:}\mathrm{nat} \mid (\mathrm{even}\ n)\}$.

Each inductively defined type is automatically equipped with a general elimination principle enabling inductive reasoning and the definition of primitive recursive functions. Further information on inductive definitions and their use in Coq can be found in [5,18,19].

The calculus of inductive constructions is supported by a proof assistant named *Coq* [6]. In Coq, proofs/functions can be developed in an incremental way using commands that transform the proof tree.

3 Continuations and Exceptions

This section presents basic tools for developing proofs/programs in continuation passing style (CPS). Exceptions are introduced only in 3.3.

3.1 Typing CPS functions

Let us take a second look at the function discussed in 1.3. Castéran proposed to prove

$$\forall t', t{:}\mathrm{tree}.\ ((\mathrm{RESU}\ t') \to (\mathrm{RESU}\ t)) \to (\mathrm{RESU}\ t). \qquad (1)$$

by induction on t'. (RESU t') → (RESU t) is the specification (or the type) of a continuation k, to be applied an object r of type (RESU t') in order to get the final result. When r is known, $\lambda k.\, kr$ is a solution to (1). If a zero leaf is found during the search, we immediately have the solution $\lambda k.\, \langle 0, p\rangle$ where p is a proof that the product is zero.

Now given a solution mult_cps of (1) and a tree t, we get an object of type (RESU t) by an application of mult_cps to t, t and the identity function. We call (mult_cps t t $\lambda r.\, r$) the main call to mult_cps.

If no exception is raised, a careful inspection of the proofs shows that direct style construction and CPS construction are very similar. The search for an object of type (RESU t), by induction on t, corresponds in CPS to the search for an object of type $((\text{RESU } t') \to (\text{RESU } t)) \to (\text{RESU } t)$, by induction on t'.

In the CPS proof (RESU t) plays actually no special rôle, except at one place, corrresponding to the main call. Hence (RESU t) could as well be replaced by an arbitrary type X.

Let us also rename the bound variable t' as t. We replace (1) by

$$\forall t{:}\text{tree}.\ ((\text{RESU } t) \to X) \to X. \tag{2}$$

A proof of $\forall t$:tree. (RESU t) in direct style is then replaced by a proof of (2) in CPS. More generally, in CPS we always suppose that the function f we construct is directly or indirectly called by some "main function" $\mathcal{M}$. If X is the type of the result of $\mathcal{M}$ and $\forall x_1{:}B_1.\ \ldots \forall x_n{:}B_n.\ A$ is the type of f in direct style, the latter type becomes $\forall x_1{:}B_1.\ \ldots \forall x_n{:}B_n.\ (A \to X) \to X$ in continuation passing style. $A \to X$ is the type of the normal continuation.

Now there is no good reason to consider that the meaning of f should be tied to $\mathcal{M}$, since the same f could be used in completely different environments. Therefore, we state that the result of f should be usable for *any* value of X. The type A is then replaced by $\forall X{:}\,\text{Set}.\ (A \to X) \to X$ in CPS. For instance in the case of example 0 we prove

$$\forall t{:}\text{tree}.\ \forall X{:}\,\text{Set}.\ ((\text{RESU } t) \to X) \to X \tag{3}$$

by induction on t. Given a proof f of (3), we get $\mathcal{M}$ of type $\forall t$:tree. (RESU t) by taking (RESU t) for X and the identity for the normal continuation:

$$\mathcal{M} = \lambda t{:}\text{tree}.\ (f\ \ t\ \ (\text{RESU } t)\ \ \lambda r^{(\text{RESU } t)}.\, r).$$

3.2 Hiding continuations

Let us define the family of types (M A).

Definition $\text{M} := \lambda A{:}\text{Set}.\ \ \forall X{:}\,\text{Set}.\ (A \to X) \to X.$

Summing up the construction of a program: after introduction of arguments, one has to prove, in direct style, some goal A whereas in continuation passing style, one has to prove (M A). In order to hide the structure of (M A), we make this family of types an abstract type.

First, a value of type (M A) can be constructed from a value of type A:

Definition M_unit : $\forall A\colon \mathrm{Set}.\ A \to \mathrm{M}A :=$
$\lambda A^{\mathrm{Set}}.\, \lambda a^{A}.\ \lambda X^{\mathrm{Set}}.\, \lambda k^{A\to X}.\, ka.$

M_try is a path in the opposite direction. What matters is not its rather trivial definition, but the way its type is written. M_try is used in a "main program" for building an inhabitant of any type X given a CPS function whose result is of type M A and a continuation of type $A \to X$. To put it another way, if, at some stage of a proof, the current goal is G, applying M_try yields two subgoals, MA and $A \to G$.

Definition M_try :
$\forall A\colon \mathrm{Set}.\ \mathrm{M}A \to \forall X\colon \mathrm{Set}.\ (A \to X) \to X :=$
$\lambda A^{\mathrm{Set}}.\, \lambda f^{\mathrm{M}A}.\ f$

Suppose that, at direct style level, we want to apply some function of type $A \to A'$ to an expression of type A in order to prove a goal of type A'. In CPS, these types become respectively $A \to \mathrm{M}A'$, MA and MA'. Of course we cannot apply $f^{A\to \mathrm{M}A'}$ to $m^{\mathrm{M}A}$, but what we want is *first* to compute m, *then* to bind the result to a^A and *finally* to compute fa. The corresponding ML expression is: **let** $a = m$ **in** $f\ a$. In CPS this means that m is applied to a continuation $\lambda a^A.\, fa \ldots$, where the dots represent the continuation for fa. This mechanism is encapsulated in M_bind.

Definition M_bind :
$\forall A, A'\colon \mathrm{Set}.\ \ \mathrm{M}A \to (A \to \mathrm{M}A') \to \mathrm{M}A' :=$
$\lambda A, A'^{\mathrm{Set}}.\ \lambda m^{\mathrm{M}A}.\ \lambda f^{A\to \mathrm{M}A'}.\ \lambda X^{\mathrm{Set}}.\, \lambda k^{A'\to X}.\, (mX(\lambda a^{A}.\, faXk)).$

3.3 Handling exceptions

If an exception can be raised, we need an assumption on X, namely that X has a distinguished inhabitant e which will be the result of the main computation in the case where an exception is raised[2]. For instance, in example 0, the value considered for X is roughly nat, and we take 0 for e. This leads us to

[2] In the case where an exception carries a value ranging over some domain D, we need to assign an inhabitant of X for each possible value of D. This is detailed in [15].

```
Function core_ow (m:nat; t:tree):nat =
  letrec comprec(t:tree; a:nat):nat =
    match t with
      leaf(n)  → (g a + n)
    | node(t1,t2)  →
        let a2=(comprec t2 a) in let a12=(comprec t1 a2) in a12
  in (comprec t 0)
  where g (n:nat):nat =
    if n ≤ m then n else raise threshold.

Function F_overweight (m:nat; t:tree):bool =
  try let r = (core_ow m t) in false
  with threshold → true.
```

Fig. 1. Example 1

consider $X \to (A \to X) \to X$ instead of $(A \to X) \to X$.

From the point of view of program correctness, we are interested in the implicit meaning of this exception: an exception is always raised for some reason, and if an exception has *not* been raised, this may also be meaningful. This is better seen on example 1 of figure 1. In this example, we want to compute a boolean which is true if the sum of the leaves of a binary tree is greater than or equal to a given threshold m, and false otherwise. The given algorithm traverses the tree t from right to left, while accumulating in a the sum of the encountered leaves [3] ; as soon as a exceeds m, we know that the answer is true; if no exception has been raised, the answer is false.

Notice that the result r is not compared with m—it is not even used at all. Here the fact that during a run an exception has not been raised is meaningful. In general, we want to say that the distinguished element of X is a correct result *provided some condition is satisfied*. Let P be the weakest condition for raising the exception. We give e the type $P \to X$. In example 0 (respectively example 1), e is intuitively a function mapping any proof of the fact that the product of the leaves is zero, to 0 (respectively, that the weight of the tree is too large, to true).

The structure of the result of $\mathcal{M}$ is now given by X, P and e. Of course, we generally cannot give a direct proof of P, but only some sufficient condition C and it remains to prove that $C \to P$. For instance, in example 0, C means for a given leaf that the leaf is zero. In addition to the true continuation of type $A \to X$, we then need a "logical continuation" of type $C \to P$. We replace $\mathrm{M}A$ above by $\mathrm{Mx}\,CA$, whose inhabitants are either values (more precisely

[3] It is a variant of an algorithm given in [13].

computations) of type A or proofs of C:

Definition Mx :=
$\lambda C^{\,\mathrm{Prop}}.\ \lambda A^{\,\mathrm{Set}}.$
$\forall X\!:\!\mathrm{Set}.\ \forall P\!:\!\mathrm{Prop}.\ \forall e\!:\!P \to X.\ (C \to P) \to (A \to X) \to X.$

To put it another way an inhabitant of $\mathrm{Mx}\,CA$ is either an object of type A or an exception saying that C is satisfied. It is not very difficult to adapt the primitives given in 3.2 (see appendix A) and we obtain a new one, Mx_raise of type $\forall C\!:\!\mathrm{Prop}.\ \forall A\!:\!\mathrm{Set}.\ C \to \mathrm{Mx}\,CA$. The meaning of these primitives is given by their type:

- Mx_try builds an object of type X from an inhabitant m of $\mathrm{Mx}\,CA$, a normal continuation and an exceptional continuation e, i.e. a value of type X to be used when C is proved. Raising an exception during the computation of m means that e is used, i.e. that this exception is caught by Mx_try.
- There are two basic ways of producing an object of type $\mathrm{Mx}\,CA$: using Mx_unit, if we get a normal object of type A and using Mx_raise, if we get a proof that an exception can be raised.
- Mx_bind plays the same rôle as M_bind, its definition propagates the justifications that an exception can be raised.

At extraction time, the propagation of justifications that an exception can be raised is removed, only e remains.

We sometimes use the infix notation $C \vee A$ for $\mathrm{Mx}\,CA$. Indeed, $\vee$ can be considered as an asymmetrical disjunction between a Prop and a Set.

4 Three Case Studies

4.1 Copying a tree without unnecessary copies

Given a tree t, we want a similar tree t', where the nodes satisfying some given property P have been modified. For some values of t, nothing has to be changed and we expect that the function returns the same result as the identity function. In such cases, the code produced by any reasonable compiler would just copy a pointer instead of the whole structure. In the general case, only some parts of the original tree have to be reconstructed: if p is a path from the root to a leaf and n is a node of p satisfying P, the subpath between the root and n must be reconstructed but the part of p between n and the leaf can be kept if n is the last node to be changed. This problem is typically

letrec core_cop = **function**
leaf(n) $\rightarrow$ **raise** nochange
| Nd1(blue, t_1) $\rightarrow$ **try let** v_1 = core_cop t_1 **in** Nd1(red, v_1)
with nochange $\rightarrow$ Nd1(red, t_1)
| Nd1(red, t_1) $\rightarrow$ Nd1(red, core_cop t_1)
| Nd1(yellow, t_1) $\rightarrow$ Nd1(yellow, core_cop t_1).

let eff_cop t = **try** core_cop t **with** nochange $\rightarrow t$.

Fig. 2. Example 2

encountered in theorem provers and some of them use the following trick in order to save space.

For illustration purposes, it is enough to consider trees with only one branch.

Inductive color : Set := blue : color | red : color | yellow : color.

Inductive tree1 : Set :=
Leaf : nat $\rightarrow$ tree1 | Nd1 : color $\rightarrow$ tree1 $\rightarrow$ tree1.

We just want to replace blue nodes by red ones. The following function specifies the desired value:

Recursive definition def_cop : tree1 $\rightarrow$ tree1 :=
(leaf n) $\Rightarrow$ (leaf n)
| (Nd1 blue t) $\Rightarrow$ (Nd1 red def_cop t)
| (Nd1 red t) $\Rightarrow$ (Nd1 red def_cop t)
| (Nd1 yellow t) $\Rightarrow$ (Nd1 yellow def_cop t).

The trick to be used in this situation is quite strange: on changing nodes exceptions are not raised but *caught* (see figure 2)! On the other hand, an exception is *always* raised at a leaf. Note the recursive use of the **try** construct. Intuitively, if an exception reaches some node n, we know that no blue node could be below n in the original tree; we can then keep the original subtree. We would like to replace such an operational argument by a more convincing proof.

4.1.1 Development in Coq

A correct by construction development of eff_cop turns out to be very simple in the framework described above. We have to prove that core_cop either returns an exception, if t = (def_cop t), or returns a tree t' equal to (def_cop t) [4]. We

[4] We mean: a tree t' denoting the same value as (def_cop t).

then look for a constructive proof of:

$$\forall t{:}\text{tree1}.\ (\text{Mx}\quad t = (\text{def\_cop}\, t)\ \ \{t'{:}tree1 \mid t' = (\text{def\_cop}\, t)\}). \tag{4}$$

We proceed by induction on t. If t is $(\text{Leaf}\, n)$ we get the subgoal

$$\begin{array}{l}(\text{Mx}\ \ (\text{Leaf}\, n) = (\text{def\_cop}(\text{Leaf}\, n)) \\ \qquad \{t' : \text{tree1} \mid t' = (\text{def\_cop}(\text{Leaf}\, n))\})\end{array} \tag{5}$$

and we apply Mx_raise; it remains to prove $(\text{Leaf}n) = (\text{def\_cop}(\text{Leaf}n))$ which is trivial.

In the inductive case, we are given a color c, a subtree t_1 and an assumption R_1 representing $(\text{core\_cop}\, t_1)$, of type (4) where t is replaced by t_1. We proceed by cases on c. If c is blue, we get the goal

$$\begin{array}{l}(\text{Mx}\ \ (\text{Nd1 blue}\, t_1) = (\text{Nd1red}\, (\text{def\_cop}\, t_1)) \\ \qquad \{t' : tree1 \mid t' = (\text{Nd1 red}\, (\text{def\_cop}\, t_1))\}).\end{array} \tag{6}$$

Following figure 2, we then try to compute R_1, while catching the exception possibly raised during this computation. That is, we use Mx_try with $t_1 = (\text{def\_cop}\, t_1)$ for C, $\{t'{:}tree1 \mid t' = (\text{def\_cop}\, t_1)\}$ for A and the type given in (6) for X. This generates the two subgoals $A \to X$ and $C \to X$. The former means that given v_1, the value returned by R_1 in the "normal" case, we must find an inhabitant of (6). It is enough to apply Mx_unit to $(\text{Nd1 red}\, t')$, where t' is the witness of v_1. The latter subgoal means that we must find an inhabitant of (6) when $t_1 = (\text{def\_cop}\, t_1)$. We just have to apply Mx_unit to $(\text{Nd1 red}\, t_1)$.

If c is red, we compute the result for t_1 without catching the exception; that is, we use Mx_bind instead of Mx_try. The subgoal $C \to X$ is replaced by $C \to C'$, that is, we have to justify the propagation of an exception. Here we have to prove:

$$t_1 = (\text{def\_cop}\, t_1)\ \to\ (\text{Nd1 red}\, t_1) = (\text{Nd1 red}\, (\text{def\_cop}\, t_1)) \tag{7}$$

The case where c is yellow is similar. In this example proof obligations are always as simple as (7).

The main function eff_cop is constructed by proving

$$\forall t{:}\text{tree1}.\ \{t'{:}\text{tree1} \mid t' = (\text{def\_cop}\, t)\}. \tag{8}$$

We just have to apply Mx_try with $(\text{core\_cop}\, t)$. In this case $A = X$, hence we provide the initial continuation $\lambda x.\, x$.

Here is the program extracted by Coq:

```
let rec core_cop = function
      Leaf n  → Mx_raise
  |  Nd1(c, t1)  →
       (match c with
          blue  → Mx_try (core_cop t1) (fun v1  → Mx_unit Nd1(red,v1))
                    (Mx_unit Nd1(red,t1))
        | red  → Mx_bind (core_cop t1) (fun v1  → Mx_unit Nd1(red,v1))
        | yellow  → Mx_bind (core_cop t1)
                    (fun v1  → Mx_unit Nd1(yellow,v1))).

let eff_cop t = Mx_try (core_cop t) (fun x  → x) t.
```

4.1.2 What has been achieved

Of course, the algorihm extracted by Coq is not identical to the one given in figure 2. It is a translation of the latter into a purely functional sublanguage of ML or more precisely a function which behaves the same and which is proved equivalent to def_cop. Functions Mx_try, Mx_raise... were not unfolded, in order to get a function almost as readable as figure 2.

Did we develop a really less space-consuming version than def_cop? We have no direct way of expressing this in the specification. Anyway, if we look at the previous specification of core_cop, nothing prevents us from forgetting about exceptions. In fact, we could use the same strategy for blue as for the other colors and would then get just a sequential version of def_cop. At this point, it is then difficult to guarantee anything about the behaviour of eff_cop.

However we can state a stronger specification for core_cop:

$$\forall t{:}\text{tree1}.\ (\text{Mx}\quad t = (\text{def\_cop}\ t)\quad \{t'{:}tree1 \mid t' = (\text{def\_cop}\ t)\ \wedge\ t' \neq t\}). \quad (9)$$

Such a core_cop cannot return a new version of t, for it simply cannot return a tree if $t = (\text{def\_cop}\ t)$. In this case, core_cop may only return an exception. The natural choice for a calling function $\mathcal{M}$ like eff_cop is then to return the original t, in which case eff_cop behaves like the identity function as desired. Note that, a stupid choice for $\mathcal{M}$ is still possible, for instance eff_cop could explicitly return def_cop(t):

```
let eff_cop t = try core_cop t with nochange  → def_cop(t).
```

But this means that an important piece of information is discarded. Linear types could prevent this but are beyond the scope of this paper.

The proof of (9) follows the same lines as the proof of (4) and provides exactly the same extracted algorithm. Additional proof obligations boil down to $\mathrm{Nd1}(c,t) \neq \mathrm{Nd1}(c',t')$ if $c \neq c'$ or $t \neq t'$. A Coq script is in appendix B.

4.2 Weighing a tree

Example 1 is perhaps more representative of usual practice and illustrates ordinary programming techniques, such as the use of an accumulator and of a locally defined auxiliary function.

First we state the specification of the main program where suml returns the sum of the leaves of a binary tree.

Definition P_overweight := λm:nat. λt:tree. $(m \leq (\mathrm{suml}\ t))$.
Definition RESU :=
  λm:nat. λt:tree. $\{(\mathrm{P\_overweight}\ m\ t)\}+\{\neg(\mathrm{P\_overweight}\ m\ t)\}$.

$\{P\}+\{Q\}$ denotes an enumerated type with two values; the first (resp. second) value can be built if P (resp. Q) is provable. When $Q = \neg P$, $\{P\}+\{Q\}$ denotes the truth value of P.

For the development of the algorithm, we need a more general form of P_overweight which takes an accumulator into account.

Definition P_overweight_accu := $\lambda m, a$:nat. λt:tree. $(m \leq a + (\mathrm{suml}\ t))$.

The result of core_ow is an exception if (suml t) exceeds m and (suml t) itself otherwise. We also want that if the function actually computes (suml t), then this value does not exceed m. The internal function comprec has a similar specification taking the accumulator into account, hence we introduce the type of a natural equal to $a + (\mathrm{suml}\ t)$ if this value is not greater or equal to m:

Inductive condsum_accu $[m, a$:nat; t:tree$]$: Set :=
  condsum_accu_intro :
    $\forall n$:nat. $(n = a + (\mathrm{suml}\ t)) \to \neg(m \leq n) \to (\mathrm{condsum\_accu}\ m\ a\ t)$.

The specification of comprec is based on condsum_accu_cps:

Definition condsum_accu_cps :=
  $\lambda m, a$:nat. λt:tree. $(\mathrm{P\_overweight\_accu}\ m\ a\ t) \vee (\mathrm{condsum\_accu}\ m\ a\ t)$.

The specification of the result of core_ow is then (condsum_accu_cps m 0 t).

The auxiliary function $g(m,n)$ returns n, but only if n is not greater or equal

to m. Otherwise, g raises an exception. The specification of (g m n) is then (le m n) $\surd$ (T_aux m n) where T_aux is defined by:

Local T_aux := $\lambda m, n$:nat. $\{n'$:nat$|\ n = n' \wedge \neg(m \leq n')\}$.

One proves the theorem:

Theorem core_ow : $\forall m$:nat. $\forall t$:tree. (condsum_accu_cps m 0 t).

Once m and t are pushed into the context, the first steps are to assume g of type

$$\forall n\text{:nat. (le } m\ n) \surd \text{(T\_aux } m\ n)$$

and comprec of type

$$\forall a\text{:nat. (condsum\_accu\_cps } m\ a\ t).$$

We get the result by a simple instantiation of comprec, and g is proved using Mx_unit and Mx_raise, depending on whether $m \leq n$ or not. The development of comprec, by induction on t, is guided by figure 1 and uses only Mx_bind and Mx_unit.

Finally, the function F_overweight is specified by $\forall m$:nat. $\forall t$:tree. (RESU m t) and is easily obtained using Mx_try and core_ow. In this process, X is instantiated to (RESU m t) and we prove (condsum_accu m 0 t) $\rightarrow$ (RESU m t) and (P_overweight_accu m 0 t) $\rightarrow$ (RESU m t) using, respectively, the witnesses false and true.

4.3 First Order Unification

Attempting to unify two terms T and U roughly consists of a double induction over T and U taking care of propagation of substitutions. The result is either a most general unifier, in case of success, or an answer "T and U are not unifiable", i.e. a failure. The obvious choice for the type of the result is a sum like $\langle$mgu$\rangle + \langle$failure$\rangle$. In his original development [20], J. Rouyer chose:

Inductive Unification [t_1, t_2:quasiterm] : Set :=
  Unif_succeed: $\forall f$:quasisubst. (unif t_1 t_2) $\rightarrow$
    further conditions for f to be an mgu
    $\rightarrow$ (Unification t_1 t_2)
| Unif_fail: $\forall f$:quasisubst. $\neg$(Subst f t_1)=(Subst f t_2)
    $\rightarrow$ (Unification t_1 t_2).

In the original development, this type is also the type of the result of the function corresponding to the double induction, hence a failure is transmitted backwards step by step until the root.

With the definitions given above, we can construct an algorithm that just tries to compute the mgu. As soon as an incompatibility is detected, e.g. between two constants, an exception is raised: this is the expected behaviour of a real implementation.

We proceed from the original development as follows, in order to minimize modifications. First we split the initial definition of Unification into two parts, Unification_s of kind Set and Unification_f of kind Prop:

Inductive Unification_s $[t_1, t_2$:quasiterm$]$: Set :=
  Unif_succeed_def: $\forall f$:quasisubst. (unif t_1 t_2) $\rightarrow$
    further conditions for f *to be an mgu*
    $\rightarrow$ (Unification_s t_1 t_2).
Inductive Unification_f $[t_1, t_2$:quasiterm$]$: Prop :=
  Unif_fail_def: $\forall f$:quasisubst. $\neg$(Subst f t_1)=(Subst f t_2))
    $\rightarrow$ (Unification_f t_1 t_2).

From Unification_s and Unification_f we inductively define Unification_or_fail which is equivalent to the original definition of Unification (using two obvious clauses).

Unification is redefined using Mx, Unif_succeed and Unif_fail are redefined using respectively Mx_unit and Mx_raise.

Definition Unification :=
  $\lambda t_1, t_2$:quasiterm. (Unification_f t_1 t_2) $\surd$ (Unification_s t_1 t_2).

The only problem is with Unification_rec, a function implicitly provided with the inductive definition of Unification for inductive reasoning. Here, we define Unif_elim with Mx_bind and the abbreviation Unifelim H for **Apply** Unif_elim **with** 3:=H. **Elim** H can then be replaced by Unifelim H when H has type (Unification t u) and when the current subgoal has type (Unification t' u')—this happens to be always the case.

Finally we adapt the script of the original development. It turns out that very few modifications are required, and that they are systematic. They split into two classes.

(i) Replacing **Elim** H when the type of H is (Unification t u), as described above.
(ii) Sometimes the current subgoal becomes (Unification_f t u) instead of (Unification t u). It is then necessary to replace Unif_fail by Unif_fail_def. Similarly the type of the result of two lemmas must be changed to (Unification_f t u).

There are 7 modifications of the first kind and 3+2 of the second kind. About 100 lines have been added for the new definitions of Unification, Unif_succeed, Unif_fail and Unif_elim. The original development takes about 2.800 lines. This can be compared to the modifications needed for the same transformation if the algorithm had been expressed in a usual programming language: each statement returning the value *failure* would be systematically replaced by a statement raising an exception.

To sum up, no complexity is added if we compare with the direct style development.

5 On the use of impredicativity

Let us conclude with some remarks on the types of our constructs and relate them with our previous work [15]. Recall that $C \surd A$ is a Set defined by

$$C \surd A = \forall X{:}\mathrm{Set}.\ \forall P{:}\mathrm{Prop}.\ \forall e{:}P \to X.\ (C \to P) \to (A \to X) \to X. \quad (10)$$

We see that X is quantified over objects including $C \surd A$ itself. Such a definition is said to be *impredicative*. Another example given in Section 2.1 is the type of iterators.

Impredicative type systems are very powerful. It is possible in system F [9] to define many data structures such as polymorphic lists, binary trees, and other mathematical objects like infinitely branching trees, streams and ordinal numbers. It is also possible, using only (higher order) primitive recursion, to define many more functions than in simple type systems.

But impredicative definitions are potentially dangerous because they involve a kind of circularity. It is then an important and non trivial matter to ensure that an impredicative type system remains consistent, i.e. that it does not leave room for a logical paradox such as Russell's paradox. This is shown in [9] for system F and in [4] for the calculus of constructions.

The use of impredicativity seems to be new in the study of control operators. The primary reason for introducing it is that it provides a very general form of polymorphism. Our original motivation was to allow a piece of code to be reused in any context. But do we really need the whole power of impredicativity? That is, do we sometimes take for X a type like $C \surd A$? Yes. It was the case in the recursive use of **try** in example 2. It just means that core_cop itself plays the rôle of $\mathcal{M}$.

In [15] impredicativity is already mentioned but an example like core_cop could not be developed in this framework. Instead of Mx, that paper uses Nx defined by:

Definition Nx :=
$\lambda X^{\mathrm{Set}}.\ \lambda P^{\mathrm{Prop}}.\ \lambda e^{P\to X}.\ \ \lambda C^{\mathrm{Prop}}.\ \lambda A^{\mathrm{Set}}.\ \ (C\to P)\to(A\to X)\to X.$

There we manage to share X, P and e during the development of a function by fixing them and state:

$$C \bigvee A = (\mathrm{Nx}\ X\ P\ e). \tag{11}$$

Nx has then an advantage over Mx in cases such as example 1: e is not passed as argument of the recursive function comprec.

Acknowledgement

I wish to thank the members of the Coq Project for their help on Coq, Pierre Crégut and Chet Murthy for introducing me to continuations and Thierry Coquand for his illuminating discussions during MPC'95. Pierre Lescanne suggested the example of first order unification. I am also grateful to Francis Klay and Kathleen Milsted for their help in the presentation of this paper.

A Abstract type for exceptions

Definition Mx :=
$\lambda C^{\mathrm{Prop}}.\ \lambda A^{\mathrm{Set}}.$
$\forall X{:}\mathrm{Set}.\ \forall P{:}\mathrm{Prop}.\ \forall e{:}P\to X.\ (C\to P)\to(A\to X)\to X.$

Definition Mx_unit :
$\forall C{:}\mathrm{Prop}.\ \forall A{:}\mathrm{Set}.\ A\to \mathrm{Mx}\,CA :=$
$\lambda C^{\mathrm{Prop}}.\ \lambda A^{\mathrm{Set}}.\ \lambda a^{A}.\ \ \lambda X^{\mathrm{Set}}.\ \lambda P^{\mathrm{Prop}}.\ \lambda e^{P\to X}.\ \lambda i^{C\to P}.\ \ \lambda k^{A\to X}.\ ka.$

Definition Mx_raise :
$\forall C{:}\mathrm{Prop}.\ \forall A{:}\mathrm{Set}.\ C\to \mathrm{Mx}\,CA :=$
$\lambda C^{\mathrm{Prop}}.\ \lambda A^{\mathrm{Set}}.\ \lambda c^{C}.\ \ \lambda X^{\mathrm{Set}}.\ \lambda P^{\mathrm{Prop}}.\ \lambda e^{P\to X}.\ \ \lambda i^{C\to P}.\ \lambda k^{A\to X}.\ e(ic).$

Definition Mx_try :
$\forall C{:}\mathrm{Prop}.\ \forall A{:}\mathrm{Set}.\ \mathrm{Mx}\,CA\to\forall X{:}\mathrm{Set}.\ (A\to X)\to(C\to X)\to X :=$
$\lambda C^{\mathrm{Prop}}.\ \lambda A^{\mathrm{Set}}.\ \lambda m^{\mathrm{Mx}\,CA}.\ \ \lambda X^{\mathrm{Set}}.\ \lambda k^{A\to X}.\ \lambda e^{C\to X}.\ \ (mXCe\ (\lambda p^{C}.\ p)\ k).$

Definition Mx_bind :
$\forall A, A'$: Set. $\forall C, C'$: Prop.
$\mathrm{Mx}\, C A \to (A \to \mathrm{Mx}\, C' A') \to (C \to C') \to \mathrm{Mx}\, C' A' :=$
$\lambda A^{\mathrm{Set}}.\ \lambda A'^{\mathrm{Set}}.\ \lambda C^{\mathrm{Prop}}.\ \lambda C'^{\mathrm{Prop}}.\ \ \lambda m^{\mathrm{Mx}\, C A}.\ \ \lambda f^{A \to \mathrm{Mx}\, C' A'}.\ \lambda j^{C \to C'}.$
$\lambda X^{\mathrm{Set}}.\ \lambda P^{\mathrm{Prop}}.\ \lambda e^{P \to X}.\ \ \lambda i^{C' \to P}.\ \lambda k^{A' \to X}.$
$(mXPe\,(\lambda c^{C}.\, i(jc))\,(\lambda a^{A}.\, faXPeik))$.

B Constructive Proof of core_cop

Theorem strong_core :
$\forall t$:tree1. (Mx t=(def_cop t) {t':tree1 | t'=(def_cop t)&$\neg(t = t')$}).

Induction t.
Intros n; **Apply** Mx_raise; **Trivial**.

Intros c t_1 R1; **Case** c; **Simpl**.
Apply Mx_try **with** 1:= R1. *(* c = blue *)*
(Success of the try *)*
Intro v_1; **Apply** Mx_unit.
Elim v_1; **Intros** t' eg di; **Exists** (Nd1 red t');
[**Elim** eg; **Trivial** | **Simplify_eq**].

(Failure of the try *)*
Intro eg; **Apply** Mx_unit; **Exists** (Nd1 red t_1);
[**Elim** eg; **Trivial** | **Simplify_eq**].

Apply Mx_bind **with** 1:= R1. *(* c = red *)*
(Succes of the computation *)*
Intro v_1; Simpl; **Apply** Mx_unit;
Elim v_1; **Intros** t' eg di; **Exists** (Nd1 red t');
[**Elim** eg; **Trivial** | **Simplify_eq**; **Assumption**].
(Justification for the propagation of the exception *)*
Intro eg; **Simpl**; **Elim** eg; **Trivial**.

Apply Mx_bind **with** 1:= R1. *(* c = yellow *)*
Intro v_1; **Simpl**; **Apply** Mx_unit;
Elim v_1; **Intros** t' eg di; **Exists** (Nd1 yellow t');
[**Elim** eg; **Trivial** | **Simplify_eq**; **Assumption**].
Intro eg; **Simpl**; **Elim** eg; **Trivial**.
Qed.

References

[1] J-R. Abrial, *The B-Book*, in preparation (Prentice Hall, 1994).

[2] H.P. Barendregt, *Lambda Calculi with Types.* In S. Abramsky,& al., eds., *Handbook of Logic in Computer Science*, vol 2, (Clarendon Press, Oxford, 1992).

[3] P. Castéran, *Pro[gramm,v]ing with continuations: A development in Coq*, (Coq contribution, 1993, available by FTP on ftp.inria.fr).

[4] Th. Coquand and G. Huet, the calculus of constructions, *Information and Computation* **76** (1988) 95–120.

[5] Th. Coquand and C. Paulin-Mohring, *Inductively defined types*, in: P. Martin-Löf and G. Mints, eds., *Proceedings Colog'88* (Springer Verlag, LNCS 417, 1990).

[6] C. Cornes, J. Courant, J-C. Filliâtre, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi and B. Werner, *The Coq Proof Assistant User's Guide*, version 5.10 (INRIA-Rocquencourt et CNRS-ENS Lyon, july. 1995).

[7] E.W. Dijkstra, *A Discipline of Programming*, (Prentice-Hall, 1976).

[8] J-Y. Girard, *A new constructive logic: classical logic*, (Mathematical Structures in Computer Science, vol 1, 1991) 225–296.

[9] J-Y. Girard, Y. Lafont, P. Taylor, *Proofs and Types*, (Cambridge Univ. Press, vol 7, 1990).

[10] T. Griffin, *A formulae-as-types notion of control.* (proceedings POPL, Orlando, 1990).

[11] S. Hayashi, H. Nakano, *PX, a Computational Logic* (Foudations of Computing, MIT Press, 1988).

[12] Philippe de Groote, A Simple Calculus of Exception Handling, in: M. Dezani-Ciancaglini and G. Plotkin, eds., *Proceedings of TLCA'95* (Springer Verlag, LNCS 902, 1995).

[13] J.L. Lawall and O. Danvy, *Separating Stages in Continuation-Passing Style Transformation* (proceedings POPL, 1993).

[14] C. Morgan, *Programming from Specification* (Prentice Hall International Series in Computer Science, 1990).

[15] J-F. Monin, Extracting Programs with Exceptions in an Impredicative Type System, in: B. Möller, ed., *Proceedings of MPC'95* (Springer Verlag, LNCS 947, 1995).

[16] C. Murthy, *An evaluation semantics for classical proofs* (proceedings LICS, Amsterdam, 1991).

[17] C. Paulin-Mohring, *Extraction de programmes dans le calcul des constructions*, thèse de doctorat de l'université Paris VII (1989).

[18] C. Paulin-Mohring, Inductive Definitions in the system Coq; Rules and Properties, in: M. Bezem and J.F. Groote, eds., *Proceedings of TLCA'93* (Springer Verlag, LNCS 664, 1993).

[19] C. Paulin-Mohring and B. Werner, Synthesis of ML Programs in the system Coq, *Journal of Symbolic Computation*, **15** (1993) 607–640.

[20] J. Rouyer, *Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs*, (Research Report 1795, INRIA-Lorraine, nov. 1992).

[21] P. Wadler, The essence of functional programming, in: *Proceedings of POPL'92* (Albuquerque, New Mexico, 1992) 1–14.