Extracting Programs with Exceptions in an Impredicative Type System

Jean-François Monin

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

Abstract. This paper is about exceptions handling using classical techniques of program extraction. We propose an impredicative formalization in the calculus of constructions and we illustrate the technique on two examples. The first one, though simple, allows us to experiment various techniques. The second one is an adaptation of a bigger algorithm previously developed in Coq by J. Rouyer, namely first order unification. Only small changes were needed in order to get a more efficient program from the original one.

1 Introduction

Several paradigms have been proposed for constructing correct programs from a formal specification. Let us sketch some among the most significant ones:

- Program calculation [6]: a program is obtained in a calculational manner as the solution of a logical assertion of the form $P \Rightarrow \mathrm{wp}.S.Q$ where P is the precondition, Q the postcondition and S the unknown (a command).
- Specification refinement [1, 13]: for instance an abstract command involving a quantifier can sometimes be refined using a loop; another kind of refinement, *data refinement*, consists roughly of replacing abstract data structures by concrete ones[1].
- Program extraction [11, 15]: one tries to build a constructive proof of a specification $\forall x\ P(x) \rightarrow \exists y\ Q(x,y)$. Such a proof can be considered a functional program through "Curry-Howard isomorphism":

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

  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. $\forall x\ P(x) \rightarrow Q(x, f(x))$. Such a mechanism is implemented in Coq, a general proof assistant devoted to the *Calculus of Inductive Constructions* [7].

[1] Refinement has also been extensively studied in the framework of algebraic specifications.

Each approach has its strong and weak points. For instance the two first are better appropriate to imperative programming (we mean "based on state transformation"); but constructing algorithms on recursive data structures is not always easy. Conversely the third approach is well suited to functional programming and can already deal with reasonably complex programs such as a mini-ML compiler.

In both the imperative and the functional world escapements – typically goto statements and exceptions – are generally considered as difficult to handle. 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 a current practice to use exceptions: not only in exceptional situations, and not only for efficiency reasons.

Therefore we consider that dealing with exceptions in recursive programs is an important step if we want to tackle real problems. The first work in this direction to our knowledge, at least in the context of Coq, is due to Pierre Castéran [3]. Recall the main steps for developing an algorithm by program extraction in Coq:

1. State the specification, a logical formula, as a goal to be proved.
2. Prove it, typically by induction on one or several variables.
3. Ask the system to extract the algorithmic content of the proof.

Castéran showed on a simple example – multiplying the leaves of a binary tree – that stating a goal of the right form very naturally leads the user to an algorithm in continuation passing style (CPS). More specifically, instead of proving the goal $\forall t$:tree $\mathrm{RESU}(t)$ by induction on t, where $\mathrm{RESU}(t) = \{n{:}\mathrm{nat} \mid n = \mathrm{leavemult}(t)\}$ and where leavemult is the obvious function – this would be *direct style* – he considered $\forall t, t'$:tree $(\forall n'{:}\mathrm{nat}\ n' = \mathrm{leavemult}(t') \to \mathrm{RESU}(t)) \to \mathrm{RESU}(t)$ and proved it by induction on t'; the desired $\mathrm{RESU}(t)$ was obtained by application of this function to t, t and $\lambda n.\, \lambda h.\, \langle n, h\rangle$ whose algorithmic content is the identity function. Moreover this proof/function was optimized in an elegant way by raising an exception as soon as a zero was detected on a leaf.

In other respects *control operators* such as `Abort` and `Callcc` have been studied in the formulae-as-type framework in the early 90's [10, 14]. It camed out that CPS transformation, which makes the interpretation of control operators possible in a purely functional setting, corresponds on the logical side to a Gödel translation. In other words, control operators are typed using *classical* theorems, hence in order to get a constructive interpretation of them we first translate them in intuitionistic logic by means of double negations[2]. As a simple example, if a has atomic type A, its translation is $\lambda k.\, ka$ whose type is $(A \to \bot) \to \bot$. In fact one often consider a variant where the empty type $\bot$ is replaced by the type of

[2] It is well known that classical logic is not constructive: the structure of the set of proofs of any theorem is made trivial by the behaviour of cut-elimination. This is discussed in depth in [9, 8].

the result of the main program, say R. The function k of type $A \rightarrow R$ is called a *continuation*, i.e. a function to be applied on a in order to get the final result. On the above example we could take RESU(t) for R, RESU(t') for A, and get an equivalent formulation.

The work of Castéran is extended in several directions in this paper. First the type of the final result is universally quantified. We need a further assumption on this type: in the simplest case it must have at least one value. We show that the gain in abstraction obtained in this way helps to nicely handle situations arising in practice, for instance when an auxilliary function is locally defined, or when an "accumulator" is used. Furthermore we gain in modularity: a function f possibly raising an exception $e{:}E$ can then be used in several contexts.

The type of the result of such a f is similar to $E \vee A$ as (impredicatively) defined in system F, i.e. $\forall X\ (E \rightarrow X) \rightarrow (A \rightarrow X) \rightarrow X$. This means that even if we develop a program using only propositional types, this typing is already more informative than the conventional one of, say, ML[3].

We introduce a general datatype called Nx for this kind of disjunction. Nx is equiped with a few constructors and destructors, namely Nx_unit, Nx_raise, Nx_handle and Nx_elim_Nx. Thanks to these abstract definitions it is possible to hide the CPS translation and to develop a program as in a functional language extended with exceptions. As a non trivial example it becomes possible to adapt the development of a first order unification algorithm [18] almost without modification.

Finally we consider handling several exceptions and a different Catch/Throw mechanism. An interesting remark is that the impredicativity of our typing can be very useful. Using impredicativity seems to be new in this context.

The rest if 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 two examples. The first one is about computing the weight of a binary tree unless a threshold is reached. The second example is an adaptation of a bigger algorithm previously developed by other people, namely first order unification. Only small changes were needed in order to get a more efficient program from the original one. In the last section we discuss how far the Coq system suits our needs.

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

[3] The type of the result of f would be A, and the type of any subexpression raise e would be an undeterminate α. Here we loose the fact that in the body of f an exception carrying a value of type E can be raised.

level, which are propositions seen as set of proofs, themselves of type Prop. Typed abstraction is denoted by $\lambda x^A\, B$ or `[x:A]B`. Application is denoted $(A\ B)$. Products are denoted by $\forall x$:A B or `(x:A)B`; when B does not depend on A, the simpler notation $A \rightarrow B$ 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. The type of natural numbers can be represented in system F style by nat $= \forall X$:Prop $X \rightarrow (X \rightarrow X) \rightarrow X$. The constant 0 of type nat is $0 = \lambda X^{\mathrm{Prop}} \lambda x^X \lambda f^{X \rightarrow X}\, x$; nat is itself of type Prop. Predicates on natural numbers, are objects of type nat→Prop. Logical operations such as $\vee$ are of type Prop→Prop→Prop.

For program extraction purposes Coq distinguishes in fact two sorts of props: Prop and Set. Only data and functions of sort Set are kept by program extraction. Objects of sort Prop are used to handle logical information on data and functions during program construction. For example, in the definition of nat above we would use Set instead of Prop.

Finally, for theoretical and practical reasons, the system allows the user to define *inductive definitions* in a secure way. The simpler ones correspond to concrete data types of ML. For example the definition of nat actually used is

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

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.
```

Such a type can also be defined using $\{x{:}A \mid (P\ x)\}$ which is a general purpose inductive type for the set of 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.

Each inductively defined type is automatically equiped 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, 16, 17].

2.2 Impredicativity

A definition of some object p is said to be *impredicative* when it involves a quantification over objects including p. A type system is impredicative when impredicative definition of types are allowed.

For instance the first definition of nat given above is impredicative because it is of the form nat $= \forall X$:Prop $\cdots$ and nat is of type Prop.

Impredicative type systems are very powerful. It is possible in system F to define many data structures like 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 much more functions than in simple type systems.

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 Russel's paradox. This is shown in [9] for system F and in [4] for the calculus of constructions

2.3 Sections in Coq

The calculus of inductive constructions is supported by a proof assistant named *Coq* [7]. In Coq proofs/functions can be developed in an incremental way using commands that transform the proof tree.

It is also possible in Coq to declare a common environment for several definitions using *sections*. For instance if we want to define several functions having some natural n as parameter, it is possible to declare n only once at the beginning of a section :

```
Section nat_functions.
  Variable n : nat.
  Definition dbl : nat := (plus n n).
  Definition square : nat := (mult n n).
  Definition pol1 : nat→nat→nat := [a,b:nat](plus (mult a n) b).
  Definition quad : nat := (plus dbl dbl).
End nat_functions.
```

After the end of this section the definition of `dbl` is actually `[n:nat](plus n n)`. The definition of `quad` becomes `[n:nat](plus (dbl n) (dbl n))`.

However this mechanism is not completely satisfactory if we want `dbl` to be just an intermediate definition for `quad`: we would prefer to get something like `[n:nat](let dbl=(plus n n) in (plus dbl dbl))`. In the following we suppose that this effect is obtained by using a *local* definition in the section :

```
Section nat_functions.
  Variable n : nat.
  Local dbl : nat= (plus n n).
  ...
End nat_functions.
```

This is not so far from the actual behavior of Coq, where these local definitions get expanded, at the price of potential replication of code.

In the general case common types, hypotheses, functions (like `X`, `P` and `e` below), are shared by several functions. Moreover `P` and `e` could depend on parameters. We would then use something similar to the functors of SML-NJ, i.e. modules parameterized by types and functions offering an interface consisting of several functions, while common auxilliary functions are hidden.

3 Continuations and Exceptions

When we develop a function f in continuation passing style, one always supposes that f is directly or indirectly called by some "main function" $\mathcal{M}$. Let X be the type of the result of $\mathcal{M}$.

If the "normal" type of the result of f is A, that is, if the type of the result of f is A when we express f in direct style, this type becomes $(A \to X) \to X$ in continuation passing style. $A \to X$ is the type of the normal continuation.

In a first attempt for introducing an exception we can assume another continuation $e{:}B \to X$, where B is the type of the value carried by the exception. The type of the result becomes $(B \to X) \to (A \to X) \to X$, which is just an instance of the encoding of $B + A$ in system F: X is arbitrarily fixed instead of beeing universally quantified. 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. Hence at some stage X will be an argument of f, i.e. will be universally quantified in the type of f. However if f has an argument using X we still cannot get exactly $B + A$.

In other respects we sometimes want several functions, say f and h, to share a common X and e. If f calls h, we prefer X and e not to be passed as parameters from f to h. A way to get this behaviour is to use the section mechanism of Coq and to declare X and e at the beginning of a section (however see the discussion above on this mechanism).

3.1 Simple Exceptions

Let us first consider the special case where e carries no value, i.e. $e{:}X$. (By the way it is enough for the first order unification algorithm studied below.) Even in this case an exception is raised for some reason, it implicitly carries some logical information. Let P be the weakest condition for raising the exception. We give e the type $P \to X$. Assuming e amounts to suppose that X is not empty if P is provable.

It is then possible to take $(C \to P) \to (A \to X) \to X$, where C is of kind Prop, as type of the result of f. In short the normal result of such a f has type A, but f may raise an exception if condition C is established. In this way anything about exceptions – excepted e itself – will be removed by program extraction. We call the whole type $(Nx\ C\ A)$, and use the infix notation `C V+ A`:

```
Section algo.
Variable X:Set.   Variable P:Prop.   Variable e:P→X.
Local Nx := [C:Prop][A:Set](C→P)→(A→X)→X.
```

Values of type `C V+ A` are constructed from either a value of type `A` or a proof of `C`:

```
Local Nx_unit : (C:Prop)(A:Set)A→(C V+ A) :=
  [C:Prop][A:Set][a:A] [i:C→P][k:A→X](k a).

Local Nx_raise : (C:Prop)(A:Set)C→(C V+ A) :=
  [C:Prop][A:Set][c:C] [i:C→P][k:A→X](e (i c)).
```

In the following we use the abbreviations[4] Nxunit v and Nxraise respectively for **Apply** Nx_unit; **Apply** v and **Apply** Nx_raise.

It is also possible to get an inhabitant of C' V+ A' from (i) an inhabitant of C V+ A, (ii) a function that yields an C' V+ A' from an A and (iii) a proof of C→C':

```
Local Nx_elim_Nx :
 (A,A':Set)(C,C':Prop)(A→(C' V+ A'))→(C→C')→(C V+ A)→(C' V+ A') :=
 [A,A':Set][C,C':Prop]
   [f:A→ C' V+ A'][j:C→C']
     [nx: C V+ A]
       [i:C'→P][k:A'→X](nx ([c:C](i (j c))) ([a:A](f a i k))).
```

Nx_elim_Nx is used when, at the direct style level, we call some function f yielding a useful value a of type A and we don't want to catch an exception possibly raised by f. Technically, when v:C V+ A is available, we use the tactic **Apply** Nx_elim_Nx **with** 3:=v (under the abbreviated form Nxelim v). This yields two subgoals corresponding respectively to f and j above. The second subgoal is purely logical and we generally manage to automatically discharge it using auxilliary lemmas. The first subgoal is handled by an introduction of a of type A, yielding the original goal (Nx C' A'). This is expressed at the direct style level by **let** a=v **in** ...

Finally we need a connection between functions f developed in this section and external world (the "main" function $\mathcal{M}$ – actually any function calling f and catching the exception). Nx_handle builds a value of type X from a proposition P, an exceptional continuation e, a function whose result has type $(Nx\ X\ P\ e\ C\ A)$, a proof of $C \rightarrow P$ and a normal continuation of type $A \rightarrow X$.

```
Definition Nx_handle : (C:Prop)(A:Set)(Nx C A)→(C→P)→(A→X)→X :=
  [C:Prop][A:Set][nx:(Nx C A)]nx.
```

3.2 Dealing with several Exceptions Carrying a Value

In practice one needs to simultaneously handle several exceptions. Moreover they can carry a value, for instance a string to be printed. This is dealt with using a base type Txlev and a type Txval depending on Txlev. Intuitively, Txlev is the type of names of exceptions, and the type of values carried by an exception l, where l inhabits Txlev, is (Txval l). Possible definitions of Txlev and Txval are discussed in appendix A. We provide an exceptional continuation for each member of Txlev.

```
Section algo.
Variable X:Set.
Local Propx:=(l:Txlev)(Txval l)→Prop.
Variable P:Propx.
Variable e:(l:Txlev)(x:(Txval l))(P l x)→X.
```

The remaining definitions are straightforward generalisations of the ones given above, for instance:

[4] In Coq V5.10 user-defined notations and abbreviations are available.

```
Local Nx :=
  [C:Propx][A:Set]((l:Txlev)(x:(Txval l))(C l x)→(P l x))→(A→X)→X.

Local Nx_elim_Nx :
  (A,A':Set)(C,C':Propx)
    (A→(C' V+ A'))→((l:Txlev)(x:(Txval l))(C l x)→(C' l x))→
      (C V+ A)→(C' V+ A') :=
  [A,A':Set][C,C':Propx]
    [f:A→(C' V+ A')]
      [j:(l:Txlev)(x:(Txval l))(C l x)→(C' l x)]
        [nx:(C V+ A)]
          [i:(l:Txlev)(x:(Txval l))(C' l x)→(P l x)][k:A'→X]
           (nx ([l:Txlev][x:(Txval l)][c:(C l x)](i l x (j l x c)))
               ([a:A](f a i k))).
```

The second subgoal mentioned in the discussion about `Nx_elim_Nx` in 3.1 is here dealt with by case analysis on `l`. That is, instead of just **Auto** we use a pattern like: **Intro** `l`; **Elim** `l`; **Simpl**; **Intros** `x Hx`; ... **Auto**.

3.3 Using Impredicativity

Suppose we are in the following common situation.

- The function $\mathcal{M}$ calls a function f which itself calls h.
- Both f and h use exceptions build upon a common pair $\langle \texttt{Txlev}, \texttt{Txval} \rangle$.
- But the exception systems e given to f and h are different. This typically arises when f filters some exceptions raised by h.

This leads to develop f and h in separate sections. Outside of its defining section h has type $\forall X\ \forall P\ (\forall l\ \forall x\ (P\ l\ x) \rightarrow X)(Nx\ X\ P\ D\ B)$ for some D and B. Inside the defining section of f the instance of Nx currently used is locally renamed as Nx_f. The type of f is then $(Nx_f\ C\ A)$ and a call to h in the body of f is represented by a call to `Nx_handle` with h as nx, and $(Nx_f\ C'\ A')$ as X, where $(Nx_f\ C'\ A')$ is the current subgoal.

Now if we examine the types involved in f after the end of the defining section of f, we see that $Nx = \forall X \ldots$ is sometimes used with $X = Nx$.

If we accept to loose the abstraction level provided by the primitives like `Nx_raise`, i.e. if we program directly at the level of continuations impredicativity is no longer needed. The example of 4.1 was earlier developed in this way. There are already similar phenomena with the representation of data structures in system F. For instance boolean negation can be encoded by $\lambda b^{\mathbf{bool}}(b\ \mathbf{bool}\ \mathbf{f}\ \mathbf{t})$, using impredicativity and the "primitive" constants $\mathbf{f}$ and $\mathbf{t}$, or by $\lambda b^{\mathbf{bool}}\lambda X\,\lambda x^X\,\lambda y^X\ (b\ X\ y\ x)$ in a simple typing-like fashion.

4 Two Case Studies

4.1 Computing the Weight of a Tree

Version with one Exception. 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. These trees are built using `leaf` and `node`:

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

We aim at the following algorithm, which travels the tree `t` from right to left while accumulating in `a` the sum of the encountered leaves; as soon as `a` exceeds `m` we know that the answer is `true`; if no exception has been raised the answer is `false`[5]. Notice that the result `r` is not compared to `m` – it is even not used at all. Therefore we must ensure that in the case where the answer is `true` an exception does have to be raised[6].

```
Function core (m:nat;t:tree):nat =
  letrec comprec(t:tree;a:nat):nat =
     match t with
       leaf(n) → (g a+n)
     | node(t1,t2) →
         let an2=(comprec t2 a) in let an=(comprec t1 an2) in an
  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 m t) in false
  with threshold→true.
```

The first thing to do is to state the specification of the final result.

```
Definition P_overweight := [m:nat][t:tree](le m (leaveplus t)).
Definition RESU :=
      [m:nat][t:tree]{(P_overweight m t)}+{¬(P_overweight m t)}.
```

$\{P\}+\{Q\}$ denotes a 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 :=
  [m,a:nat][t:tree](le m (plus a (leaveplus t))).
```

The result of `core` is *exception* if `(leaveplus t)` exceeds `m` and `(leaveplus t)` otherwise. We also want that if the function actually computes `(leaveplus t)` then this value does not exceed `m`. The internal function `comprec` has a similar specification taking the accumulator into account, hence we introduce:

[5] A first version of this development, without accumulator, came from [12]. Two calls to `g` were then needed, one for `leaf` and one for `node`. In the present version there is no real reason to keep `g`, except for showing how such a local function can be dealt with.

[6] This constraint is stronger than the one considered in the example of Castéran.

```
Inductive condsum_accu [m,a:nat;t:tree] : Set :=
    condsum_accu_intro : (n:nat)(n=(plus a (leaveplus t)))→
                ¬(le m n)→(condsum_accu m a t).

Definition condsum_accu_cps :=
   [m,a:nat][t:tree] (P_overweight_accu m a t) V+ (condsum_accu m a t).
```

The specification of `(g m n)` is quite naturally `(le m n) V+ (T_aux m n)` where `T_aux` is defined by:

```
Local T_aux := [m,n:nat]{n':nat|n=n'&¬(le m n')}.
```

The proof/function of `core` has exactly the same structure as the definition given above, see appendix B. One proves the theorem:

```
Theorem core : (m:nat)(t:tree)(condsum_accu_cps m 0 t).
```

Finally the function `F_overweight` is specified by `(m:nat)(t:tree)(RESU m t)` and is easily obtained using `Nx_handle` and `core`. In this process X is instanciated to `(RESU m t)` and we prove `(condsum_accu m 0 t)→(RESU m t)` and `(P_overweight_accu m 0 t)→(RESU m t)` using respectively the witnesses `false` and `true`.

Version with Two Exceptions. The problem can be slightly complicated as follows: *given a list* l *of trees and an integer* m, *compute a list of booleans such that each boolean indicates whether the weight of the corresponding tree exceeds* m, *but abort the whole execution if a zero is detected on one of the leaves of the trees of* l.

This leads to introduce two exception levels, `xl1` and `xl2`. The first carries no value, the second carries a boolean (other choices are possible). Here is a way for specifying this:

```
Inductive Txlev : Set :=  xl1 : Txlev | xl2 : Txlev.
Inductive Txval : Txlev → Set :=
  xv1 : (Txval xl1)   |   xv2 : bool→(Txval xl2).
```

We need to specify assertions about exceptions. The following definition of `(prop_exc Pz Pw)` indicates that if `xl1` has been raised then `Pz` is provable, and if `xl2` has been raised then the carried boolean cannot be `false` and `Pw` is provable.

```
Definition prop_exc: (Pz,Pw:Prop)Propx.
  Unfold Propx; Intros Pz Pw l.
  Case l.
    Intro zer; Exact Pz.    (*first kind of exception*)
    Intro b;                (*second kind of exception*)
      Exact (b=(xv2 true)→Pw) ∧ (b=(xv2 false)→False).
Defined.
```

In the specification of `core` we just replace P_w = `(P_overweight_accu m a t)` by `(prop_exc (Pposs_zer t)` P_w`)`.

```
Definition condsum2_accu_cps :=
  [m,a:nat][t:tree]
    (prop_exc (Pposs_zer t) (P_overweight_accu m a t)) V+
    (condsum_accu m a t).
```

The specification of `g` says that the first kind of exception cannot be raised.

```
 (n:nat) (prop_exc False (le m n)) V+ (T_aux m n).
```

The function working on a list of trees is developed using similar techniques. This function calls `core` via `Nx_handle` using the impredicativity of `Nx` as described in 3.3.

4.2 First Order Unification

Attempting to unify two terms T and U roughly consists in 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 an U are not unifiable", i.e. a failure. The obvious choice for the type of the result is a sum like `<mgu>+<failure>`. In his original development [18] J. Rouyer chose:

```
Inductive Unification[t1,t2:quasiterm]:Set :=
   Unif_succeed:(f:quasisubst)(unif t1 t2)→
        further conditions for f to be an mgu
       →(Unification t1 t2)
 | Unif_fail:(∀f:quasisubst ¬(Subst f t1)=(Subst f t2))
       →(Unification t1 t2).
```

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 behavior of a real implementation.

We proceed from the original development as follows.

- Split the initial definition of `Unification` in two parts, `Unification_s` of kind Set and `Unification_f` of kind Prop:

  ```
  Inductive Unification_s[t1,t2:quasiterm]:Set :=
     Unif_succeed_def:(f:quasisubst)(unif t1 t2)→
          further conditions for f to be an mgu
         →(Unification_s t1 t2).
  Inductive Unification_f[t1,t2:quasiterm]:Prop :=
     Unif_fail_def:(∀f:quasisubst ¬(Subst f t1)=(Subst f t2))
         →(Unification_f t1 t2).
  ```

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

– Redefine `Unification` with `Nx`:

  Definition `Unification :=`
  `[t1,t2:quasiterm] (Unification_f t1 t2) V+ (Unification_s t1 t2).`

  Redefine `Unif_succeed` and `Unif_fail` using respectively `Nx_unit` and `Nx_raise`. Define `Unif_elim` with `Nx_elim_Nx`. The latter plays the rôle of `Unification_rec` although it is less general, see below.
– Adapt the script of the original development.

It turns out that the last step requires very few modifications, and that they are systematic. They split into two classes.

1. Replacing **Elim** H by `Unifelim` H (an abbreviation for **Apply** `Unif_elim` **with** `3:=`H) when H has type `(Unification` t u`)`. In this development the current subgoal is always of type `(Unification` t' u'`)`, and this fits well the restriction on the use of `Unif_elim` mentioned above. Otherwise we should have used the more primitive `Nx_elim_Nx`.
2. 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 was expressed in a usual programming language: each statement returning the value *failure* would be systematically replaced by a statement raising an exception.

5 Concluding Remarks

We have shown that it is possible to add – or to simulate the addition of – control operators to the programming language of extracted programs in the framework of calculus of constructions. All the examples mentioned in this paper have been completely and mechanically verified.

Writing a lambda-term using the introduced primitives is not to be recommended, because explicit typing makes the terms quite heavy. However the tactics language of Coq can be seen as a programming language, via Curry-Howard isomorphism. For instance **Intro** `a`; **Exact** `E` is just a peculiar syntax of $\lambda a.\, E$.

In this language types are most of the time inferred. This is an advantage when using constructs with long types, like `Nx_xxx`. In other respects scripts can be fully understood only when interactively input in the system.

Note that thanks to the tactic `Realizer` recently introduced in Coq one can give the algorithmic structure of a proof using a more conventional syntax called *Real*. Further work is needed to see how this tactic can be combined with our primitives. Another approach would be to internalize these primitives in *Real*.

Acknowledgement

I wish to thank the members of the Coq Project, for their help on Coq, and Pierre Crégut and Chet Murthy for their explanations on continuations. Pierre Lescanne suggested the example of first order unification.

References

1. J-R. Abrial. The B-Book. Prentice Hall, 1994 (in preparation).
2. H.P. Barendregt. Lambda Calculi with Types. In S. Abramsky & al., editors, Handbook of Logic in Computer Science, vol 2, S. Abramsky & al. Eds. 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, 76:95–120, 1988.
5. Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, Proceedings of Colog'88. Springer Verlag, 1990. LNCS 417.
6. E.W. Dijkstra. A Discipline of Programming. Prentice-Hall, 1976.
7. G. Dowek & al. The Coq Proof Assistant User's Guide. version 5.8, INRIA-Rocquencourt et CNRS-ENS Lyon, fév. 1993.
8. J-Y. Girard. A new constructive logic: classical logic. Mathematical Structures in Computer Science, vol 1, pp. 225–296, 1991.
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. POPL, Orlando, 1990.
11. S. Hayashi, H. Nakano. PX, a Computational Logic. Foudations of Computing, MIT Press, 1988.
12. J.L. Lawall and O. Danvy. Separating Stages in Continuation-Passing Style Transformation. POPL, 1993.
13. C. Morgan. Programming form Specification. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.
14. C. Murthy. An evaluation semantics for classical proofs. LICS, Amsterdam, 1991.
15. C. Paulin-Mohring. Extraction de programmes dans le calcul des constructions. thèse de doctorat de l'université Paris VII, 1989.
16. C. Paulin-Mohring. Inductive Definitions in the system Coq; Rules and Properties. In M. Bezem and J.F. Groote, editors, Proceedings of TLCA'93. Springer Verlag, 1993. LNCS 664.
17. C. Paulin-Mohring and B. Werner. Synthesis of ML Programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
18. J. Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Research Report 1795, INRIA-Lorraine, nov. 1992.

A Dependent Types for Exceptions

The base type `Txlev` for exceptions is generally a simple enumeration. It is unfortunately impossible in the calculus of constructions to define `Txval: Txlev→Set` by

elimination on the argument. However we can obtain an isomorphic structure using an inductive definition. For instance in the example above we consider two exceptions `xl1`, which carries nothing, and `xl2` which carries a boolean. Let us suppose here that `xl1` and `xl2` carry respectively a value of type A and B.

```
Inductive Txlev : Set :=  xl1 : Txlev   |   xl2 : Txlev.
Inductive Txval : Txlev→Set :=
    xv1 : A→(Txval xl1)   |  xv2 : B→(Txval xl2).
```

It is useful and easy (in goal mode) to define left inverses `xl1_A` and `xl2_B` of `xv1_B` and `xv2_B` (we consider only the latter in the sequel), at least when `B` is not empty. But this assumption gives a special rôle to a specific inhabitant of `B`. This can be avoided using an auxilliary function defined on `(Txval l)` with `l=xl2`

```
Definition xl2_B_eq : (l:Txlev)(Txval l) → l=xl2 →B.
  Destruct 1; Intros v e; Discr e Orelse Exact v.
Defined.
Definition xl2_B : (Txval xl2)→B.
  Intro v2; Apply (xl2_B_eq xl2 v2); Auto.
Defined.
```

It is also useful to prove that `xl2_B` is really a left inverse.

```
Theorem left_inv_xv2 : ∀v:(Txval xl2) v=(xv2 (xl2_B v)).
```

A method is to define a conversion function

```
Definition l_xl2 : ∀l:Txlev (l=xl2)→(Txval l)→(Txval xl2).
```

by induction on `l`, and to prove

```
∀l:Txlev ∀v:(Txval l) ∀e:l=xl2 (l_xl2 l e v)=(xv2 (xl2_B (l_xl2 l e v))).
```

We get the desired theorem by using `(l_xl2 xl2 e v)=v`. As an interesting consequence, we get the induction principle :

```
Theorem xv2_rec :
  ∀P:(Txval xl2)→Set (∀b:B (P (xv2 b)))→ ∀v:(Txval xl2) (P v).
```

Reciprocally `left_inv_xv2` is a special case of `xv2_rec`. Christine Paulin pointed out that theorems like `xv2_rec` have an independent proof to me. The trick is to define an alter ego of `(eq_rec Txlev)`.

```
Definition eqTxlev_rec :
  ∀P:Txlev→Set ∀k,l:Txlev (eqTxlev k l)→(P k)→(P l).
  Destruct k; Destruct l; Simpl; Intros; Trivial Orelse Contradiction.
Defined.
```

where `(eqTxlev)` is a typical auxilliary function used in "inversion lemmas":

```
Definition eqTxlev : Txlev→Txlev→Prop :=
  xl1 xl1 => True   |  xl1 xl2 => False  |
  xl2 xl1 => False  |  xl2 xl2 => True.
Defined.
```

Of course dependent types like `Txval` are needed in other situations, such as the modelling of records.

B Constructive Proof of core

B.1 Coq Script

In the following script, Split_condsum n H1 H2 is an abbreviation for **Intro** s;**Elim** s;**Clear** s; **Intros** n H1 H2. Hence NxElim R; [Split_condsum an1 Han1t1 Hman1 | **Auto**] means that we compute R of type P V+ (consum_accu ...), and in case of success we decompose the result into an integer n and two hypotheses H1, H2 on n.

```
Section algo.

Variable X:Set.   Variable P:Prop.   Variable e:P→X.
Local Nx := [C:Prop][A:Set](C→P)→(A→X)→X.

Local Nx_unit : (C:Prop)(A:Set)A→(C V+ A).
...

Theorem core : (m:nat)(t:tree)(condsum_accu_cps m O t).
Intros m t.
Cut (n:nat) (le m n) V+ (T_aux m n).
  Intro g.
  Cut (a:nat)(condsum_accu_cps m a t);
     [Intro comprec; Apply comprec | Idtac].

  (* definition of comprec *)
  Unfold condsum_accu_cps; Elim t.
    Intros n a.
    Nxelim (g (plus a n)); [Intro sn'; Nxunit Realizer sn' | Auto].

    Intros t1 R1 t2 R2 a.
    NxElim (R1 a);   [Split_condsum an1 Han1t1 Hman1 | Auto].
    NxElim (R2 an1); [Split_condsum an Hant2 Hmn | Rewrite Han1t1;Auto].
  NxUnit Realizer an.
  Rewrite Hant2; Rewrite Han1t1; Rewrite plus_assoc_r; Simpl; Auto.

  (* definition of g *)
  Intro n; Case (le_dec m n).
    Intro Hlemn; Nxraise; Assumption.
    Intro Nlemn; Nxunit Realizer n.
Save.

End algo.
```

The proof is almost the same in the case with two exceptions. Nxraise must be used with an argument, for instance in the definition of g:

```
Intro Hlemn; Nxraise (xv2 true); Simpl; Auto.
```

A few tactics must be added in order to automatically discharge the propositional subgoal of Nx_elim_Nx, by case on 1, for instance concerning (R2 a):

```
Nxelim (R1 a);
     [Split_condsum an1 Han1t1 Hman1 |
      Intro l; Elim l; Simpl; Intros v Hv; Elim Hv; Auto].
```

B.2 Extracted program

The program obtained with extract and conversion to Caml code, in Coq V5.8, is as follows (plus is expanded in the actual code):

```
let plus = fun n -> fun m ->
  let rec REC1 = function O_C -> m | S_C VAR3 -> S_C ((REC1 VAR3)) in
  REC1 n;;

(* core : 'a -> nat -> tree -> (nat -> 'a) -> 'a *)
let core = fun e -> fun m -> fun t ->
  let rec REC1' = function
     leaf_C VAR3 ->
       (fun a -> fun k ->
         match le_dec m (plus VAR3 a) with
            true_C -> e
          | false_C -> k (plus VAR3 a))
   | node_C(VAR4,VAR5) ->
       (fun a -> fun k ->
          REC1' VAR4 a (fun a' -> REC1' VAR5 a' (fun a' -> k a')))
  in REC1' t (O_C);;
```

Replacing nat by int, plus by + and renaming bound variables gives :

```
type itree = ileaf_C of int | inode_C of itree * itree;;

let core e m t =
  let rec COMPREC = function
     ileaf_C n ->
       (fun a -> fun k ->  if m < n+a then e else k (n+a))
   | inode_C(t1,t2) ->
       (fun a -> fun k ->
          COMPREC t1 a (fun a' -> COMPREC t2 a' (fun a' -> k a')))
  in COMPREC t 0;;
```

We recognise the CPS translation of the intended function where g has been unfolded.

This article was processed using the LaTeX macro package with LLNCS style