ProPre
A Programing Language with Proofs

Pascal MANOURY Marianne SIMONOT

Equipe de logique
CNRS URA 753
UFR de mathématiques,
Université Paris VII,
2 place Jussieu,
75251 Paris cedex 05

1 Introduction

Because of the increasing role of computers in our world, the problem of the correctness of programs becomes a crucial one. A lot of work in computer science is concerned with. Through the Heyting's semantics, the intuitionistic logic gives the opportunity to build a programing language which correctness is guaranteed. From this point of view, proofs are nothing else but programs.
Considering the general paradigm of *proofs as programs*, we intend to exploit the correspondance between intuitionnistic proofs and λ-calculus, namely : the typed λ-calculus. So far that λ-calculus can be regarded as a low level language, typed λ-calculus gives us an implementation of logic regarded as high level language.
Usually, programs are extracted from a proof of the logical specification of a given problem (ie : a proof of a formula of the form : $\forall \overline{x} \exists y P(\overline{x}, y)$). We adopt a different approch by introducing the notion of algorithm between the problem specification and the program. Rougthly speaking :

- the specification of the problem is a logical formula ;
- the algorithm is an equationnal definition of function ;
- the program is a term of λ-calculus.

All of these three objects are mathematical ones. On one side we can prove that the algorithm meets the specification using usual mathematics (including classical logic) ; on the other side we can obtain a correct program by proving, using intuitionnistic logic, that the algorithm is a total function of the intended type.

The system of typed λ-calculus we use is TTR[1] (for Théorie des Types Récursifs). It is based on second order predicate calculus with equality extended with a least fixed point operator. This added fixed point operator allows definitions of recursive data types which gives us their *intended implementation*.
The work we want to present is only concerned by the second part of our approch : the search of termination proofs for equationaly defined functions.

2 The language

We saw that the theory offers us the opportunity for building a programing language in which programs are correct. But what about the pratical point of view ?
One can consider that the programmer has to become a logician and write proofs instead of programs. But it seems to us that programmers have to concentrate their activity in elaborating algorithms.
That is why we try to build a programming language being as near as possible to usual functional ones (LISP, ML, etc ...). This leads us to develop automatical proof strategies for termination so that, in our language, the search of termination proofs (from which the programs will be extracted) can be identify with the compilation phase of other languages.
In our language, the programmer has only to define data types and functions.
Let us present now, for data types and functions, successively the syntax proposed to the user and the internal process applied on those syntactic objects by our system.

2.1 Data types

The syntax of data types defintion is close to ML's one. For instance, here is the definition of lists of natural number :
`Type Ln :  Nil | Cons N Ln ;`
The system interprete this definition as the following formula :

$$\mu K \forall X (X(Nil) \to (\forall n \forall y (N(n), K(y) \to X(Cons\ n\ y)) \to X(x)))$$

Where μ is the logical least fixed point operator. This formula, which will be called $Ln(x)$, gives us the smaller set containning the constant Nil and closed by the constructor $Cons$.
We obtain program for each constructor from proofs of the formulae :

$$Ln(Nil)$$

$$\forall n \forall x (N(a) \to (Ln(x) \to Ln(Cons\ n\ x)))$$

We obtain two kinds of recursion operators by proving the formulae :

$$\forall x \forall P (Ln(x) \to (P(Nil) \to \forall n \forall y (N(n), Ln(y), P(y) \to P(Cons\ n\ y)) \to P(x))$$

[1]This system has been design by M. Parigot extending the works of D, Leivant and J.L. Krivine on this subject

$$\forall x \forall P(Ln(x), \forall y(Ln(y), \forall z(Ln(z), z < y \rightarrow P(z)) \rightarrow P(y)) \rightarrow P(x))$$

Where $<$ is the structural well ordering of the data type.Those two formulae correspond, in fact, to two way of reasonning by inductions on the objects of type Ln.
For each multisorted algebra there is a canonical way to get proofs of those formulae from which we extract programs for constructors and recursion operators.

2.2 The functions

The syntax of function definitions is inspired by ML's one. Let us see for instance the defintion of the `append` function :

```
Let append : Ln, Ln -> Ln
        Nil y => y
        (Cons n x) y => (Cons x (append x y))
;
```

Remark that :

- on the left side of => each name of a variable can't appear more than once ;
- on the left side of => each functional symbol can only be a data type constructor's name ;
- each variable name occuring on the right side of => must occur on the corresponding left side ;
- each functional symbol appearing on the right side of => must be either an already known functional symbol or the defined function name.

Given this definition, the deterministic strategy of the system tries to find a proof of :

$$\forall x(Ln(x) \rightarrow \forall y(Ln(y) \rightarrow Ln(append\ x\ y)))$$

2.3 About the strategy

The basic idea of the strategy is that the equationnal definition is a guide for the proof. That is to say : if a function is defined by recursion on one (or many) argument(s), we have to search a proof by induction on this same argument. This method presents two advantages :

1. if the strategy succeeds then the program obtained has the behaviour expected by the programmer ;
2. if the strategy fails, we can provide error messages in terms suggested by the programer's definition.

Rouhgtly speaking, the proofs obtained by our strategy can be divided in three parts. Starting from the statement to prove which is the root of the proofs tree we are looking for, :

a) we first build an initial sub-proof tree by using only induction scheme

b) then, on each leaf of this partial proof tree, we rewrite the resulting formula using the corresponding equation in the function definition

c) finally, we use induction hypotheses, termination theorems of already defined functions and axioms in order to achieve the proof.

3 To conclude

We tried successfully our strategy on classical functionnal algorithms as : Ackermann function, Quick sort, Heap sort, etc ...
Around this first strategy we implement the kernel of an experimental functional language called ProPre. It is interfaced with a λ-calculus machine on which we can execute the programs obtained by proofs. This is this system we intend to present at the LPAR 92.