Programming + Verification = Progification
(Draft)

Thorsten Altenkirch

September 94

Abstract

We discuss the rôle Type Theory should play in the formal development of correct programs. We view verification as a programming problem in a sophisticated programming language and evaluate this approach by presenting a number of examples developed in the ALF system. Thereby we exploit the recent advantages in the presentation of Type Theory in particular by using pattern matching as proposed in [Coq92].

1 Introduction

The steps in the development of a software system can be classified in **exploration** and **consolidation**. Exploratory steps are the implementation of a prototype or the experimental addition of new features to an existing system. Examples of consolidation are debugging, testing, extending and updating documentation, etc.

One important objective of a good software development methodology is to support consolidation without making exploration too hard. We have to reject an anarchic approach to software development (*hacking*) because it ignores the need for consolidation. On the other hand we also have to reject a discipline which is too rigid because it ignores the need for exploration.

A specification is a part of the documentation. As such it will usually be the case that it is not the main content of the first exploratory steps of a project. Any initial specification, like a statement of requirements, will often be rather vague. Finding out the precise requirements is already part of the development process. Thus we reject the idea of starting with a formal specification.

However, we believe that formal specifications and computer aided verification can play an essential part in the later consolidation steps of software development. Formal verification makes it possible to deliver code with a degree of confidence in its correctness that is essentially higher than with any other approach. If this different quality of consolidation is an essential feature of the

system it may well be worth the price in development costs one has to pay for it.

We already know that the costs of formally verifying a program is much higher than the cost of developing it. Therefore it is much more important to reduce the cost of verification than the cost of programming. A programming language like C may be a good language to program in but it is rather doubtful that it is a good language for program verification. Even, if we have to completely rewrite a prototype to make it verifiable, this may still be a minor factor in the cost of the complete verification.

Here we propose to use Type Theory as a language for the formal development of correct programs. We view Type Theory as a programming language in which it is possible to express specifications by the type of a program. Thus we do not view Type Theory as a language to reason *about* programs but as a programming language itself. We have coined the word "Progification", since developing and verifying programs is reduced to writing programs in a pure functional language with dependent types, subject to some restrictions to ensure that the programs/proofs are total.

This is essentially the original approach of using Type Theory for program verification as proposed by Martin Löf and many others, e.g. see [NPS90, BCMS89]. However, it has been proposed to reintroduce a difference between proofs and programs and between data types and propositions in Type Theory - either for pragmatic [PM89] or for philosophical reasons [Luo94]. We shall attempt to show by means of example that a pure approach is not only feasible but also preferable. Here we exploit the progress in the presentation of Type Theory which has been made, especially by using the pattern matching notation as proposed by T. Coquand [Coq92].

In the rest of the paper we shall present and discuss some examples of *progification* using the ALF system [MN94, AGNvS94] to illustrate our approach.

2 An introductory example

Consider the following definition of the function le - less than on natural numbers - as a simple functional program. We first define the basic types Bool and Nat inductively:

```
Bool ∈ Set
  true ∈ Bool
  false ∈ Bool
Nat ∈ Set
  0 ∈ Nat
  s ∈ (Nat) Nat
```

This corresponds to `datatype` declarations in ML. The function le can be defined as a non canonical constant using pattern matching and recursion as in

ML:

```
le ∈ (Nat; Nat) Bool
  le(0, j) ≡ true
  le(s(i), 0) ≡ false
  le(s(i), s(j)) ≡ le(i, j)
```

One can say that the definition of this function is so simple that it does not require a specification. However, it is not obvious how to reason about le, e.g. how to prove transitivity. The most naive approach - induction on natural numbers - will end up in a combinatory explosion of cases. Our experience is that it is easier to reason about inductively defined relations than about boolean valued functions. It is also more general because we are not restricted to decidable relations, indeed we may not be interested in the decidability of a relation even if it is decidable.

Thus, we define a relation Le as follows:

```
Le ∈ (Nat; Nat) Set
  le0 ∈ (i ∈ Nat) Le(0, i)
  leS ∈ (i, j ∈ Nat; Le(i, j)) Le(s(i), s(j))
```

Le is an inductively defined family of sets, indexed by pairs of natural numbers. Thus the basic idea is the same as for the previously defined Bool and Nat, the only difference is that we use dependent types. Indeed, it is straightforward to extend the set-theoretic explanation of an inductive type as the least fix point of a monotone operator to families of sets. The same holds for the categorical view of inductive types as initial T-algebras, dependent inductive types can be viewed as initial T-algebras in a slice category.

The proof of transitivity of Le is a non-canonical constant defined by pattern matching [1]:

```
transLe ∈ (i, j, k ∈ Nat; Le(i, j); Le(j, k)) Le(i, k)
  transLe(_, j, k, le0(_), h1) ≡ le0(k)
  transLe(_, _, _, leS(i1, j1, h2), leS(_, j, h)) ≡
    leS(i1, j, transLe(i1, j1, j, h2, h))
```

We continue our analogy - there is no fundamental difference between the program le and the program transLe but that the second uses dependent types. However, to accept transLe as a valid proof we have to apply stricter rules then the ones which are checked by a compiler for a functional language like ML. In particular we have to be sure that the function defined is *total*.

This can be established by the following facts: The left hand sides of the pattern defining transLe constitute a complete covering, e.g. every closed canonical instance of the domain is the substitution instance of a left hand side [2] .

[1] The *wild cards* _ are introduced by ALF to avoid non-left-linear patterns. Thus the Church-Rosser property is preserved.

[2] Actually we also require that the cases are non-overlapping, e.g. that there is a unique

On the other hand it has to be established that the function is terminating, i.e. that there is a well founded ordering on the domain which is reduced during recursion.

Both conditions are in general not decidable, but we shall restrict ourselves to a decidable subset, which we believe is sufficient for practical purposes. As far as the completeness is concerned we satisfy ourselves with patterns whose completeness can be shown using a slight generalisation of first order unification. As far as termination is concerned we reduce the problem to structural recursion, i.e. to recursion which reduces the structure of the inductively defined arguments [3].In the current implementation of ALF the completeness of the pattern is guaranteed by the interactive generation of the pattern by the user [4]. The test of the termination condition is currently not implemented which we consider as a serious shortcoming since this opens a loophole for errors in the verification process.

A short remark about notation is appropriate here: Dependently typed programs contain a lot of redundant information which tends to make the code unreadable. Therefore we will hide arguments which can be inferred from the context in the presentation, e.g. the last two definitions can be presented as follows:

```
Le ∈ (Nat; Nat) Set
  le0 ∈ Le(0, i)
  leS ∈ (Le(i, j)) Le(s(i), s(j))
transLe ∈ (Le(i, j); Le(j, k)) Le(i, k)
  transLe(le0, h1) ≡ le0
  transLe(leS(h2), leS(h)) ≡ leS(transLe(h2, h))
```

E.g. the first argument to le0, the first two to leS and the first three to transLe are hidden. However, their types can be easily inferred from the usage. Note that argument hiding has the same goal as type inference in a language like ML but it has the advantage that it also works for a language for which type inference is not computable as it is the case for almost every language with dependent types.

So far we did not relate the boolean valued function le and the relation Le, e.g. prove that le decides Le. We will actually prove a slightly different statement, namely that le(i,j) returns true iff $i \leq j$ and false iff $j < i$.

matching left-hand side. This, again, is necessary for the Church-Rosser property.

[3]For a more detailed description see [Coq92].

[4]It is a shortcoming of the current version that there is no trace of this in the proof term, i.e. it cannot be checked afterwards that the pattern is complete.

```
leLem1 ∈ (i, j ∈ Nat; Id(le(i, j), true)) Le(i, j)
  leLem1(0, j, h) ≡ le0
  leLem1(s(i), s(j), h) ≡ leS(leLem1(i, j, h))
leLem1' ∈ (i, j ∈ Nat; Le(i, j)) Id(le(i, j), true)
  leLem1'(_, j, le0) ≡ id(le(0, j))
  leLem1'(_, _, leS(h1)) ≡ leLem1'(i1, j1, h1)
leLem2 ∈ (i, j ∈ Nat; Id(le(i, j), false)) Lt(j, i)
  leLem2(s(i), 0, h) ≡ leS(le0)
  leLem2(s(i), s(j), h) ≡ leS(leLem2(i, j, h))
leLem2' ∈ (i, j ∈ Nat; Lt(j, i)) Id(le(i, j), false)
  leLem2'(_, 0, leS(h1)) ≡ id(le(s(j1), 0))
  leLem2'(_, s(h), leS(h1)) ≡ leLem2'(j1, h, h1)
```

Here identity Id is defined inductively and strictly less than Lt by an explicit definition:

```
Lt ∈ (Nat; Nat) Set
  Lt ≡ [h, j]Le(s(h), j)
Id ∈ (a, b ∈ A) Set
  id ∈ (x ∈ A) Id(x, x)
```

The proof of the leLem1,leLem2 follows closely the recursive structure of `le`, we are doing the same analysis several times. This may be comparatively harmless for a function like `le` but it becomes a more serious factor for bigger functions. Another observation is that wherever we use `le` in a program we will use these lemmas in almost all proofs about this program.

Therefore we propose not to implement le at all but to use a dependently typed function instead which corresponds to a proof of trichotomy, e.g. we show that for each $i, j \in \mathsf{Nat}$ we have that either $i \leq j$ or $j < i$. We could present this specification using standard connectives but we feel that it improves the readability of programs if we use more specific definitions, e.g. we define the family Trich representing the solution to the problem:

```
Trich ∈ (Nat; Nat) Set
  isLe ∈ (i, j ∈ Nat; Le(i, j)) Trich(i, j)
  isGt ∈ (i, j ∈ Nat; Lt(j, i)) Trich(i, j)

le' ∈ (i, j ∈ Nat) Trich(i, j)
  le'(0, j) ≡ isLe(0, j, le0)
  le'(s(i), 0) ≡ isGt(s(i), 0, leS(le0))
  le'(s(i), s(j)) ≡ case le'(i, j) ∈ Trich(i, j) of
                      isLe(_, _, h2) ⇒ isLe(s(i), s(j), leS(h2))
                      isGt(_, _, h2) ⇒ isGt(s(i), s(j), leS(h2))
                    end
```

Note the close similarity of le and le'. Indeed we can view le' as a dependently

typed version of le by identifying isLe with true and isGt with false. It seems that le' is a more compact presentation of both le and its essential properties. Another advantage is that we can use le' as a building block to derive other dependently typed programs or proofs.

However, apparently le' is less efficient since it always analyses the recursive case whereas we were able to use a simple identity in the case of le. Thus if we want to generate efficient code from such programs we have to do some optimisations which are not necessary in conventional code. In this case we have to optimise a program of the form if x then true else false to x which is an instance of the η-rule for Bool.

We will later see other optimisations which are necessary to recover the conventional versions of algorithms. For the examples we have considered so far these optimisations could be all done by a simple static analysis of the program.

3 Progifying sorting

Our next goal is to apply the progification approach to simple sorting algorithms, e.g. to insertion sort and merge sort. The second case is interesting since the algorithm is not structurally recursive.

We start by translating the sorting problem to Type Theory. In a simply typed programming language (like ML) we may expect that the sorting function has a type like:

```
val sort = fn : ('a * 'a -> bool) -> 'a list -> 'a list
```

Here we want to find a *finer* type which does only contain sorting programs. We define lists and two predicates Adjoin and Perm inductively:

```
List ∈ (A ∈ Set) Set
  nil ∈ List(A)
  cons ∈ (a ∈ A; l ∈ List(A)) List(A)
Adjoin ∈ (A; List(A); List(A)) Set
  ad0 ∈ Adjoin(a, xs, cons(a, xs))
  ad1 ∈ (Adjoin(a, xs, axs);
         b ∈ A) Adjoin(a, cons(b, xs), cons(b, axs))
Perm ∈ (List(A); List(A)) Set
  perm0 ∈ Perm(nil, nil)
  perm1 ∈ (Perm(xs, ys);
           Adjoin(a, xs, axs);
           Adjoin(a, ys, ays)) Perm(axs, ays)
```

We say that a list is sorted if every element is related to all it's successors w.r.t. some relation R:

```
ListAll ∈ (P ∈ (A) Set; List(A)) Set
  la0 ∈ ListAll(P, nil)
  la1 ∈ (P(a); ListAll(P, l)) ListAll(P, cons(a, l))
Sorted ∈ ((A; A) Set; List(A)) Set
  sort0 ∈ (A ∈ Set; R ∈ (A; A) Set) Sorted(R, nil)
  sort1 ∈ (Sorted(R, l); ListAll([h]R(a, h), l)) Sorted(R, cons(a, l))
```

This is sufficient to specify what a solution to a sorting problem is: given a list l we request another list l' together with proofs that l' is a permutation of l and that l' is sorted:

```
SortSpec ∈ (A ∈ Set; R ∈ (A; A) Set; List(A)) Set
  sortSpec ∈ (A ∈ Set;
              R ∈ (A; A) Set;
              l, l' ∈ List(A);
              Perm(l, l');
              Sorted(R, l')) SortSpec(A, R, l)
```

We cannot expect to have a solution to every sorting problem, but we require some additional properties about the relation R. To be able to construct a sorted version of a list at all it is essential that the relation is total. Another natural property we require is that the R is transitive.

```
Tot ∈ (R ∈ (A; A) Set) Set
  tot ∈ ((a, a' ∈ A) Plus(R(a, a'), R(a', a))) Tot(R)
Trans ∈ (R ∈ (A; A) Set) Set
  trans ∈ ((a, a', a'' ∈ A; R(a, a'); R(a', a'')) R(a, a'')) Trans(R)
```

Note that these are quite trivial inductive definitions. However, we find that by using these definitions we improve the readability of definitions compared with the using Σ and other standard connectives.

Our sorting programs will all have the following type:

```
sort ∈ (r ∈ Tot(R); tr ∈ Trans(R); l ∈ List(A)) SortSpec(A, R, l)
```

It is straightforward to prove that le is total form the proof of trichotomy in the previous section. Together with the proof of transitivity this enables us to derive a sorting function for Nat from the general one.

How does the ALF type of sorting programs compares with the ML type? Certainly we can project the sorted list out of SortSpec. The proof of totality corresponds to the boolean function. However, the proof of transitivity turns out to have no computational use at all. A simple analysis of the sorting programs we verified shows that the proof of transitivity is only used to construct the proof that the resulting list is sorted - therefore we can optimise the programs by omitting that argument.

3.1 Insertion sort

Our prototype is the following ML program:

```
fun insert lt a [] = [a]
  | insert lt a (l as (b::bs)) =
      if lt(a,b) then a::l
      else b::(insert lt a bs);

fun sort lt [] = []
  | sort lt (a::xs) =
      insert lt a (sort lt xs);
```

Our goal is to refine the program such that we obtain a an ALF program of the appropriate type. Note that there is no problem with totality, since the prototype already full fills the requirements we discussed above.

We also have to find a more refined type for

```
insert : ('a * 'a -> bool) -> 'a -> 'a list -> 'a list+:
```

which is given by

InsertSpec ∈ (a ∈ A; l ∈ List(A)) Set
insertSpec ∈ (a ∈ A;
l, l' ∈ List(A);
Adjoin(a, l, l');
Sorted(R, l')) InsertSpec(a, l)

The progification of `insert` requires two lemmas whose derivation is straightforward:

mapLA ∈ ((a ∈ A; P(a)) Q(a); l ∈ List(A); ListAll(P, l)) ListAll(Q, l)
laAdjoinLem ∈ (ListAll(P, xs);
P(a);
Adjoin(a, xs, axs)) ListAll(P, axs)

We present here the complete derivations of insert (figure 1) and sort (figure 2) to illustrate two things:

- The proofs are quite compact and readable, this should be compared with a presentation of a tactic-based proof.
- The close similarity of the proofs and the original algorithm.

The code should be understandable because of its similarity with standard functional programs as written in ML or Haskell. Note the use of case to deconstruct the recursive calls. This corresponds to the use of `let` in ML with a non trivial pattern on the left hand side.

```
insert ∈ (r ∈ Tot(R);
          t ∈ Trans(R);
          a ∈ A;
          l ∈ List(A);
          ss ∈ Sorted(R, l)) InsertSpec(a, l)
  insert(r, t, a, nil, ss) ≡
    insertSpec(a,
               nil,
               cons(a, nil),
               ad0(a, nil),
               sort1(sort0(R), la0([h]R(a, h))))
  insert(tot(h), trans(h1), a, cons(a1, l1), sort1(h2, h3)) ≡
    case h(a, a1) ∈ Plus(R(a, a1), R(a1, a)) of
      inl(h4) ⇒
        insertSpec(a,
                   cons(a1, l1),
                   cons(a, cons(a1, l1)),
                   ad0(a, cons(a1, l1)),
                   sort1(sort1(h2, h3),
                         la1(h4, mapLA([a', h5]h1(a, a1, a', h4, h5), l1, h3))))
      inr(h4) ⇒
        case insert(tot(h), trans(h1), a, l1, h2) ∈ InsertSpec(a, l1) of
          insertSpec(–, –, l', h5, h6) ⇒
            insertSpec(a,
                       cons(a1, l1),
                       cons(a1, l'),
                       ad1(h5, a1),
                       sort1(h6, laAdjoinLem(h3, h4, h5)))
        end
    end
```

Figure 1: insert

```
sort ∈ (l ∈ List(A)) SortSpec(l)
  sort(nil) ≡ sortSpec(nil, nil, perm0(A), sort0(R))
  sort(cons(a, l1)) ≡
    case sort(l1) ∈ SortSpec(l1) of
      sortSpec(–, l', h, h1) ⇒
        case insert(r, t, a, l', h1) ∈ InsertSpec(a, l') of
          insertSpec(–, –, l'1, h2, h3) ⇒
            sortSpec(cons(a, l1), l'1, perm1(h, ad0(a, l1), h2), h3)
        end
    end
```

Figure 2: sort (*insertion sort*)

3.2 General recursive sorting algorithms

It is well known that insertion sort and in fact any structural recursive implementation of sorting is inherently inefficient. How can we *progify* a sorting algorithm like merge sort, e.g. given by the following prototype:

```
fun split [] = ([],[])
  | split (a::xs) =
      let val (ys,zs) = split xs
      in (a::zs,ys)
      end;

fun merge [] ys = ys
  | merge xs [] = xs
  | merge (l as (a::xs)) (m as (b::ys)) =
      if a<b then a::(merge xs m)
      else b::(merge l ys);

fun sort [] = []
  | sort [a] = [a]
  | sort xs =
      let val (ys,zs) = split xs
      in merge (sort ys) (sort zs)
      end;
```

There is no problem with `split` and `merge` which are structural recursive, but `sort` obviously is not. There are two principal ways out of the dilemma:

1. Allow general recursive but terminating definitions in Type Theory.
2. Make the termination ordering on the arguments explicit to obtain a structural recursive program.

The first way will certainly lead to short and elegant proofs but has the disadvantage that the correctness of the proof, which depends on its totality cannot be checked anymore. Note that this does not only mean that the program may not terminate but it may even terminate but fail to have some properties which are proven by lemmas which are not total. If our goal is to obtain certified correct programs then this approach has to be rejected. The approach to generalise the termination condition does not seem to be promising either, since the termination proof may be arbitrarily complicated and cannot be generated mechanically.

The second way seem to have the disadvantage that it spoils the goal: by rewriting the program this way we loose any possible gain of efficiency due to using a more efficient recursion scheme. However, we will see that this is not the

case, indeed it is straightforward to optimise the resulting program such that the gain of efficiency is maintained.

This can be exemplified by using the inductive presentation of general well founded recursion in Type Theory as proposed by [Nor88, Pau86]. We define the predicate Acc which defines the accessible subset of a relation [5]. We can define Acc in ALF inductively:

$$\begin{array}{l} \mathsf{Acc} \in ((A; A)\,\mathsf{Set};\ A)\ \mathsf{Set} \\ \quad \mathsf{acc} \in (a \in A;\ (b \in A;\ R(b, a))\ \mathsf{Acc}(R, b))\ \mathsf{Acc}(R, a) \end{array}$$

From these it is possible to derive a *typed fix point combinator* which given a function which during recursion only accesses smaller arguments can calculate the fix point of this function for all element for which the relation is accessible. This can be shown by a simple structural recursion over Acc

$$\begin{array}{l} \mathsf{rec} \in (f \in (a \in A;\ (b \in A;\ R(b, a))\ B)\ B;\ a \in A;\ \mathsf{Acc}(R, a))\ B \\ \quad \mathsf{rec}(f, a, \mathsf{acc}(\_, h_1)) \equiv f(a, [b, h]\mathsf{rec}(f, b, h_1(b, h))) \end{array}$$

Note that using an instance of the higher order argument is covered by the definition of structurally smaller as given in [Coq92].

We argue now that the last argument is only used for the calculation of the last argument. For this argument to be valid it is important that the pattern is already *linearised*, otherwise there would be a possible dependency from the first argument off acc in the left hand side of the pattern. Furthermore removing the last arguments collapses no cases. Therefore it has no computational relevance and can be eliminated. This means that from the typed fix point combinator we can extract the ML version of a fix point combinator:

```
fun fix f a =
    f a (fn b => fix f b);

fix : ('a -> ('a -> 'b) -> 'b) -> 'a -> 'b
```

It is interesting to note that rec gives us indeed a partial algorithm, which terminates only for applications for which a proof of accessibility exists.

Moreover, even for specific instances of rec it is the case that the optimised algorithm is no longer strongly normalising. However, it is still weakly normalising, e.g. excluding reductions under a λ-abstraction and inside a pattern. Since all functional programming languages implement weak reduction strategies, this does not seem to be a serious shortcoming.

Using this strategy we completely verified merge sort. That is we *progified* the recursive functional underlying `sort`:

[5] An example is Acc(red1) where red1 is the one-step reduction relation. Then Acc(red1) is the subset of strongly normalising terms. This is precisely the presentation we used in [Alt93]

sortRec ∈ (A ∈ Set;
R ∈ (A; A) Set;
r ∈ Tot(R);
tr ∈ Trans(R);
l ∈ List(A);
sort ∈ (l_1 ∈ List(A); Lt(length(l_1), length(l))) SortSpec(l_1)) SortSpec(l)

Based on this we can derive a function sort-aux which depends on a proof that Lt is well-founded. Now using a proof that Lt is well founded we can finally derive the sort function

sortAux ∈ (r ∈ Tot(R); tr ∈ Trans(R); l ∈ List(A); Acc(Lt, length(l))) SortSpec(l)
sortAux(r, tr, l, acc(h_1)) ≡ sortRec(A, R, r, tr, l, [l_1, h]sortAux(r, tr, l_1, h_1(length(l_1), h)))
sort ∈ (A ∈ Set; R ∈ (A; A) Set; r ∈ Tot(R); tr ∈ Trans(R); l ∈ List(A)) SortSpec(l)
sort ≡ [A, R, r, tr, l]sortAux(r, tr, l, accLt(length(l)))

During a visit by Simon Thompson we also progified quick sort. Using the libraries of lemmas developed during the verification of insertion sort and merge sort this turned out to be just a matter of several hours.

4 Progifying an LL(1) parser

The aim is to verify a simple LL(1) parser for the following grammar of simple expressions

$$E :: i \mid uE \mid (E \circ E)$$

This example was presented by Chisholm [Chi87] and informally verified using Type Theory. Some parts have been formally verified in an early implementation of Type Theory. This example has also been formalised in the NuPRL based Oyster system [6] and in the LEGO system [7].

When formalising this example we discovered that an important point has been overlooked in Chisholm's informal verification which is only compensated by assuming a strong lemma in the formalised part. Our solution is to strengthen the invariant, which not only seems more natural but also opens the way to generalise the approach to a more general class of grammars.

As before we use an ML program as the prototype, which given a list of symbols returns a a parse tree (PT) or a negative answer:

```
datatype Sy = lpar | rpar | var | uop | bop;
datatype PT = v | u of PT | b of PT*PT;

fun parse1 (var::rest) = SOME (v,rest)
  | parse1 (uop::rest) =
```

[6] By Christian Horn, Fachhochschule Furtwangen, Germany
[7] By Thomas Schreiber, University of Edinburgh, Great Britain

```
      (case parse1 rest of
          (SOME (e,rest')) => SOME (u e,rest')
        | _ => NONE )
  | parse1 (lpar::rest) =
      (case parse1 rest of
          (SOME (e1,bop::rest1)) =>
              (case parse1 rest1 of
                   (SOME (e2,rpar::rest2))
                         => SOME (b(e1,e2),rest2)
                 | _  => NONE)
        | _ => NONE)
  | parse1 _ = NONE;

fun parser w =
      case parse1 w of
           (SOME (e,[])) => SOME(e)
         | _ => NONE;
```

Following Chisholm we define the structural recursive function unparse which can be derived from the definition of the grammar above:

unparse ∈ (PT) List(Sy)
  unparse(v) ≡ sgl(var)
  unparse($u(h_1)$) ≡ cons(uop, unparse(h_1))
  unparse($b(h_1, h_2)$) ≡ cons(lpar, append(unparse(h_1), cons(bop, append(unparse(h_2), sgl(rpar))))))

The dependent type ParseAll defines a solution to a parsing problem, e.g. finding an inverse to unparse:

ParseAll ∈ (w ∈ List(Sy)) Set
  parseAll ∈ (w ∈ List(Sy); p ∈ PT; Id(w, unparse(p))) ParseAll(w)

The parsing algorithm has to decide ParseAll, i.e. either to give a solution or a proof that no such solution exists - therefore it has the following type:

parser ∈ (w ∈ List(Sy)) Dec(ParseAll(w))

where Dec is defined as

Dec ∈ (Set) Set
  yes ∈ (P) Dec(P)
  no ∈ ((P) Empty) Dec(P)

Already from the structure of the parsing algorithm it should be obvious that we cannot derive parser directly but that we have to allow partial solutions, e.g. the parsing of an initial segment of the list. This is expressed by the inductive family Parse:

```
Parse ∈ (w ∈ List(Sy)) Set
  parse ∈ (w ∈ List(Sy); p ∈ PT; r ∈ List(Sy); Id(w, append(unparse(p), r))) Parse(w)
```

However, it is not sufficient to use Dec(Parse) as the main invariant of the algorithm. Consider a successful parse of the string "(i". We expect "o" as the next symbol, but how can we show that the string is unparsable otherwise? Indeed, it is just the fact that the grammar is LL(1) which implies that there is at most one parse. Instead of assuming this property (as Chisholm does) we specialise the invariant by introducing a new predicate which requires that in the positive case not only a solution is given but it is also shown that it is unique. This leads to the definition of ParseSpec

```
ParseSpec ∈ (w ∈ List(Sy)) Set
  noParse ∈ (w ∈ List(Sy); (Parse(w)) Empty) ParseSpec(w)
  uniqueParse ∈ (w ∈ List(Sy);
                 p ∈ Parse(w);
                 (p' ∈ Parse(w)) Id(parse_PT(p), parse_PT(p'))) ParseSpec(w)
```

Following the structure of the ML program we are able to implement parseRec — this requires a number of laborious lemmas to be proven:

```
parseRec ∈ (w ∈ List(Sy);
            p ∈ (w' ∈ List(Sy); Lt(length(w'), length(w))) ParseSpec(w')) ParseSpec(w)
```

Applying a variant of the typed fix point combinator as described above and using the proof that Lt is accessible for every natural number, we derive

```
parser1 ∈ (w ∈ List(Sy)) ParseSpec(w)
```

From here it is quite straightforward to derive 1 parser by specialising the result type.

This approach can be generalised to a general backtracking recursive descent parser for non left recursive grammars. In the general case we extend the invariant to a list of parse trees which cover all possible parses of the list. Such LL(1) arises just as the special case where this list is always either empty or a singleton.

5 Concluding remarks

We presented a completely formal way to derive correct functional programs and argued that this approach could be used in a program development system which can generate code comparable to the code generated by a compiler for a functional programming language.

One of the main differences of our presentation compared to similar studies is that we do need to use sophisticated tactics to generate an incomprehensible

proof object and then to magically extract a program, but that the construction of the program together with its correctness proof becomes a task which is not too different from normal programming. Indeed, we are able to view our proof objects as annotated programs. The ALF system we are using is essentially a sophisticated program editor which supports the production of type correct programs in a dependently typed language.

It should also be noted that we benefitted greatly from recent proposals to simplify the notation of Type Theory, like the introduction of pattern matching. We hope that this trend can be continued by copying other ideas from the functional programming community - a good example are records and subtyping.

We emphasised the importance of prototypes and the gradual verification — improvement — of programs. At the moment our approach is quite crude: we started with a functional program and rewrote it in type theory. In an integrated development system, this step should be smoother. A first proposal in this direction is the program tactic for the Coq system [Par94].

We *completely* formalised a number of simple programs, our observation is that although the effort is reasonable in some cases, it is still unreasonable for others of a small size. However, it seems that the effectiveness could still be vastly improved by improving the tools and extending the libraries.

Access to the code

All the formal developments mentioned or presented here are available by ftp. They can be either loaded into the ALF system which is also available on ftp or obtained in a postscript version for reading.

The alf code can be obtained via the following URL:

`file://ftp.cs.chalmers.se/pub/users/alti/alf-examples`

References

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. A user's guide to ALF. Draft, May 1994.

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13 – 28, 1993.

[BCMS89] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[Chi87] P. Chisholm. Derivation of a Parsing Algorithm in Martin-Löf's theory of types. *Science of Computer Programming*, 8:1–42, 1987.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, 1994.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[Nor88] Bengt Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[Par94] Catherine Parent. Developing certified programs in coq — the program tactic. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, number 806 in LNCS, pages 3 – 18, 1994.

[Pau86] Lawrence C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[PM89] Christine Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions*. PhD thesis, Universite Paris VII, 1989.