An Alf implementation of Insertion Sort
Patricia Peratto
peratto@incony.edu.uy
Programa de Desarrollo de Ciencias Básicas
PEDECIBA Informática
Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Montevideo, Uruguay
February 1995

Abstract
This paper describes an Alf [1] implementation of insertion sort. Alf is a system for editing proofs and theories. Martin-Löf's monomorphic set theory is a programming logic which can be represented in Alf [3]. Under the Curry-Howard isomorphism, types are identified with specifications and elements in types with proofs. Programs developed in Martin-Löf's set theory are totally correct, i.e. all well-typed programs terminate. Martin-Löf's set theory is a functional programming language with dependent types. Dependent types allow the definition of types not usually present in functional programming languages. These types are the cartesian product of a family of sets and the disjoint sum of a family of sets which correspond under the Curry-Howard analogy to the universal and existential quantifiers respectively. This strengthening in the types of the language, allows to express non-trivial specifications as is the one in this example. The development of a program satisfying a specification, corresponds with the construction of a well-typed object by application of the rules of the theory.

Keywords: Programming logics, Type Theory, Program development in type theory.
1 Introduction

This paper describes an Alf implementation of insertion sort. Alf is a system for editing proofs and theories. This example was developed on the first version of Alf which is based on a combination of a general type system (GTS) and Martin-Löf’s logical framework [1].

We have chosen insertion sort as an interesting example to be developed in type theory. An early version of this algorithm developed into the subset version of type theory was presented in [4]. Here we develop this algorithm in Martin-Löf’s monomorphic set theory for which the judgement forms are decidable; this allows the mechanization and so the development of certified programs.

The complete formal proofs are not given; this is out of question because of the size of the proofs. The compromise adopted is to omit all proofs of general facts and present the main body of the development of the program. In any case, all the needed properties for the complete formalization are enumerated to make it possible to reconstruct the proof. We also give some hints about the way they were developed. In the mechanization, the proofs were developed completely formally.

The goal of this paper is threefold:

- to experiment program development by using a logical framework. To this purpose we have chosen ALF [1]. This is important not only since proofs developed in a framework are verified but because the framework really help us to develop the proofs.

- to illustrate distinct ways of defining new sets, i.e. sets defined by recursion (propositional or type valued functions) and inductive definitions.

- to exemplify a methodology for program development in type theory: that of finding a proof following the structure of a functional program.

We do not claim that the proof developed here is the best nor the most efficient one. Undoubtedly there must be many other ways of representing intermediate results used to get the final one.

The rest of the paper is organized as follows:

In section 2 notational conventions used are presented.

Section 3 presents basic concepts concerning Martin-Löf’s type theories and Alf.

Section 4 describes insertion sort development in Martin-Löf’s monomorphic set theory; basic sets and defined constants are presented and the more important properties that follow from these definitions are enumerated.

Section 5 presents conclusions and further work.

2 Notational conventions

The proofs we present are edited versions of what was produced in Alf.

Some constants are defined with hidden parameters, which is indicated with the symbol ↓ in front of them. In this case, the parameter is not visible but the information is there. In this form, we can get a
polymorphic "vision" of the system by hiding type information. In some cases, we hide other parameters that are redundant and can be inferred from the context.

The notation used for expressions is as follows: if \( e \) is an expression and \( z \) a variable, then \([z]e\) is used for abstraction of \( z \) from \( e \). If \( e \) and \( t \) are expressions then \( e(t) \) is used for the application of \( e \) to \( t \).

To introduce definitions, the notation \( a := b \) is used.

We will denote the intentional propositional equality on the set \( N \) as \( IdN \) or simply \( Id \) without set argument. We will also denote \( Id(A, a, b) \) as \( a \equiv_A b \) omitting the set when it is clear from the context.

We will denote the constructors of the cartesian product (conjunction) and the disjoint union of a family of sets (existential quantifier) with braces (\(<, >\)). As selectors for the existential quantifier we use \( Fst \) and \( Snd \), while for the cartesian product we use \( fst \) and \( snd \).

### 3 Martin-Löf’s type theories and Alf

Martin-Löf's type theory, was originally developed with the aim of being a clarification of constructive mathematics but unlike most other formalizations of mathematics, type theory is not based on first order predicate logic. Instead predicate logic is interpreted within type theory through the correspondence between propositions and sets. A set can not only be viewed as a proposition, it is also possible to see a set as a problem description. This possibility is important for programming because if a set can be seen as a description of a problem it can in particular be used as a specification of a programming problem. Hence set membership and program correctness are the same problem in type theory and because all programs terminate, correctness means total correctness [3]. It is in this way that we can see type theory as a programming logic which has a programming language a specification language and rules relating them. The specification (type) language is like the language of first order predicate logic, we use it to express propositions (sets, descriptions of problems). The programming language is a typed functional programming language, the programs are typed programs (elements in sets, proofs of propositions) and the rules are typing rules (rules of this logic) expressed in a natural deduction style.

The theories originally developed were polymorphic, i.e. one program could have more than one type; an example of this is the polymorphic identity function \( \lambda[z].x \) which has both types \( N \rightarrow N \) and \( Bool \rightarrow Bool \). Afterwards, Martin-Löf develops a logical framework, i.e. a more general logic in which type theory could be expressed. This gives rise to a monomorphic type theory (to which we refer as Martin-Löf's monomorphic set theory) where the terms are monomorphic i.e. they have inside them type information (this makes the judgement forms decidable). For a complete explanation of Martin-Löf’s logical framework and the representation of the monomorphic theory of sets I refer to [3]. Here we will work on a representation of the monomorphic theory over Alf.

### 3.1 Overview of Alf

Alf is a system for editing proofs and theories. It is based on a combination of a general type system (GTS) and Martin-Löf’s logical framework. It is possible to add equations when defining a theory (it is
thus possible to define the operations of Martin-Löf's monomorphic set theory inside Alf).

Because of the possibility of introducing sets by induction, type theory is an open theory; it was presented in [2] syntactic schemas for defining correct extensions of the theory, i.e., new inductively defined sets together with their elimination rules. One can also define new types by recursion, by using type valued functions. This could be done by following an idea presented in [6].

3.1.1 Definitions in Alf

Inductive sets are introduced by giving axioms (primitive constants) expressing how to form the set and how to construct their elements. Examples of primitive constants are $N$, $\text{succ}$ and 0, which are defined in Alf as

\[
N : \text{Type} \\
0 : N \\
\text{succ} : (N)N
\]

Primitive constants have only a type, they don't have a definition. They get their meaning in other way (outside the theory). Such constants are also called constructors, since they compute to themselves.

To express computations one uses defined constants. A defined constant has a type and a definition

\[
c : A == a
\]

The definiendum $c$ computes in one step to its definiens $a$. A defined constant can either be explicitly or implicitly defined. An implicitly defined constant is a definition that may be recursive. They are expressed by using case expressions over the way the elements of the set where constructed. One usually use them to define elimination rules expressing a principle of structural induction over the elements in the set. The recursion operator over natural numbers is an example of an implicitly defined constant:

\[
\text{natrec} : (\downarrow C : (N)\text{Set}; d : C(\text{zero}); e : (x : N; y : C(x))C(s(x)); n : N)C(n) == \\
case n of \text{zero} & d | s(x) & e(x,y,\text{natrec}(C,d,e,x))
\]

By using Alf's hierarchy of types one can also define elimination rules that give a type as result. As an example, for the set of natural numbers presented before, one could define

\[
\text{Natrec} : (d : \text{Type}; e : (x : A; y : \text{Type})\text{Type}); n : N)\text{Type} == \\
case n of \text{zero} & d | s(x) & e(x,N\text{atrec}(d,e,x))
\]

This way of defining types by recursion is useful for proving properties that could not be proved in type theory without universes, as Peano4 [5]. It is also useful for defining new sets as will be seen afterwards.

An explicitly defined constant is just an abbreviation of its definiens (which has to be a welltyped expression). One gives a name for the constant and an expression constructed by using previously defined constants, variables, application and abstraction. It is not possible to give a recursive definition in this
case. We represent derived rules by using explicitly defined constants.

When we read the constant as the name of a rule, then a primitive constant is usually a formation or introduction rule, an implicitly defined constant is an elimination rule (with the contraction rule expressed as the step from the definiendum to the definiens) and an explicitly defined constant is a derived rule.

We will not present the definition of all the basic sets of Martin L"of's monomorphic set theory. I refer to [3] chapter 20 for this complete presentation.

4 Insertion Sort Development in Martin-L"of's Set Theory

I will develop the example for the case of lists of natural numbers. It can be generalized to any set \(A\) which has a decidable equality and a total order relation defined over their elements. I have chosen as specification for this problem the following

\[
\forall(l \in \text{List}(N)) \exists(y \in \text{List}(N)). \text{Perm}(y, l) \land \text{Ordered}(y)
\]

where \(\text{Perm}\) expresses that one list is a permutation of another and \(\text{Ordered}\) that a list is ordered.

An element in this type is a function which when applied to a list gives a pair whose first element is the expected list and whose second element is another pair consisting of the proof of correctness, i.e. that the new list is a permutation of the given one and that it is ordered.

4.1 The development of the proof

We have a formal specification and we know that a functional program for insertion sort is:

\[
\text{isort } \text{nil} = \text{nil}\\
\text{isort } \text{cons } a \ b = \text{insert } a \ (\text{isort } b)
\]

\[
\text{insert } a \ \text{nil} = \text{cons } a \ \text{nil}\\
\text{insert } a \ \text{cons } b \ c = \text{if } a \leq b \ \text{then } \text{cons } a \ \text{cons } b \ c \ \text{else } \text{cons } b \ \text{insert } a \ c
\]

In this case, the process of proof construction was guided by the structure of the program, and what we did was following this structure construct the proof that meets the specification.

4.2 Basic Sets and Relations

In the formalization of the proof we will need definitions for \(\text{Ordered}\) and \(\text{Perm}\) and also for other basic sets. We will first introduce the order for natural numbers which will be defined as an inductive binary relation. Then we will define other two sets, one for representing the \(\text{Ordered}\) relation and the other for representing the membership relation. These sets will be both defined by recursion and inductively. We
show in these cases that the definitions are isomorphics, i.e. we can define translation functions from one kind of definition to the other and show that they are mutually inverses.

We will also introduce a definition for \textit{Perm}. In this case, we introduce it in a definitional way, i.e. as an abbreviation for an expression constructed in terms of previously defined ones.

When defining these sets in Alf, we group the definitions in \textit{logics}, each one (generally) depending on previously defined ones. We will present the definitions respecting this order.

\textbf{Order}

We will define the order for natural numbers by defining an inductive family of sets \textit{less}

\[
\text{less} : (N; N) \text{Set}
\]

\[
\text{zeroless} : (n : N) \text{less}(0, s(n))
\]

\[
\text{succless} : (u, v : N; \text{less}(u, v)) \text{less}(s(u), s(v))
\]

The elimination rule provides an induction principle for the set defined. Once given the formation and introduction rules, elimination and equality rules can be deduced automatically, following [2].

\[
\text{lesselim} : (\perp C : (x, y : N; \text{less}(x, y)) \text{Set};
\]

\[
\quad \text{ezero} : (x : N)C(0, s(x), \text{zeroless}(x));
\]

\[
\quad \text{esucc} : (x, y : N; p; \text{less}(x, y); q : C(x, y, p))C(s(x), s(y), \text{succless}(x, y, p));
\]

\[
\quad x, y : N; p : \text{less}(x, y)C(x, y, p) =\]

\[
\text{case } p\text{ of } (\text{zeroless}(n) \rightarrow \text{ezero}(n) | 
\text{succless}(x, y, q) \rightarrow \text{esucc}(x, y, q, \text{lesselim}(C, \text{ezero}, \text{esucc}, x, y, q))
\]

Finally, we define $\leq$ by:

\[
\leq : (x, y : N) \text{Set} =\text{less}(x, y) \lor \text{IdN}(x, y)
\]

\textbf{nroof}.

Given a natural number and a list of natural numbers, \textit{nroof} computes the number of times that the given number occurs in the list. It is defined by recursion on \textit{l} as:

\[
\text{nroof} : (n : N; l : \text{List}(N)) \text{N} =\]

\[
\text{listrec}(0, [x, y, z] \text{when}(\text{decideNat}(n, x),
\text{[eq]}s(x)
\text{[ne}x], l)
\]

where \text{decideNat}(n, x) : \text{IdN}(n, x) \lor \neg \text{IdN}(n, x) is a function computing the decidability of the equality over \textit{N}.
Perm

A possible way of defining that a list is a permutation of another one, is by proposing that they have the same number of occurrences of each element. This gives rise to the following definition:

\[\text{Perm} : (l_1, l_2 : \text{List}(N)) \text{Set} == \forall n \in N. \text{IdN}(n \text{roof}(n, l_1), n \text{roof}(n, l_2))\]

Ordered

We will define sets representing the \textit{Ordered} relation. The first definition is by recursion on the list. It makes use of the propositional function \textit{Listrec} which expresses type valued recursion on lists (we omit its definition, it is very similar to \textit{Natrec} one). We define:

\[\text{Ordered} : (l : \text{List}(N)) \text{Set} == \text{Listrec}(T, [x, y, z] \text{Listrec}(T, [x', y', z'] \text{leq}(x, x') \land z, y), l)\]

Another possibility is to define it as an inductive family of sets as

\[\text{Ordered'} : (l : \text{List}(N)) \text{Set}\]

\[\text{Ordnil} : \text{Ordered'}(\text{nil})\]
\[\text{Ordcons1} : (x : N) \text{Ordered'}(\text{cons}(x, \text{nil}))\]
\[\text{Ordcons2} : (l, x, y : N; l : \text{List}(N); \text{leq}(x, y); \text{Ordered'}(\text{cons}(y, l))) \text{Ordered'}(\text{cons}(x, \text{cons}(y, l)))\]

where the elimination constant is defined by

\[\text{Ordelim} : (C : (l : \text{List}(N); \text{Ordered'}(l)) \text{Set};\]
\[e_{\text{nil}} : C(\text{nil}, \text{Ordnil});\]
\[e_{\text{cons1}} : (x : N) C(\text{cons}(x, \text{nil}); \text{Ordcons1}(x));\]
\[e_{\text{cons2}} : (x, y : N; l : \text{List}(n); \text{leq}(x, y);\]
\[p : \text{Ordered'}(\text{cons}(y, l));\]
\[C(\text{cons}(y, l), p)) C(\text{cons}(x, \text{cons}(y, l)), \text{Ordcons2}(x, y, l, \text{le}, p))\]
\[l : \text{List}(N); o : \text{Ordered'}(l)) C(l, o) ==\]
\[\text{case o of (Ordnil} \Rightarrow e_{\text{nil}} |\]
\[\text{Ordcons1}(x) \Rightarrow e_{\text{cons1}}(x) |\]
\[\text{Ordcons2}(x, y, l, \text{le}, p) \Rightarrow e_{\text{cons2}}(x, y, l, \text{le}, p, \text{Ordelim}(C, e_{\text{nil}}, e_{\text{cons1}}, e_{\text{cons2}}, p))))\]

There is an isomorphism between both \textit{Ordered} definitions. This can be seen by defining the following translation functions

\[\text{OrderedtoOrdered'} : (l : \text{List}(N)) \text{Ordered}(l) \Rightarrow \text{Ordered'}(l) ==\]
\[\text{Listrec}(\lambda([h] \text{Ordnil},\]
\[\text{[x, y, z]} \text{piapply}(\text{Listrec}(\lambda([x] \lambda([h] \text{Ordcons1}(x))),\]
\[\text{[x', y', z']} \lambda([x] \lambda([h] \text{Ordcons2}(y', \text{fst}(h), \text{apply}(\text{piapply}(x, x'), \text{snd}(h))))), y);\]
\[x),\]
\[l)\]
and

\[ \text{Ordered}'\text{toOrdered} : (l : \text{List}(N); \text{Ordered}'(l))\text{Ordered}(l) = \]
\[ [l, o] \text{Ord'}\text{elim}(tt, [x]tt, [x, y, l', le, p, q] < le, p >, l, o) \]

and proving that they are mutual inverses, i.e., given \( l : \text{List}(N); o : \text{Ordered}(l); o' : \text{Ordered}'(l) \)

\[ \text{Ordered}'\text{toOrdered}(l, \text{apply}(\text{OrderedtoOrdered}'(l), o)) = \text{Ordered}(l) \circ o \]

and

\[ \text{apply}(\text{OrderedtoOrdered}'(l), \text{OrderedtoOrdered}'(l, o')) = \text{Ordered}'(l) \circ o' \]

which are proved by list-elimination and Ord'-elimination respectively, and following the same structure that in the previous definitions.

**Member**

A possible definition is as a propositional function defined by cases on the list. Instead of writing the expression in terms of \textit{Listrec} we will use the following more readable notation:

\[
\begin{align*}
\text{member}(a, \text{nil}) &= \bot \\
\text{member}(a, \text{cons}(u, v)) &= \text{Id}(a, u) \lor \text{member}(a, v)
\end{align*}
\]

another possibility is to define it inductively as

\[
\begin{align*}
\text{member} : (a : N; l : \text{List}(N))\text{Set} \\
\text{memcons1} : (a : N; v : \text{List}(N))\text{member}(a, \text{cons}(a, v)) \\
\text{memcons2} : (a1, a2 : N; v : \text{List}(N); \text{member}(a1, v))\text{member}(a1, \text{cons}(a2, v))
\end{align*}
\]

Also in this case we have defined an isomorphism between both definitions. The translation functions and the proof that they are isomorphic follow the same lines as before.

There is an important consequence of the existence of the previously mentioned isomorphisms, that of the independence of the proved properties from the definitions. This can be expressed in the following way:

If \( O \) and \( O' \) are isomorphic and we have defined a function \( f \) from \( O \) to other set \( B \), then we get the corresponding function from \( O' \) to \( B \) by composing the translation function with \( f \), and the same holds in the inverse direction. In other terms, we get that the following diagram commutes:
We will continue the development of the proof considering the recursive definitions of *Ordered* and *Member*. The reason for choosing these definitions is a practical one, i.e. that we have just many lemmas developed for these definitions.

### 4.3 Operations and Properties

In this section, we will first define the operations that are used in the proof. Then we will enumerate which mathematical properties involving these operations, the order and equality over natural numbers have been proved. We will also present derived properties holding between these operations and the given definitions for *Perm*, *Member* and *Ordered*.

**min**

It is defined by using the decidability of the order relation over $N$, as

$$\text{min}(a, b) = \text{when}(\lnot \epsilon a, \epsilon b, \text{order}(a, b))$$

**Least**

We will define it for a pair $x, y$ where $x \in N$ and $y \in \text{List}(N)$. This allows to avoid the case of the empty list for which it is not defined.

$$\text{Least} : (x : N; y : \text{List}(N))N$$

is could be defined by defining a higher order function by list recursion and applying it to $x$. The function is the following

$$\text{Least} : (x : N; y : \text{List}(N))N \equiv \text{apply}(\text{listrec}(\lambda([x_1]u, v, w)\lambda([x_1]\text{min}(x_1, \text{apply}(w, u))), y), x)$$

Another (more easy) way to define it is as

$$\text{Least} : (x : N; y : \text{List}(N))N \equiv \text{listrec}(x, [u, v, w]\text{min}(u, w))$$

Both definitions are equivalent for our purposes, i.e. the same properties have been proved for them.
Properties of \( \text{Id} \)

We have used the following properties of equality

1. \( \text{Idrefl} : (A : \text{Set}; a : A)\text{Id}(A, a, a) \)
2. \( \text{Idsymm} : (A : \text{Set}; a, b : A)\text{Id}(A, a, b) \)
3. \( \text{Idtrans} : (A : \text{Set}; a, b, c : A; \text{Id}(A, a, b); \text{Id}(A, b, c))\text{Id}(A, a, c) \)
4. \( \text{Idsusbet} : (A : \text{Set}; P : (A)\text{Set}; a, b; A; \text{Id}(A, a, b); P(a))P(b) \)
5. \( \text{Idcongr} : (A, B : \text{Set}; f : (A)B; a, b : A; \text{Id}(A, a, b))\text{Id}(B, f(a), f(b)) \)

\( \text{Idrefl} \) more than a property is the introduction rule for the \( \text{Id} \) set, i.e. it is an axiom and is introduced as a primitive constant. The other properties are derived ones and were proved by using the elimination rule associated to the \( \text{Id} \) set.

Transitivities holding between \( \text{Id} \), less and \( \text{leq} \)

1. \( \text{leqtrans} : (\downarrow x, y, z : N; x \leq y; y \leq z)z \leq z \)
2. \( \text{leqsucc} : (x : N)x \leq s(x) \)
3. \( \text{leqlesstoleq} : (\downarrow x, y, z : N; x \leq y; y < z)x \leq z \)
4. \( \text{leqlesstoleqless} : (\downarrow x, y, z : N; x \leq y; y < z)x < z \)
5. \( \text{Idtoleq} : (x, y, z : N; \text{Id}(x, y))z \leq y \)
6. \( \text{leqIdtoleq} : (x, y, z : N; \leq (z, y); \text{Id}(y, z))z \leq z \)
7. \( \text{Idleqtoleq} : (x, y, z : N; \text{Id}(x, y); \leq (y, z))x \leq z \)

The are many other properties of \( < \) and transitivities which are useful in the proof of the above ones, we present here only the more directly involuted in the part of the proof developed in the paper.

Properties of addition

The addition of two elements of the set \( N \) is defined by the expression:

\[ x + y \equiv \text{natrec}(y, [u, v]\text{succ}(v), x) \]

from which we can prove the following propositions:

1. \( \text{leqdifff} : (\downarrow x, y : N; x \leq y)N \)
2. \( \text{leqdifffproof} : (\downarrow x, y : N; q : z \leq y)\text{leqdifff}(q) + x =_{N} y \)
3. \( \text{leqaddR} : (x, y : N)x \leq y + z \)
4. \( \text{addleqtoleq} : (\downarrow x, y, k : N; q : x + y \leq k) : y \leq k \)

The same comment as before is valid here.
Properties of min

From the given definition we have derived the following properties:

1. \( a \leq b \Rightarrow \text{IdN}(a, \min(a, b)) \)
2. \( b < a \Rightarrow \text{IdN}(b, \min(a, b)) \)
3. \( a < b \Rightarrow \text{IdN}(a, \min(a, b)) \)
4. \( b \leq a \Rightarrow \text{IdN}(b, \min(a, b)) \)
5. \( a \leq b \land a \leq c \Rightarrow a \leq \min(b, c) \)
6. \( a \leq \min(b, c) \Rightarrow a \leq b \land a \leq c \)
7. \( \text{IdN}(a, \min(b, c)) \Rightarrow a \leq b \land a \leq c \)
8. \( \text{IdN}(\min(a, b), \min(b, a)) \)
9. \( \text{IdN}(\min(a, \min(b, c)), \min(b, \min(a, c))) \)
10. \( \text{IdN}(a, \min(b, c)) \Rightarrow \text{IdN}(a, b) \lor \text{IdN}(a, c) \)
11. \( (a \leq c) \Rightarrow \min(a, b) \leq c \)
12. \( (b \leq c) \Rightarrow \min(a, b) \leq c \)
13. \( \min(a, \min(b, c)) = \min(\min(a, b), c) \)

They were proved using the decidability of the order relation over \( N \), and the properties of \( \text{Id} \), \( \text{<} \), and \( \leq \).

Properties of Least

The following properties follow for both Least definitions

1. \( \text{Leastswap}(x, x_1, y_1) = \text{IdN}(\text{Least}(x, \text{cons}(x_1, y_1)), \text{Least}(x_1, \text{cons}(x, y_1))) \)
2. \( \text{Leastmminswap}(n, u, x, y) : \text{IdN}(\text{Least}(n, \text{cons}(u, \text{cons}(x, y))), \min(x, \min(n, \text{Least}(u, y))) \)

The first is proved by list induction and applying properties of min. The second one follows from the first, properties proved for \( \text{min} \), and \( \text{Id} \)-transitivities.
Properties of nrooff

From nrooff definition, we get the following properties

1. \( \text{nrooffprop1} : (x, y : N; l : \text{List}(N); \text{IdN}(x, y)) \text{IdN}(\text{nrooff}(x, \text{cons}(y, l)), s(\text{nrooff}(x, l))) \)

2. \( \text{nrooffprop2} : (x, y : N; l : \text{List}(N); \text{not}(\text{IdN}(x, y))) \text{IdN}(\text{nrooff}(x, \text{cons}(y, l)), \text{nrooff}(x, l)) \)

3. \( \text{nrooffprop3} : (x, y : N; l : \text{List}(N); \text{IdN}(\text{nrooff}(x, \text{cons}(y, l)), 0)) \text{IdN}(\text{nrooff}(x, l)), 0) \)

the three are easily proved by doing a case analysis over the decidability of the equality between \( x \) and \( y \).

Properties of Member

1. \( \text{memswap} : \\
\quad (x, x_1, z : N; y : \text{List}(N)) \text{member}(x, \text{cons}(x, \text{cons}(x_1, y))) \Rightarrow \text{member}(x, \text{cons}(x_1, \text{cons}(x, y))) \)

follows from member definition.

Properties relating Member and nrooff

1. \( \text{membertonrooff} : (x : N; y : \text{List}(N)) \text{member}(x, y) \Rightarrow \text{not}(\text{IdN}(\text{nrooff}(x, y), \text{zero})) \)

2. \( \text{nroofftombert} : (x : N; y : \text{List}(N)) \text{not}(\text{IdN}(\text{nrooff}(x, y), \text{zero})) \Rightarrow \text{member}(x, y) \)

They were proved by list-recursion. In the inductive step, we used the decidability of equality between \( x \) and the first element of the list, and the properties of nrooff. In the first proof we also apply peano4.

Properties relating Member and Least

1. \( \text{leqtoIdLeast} : (a, x : N; y : \text{List}(N)) a \leq \text{Least}(x, y) \Rightarrow \text{member}(a, \text{cons}(x, y)) \Rightarrow \text{IdN}(a, \text{Least}(x, y)) \)

2. \( \text{Idleasttonmember} : (a, x : N; y : \text{List}(N)) \text{IdN}(a, \text{Least}(x, y)) \Rightarrow \text{member}(a, \text{cons}(x, y)) \)

3. \( \text{leqmemberstoleqLeast} : (a, x, v : N; y : \text{List}(N)) \\
\quad (\forall v \in N. \text{member}(v, \text{cons}(x, y)) \Rightarrow (a \leq v)) \Rightarrow a \leq \text{Least}(x, y) \)

4. \( \text{IdLeaststoleqmembers} : (x : N; y : \text{List}(N)) \\
\forall (m \in N. \text{IdN}(m, \text{Least}(x, y)) \Rightarrow \forall n \in N. \text{member}(n, \text{cons}(x, y)) \Rightarrow m \leq n \)

1 is proved by list recursion. In the inductive step we do a case analysis over the proof of member(a, cons(x, ...). 2 and 3 follow by recursion and applying some previous lemmas. 4 is the more "large" one. We apply list recursion. In the inductive step, we assume the corresponding elements in the domains of the functions and apply a case analysis over memswap(p), where p is the assumption corresponding to member(n, cons(x, ...)). In both cases we apply inductive hypothesis with the corresponding arguments.
Properties of Perm

The following properties follow from the given definition of Perm and the properties proved for nroof.

1. reflexivity \( \text{Permrefl} : (l : \text{List}(N)) \text{Perm}(l, l) \)
2. symmetry \( \text{Permsym} : (l_1, l_2 : \text{List}(N); \text{Perm}(l_1, l_2)) \text{Perm}(l_2, l_1) \)
3. transitivity \( \text{Permttrans} : (l_1, l_2, l_3 : \text{List}(N); \text{Perm}(l_1, l_2); \text{Perm}(l_2, l_3)) \text{Perm}(l_1, l_3) \)
4. unvariability under cons
   \( \text{Permcons} : (n : N; l_1, l_2 : \text{List}(N); \text{Perm}(l_1, l_2)) \text{Perm}(\text{cons}(n, l_1), \text{cons}(n, l_2)) \)
5. unvariability under tail
   \( \text{Permmfromcons} : (n : N; l_1, l_2 : \text{List}(N); \text{Perm}(\text{cons}(n, l_1), \text{cons}(n, l_2))) \text{Perm}(l_1, l_2) \)
6. unvariability under interchange of elements
   \( \text{Permswap} : (x_1, x_2 : N; y : \text{List}(N)) \text{Perm}(\text{cons}(x_1, \text{cons}(x_2, y)), \text{cons}(x_2, \text{cons}(x_1, y))) \)

Properties relating Perm and Member

1. \( \text{PermtomemberR} : (z : N; l_1, l_2 : \text{List}(N); p : \text{Perm}(l_1, l_2); h : \text{member}(z, l_1)) \text{member}(z, l_2) \)
2. \( \text{PermtomemberL} : (z : N; l_1, l_2 : \text{List}(N); p : \text{Perm}(l_1, l_2); h : \text{member}(z, l_2)) \text{member}(z, l_1) \)

   Follow from the properties relating member and nroof.

Properties relating Perm and Least

1. \( \text{PermtIdLeast} : (x_1, x_2 : N; y_1, y_2 : \text{List}(N); p : \text{Perm}(\text{cons}(x_1, y_1), \text{cons}(x_2, y_2))) \text{IdN}(\text{Least}(x_1, y_1), \text{Least}(x_2, y_2)) \)

   Follow from the properties relating member and Least.

Properties of Ordered

1. \( \text{Ordfromcons} : (\downarrow u : N; \downarrow v : \text{List}(N); h : \text{Ordered}(\text{cons}(u, v))) \text{Ordered}(v) \)
2. \( \text{Ordprop} : (x_1, x_2 : N, y : \text{List}(N)) \text{Ordered}(\text{cons}(x_1, \text{cons}(x_2, y))) \Rightarrow \text{Ordered}(\text{cons}(x_1, y)) \)

   Can be proved by list recursion.

Properties relating Perm, Ordered and Least

1. \( \text{IdLeastProof} : (\downarrow x, n, u : N; \downarrow v, y : \text{List}(N); h : \text{Ordered}(\text{cons}(u, v)); gt : u < n; p : \text{Perm}(\text{cons}(x, y), \text{cons}(n, v))) \text{IdN}(u, \text{Least}(u, \text{cons}(x, y))) \)

   We first prove \( \text{Ordered}(\text{cons}(u, v)) \Rightarrow \text{IdN}(u, \text{Least}(n, \text{cons}(u, v))) \) by list-recursion. Then we apply this result to \( h \). Finally we apply transitivity of Id to this result, \( \text{Leastswap}(n, u, v) \), and \( \text{Idcongr} \) with \( \text{min} \) applied to \( \text{PermtIdLeast}(n, x, y, \text{Permsym}(a)) \).
4.4 The construction of the proof

To find an element in

\[ \forall (l \in \text{List}(N)) \exists (y \in \text{List}(N)). \text{Perm}(y, l) \land \text{Ordered}(y) \]

the first we do is to assume that we have \( l \in \text{List}(N) \). Now, we have to prove:

\[ \exists (y \in \text{List}(N)). \text{Perm}(y, l) \land \text{Ordered}(y) \]

We know as functional programmers that the structure of the program, i.e. the list we get applying the program fits the conditions, but our intention is prove that, i.e., to get a proof from which we can get the expected list by selection. So, we will see the program as defining the first component of this type and our work now is to find expressions for the other components (the correctness proof).

As well as in the program we will apply induction on \( l \). We have to distinguish two cases:

\( l = \text{nil} \)

in this case the solution is a triple \( \text{nil}, p, q \), where \( p \) is a proof of \( \text{Perm}(\text{nil}, \text{nil}) \) and \( q \) is a proof of \( \text{Ordered}(\text{nil}) \). From the given definitions, it follows that \( \lambda ([z] \text{refl}(0)) : \text{Perm}(\text{nil}, \text{nil}) \) and \( \text{tt} : \text{Ordered}(\text{nil}) \) so this case is solved.

\( l = \text{cons}(a, b) \)

We have to find \( y \in \text{List}(N) \), such that \( \text{Perm}(y, \text{cons}(a, b)) \) and \( \text{Ordered}(y) \). We know from the functional program that the expected form of \( y \) is given by

\[ \text{isort cons a b = insert a (isort b)} \]

Following the same idea, we will define the proof corresponding to the function \( \text{insert} \), and then express the solution for \( l = \text{cons}(a, b) \) in terms of this proof.

**insert definition**

Assume

\[
\begin{align*}
  a & : N \\
  b & : \text{List}(N) \\
  z & : \exists y \in \text{List}(N). \text{Perm}(y, b) \land \text{Ordered}(y)
\end{align*}
\]

Given an Ordered list, \( \text{insert} \) will put inside it the given element, in such a way that the list remains \( \text{Ordered} \). This is specified as follows:

\[ \text{insert} : (a : N; l : \text{List}(N)) \text{Ordered}(l) \Rightarrow \exists y \in \text{List}(N). \text{Perm}(y, \text{cons}(a, l)) \land \text{Ordered}(y) \]

Following the structure of the program for \( \text{insert} \), we will define the proof term by recursion on \( l \).
\[ l = \text{nil} \]

\[ \text{insert}(a, \text{nil}) = \lambda h. < \text{cons}(a, \text{nil}), < \lambda z. \text{id}(\text{nroof}(z, \text{cons}(a, \text{nil}))), h > > \]

which corresponds to the first line of \text{insert} program, and where the witness of correctness is given by:

1. \[ \lambda z. \text{id}(\text{nroof}(z, \text{cons}(a, \text{nil}))) : \text{Perm}(\text{cons}(a, \text{nil}), \text{cons}(a, \text{nil})) \]

2. \[ h \in \text{Ordered}(\text{cons}(a, \text{nil})) \]

\[ l = \text{cons}(b, c) \] We have to define an element in

\[ \text{Ordered}(\text{cons}(b, c)) \Rightarrow \exists y_1 \in \text{List}(N). \text{Perm}(y_1, \text{cons}(a, \text{cons}(b, c))) \land \text{Ordered}(y_1) \]

and since definitions by recursion and proofs by induction are identified in type theory, we will assume the solution of the inductive step, i.e.:

\[ z : \text{Ordered}(c) \Rightarrow \exists y_1 \in \text{List}(N). \text{Perm}(y_1, \text{cons}(a, c)) \land \text{Ordered}(y_1) \]

First assume \( h : \text{Ordered}(\text{cons}(b, c)) \). Applying \text{Ordfromcons} to \( h \), we get a proof of \( \text{Ordered}(c) \). Applying \( z \) to this proof and the projections associated to the \( \exists \) and \( \land \) sets we get:

1. \( \text{steplist} : \text{List}(N) \)

2. \( \text{Permproof} : \text{Perm}(\text{steplist}, \text{cons}(a, c)) \)

3. \( \text{Ordproof} : \text{Ordered}(\text{steplist}) \)

Now what remains to prove is

\[ \exists y_1 \in \text{List}(N). \text{Perm}(y_1, \text{cons}(a, \text{cons}(b, c))) \land \text{Ordered}(y_1) \]

As well as in the program we will compare \( a \) and \( b \). This is done in the form of a case analysis over \( \text{order}(a, b) \). We have to distinguish the following two cases:

1. \( \text{le} : a \leq b \)

2. \( \text{gt} : b < a \)

We will define proof terms for each case named as \text{insconsle} and \text{insconsgt} respectively.

- \text{insconsle} definition is

\[ \text{insconsle}(x, h, \text{le}) = < \text{cons}(a, \text{cons}(b, c)), \text{Permrefl}(\text{cons}(a, \text{cons}(b, c))), < \text{le}, h > > \]
where the proof of being ordered is constructed by following case c) of Ordered definition i.e. from the proof le of $a \leq b$ and the proof $h$ of Ordered($cons(b,c)$).

- insconsgt definition

This case is more complicated, and involves the proof of additional properties.

insconsgt will be a triple, whose first element is the ordered list, the second a proof that it is a permutation of $cons(a,cons(b,c))$ and the third a proof that it is ordered.

We know that $b < a$ and we have a list steplist which is a permutation of $cons(a,c)$ and that is ordered. Then, $b$ will be the first element and steplist the rest of the resulting list since it is the result of inserting $a$ at the corresponding place inside $c$. \(^1\)

We will call our list Ordpermlist and it will have the form

$\text{Ordpermlist} = cons(b,\text{steplist}) \ (I)$

Now, it remains to prove that this list is ordered and that it is a permutation of $cons(a,cons(b,c))$. I will call these proofs OrdProofInsGt and PermProofInsGt respectively.

- PermproofInsGt : Perm(Ordpermlist,cons(a,cons(b,c)))

we know

$Perm\text{swap}(a,b,c) : Perm(cons(b,cons(a,c)),cons(a,cons(b,c)))$

and

$Perm\text{cons}(b,\text{steplist},cons(a,c),\text{Permproof}) : Perm(cons(b,cons(a,c)),cons(b,\text{steplist}))$

so by transitivity and symmetry of Perm we get the expected result.

- OrdproofInsGt : Ordered(Ordpermlist)

To prove it, we will prove a lemma called OrdProofInsGtLemma which given $l : List(N)$, $gt : u < n$ and $h : Ordered(cons(u,v))$ gives a proof of

$Perm(l,cons(n,v)) \Rightarrow Ordered(l) \Rightarrow Ordered(cons(u,l))$

**Lemma 1 (OrdProofInsGtLemma)** We will prove it by cases on $l$.

1. $l=\text{nil}$ in this case, $\lambda q.t$ is a proof
2. $l=\text{cons}(x,y)$ we will assume $p : Perm(cons(x,y),cons(n,v))$ and $q : Ordered(cons(x,y))$ and prove $Ordered(cons(u,cons(x,y)))$

\(^1\)compare this which the second line of the program for insert, you will see that we are doing exactly the same, i.e., the recursive call in the program and the inductive step in the proof occur at the same place.
\[
\textit{IdLeasttoLeqmembers}(u, \text{cons}(x, y)):
\forall m \in N.\text{Id}(m, \text{Least}(u, \text{cons}(x, y))) \Rightarrow \forall n \in N.\text{member}(n, \text{cons}(u, \text{cons}(x, y))) \Rightarrow m \leq n
\]

so, applying this proof to \(u : N\) and \(\text{IdLeastProof}(h, gt, p) : \text{IdN}(u, \text{Least}(u, \text{cons}(x, y)))\) we get an element in \(\forall n \in N.\text{member}(n, \text{cons}(u, \text{cons}(x, y))) \Rightarrow u \leq n\)

applying it to \(x : N\) and \(\text{inr}(\text{inl}((\text{refl}(x)))) : \text{member}(x, \text{cons}(u, \text{cons}(x, y)))\) we get a proof of \(u \leq x\);

from this and \(q : \text{Ordered}(\text{cons}(x, y))\) we get a proof of \(\text{Ordered}(\text{cons}(u, \text{cons}(x, y)))\). By function introduction (\(\lambda\) abstraction) to \(q\) and \(p\), we get the proof of the inductive step.

Now, from the proof term of \(\text{OrdProofInsGt}\) Lemma, applying it to \(\text{Permproof}\) and \(\text{Ordproof}\), we get a proof term for \(\text{OrdproofInsGt}\).

This ends \textit{insert} definition. Now we are ready to finish the definition of the proof term for \textit{isort}

\[
\text{isort} : (l : \text{List}(N))\exists(y \in \text{List}(N)).\text{Perm}(y, l) \wedge \text{Ordered}(y)
\]

We had just defined the proof term for \(l = \text{nil}\). We will finish the proof for \(l = \text{cons}(a, b)\).

We had assumed

\[
z : \exists y \in \text{List}(N).\text{Perm}(y, b) \wedge \text{Ordered}(y)
\]

let \(i = \text{apply}(\text{insert}(a, \text{Fst}(z)), \text{snd}(\text{Snd}(z)))) : \exists(y' \in \text{List}(N).\text{Perm}(y', \text{cons}(a, \text{Fst}(z))) \wedge \text{Ordered}.y')\)

then it follows that

\[
\begin{align*}
\text{Fst}(i) : & \text{List}(N) \\
\text{fst}(\text{Snd}(i)) : & \text{Perm}(\text{Fst}(i), \text{cons}(a, \text{Fst}(z))) \\
\text{Permcons}(a, \text{fst}(\text{Snd}(z))) : & \text{Perm}(\text{cons}(a, \text{Fst}(z)), \text{cons}(a, b))
\end{align*}
\]

and

\[
\text{snd}(\text{Snd}(i)) : \text{Ordered}(\text{Fst}(i))
\]

so, the following term gives a solution for \(\text{isort}(\text{cons}(a, b))\)

\[
< \text{Fst}(i), < \text{Permtrans}(\text{fst}(\text{Snd}(i)), \text{Permcons}(a, \text{fst}(\text{Snd}(z)))), \text{snd}(\text{Snd}(i)) >>
\]

finally, from this proof we can extract the ordered list and the proof of correctness.
sortprogram : (l : List(N))List(N) = Fst(isort(l))

sortproof : (l : List(N))Perm(sortprogram(l), l) ∧ Ordered(sortprogram(l)) = Snd(isort(l))

This ends the proof.

5 Conclusions and further work

From the proof developed, it seems that the idea of following the structure of a functional program to find a proof-program in type theory is sensible. It seems interesting to see the connection between this idea and the method presented in [7], where it is explained given an ML program, how to generate automatically a proof (in the system Coq) from which it could be extracted. It seems also interesting to try to develop this kind of method for Alf.

I also find of interest to study more deeply the relation between families of sets defined by recursion and families of sets defined by induction.

References


