Logic and dependent types in the Aldor Computer Algebra System

Simon Thompson
Computing Laboratory,
University of Kent at Canterbury, UK
`S.J.Thompson@ukc.ac.uk`

April 2000, revised June 2000, August 2000

Abstract. We show how the Aldor type system can represent propositions of first-order logic, by means of the 'propositions as types' correspondence. The representation relies on type casts (using `pretend`) but can be viewed as a prototype implementation of a modified type system with *type evaluation* reported elsewhere [9]. The logic is used to provide an axiomatisation of a number of familiar Aldor categories as well as a type of vectors.

1 Introduction

An earlier paper [9] outlined a new combination of symbolic computation and formal reasoning by means of an embedding of a logic in the type system of the computer algebra system Axiom [6] or, more specifically, its library compiler Aldor [12]. Such an embedding has two important consequences.

It can make the system *sound*, so that, for instance, $x^{n+1}/(n+1)$ is only used as the integral of x^n when n is not equal to -1. It can also make a CAS *more expressive* in combining logical steps with calculational ones. We illustrate this in Section 5 by giving axiomatisations of various algebraic structures within the CAS.

The literature contains a number of different strategies proposed for combining computer algebra and theorem proving; see, for instance, [2, 3, 1]. Our approach is distinctive in requiring minimal changes to an existing system; indeed, using the strategy outlined in this paper it can be done to an *unchanged* system.

The embedding is based on the *propositions as types* correspondence, whereby a logical proposition in a constructive logic is represented by a type in a programming language [7, 10]. Proofs of the proposition are then elements of the type in question. A more detailed exposition follows in the body of the paper, but as an illustration observe that a conjunction of two formulas is represented by the product of two types, each type representing one of the conjuncts; elements of this product will be pairs of elements which correspond to proofs of the two conjuncts.

To represent quantified formulas, it is necessary to use *dependent types*. A dependent function `f` is one for which the *type* of the result `f(a)` depends on the *value* of the argument `a`. A dependent function type represents a universal formula and a dependent record represents an existential statement. The dependent types of Aldor are thus crucial to the embedding.

To realise the proposed embedding, it is necessary to modify the way that the system handles dependent types; the earlier paper explains this in detail, but suffice it to say here that we require that *types are evaluated* in the same way that values are. To take an example which will be developed in the present paper, in the current Aldor system the vector types `Vector(2+3)` and `Vector(5)` are *different*, since the vector lengths `2+3` and `5` are left unevaluated; after evaluation they are, of course, the same.

This paper discusses how a *prototype* of the modifications can be implemented in Aldor using type casts using `pretend`. An expression `(e pretend T)` is accepted by the typechecker as having type `T`; typechecking and code-generation then proceed on this basis.

Our implementation is useful for a variety of reasons. First, it provides a *proof of concept* for our ideas. Secondly, the implementation *guides the design* of a modified type checker by illustrating the particular places that type evaluation and normalisation need to take place. Finally, it provides a *testbed* in which to explore the different ways that a logical view of mathematical objects can be integrated into an already existing CAS.

This work also shows that the code generated by the Aldor system for our technically type-incorrect programs is correct, and thus shows that the back end of the Aldor system can support code generation for our modified notion of type dependency without itself being changed.

The structure of the paper is as follows. Section 2 contains a brief introduction to some of the salient features of Aldor; others are introduced in the remainder of the paper. Sections 3 and 4 describe how propositional and predicate logic are represented in Aldor. Building on these, Section 5 shows how this framework can be used to axiomatise various algebraic structures in Aldor. Section 6 shows how a dependent type of vectors can be developed in Aldor and we make some concluding remarks in Section 7.

I am grateful to John Shackell, Leonid Timochouk and James Beaumont as well as Erik Poll and Therèse Hardin and her group for discussions about this work. Stephen Watt patiently explained to me some of the finer details of Aldor. Martin Dunstan and two anonymous referees gave very helpful feedback on an earlier version of the paper. NAG Ltd. have kindly given us access to the Aldor compiler for research in this area.

2 An Introduction to Aldor

The core of Aldor [12] (also known in the past as AXIOM-XL and $A^\sharp$ [11]) is a strongly-typed functional language which has much in common with modern functional languages like SML and Haskell [5].

Since Aldor is designed with mathematics in mind, its type system is more complex than those of most programming languages. It supports *overloading* of symbols and *coercion* between types as well as permitting functions to have *dependent* types. Moreover the language allows an entity like the collection of integers to be seen in various different ways, depending on its context. For example, the integers might be seen as a set of values, a group, an integral domain, a subset of the real numbers and so forth. To do this, the language allows types and functions to be collected into abstract data types which are known as *domains* in Aldor.

The type of a domain, which is described by a signature, is called a *category*. Categories bear a strong resemblance to Java interfaces, and thus the effect of allowing a domain to belong to different categories is to support different views of the same structure. An example of a domain and its category is given in Section 3, which also contains a brief overview of the Aldor mechanisms supporting these definitions. Categories can be built on top of other categories, giving a version of inheritance between domains. Categories can also be parameterised by values including domains, since types are themselves values (of type `Type`). Because of this Aldor possesses a rich structure of interdependent program units.

Current descriptions of Aldor, [12, 11], give informal definitions of the type system; [8] gives a formal description of the essence of the Aldor type system. Examples of Aldor programs are given in remainder of this paper as well as in [9, 12]; features of the language are introduced as they are needed. The paper uses Aldor version 1.1.12p5 with the `axllib` library.

3 Propositional logic

This section gives an overview of how propositional logic is implemented in Aldor. The logical connectives become constructors of types, and the logical rules to introduce and eliminate connectives similarly become functions over logical formulas.

Implication

The simplest connective to represent is implication, `=>`, which is represented by the function type constructor, `->`.

An implication is introduced by a function definition, and eliminated by a function application. For example, a proof that a formula `A` implies itself is given by the function

```
(x:A):A +-> x
```

The notation `(a:A):C +-> e` denotes a function: in this case it is the function which takes a value `a` of type `A` to the result `e` of type `C`.

```
And(A:Type,B:Type) : with{
        andIntro : (a:A,b:B) -> %;
        andElim1 : (p:%)     -> A;
        andElim2 : (p:%)     -> B; }
    == add {
    Rep      == Record(fst:A,snd:B);
    import from Record(fst:A,snd:B);

    andIntro(a:A,b:B):% == per [a,b];
    andElim1(p:%):A     == (rep p).fst;
    andElim2(p:%):B     == (rep p).snd; }
```

Figure 1. The Aldor domain for conjunction: `And`.

Given proofs `a` of `A` and `f` of `A->B`, a proof of `B` is given by `f(a)`. Other examples will emerge in the course of discussion of the other connectives.

Conjunction

As was said in the introduction, conjunction can be represented by a product type; in Aldor this means that a `Record` is used. A direct representation would be given by defining

```
And(A:Type,B:Type):Type == Record(fst:A,snd:B);
```

A function to introduce a conjunction would have type

```
andIntro(a:A,b:B):And(A,B)
```

since this takes proofs of the two conjuncts to build a proof of the conjunction. From a proof of a conjunction we may recover proofs of the two conjuncts, so that there should be two function to *eliminate* a conjunction:

```
andElim1(p:And(A,B)):A
andElim2(p:And(A,B)):B
```

Instead of this direct approach, we define an Aldor *domain* to represent conjunction; this approach is more in keeping with the idioms of Aldor; the full definition appears in Figure 1.

The signature of the domain is given in the `with{...}`, where the symbol `%` stands for 'the type being defined', that is `And(A,B)`. This part of the definition gives the category of `And(A,B)`; the part following the keyword `add` gives the implementation of the domain itself. The representation of the domain is given by `Rep`; note that it is necessary explicitly to import from the representation, `Record(fst:A,snd:B)`, to allow the overloaded record operations to be used at this particular type.

Following this are the definitions of the functions themselves. In `andIntro` a record is built, and in the elimination functions record fields are selected. The functions `rep` and `per` are used to convert from the type being defined (`%`) to the representation chosen (`Rep`) and *vice versa*.

Given these functions one can begin to write proofs using the introduction and elimination rules. A proof of `B&A` from `A&B` is given by the function

```
flip(A:Type,B:Type,p:And(A,B)):And(B,A)
    == andIntro(andElim2(p),andElim1(p));
```

which takes the two components of the proof `p` of `And(A,B)` and 'flips' their order to give a proof of the 'flipped' formula, `And(B,A)`.

Disjunction

A disjunction is represented by a `Union` of two types, and so in defining the domain `Or(A:Type,B:Type)` the representation is given by

```
Rep == Union(inl:A,inr:B);
```

To introduce a proof of a disjunction it is sufficient to give a proof of either disjunct, and so there are two introduction rules, given by functions of type

```
orIntro1(a:A):%
orIntro2(b:B):%
```

(recall that `%` stands for `Or(A,B)` in this case.)

How is a disjunction eliminated? To prove an arbitrary formula `C` from `A\/B` it is enough to have proofs of `C` from `A` and `B` separately:

```
orElim(C:Type,f:A->C,g:B->C,p:%):C
    == {
    val == (rep p);                                   (1)
    if (val case inl)                                 (2)
        then f(val.inl)                               (3)
        else g(val.inr)};
```

Converting the proof `p` of type `Or(A,B)` to its representation `val` (line (1)), a case switch is performed on `val` (line (2)). If `val` comes from the left half of the union (line (3)) then a proof of `C` is produced by applying `f` – a proof of `A->C` – to the proof of `A` contained in `val`; otherwise, `g` is used.

An example combining implication, conjunction and disjunction is a proof of the formula `(A=>C)&(B=>C)` from `((A\/B)=>C)`:

```
andOr(A:Type,B:Type,C:Type,p:Or(A,B)->C):And(A->C,B->C)
    ==
    { import from Or(A,B);  -- Needed to give the appropriate
                            -- meaning to orIntro1 and orIntro2.
    andIntro((a:A):C +-> p(orIntro1(a)),
             (b:B):C +-> p(orIntro2(b)));};
```

Absurdity and negation

In logic we can represent the proposition `¬A` by `A => Abort`. Here `Abort` is an 'absurd' proposition, *i.e.* a proposition with no conceivable proofs, or in other words an empty type.

In Aldor we can represent `Abort` by `Union()`, a union with no components, and then negation is given by `Not(A)`, which in turn is represented by `(A->Abort)`. Given a proof of absurdity, we can however prove everything; this we represent by

```
exfalso(a:Abort,B:Type):B == error "impossible value";
```

and from this we may derive a rule of contradiction

```
contraRule(B:Type,p:Not(A),q:A):B
```

which allows us to deal directly with `Not(A)`. Example proofs which involve negation, including `((A \/ ¬A)&¬¬A) => A`, can be found at

```
http://www.cs.ukc.ac.uk/people/staff/sjt/Atypical/AldorExs
```

from which all the code discussed in this paper can be downloaded.

In our discussion of absurdity we have used the `error` function to implement the law *Ex Falso Quodlibet*: from a contradiction anything can be proved. Provided that this is the *only* use of `error` in a proof then the logic given here is consistent; obviously a wider use of `error` could render the logic inconsistent.

4 Predicate logic

This section explores the representation of predicate logic within Aldor using dependent types and the `pretend` mechanism.

How is a predicate over a type `A` to be represented? We use a *propositional function* of type `A->Type`, so that if `F` is such a predicate then `F(a)` represents the proposition that `F` holds for the value `a`.

Existential quantification

An existential formula `(∃x:A)F(x)`, 'there exists an `x` of type `A` for which `F(x)` holds', is given by a *record* of dependent type, and so the domain

```
Exists(A:Type,F:A->Type)
```

has the representation

```
Record(fst:A,snd:F(fst))
```

A member of this type will be a record `[a,b]` where `a` is a member of `A` and `b` is a member of `F(a)`, that is a proof of `F(a)`. This is a *constructive* interpretation of existence, since the `a` in question is an explicit *witness* to the validity of the existential statement. It is therefore easy to see that an existential proposition is introduced by a function

```
existsIntro(a:A,b:F(a)):%
```

(where recall that `%` means 'the type being defined', that is `Exists(A,F)`).

How can we use (or 'eliminate') a proof of an existential formula? We can extract either of the component parts. It is straightforward to extract the first component, which has type `A`:

```
existsElim1(p:%):A
    == (rep p).fst;
```

but extracting the second component is somewhat more involved. Suppose that `[a,b]` has type `Record(fst:A,snd:F(fst))` then `b` will have the type `F(a)`. If `p` names the whole record, then the type of its second component will necessarily involve its first, and hence the appearance of the function `existsElim1` in the type of `existsElim2`:

```
existsElim2(p:%): F(existsElim1(A,F,p))
    == (rep p).snd pretend F(existsElim1(A,F,p));
```

This definition uses the `pretend` facility by which `(e pretend T)` has type `T` irrespective of the type for `e` deduced by the system. This type casting mechanism is used here to compensate for the fact that Aldor does not have type evaluation. In this example it does not recognise the identity of `F(fst)` – the type deduced for `(rep p).snd` by Aldor – and

```
existsAnd3(A:Type,P:A->Type,Q:A->Type,
           r:Exists(A,And1(A,P,Q))):Exists(A,P)
    == {
    val ==> existsElim1(r);             -- The witness.
    import from And(P(val),Q(val));     -- To ensure that
                                        -- andElim1 can be used.
    existsIntro(val,
                andElim1(existsElim2(r)
                           pretend And(P(val),Q(val)) ) )};
```

Figure 2. A proof of `(∃x:A)(P(x)&Q(x)) => (∃x:A)P(x)`

`F(existsElim1(A,F,p))`, despite the definition of `existsElim1`. This is the first case where *type evaluation* plays an essential role in the definitions that we make.

Another example, a proof of `(∃x:A)(P(x)&Q(x)) => (∃x:A)P(x)`, is given in Figure 2. In this proof `pretend` is used to identify the fact that substitution commutes with a 'lifted' form of conjunction,

```
And1(A:Type,P:A->Type,Q:A->Type):(A->Type)
    == (a:A):Type+->And(P(a),Q(a));
```

informally giving the equivalence: $\texttt{(P} \ \&_1 \ \texttt{Q)(x)} \equiv \texttt{(P(x) \& Q(x))}$.

Universal quantification

A universal formula `(∀x:A)F(x)`, 'for all `x` of type `A`, `F(x)` holds', is given by a *function* of dependent type, `(x:A)->F(x)`. If `f` has this type then for each `a` in `A`, `f(a)` is of type `F(a)`, that is for each `a` in `A`, `f(a)` is a proof of the proposition `F(a)`. We therefore define a domain

```
All(A:Type,F:A->Type)
```

with the representation

```
(x:A)->F(x)
```

In many examples which use universal quantification it again becomes necessary to use `pretend` for type evaluation. In particular it is often the case that we have to identify a function application like `((x:A):B +-> e)(a)` with its result, namely `e[a/x]` (that is the expression `e` in which the argument `a` has been substituted for the parameter `x`).

Reasoning with identity

A fundamental part of first-order reasoning is the logic of identity, under which equals can be substituted for equals: Leibnitz's principle. In order to implement this fully requires some support from the implementation, but we have implemented a limited version of equality reasoning using `pretend` and Aldor equality. The Aldor category `BasicType` contains a Boolean equality operation, and so all types supporting this interface have elements which can be compared for equality. We define the `I`-types which represent the proposition that two expressions are equal:

```
I(A:BasicType,a:A,b:A):Type
    == if (a=b) then Integer else Abort;
```

so that if `a` and `b` are equal any integer is a proof of this fact whereas there are no proofs that unequal elements are equal. There is no special reason for choosing to use `Integer` here: any non-empty type would do.

For reasoning with `I`-types we have an introduction rule that any element equals itself:

```
refl(A:BasicType,a:A):I(A,a,a)
   == trivial pretend I(A,a,a);
```

(where `trivial` is simply `0`). Leibnitz's principle is embodied in the substitution rule:

```
subst(A:BasicType,a:A,b:A,eq:I(A,a,b),F:A->Type,p:F(a)):F(b)
   == p pretend F(b);
```

which states that if `F(a)` is valid (as witnessed by `p`) and if `a` and `b` are equal (as witnessed by `eq`), then `F(b)` is valid. The proof of `F(b)` is again `p`, but this time coerced into type `F(b)` by `pretend`.

As we remarked earlier, unrestricted use of `pretend` will result in an inconsistent system, but if it is encapsulated in a rule like `subst` *it can only be used when there is evidence that use of it is sound*; this is the exact role of the equality witness `eq`.

Note that this treatment is not simply syntactic sugar for the Boolean operation; we are able to perform *hypothetical* reasoning using this implementation, which is not possible using the Boolean-valued operation.

For example, we can give a general proof that identity is symmetric

```
symm(A:BasicType,a:A,b:A):(I(A,a,b) -> I(A,b,a))
```

If we are supplied with a proof of `I(A,a,b)` we can use that proof in an application of `subst` to replace by `b` the (boxed occurrence of) `a` in

```
I(A,[a],a)
```

Again, this proof will use `pretend` to mimic type evaluation; full proofs of symmetry and transitivity are at the Web site mentioned earlier.

This completes our discussion of an embedding of a constructive, many-sorted, logic in the type theory of Aldor, modulo use of the `pretend` operation. In the presence of type evaluation we will be able to eschew use of `pretend`.

5 Categories and axioms

We have now developed enough logical machinery to introduce an axiomatisation of some algebraic notions. In the standard Aldor library we can find categories (that is interfaces) which are intended to capture the notions of monoid and group:

```
Monoid: Category == BasicType with {
        *         : (%,%) -> %;
        1         : %; };

Group : Category == Monoid with {
inv       : % -> %; };
```

As we have argued elsewhere [9] these categories fail to capture what it means to be a group or a monoid since they lack any axiomatisation. We can correct that by adding axioms to the categories thus:

```
MonoidAx (M:Monoid): Category == with {
        import from M;
        leftUnit  : (m:M) -> I(M,m,1*m);
        rightUnit : (m:M) -> I(M,m,m*1);
        assoc     : (m:M,n:M,p:M) -> I(M,m*(n*p),(m*n)*p); };

GroupAx (G:Group): Category == MonoidAx(G) with {
        import from G;
        leftInv  : (g:G) -> I(G,1,g*inv(g));
        rightInv : (g:G) -> I(G,1,inv(g)*g); }
```

Note that we use the logic developed earlier to express universally quantified formulas by means of dependent functions and identity using `I`-types.

We are also able to use *inheritance* in writing the axiomatisation: in writing the axioms for a group `G`, `GroupAx(G)`, we extend the monoid axiomatisation for the same structure, `Monoid(G)`. In the latter expression we use the fact that `Group` inherits from `Monoid`, so that a group may be passed as a parameter at any point where a monoid is expected.

Using the techniques introduced here it is possible to build a hierarchy of *axiomatic* categories which shadows the *signature* hierarchy already implemented in Aldor. The axiomatic hierarchy allows more distinctions to be made: a commutative monoid has the same signature as a monoid but is different axiomatically.

```
CommutativeMonoidAx (M:Monoid): Category == MonoidAx(M) with {
        import from M;
        comm      : (m:M,n:M) -> I(M,m*n,n*m); };
```

In the remainder of the section we introduce the booleans as a monoid and show how they conform to the axioms.

The logic of the `Boolean` type

The `Boolean` type in Aldor consists of the values `true` and `false`. We prove that a property `F` – that is a propositional function `F:Boolean->Type` – is valid *for all* Booleans by showing that it holds for both values, that is, that we have proofs for `F(true)` and `F(false)`. This proof rule is embodied in the function

```
boolAll(F:Boolean->Type,t:F(true),f:F(false),b:Boolean):F(b)
    == if b then (t pretend F(b))
            else (f pretend F(b));
```

Note that `pretend` is used here to give `if...then...else...` a dependent type. In a system with type evaluation it would be given this type, which is a generalisation of its current type.

The `Boolean` monoid

The Booleans form an additive monoid thus:

```
(a:Boolean) * (b:Boolean) : Boolean
    == if a then (not b) else b;
1:Boolean
    == false;
```

and an illustrative proof of an axiom is given by

```
leftUnit(b:Boolean):I(Boolean,b,1*b)
 == boolAll(G,
```

```
        trivial pretend G(true),
        trivial pretend G(false),
        b) pretend I(Boolean,b,1*b)
    where {
    G:Boolean->Type == (x:Boolean):Type+->I(Boolean,x,1*x); };
```

It cannot be stressed too strongly that the three instances of `pretend` are only necessary in this proof because the current Aldor system lacks type evaluation. In such a system the undecorated term will in itself be a proof.

Other monoids and groups

It is possible in a similar way to show that the `Integers` form a monoid; in this case the proofs of the universal statements will be by *induction* (or equivalently, recursion). General results – such as the fact that the direct product of two monoids forms a monoid – allow other structures to be shown to be monoids. Similar considerations also apply to groups and other algebraic structures.

An alternative approach to axiomatisation

The categories `MonoidAx` and `GroupAx` are parametric; for a given structure `S`, the axioms are expressed by the categories `MonoidAx(S)` and `GroupAx(S)`. An alternative approach is given by

```
MonoidAndAx: Category == Monoid with {
        import from %;
        leftUnit  : (m:%) -> I(%,m,1*m);
        rightUnit : (m:%) -> I(%,m,m*1);
        assoc     : (m:%,n:%,p:%) -> I(%,m*(n*p),(m*n)*p); };

GroupAndAx: Category == Join(MonoidAndAx,Group) with {
        import from %;
        leftInv  : (g:%) -> I(%,1,g*inv(g));
        rightInv : (g:%) -> I(%,1,inv(g)*g); };
```

These categories `extend` the categories `Monoid` and `Group`; the statement `import from %` ensures that the operations of the base category are visible in the extension.

Under this approach, the algebraic operations are grouped with the proofs that they satisfy the appropriate axioms. Using the `extend` operation it is possible to extend a domain with proof objects constructed with reference to the underlying representation; this is not possible under the

```
-- The definition of Vectors of length n.
Vector(n:Integer): Type
 == if n<=1 then Integer else Record(fst:Integer,rst:Vector(n-1));

-- Constructing a Vector(n) from an Integer and a Vector(n-1).
vCons(n:Integer,x:Integer,v:Vector(n-1)):Vector(n)
 == { import from Record(fst:Integer,rst:Vector(n-1));
      [x,v]@Record(fst:Integer,rst:Vector(n-1)) pretend Vector(n);
};

-- A zero vector of length n.
zVec(n:Integer):Vector(n)
 == if n<=1 then 0 pretend Vector(n) else vCons(n,0,zVec(n-1));

-- Append two Vectors.
append(n:Integer,m:Integer,v:Vector(n),w:Vector(m)):Vector(n+m)
 == { if n<=0 then w pretend Vector(n+m)
      else if n=1 then vCons(m+1,
                             (v pretend Integer),
                             (w pretend Vector((m+1)-1)))
                             pretend Vector(n+m)
      else vCons(n+m,
                 vec.fst,
                 append(n-1,m,vec.rst,w) pretend
Vector((n+m)-1))
      where { vec == (v pretend
                       Record(fst:Integer,rst:Vector(n-1))); }};
```

Figure 3. The `Vector` type and functions

first approach. It remains to be seen which style of axiomatisation is more flexible in practice.

6 Programming with dependent types: vectors

In this section we revisit the example of vectors mentioned earlier and in [9]. As in the previous sections, we use the `pretend` operation to sidestep the typechecker, but we stress that the code written here executes perfectly well in the unmodified system, which is based on a first-order, weakly typed, abstract machine.

We have implemented a variety of operations over vectors; this section gives an overview of the code presented in Figure 3. The type of `Integer` vectors, `Vector`, is defined by recursion over the natural numbers: a vector of length `1` is simply an Integer, whilst a general vector of length `n` is a record `[x,v]` consisting of a (first) value `x` and a vector `v` of length `n-1`. The function `vCons` is the constructor for this type, and it is illustrated in the construction of the zero vector of length `n`, `zVec(n)`.

The most interesting function from the type-checking point of view is `append`, which joins two vectors of length `n` and `m` to give a vector of length `n+m`. The three slanted instances of *`pretend`* illustrate non-trivial coercions which identify `n+0` and `n`; `(m+1)-1` and `m`; and `(n+m)-1` and `(n-1)+m` respectively. It is arguable how easy it is to automate these. From the point of view of a user of this library, however, when we write a concrete application like `addVec(5,append(2,3,vec2,vec3))` it is possible to recognise the type of `append(2,3,vec2,vec3)` as `Vector(5)` and thus to allow the expression to be type checked successfully.

7 Conclusions

We have shown that using the '`pretend`' type casting mechanism of Aldor we are able to implement a prototype of a version of Aldor with type evaluation. This was illustrated by a number of examples.

As this is an exercise in prototyping, we have paid no attention to the consistency of the logical system; [9] discusses this issue in greater detail.

The development of proofs given here shows one mechanism for integration of symbolic computation and reasoning, namely that the proofs are written in the Aldor language. An alternative would be provided by interfacing the system to a theorem prover such as Coq [4], which is based on a logic similar to that reported here.

References

[1] Andrej Bauer, Edmund Clarke, and Xudong Zhao. Analytica - an experiment in combining theorem proving and symbolic computation. In *AISMC-3*, volume 1138 of *LNCS*. Springer, 1996.

[2] Bruno Buchberger. Symbolic Computation: Computer Algebra and Logic. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*. Kluwer, 1996.

[3] Jaques Calmet and Karsten Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In *FroCos'96*. Kluwer, 1996.

[4] C. Cornes et al. The Coq proof assistant reference manual, version 5.10. Rapport technique RT-0177, INRIA, 1995.

[5] John Hughes and Simon Peyton Jones, editors. *Report on the Programming Language Haskell 98*. `http://www.haskell.org/report/`, 1999.

[6] Richard D. Jenks and Robert S. Sutor. *Axiom: The Scientific Computation System*. Springer, 1992.

[7] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984. Based on a set of notes taken by Giovanni Sambin of a series of lectures given in Padova, June 1980.

[8] Erik Poll and Simon Thompson. The Type System of Aldor. Technical Report 11-99, Computing Laboratory, University of Kent at Canterbury, 1999.

[9] Erik Poll and Simon Thompson. Integrating Computer Algebra and Reasoning through the Type System of Aldor In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems: FroCoS 2000, LNCS 1794*. Springer-Verlag, Heidelberg, 2000.

[10] Simon Thompson. *Type Theory and Functional Programming*. Addison Wesley, 1991.

[11] Stephen M. Watt et al. A First Report on the $A^{\#}$ Compiler. In *ISSAC 94*. ACM Press, 1994.

[12] Stephen M. Watt et al. *AXIOM: Library Compiler User Guide*. NAG Ltd., 1995.