Integrating Computer Algebra and Reasoning through the Type System of Aldor

Erik Poll[1] and Simon Thompson[2]

[1] Computing Science Department
University of Nijmegen, The Netherlands
erikpoll@cs.kun.nl
[2] Computing Laboratory,
University of Kent at Canterbury, UK
S.J.Thompson@ukc.ac.uk

Abstract. A number of combinations of reasoning and computer algebra systems have been proposed; in this paper we describe another, namely a way to incorporate a logic in the computer algebra system Axiom. We examine the type system of Aldor – the Axiom Library Compiler – and show that with some modifications we can use the dependent types of the system to model a logic, under the Curry-Howard isomorphism. We give a number of example applications of the logic we construct and explain a prototype implementation of a modified type-checking system written in Haskell.

1 Introduction

Symbolic mathematical – or computer algebra – systems, such as Axiom [13], Maple and Mathematica, are in everyday use by scientists, engineers and indeed mathematicians, because they provide a user with techniques of, say, integration which far exceed those of the person themselves, and make routine many calculations which would have been impossible some years ago. These systems are, moreover, taught as standard tools within many university undergraduate programmes and are used in support of both academic and commercial research.

There are, however, drawbacks to the widespread use of automated support of complex mathematical tasks, which has been widely noted: Fateman [10] gives the graphic example of systems which will assume that $a \neq 0$ on the basis that $a = 0$ has not been established. This can have potentially disastrous consequences for the naive user of the system or indeed, if it occurs within a sufficiently complicated context, *any* user.

Symbolic mathematics systems are also limited by their reliance on algebraic techniques. As Martin [14] remarks, in performing operations of analysis it might be a precondition that a function be continuous; such a property cannot be guaranteed by a computer algebra system alone.

All this makes the combination of computer algebra with theorem proving a topic of considerable interest. Reasoning capabilities can allow a user to track assumptions, and thus to ensure that symbolic computations are *sound*, in contrast to the current situation in many CA systems.

Reasoning can also *extend* the capability of a CA system. A scenario might involve working with a particular monoid: if during the course of computation it can be shown, for instance, that the monoid is commutative then it is possible to use different, more efficient, simplification algorithms for expressions. The addition of reasoning here has made computation more efficient; in other situations - such as Martin's analysis example mentioned earlier – reasoning can allow computations to proceed where in general this would not be possible.

The literature contains a number of different strategies proposed for combining computer algebra and theorem proving; see, for instance, [4, 6, 3]. This paper describes another approach: we use the type system of the Axiom computer algebra system [13] to represent a logic, and thus to use the constructions of Axiom to handle the logic and represent proofs and propositions, in the same way as is done in theorem provers based on type theory such as Nuprl [7] or Coq [8].

This paper particularly explores the recent Axiom Library Compiler, Aldor [30], which is unusual among computer algebra systems in being strongly typed, and moreover in having a very powerful type system, including dependent types which are central to our work.

The implementation of dependent types in Aldor is somewhat nonstandard: there is no evaluation within type expressions, so that, for example, 'vectors of length `2+3`' are distinct from 'vectors of length `5`'; we show how this limits the expressivity of the dependent types. We describe a modification of the Aldor system which allow the types to represent the propositions of a constructive logic, under the Curry-Howard correspondence. We argue that this integrates a logic into the Aldor system, and thus permits a variety of logical extensions to Aldor, including adding pre- and post-conditions to function specifications, axiomatisations to categories of mathematical objects as well as the ability to reason about the objects in Aldor.

The structure of the paper is as follows. Section 2 introduces Aldor and in particular examines its system of types. In Section 3 we examine the issue of type equality in Aldor since it is central to our approach to embedding a logic in Aldor. The section also contains a number of strategies for modifying the Aldor compiler. We show how a logic can be defined in a modified variant of the Aldor system in Section 4 and Section 5 gives some example applications. We conclude with a discussion of related and future work.

2 An Introduction to Aldor

The Axiom Library Compiler, Aldor [30] (known in the past as AXIOM-XL and $A^\sharp$), provides the user with a powerful, general-purpose programming language in which to model the structures of mathematics. Aldor is compiled, in contrast to most computer algebra languages, and so it can provide much more efficient implementations of algorithms than interpreted languages.

The core of Aldor is a functional programming language which provides higher-order functions, generators (which bear a strong relationship to list comprehensions) and other features of modern functional languages like Standard ML [17] and Haskell [21]. It is also strongly typed, in common with these languages and indeed the majority of modern programming languages. Under this type discipline any type error – such as adding a character to a boolean operator – can be caught at compile time rather than at run time. This has two consequent advantages. First, a whole class of programming errors can be detected prior to program execution, thus increasing the dependability of the compiled code. Secondly, it means that it is possible to produce more efficient compiled code since no run-time type tags on program data need to be maintained to support type checking at run-time.

Since Aldor is designed with mathematics in mind, its type system is more complex than those of most programming languages. Mathematicians take a flexible approach to terminology, with the consequence that often the meaning of a symbol or phrase is only determined by its context. This requires of a programming language that symbols can be *overloaded*, and that sometimes values need to be *coerced* from one type to another: from the integers to floating-point numbers, for example.

More importantly this flexibility necessitates an entity like the collection of integers to be seen in various different ways, depending on the context. In the case of the integers this might be 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 *domains*, and the type of a domain, which is described by a signature, is called a *category*.

Categories can be built on top of other categories, giving a version of inheritance between domains. Categories can also be parametrised by values including domains; rather than implement a theory of parametric categories, Aldor takes types to be values just like more traditional values like 23 and the Boolean value 'false'. This has far-reaching consequences for the language.

Current descriptions of Aldor, [30, 29], give informal definitions of the type system. We have given a formal description of the essence of the Aldor type system in [22]. In the remainder of this section we summarise our approach in that paper and the conclusions that are drawn there.

2.1 An Overview of the Type System of Aldor

Unusually among languages for computer algebra, but in keeping with the functional school, Aldor is strongly typed. Each declaration of a binding can be accompanied by a declaration of the type of the value bound, as in the definition

```
a : Integer == 23;
```

The type of an expression can be declared explicitly to resolve any uses of overloaded identifiers. This cannot simply be done by the typing rules, since arbitrary overloading is allowed, so that, for instance, a single identifier `fun` may be overloaded to have types `Int -> Int`, `Int -> Bool` and `Bool -> Int` so that neither the type of the argument nor the type of result expected can disambiguate an application of `fun`.

Some 'courtesy' coercions are provided by the system automatically: these convert between multiple values (*à la* LISP), cross products and tuples. It is also possible to make explicit conversions – by means of the `coerce` function – from integers to floating point numbers and so forth.

As mentioned earlier, Aldor treats types as values. In particular, a type such as `Integer` has itself a type. The type of types is called `Type`. Having this type of all types means that the system supports functions over types, such as the identity function over (the type of) types:

```
idType (ty : Type) : Type == ty;
```

and explicit polymorphism, as in the polymorphic identity function which takes two arguments. The first is a type `ty` and the second is a value of that type which is returned as the result.

```
id (ty : Type, x : ty) : ty == x;                                (id)
```

Aldor permits functions to have dependent types, in which the type of a function result depends upon the value of a parameter. An example is the function which sums the values of vectors of integers. This has the type

```
vectorSum : (n:Integer) -> Vector(n) -> Integer
```

in which the result of a function application, say

```
vectorSum(34)
```

has the type `Vector(34) -> Integer` because its argument has the value `34`. In a similar way, when the `id` function of definition `(id)` is applied, its result type is determined by the type which is passed as its first argument. We discuss this aspect of the language in more detail in Section 2.3.

The system is not fully functional, containing as it does variables which denote storage locations. The presence of updatable variables inside expressions can cause side-effects which make the elucidation of types considerably more difficult. There is a separate question about the role of 'mathematical' variables in equations and the like, and the role that they play in the type system of Aldor.

Categories and domains provide a form of data abstraction and are addressed in more detail in Section 2.5.

The Aldor type system can thus be seen to be highly complex and we shall indeed see that other features such as macros (see Section 2.5) complicate the picture further.

2.2 Formalising the Type System of Aldor

This section outlines the approach we have taken in formalising the type system of Aldor. Our work is described in full in [22]; for reasons of space we can only give a summary here.

The typing relation is formally described by typing judgements of the form

$$\Gamma \vdash t : T$$

which is read 't has the type T in the context Γ'. A context here consists of a list of variable declarations, type definitions and so on. Contexts represent the collection of bindings which are in scope at a point in a program text. Note that t might have more than one type in a given context because of overloading of identifiers in Aldor, and so it would be perfectly legitimate for a well-formed context Γ to imply that $t : T$ and $t : T'$ where T and T' are different types.

Complex typing judgements are derived using deduction rules that codify conditions for a typing judgement to hold. For example,

$$\frac{\Gamma \vdash f : S\text{->}T \qquad \Gamma \vdash s : S}{\Gamma \vdash f(s) : T} \text{ (function elim)}$$

describes the type-correct application of a function. This deductive approach is standard; we have adapted it to handle particular features of Aldor such as overloading, first-class types and categories.

Our discussion in [22] examines the essential features of the full type system of Aldor; in this paper we concentrate on those aspects of the language relevant to our project. These are dependent function and product types; equality between types; and categories and domains, and we look at these in turn now.

2.3 Dependent Types

As we have already seen with the examples of `id` and `vectorSum`, the Aldor language contains dependent types. To recap, the function `vectorSum` defines a sum function for vectors of arbitrary length and has the type

```
vectorSum : (n:Integer) -> Vector(n) -> Integer
```

Similarly one can define a function `append` to join two vectors together

```
append : (n:Integer,m:Integer,Vector(n),Vector(m)) -> Vector(n+m)
```

The typing rule for dependent function elimination modifies the rule (function elim) so that the values of the arguments are *substituted* in the result type, thus

$$\frac{\Gamma \vdash f : (x : S)\text{->}T \qquad \Gamma \vdash s : S}{\Gamma \vdash f(s) : T[x := s]} \text{ (dependent function elim)}$$

Given vectors of length two and three, `vec2` and `vec3`, we can join them thus

```
append(2,3,vec2,vec3) : Vector(2+3)
```

where `2` and `3` have been substituted for `n` and `m` respectively.

We would expect to be able to find the sum of this vector by applying `vectorSum 5`, thus

```
(vectorSum 5) append(2,3,vec2,vec3)
```

but this will fail to type check, since the argument is of type `Vector(2+3)`, which is not equal to the expected type, namely `Vector(5)`. This is because no evaluation takes place in type expressions in Aldor (nor indeed in the earlier version of Axiom). We examine this question in the next section, and in Section 3 we discuss how the Aldor type mechanism can be modified to accommodate a more liberal evaluation strategy within the type checker. Similar remarks apply to dependent product types in which the type of a field can depend on the value of another field.

2.4 Equality of Types in Aldor

When are two types in Aldor equal? The definition of type equality in any programming language is non-trivial, but in the presence of dependent types and types as values it becomes a subtle matter.

Type equality is fundamental to type checking, as can be seen in the rule (function elim): the effect of the rule in a type-checker is to say that the application $f(s)$ is only legitimate if f has type S`->`T, s has type S', *and the types S and S' are equal.* Non-identical type expressions can denote identical types for a number of reasons.

– A name can be given to a type, as in

  ```
  myInt : Type == Int;
  ```

  and in many situations `myInt` and `Int` will be treated as identical types. [This is often called δ-equality.]
– The bound variables in a type should be irrelevant and Aldor treats them as so. This means that the types

  ```
  vectorSum : (n:Integer) -> Vector(n) -> Integer
  vectorSum : (int:Integer) -> Vector(int) -> Integer
  ```

  should be seen as identical. [α-equality]
– Types are values like any other in Aldor, and so can be evaluated. In particular a function over types like `idType` will be used in expressions such as `idType Int`. It would be expected that this would evaluate to `Int` and thus be seen as equivalent. [β-equality]
– In the presence of dependent types, expressions of any type whatsoever can be subexpressions of type expressions, as in `Vector(2+3)`. Equality between these subexpressions can be lifted to types, making `Vector(2+3)` equal to `Vector(5)`. [Value-equality]

Our report on the type system examines the practice of equality in the Aldor system and shows it to be complex. The Aldor system implements α-equality in nearly all situations, but δ-equality is not implemented in a uniform way. Over types neither β-equality nor value-equality is implemented, so that type equality in Aldor is a strong relation, in that it imposes finer distinctions than notions like β- or value-equality.

A rationale for the current definition in Aldor is that it is a simple notion of type equality which is strong enough to implement a weak form of type dependency in which arguments to types are themselves (literal) types which are not used in a computational way. This form of dependency is useful in the module system of Aldor where it can be used to formulate mathematical notions like 'the ring of polynomials in one variable over a field F' where the field F is a parameter of the type.

Our approach to integrating reasoning into Aldor requires a weaker notion of type equality, which we explore in Section 3.

2.5 Categories and Domains

Aldor is designed to be a system in which to represent and manipulate mathematical objects of various kinds, and support for this is given by the Aldor type system. One can specify what it is to be a monoid, say, by defining the `Category`[1] called `Monoid`, thus

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

This states that for a structure over a type '`%`' to be a monoid it has to supply two bindings; in other words a `Category` describes a signature. The first name in the signature is '`*`' and is a binary operation over the type '`%`'; the second is an element of '`%`'.

In fact we have stated slightly more than this, as `Monoid` *extends* the category `BasicType` which requires that the underlying type carries an equality operation.

```
BasicType : Category == with {
    = : (%,%) -> Boolean; }
```

We should observe that this `Monoid` category does not impose any constraints on bindings to '`*`' and '`1`': we shall revisit this example in Section 5.2 below.

Implementations of a category are abstract data types which are known in Aldor as domains, and are defined as was the value `a` at the start of Section 2.1, e.g.

```
IntegerAdditiveMonoid : Monoid == add {
    Rep == Integer;
    (x:%) * (y:%) : % == per((rep x) + (rep y));
    1 : %             == per 0; }
```

The category of the object being defined – `Monoid` – is the type of the domain which we are defining, `IntegerAdditiveMonoid`. The definition identifies a representation type, `Rep`, and also uses the conversion functions `rep` and `per` which have the types

[1] There is little relation between Aldor's notion of category and the notion from category theory!

```
rep : % -> Rep                    per : Rep -> %
```

The constructs `Rep`, `rep` and `per` are implemented using the macro mechanism of Aldor, and so are eliminated before type checking. In our report [22] we show how definitions of domains can be type checked without macro expansion, which allows, for instance, more accurate error diagnosis.

Categories can also be parametric, and depend upon value or type parameters; an example is the ring of polynomials over a given field mentioned earlier.

2.6 Conclusion

This section has given a brief overview of Aldor and its type system. It has shown that the notion of type equality in Aldor is a strong one, which makes distinctions between types which could naturally be considered equivalent. This is especially relevant when looking at the effect of type equality on the system of dependent types. In the sections to come we show how a logic can be incorporated into Aldor by modifying the notion of type equality in Aldor.

3 Modifying Type Equality in Aldor

Section 2.4 describes type equality in Aldor and argues that it is a strong notion which distinguishes between type terms which can naturally be identified. In this section we examine various ways of modifying type equality including the way we have chosen to do this in our prototype implementation.

3.1 Using the Existing System

It is possible to use the existing Aldor system to mimic a different – weaker – type equality by explicitly casting values to new types, using the `pretend` function of Aldor.[2] This effectively sidesteps the type checker by asserting the type of an expression which is accepted by the type checker without verification.

For instance, the vector example of Section 2.3 can be made to type check in Aldor by annotating it thus

```
(vectorSum 5) (append(2,3,vec2,vec3) pretend Vector(5))
```

or thus

```
((vectorSum 5) pretend (Vector(2+3) -> Integer)) append(2,3,vec2,vec3)
```

This achieves a result, but at some cost. Wherever we expect to need some degree of evaluation, that has to be shadowed by a type cast; these casts are also potentially completely unsafe.

[2] The `pretend` function is used in the definition of `rep` and `per` in the current version of Aldor; a more secure mechanism would be preferable.

3.2 Coercion Functions

Another possibility is to suggest that the current mechanism for coercions in Aldor is modified to include coercion functions which would provide conversion between type pairs such as `Vector(2+3)` and `Vector(5)`, extending the coercion mechanism already present in Aldor. This suggestion could be implemented but we envisage two difficulties with it.

- In all but the simplest of situations we will need to supply uniformly-defined *families* of coercions rather than single coercions. This will substantially complicate an already complex mechanism.
- Coercions are currently not applied transitively: the effect of this is to allow us to model single steps of evaluation but not to take their transitive closure.

Putting these two facts together force us to conclude that effectively mimicking the evaluation process as coercions is not a reasonable solution to the problem of modifying type checking.

3.3 Adding Full Evaluation

To deal with the problem of unevaluated subexpressions in types, we have implemented a prototype version of Aldor using Haskell [23]. In this implementation all type expressions are fully evaluated to their *normal form* as a part of the process of type checking. To give an example, the rule (function elim) will be interpreted thus:

> $f(s)$ is well-formed if and only if f has type S`->`T, s has type S', *and the normal forms of S and S' are equal modulo α-equality.*

The effect of this modification is to force the type checker to perform evaluation of expressions at compile time. Clearly this can cause the type checker to diverge in general, since in, for instance, an application of the form `vectorSum(e)` an arbitrary expression `e:Nat` will have to be evaluated.

More details of the prototype implementation of Aldor in Haskell are given in the technical report [23].

3.4 Controlling Full Evaluation

A number of existing type systems, Haskell among them, have undecidable type systems [12] which can diverge at compile time. In practice this is not usually a problem as the pathologies lie outside the 'useful' part of the type system. This may well be the case with Aldor also, but it is also possible to design a subset of the language, `Aldor--`, whose type system is better behaved.

There is considerable current interest in defining *terminating* systems of recursion [27, 16]. A system like this is sufficient to guarantee the termination of expressions chosen for evaluation as part of the type checking process. The main effect of the restricted system is to force recursion to be structural (in a general sense); in practice this is acceptable, particularly in the subset of the language used within type expressions.

4 Logic within Aldor

In this section we discuss the Curry-Howard isomorphism between propositions and types, and show that it allows us to embed a logic within the Aldor type system, if dependent types are implemented to allow evaluation within type contexts.

4.1 The Curry-Howard Correspondence

Under the Curry-Howard correspondence, logical propositions can be seen as types, and proofs can be seen as members of these types. Accounts of constructive type theories can be found in notes by Martin-Löf [15] amongst others [19, 26]. Central to this correspondence are dependent types, which allow the representation of predicates and quantification.

Central to the correspondence is the idea that a constructive proof of a proposition gives enough evidence to witness the fact that the proposition stands.

- A proof of a conjunction $A \wedge B$ has to prove each half of the proposition, so has to provide witnessing information for each conjunct; this corresponds precisely to a product type, in Aldor notation written as (A, B), members of which consist of pairs of elements, one from each of the constituent types.
- A proof of an implication $A \Rightarrow B$ is a proof transformer: it transforms proofs of A into proofs of B; in other words it is a function from type A to type B, i.e. a function of type A`->`B.
- In a similar way a proof of a universal statement $(\forall x : A)B(x)$ is a function taking an element a of A into a proof of $B(a)$; in other words it is an element of the *dependent* function type `(x:A) -> B`.
- Similar interpretations can be given to the other propositional operators and the existential quantifier.

We can summarise the correspondence in a table

Programming		**Logic**
Type		Formula
Program		Proof
Product/record type	`(...,...)`	Conjunction
Sum/union type	`\/`	Disjunction
Function type	`->`	Implication
Dependent function type	`(x:A) -> B(x)`	Universal quantifier
Dependent product type	`(x:A,B(x))`	Existential quantifier
Empty type	`Exit`	Contradictory proposition
One element type	`Triv`	True proposition
...		...

Predicates (that is dependent types) can be constructed using the constructs of a programming language. A direct approach is to give an explicit (primitive recursive) definition of the type, which in Aldor might take the form

```
lessThan(n:Nat,m:Nat) : Type ==                             (lessThan)
    if m=0 then         Exit
    else (if n=0 then Triv
                 else lessThan(n-1,m-1));
```

The equality predicate can be implemented by means of a primitive operation which compares the normal forms of the two expressions in question.

4.2 A Logic within Aldor

We need to examine whether the outline given in Section 4.1 amounts to a proper embedding of a logic within Aldor. We shall see that it places certain requirements on the definition and the system.

Most importantly, for a definition of the form (lessThan) to work properly as a definition of a predicate we need an application like lessThan(9,3) to be reduced to Exit, hence we need to have evaluation of type expressions. This is a modification of Aldor which we are currently investigating, as outlined in Section 2.3. In the case of (lessThan) the evaluation can be limited, since the scheme used is recognisable as terminating by, for instance, the algorithm of [16].

The restriction to terminating (well-founded) recursions is also necessary for consistency of the logic. For the logic to be consistent, we need to require that not all types are inhabited, which is clearly related to the power of the recursion schemes allowed in Aldor. One approach is to expect users to check this for themselves: this has a long history, beginning with Hoare's axiomatisation of the function in Pascal, but we would expect this to be supported with some automated checking of termination, which ensures that partially or totally undefined proofs are not permitted.

Consistency also depends on the strength of the type system itself; a sufficiently powerful type system will be inconsistent as shown by Girard's paradox [11].

5 Applications of an Integrated Logic

Having identified a logic within Aldor, how can it be used? There are various applications possible; we outline some here and for others one can refer to the number of implementations of type theories which already exist, including Nuprl [7] and Coq [8].

5.1 Pre- and Post-Conditions

A more expressive type system allows programmers to give more accurate types to common functions, such as the function which indexes the elements of a list.

```
index : (l:List(t))(n:Nat)((n < length l) -> t)
```

An application of `index` has *three* arguments: a list `l` and a natural number `n` – as for the usual index function – and a third argument of type `(n < length l)`, that is a *proof* that `n` is a legitimate index for the list in question. This extra argument becomes a *proof obligation* which must be discharged when the function is applied to elements `l` and `n`.

In a similar vein, it is possible to incorporate post-conditions into types, so that a sorting algorithm over lists might have the type

```
sort : ((l:List(t))(List(t),Sorted(l))
```

and so return a sorted list together with a proof that the list is `Sorted`.

5.2 Adding Axioms to the Categories of Aldor

In definition `(Mon)`, Section 2.5, we gave the category of monoids, `Monoid`, which introduces two operation symbols, `*` and `1`. A monoid consists not only of two operations, but of operations with properties. We can ensure these properties hold by extending the definition of the category to include three extra components which are proofs that `1` is a left and right unit for `*` and that `*` is associative, where we assume that '≡' is the equality predicate:

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

    leftUnit  : (g:%) -> (1*g ≡ g);
    rightUnit : (g:%) -> (g*1 ≡ g);
    assoc     : (g:%,h:%,j:%) -> ( g*(h*j) ≡ (g*h)*j );
}
```

For example, the declaration of `leftUnit` has the logical interpretation that `leftUnit` is a proof of the statement 'for all `g` in the monoid (%), `1*g` is equal to `g`'.

The equality predicate is implemented as follows: the type `a ≡ b` contains a value if and only if `a` and `b` have the same normal form. The extension operation (i.e. the `with` in the definition above) over categories will lift to become operations of extension over the extended 'logical' categories such as `(MonL)`.

5.3 Commutative Monoids

In the current library for Axiom it is not possible to distinguish between general monoids and commutative monoids: both have the same signature. With logical properties it is possible to distinguish the two:

```
CommutativeMonoid : Category == Monoid with {
comm : (g:%,h:%) -> ( g*h ≡ h*g );
}
```

To be a member of this category, a domain needs to supply an extra piece of evidence, namely that the multiplication is commutative; with this evidence the structure can be treated in a different way than if it were only known to be a monoid. This process of discovery of properties of an mathematical structure corresponds exactly to a mathematician's experience. Initially a structure might be seen as a general monoid, and only after considerable work is it shown to be commutative; this proof gives entry to the new domain, and thus allows it to be handled using new approaches and algorithms.

5.4 Different Degrees of Rigour

One can interpret the obligations given in Sections 5.1 and 5.2 with differing degrees of rigour. Using the `pretend` function we can conjure up proofs of the logical requirements of `(MonL)`; even in this case they appear as important documentation of requirements, and they are related to the lightweight formal methods of [9].

Alternatively we can build fully-fledged proofs as in the numerous implementations of constructive type theories mentioned above, or we can indeed adopt an intermediate position of proving properties seen as 'crucial' while asserting the validity of others.

6 Conclusion

We have described a new way to combine – or rather, to integrate – computer algebra and theorem proving. Our approach is similar to [3] and [4] in that theorem proving capabilities are incorporated in a computer algebra system. (In the classification of possible combinations of computer algebra and theorem proving of [6], all these are instance of the "subpackage" approach.) But the way in which we do this is completely different: we exploit the expressiveness of the type system of Aldor, using the Curry-Howard isomorphism that also provides the basis of theorem provers based on type theory such as Nuprl [7] or Coq [8]. This provides a logic as part of the computer algebra system. Also, having the same basis as existing theorem provers such as the ones mentioned above makes it easier to interface with them.

So far we have worked on a formal description of the core of the Aldor type system [22], and on a pilot implementation of a typechecker for Aldor which does evaluation in types which can be used as a logic [23]. This pilot forms the model for modifications to the Aldor system itself, as well as giving a mechanism for interfacing Aldor with other systems like the theorem prover Coq, complementary to recent work on formalising the Aldor system within Coq [1]. The logic is being used in a mathematical case study of symbolic asymptotics [25].

It is interesting to see a convergence of interests in type systems from a number of points of view, namely

– computer algebra,
– type theory and theorem provers based on type theory,
– functional programming.

For instance, there seem to be many similarities between structuring mechanisms used in these different fields: [5] argues for functors in the sense of the programming language ML as the right tool for structuring mathematical theories in Mathematica, and [24] notes similarities between the type system of Aldor, existential types [18], and Haskell classes [28]. More closely related to our approach here, it is interesting to note that constructive type theorists have added inductive types [20], giving their systems a more functional flavour, while functional programmers are showing an interest in dependent types [2] and languages without non-termination [27]. We see our work as part of that convergence, bringing type-theoretic ideas together with computer algebra systems, and thus providing a bridge between symbolic mathematics and theorem proving.

Acknowledgements We are grateful to Stephen Watt of the University of Western Ontario and to Ursula Martin and her research group at the University of St Andrews for feedback on these ideas. We would also like to thank NAG for granting us access to the Aldor compiler, and in particular to Mike Dewar for his help in facilitating this. We are indebted to Dominique Duval who first introduced us to the type system of Aldor, and to EPSRC for supporting her visit to UKC under the MathFIT programme. Finally we are grateful to Therèse Hardin of LIP6, Paris, for her comments on this work.

References

1. Guillaume Alexandre. *De* ALDOR *à Zermelo.* PhD thesis, Université Paris VI, 1998.
2. Lennart Augustsson. Cayenne – a language with dependent types. ACM Press, 1998.
3. 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.
4. Bruno Buchberger. Symbolic Computation: Computer Algebra and Logic. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*. Kluwer, 1996.
5. Bruno Buchberger, Tudor Jebelean, Franz Kriftner, Mircea Marin, Elena Tomuta, and Daniela Vasaru. A survey of the Theorema project. In *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation)*, pages 384–391. ACM, 1997.
6. Jaques Calmet and Karsten Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In *FroCos'96*. Kluwer, 1996.
7. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall Inc., 1986.
8. C. Cornes et al. The Coq proof assistant reference manual, version 5.10. Rapport technique RT-0177, INRIA, 1995.

9. Martin Dunstan and Tom Kelsey. Lightweight Formal Methods for Computer Algebra Systems. ISSAC'98, 1998.
10. Richard Fateman. Why computer algebra systems can't solve simple equations. *ACM SIGSAM Bulletin*, 30, 1996.
11. Jean-Yves Girard. Intérpretation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieure. Thèse d'Etat, Université Paris VII, 1972.
12. Fritz Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15, 1993.
13. Richard D. Jenks and Robert S. Sutor. *Axiom: The Scientific Computation System.* Springer, 1992.
14. Ursula Martin. Computers, reasoning and mathematical practice. In Helmut Schwichtenberg, editor, *Computational Logic, Marktoberdorf 1997.* Springer, 1998.
15. 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.
16. D. McAllester and K. Arkondas. Walther recursion. In M.A. Robbie and J.K. Slaney, editors, *CADE 13.* Springer, 1996.
17. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.
18. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Prog. Lang. and Syst.*, 10(3):470–502, 1988.
19. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory — An Introduction.* Oxford University Press, 1990.
20. Christine Paulin-Mohring. Inductive definitions in the system Coq. In *TLCA*, volume 664 of *LNCS*. Springer, 1993.
21. John Peterson and Kevin Hammond, editors. *Report on the Programming Language Haskell, Version 1.4.* `htttp://www.haskell.org/report/`, 1997.
22. Erik Poll and Simon Thompson. The Type System of Aldor. Technical Report 11-99, Computing Laboratory, University of Kent at Canterbury, 1999.
23. Chris Ryder and Simon Thompson. Aldor meets Haskell. Technical Report 15-99, Computing Laboratory, University of Kent at Canterbury, 1999.
24. Philip S. Santas. A type system for computer algebra. *Journal of Symbolic Computation*, 19, 1995.
25. J.R. Shackell. Symbolic asymptotics and the calculation of limits. *Journal of Analysis*, 3:189–204, 1995. Volume commemorating Maurice Blambert.
26. Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.
27. David Turner. Elementary strong functional programming. In Pieter Hartel and Rinus Plasmeijer, editors, *Functional programming languages in education (FPLE), LNCS 1022.* Springer-Verlag, Heidelberg, 1995.
28. Philip Wadler and Stephen Blott. Making *ad hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages.* ACM Press, 1989.
29. Stephen M. Watt et al. A First Report on the $A^{\#}$ Compiler. In *ISSAC 94*. ACM Press, 1994.
30. Stephen M. Watt et al. *AXIOM: Library Compiler User Guide.* NAG Ltd., 1995.