Adding the axioms to Axiom
Towards a system of automated reasoning in Aldor

Erik Poll & Simon Thompson
Computing Laboratory, University of Kent
Canterbury, Kent, CT2 7NF, UK

{E.Poll,S.J.Thompson}@ukc.ac.uk

Abstract

A number of combinations of theorem proving 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.

1 Introduction

Symbolic mathematical – or computer algebra – systems, such as Axiom [JS92], 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 [Fat96] 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 [Mar98] 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: the logical capabilities of a theorem prover could be used to express the assumptions upon which an answer rests, and in critical cases be used to establish the truth of those assumptions.

The literature contains a number of different strategies proposed for combining computer algebra and theorem proving; see, for instance, [Buc96, CH96, BCZ96]. This paper examines another proposal: that of using the type system of the Axiom [JS92] computer algebra system 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 [C^{+}86] or Coq [Cor95].

This paper particularly explores the recent Axiom Library Compiler, Aldor [W^{+}95], which is unusual among computer algebra systems in being strongly typed, and moreover in having a very powerful type system, including dependent types.

The implementation of dependent types in Aldor is without evaluation of type expressions – so each type expression is its own normal form – and we show how this limits the expressivity of the dependent types. We propose 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 Axiom.

The structure of the paper is as follows. Section 2 introduces Aldor, and more details of Aldor dependent types appear in Section 3. We show how a logic can be defined in (a 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 [W^{+}95] (previously known as AXIOM-XL or $A^{\sharp}$), provides the user with a powerful, general-purpose programming language in which to model the structures of mathematics. This language is functionally based and provides higher order-functions, generators (which bear a strong relationship to list comprehensions) and other features of modern functional languages like Standard ML [MTH90] and Haskell [PH97], as well as being strongly typed.

2.1 The type system of Aldor

Unusually among languages for computer algebra, but in keeping with the functional school, Aldor is strongly typed, and each declaration of a binding is accompanied by a declaration of the type of the value bound, as in the example

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

This contrasts with languages like Haskell in which types need not be declared explicitly since they can be deduced by the system.[1] In Aldor types have to be declared explicitly since the type system has a variety of complex features including the following.

Overloading A single identifier can be used to denote values of different type, such as `Int -> Int` and `Int -> Bool -> Int`.[2] Some support is provided for users to resolve overloading when that proves to be necessary.

Coercions 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.

Types as values The type `Type` is itself a type; it is by this means that the system supports functions over types, such as

```
idType (ty : Type) : Type == ty;                    (1)
```

and explicit polymorphism, as in

```
id (ty : Type, x : ty) : ty == x;                   (2)
```

This resembles the quantification in System F in which functions can depend on type parameters, and in F^ω where functions can be defined from types to types. This is investigated further in Section 3.1.

Dependent types 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

[1] Most Haskell programmers would, however, tend to give type declarations for their definitions, since they serve both as an important check on the programmer's intention as well as providing crucial documentation.

[2] Note that this is a much more powerful overloading facility than that provided by Haskell type classes [PH97] in which overloaded functions have to be of the same arity.

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

in which the result of a function application, say

```
vectorSum(34)
```

has the type `Vector(34) -> Double` because its argument has the value `34`. In a similar way, when the `id` function of definition `(2)` 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 3.

Variables 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 Axiom.

Categories and domains These features which provide a form of data abstraction are addressed in more detail in Section 2.2.

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 2.2) complicate the picture further; one of the aspects of our work is to try to reach a more formal description of (substantial parts of) the typing mechanism of Aldor.

2.2 Categories and domains

Axiom 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` [3] called `Monoid`, thus

```
Monoid : Category == BasicType with {                    (3)
    * : (%,%) -> %;
    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 '`%`'.

[3] There is no relation between Axiom's notion of category and the notion from category theory!

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 Axiom as domains, and are defined as was the value a above

```
IntegerAdditiveMonoid : Monoid == add {                          (4)
    Rep == Integer;
    import from 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

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

In fact, Rep, rep and per are implemented using the macro mechanism of Aldor, and so are eliminated before type checking. Another aspect of Aldor which we intend to explore are ways in which the macro system can be eliminated in favour of a properly type checked mechanism, and so in particular support the type abstraction machinery introduced here.

We have seen already that one category can extend another; this can be seen as a form of inheritance. Other operations are available on categories, including the Join operation which joins two categories, thus allowing a form of multiple inheritance.

Categories provide a powerful abstraction mechanism which allows functions to be written which depend on the bindings in a category, and which can therefore be used over any domain which implements the category; that is any abstract data type which implements the signature in question. Categories resemble existential types [MP88, San95] but are implemented in a similar way to Haskell classes in that the type of the operands in an application determine which implementation of the operation is used.

One of the advantages of the Haskell type class mechanism is that it is possible to declare a type as an instance of different classes at different points in a program. In the first version of Axiom this was not allowed, so that an Axiom domain would have a signature (this is category) fixed at the point of definition and could not be extended to become an instance of a newly-defined category; this is possible in Aldor, using a *post facto* extension. Details of this and other features can be found in [W$^+$95].

Categories can also be parametric, and depend upon value or type parameters. We shall see an example of this in Section 3.1 below. We continue to explore the Aldor type system in the following section where we look in more detail at the dependent types of the system.

3 Dependent types

The Aldor language contains dependent types, so that one can define functions such as `vectorSum` which defines a sum function for vectors of arbitrary length, of type

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

and a function `append` to join two vectors together

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

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

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

and 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).

In Section 3.2 we discuss how the Aldor type mechanism can be modified to accommodate a more liberal evaluation strategy within the type checker. Before doing that we look at another example of the problems caused by the failure of Aldor to evaluate type expressions.

3.1 Types as values

Another example is provided by trying to formalise the notion from (mathematical) category theory [Mac72], namely that of a functor. Specifically we are thinking of the types of the language as forming a category, whose objects are the types themselves, and an arrow from `A` to `B` is a function of type `A->B`. What is a functor from type to itself? It has two components

- a mapping F, say, from Type to Type; that is the object part of the functor; and
- a mapping on functions which respects their types, so that the image of an arrow a->a is an arrow (F a)->(F b).

(There are also some logical constraints on the behaviour of the these mappings; we cannot formalise these in Aldor.) How can this be formalised in Aldor? We say

```
Functor (F : Type -> Type) : Category == with {              (5)
map : (a:Type) -> (b:Type) -> (a->b) -> (F a) -> (F b)
};
```

and we can show that the List functor, which builds the type List(a) from the type a, is an instance of Functor thus:

```
listFunctor : Functor(List) == add {                          (6)
map (a:Type)(b:Type)(f:a->b)(x:List(a)) : List(b)
  == if x=nil then nil
              else cons ( f (first(x)),
                        (((map a) b) f) rest(x))
};
```

which is the standard definition of map over lists.

In a similar vein we can try to make idType (as defined in Section 2.1 above) into an instance of Functor,

```
Ident : Functor(idType) == add {                              (7)
map (a:Type)(b:Type)(f:a->b)(x:idType(a)) : idType(b)
  == f x
};
```

but this will fail to be type correct since the types idType(a) and a will not be identified as would be required for the application of f to x to be well-typed.

Observe, however, that (6) does type check. This is because List is a constructor of types, that is the application List(a) is already fully evaluated, and so there is no problem in identifying it with other Type expressions.

In the next section we examine possible modifications to the Aldor type mechanism to accommodate evaluation within type expressions.

3.2 Investigating the Aldor dependent type mechanism

We are currently investigating ways in which the Aldor dependent type mechanism can be modified to allow evaluation within type contexts as well as within value contexts. A number of possibilities suggest themselves.

- The type system provides the `pretend` conversion routine which converts or (type) casts any type to any other type,[4] so that one can rewrite (7) as follows

```
Ident : Functor(idType) == add {                              (8)
map (a:Type)(b:Type)(f:a->b)(x:idType(a)) : idType(b)
  == (f (x pretend a) pretend idType(b))
};
```

  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.

- Another possibility is to suggest that the system is modified to include coercion functions which would provide conversion between type pairs such as `idType(a)` and `a`. 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 to hand.

Instead of pursuing either of these solutions we are currently investigating the possibility of performing some evaluation during type checking. 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. We therefore intend to use current work on terminating systems of recursion [MA96] as well as restrictions on the types of expression chosen for evaluation (as heuristics at least) to guide the form of evaluation that is supported.

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] 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.

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 [ML84] amongst others [NPS90, Tho91]. Central to this correspondence are dependent types, which allow the representation of predicates and quantification. 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 in a number of different ways. We look at the example of the 'less than' predicate over the natural numbers.

- A first 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 ==                    (9)
    if m=0 then       Exit
    else (if n=0 then Triv
                 else lessThan(n-1,m-1));
```

- A second approach introduces them inductively as a generalisation of the algebraic types which appear in Haskell and SML. We might define the less than predicate '`<`' of type `Nat -> Nat -> Type` by saying that there are two constructors for the type of the following signature

```
ZeroLess : (n:Nat)(0 < S n)
SuccLess : (m:Nat)(n:Nat)((m < n) -> (S m < S n))
```

  This approach leads to a powerful style of proof in which inductions are performed over the form of proof objects, that is the elements of types like `(m < n)`, rather than over (say) the natural numbers, and such a method makes much more manageable a proof of the transitivity of '`<`' over `Nat`, say. Inductive types have been used extensively in the type theory community, see, for instance, [PM93].

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 `(9)` 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 3. In the case of `(9)` the evaluation can be limited, since the scheme used is recognisable as terminating by, for instance, the algorithm of [MA96].

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 [Gir72].

5 Applications of an integrated logic

If we identify 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 [C^+86] and Coq [Cor95].

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 standard index function — and a third argument which is 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 Axiom

In definition `(3)`, Section 2.2 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 identity predicate:

```
Monoid : Category == BasicType with {                         (10)
    * : (%,%) -> %;
    1 : %;

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

The extension and join operations over categories will lift to become operations of extension and join over the extended 'logical' categories such as `(10)`.

5.3 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 `(10)`; even in this case they appear as important documentation of requirements, and they are related to the lightweight formal methods of [DK98].

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 proposed a new way to combine – or rather, to integrate – computer algebra and theorem proving. Our proposal is similar to [BCZ96] and [Buc96] 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 [CH96], all these are instance of the "subpackage" approach.) But the way in which we propose to do this is completely different: we propose to exploit the expressiveness of the type system of Axiom, using the Curry-Howard isomorphism that also provides the basis of theorem provers based on type theory such as Nuprl [C$^+$86] or Coq [Cor95]. 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 should make it easier to interface with them.

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: [BJK$^+$97] argues for functors in the sense of the programming language ML as the right tool for structuring mathematical theories in Mathematica, and [San95] notes similarities between the type system of Axiom, existential type [MP88], and Haskell classes [WB89]. More closely related to our proposal here, it is interesting to note that constructive type theorists have added inductive types [PM93], giving their systems a more functional flavour, while functional programmers are showing an interest in dependent types [Aug97] and languages without non-termination [Tur95]. 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.

Our future work will involve investigating the type system of Aldor, and integrating a logical approach to Aldor types; this is complementary to recent work on formalising the Aldor system within Coq [Ale98]. We also expect to apply our work in a case study with mathematical colleagues.

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. Finally, 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.

References

[Ale98] Guillaume Alexandre. *De* ALDOR *à Zermelo*. PhD thesis, Université Paris VI, 1998.

[Aug97] Lennart Augustsson. Cayenne - a language with dependent types. `www.cs.chalmers.se/~augustss/cayenne/`, 1997.

[BCZ96] Andrej Bauer, Edmund Clarke, and Xudong Zhao. Analytica - an experiment in combining theorem proving and symbolic computation. In *Artificial Intelligence and Symbolic Mathematical Computation (AISMC-3)*, volume 1138 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 1996.

[BJK+97] 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.

[Buc96] Bruno Buchberger. Symbolic Computation: Computer Algebra and Logic. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems*, Applied Logic Series. Kluwer, 1996.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall Inc., 1986.

[CH96] Jaques Calmet and Karsten Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In *FroCos'96*, pages 133–146. Kluwer Series on Applied Logic, 1996.

[Cor95] C. Cornes et al. The Coq proof assistant reference manual, version 5.10. Rapport technique RT-0177, INRIA, 1995.

[DK98] Martin Dunstan and Tom Kelsey. Lightweight Formal Methods for Computer Algebra Systems. To appear in the proceedings of ISSAC'98, 1998.

[Fat96] Richard Fateman. Why computer algebra systems can't solve simple equations. *ACM SIGSAM Bulletin*, 30, 1996.

[Gir72] Jean-Yves Girard. Intérpretation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieure. Thèse d'Etat, Université Paris VII, 1972.

[JS92] Richard D. Jenks and Robert S. Sutor. *Axiom: The Scientific Computation System*. Springer-Verlag, 1992.

[MA96] D. McAllester and K. Arkondas. Walther recursion. In M.A. Robbie and J.K. Slaney, editors, *CADE 13*. Springer-Verlag, 1996.

[Mac72] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1972.

[Mar98] Ursula Martin. Computers, reasoning and mathematical practice. In Helmut Schwichtenberg, editor, *Computational Logic, Marktoberdorf 1997*. Springer-Verlag, 1998. *To appear.*

[ML84] 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.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Prog. Lang. and Syst.*, 10(3):470–502, 1988.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory — An Introduction*, volume 7 of *International Series of Monographs on Computer Science.* Oxford University Press, 1990.

[PH97] John Peterson and Kevin Hammond, editors. *Report on the Programming Language Haskell, Version 1.4.* `www.haskell.org/report`, 1997.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[San95] Philip S. Santas. A type system for computer algebra. *Journal of Symbolic Computation*, 19(1–3):79–110, 1995.

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.

[Tur95] David Turner. Elementary strong functional programming. In *Functional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995.

[W+95] Stephen M. Watt et al. *AXIOM: Library Compiler User Guide.* The Numerical Algorithms Group Limited. First edition, reprinted with corrections, 1995.

[WB89] 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.