Formal Methods for Extensions to CAS

Martin N. Dunstan*, Tom Kelsey, Ursula Martin, and Steve Linton

Division of Computer Science,
University of St Andrews,
North Haugh, St Andrews, UK
{mnd,tom,um,sal}@dcs.st-and.ac.uk

Abstract. We demonstrate the use of formal methods tools to provide a semantics for the type hierarchy of the AXIOM computer algebra system, and a methodology for Aldor program analysis and verification. We give a case study of abstract specifications of AXIOM primitives, and provide an interface between these abstractions and Aldor code.

1 Introduction

In this paper we report on the status of our work at St Andrews on the application of formal methods and machine assisted theorem proving techniques to improve the robustness and reliability of computer algebra systems (CAS). We present a case study which demonstrates the use of formal methods for extending existing CAS code. This paper is an extension of the work described in [9]. We have adopted the Larch [16] system of formal methods languages and tools, and applied them to the AXIOM [19] computer algebra system. NAG Ltd, who develop AXIOM and partially fund this project, are optimistic that our formal methods approach will aid system users.

We have constructed a formal model of the AXIOM algebraic category hierarchy, and developed a methodology for formally verifying type assertions contained in the AXIOM library. We have also created a Larch behavioural interface specification language (BISL) called Larch/Aldor and a prototype verification condition generator for the AXIOM compiled language, Aldor (see Section 2.4). This work enables interface specifications (also known as annotations) to be added to Aldor programs. These can be used for

- clear, concise, unambiguous and machine checkable documentation.
- lightweight verification (described in more detail in Section 3): helps users to identify mistakes in programs which compilers are unable to detect.
- compiler optimisations: specifications could be used to select between different function implementations, as described in [29].
- method selection: users could interrogate libraries for functions which perform a particular task under specific conditions, as described in [31].

* Funded by NAG Ltd

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1758–1777, 1999.

Although we have chosen to follow the Larch methodology which is based on a two-tiered specification system, we do not preclude the use of other formal methods such as VDM [20], or Z [27]. Other proof tools, especially those with higher order functionality such as PVS [25] or HOL [14], could be used. Nor do we rule out the application to other CAS such as Maple [4] and Mathematica [32]; in fact the weaker type systems of these and other CAS may benefit more from our approach than AXIOM has. Our approach is to use an automated theorem prover as a tool for debugging formal specifications used in the design and implementation of libraries for CAS. Our goal is to increase the robustness of CAS.

In the rest of this introduction we motivate our work and discuss the uses of verification conditions (VC's) generated from annotations. In Section 2 we introduce Larch and its algebraic specification language LSL. Then in Section 2.2 we explain how proofs of LSL specifications can be used to investigate claims made in the documentation of AXIOM categories and domains. This is followed by Sections 2.3 and 2.4 which describe Larch BISL's, with particular reference to Larch/Aldor. In Section 3 we describe the application of our technique of specification lightweight verification and condition generation to CAS in general, and to AXIOM in particular. Section 4 is a case study concerning AXIOM complex numbers, which illustrates how incorrect behaviour within AXIOM can be corrected both by abstract specification and the use of annotations. The final section is an outline of our conclusions and related work.

1.1 Motivation

Computer algebra systems are environments for symbolic calculation, which provide packages for the manipulation of expressions involving symbols. These symbols may, at some point, be assigned concrete numeric values. General purpose computer algebra systems, such as AXIOM [19], Maple [4], or Mathematica [32], as well as more specialised tools such as GAP [12] for computational discrete mathematics or the AXIOM/PoSSo library for high-performance polynomial system solving, are used by many different communities of users including educators, engineers, and researchers in both science and mathematics. The specialised systems in particular are extremely powerful. The PoSSo library has been used to compute a single Gröbner basis, used to obtain a solution of a non-linear system, which (compressed) occupies more than 5GB of disk space, while GAP is routinely used to compute with groups of permutations on millions of points.

After pioneering work in the 1960s CAS have become mainstream commercial products: everyday tools not only for researchers but also for engineers and scientists. For example Aerospatiale use a Maple-based system for motion planning in satellite control. The systems have become more complicated, providing languages, graphics, programming environments and diverse sophisticated algorithms for integration, factorisation and so on, to meet the needs of a variety of users, many not expert in mathematics. All the usual software engineering issues arise, such as modularity, re-use, interworking and HCI. NAG's AXIOM [19] is a sophisticated, strongly typed CAS: user and system libraries are written in

the Aldor language which supports a hierarchy of built-in parameterised types and algorithms for mathematical objects such as rings, fields and polynomials. Aldor is interpreted in the AXIOM kernel which provides basic routines such as simplification and evaluation: code developed in Aldor may also be compiled to C for export to other products. Because such systems are large and complicated (and the algorithms are often developed by domain experts with considerable specialist knowledge) a body of library material has accrued, much of which is widely used even if not necessarily well documented or even entirely understood. For example, it may be known to experts that a certain routine is correct if the input is a continuous function, but because continuity is undecidable this may never be checked at run-time, and it may not even be noted in any obvious way in the documentation, so that an inexpert user may easily make mistakes.

AXIOM/Aldor users can be grouped into three types:

- command line users, who have access to a comprehensive graphical hypertext system of examples and documentation
- system developers, who may be expected to know about any pitfalls involving the libraries
- library developers (writing Aldor programs), who need more support than the description of routines in isolation, and who may be unaware of the subtle dependencies and conditions contained in the AXIOM type system.

Our project aims to improve the provision of support for this third group of users. It also encourages the reuse of software by providing unambiguous documentation for functions. We do not address the accuracy of the results of procedures; computer algebra algorithms have been developed by experts and are generally sound when applied correctly. However there can be hidden dependencies and implicit side conditions present which can lead to erroneous or misinterpreted results. Examples include inconsistent choice of branch cuts in integration algorithms [7], invalid assumptions for the types of arguments of a function or poorly documented side-conditions. Moreover CAS often contain several procedures which perform the same task, but which are optimised for a particular input domain. It is often not easy to select the best procedure without either a detailed knowledge of the system or a lengthy perusal of the documentation.

1.2 Using Verification Conditions

Part of our work is concerned with the generation of verification conditions (VC's) from Aldor programs which have been annotated with Larch/Aldor specifications. VC's are logical statements that describe the conditions under which a program satisfies its specification; they may be created during attempts of correctness proofs (see Section 3.1). However, once VC's have been generated one might ask what can we do with them? Ideally we would attempt to prove or disprove them but in practice this may be infeasible. For example, the GAP4 CAS [12] contains a small module which would generate an apparently simple verification condition. However, the proof of this VC relies on the "Odd Order

Theorem" whose proof occupied an entire 255 page issue of the Pacific Journal of Mathematics [11]. Other examples might include statements about continuity of mathematical functions or computational geometry. Generating verification conditions by hand is tedious even for tiny programs and so a mechanical program would normally be used. Once the verification conditions have been generated there are several options:

- trivial VC's might be automatically discharged by the generator
- theorem provers or proof assistants might be utilised
- hand-proofs might be attempted
- the user may appeal to their specialist knowledge or authoritative sources
- VC's may be ignored unless they are obviously unsatisfiable
- VC's can be noted in the documentation as extra requirements

We believe that our suggestion that the user may wish to ignore VC's unless they are clearly invalid is justified because obvious mistakes can sometimes be detected more quickly by inspection than by attempting to formally prove/disprove them. For example the VC

$$(\tan x)\ \textit{is-continuous-on}\ (0, \pi)$$

is clearly false and this can be easily seen from the graph of $\tan x$ over the specified interval $(0, \pi)$. However, attempting to show that this is false within a theorem prover is very difficult, requiring a model of the real numbers which is a topic of active research [17, 18].

Proof attempts which fail to show whether a VC is valid or invalid may indicate that the program annotations and/or the background theory needs to be extended. VC's which are found to be invalid mean that there is a mistake, probably in the program or the annotations but possibly in the theory used during the proof. If all VC's can be proved then the program satisfies its specification and the user will have increased confidence that it will behave as expected.

2 Specification and the Larch Approach

In this section we describe the languages and tools which comprise the Larch formal specification system, and propose a methodology for using Larch to specify AXIOM and Aldor components. Examples of specifications which relate directly to the AXIOM/Aldor CAS are provided.

Larch [16] is based on a two-tiered system. In the first tier users write algebraic specifications in a programming-language independent algebraic specification language called the Larch Shared Language (LSL). These specifications provide the background theory for the problem domain and allow the investigation of design options. The second tier consists of a family of behavioural interface specification languages (BISL's), each tailored to a particular programming language. User programs are annotated in the BISL of their choice. BISL specifications are primarily concerned with implementation details such as side-conditions on functions, memory allocation and pointer dereferencing. The Larch philosophy is to

do as much work as possible at the LSL level, leaving implementation-specific details to be described using the BISL. This allows BISL specifications to be both concise and unambiguous.

2.1 The Larch Shared Language

The LSL tier allows the user to define operators and sorts (types) which provide semantics for terms appearing in the BISL annotations. The basic unit of LSL specification is a trait. The following example provides a basic abstraction of complex numbers (providing a constructor of ordered pairs from a commutative ring, and observers for the real and imaginary parts of a complex entity) which will be used in the case study:

```
RequirementsForComplex (CR) : trait
 assumes CommRingCat (CR)
 introduces
   complex : CR,CR → T
   imag, real : T → CR
 asserts
   T partitioned by real, imag
   T generated by complex
   ∀ x,y : CR
     complex(x,y) = complex(u,v) ⇒ x = u ∧ y = v;
     imag(complex(x,y)) == y;
     real(complex(x,y)) == x;
 implies
   ∀ z : T
     z == complex(real(z),imag(z))
```

The sections of the trait have the following meanings:

- `assumes`—textually include other traits (with renaming)
- `introduces`—declare new mix-fix operators
- `asserts`—define a set of axioms
- `implies`—statements implied by the axioms of this trait

The trait defines values of sort `T`, and is parameterized by the sort name `CR`. The `partitioned by` clause states that all distinct values of sort `T` can be distinguished using `real` and `imag`. The `generated by` clause states that all `T` values can be obtained using `complex`. What it means to be a value of sort `CR` is defined in the assumed trait `CommRingCat`. This assumption generates a proof obligation: any supplied argument must be shown to satisfy the axioms of a commutative ring (the LSL `includes` command is used to inherit properties without justification). LaTeX is used for graphical symbols/operators, *e.g.* ∀ is written `\forall`. The first assertion formalises equality for complex values; the reverse implication is automatically true, since LSL operators always return equal results for equal arguments. The remaining assertions provide straightforward semantics for the

observers in terms of the constructor. The `implies` section is used as checkable redundancy; proving the statements provides confidence that the axioms defined are specified correctly. Failed proof attempts may indicate the presence of errors or omissions in the original traits. This section can also provide extra information and lemmas which might not be obvious from the rest of the trait, but are useful properties for another trait to inherit.

A tool called `lsl` can be used to perform syntax and type checking of LSL specifications. It can also convert LSL specifications into the object language of the Larch Prover (LP), a proof assistant for a many-sorted first order logic with induction which can be used to check properties of LSL specifications.

2.2 Specifying AXIOM Using LSL and LP

The specification of AXIOM categories in LSL was described in [9]. The next stage is to specify AXIOM functors and debug these specifications using the Larch Prover. The resulting abstract specifications provide concrete definitions of the primitives which are used in interface specifications (annotations) to produce verification conditions.

An AXIOM category is a set of operator names, signatures and methods which provide an abstract framework for the definition of computer algebra types. A category will, in general, have many models; each implemented model is an AXIOM domain. For example, the AXIOM domains `Matrix Integer` and `Polynomial Integer` are both implementations of the AXIOM category `Ring`. We say that these domains have type `Ring`; their basic operators were defined in the `Ring` category.

AXIOM domains are constructed by functors. These take domains as argument, and return a domain as output. In the above examples `Matrix` and `Polynomial` are the functors, each taking the domain `Integer` as argument. AXIOM assigns a type to each domain returned by a functor. This assignment follows informal inbuilt rules which are not always valid. Thus AXIOM can assign an incorrect type to a functor, and hence obtain incorrect results. We give an example of this incorrect typing behaviour in our case study: AXIOM axioms asserts that a domain with non-zero zero divisors is a field. Prior to our work, the only method of checking the correctness of these assignments was experimentation with AXIOM code in conjunction with detailed examination of AXIOM documentation. This method is unsatisfactory: even if each existing AXIOM domain is tested, there remains the problem of testing domains not yet implemented.

Our approach is to provide a generic methodology, applicable both to existing and potential implementations. We supply LSL specifications of functors which allow us to formally verify that a given implementation is a model of the categories which define the type of a resulting domain. These proofs can be thought of as providing enhanced type-checking. Proof obligations are obtained by adding the clause

```
implies TypeTrait(Sortname, Opnames for names)
```

to the functor trait, where `TypeTrait` is a trait representing an AXIOM category, `Sortname` is the sort name for the domain produced by the functor, and `Opnames for names` replaces high level operator names with appropriate implementation level operator names.

The specifications also allow formal checks that implementations of operators act as expected in the model. For example we can check that abstract algebraic properties hold at the domain level, or that the implementation operators combine together in a satisfactory manner. Moreover an LSL clause of the form `assumes CategoryName(CN)` generates a proof obligation that a specification of an argument domain (with sort name `CN`)) is a model of the specification of `CategoryName`. Hence we can verify that argument domains are of the intended type. Examples of enhanced type-checking, operator suitability proofs, and argument correctness verification are given in Section 4.1

2.3 Larch BISL's

Once the necessary theories have been defined in LSL (and checked with LP), the user can proceed to write their program. In the ideal world implementations would be developed in conjunction with the annotations but in the case of legacy systems this may not be possible. For such systems specifying their behaviour as it has been implemented may be the only option, at least as the first step.

To date there are around 14 different Larch BISL'S for languages ranging from CLU [30] and Modula-3 [21] to C [16] and C++ [23]. Each has been designed to investigate various aspects of imperative programming such as inheritance [23] and concurrency [21] as well as different development methodologies such as specification browsing [5] and interactive program verification [15]. The syntax and use of BISL specifications is essentially the same in all languages. Functions and procedures can be annotated with statements defining their pre- and post-conditions as well as indicating any client-visible state which *might* be modified when the function is executed.

Below is a simple example of an annotated Aldor function declaration for `iqsrt` which computes the integer square root of a positive number:

```
++} requires ¬(x < 0);
++} ensures  (r*r ≤ x) ∧ (x < (r+1)*(r+1));
++} modifies nothing;
iqsrt(x:Integer):(r:Integer);
```

Annotations are embedded in the program source code and appear as special comments marked by lines beginning with "`++}`". In the example above the **requires** clause defines the pre-condition of the function and states that the argument must be a non-negative integer. The **ensures** clause defines the post-condition in terms of the return value "`r`" and places restrictions on the possible set of values that "`r`" may hold such that

$$\forall x \bullet x \geq 0 \Rightarrow \texttt{isqrt}(x) = \lfloor \sqrt{x} \rfloor$$

The **modifies** clause specifies which parts of the client-visible state (such as global variables) *might* be modified when this function is executed. A function is permitted to mutate *at most* the objects listed in the **modifies**—it may alter some or none of them if appropriate.

2.4 Larch/Aldor

As part of our work we have designed a Larch BISL for Aldor, the extension programming language for AXIOM, which we are using to investigate how program annotations can improve the reliability and robustness of computer algebra routines. Aldor programs may be annotated with Larch BISL specifications which can be used as clear, concise and machine-checkable documentation; they may also be used for verification condition generation (see Section 3). An example of a Larch/Aldor program which implements the integer division algorithm is given below.

```
++} requires ¬(g = 0);
++} ensures  (f = ((result.q)*g + result.r))
++}          ∧ (abs(result.r) < abs(g));
++} modifies nothing;
integerDivide(f:INT, g:INT):Record(q:INT, r:INT) == {
   local quo:INT := 0;
   local rem:INT := f;

   ++} requires   ¬(g = 0) ∧ (quo^ = 0) ∧ (rem^ = f);
   ++} ensures    (f = (quo'*g + rem')) ∧ (abs(rem') < abs(g));
   ++} invariant  f = (quo*g + rem);
   ++} measure    abs(rem);
   ++} modifies   quo, rem;
   while (abs(rem) ≥ abs(g)) repeat {
      quo := quo + sign(f)*sign(g);
      rem := rem - sign(f)*abs(g);
   }

   record(quo, rem);
}
```

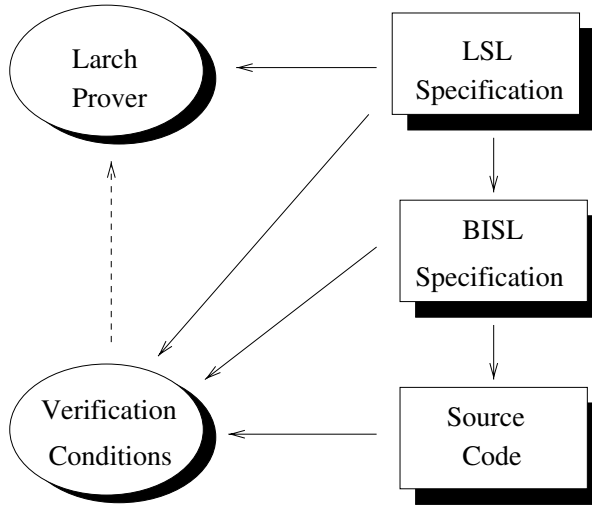
In the annotations of the example above, identifiers represent *logical* values of the corresponding Aldor variables. The identifiers marked with a caret (`^`) indicate that the value is with respect to the state of the program before the function is executed (the pre-state) while the primed identifiers correspond to values in the post-state. Unadorned identifiers are interpreted according to the context and usually have the same value in the pre- and post-states. The identifier `result` is known as a specification or ghost-variable and its value is the return value of the function. It is important to note that operators and functions that appear in the annotations are LSL operators and *not* Aldor functions.

3 Application of the Larch Method to CAS

The AXIOM computer algebra system has a large library containing numerous functors, as described in Section 2.2. Although a few of these can be applied to any type, such as `List(T:Type)`, many have restrictions on the types of domains which they can accept as arguments and which they will return. As shown in the case study in Section 4, the functor `Complex` can only be applied to domains `CR` which are of type `CommutativeRing`. This means that the operations defined by `Complex` are able to rely on the fact that `CR` is a `CommutativeRing`, irrespective of the concrete instance of `CR`. This creates the risk that functors may contain errors that are not revealed by their application to any domain in the existing library, but may appear when new domains are added.

3.1 Lightweight Verification Condition Generation

Our proposal is to formally specify the requirements of the categories and the behaviour of functors, to allow checks that do not depend on specific domains. The diagram below is intended to describe the development used for Larch/Aldor programs. Users begin by writing LSL specifications which provide the theory for their problem. Next the interface specifications and Aldor source are produced, perhaps leaving some functions as stubs without a proper implementation. A separate tool can then generate verification conditions which can be analysed using LP, by hand or by some other theorem prover as appropriate. A prototype VC generator for Larch/Aldor has been implemented in Aldor by the authors.



We use the notation $\{P\}\,\mathtt{C}\,\{Q\}$ to state that the program fragment `C` has the pre-condition P and post-condition Q; P and Q are the specification of `C`. If $\{P\}\,\mathtt{C}\,\{Q\}$ is interpreted as a "partial correctness" statement then it is true, if whenever `C` is executed in a state satisfying P and if the execution of `C` terminates,

then it will be in a state which satisfies Q. If $\{P\}\,\mathtt{C}\,\{Q\}$ is interpreted as being "totally correct" then it must be partially correct and `C` must always terminate whenever P is satisfied. The approach often taken to prove that $\{P\}\,\mathtt{C}\,\{Q\}$ is partially or totally correct is to reduce the statement to a set of purely logical or mathematical formulae called verification conditions [13] or VC's. This is achieved through the use of proof rules which allow the problem to be broken into smaller fragments. For example, the rule for assignment might be:

$$\frac{P \Rightarrow Q\,[\mathtt{e}/\mathtt{v}]}{\{P\}\,\mathtt{v} := \mathtt{e}\,\{Q\}}$$

which states that to prove the partial correctness of $\{P\}\,\mathtt{v} := \mathtt{e}\,\{Q\}$ we need to prove that $P \Rightarrow Q\,[\mathtt{e}/\mathtt{v}]$ where $Q\,[\mathtt{e}/\mathtt{v}]$ represents the formula Q with every occurrence of v replaced with e. For example, the partial correctness proof of $\{x = 0\}\,\mathtt{x} := \mathtt{x} + \mathtt{1}\,\{x = 1\}$ generates the VC $(x = 0) \Rightarrow (x + 1) = 1$; for total correctness we must also show that the evaluation of `e` terminates.

Our approach to verification condition generation is different—the assignment rule in the previous section is relatively simple but the construction of rules for other features of a programming language such as Aldor is not so easy. In particular, determining the verifications resulting from calling a procedure which mutates the values of its arguments is difficult. In [9] we proposed the use of lightweight formal methods to step around this problem in computer algebra systems. Rather than undertaking long verification proofs, we suggest that the correctness of a procedure may be taken on trust.

Using our notation, $\{P\}\,\mathtt{C}\,\{Q\}$ might represent the correctness of a standard library procedure `C`. In any context which executes `C` we have the verification condition that P is satisfied in this context; we can then assume that Q is satisfied in the new context after `C` has terminated. Our justification for this is that we believe it is more likely that programming errors will be due incorrect application of functions or procedures than due to mistakes in the the implementation of computer algebra routines. After all the algorithms upon which they are based have almost certainly been well studied.

As an example, consider the '`isqrt`' function specified in Section 2.3. With our approach we trust that the implementation of this function satisfies its specification, namely that if $\neg(x < 0)$ then the result r satisfies $r * r \leq x < (r + 1) * (r + 1)$. Now whenever we see a statement such as '`a := isqrt(z)`' we can generate the verification condition that $\neg(z < 0)$ holds before the assignment and from the post-condition we infer that $a * a \leq z < (a+1) * (a+1)$ holds afterwards. This inference may help to discharge other verification conditions. Furthermore the user may wish to apply the VC generator to the implementation of '`isqrt`' to check that it does indeed satisfy its specification.

4 Case Study

In this section we analyse the behaviour of specific examples of the AXIOM categories and domains described in Section 2.2. We use these examples to illustrate

incorrect AXIOM output. In Section 4.1 we provide LSL specifications which provide a formal check on the type-correctness of the example domains. The use of BISL's to provide a complementary methodology for checking type-correctness is described in Section 4.3.

Our case study concerns essential side-conditions for a functor in the AXIOM library. These conditions are present only as informal comments in the documentation, which are themselves inaccurate. This can result in erroneous AXIOM development in two ways: (i) the library developer may not be aware of the comments and hence the existence of side-conditions, (ii) the library developer may take the side-conditions into account, but be misled by the inaccurate comments. The AXIOM category `ComplexCategory`

- contains domains which represent the Gaussian integers (`Complex Integer`) and Gaussian rationals (`Complex Fraction Integer`)
- contains analogous domains, also obtained by the use of the functor `Complex`
- defines the constants 0, 1, and the square root of -1
- defines multiplication, addition, and subtraction operators
- defines other useful operators, such as norm and conjugate.

The AXIOM functor `Complex`

- takes an AXIOM domain of type `CommutativeRing`, for example `Integer`
- represents ordered pairs as records of two elements
- implements the operators defined in `ComplexCategory` in terms of the record representation and the structure of the argument domain
- returns an AXIOM domain of computation of type `ComplexCategory`.

AXIOM can behave incorrectly when the argument to `Complex` is an integral domain or a field. An integral domain is a commutative ring in which the product of two non-zero elements is always non-zero. This is known as the "no zero divisors" axiom, and can be written as $\forall x, y\ xy = 0 \Rightarrow x = 0 \vee y = 0$. For example, `Integer` is an AXIOM integral domain. A field is an integral domain in which each non-zero element has a multiplicative inverse.

In AXIOM, a domain of type `ComplexCategory(K)` (where `K` is either an integral domain or a field), is assigned type `IntegralDomain` or `Field` respectively. However, the correctness of this type-assignment is dependent on

(i) $x^2 + y^2 = 0$ having no non-trivial solutions in `K` when `K` is an integral domain
(ii) $x^2 + 1 = 0$ having no solutions when `K` is a field.

These properties do not hold for every integral domain and field. The following AXIOM session demonstrates this: we take the field containing exactly five elements, `PrimeField 5`, and show that `Complex PrimeField 5` is incorrectly given type `Field`, even though $3+i$ and $3-i$ are zero divisors, contradicting one of the field axioms. This behaviour is a consequence of the fact that $x^2 + 1 = 0$ has the solutions 2 and 3 in `PrimeField 5`.

```
(1) → K := PrimeField 5
   (1)  PrimeField 5
                                        Type: Domain

 (2) → Complex K has Field
   (2)  true
                                       Type: Boolean

 (3) → a := 3 + %i :: Complex K
   (3)  3 + %i
                           Type: Complex PrimeField 5

 (4) → b := 3 - %i :: Complex K
   (4)  3 + 4%i
                           Type: Complex PrimeField 5

 (5) → a*b
   (5)  0
                           Type: Complex PrimeField 5
```

Our solution to this incorrect type-assignation, presented in the following section, is to (i) specify the AXIOM category, (ii) provide formal axiomatisations and proofs of the conditions for type correctness, and (iii) import these into the specification of the `Complex` functor. The Aldor library developer is then able to view the conditions in the specification as conditional attributes of the particular argument domain under consideration. Section 4.2 illustrates the verification techniques that we have developed. In Section 4.3 we show how interface specifications can reinforce the properties of the LSL specification of `Complex` by allowing the generation of VC's.

4.1 LSL Specification of the AXIOM Functor `Complex`

The LSL trait `RequirementsForComplex` (given in Section 2.1) defined, at a high level of abstraction, the constructor and observer operations required by an implementation of complex numbers. We specified that elements of sort `T` have extractable real and imaginary parts, can be obtained only as a result of applying the `complex` operator, and are equal iff they have equal real and imaginary parts. The trait assumed that the argument domain has AXIOM type `CommutativeRing`. The `ComplexCategory` trait below lowers the level of abstraction by the provision of (i) constants of sort `T`, (ii) the useful shorthand operators `conjugate` and `norm`, and (iii) multiplication, addition and subtraction over `T`. The assertions supply the standard algebraic notions of multiplication, addition and subtraction of complex ring elements represented (in terms of `complex`) as ordered pairs of elements from the underlying ring. The operators `norm`, `conjugate` and `imaginary` have standard mathematical definitions.
The implications of the trait are:

A Larch handbook [16] traits, which combine to require that `T` is shown to be a commutative ring with unity. Hence `Complex (CR)` is shown to be a commutative ring whenever `CR` is.

```
ComplexCategory (CR) : trait
assumes CommRingCat (CR)
includes RequirementsForComplex (CR)
introduces
  imaginary, 0, 1 : → T
  conjugate : T → T
  norm : T → CR
  __+__, __*__ : T,T → T
  -__ : T → T
asserts ∀ w,z : T
  imaginary == complex(0,1);
  0 == complex(0,0);
  1 == complex(1,0);
  conjugate(z) == complex(real(z),-imag(z));
  norm(z) == (real(z)*real(z)) + (imag(z)*imag(z));
  w + z  == complex(real(w)+real(z),imag(w)+imag(z));
  w*z == complex((real(w)*real(z)) - (imag(w)*imag(z)),
                 (real(w)*imag(z)) + (imag(w)*real(z)));
  -z == complex(-real(z),-imag(z))
implies
   AC (*, T), AC (+, T), Distributive(+, *, T),
   Group(T for T, + for ∘, 0 for unit, -__ for ⁻¹),        } A
   Monoid(T for T, * for ∘, 1 for unit)
  ∀ z,w : T
    imaginary*imaginary == -1;                             } B
```

B A check that `imaginary` has been defined correctly as a square root of the additive inverse of the multiplicative unity element of the underlying ring.

4.2 Proving Properties

Proving the implications labelled *A* and *B* shows directly that an AXIOM domain of type `ComplexCategory` will have inherited the correct properties asserted informally in the AXIOM documentation. These straightforward proof goals normalise immediately in LP. We now address type correctness in the case that the argument `CR` is an integral domain or a field.

The following trait provides the necessary conditions for type-correctness of an AXIOM domain of type `ComplexCategory`. The implications are:

A if the argument type is a field in which $x^2 = -y^2 \iff x = 0$, then the resulting complex type will have multiplicative inverses

B if the argument type is a field in which $x^2 = -1$ never holds, then the complex type will have no zero divisors

C if the argument type is an integral domain in which $x^2 = -y^2 \iff x = 0$, then the complex type is an integral domain.

```
TypeConditions (CR,T) : trait
includes
  CommRingCat (CR), ComplexCategory (CR)
introduces
  TypeCondition_1, TypeCondition_2 : → Bool
  InverseExistence : → Bool
asserts ∀ a,b,c : CR
  TypeCondition_1 ⇒ (a ¬= 0 ⇒ a*a ¬= -(b*b));
  TypeCondition_2 ⇒ (a*a ¬= -1);
  InverseExistence ⇒ (a ¬= 0 ⇒ ∃ c (a*c = 1))
implies ∀ v,z,w : T
  TypeCondition_1 ∧ noZeroDivisors ∧ InverseExistence
                  ⇒ (w ¬= 0 ⇒ ∃ v (w*v = 1));          } A
  TypeCondition_2 ∧ noZeroDivisors ∧ InverseExistence
                  ⇒  (w*z=0 ⇒ w=0 ∨ z=0);              } B
  TypeCondition_1 ∧ noZeroDivisors ⇒ (w*z=0 ⇒ w=0 ∨ z=0) } C
```

Proof of implication A:
Suppose that the relevant conditions hold, and that $w = (a, b)$ is non-zero. Then $a^2 + b^2 \neq 0$ (by type condition 1), and so there exists a c such that $c(a^2 + b^2) = 1$ (by inverse condition). By setting $v = (ca, c(-b))$ we obtain $vw = (ca, -cb)(a, b) = (ca^2 + cb^2, -cba + cba) = (c(a^2 + b^2), 0) = (1, 0)$ and hence v is the required multiplicative inverse. □

Proof of implications B and C:
Suppose the relevant conditions hold, and that $z * w = 0$ with $z = (a, b)$ and $w = (c, d)$. Then we have

$$\left.\begin{aligned} ac - bd &= 0 \\ ad + bc &= 0 \end{aligned}\right\} \qquad (*)$$

If $a = 0$ and $b \neq 0$, then $bd = 0$ and $bc = 0$, giving $d = c = 0$ and hence $w = (0, 0) = 0$. Similar arguments hold whenever b, c, or d are zero, and the implications are proved for all these cases. If a, b, c, and d are all nonzero then, by equations $(*)$, $ab(ac) = ab(bd)$, or $a^2(bc) = (-bc)b^2$ after substituting for ad. Hence $a^2 = -b^2$ holds for non-zero a and b, immediately contradicting type condition 1 for implication B. When b has the multiplicative inverse c, we have that $a^2 = -b^2$ gives $(ac)^2 = -bcbc = -1$, contradicting type condition 2 for implication C. Hence the result is proved for both implications □

The Aldor library developer, by using this specification, can check the conditions for the particular domain of computation under consideration. For example, neither type condition holds in `PrimeField 5`, so `Complex PrimeField 5` will have type `CommutativeRing` (justified by the implications of the specification of `ComplexCategory`) but not type `Field`. Conversely, since type condition 1 holds in the type `Integer`, `Complex Integer` can correctly be assigned type `IntegralDomain`, with implication C above as formal justification.

```
Complex (CR) : trait
assumes CommRingCat(CR)
includes ComplexCat(CR), TypeConditions (CR,T)
 BiRecord(T, CR, CR, .real for .first, .imag for .second)
introduces
 coerce : CR → T
 __*__  : N,T → T
 isZero, isOne : T → Bool
asserts ∀ x,y : CR, z : T, n : N
 complex(x,y) == [x,y];
 coerce(x) == [x,0];
 n*z == [n*(z.real), n*(z.imag)];
 isZero(z) == z = 0;
 isOne(z) == z = 1
implies
 RequirementsForComplex(CR, __.real for real, __.imag for imag)
 ∀ z, w : T
   norm(z*w) == real((z*w)*conjugate(z*w));
   imag((z*w)*conjugate(z*w)) == 0;
   conjugate(z)*conjugate(w) == conjugate(z*w)
 converts complex
```

We now wish to show that the record representation for complex numbers used by AXIOM satisfies our high level requirements. The trait `Complex(CR)` above is simply a copy of the AXIOM documentation with the element $x + iy$ represented by the record `[x,y]`. By implying `RequirementsForComplex` we generate the required proof goal. The proof (although straightforward in LP) is not trivial: we have included the specification of `ComplexCategory`, which itself includes `RequirementsForComplex`, but not under the renaming of operators given in the implies clause. Hence we are checking that the record representation is suitable, where suitability was defined in the trait `RequirementsForComplex`. The same methodology would be used to show that a representation of $x + iy$ as (r, θ) (i.e. the standard modulus/amplitude representation) satisfied our abstract requirements. The remaining implications check that the combined operator definitions satisfy standard results from the abstract theory of complex numbers.

4.3 The Interface Specification

In the previous section we described how the AXIOM functor `Complex(CR)` allows the user to construct an object which AXIOM considers to be a field even though it is not. Here we show how interface specifications may be used to deal with the problem in a different yet complementary way to that adopted in the previous section. Since functors are functions from types to types, it is quite natural to use interface specifications such as those described earlier to describe their behaviour. In general a functor will not make any modifications to client-visible state which simplifies any reasoning about them. However, since the arguments and return values are types we may need to resort to a higher

order logic to capture their meaning. This is not always the case as can be seen here. In the example below we present the skeleton of a Larch/Aldor program which describes the `Complex(CR)` domain.

```
++} requires isIntegralDomain(CR) ∧ ¬(∃ x,y:CR • (x*x + y*y = 0));
++} ensures  isIntegralDomain(%);
++} modifies nothing;
Complex(CR:CommutativeRing):CommutativeRing;
```

The predicate `isIntegralDomain(CR)` in the pre-condition corresponds to a trait in our LSL theory and is true iff the domain `CR` satisfies the properties of a mathematical integral domain; the statement $\neg(\exists x, y : CR \bullet (x^2 + y^2 = 0))$ is intended to capture the notion of type correctness described in the previous section. In the post-condition the concrete instance of `Complex(CR)` is represented by the AXIOM symbol `%`.

If the user instantiates the domain `Complex(Integer)` we can generate the verification condition

$$\texttt{isIntegralDomain}(\texttt{Integer}) \wedge \neg\exists x, y : \texttt{Integer} \bullet (x^2 + y^2 = 0)$$

Since `Integer` is an integral domain `isIntegralDomain(Integer)` holds; in fact the interface specification for `Integer` will state this property as part of its post-condition. This means that the VC can be simplified to

$$\neg\exists x, y : \texttt{Integer} \bullet (x^2 + y^2 = 0)$$

and if the user is familiar with elementary mathematics, they will be able to show that this is true. In doing so they will hopefully gain confidence that the `Complex(Integer)` domain will behave in the way that they expect it to. In addition to the verification condition we infer from the post-condition that

$$\texttt{isIntegralDomain}(\texttt{Complex}(\texttt{Integer}))$$

and as mentioned earlier, this may help to discharge other VC's.

If we repeat the process with `Complex(PrimeField 5)` (which AXIOM considers to be valid even though it isn't an integral domain) we obtain a similar VC to the one above

$$\neg\exists x, y : \texttt{PrimeField 5} \bullet (x^2 + y^2 = 0)$$

since `PrimeField 5` is a finite integral domain (and hence a field). However, this VC can be shown to be false by providing the witnesses $x = 2$ and $y = 4$.

5 Conclusions and Future Work

We have augmented our specification of the AXIOM algebraic category hierarchy with LSL specifications of AXIOM functors. The methodology used allows

enhanced type-checking and verification of argument types, as well as proofs of operator properties with respect to value representations. We have implemented a prototype lightweight verification condition generator in Aldor for Larch/Aldor programs. To achieve this the grammar of an Aldor compiler was extended to allow Larch annotations to be recognised. Further modifications to the compiler were made so that it could generate an external representation of the parse tree complete with types and specifications. The prototype analyser uses the parse tree to generate verification conditions and inferences from the user's program. For example, given the annotated Aldor program in Section 2.4 and the program statement "`ans := integerDivide(23, 6)`" our tool could, in principle, produce the VC $\neg(6 = 0)$ which is obviously true and the inference that:

$$(23 = ((ans.q) * 6 + ans.r)) \wedge (abs(ans.r) < abs(6))$$

The prototype VC generator is by no means completely finished and there is scope for further improvement. Indeed it would be interesting to incorporate it into the compiler itself so that existing control-flow functions and data-structures could be utilised, and so that VC generation could be used to provide additional compiler warnings. In spite of its limitations the authors feel that the prototype is useful as a proof-of-concept and given time it could be extended to analyse functions and domains as well. At present the LP proof assistant is more than capable of discharging the simple verification conditions that it has generated so far. However, we believe that more interesting case studies will probably require the use of a more developed theorem prover such as HOL [14] or PVS [25].

5.1 Related Work

There are a number of other systems which are related to our work and from which we have drawn upon for our ideas. Examples of ways in which CAS and automated theorem proving technology have been used together include work linking HOL and Maple [1] where simplification rules were added to HOL to make selected Maple routines available; the Analytica system which implements automated reasoning techniques in the Mathematica CAS [2]; the Theorema project uses the rewriting engine of Mathematica as a logical system to provide a single framework for both symbolic computation and proof [3]; REDLOG is an extension of the REDUCE to allow symbolic manipulation of first order formulas in a CAS [8]. These approaches differ in the amount of trust given to CAS and ATP results, their overall goals (better ATP, better CAS, or possibly better formalised mathematics), and in the hierarchy of the systems (for example ATP slave to the CAS master or *vice versa*).

Closer to our work is that of [26] where the Aldor type system is being extended to increase the potential of its dependent types. This work can be used to incorporate pre- and post-conditions into type declarations and admit proofs that properties in the documentation also hold at the computational level.

On the Larch side of our work (see Section 2) we are aware that many of the Larch behavioural interface specification languages (BISL's) do not have

any program analysis tools associated with them—they are primarily used as clear and concise documentation. One exception is Larch/Ada [15] which uses a syntax-directed editor called Penelope [15] for the interactive development and verification of Larch/Ada programs. Another exception is Larch/C [10] for which the LcLint [10] static program checker has been written. This tool is able to detect violations of subset of Larch/C interface specifications and check other special program annotations. Also in the Larch world, Speckle [29] is an optimising compiler for the CLU language which uses Larch-style interface specifications to select specialised procedure implementations.

The Extended Static Checking (ESC) system [6] provides automatic machine checking of Modula-3 programs to detect violations of array bounds, `NIL` pointer dereferencing, deadlocks and race conditions through the use of simple yet powerful annotations. ProofPower is a commercial tool developed by the High Assurance Team at ICL [22] based on the HOL theorem prover and the Z notation for a subset of Ada. Programs are prototyped and refined using Compliance Notation into Ada. Verification conditions generated from the Compliance Notation can be discharged via formal or informal arguments as required.

Also of note are the Eiffel [24] and Extended ML [28] programming languages. In Eiffel pre- and post-conditions are an integral part of the language syntax. These annotations can be converted into runtime checks by the compiler and violations may be handled by the programmer via exception handlers. Extended ML also incorporates specifications into its syntax—users can write algebraic specifications describing the properties of functions and use stepwise refinement (*c.f.* reification [20]) to obtain suitable implementations.

Acknowledgements

We acknowledge support of the UK EPSRC under grant number GR/L48256 and of NAG Ltd. We also thank James Davenport of the University of Bath and Mike Dewar from NAG for their interest and suggestions.

References

[1] Ballarin, C., Homann, K., and Calmet, J. Theorems and algorithms: An interface between Isabelle and Maple. In *Proceedings of International Symposium on Symbolic and Algebraic Computation* (1995), A.H.M.Levelt, Ed., ACM Press, pp. 150–157.

[2] Bauer, A., Clarke, E., and Zhao, X. Analytica—an experiment in combining theorem proving and symbolic computation. *J. Automat. Reason. 21*, 3 (1998), 295–325.

[3] Buchberger, B. Symbolic computation: computer algebra and logic. In *Frontiers of combining systems (Munich, 1996)*. Kluwer Acad. Publ., Dordrecht, 1996, pp. 193–219.

[4] Char, B. W. *Maple V language Reference Manual*. Springer-Verlag, 1991.

[5] Cheon, Y., and Leavens, G. T. A gentle introduction to Larch/Smalltalk specification browsers. Tech. Rep. TR 94-01, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011-1040, USA, Jan. 1994.

[6] Detlefs, D. L. An overview of the Extended Static Checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice* (Jan 1996), ACM (SIGSOFT), pp. 1–9.

[7] Dingle, A., and Fateman, R. J. Branch cuts in computer algebra. In *Symbolic and Algebraic Computation* (1994), ISSAC, ACM Press.

[8] Dolzmann, A., and Sturm, T. REDLOG: Computer algebra meets computer logic. *ACM SIGSAM Bulletin 31*, 2 (June 1997), 2–9.

[9] Dunstan, M., Kelsey, T., Linton, S., and Martin, U. Lightweight formal methods for computer algebra systems. In *ISSAC* (1998).

[10] Evans, D. Using specifications to check source code. Master's thesis, Department of Electrical Engineering and Computer Science, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, June 1994.

[11] Feit, W., and Thompson, J. G. Solvability of groups of odd order. *Pacific Journal of Mathematics 13* (1963), 775–1029.

[12] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4*. Aachen, St Andrews, 1998. (`http://www-gap.dcs.st-and.ac.uk/~gap`).

[13] Gordon, M. J. C. *Programming language theory and its implementation*. Series in Computer Science. Prentice Hall International, 1988.

[14] Gordon, M. J. C., and Melham, T. F., Eds. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993. A theorem proving environment for higher order logic, Appendix B by R. J. Boulton.

[15] Guaspari, D., Marceau, C., and Polak, W. Formal verification of Ada programs. In *First International Workshop on Larch* (July 1992), U. Martin and J. Wing, Eds., Springer-Verlag, pp. 104–141.

[16] Guttag, J. V., and Horning, J. J. *Larch: Languages and Tools for Formal Specification*, first ed. Texts and Monograps in Computer Science. Springer-Verlag, 1993.

[17] Harrison, J., and Théry, L. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In *Higher order logic theorem proving and its applications (Vancouver, BC, 1993)*. Springer, Berlin, 1994, pp. 174–184.

[18] Jackson, P. *Enhancing the NUPRL Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, Apr. 1995.

[19] Jenks, R. D., and Sutor, R. S. *AXIOM*. Numerical Algorithms Group Ltd., Oxford, 1992. The scientific computation system, With a foreword by David V. Chudnovsky and Gregory V. Chudnovsky.

[20] Jones, C. B. *Systematic Software Development using VDM*, second ed. Computer Science. Prentice Hall International, 1990.

[21] Jones, K. D. LM3: a Larch interface language for Modula-3, a definition and introduction. Tech. Rep. 72, SRC, Digital Equipment Corporation, Palo Alto, California, June 1991.

[22] King, D. J., and Arthan, R. D. Development of practical verification tools. *The ICL Systems Journal 1* (May 1996).

[23] Leavens, G. T., and Cheon, Y. Preliminary design of Larch/C++. In *First International Workshop on Larch* (July 1992), U. Martin and J. M. Wing, Eds., Workshops in Computing, Springer-Verlag, pp. 159–184.

[24] Meyer, B. *Object-Oriented Software Construction*. Computer Science. Prentice Hall International, 1988.

[25] Owre, S., Shankar, N., and Rushby, J. M. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[26] Poll, E., and Thompson, S. Adding the axioms to Axiom: Towards a system of automated reasoning in aldor. Technical Report 6-98, Computing Laboratory, University of Kent, May 1998.

[27] Potter, B., Sinclair, J., and Till, D. *An introduction to formal specification and Z*. Prentice Hall International, 1991.

[28] Sannella, D. Formal program development in Extended ML for the working programmer. In *Proceedings of the 3rd BCS/FACS Workshop on Refinement* (1990), Springer Workshops in Computing, pp. 99–130.

[29] Vandevoorde, M. T., and Guttag, J. V. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *Proceedings of the 1994 ACM/SIGSOFT Foundations of Software Engineering Conference* (1994).

[30] Wing, J. M. A two-tiered approach to specifying programs. Tech. Rep. LCS/TR–299, Laboratory for Computer Science, MIT, May 1983.

[31] Wing, J. M., Rollins, E., and Zaremski, A. M. Thoughts on a Larch/ML and a new application for TP. In *First International Workshop on Larch* (July 1992), U. Martin and J. M. Wing, Eds., Workshops in Computing, Springer-Verlag, pp. 297–312.

[32] Wolfram, S. *Mathematica: A system for doing mathematics by computer*, 2 ed. Addison Wesley, 1991.