A System For Axiomatic Programming

Gabriel Dos Reis

Texas A&M University
`gdr@cse.tamu.edu`

Abstract. We present the design and implementation of a system for axiomatic programming, and its application to mathematical software construction. Key novelties include a direct support for user-defined axioms establishing local equalities between types, and overload resolution based on equational theories and user-defined local axioms. We illustrate uses of axioms, and their organization into *concepts*, in structured generic programming as practiced in computational mathematical systems.

1 Introduction

We want our programs to look as much as possible like the mathematical formulation of the algorithms they implement. If the current state of our mathematical software is of any indication, an abstraction gap between a published algorithm and its realization as computer program grows to the point where we have trouble convincing ourselves that a program really does what it is supposed to. Of course, this situation is not unique to computer algebra. It is a general problem in the software industry. However, computer algebra deals with entities with rich mathematical structures. Therefore, it is a natural place to try to understand and to attempt to solve the abstraction gap problem.

The weak programming language support for direct expression of mathematical algorithms as computer codes has baffling consequences, and may at times prove to render some of our computer algebra systems quite embarrassing tools for education. For example, in the AXIOM computer algebra system [13] (and its derivatives), a domain that satisfies the `AbelianMonoid` category does not necessarily satisfy those of the `Monoid` category. Yet, we don't think a freshman student could easily get away with pretending that an Abelian monoid is not a monoid. The problem has been known for quite a long period of time. It has several root causes. The most fundamental one was the lack of programming language features to express concise mathematical properties of operations in ways that could be effectively used by compilers. In the case of AXIOM, the notion of *axiom* was considered in the early 1980s but never implemented [2]. With the lack of linguistic features to directly express that some operations obey certain laws, programmers have developed elaborate schemes, often culminating with conferring to names — *i.e.* syntax — far more importance than semantics, at the very expense of mathematical correctness. As a result, we have seen proposals (and software) that use `AdditiveMonoid` to represent the mathematical idea of Abelian monoid, where the operation is decided once for all to be named +, and the name `MultiplicativeMonoid` to designate a monoid that is not necessarily known to be Abelian, where its operation is decided (again) once for all to be named *. As to be expected, one runs into serious problems when the same datatype carries simultaneously several instances of the same abstract structure. For example, the set of integers is obviously an Abelian monoid with respect to both usual + and *, and we do want our software to reflect the fact that the operation * is commutative. Furthermore, those are not the only Abelian monoid operations on integers; $\gcd$ is another example; so is the operation φ defined by

$$\varphi(x, y) = x + y + xy.$$

In short, the workaround that consists of conferring more importance to syntax than semantics is not just conceptually incorrect: it does not scale.

The main contribution of this paper is the design and implementation of a system (named Liz) that directly supports axioms, and structured generic programming as practiced in computer algebra. The system is an extension of a subset of the popular programming language C++ [21, 12]. The long term goal is to bring structured generic programming to mainstream and accessible to "ordinary programmers". This can be achieved if we have enough automation. The extensions are "*concepts*" [7, 10] and "*axioms*" [9]. A key design concern has been to reduce the amount of "boilerplate" that programmers have to write in order for the compiler to "understand" that their programs have structures. The core of the type system is based on equational reasoning and axioms introduced by users. A central problem is: How can the compiler use user-defined constraint to resolve calls to overloaded functions? Overall, the Liz system offers a remarkable application of symbolic mathematics and deduction techniques [27] to the field of programming language design and implementation. Liz's type checker combines pattern matching and automated logical deduction.

The remaining of this paper is structured as follows. We introduce axiomatic programming and the key language features behind Liz in (Section 2.) The general architecture, and the problems of Type checking, code generation, and constraint satisfaction are discussed in Section 3. Finally we discuss related work and we conclude in Section 4.

2 Axiomatic Programming

We define *axiomatic programming* as the practice of structured generic programming [4, 3, 15, 16, 5, 19] that expresses and uses axioms or mathematical properties directly in code. The idea has been recently illustrated by Stepanov and McJones [19]. They show-cased programming as a mathematical activity, a wonderful journey in the land of simplicity and generality. Their approach and exposition makes essential use of *properties* and *concepts*. In spite of this, all of their codes is compilable as almost C++03 [12] program fragments. There are two reasons for this. First, although the exposition is centered around concepts, the book never actually presents any single concept in concrete code — concepts were defined as abstract mathematical entities. From the C++ compiler point of view, concepts — the propellers — are effectively equivalent to comments, *i.e.* white spaces. Second, the actual computer codes use a few simple macros. In particular, the `requires` "keyword" is actually a C99 variadic macro defined [19, Appendix B.2] that ignores its arguments. Consequently, one of the benefits of concepts

— turning informal descriptions into codes that can be verified and used for type checking template definitions and uses — is not realized. The next subsections introduce the programming system Liz designed to support axiomatic programming.

2.1 Axioms and Concepts

In Liz, an axiom is a property that states what an algorithm may assume of values, operations, etc. but also what a user should and should not assume about these entities. Those properties are logical expressions about semantics we assume to hold for a proper execution of an algorithm. As such, what we call an axiom in Liz corresponds to the usual mathematical practice of "hypothesis forming" in developing a theorem. For example the notion of associativity is expressed in Liz as

```
template<BinaryOperation Op>
   axiom associative(Op op) {
      forall(Domain(Op) a, Domain(Op) b, Domain(Op) c)
         op(op(a, b), c) == op(a, op(a, b));
   }
```

That is, the axiom `associative` can be applied to a specific binary operation — not all binary operations are associative. The `template` header is used here to introduce the type `Op` of the parameter `op`. It also indicates that when used as in

```
associative(gcd);
```

Liz should magically figure out the value for `Op` — assuming `gcd` is not overloaded. The "type" `BinaryOperation` of the template parameter `Op` is a concept.

A concept [7, 19] is a collection of properties (existence of operations, values, semantics, etc.) about types, operations, and values. For example the notion of a magma operation is rendered in Liz as the concept `BinaryOperation`. In general, concepts are organized in hierarchies. A magma operation is a binary homogeneous function, *i.e.* a function whose argument types are identical. In Liz, that concept is expressed as

```
concept HomogenousFunction(Function F) {
   Arity(F) > 0;
   forall(int i, int j) i < Arity(F) and j < Arity(F) =>
      InputType(F, i) == InputType(F, j);
}
```

which says that a function type `F` satisfies the requirements of `HomogenousFunction` if and only if, it is of positive arity (*i.e.* functions of that type take at least one argument), and all parameter types are identical. The function `InputType` is a builtin binary function that returns the argument type of a function. It is an example of what we call *type function*, *i.e.* type producing functions. Programmers can define their own type functions, as we will shortly see. Notice that we use a logical formula to state that property at a sufficiently abstract level. The homogeneity property implies that it makes sense to define the following function

```
typename Domain(HomogenousFunction F) {
   return InputType(F, 0);
}
```

Here, `Domain` is a function that accepts any homogenous function type argument and returns the common argument type. Notice the use of `typename` as return type. Indeed, `typename` is a concept that designates the collection of all types in Liz. Type functions can be evaluated at both compile and run time.

We refine further the notion of homogenous function to that of operation, where the domain is the same as the target

```
concept Operation(HomogenousFunction Op) {
   Codomain(Op) == Domain(Op);
}
```

Here, we use a builtin type function, `Codomain`, to specify that constraint on the return type. Again, so far the expressions of the concepts are very close to their abstract mathematical statements. Finally, we define a magma operation as an operation of arity 2.

```
concept BinaryOperation(Operation Op) {
   Arity(Op) == 2;
}
```

As an illustration of use, here is how in Liz one defines a function that applies a binary operation to the same argument

```
template<BinaryOperation Op>
  Domain(Op) square(const Domain(Op)& x, Op op) {
     return op(x, x);
  }
```

As defined, the `square` function can be applied to any binary Liz function like in

```
int mult(int x, int y { return x * y; }
int i = square(4, mult);
```

but also to any binary function object like in

```
int j = square(3, plus<int>());
```

where `plus<int>` is the equivalent of C++ standard function object of the same name. This versatility is possible precisely because the specification of `BinaryOperation` is abstract enough to allow several concrete representations of operations to be used. Yet, it is also precise enough to allow for complete type checking at the definition site — unlike the case of traditional C++ templates.

2.2 Associated types and values

Algebraic structures are usually determined by fundamental operations and derived or *associated values and types*. For instance, a monoid structure is uniquely defined by a monoid operation. In turn the neutral element of a monoid structure is uniquely associated with the monoid operation, *i.e.* the neutral value of a monoid is a function of the monoid operation. The notion of neutral value can be generally defined for any binary operation:

```
template<BinaryOperation Op>
  axiom has_neutral(Op op, Domain(Op) e) {
     forall(Domain(Op) x)
       op(x, e) == x and op(e, x) == x;
  }
```

From there, the notion of monoid operation comes as:

```
concept MonoidOperation(BinaryOperation Op) {
  associative(op);
  exist(Domain(op) e) {
    has_neutral(op, e);
    using neutral_value = e;   // name it from now on
  }
}
```

Here, we express a monoid operation as a binary operation that is associative and that happens to admit a left- and rigt-neutral value. Furthermore, the `MonoidOperation` definition introduces the name `neutral_value` as an alias for that neutral value. It is clear that alias *depends* on the operation `op`, as indicated by existential quantification. Associated values and types can be referred to outside concepts definitions using a functional notation. The Liz compiler internally uses Skolemization to process associated entities. For example, after the statement

```
assume has_neutral(gcd, 0);
```

the Liz compiler can determine that the operation `gcd` satisfies the `MonoidOperation` concept and that `neutral_value(gcd)` is `0`.

3 Implementation

Given the abstract level of specification of operations in Liz, it is natural to expect the implementation to depart from conventional type checking. Below, we describe the challenges we face and solutions currently implemented in Liz. Despite being based on C++, Liz's template system behaves quite differently from standard C++ templates. Indeed, in C++ the following function template

```
template<typename T>
  T twice(T x) {
      return x + x;
  }
```

is perfectly fine, even though the compiler cannot determine all possible `operator+` that could be used. That decision is deferred until the function template `twice` is instantiated.

In Liz, that template definition will not type check. The elaborator objects with:

```
no match for operation ‘operator+’ with argument
     type list (T, T)
candidates are
   operator+: (int, int) -> int
   operator+: (double, double) -> double
```

This difference is deliberate. One of our goal is to understand what it takes to support axiomatic programming with C++-like template-like system with early checking by default.

3.1 Structure of the Liz system

The Liz system is designed with careful phase distinction considerations in mind. However, it is currently implemented as an interactive read-eval-print system. An input program fragment is decomposed into a token stream by a lexer. A parser arranges the token stream into a sequence of AST objects following the Liz grammar. Then, an elaborator takes the AST object sequence and type checks it, generating an alternative representation of the input program in a much simpler expression-based intermediate language. The intermediate expression language is designed in such a way that the evaluator does not require type information, *i.e.* the elaboration phase implements a type-erasure semantics. This design choice reflects our desire to ultimately reflect core C++ semantics and efficiency. Liz deliberately blurs the syntactic distinction between "type expression" and "ordinary expression". This stems from the fact that Liz supports *type function*, e.g. functions (such as `Domain` from Section 2.1) which when applied to arguments produce types. Consequently, while the elaborator can most of the time determine through type checking that an expression designates a type, it occasionally needs some form of evaluation to reduce type function calls. Finally, the main feature of Liz — *axioms* — requires compile-time expression evaluation (involving function calls) to determine whether a combination of types and values satisfy certain logical formulæ. Compile-time evaluation of calls to user-defined functions with constant expressions is now part of standard C++ [8].

3.2 Type checking

The main features of the Liz system are axioms and concepts. And that is the focus of this section.

Intermediate language The elaborator type checks and translate the input source program into an internal intermediate language. We briefly describe that intermediate language here for the benefit of the following subsections. First, it should be noted that the intermediate language is an expression-based language. This means that there is no arbitrary syntactic distinction between statements, expressions, or definitions. Every expression produces a value of one sort or the other.

Second, the intermediate language does not involve any notion of overloading. However, since Liz programs can overload functions, we must represent overloaded symbols in some way. We do this by pairing every symbol with its type (as computed by the elaborator).

Third, we present the syntax of intermediate language as s-expressions to ease visualization. Note, however, that Liz is itself implemented in ISO C++. The building blocks are:

– (`@formal` $p\,l\,n$) represents a formal parameter at position p, nesting level l, and named n
– (`@symbol` $n\,t$) represents a reference to symbol named n, and with declared type t
– (`@read` a) represents a read operation from a location designated by a
– (`@write` $a\,v$) represents a write operation to location a with value v
– (`@unary` $f\,x$) represents the call of a builtin unary operation f with argument x
– (`@binary` $f\,x\,y$) represents the call of a builtin binary operation f with arguments x and y
– (`and` $x\,y$) represents a conjunction logical formula
– (`or` $x\,y$) represents a disjunction logical formula
– (`=>` $x\,y$) represents an implication logical formula
– (`@call` $f\,x_0\,\ldots\,x_{n-1}$) represents the call of a user defined operation f with arguments $x_0, \ldots, x_{n-1}$
– (`@if` $p\,x\,y$) represents an if-statement with condition p, and branches x and y
– (`@while` $p\,x$) represents an do-statement with condition p, and body x
– (`@return` v) represents a return-statement with value v
– (`@block` $x_0, \ldots\, x_{n-1}$) represents a block composed of the statement sequence x_0, $\ldots, x_{n-1}$
– (`@bind` $n\,t$) represents a symbol n with type t in the current frame.
– (`@forall` (`@parameters` $p_0\,\ldots\,p_{n-1}$) x) represents a universally quantified logical formula x, and that binds the parameters $p_0, \ldots, p_{n-1}$

In addition to these forms, there are are representations for basic type, basic constants. Unary builtin and binary builtin operations are represented as

– (`@builtin` $n\,t$) where n is the name of the operation and t is its (function) type

Finally, because the implementation of the intermediate language is strongly typed, we use the form

– (`@type_expr` e) for expressions that represent types; this is typically the case when a type function is applied to archetypes (Section 3.2)

Elaborating axioms Axioms are first-order predicate formulæ. They can make references to any user-defined function. Before type checking and code generation, an axiom is first put in prenex form. The body is then elaborated as a Boolean expression, where names bound by the quantifiers prefix are treated as if they where formal (function) parameters with the declared types. For example, the following axiom taken from the definition of `HomogenousFunction`

```
forall(int i, int j) i < Arity(F) and j < Arity(F) =>
   InputType(F, i) == InputType(F, j);
```

is elaborated as

```
(@forall (@parameters (@formal 0 1 i) (@formal 1 1 j))
   (=> (and (@binary
               (@builtin operator< (int, int) -> bool)
```

```
                (@formal 0 1 i)
                (@unary (@builtin Arity (Function) -> int)
                   (@type_expr (@formal 0 0 F))))
             (@binary
                (@builtin operator< (int, int) -> bool)
                (@formal 1 1 j)
                (@unary (@builtin Arity (Function) -> int)
                   (@type_expr (@formal 0 0 F)))))
      (@binary
        (@builtin operator== (typename, typename) -> bool)
        (@type_expr
           (@binary
              (@builtin InputType (Function, int) -> typename)
              (@type_expr (@formal 0 0 F))
              (@formal 0 1 i)))
        (@type_expr
           (@binary
              (@builtin InputType (Function, int) -> typename)
              (@type_expr (@formal 0 0 F))
              (@formal 1 1 j))))))
```

Notice that all overloaded operators have been resolved. It is this elaboration that is stored for future uses, in particular in deciding constraints satisfaction (see Section 3.2.)

Concept elaboration Recall that a concept is a collection of syntactic, semantics, and complexity requirements on a collection of operations, types, and values. In Liz, concepts are expressed as a sequence of axioms, refinements, and operation signature specifications. The result of elaborating a concept is a 5-tuple:

1. a concept name
2. a sequence of elaboration of parameters explicitly bound in the concept definition — we refer to these as explicit concept parameters
3. a sequence of elaborations of formulæ mentioned in the concept definition
4. a sequence of elaborations of refined concepts
5. a sequence of elaborations of signatures explicitly mentioned in the concept definition — we refer to these operations as implicit concept parameters.

An explicit concept parameter is elaborated just like any other kind of parameter. If the type mentioned in the parameter declaration is a (unary) concept, then the elaboration goes through an additional step called *dressing* to create an archetype as will be explained in Section 3.2.

Refined concepts are elaborated as predicates, with the understanding that they are properties that a function template definition can assume and use, while they acts as preconditions at a function template call site. These refined concepts are also used during dressing of archetypes.

An operation signature explicitly mentioned in a concept definition is to be thought of as a requirement, a proof obligation to be fulfilled at the point of the concept use (through function template call.) Consequently, they are handled as parameters — except that their values are implicitly deduced during constraint satisfaction.

The collection of explicit concept parameters and implicit concept parameters form the domain of the substitution that results from a successful constraint satisfaction checking. That substitution is then used to expand or instantiate the elaboration of the selected function template (see Section 3.3.)

Archetypes and dressing Conventional type checking assumes that types are values that are known at compile-time. To deal with type variables (used by generic functions), one synthesizes an arbitrary value for the type variable, and type checking proceeds as usual. That arbitrary value is what we call *archetype* of a type parameter. It symbolizes any value that parameter may take on when the generic function is instantiated. At the basic level, the archetype does not have any properties, except that it is a type value. To be useful in generic algorithms, the archetype `T` needs to carry information useful for type-checking purposes. The process of endowing an archetype with additional assumptions is what we call *dressing*.

Dressing of an archetype `T` with a concept type $\mathcal{C}$ is given by the following algorithm:

1. for each logical formula f in $\mathcal{C}$, simplify the instantiation of f, based on the existing property set of `T`. If the result is not vacuously true, add it to the property set of `T`.
2. for each refined concept $\mathcal{C}'$ in $\mathcal{C}$, dress the archetype `T` with $\mathcal{C}'$.

Note that in abstract, the order in which properties are added to the property set of an archetype does not matter. However, from a practical point of view, we do want to keep property sets as small as possible for reasons that will become obvious by the end of the type equivalence subsection. We observe that in a concept hierarchy, a refining concept usually adds more information that restricts the collection of satisfying types. In particular, formulæ from refined concepts may get simpler with the addition of new constraints. We can see this with our `HomogenousFunction` example. With that concept, all we know is that the arity of any function type `F` that satisfies `HomogenousFunction` must be a positive integer. The definition concept of `BinaryOperation` gives a definite value to the arity. Consequently, the dressing of an archetype `Op` of `BinaryOperation` produces the following trace of its property set:

1. Start with $P_0 = \{$`Arity(Op) == 2`$\}$, which is the sole formula in `HomogenousFunction`
2. Dress `Op` with `Operation`
   (a) simplify the formula
   `Codomain(Op) == Domain(Op)`
   with P_0. This involves reducing each side of the equality operator in irreducible form. In particular the type function call `Domain(Op)` is reduced to `InputType(Op,0)`. There is no other formula in P_0 that would reduce the formula.
   (b) Add `Codomain(Op) == InputType(Op,0)` to P_0 to obtain the new property set
$$P_1 = \left\{ \begin{array}{l} \texttt{Arity(Op) == 2,} \\ \texttt{Codomain(Op) == InputType(Op,0)} \end{array} \right\}$$
3. Dress `Op` with `HomogenousFunction`

(a) simplify

```
forall(int i, int j)
   i < Arity(Op) and j < Arity(Op) =>
      InputType(Op,i) == InputType(Op,j)
```

with the property set P_1 to obtain

```
forall(int i, int j)
   i < 2 and j < 2 =>
      InputType(Op,i) == InputType(Op,j)
```

Note that although we show the input-source form above, the simplification is really done on the *elaboration* of the formula. The simplification is implemented a typefull term rewrite engine.

(b) Add the resulting formula to P_1 to obtain

$$P_2 = \left\{ \begin{array}{l} \texttt{Arity(Op) == 2,} \\ \texttt{Codomain(Op) == InputType(Op,0),} \\ \texttt{forall(int i, int j)} \\ \quad \texttt{i < 2 and j < 2 =>} \\ \quad\quad \texttt{InputType(Op,i) == InputType(Op,j)} \end{array} \right\}$$

(c) simplify the formula `Arity(2) > 0` with respect to the property set P_2 to obtain $2 > 0$, which is vacuously true. So the final property set of the archetype `Op` is P_2.

4. Dress `Op` with the builtin concept `Function`, which does not actually add any formula.

During the dressing procedure we interpret an equality formula `a == b` as defining the expression `a` in terms of the expression `b`, even though the equality operator is in fact commutative. We could avoid this restriction by using unification algorithms that cope with commutativity. That is an improvement we consider as future work.

Type equivalence During type checking we need to determine when an expression of type `T` is acceptable in a context where a value of type `S` is expected. Conventional type checking solves this essentially as an equality problem in the free algebra generated by base types and type constructors.

That approach is inadequate for axiomatic programming. In fact, the defining trait of this style of programming is precisely that programmers can state in very abstract, if somewhat stylized fashion, relations between types. Consequently determining the equivalence of two type expressions amounts to determining identities in an equational theory. The set of equations to consider varies from one context to the next, depending on the set of assumptions in scope. Consider the `square` function example

```
template<BinaryOperation Op>
  Domain(Op) square(const Domain(Op)& x, Op op) {
     return op(x, x);
  }
```

Here, to check that the call `op(x,x)` is well-formed, we need to check that `x` can be used as both the first and second argument to the operation `op`. First, we discuss the type of `op`, then we move on the matching of arguments.

When the type checker sees the call `op(x,x)`, it first tries to determine that `op` is an expression that can be used in an operator position. The answer is yes, because the archetype `Op` satisfies `Function`. Next, it needs to determines its arity. This information is found by pattern matching the expression `Arity(Op) == n` against the formulæ in the property set of `Op`. Which gives the value 2. The elaborator then synthesizes the type

```
(InputType(Op, 0), InputType(Op, 1)) -> InputType(Op, 0)
```

The return type is the result of rewriting `Codomain(Op)` with respect to the property set P_2. With this type for `op` the elaborator proceeds to check the arguments.

The type of the parameter `x` is `const Domain(Op)&`. This type expression involves a type function, `Domain`, which is defined only on types that satisfy `HomogenousFunction`. Since the archetype `Op` satisfies `BinaryOperation`, it also satisfies `HomogenousFunction`. Hence, evaluation of `Domain(Op)` is legitimate, and yields `InputType(Op,0)`. So the type of `x` is `const InputType(Op,0)&`.

Next, we need to determine if the use of `x` as first argument to `op` is well-formed. In general, the use of a reference in a non reference context implies a read operation. The read operation yields an expression of type `const InputType(Op,0)`. Finally, we observe that using a *value* of type `const T` in a context where a value of type `T` is expected is OK — in another term, it is fine to lose toplevel const-qualification on values.

We now need to determine whether the use of `x` as second argument of `op` is well-formed. We follow the same procedure as in the previous paragraph. Which leads to determining the equivalence `InputType(Op,0)` and `InputType(Op,1)`. At this point, we consider again the property set P_2. We obtain the equality we were looking for by considering the formula

```
forall(int i, int j)
  i < 2 and j < 2 =>
    InputType(Op,i) == InputType(Op,j)
```

and using standard deduction system based on sequent calculus. This deduction engine is part of the elaborator.

As can been seen from the example just discussed, every single type equivalence problem implies a search of a set of databases formed by property sets of relevant archetypes, intertwined with possible logical formulæ satisfiability. For this reason, it is beneficial to keep property sets sizes as small as possible.

Concept satisfaction The problem of concept satisfaction is a fruitful source of debates. Essentially, there are two schools of thoughts.

On the one hand, there is explicit conformance, that is the elaborator should consider that a type satisfies a concept only if there is an explicit statement to that effect — not just because some function declarations are in scope and predicates are satisfied. This is the approach implemented by the AXIOM system (and its variants including Aldor), and the Haskell programming language. It certainly is the favorite approach in certain type theory circles. A problem with this approach, in our opinion, is that it does not scale well in practice. One of the key aspects of C++ templates, that contributed to the success of the Standard Template Library [18], is the implicit matching of interfaces — or "duck typing" as it is called. Also, we believe that this approach has a deep justification rooted in the lack of language features to differentiate operations based on mathematical properties (or lack thereof.)

On the other hand, we have the notion of implicit conformance: a type satisfies a concept if it meets all its predicates, and all signatures have matching concrete function definitions. This is the approach we take for the Liz system. To determine that a type τ satisfies a concept $\mathcal{C}$, we use the following algorithm:

1. Instantiate $\mathcal{C}$ by substituting τ for its parameter. Simplify all logical formulæ. If any refined concept of $\mathcal{C}$ is not satisfied, then satisfaction of $\mathcal{C}$ fails.
2. For each signature specification in $\mathcal{C}$, try to find a matching declaration in scope. If the matching fails, or has more than one solution, satisfaction of $\mathcal{C}$ fails. This step finds values for the implicit parameters.
3. Substitute implicit parameters in logical formulæ in $\mathcal{C}$, if any of them is not satisfiable then satisfaction of $\mathcal{C}$ fails.

3.3 Code generation

There are several code generation techniques for handling generic functions. We briefly mention two, which are used in mainstream programming languages that support generic programming.

A common technique is to associate with each generic function a vector of used operations (or dictionary) that maps abstract operations to their concrete implementations. When a generic function is called, a dictionary argument is constructed and passed as an implicit additional argument. This technique is used in the implementation of the AXIOM system [13], in the implementation of Aldor, and is the conventional implementation [1, 11] of the Haskell programming language [17] for its type classes features [26]. This technique is very attractive in the sense that code generated for a program that uses a generic function contains only one definition or instance of that function, no matter how many times it is statically used. The technique also supports separate compilation. However, in practice it does bring a non-negligible abstraction penalty. This is because every (abstract) operation used in a generic function (no matter how cheap its concrete realization is) is looked up at runtime through the dictionary argument. It is obvious that the cost of this implementation technique is unacceptable for certain operations. There are a number of implementation tricks to reduce the dynamic lookup overhead. For example, if it is known that an abstract operation designates the same concrete realization during the execution of a generic function call, the result of the first lookup can be cached and reused during the lifetime of that particular call instance. But, it is equally clear that for a modular algorithm that operates directly on machine integers, the cost of adding (or multiplying) two machine integers becomes prohibitive if the integer operation is not directly inlined, resulting in simple machine code [7]. The Aldor programming language uses the notion of *domain inlining* to help programmers work around this efficiency issue. The idea is that a programmer, after careful analysis, would single out domains that are tightly coupled with generic function (mostly for efficiency reasons). The compiler will then resolve statically all calls to operations implemented by those domains, thereby bypassing the runtime dictionary lookup overhead. The technique is effective. However, separate compilation is lost because the resulting function now depends on implementation details of the domain. Moreover, we believe it requires a fair amount of foresight from programmers, especially library writers; and it is not clear how that approach scales when independently developed components (written by several independent groups of people) are composed.

The second code generation technique for generic function expands every generic function reference (but unique in its parameter types) into a new and distinct copy of the original generic function, where abstract parameters are replaced by their concrete values. The result is then subject to normal conventional code generation technologies. This technique is popular with C++ template implementations and some implementations of Ada generics [25]. Codes generated for generic functions by this approach are near optimal. However, it should be observed this technique may lead to code bloat in case of undisciplined programs that are not properly organized to take advantage of structural and semantics commonalities. In the case of C++ template, this technique can actually lead to code bloat removal, as surprising as it may come contrary to the conventional wisdom. The reason is very simple: a template function is instantiated if and only if it is used. That is, the C++ language type system contains explicit provision for "dead code" removal for generic function instances.

The second code generation technique just discussed is the approach we use in the Liz system. However it differs from conventional C++ template compilation in the sense that we expand the result of elaboration. That is, a function template definition is fully type checked, with corresponding generated code. It is the result of that elaboration that is expanded when the function is called. The concept system as currently designed and implemented ensures that no type error will occur at code expansion time. This is to be contrasted with what happens with current C++ templates. We achieve this result by rejecting some popular implicit conversions (mostly between basic types such has `bool`, `int`, `double`), which insures that expression types are preserved under substitution. We fully acknowledge that this restriction rules out many real world C++ programs, but the purpose of the Liz system is not to emulate anarchic type conversions [20].

4 Related work and conclusion

There is a large body of work in the general area of advanced language features for generic programming but, to our knowledge, none in major use today aims at direct support for axiomatic programming. Within the computer algebra community, Jenks and Trager [14] explained the design and implementation of the Scratchpad system, which later became AXIOM. The Aldor programming language (the better version of the AXIOM library extension language) aims at a more categorial view of programs. However, it does not have support for axiomatic constructions. With the C++ community, the most relevant work is the collaborative effort [7, 10] to introduce concepts into C++0x. While axioms were identified [6] early as key aspects of concepts, they are formally proposed only very late in the process [9]. At that point, the C++ concepts proposal was already exhibiting worrisome complexities that would prompt its eventual removal [22, 23] from the C++0x draft. Our opinion is that a good part of the complexity came from the fact that the proposal did not provide good enough support for more abstract definitions of algorithms. For a language as complex as C++ that has been in industrial use for nearly 3 decades, a successful proposal to support axiomatic programming must abstract over details, as opposed to aiming at reflecting them. Andrew Sutton and Bjarne Stroustrup recently begun investigation of semantics-oriented libraries for C++ [24]. The Liz system was designed to support more abstract generic algorithm definitions. It needs a terrain for experimentations, and computer algebra is a natural testbed — for it is hard to quibble with maths (semantics). Its implementation is a remarkable application of computer algebra and symbolic mathematics techniques.

5 Acknowledgment

I would like to thank Bjarne Stroustrup, Erik Katzen, Jasson Casey for comments on earlier drafts of this paper. Carla Villoria contributed to an early implementation of the Liz parser. This work was partially supported by NSF grants CCF-1035058 and CCF-1150055.

References

1. Lennart Augustsson. Implementing Haskell Overloading. In *Functional Programming Languages and Computer Architecture*, pages 65–73, 1993.
2. James H. Davenport. Private communication, May 2009.
3. James. H. Davenport, P. Gianni, and B. M. Trager. Scratchpad's View of Algebra II: A Categorical View of Factorization. In *ISSAC '91: Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation*, pages 32–38, New York, NY, USA, 1991. ACM Press.
4. James H. Davenport and Barry M. Trager. Scratchpad's View of Algebra I: Basic Commutative Algebra. In *DISCO '90: Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 40–54, London, UK, 1990. Springer-Verlag.
5. James C. Dehnert and Alexander Stepanov. Fundamentals of Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11, Schloss Dagstuhl, Germany, April 1998.
6. Gabriel Dos Reis. Generic Programming in C++: The next level. *The Association of C and C++ Users Spring Conference*, April 2002.
7. Gabriel Dos Reis and Bjarne Stroustrup. Specifying C++ Concepts. In *Conference Record of POPL '06: The 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, Charleston, South Carolina, USA, 2006.
8. Gabriel Dos Reis and Bjarne Stroustrup. General Constant Expressions for System Programming Languages. In *Proceedings of the 25th Symposium on Applied Computing*, pages 2133–2138, Sierre, Switzerland, March 2010. ACM Press.
9. Gabriel Dos Reis, Bjarne Stroustrup, and Alisdair Meredith. Axioms: Semantics Aspects of C++ Concepts. Technical Report N2887=09-0077, ISO/IEC SC22/JTC1/WG21, June 2009. `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2887.pdf`.

10. Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic Support for Generic Programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Languages, Systems, and Applications*, pages 291–310, New York, NY, USA, 2006. ACM Press.
11. Cordelia V. Hall, Kevin Hammond, Simon Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.
12. International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.
13. Richard D. Jenks and Robert S. Sutor. *AXIOM: The Scientific Computation System*. Springer-Verlag, 1992.
14. Richard D. Jenks and Barry M. Trager. A Language for Computational Algebra. *SIGPLAN Not.*, 16(11):22–29, 1981.
15. David A. Musser and Alexander A. Stepanov. Generic Programming. In *In proceeding of International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Rome, Italy, 1988.
16. David R. Musser and Alexander Stepanov. Algorithm-oriented Generic Libraries. *Software–Practice and Experience*, 24(7):623–642, July 1994.
17. Simon Peyton Jones. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
18. Alexander Stepanov and Meng Lee. The Standard Template Library. Technical Report N0482=94-0095, ISO/IEC SC22/JTC1/WG21, May 1994.
19. Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley, 2009.
20. Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
21. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.
22. Bjarne Stroustrup. Simplifying the use of concepts. Technical Report N2906, ISO/IEC SC22/JTC1/WG21, June 2009.
23. Bjarne Stroustrup. The C++0x "Remove Concepts" Decision. *Dr. Dobb's Journal*, 2009. `http://www.ddj.com/cpp/218600111?pgno=1` Republished with permission in Overload Journal, Vol 92. August 2009.
24. Andrew Sutton and Bjarne Stroustrup. Design of Concept Libraries for C++. In *International Conference on Software Language Engineering*. Springer, July 2011.
25. S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, and Erhard Ploederer, editors. *Consolidated Ada Reference Manual*, volume 2219 of *Lecture Notes in Computer Science*. Springer, 2000.
26. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 60–76, Austin, Texas, USA, 1989.
27. Stephen Watt. What Happened to Languages for Symbolic Mathematical Computation? In *Proceedings of Programming Languages for Mechanized Mathematics, (PLMMS)*, pages 81–90, Hagenberg, Austria, June 29-30 2007. RISC-Linz.