Generic Programming with Inferred Models

Richard C. Bilson Glen Ditchfield Peter A. Buhr
University of Waterloo, Canada
rcbilson@plg.uwaterloo.ca

Abstract

The effective design of software systems using generic programming can benefit greatly from appropriate programming language support. Essential support includes the direct representation of *concepts*, i.e., collections of type properties, and the ability to statically determine that concrete types model these concepts. This paper discusses the language C∀, which facilitates generic programming by supporting concepts explicitly and automatically inferring many cases where types model concepts. It also demonstrates the flexibility of C∀ in expressing relationships among types, especially so-called associated types, and presents some possible extensions to the language to improve support for these relationships.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—abstract data types, constraints, polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—reusable libraries; D.3.2 [*Programming Languages*]: Language Classifications—C

General Terms Languages, Design

Keywords generic programming, generics, polymorphism

1. Introduction

Stemming from the pioneering work of Musser and Stepanov [10], generic programming has become an important paradigm for encouraging the design of reusable and composable software. Fundamental to generic programming is the idea that algorithms are expressed in terms of properties of types, rather than in terms of specific types. A *concept* brings together a set of useful properties. An example is the mathematical concept of a semigroup, which abstracts the existence of a binary operation. If a type possesses all of the properties specified by a concept, it is said to *model* the concept.

Although this approach to software design can be used in many different programming languages, certain language features are especially conducive. Key among them is parametric polymorphism [3], the ability to write type-generic functions and data types that can be statically type-checked. Polymorphism itself, however, may still leave much complexity to be managed by the generic programmer. Based on their experience in implementing large-scale generic libraries in a variety of polymorphic programming languages, Garcia *et al*. [7] identified five refinements that can significantly ease generic programming:

- It should be possible to precisely represent and verify the modelling relation between types and concepts at compile time, and possible to extend this relation independently of both the definition of the type and the definition of the concept.
- It should be possible for a concept to express a relationship among multiple types.
- It should be possible for concepts to include types (*associated types*) as well as values and operations.
- It should be possible for generic functions to be instantiated implicitly, with type variables inferred from argument types at a call site.
- It should be possible to provide an alternative name for a type, i.e., a *type alias*.

Recently, Siek and Lumsdaine [12, 13] have created the G language, designed to support all of these features. For the past 15 years, we have been exploring and refining properties of the language C∀ [5] (pronounced ”C for all”), which is an imperative, statically-typed programming language that extends the popular C language with overloading, polymorphism, multiple return-values, and other features. The C∀ type system possesses the desirable features for generic programming, except associated types. Unlike G, C∀ provides all of these features without requiring explicit declarations to establish the modelling relation; we believe it to be unique in this respect. This paper describes how the design of C∀ facilitates generic programming, and various approaches for dealing with the issues of associated types.

2. C∀ Language

This section presents a brief, practical overview of C∀. A formal treatment of the underlying type system can be found in the second author’s Ph.D thesis [6]. The first author’s M.Math thesis [1] describes a practical implementation of the language.

2.1 Overloading

Overload resolution is the process by which the use of an overloaded name is matched with one particular definition of that name, based on the context in which the name is used. In simple overloading systems, multiple functions can be defined with the same name but with different parameter types and/or a different number of parameters, e.g., C++ or Java. In C∀, overloaded functions may be defined with identical parameters but with different return types and/or a different number of return types:

```
void f( int );                  // different return types
int  f( int );
[ int, int ] f ( int );         // return 2 values
```

To select the proper function for a call, both the call arguments and the context in which the result of an expression is used must be considered. In this complex overloading system, overload resolution can still be efficient but requires a more sophisticated algorithm for

expression analysis. Once an algorithm can deal with return-type overloading, it is straightforward to extend it to variables:

```
int x;                          // multiple variable names
double x;                       // in same scope
char x;
```

The appropriate variable is selected based on the context in which the name is used. The constants 0 and 1 can be overloaded because of their special meaning in many application domains:

```
struct complex { double re; double im; };
complex 0 = { 0., 0. };         // declare useful constants
complex 1 = { 1., 1. };
```

Finally, an interesting extension to overload resolution is required for functions returning multiple values when the result can be used for multiple assignment and function composition (i.e., multiple results can provide multiple arguments for another call):

```
[ char, int ] f( void );        // return 2 values
[ int, double ] f( void );
void g( int, double, char, int );  // 4 parameters
// multiple assignment: 2 variables assigned by 1 call
[ i, d ] = f();
// function composition: 4 arguments provided by 2 calls
g( f(), f() );
```

Although the topic of expression analysis and code generation for routines returning multiple values has been discussed previously [2], the combination of multiple return values with overloaded names leads to significant new complexity in the analysis of expressions.

2.2 Polymorphism

A polymorphic function is written using a **forall** clause:

```
forall( type T )
  T identity( T x ) {
    return x;
}
```

The resulting function can be applied to values of any type. [1]

Non-trivial polymorphic functions are written by constraining type parameters using contexts, which specify operations or values that must be provided. Contexts are analogous to the concepts of G or the type classes of Haskell [14]. For example, this context specifies the set of types that have a binary + operator:

```
context semigroup( type T ) {
    T ?+?( T, T );
};
```

where "?+?" is C∀ syntax used to name this operator. A function that constrains a type parameter using context semigroup can then make use of +:

```
forall( type T | semigroup( T ) )
  T twice( T x ) {
    return x + x;
}
```

A similar syntax allows one context to refine another, adding additional constraints:

```
context monoid( type T | semigroup( T ) ) {
    T 0;
};
```

A function that operates on a monoid can use both the operation + and the value 0.

C∀ contexts differ from G concepts in two important ways:

- C∀ does not have the model declarations of G. Verifying a type models a concept is done implicitly by the compiler whenever necessary, based on the presence of variables and functions in accessible scopes having the proper names and types.
- C∀ has no direct equivalent for the associated types and same-type constraints of G. Equivalent constraints are expressed in C∀ using multi-parameter contexts.

Implicit modelling means the following fragment is valid, given the above declaration of semigroup:

```
int ?+?( int, int );
int f( int x, int y ) {
    return twice( x, y );
}
```

The declaration of ?+? is sufficient to establish that the built-in type **int** models the context semigroup. In this particular situation, this declaration is provided automatically by the compiler, since integer addition is defined by the language. In general, the name ?+? may be overloaded with other meanings to allow other types to model semigroup. The context members serve as implicit parameters to a polymorphic function; their precise interpretation is defined as if they are explicit parameters. That is, the declaration of twice is interpreted as:

```
forall( type T | semigroup( T ) )
  T twice( T (*plus)( T, T ), T x ) {
    return plus( x, x );
}
```

and the corresponding use within the function f is interpreted as:

```
    return twice( ?+?, x, y );
```

The type T is inferred by attempting pair-wise unification of the types of the resulting formal parameters with the types of different combinations of interpretations of the arguments. In this case, the fact that ?+? is defined with an appropriate signature on **int** allows the inference to succeed. If there is no appropriate definition of ?+?, the inference fails and a compile-time error is reported at the point of use. The dependence on names to establish modelling means that overloading in C∀ is essential and pervasive.

Combining parametric polymorphism with operation inference in this way was first suggested by Cormack and Wright [4]. The alternative is an explicit language mechanism, such as the model declaration of G or the instance declaration of Haskell. These languages require a declaration for every combination of type and concept in the modelling relation. In comparison, C∀ needs none of these declarations, which we believe to be a real and significant saving.

3. Accidental Conformance

Implicit modelling gives additional flexibility and economy of notation, but adds the risk that a type may accidentally conform to a context by defining operations with the appropriate names but inappropriate semantics. Gregor and Siek [8] provide good examples of cases where accidental conformance arises. One example, adapted into C∀ syntax, is:

```
context Groupoid( type T ) {
    T ?+?( T, T );
};
context Semigroup( type T | Groupoid( T ) ) {
    // operator + is associative
};
```

A type can model Semigroup with an operation of the same name and type as is needed to model Groupoid. The difference is that there are additional constraints on the behaviour of the operation. Since implicit modelling is inferred based only on names and types, there is no way for it to be robust in the face of subtle semantic differences such as this. C∀ accepts any type that models Groupoid as a model of Semigroup, even when the + operation is not associative. In some cases of accidental conformance, an appropriate renaming of operations can solve the problem (albeit painfully), but not in a case of semantic refinement such as this:

[1] Strictly speaking, any *complete* data type, i.e., a type that is not a function type and whose size is known to the compiler.

a type with a + operation that models Semigroup should use the same operation to model Groupoid; it is only the reverse case that is in error. As a result, there are certain cases where explicit modelling is necessary.

Explicit modelling can be expressed in C∀ by using an overloaded identifier to indicate conformance:

```
context Semigroup( type T | Groupoid( T ) ) {
    T plus_is_associative; // operator + is associative
};
```

With this new requirement, a type defining the + operator by itself only models Groupoid, not Semigroup; if the operator is associative, a variable plus_is_associative can be defined for that type, which allows it to also model Semigroup.

In practice, it seems accidental conformance should be the exception, rather than the rule. Garcia *et al.* note that "...in our experience, accidental conformance is not a significant source of programming errors" [7, p. 130]. While explicit modelling removes the possibility of accidental conformance, the disadvantage is that it requires a model declaration for every pair of type and context, even when there is no semantic confusion. There is no way to provide implicit modelling given explicit modelling, whereas it is straightforward to require explicit modelling in an implicit language such as C∀. It is therefore appropriate to allow a context designer the choice of explicit modelling, rather than the designer of the language.

4. Associated Types

The fundamental purpose of associated types is to establish a relationship between the primary type or types in a concept and the associated type, the latter being uniquely determined by the former. This section explores the way in which type relationships are currently expressed in C∀ contexts and ways in which C∀ could be extended to provide the advantages of associated types while retaining implicit modelling.

4.1 Representing Associated Types

Here is an example of an iterator concept with an associated type, as expressed in G:

```
concept Iterator<Iter> {
    type Elt;
    next : fn(Iter)->Iter;
    curr : fn(Iter)->Elt;
    at_end : fn(Iter)->bool;
}
```

Each type that is a model of Iterator must have a corresponding type Elt, which serves as a return type for the function curr. The primary type (Iter) uniquely determines the associated type. Given a type that models Iterator, its associated Elt type can be accessed using a qualified name, to be constrained in a where clause or used in a declaration, e.g.:

```
fun accumulate<Iter>
  where { Iterator<Iter>, Groupoid<Iterator<Iter>.Elt> }
(Iter first, Iter last, Iterator<Iter>.Elt init) -> Iterator<Iter>.Elt
```

This declaration declares a function accumulate, parameterized on a type Iter that models Iterator and whose associated Elt type models Groupoid. The function takes two values of type Iter and one of type Elt as parameters, and returns a value of type Elt.

By contrast, C∀ makes no provision for types to be included in contexts, except as context parameters. This restriction rules out associated types as found in G. C∀ establishes type relationships without associated types or same-type constraints by using multi-parameter contexts. To represent an associated type in a C∀ context, it is only necessary to add a corresponding name to the type parameter list, and introduce an operation to the context whose type relates the associated type to the primary type:

```
context Iterator( type Iter, type Elt ) {
    // functional dependency
    Elt iterator_elt( Iter );
    // context operations
    Iter next( Iter );
    Elt curr( Iter );
    bool at_end( Iter );
};
```

The run-time behaviour of the iterator_elt operation is irrelevant; it is useful only to guide the inference. In this way, the added operation plays the role of a *functional dependency* in Haskell [9], defining relationships among type parameters. As a result, there is no explicit notion of primary and associated type. The various overloads of iterator_elt must uniquely determine, for each Iter, the corresponding Elt. Rather than use a qualified name, the associated type is declared separately in the **forall** clause:

```
forall( type Iter, type Elt
      | Iterator( Iter, Elt ), Groupoid( Elt ) )
  Elt accumulate( Iter first, Iter last, Elt init );
```

A type can model the context Iterator by defining the iterator_elt operation to encode an appropriate choice of associated type, just as it would specify the type in an explicit model declaration:

```
struct MyType {
    // implementation
};
struct MyIterator {
    // implementation
};
MyType iterator_elt( MyIterator ) {}
```

The striking feature of C∀ contexts, however, is that in most cases this additional operation is unnecessary. Usually if one type is associated with another, that relationship is already expressed within the context by some useful operation. In the Iterator context, for instance, the iterator_elt function is superfluous: it has the same signature as curr, which suffices to relate the two types, eliminating the need for an additional operation. In the worst case, sets of unrelated types can be associated within the context using extra operations to guide inference; in the usual case, the set of associated types is already encoded within the context, so the type associations can be made implicitly and automatically.

In addition, multi-parameter contexts are capable of expressing other relationships among concepts. For instance, consider expressing mixed-type multiplication as a context (adapted from Jones [9]):

```
context Multiplicable( type X, type Y, type Z ) {
    Z multiply( X, Y );
};
int multiply( int, int );
double multiply( int, double );
double multiply( double, int );
double multiply( double, double );
```

These declarations specify that multiplication of two **int**s returns an **int**, while multiplication of an **int** and a **double**, or two **double**s, returns a **double**. A multi-parameter context is necessary since there is no single primary type: it is a particular combination of the two types X and Y that determines Z. Contexts in C∀ are therefore sufficient to represent concepts with associated types, as well as other type relationships that cannot be represented using associated types.

4.2 Associated Types and Notation

Even if associated types are logically unnecessary, they may still be useful if they provide a more concise or convenient notation. Multi-parameter contexts can express complex type relationships, but they have the side effect of giving all types equal prominence in the interface. The context parameter list and every function

constraint or refinement that uses the context must mention every associated type. In a highly-generic system using many contexts, each involving many associated types, significant complexity and reduced readability can occur [7].

Furthermore, a consequence of the notational economy of associated types is that the resulting program is better prepared for augmentation of the concepts. A concept can be extended to include more associated types without requiring all functions using the concept to be changed as well, so long as those functions remain independent of the choice of associated type.

In order to assess the impact of associated types on notation, four cases are identified:

1. A function whose signature uses a context involving an associated type, where the signature does not constrain the associated type, and where the function implementation does not mention the associated type.
2. A function such as in case 1, but where the associated type is used in the implementation of the function.
3. A function such as in case 1 or 2, but where the associated type is used in the signature of the function.
4. A context that refines one or more contexts involving associated types.

To illustrate these cases, consider an Sequence context that, conceptually, has two associated types:

```
context Sequence( type Seq, type Elt, type Iter
                | Iterator( Iter, Elt ) ) {
    Iter begin( Seq );
    void insert( Seq, Iter, Elt );
    void remove( Seq, Iter );
    bool is_empty( Seq );
    void push_back( Seq, Elt );
};
```

Case 1 presents the most obvious benefit: the function does not need to name the associated type. In implementing such a function in C∀, the need to mention all associated types in the signature makes the function overly verbose and sensitive to additions to the context. An example is a function that uses the Sequence concept, without referring to the Iter type:

```
// push back n copies of e
forall( type Seq, type Elt, type Iter
        | Sequence( Seq, Elt, Iter ) )
  void push_back_n( Seq s, Elt e, int n ) {
    for( int i = 0; i < n; i += 1 ) {
        push_back( s, e );
    }
}
```

Here, the function is completely independent of the type Iter, yet it must give that type a name in its signature.

In case 2, the C∀ function must give the type a name, but that name is used within the implementation of the function.

```
// add e if not already present in s
forall( type Seq, type Elt, type Iter
        | Sequence( Seq, Elt, Iter ),
          EqualityComparable( Elt ) )
  void set_add( Seq s, Elt e ) {
    for( Iter i = s.begin(); ! at_end( i ); next( i ) ) {
        if( e == curr( i ) )  return;
    }
    push_back( s, e );
}
```

Assume that the context EqualityComparable provides the == operator. Here, the use of the type Iter is confined to the implementation. In this case, the convenience of associated types is solely that the type need not be named in the function signature; the implementation still relies on the type, and must refer to it using a qualified access. An informal survey of large C++ template libraries (e.g., the Boost Graph Library [11]) shows that programmers tend to avoid such qualifications by introducing local names for associated types. If a local name is desirable in any case, the fact that C∀ requires a local name is of little concern. One other disadvantage, however, is that given a separately declared interface signature for the function, the name must appear there as well. As a result implementation details are exposed to users of the function.

In case 3, it is necessary to mention the type in the function signature; an example is the use of the Elt type in the above set_add function. The advantage of associated types is minimized, since the type must be mentioned using a qualified name. In fact, multiple uses of an associated type might render the associated type version *more* verbose than a C∀ equivalent, without a facility to define a local name for an associated type for use in the signature.

In case 4, a refinement of one or more concepts involves the complete set of associated types in all of the base concepts. In a C∀ context, all of these types must be explicitly named in the type parameter list of the refinement context. As a result, the number of type parameters can only grow as concepts are progressively refined and combined.

```
context Array( type Seq, type Elt, type Iter, type Index
             | Sequence( Seq, Elt, Iter ) ) {
    Elt at( Index );
};
```

Mitigating this problem is the fact that context definitions should be much more rare than uses of the contexts. However, this accumulation of type parameters imposes a burden on all users of the context, as described in the other cases.

Given that associated types can be represented by C∀ even without special support, it is worth asking whether some of the notational difficulties mentioned here can be helped by minor changes to the language, while preserving implicit modelling. One approach is to allow function signatures to leave some type parameters unnamed:

```
forall( type Seq, type Elt | Sequence( Seq, Elt ) ) // no Iter
  void push_back_n( Seq s, Elt e, int n );
```

Here, the Seq type is subject to the same constraints as before; the difference is only that there is no name defined for the sequence's Iter type now, so it cannot play a role in the signature or in the implementation. Since the Seq type uniquely determines the Iter type, there is no resulting ambiguity. This construct can be converted into an equivalent form by a translator that generates an arbitrary name for the omitted type; this version can be interpreted according to the traditional C∀ rules. To provide the most flexibility, it should be possible to mention an arbitrary set of context arguments, while leaving the rest out. This capability can be achieved using named, rather than (or in addition to) positional associations in context parameter lists.

```
forall( type Seq, type It
        | Sequence( Seq, Iter = It ) ) // no Elt
  It nth_iterator( Seq s, int n ) {
    for( It i = s.begin(); ! at_end( i ) && n > 0; n -= 1 ) {
        next( i );
    }
    return i;
}
```

The ability to omit types from **context** argument lists addresses case 1 above. It also mitigates case 2, since the interface signature can omit arguments, even if the implementation signature needs to name the type; i.e., the type constraint is the same regardless of which types are named, so long as the named types are sufficient to uniquely determine the unnamed types. This condition holds for any type that could be otherwise written as an associated type.

Unfortunately, this extension does little to mitigate the direct issue of case 4. If a context declaration leaves some type parameters unnamed in a refinement, it precludes any users of that context from naming those parameters:

```
context Array( type Seq, type Elt, type Index
             | Sequence( Seq, Elt ) ) { // no Iter
    Elt index( Index );
};
```

While this definition of Array might be useful in cases where elements are only accessed using index, it prevents users from taking full advantages of Sequence, since the Iter type cannot be named. This seems presumptuous, and is not a general solution in any case. However, this solution does help the users of a refined context, since they can pick and choose relevant type parameters.

The advantage of this extension is that it succeeds without significantly affecting the syntax or semantics of the language. A different approach is to explicitly add associated types into C∀ contexts. This method adds more complexity to the definition of the language, since it involves introducing new rules for these types as well as a new syntax to qualify associated type names. However, it is technically possible, and can be done while preserving implicit modelling. The meaning of:

```
context Iterator( type Iter ) {
    type Elt;
    Iter next( Iter );
    Elt curr( Iter );
    bool at_end( Iter );
};
```

can be defined to be the same as:

```
context Iterator( type Iter, type Elt ) {
    Iter next( Iter );
    Elt curr( Iter );
    bool at_end( Iter );
};
```

with the exception that the associated type version omits the Elt type from the type parameter list, requiring a qualified name in situations where it must be used. The signature for the accumulate function above changes correspondingly to use a qualified name:

```
forall( type Iter
      | Iterator( Iter ), Groupoid( Iterator( Iter ).Elt ) )
  Iterator( Iter ).Elt accumulate( Iter first, Iter last,
                                   Iterator( Iter ).Elt init );
```

Implicit modelling still requires that the operations in the context uniquely determine the associated types. In some cases, this restriction may require the introduction of an additional operation to the context in order to capture the functional dependency, as described in the previous section.

5. Conclusion

Generic programming has great potential to facilitate software reuse. With greater abstraction, however, comes a greater need for languages and tools that help to structure and manage the abstractions. Details matter: as Garcia *et al.* found in their survey [7], small inconveniences can become large as problems increase in size. Two relevant inconveniences are the need to establish concept conformance, and the need to manage associated types. This paper presents, in C∀, an example of a language that significantly reduces the burden of modelling through automatic inference. At the same time, C∀ can represent associated types in their full generality, and it is possible to make minor adjustments to the language to ease the use of these types while still preserving implicit modelling. While the choice between implicit and explicit modelling is likely to remain controversial, it is important to expose the possibilities for future discussion; C∀ serves to illustrate these possibilities by example.

References

[1] Bilson, R. C. "Implementing Overloading and Polymorphism in Cforall". Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2003.

[2] Buhr, P. A., Till, D., and Zarnke, C. R. "Assignment as the Sole Means of Updating Objects". *Software—Practice and Experience*, 24(9):835–870, Sept. 1994.

[3] Cardelli, L. and Wegner, P. "On Understanding Types, Data Abstractions, and Polymorphism". *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985.

[4] Cormack, G. V. and Wright, A. K. "Type-dependent Parameter Inference". *SIGPLAN Notices*, 25(6):127–136, June 1990. Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A.

[5] Ditchfield, G. *Cforall Reference Manual and Rationale*, revision 1.82 edition, Jan. 1998. ftp://plg.uwaterloo.ca/pub/Cforall/-refrat.ps.gz.

[6] Ditchfield, G. J. *Contextual Polymorphism*. PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1992. ftp://plg.uwaterloo.ca/pub/theses/-DitchfieldThesis.ps.gz.

[7] Garcia, R., Järvi, J., Lumsdaine, A., Siek, J. G., and Willcock, J. "A Comparative Study of Language Support for Generic Programming". In *Proceedings of the 2003 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'03)*, Oct. 2003.

[8] Gregor, D. and Siek, J. "Explicit model definitions are necessary". Technical Report N1798=05-0058, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2005.

[9] Jones, M. P. "Type Classes with Functional Dependencies". In *Proceedings of the 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 2000.

[10] Musser, D. R. and Stepanov, A. A. "Generic Programming". In *ISAAC '88: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 1989.

[11] Siek, J., Lee, L.-Q., and Lumsdaine, A. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[12] Siek, J. and Lumsdaine, A. "Essential Language Support for Generic Programming". In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, pages 73–84, New York, NY, USA, June 2005. ACM Press.

[13] Siek, J. and Lumsdaine, A. "Language Requirements for Large-Scale Generic Libraries". In *GPCE '05: Proceedings of the fourth international conference on Generative Programming and Component Engineering*, September 2005. accepted for publication.

[14] Wadler, P. and Blott, S. "How to make *Ad-Hoc* Polymorphism Less *Ad-Hoc*". In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 60–76. Association for Computing Machinery, 1989.