Doc no: N2887=09-0077
Date: 2009-06-21
Project: ISO/IEC JTC1/SC22/WG21
Reply-To: Gabriel Dos Reis
gdr@cs.tamu.edu

Axioms: Semantics Aspects of C++ Concepts

Gabriel Dos Reis Bjarne Stroustrup Alisdair Meredith

Abstract

This paper clarifies the semantics of "axioms" in the C++ concept proposal and provides standard wording, following the C++ committee vote and resolution at the Spring 2009 meeting at Summit, NJ.

1 Motivations

What are axioms for and what good can they do if used well? The primary motivations for axioms as language constructs are:

- making assumptions about program fragments explicit and clear. Such assumptions are typically made anyway; and in the absence of language support, they are free style English text comment, which are more likely to be misunderstood than the predicate logic forms
- provide support for tool builders: Many forms of analysis and transformation is beyond the scope of a compiler (for reasons of complexity, commonality, and cost.) However, many such uses (*e.g.* for program transformation or analysis for concurrency) are quite feasible (*e.g.* done today in special-purpose languages) with no more syntactic and semantic support than we are proposing.

Note that "helping the optimizer" and especially *not* "directing the optimizer" are not among the primary motivations. Even though optimization benefits might be had for some axioms and some compilers, a general attempt to blindly perform transformations based on axioms in a compiler is doomed to fail. We consider a compiler that does nothing beyond syntax

checking the ideal general implementation. Every transformation benefit will occur from special cases taken advantages of by specially written tools (including optimizers) which are selective about axioms in which they are interested.

Such tools traditionally assume that the right hand side of an axiom is "simpler" or "better" than the left hand side. Such an assumption is sufficient for quite powerful tools. We deliberately do not propose a "richer transformation language" because doing so would be far beyond the scope of the C++ standards and at best premature. What we propose is conventional, understood, and very powerful for expressing semantics assumptions.

2 What are "axioms"?

An axiom states what an implementation may assume of values, but also what properties a user should and should not assume about values. Those properties are logical statements about runtime semantics we assume to hold for a proper execution of an algorithm. Axioms have all been part of both C and C++: Traditionally, we write `*p = 0` and know that the implementation may assume that `p` points to a valid object of the appropriate type, and blindly dereferences `p`. As users, it is our obligation to ensure that the assumption is not violated and we should not complain too much if it was. It is not the obligation of the obligation of the implementation to "verify" or "implement" the assumption (whatever that might mean.) Obviously, an implementation may try to check the validity of the assumption and give us a suitable error message if it is violated. The "assumption" in the previous example is just an axiom that has been in C from its inception. Having axioms in the language itself formalizes that without placing new obligations on implementations or users. As for the "assumptions" that we have had for decades, an implementation may (optionally as a quality of implementation issue) check some of our uses that can be considered constrained by axiom.

For libraries (especially generic libraries) we have missed language constructs to express assumptions about user-defined abstractions. For example, C++ programmers have no way (other than in form of comments or implementation details) to express — at an abstract level — the notion that their overloaded operators == and `!=` are consistent; *i.e.* the expression `x != y` is functionally equivalent to `!(x == y)`. This is one of the places where we have failed our long standing C++ design rule to strive for equal sup-

port for builtin abstractions and user-defined abstractions.

Axioms are essential components of C++ concepts as envisioned since their inception. In fact, axioms constitute a fundamental component of the STL in particular, and of generic programming in general. Axioms cannot meaningfully be decoupled from the syntactic requirements of algorithms. For instance, paragraph 24.1.1/2 from the C++03 specification of the iterator library component reads

> In Table 72, the term the *domain of* == is used in the ordinary mathematical sense to denote the set of values over which == is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of == for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of == and `!=`. [*Example:* the call `find(a,b,x)` is defined only if the value of `a` has the property `p` defined as follows: `b` has property `p` and a value `i` has property `p` if (`*i==x`) or if (`*i!=x` and `++i` has property `p`).]

There are various observations to make here. Perhaps the most important aspect is that the assumptions on `find` are expressed in terms of the *actual values* used to invoke `find`, as opposed to a blanket statement about an iterator type that satisfies the syntactic requirements in table 72. We cannot stress this point enough. The second point, which in some sense is a continuation of the first, is that the domain of an operation can change over time, *i.e.* is dependent on call instances. For example, the above description says that the equality operator is an equivalence relation over the domain of iterators involved in the call to a particular algorithms. Third, the property `p` is in fact a termination assumption on the function call `find(a,b,x)`. A C++ implementation is not required to prove[1] that the assumption `p` holds for each call to `find`. In fact, since it is a well-formed condition, the implementation may as well assume that the assumption holds[2] and use that knowledge for internal implementation or any code generation purposes.

Abundant examples are available throughout the entire iterator, algorithms, and container libraries of C++98 and C++03.

Axioms are expressions of semantics assumptions that used to be kept in informal comments (because of lack of language support), just like syntactic requirements for generic algorithms used to be kept in informal comments.

[1] Of course, under the *as-if* rule, an implementation may elect to conduct such proof, as long as the observable behaviour of the program is retained.

[2] which is what most implementations do.

3 How do we express axioms?

We think of axioms as logical statements, and the question is which logic to use and which meta logical language to use to express axioms. The proposal suggests the usual first order predicate logic, based on the practical observation that for most practical cases, it is sufficient. In fact, the logic language we are about to propose would let us express only a restricted subset of first order predicate logic.

We suggest to use the usual C++ builtin logical connectives

- `!` (`not`) for logical negation;
- `&&` (`and`) for logical conjunction;
- `||` (`or`) for logical disjunction;
- `if`-statement for logical implication;
- a new primitive <=> for expressing equivalence of behaviour of the C++ abstract machine.

For example, the statement that if the iterator `p` and `q` are dereference-able and such that `p` compares equal to `q`, then the expression `*p` is equivalent to the expression `*q` can be expressed as

```
if (p == q)
   *p <=> *q;
```

Similarly the traditional description of the expression `*p++`, where `p` is a dereference-able input iterator can be expressed as

```
*p++ <=> { T tmp = *p;  ++p; return tmp }
```

The `return`-statement in the block on the right hand side says that the execution of the whole block yields a value which is that of the expression in the `return`-statement.

It is imperative to note that the above is *not* saying that every time the expression `*p++` occurs then the compiler should go out of its way and forcefully replace that expression by the longer block-statement. Rather, the above logical statement is establishing an equivalence of behaviour that the C++ abstract machine can assume. Note also that under the as-if rule, a compiler may use that information — which is part of the semantics of * for input iterators — for various purposes.

In addition to the usual propositional connectives, we also need to quantify propositions or expressions. Where it is customary to use the symbol

$\forall$ to signify universal quantification, we use the usual formal parameter binding in C++ function definitions. For example, we write

```
axiom symmetry(T x, T y) {
  x + y <=> y + x
```

to say:

> for all expressions `x` and `y` of type `T`, the expression `x + y` is equivalent (from the *as-if* rule point of view) to the expression `y + x`.

4 Formal wording

4.1 A new token

This proposal, following a NB comment unanimously agreed on by the committee, introduces a new preprocessing token <=> for expressing equivalence of computations.

> **In paragraph** 2.13/1**, augment the rule *preprocessing-op-or-punc* with <=>.**

4.2 Axiom syntax

Replace the grammar rule *axiom* in paragraph 14.10.1.4/1 with

atomic-proposition:
    expression
    compound-statement

proposition:
    atomic-proposition
    atomic-proposition `<=>` *proposition*

axiom:
    proposition $_{opt}$ `;`
    `if (` *logical-or-expression* `)` *axiom*

The example in that paragraph should read

```
concept Semigroup<typename Op, typename T> : CopyConstructible<T> {
  T operator()(Op, T, T);
  axiom Associativity(Op op, T x, T y, T z) {
    op(x, op(y, z)) <=> op(op(x, y), z);
```

```
  }
}

concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
  T identity_element(Op);
  axiom Identity(Op op, T x) {
    op(x, identity_element(op)) <=> x;
    op(identity_element(op), x) <=> x;
  }
}
```

4.3 Axiom semantics

We do not require that C++ implementations are turned into automatic theorem provers. In particular, we suggest to edit the second sentence of paragraph 14.6.8/7 as follows

> The specialization is considered to have an incompatible definition if the specialization's definition causes a different definition of any associated type or associated class template in the concept map, if its definition causes any of the associated function definitions to be ill-formed~~, or if the resulting concept map fails to satisfy the axioms of the corresponding concept~~.

We also propose

to remove the paragraph 14.10.2.4/14**.**

Since this proposal is a rewrite of the existing section for axioms, we propose to place all paragraphs starting from 14.10.1.4/2 onwards with the following:

2 A semicolon (;) is required to terminate an *axiom* when the last token is not a closing brace (}). An *atomic-proposition* of the form of a *block-declaration* shall contain at most one `return`-statement.

3 Within the body of an *axiom-definition*, name lookup is performed as if in the body of constrained function template definition. A parameter of an *axiom-definition* is a placeholder for expressions of the corresponding type. Consequently, in an *axiom-body* an expression (resp. block-statement) containing axiom parameters is an *expression pattern* (resp. *block-statement patterns*).

4 If an axiom has the form *expression-statement*, then the *expression* shall be a *logical-or-expression* contextually convertible to `bool`. The semantics is that whenever a function constrained by the enclosing concept is executed, any contained expression matching (both in type and in

structure) the form designated by the axiom holds as true during the execution of that function. If an axiom is of the form *proposition* <=> *proposition*, then during the execution of any function constrained by the enclosing concept, any contained expression or block statement matching (both in type and structure) the form designated by the left side of the axiom is equivalent (under the *as-if* rule) to the corresponding expression or block statement on the right hand side of the axiom. Finally, if an axiom is of the conditional form then, during the execution of any function constrained by the enclosing concept, if condition holds then the body of the conditional axiom also holds.

Replace the entire paragraph 14.10.1.4/5 with the following example

5 *[Example:*

```
concept TotalOrdering<typename Op, typename T> {
  bool operator()(Op, T, T);

  axiom Antisymmetry(Op op, T x, T y) {
    if (op(x, y) && op(y, x))
       x <=> y;
  }

  axiom Transitivity(Op op, T x, T y, T z) {
    if (op(x, y) && op(y, z))
       op(x, z);
  }

  axiom Totality(Op op, T x, T y) {
    op(x, y) || op(y, x);
  }
}
```

—end of example]

5 An auxiliary proposal

This section describes an auxiliary proposal, *i.e.* it is independent from the main proposal. On the one hand it is a concise, simple, and conventional notation for an existing feature, *i.e.* it suggests to introduce a new token => for expressing implications instead of using one-armed `if`-statement. For instance, the axiom

```
axiom antisymmetry(T a, T b) {
  if (a < b)
    !(b < a);
```

would be written as

```
axiom antisymmetry(T a, T b) {
  a < b => !(b < a);
```

On the other hand, most programmers don't remember that => is conventional for implication — it was defined and almost universally unused for Algol60. Some may argue that it would make axioms and C++ look more elitist. Secondly, there is the danger that that it would almost certainly be read directionally as "transforms into".

1. **In paragraph** 2.13/1**, augment the rule** ***preprocessing-op-or-punc*** **with =>.**

2. Replace the grammar rule *axiom* in paragraph 14.10.1.4/1 with

   axiom:
   proposition $_{opt}$;
   ~~`if (` *logical-or-expression* `)` *axiom*~~
   logical-or-expression **`=>`** ***axiom***

3. Change all the other places where an implication is expressed as an `if`-statement.