Christopher Patton
Professor Karl Levitt
ECS 240 – Programming Languages
14 June 2012
Final research proposal

An algebraic semantics for C++

Introduction and research goal

The influence and impact that C++ has had on the world at large is not to be understated. It's an integral piece of many software systems that require higher level programming paradigms, but cannot sacrifice the efficiency of native compiled code. Since its invention in the 1983, C++ has paralleled the evolution of the object oriented and generic programming paradigms. Today, C++ comprises most features related to these coding schemes, including many features uncommon to other comparable languages. However, in many peoples' estimation, this complexity adds as much frustration to programming in C++ as it does usability. Writing a parser for C++ is already challenging, as it adds a lot of ambiguity to the grammar of C. Developing a formal semantics for the language seems exponentially more difficult, and perhaps not worth the effort it would require. I would like to side-step this somewhat religious argument and adapt the following perspective: the fact that C++ embodies so many features and paradigms that are of fundamental importance to Computer Science means that understanding the language semantically has value to the community in its own right, outside the language itself. In studying C++ semantically, I would like to draw attention away from the discussion of whether industry and software community *should* use C++ and focus on the paradigms it comprises that have changed our way of thinking.

The idea for this research project is to develop an algebraic semantics for C++, particularly those aspects pertaining to object oriented programming and data abstraction via

generic programming. In order to establish a framework for my research efforts, I will first review previous work developing the semantics of the critical code. Then I will identify the specific problems that need to be solved in order for successful development of a C++ algebraic semantics and a proposed methodology for moving forward on solving them.

Background

Many papers, most successfully, have attempted to specify a formal semantics for C/C++. In fact, there are a few compiler tools that perform semantic verification of C programs. In “Experiments in validating formal semantics for C”, Sandrine Blazy describes the specification of one such tool, a verifying C compiler called CompCert (Blazy 2007). CompCert is designed to detect common runtime errors at compile time that plaque all programmers, such as buffer overflows and dangling pointers. Blazy takes an operational perspective in this approach to the problem. The compiler verifies the code by successively translating it into intermediate languages, each designed to capture a different semantic entity and check for errors. The work is based on the proof for the theorem that if a language S is translated into S' via a verified compiler without reporting errors, then S' has the same well-defined semantics of S (Blazy 2007).

While this operational view provides an efficient and well-formed way of verifying C. C++ adds a lot of complexity to C and the assumption is that Blazy's method is not sufficient to fully capture the semantics of C++, especially its dynamic features.. Michael Norrish takes a slightly different approach and provides a complete operational semantics for C++ using the HOL theorem prover for verification (Norrish 2008) that warrants investigation. He distinguishes between three phases of a C++ program that he calls name resolution (dealing with namespaces and scope, a relatively complex aspect of C++), templates, and dynamics. This work does not attempt to describe an actual compiler as in CompCert, but rather provides a semantics with

which others could potentially construct useful tools. Norrish's method provides a great basis for studying C++ semantics. This HOL specification developed by Norrish even opens the door for proving C++ programs (albeit from abstract syntax). However, the fact that this work takes an operational view renders this system limited in a few different ways. In my estimation, the Norrish approach is a bulky system that does not capture the notion of an object in a natural way[1]. It is possible that an operational view may be fundamentally unsuited for a formal semantics of object oriented languages.

In response to this problem, the principle motivation for this research project is to create a semantics for C++ which would provide a more natural description of objects. This project will build upon work done by Alexander Fronk on the algebraic semantics of object oriented languages (Fronk 2002). In his paper “Algebraic Semantics of Object-Orientated Languages”, Fronk considers a simple OO language, *DoDL,* and presents a transformation of syntactic entities in that language into an algebraic semantics. He does this in two steps. First, Fronk defines each class as a Σ-algebra, where the classes' attributes and methods correspond to operations. The name of the class becomes one of the algebra's sorts. (He thinks of the class name as the type of object). The implementations of the class methods are transformed into the algebra's axioms. He also provides a scheme for inheritance similar to the way formal algebras import sorts, such as booleans, integers, or strings. In the second step, Fronk shows how imperative elements of *DoDL* described denotationally (ie. in a formal denotational semantics) can be reduced to axioms in the *DoDL* algebra, thus completing a formal semantics of the whole language.

1 I think the intuition for why an operational semantics is not desirable is fairly sound, although I offer no evidence here. To demonstrate that an algebraic semantics provides a much more natural framework for proof and verifications of C++ programs will require a much closer reading of Norrish's work.

Methodology

In writing an algebraic semantics for C++ classes, I will follow Fronk's two step approach. First, I will demonstrate a reduction of C++ classes into Σ-algebras. This work will extend Fronk's research by including the features of C++ that he didn't consider in *DoDL*, *n*amely multiple inheritance, virtual and abstract classes, scoping rules, protection rules (`public`, `protected`, `private`), as well as method and operator overloading. As part of the development process, I will write the algebraic semantics in three steps based on Norrish's work: name resolution, templates, and dynamics. It will be an incremental process of including one feature at a time and modifying the model as needed. Second, I will formalize a reduction of a denotational semantics of the imperative characteristics of C++ into algebraic axioms in the semantic model. In order to control the scope of this project, however, I would like to think of implementation of the denotational semantics C++ (and by extension C) as a black box, since such a semantics is an extremely challenging problem on its own. Instead, I will only consider the specification of a theoretical denotational semantics.

Challenges and conclusions

The present project is full of a number of research challenges that result from the complexity of C++. In particular, modeling polymorphism will be a difficult problem to solve. Polymorphism in C++ is in some sense a superset of of Fronk's *DoDL*. Polymorphism is explicit in the *DoDL*-syntax via the keywords `this` and `super.` C++ also allows implicit polymorphism by way of virtual methods and pointer/reference casting. For example: `(SuperClass*)anInstance->method();`, where `anInstance` is an instance of class `SubClass public: SuperClass`. A second hurdle to jump will be multiple inheritance. As of now, it appears to me that this will simply require an extension on Fronk's original concept of importing

the super class into the algebra of the subclass, though it may prove problematic in the context of polymorphism. Finally, the notion of abstract classes (`template`) is arguably the most important feature of C++ that isn't explored on in Fronk's paper. It will be challenging to provide a formal definition for this idea, algebraic semantics lends itself to this solution. The fundamental importance of abstract data types in C++ programming and as a paradigm in general provides a huge motivation for solving this problem.

A major criticism for this research topic is the question of how to make it practical, especially in the absence of a denotational semantics of C and the imperative features of C++, on which it critically depends. Assuming we had such a semantics, an algebraic semantics of C++ objects would give us all the advantages of a good semantics, such as compiler-time verification and code proving. Ideally, we could build upon the work done for CompCert for C, capturing C++ constructs algebraically. In order to accomplish such a task, we must develop a way to translate this operational view into a C++ class algebra. Another immediately useful possibility would be to extend Norrish's HOL prover with C++ algebraic semantics. I conclude in remarking that the work I do towards an algebraic semantics will be theoretical in order to prove the principals; making it practical will require modifying it to work with an existing operational semantics of the imperative aspects of C++.

For good or for bad, C++ is one of the most proliferate language in the world, trailing only its predecessor C. The language's success is a result of its support for many programming paradigms that have pushed the world into the future of software development. However, these features have made C++ as complex as it is useful, leaving it as one of the least-understood languages we use. The motivation for a formal semantics of C++ is for provably good code, a goal that has its own merit. However, understanding the features it embodies, such as

polymorphism, multiple inheritance, and generic programming (abstraction) is of fundamental value to Computer Science as well. The goal of this research project is to develop an algebraic semantics for C++ classes, building upon the work of Norrish and Fronk. This will be accomplished in two steps: one, show a reduction of classes into Σ-algebras (Fronk), considering the three phases of C++ programs as proposed by Norrish; two, transform denotational semantics of C++ into algebraic axioms (Fronk), in order to capture the imperative features of C++. In doing so, I will keep a specification of the denotational semantics in mind and leave the implementation out of the scope of this project. As a result of completing this project, I hope to create the foundation for the creation practical tools for verifying features of C++ programs, especially as they pertain to classes.

References

S. Blazy. Experiments in validating formal semantics for c. In C/C++ Verification Workshop, pages 95–102, Oxford United Kingdom, 2007.

M. Norrish. A formal semantics for C++. Technical report, NICTA, 2008.

A. Frank. Algebraic Semantics of Object-Oriented Programming Languages. Software-Technology, University of Dortmund, Germany, 2002.