Project 2 Report

June 12, 2014

1 Overview

Discuss the high level goals of your work, along with any interesting/key findings.

Our objective was to understand and experience the issues that arise when actually implementing program analyses. To this end, we constructed a dataflow analysis framework based on the LLVM compiler infastructure.

We confirmed through experience that SSA makes flow functions *much* easier to write. We found that the most helpful part of the LLVM API is the object hierarchy for dealing with the graph of Value objects representing LLVM IR. Subclasses of Value are rich, representing functions, basic blocks, instructions, constants, etc. These IR object classes provide many useful convienence functions and make it easy to traverse the program. The least helpful parts of the LLVM API is the pass manager and the special LLVM replacement for C++ run-time type information; these features are obscure and difficult to use.

The most surprising feature of LLVM was the InstVisitor template class. This class made our implementation much easier and cleaner than it would have been otherwise, but examples in the introductory documentation do not use it. For something so helpful to LLVM newcomers, InstVisitor is poorly advertised in the LLVM documentation.

Though the analyses were relatively straightforward in terms of flow functions, the technical hurdles of actually making them work were nontrivial, and we learned a great deal. This project provided a solid foundation of practical experience for building programming-language tools based on LLVM. We document our implementation efforts in the following pages, beginning with the overall design for the dataflow analysis framework.

2 Interface Design

Describe interface. Discuss what alternative designs you may have also considered, and explain the tradeoffs that ultimately led to your choice.

LatticePoint Every dataflow analysis requires a Lattice to work over. The *LatticePoint* class describes our interface to these points. Anything that inherits from *LatticePoint* must implement:

- **equals**(LP) $\rightarrow$**Bool** Compare this latticepoint and the argument for equality, returning a Boolean.
- **join**(LP) $\rightarrow LP$ Return a new LatticePoint containing the join of this LatticePoint and the argument.
- **isBottom**() $\rightarrow$**Bool** Returns true if this latticePoint is Bottom
- **isTop**() $\rightarrow$**Bool** Returns true if this latticePoint is Top.

FlowFunction This class provides an interface to calling flow functions. Derived classes must implement a single special function:

$$\textbf{operator}(Instruction, Vector\langle LP\rangle) \rightarrow Vector\langle LP\rangle$$

Implementing the "operator" function makes objects of this type callable. To define each analysis we make a subclass of *FlowFunction* that can operate on a particular type of *LatticePoint*. When "operator" is called, a single step of the flow function is exectued with the argument *LatticePoints* as incoming edges on the control flow graph. A vector of *LatticePoints* that should be on the outgoing edges of this instruction is returned.

Analysis This class contains the implementation of the worklist algorithm. It is not an interface, and is not meant to be derived from. It has one public function:

$$\textbf{analyze}(Function, FlowFunction, LP) \rightarrow Map\langle Instruction, LP\rangle$$

To use our framework, a caller first instanciates a "bottom" *LatticePoint* and a *FlowFunction* of matching types. Then, in an LLVM Pass, the caller supplies a function and the aforementioned objects to **analyze**.

Adding an additional analysis to the framework amounts to providing new derived classes of *LatticePoint* and *FlowFunction*.

Unsure how honest to be here. Tradeoffs ended up being, how many C++ features can we avoid and still have a functioning C++ program? Other than this issue, describing our interface (flowfunction object, latticepoint object, and a blob of code implementing the worklist algorithm that uses runtime polymorphism to use these objects) is pretty straightforward. I (marco) can write this.

3 Analyses

3.1 Constant Propagation

3.1.1 Mathematical Flow Functions

define your lattice and flow functions in mathematical notation (i.e., in the style used in the lecture notes and on the midterm).

The constant propagation analysis is a 'must' analysis. Thus, we define the lattice to be $(D, \sqsubseteq, \top, \bot, \sqcup, \sqcap) = (2^A, \supseteq, A, \emptyset, \cap, \cup)$ where

$$A = \{x \to N \mid x \in \text{Vars} \wedge x \in \mathbb{Z}\}$$

.

The flow functions implemented in our framework are as follows:

$$F_{X:=Y\,\text{op}\,Z}(in) = in - \{X \to *\} \cup \{X \to N \mid (Y \to N_1) \in in \wedge (Z \to N_2) \in in \wedge N = N_1 \; op \; N_2\}$$

$$F_{if(X==C)\text{true-branch}}(in) = in - \{X \to *\} \cup \{X \to C\}$$

$$F_{if(X!=C)\text{false-branch}}(in) = in - \{X \to *\} \cup \{X \to C\}$$

At a join (i.e. PHI node), we take the intersection between the two incoming lattice points.

$$A \sqcup B = A \cap B$$

3.1.2 Implementation Considerations

discuss how you actually went about implementing the lattice and flow functions. For instance, some interesting questions are: what data structure(s) did you use, and how do you represent potentially infinite sets? How are input facts passed to your flow functions, and how do output facts get propagated?

3.2 Available Expressions

3.2.1 Mathematical Flow Functions

define your lattice and flow functions in mathematical notation (i.e., in the style used in the lecture notes and on the midterm).

3.2.2 Implementation Considerations

discuss how you actually went about implementing the lattice and flow functions. For instance, some interesting questions are: what data structure(s) did you use, and how do you represent potentially infinite sets? How are input facts passed to your flow functions, and how do output facts get propagated?

3.3 Range Analysis

3.3.1 Mathematical Flow Functions

Unlike the previous two analyses, range analysis is a ‘may’ analysis. Thus the most conservative range for a variable is the full set and the most optimistic is the empty set. With these considerations in mind, let us define our lattice and flow functions. The domain of the analysis is the set of maps from variables to the set of ranges specified by the extended integers (i.e. allowing $+\infty$ and $-\infty$). Thus,

$$D = \text{Vars} \to \mathbb{Z}^\infty \times \mathbb{Z}^\infty.$$

For $A \in D$ and $x \in \text{Vars}$, we will interchangeably think of $A(x) = [a, b]$ as a range (i.e. a set of numbers) and a tuple (a pair of numbers). With this reasoning, let top will correspond to the constant map of full sets, i.e. for any variable $x \in \text{Vars}$,

$$\top(x) = [-\infty, \infty].$$

Similarly, bottom corresponds to the constant map of empty sets, i.e. for any variable $x \in \text{Vars}$,

$$\perp(x) = [\infty, -\infty].$$

Here we take the convention that $[a, b] = \emptyset$ if and only if $b < a$. Given two elements of our domain, $A, B \in D$, we write $A \sqsubseteq B$ if and only if $A(x) \subseteq B(x)$ for all $x \in \text{Vars}$. We can also say for any $x \in \text{Vars}$,

$$A \sqcup B(x) = [\min\{\underline{A(x)}, \underline{B(x)}\}, \max\{\overline{A(x)}, \overline{B(x)}\}]$$

and

$$A \sqcap B(x) = [\max\{\underline{A(x)}, \underline{B(x)}\}, \min\{\overline{A(x)}, \overline{B(x)}\}]$$

where we take the convention that $\overline{[a, b]} = b$ and $\underline{[a, b]} = a$. Now that the lattice has been defined, let’s turn our attention to the various flow functions. To make the analysis more concrete, given an element $A \in D$ and variable x, let $A[x \to [a', b']]$ denote the exact same element of D with the exception that $A(x) = [a', b']$. Now we enumerate the various flow functions.

$$F_{X:=C}(in) = in[X \to [C, C]]$$

$$F_{X:=Y\,\mathrm{op}\,Z}(in) = in[X \to [\min L, \max L]] \text{ where } L = \{a \text{ op } b \,|\, a \in in(Y) \wedge b \in in(Z)\}$$

$$F_{if(X \le C) \text{ true-branch}}(in) = in[X \to [-\infty, C] \cap in(X)]$$

$$F_{if(X \le C) \text{ false-branch}}(in) = in[X \to [C+1, \infty] \cap in(X)]$$

$$F_{if(X == C) \text{ true-branch}}(in) = in[X \to [C, C] \cap in(X)]$$

$$F_{if(X == C) \text{ false-branch}}(in) = in$$

$$F_{if(X < C) \text{ true-branch}}(in) = in[X \to [-\infty, C-1] \cap in(X)]$$

$$F_{if(X < C) \text{ false-branch}}(in) = in[X \to [C, \infty] \cap in(X)]$$

$$F_{if(X \ge C) \text{ true-branch}}(in) = F_{if(X < C) \text{ false-branch}}(in)$$

$$F_{if(X \ge C) \text{ false-branch}}(in) = F_{if(X < C) \text{ true-branch}}(in)$$

$$F_{if(X > C) \text{ true-branch}}(in) = F_{if(X \le C) \text{ false-branch}}(in)$$

$$F_{if(X > C) \text{ false-branch}}(in) = F_{if(X \le C) \text{ true-branch}}(in)$$

$$F_{\text{merge}}(in_1, in_2) = in_1 \sqcup in_2$$

3.3.2 Implementation Considerations

discuss how you actually went about implementing the lattice and flow functions. For instance, some interesting questions are: what data structure(s) did you use, and how do you represent potentially infinite sets? How are input facts passed to your flow functions, and how do output facts get propagated?

LLVM has a nice data structure for handling ranges called ConstantRange. This is a one-side inclusive interval $[a, b)$ that is meant to be taken as intersection with the integers. ConstantRange handles empty sets by specifying that the interval $[a, a)$ is empty for all a and full-sets by only considering finite ranges as valid ranges. ConstantRange even has some convenience functions that make computing some of the above flow functions more bearable.

As with the above analyses, we explicitly specify whether or not a lattice point is top or bottom with boolean variables, and we use algebraic identities when working with these special cases. For the rest of the cases, a range analysis lattice point is represented by a map from LLVM Value pointers to ConstantRange pointers. Since this is an optimistic 'may' analysis, we use the convention that variables that are not in our map have the empty range.

3.4 Intra-Procedural Pointer Analysis

3.4.1 Mathematical Flow Functions

define your lattice and flow functions in mathematical notation (i.e., in the style used in the lecture notes and on the midterm).

3.4.2 Implementation Considerations

discuss how you actually went about implementing the lattice and flow functions. For instance, some interesting questions are: what data structure(s) did you use, and how do you represent potentially infinite sets? How are input facts passed to your flow functions, and how do output facts get propagated?

4 Testing

Make sure to explain assumptions you make about the code you analyze. For instance, for pointer analysis, you may have made some assumptions about the aliasing information known about input parameters. Explain those assumptions and why they are reasonable.

Part of this project is to come up with a useful set of benchmarks on which to test and improve your analysis. Discuss why you chose those benchmarks, and what makes them interesting. If your implementation fails on some benchmarks (there's no shame in it!), then explain why and how the analysis might be improved.

The difficulty of getting the various analyses to work properly on a piece of code is tightly coupled with the complexity of the underlying control flow graph.

Pathologies that underly implementations of flow functions may not arise in straight-line programs but become painfully obvious when branches are introduced. Our benchmarks are designed to illustrate that our analyses are robust to non-trivial control flow structures.

Broadly speaking, we have three types of benchmark. The first type is straight-line programs, which introduce no branches in control structure. They are the easiest to handle, and our analyses are accordingly precise on them. Simple branching programs are the second type, and they introduce conditional branches into fold, but do not exhibit loops. They are slightly more challenging, but SSA makes them much easier to handle. Our final type of benchmark is looping programs. As their name suggests, they have loops, which makes precision quite difficult.

4.1 Benchmarks/Assumptions in Common

Because we have a common pool of benchmarks, we can list them here along with common assumptions on code. More specialized discussion of per-analysis benchmark goes below. Here, we can also introduce the Straight Line Program/Branching Program distinction.

4.2 Constant Propagation

4.3 Available Expressions

4.4 Range Analysis

4.5 Intra-Procedural Pointer Analysis

5 Conclusion/Challenges

As this project is significantly more exploratory than the first, we want you to tell us what you found particularly interesting/challenging/frustrating. What extensions to the project did you attempt? (for instance, did you try combining the results of analyses, or did you try your hand at interprocedural analysis?). What aspects of LLVM made it easy/hard to implement your design? If you could redo your project with what you know now, what changes would you make?

Also unsure how honest to be here. Most of our challenges had to do with counter-intuitive LLVM design and poor documentation, as well as the unpredictability of C++ features.