Using Gated SSA Form to Determine Function Equivalence

Matthew Plant
maplant2@illinois.edu

1 Introduction

Computing systems are becoming increasingly complicated. Often times systems are composed of a large amount of similar or repeated code. Even worse, code is often written by disparate developers with differing code styles that make it hard to analyze with traditional methods. With complicated full systems being ported to smaller more embedded spaces it is beneficial to reduce the amount of redundant code to keep these systems as small as possible.

To that end we would like to determine whether or not two functions are equivalent with as much accuracy as possible. If we are able to determine two functions are equivalent we can merge them and reduce the size of our codebase.

Function equivalence is concerned with determining if two functions result in the same output given the same inputs. There are two general methods for this:

- Syntactic equivalence is concerned with determining if two functions have equivalent instructions.
- Semantic equivalence is concerned with determining if two functions have the same effect, regardless of instructions.

We present a method for determining semantic equivalence of functions expressed in SSA form by adding gating functions to phi nodes.

SSA is not suitable for comparing code for equivalence because ϕ-nodes are not referentially transparent[2]. The result of a ϕ-node depends not only on its inputs but on which edge control flow is transferred from. Gated SSA is a form of SSA that makes ϕ-nodes referentially transparent. GSA was originally designed to facility symbolic analysis making it an appropriate tool for comparing functions.

2 Syntactic equivalence

The state of the art in determining function equivalence is almost entirely concerned with syntactic equivalence.

One method introduced by Debray[3] is to determine syntactic equivalence of code that has already been compiled. Basic blocks are extracted from the code and continually compared with others using a fingerprinting method.

Fingerprinting involves producing a value for a function such that two functions with different semantics have different values. Functions with the same fingerprints are possibly equivalent, while functions with differing fingerprints are not equivalent.

Fingerprinting is useful as an optimization. In the case of Debray, a fingerprint is a 64-bit value formed by concatenating 4-bit encodings of the first 16 instructions in the block. Then, basic blocks with the same fingerprint are tested for equivalence by straight-line comparison of their instructions.

Another method for syntactic equivalence is present in the LLVM backend as an optional transform, called function merging[6]. This method involves two optimizations. First, whole functions are fingerprinted by the number of arguments the function takes plus if the function takes a variable number of arguments. This is an effective scheme because functions that have different number of arguments cannot be equivalent. The function's signature essentially becomes a tool we can use to quickly rule out functions that are not equivalent.

The second optimization and the method for determining syntactic equivalence is to impose a total order on functions. This is done by considering a basic block as a vector of values that can be compared, somewhat arbitrarily, as being greater than, less than, or equal to each other. Thus in order to determine the order of two basic blocks their instructions are compared lexicographically. If the two blocks have the

same instructions, they are equivalent basic blocks. Otherwise an arbitrary ordering on the instructions will determine the ordering of the basic blocks.

Lexicographically comparing and ordering basic blocks is an interesting optimization because it allows for a basic block to be merged with a sorted list of other basic blocks in $\log(n)$ time.

3 Problems with syntactic equivalence

Syntactic equivalence is fast but unfortunately not very suitable in a lot of cases. Many times the order a series of instructions are performed in a function can be swapped arbitrarily without any effect on the end computation. For example, consider the following two equivalent functions:

```
F1(%a0, %a1):
        %t0 = %a0 * 3
        %t1 = %a1 * 4
        ret %t0 + %t1

F2(%a0, %a1):
        %t0 = %a1 * 4
        %t1 = %a0 * 3
        ret %t0 + %t1
```

Here the order of instructions and their names have changed but the semantics of the functions have not. One possible approach to proving these functions as equivalent is to inspect their def/use chains.

For F1:

Definition:	Use:
%a0	%a0 * 3
%a1	%a1 * 4
%a0 * 3	ret %t0 + %t1
%a1 * 4	ret %t0 + %t1

For F2:

Definition:	Use:
%a0	%a0 * 3
%a1	%a1 * 4
%a0 * 3	ret %t0 + %t1
%a1 * 4	ret %t0 + %t1

We can see that these tables are nearly identical. If we trace paths of execution - instructions that modify the arguments and return a value - there is no difference between F1 and F2.

Although in this example it is suitable syntactic equivalence of def/use chains is also problematic in that for the most part it ignores control flow completely. To this end, we need to find a way to ensure that control flow is preserved without having to explicitly determine equivalent flow graphs.

4 Motivating example

To understand our algorithm, we present two equivalent functions that are often seen in production that cannot be proven to be equivalent syntactically:

```
Min1(%a0, %a1):
    %cond = icmp gt %a0, %a1
    br i1 %cond, label greater, label less
greater:
        %t0 = %a1
    br label done
less:
    %t1 = %a0
done:
    %t3 = φ %t1, %t0
    ret %t3

Min2(%a0, %a1):
    %cond = icmp lt %a0, %a1
    br i1 %cond, label less, label greater
less:
    %t0 = %a0
    br done
greater:
    %t1 = %a1
done:
    %t3 = φ %t1, %t0
    ret %t3
```

Despite these functions being obviously equivalent, they are in no way syntactically equivalent. Additionally, neither are the def/use chains. This is because while `%cond` is only used to determine which of t0 and t1 is selected, it is a bonafide definition with uses.

5 Gated SSA

In order to prove that these functions are equivalent we would like to trace paths of effect while capturing control flow.

We model our programs so that for any conditional both branches are taken in parallel. Then, when we reach a phi node the value belonging to the taken branch is selected based on the result of the conditional expression. We can express this model by converting SSA into Gated SSA form and removing branching instructions.

Gated SSA is a way of ensuring that def use chains preserve control flow. It is a form typically used to perform symbolic analysis. One such example is to ensure that a function transformations preserve semantics. To do this, one needs to prove a theorem of the form

$$\forall p. \vdash [p] \equiv [\Upsilon(p)]$$

where p is some program and $\Upsilon(p)$ is transformation that preserves semantics [4]. If we frame Υ as the variations present in differing programming styles, we can use Gated SSA as a way to more accurately prove functions from differing code styles - and thus, differing code bases - to be equivalent. Our motivating example presents one such example in which a simple change in source code can lead to an incorrect analysis.

Gated SSA is very similar to SSA, with the only modification being in how we identify ϕ-nodes. Semantically these nodes do not need to change, but they need to be extended in a way so that comparisons between other ϕ-nodes preserves control flow information. In other words, we need the equivalence of ϕ-nodes to be necessarily dependent on the equivalence of CFGs.

To this end we extend our definition of ϕ-nodes to include the condition that results in each argument - a branch in the CFG - to be taken. You can imagine that for a simple if statement taken on condition c that the resulting syntax for a ϕ-node would be:

$$\phi(c \to x, !c \to y)$$

Trouble arises with ϕ-nodes present in loops. The naive method would be to unroll the loop infinitely until we find a (possibly infinite) recurrence pattern. To keep things finite and computable, we define η- and μ-nodes.

μ-nodes are used to define variables modified within a loop. The first parameter is the initial value while the second parameter represents subsequent iterations. η-nodes are used outside of loops to refer to loop defined variables. η-nodes carries the variable being referred to with the condition required to reach the η-node.

Here is an example of a regular SSA converted to GSA.

Regular:

```
    %x_0 = %c
loop:
    %x_p = φ(%x_0, %x_k)
    %b = x_p < n
    br i1 %b, label loop1, label exit
loop1:
    %x_k = %x_p + 1
    br label loop
exit:
    %x = %x_p
```

GSA:

```
    %x_0 = %c
loop:
    %x_p = μ(%x_0, %x_k)
    %b = x_p < n
    br i1 %b, label loop1, label exit
loop1:
    %x_k = %x_p + 1
    br label loop
exit:
    %x = %ν(b, x_p)
```

6 Converting SSA to GSA

In order to convert SSA form we must determine which ϕ-nodes must be converted to η- and μ-nodes and find for all the ϕ-nodes that remain the gating function for each parameter (i.e., which conditions apply to which parameters). According to [4], the problem of computing gating functions is a single-source path expression problem. Therefore, in order to convert SSA into GSA, we need to computer the single-source path expression for the CFG of the function we are trying to convert.

7 Determining function equivalence with GSA

Using GSA we can def/use chains along paths of effect. It is thus important to define what we consider a path of effect and ensure that our model aligns well with real world code.

We define a path of effect as a def/use chain that ends in some notionally unsafe instruction. For example, we can define such instructions as returns and stores. If functions have side effects then function calls can be added to that list.

8 Analyzing our motivating example

Now that we have a method for proving our motivating functions as equivalent we can convert them to GSA form:

```
Min1(%a0, %a1):
    %cond = icmp gt %a0, %a1
    br i1 %cond, label greater, label less
greater:
        %t0 = %a1
    br label done
less:
    %t1 = %a0
done:
    %t3 = φ  %a0 > %a1  → %t0,
           !(%a0 > %a1) → %t1
    ret %t3

Min2(%a0, %a1):
    %cond = icmp lt %a0, %a1
    br i1 %cond, label less, label greater
less:
    %t0 = %a0
    br done
greater:
    %t1 = %a1
done:
    %t3 = φ  %a0 < %a1  → %t0,
           !(%a0 < %a1) → %t1
    ret %t3
```

Now that our code is in the correct form we can analyze the def/use chains along paths of effect:

For Min1:

Definition:	Use:
%a0	%t1
%a1	%t0
%t0	ϕ a0 > a1 → t0, !(a0 > a1) → t1
%t1	ϕ a0 > a1 → t0, !(a0 > a1) → t1

For Min2:

Definition:	Use:
%a0	%t0
%a1	%t1
%t0	ϕ a0 < a1 → t0, !(a0 < a1) → t1
%t1	ϕ a0 < a1 → t0, !(a0 < a1) → t1

The phi nodes are equivalent, but we need a way to prove that.

9 Using simple algebraic properties to express human variation

Often times human program vary in small mathematical differences that end up being equivalent. by accumulating a set of renaming rules we can express the human transformation of code and determine equivalence of a more broad class of functions.

By renaming expressions based on algebraic properties we can often prove that two expressions are equivalent.

In this case we can use the converse property of inequalities for integers to prove that our phi nodes are equivalent:

Min1:

$$\begin{aligned}\phi(a_0 > a_1 \to t_0, !(a_0 > a_1) \to t_1) = \\ \phi(a_0 > a_1 \to t_0, a_0 \leq a_1 \to t_1) = \\ \phi(a_0 > a_1 \to a_1, a_0 \leq a_1 \to a_0)\end{aligned}$$

Min2:

$$\begin{aligned}\phi(a_0 < a_1 \to t_0, !(a_0 < a_1) \to t_1) = \\ \phi(a_0 < a_1 \to t_0, a_0 \geq a_1 \to t_1) = \\ \phi(a_0 < a_1 \to a_0, a_0 \geq a_1 \to a_1)\end{aligned}$$

As a special case we can rewrite the phi as a three-branch conditional:

Min1:

$$\phi(a_0 > a_1 \to a_1, a_0 < a_1 \to a_0, a_0 = a_1 \to a_0)$$

Min2:

$$\phi(a_0 < a_1 \to a_1, a_0 > a_1 \to a_1, a_0 = a_1 \to a_0)$$

We can see that the arguments for both of the ϕ-nodes are equivalent and differ only in their order.

The question arises of how we decide whether or not to use algebraic properties, and which to use. Firstly, we only ever need to apply these rules when two links in a def/use chain are being compared are naively determined to be nonequivalent. From there, we can essentially permute all of the possible rewritings of one link in the chain until we find that one is equal to the other or not.

Most of the time this can be avoided by letting the compiler perform canonicalization on expressions. With this performed, we only need to perform renaming on nonequivalent phi nodes.

Since phi nodes take boolean conditionals, the problem of determining the equivalence of phi nodes reduces to the boolean satisfiability problem. However, if we wish to avoid NP-completeness at the cost of reduced accuracy, we can perform simple renaming rules and avoid conditionals that are too complex to be solved quickly.

10 Supporting instructions and function calls with side effects

Often times we cannot suppose that instructions or function calls are pure. For those instructions that are not safe we must ensure that the order they are called is preserved.

The first transformation that needs to be applied is to number all unsafe functions and instructions by the order they are called. This way a value returned by an unsafe function is dependent on the relative order of unsafe functions

This transformation is useful for preserving the notion of order in instruction and function calls without introducing extra values and thus more def/use chains.

For example, this transformation is necessary to prove that the following two functions are not equivalent:

```
Func1():
        %t0 = call UnsafeFunc()
        %t1 = call UnsafeFunc()
        ret %t0

Func2():
        %t0 = call UnsafeFunc()
        %t1 = call UnsafeFunc()
        ret %t1
```

If the unsafe function being called were in fact safe it would be clear that these two functions are equivalent. However, because they are not, we need to apply this transformation to show their nonequivalence:

```
Func1():
        %t0 = call UnsafeFunc_1()
        %t1 = call UnsafeFunc_2()
        ret %t0

Func2():
        %t0 = call UnsafeFunc_1()
        %t1 = call UnsafeFunc_2()
        ret %t1
```

With this transformation applied unsafe functions become sufficiently ordered to reflect the unsafe semantics of an imperative language.

The second transformation is to add a def/use chain for any unsafe instruction that lacks representation as a use. Some unsafe functions do not take any arguments and thus do not appear in any def/use chain. While their order is preserved the primary fact that they are called at all is not. Therefore we must artificially add a chain that includes these instructions as a use. One method to do this to add as the first parameter to every function a typeless "state" variable that is simply passed to other functions.

For example, given a function that only calls an unsafe function that has no return value:

```
Func():
        call UnsafeFunc()
```

There is no def/use chain that contains the call to `UnsafeFunc`. The way we solve this is to add a superfluous "state" argument that only serves to add a suitable def/use chain. The end result is:

```
Func(%s):
        call UnsafeFunc(%s)
```

In practice this transformation does not really need to take place, so long as our equivalence analysis makes sure to consider unsafe function calls with no arguments as a use of entering the function.

11 Complexity analysis of GSA transformation and equivalence step

Computing the GSA form of a function requires computing the single source path expression[4] rooted at the entry block for a control flow graph. For reducible flow graphs, the smallest known time complexity is $O(m * \alpha(m, n))$ where m is the number of edges, n is the number of vertices, and α is the inverse Ackerman function[5].

Additional time is required to prove instruction equivalence, but this step is no more expensive than traditional syntactic equivalence checkers. There are many opportunities to optimize this step, such as by capturing the fingerprint of basic blocks and eliminating comparisons a priori.

Renaming instructions to determine equivalence also requires more time, but the amount of time allocated to this step is completely dependent on how rigorous of a checker the user wants to create. A less accurate checker will attempt to permute instructions fewer times or not at all while a more accurate checker will permute all possible expressions derived from an instruction.

12 Status and future work

There is software available[1] that converts the most well known SSA format - LLVM bitcode - to a suitable GSA format we can use for equivalence checking.

Given two functions in GSA format we can compare them with the same amount of accuracy as traditional syntactic comparisons.

More work needs to be done in exploring how renaming rules can improve accuracy of function equivalence and its impact on code compaction. Additionally we are interested in how this method could be a applied to a system wide approach where functions across all abstraction layers.

13 Conclusion

Gated SSA is a much more accurate method for proving function equivalence than current methods. While it is ultimately slower than the state of the art it can prove function equivalence even in functions with differing control flows.

References

[1] `http://www.http://llvm-md.org/`

[2] Paul Havlak. Construction of Thinned Gated Single-Assignment Form. *In Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 477–499. Springer Verlag, 1993.

[3] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *CM Transactions on Programming Languages and Systems (TOPLAS'00)*, Vol. 22(2):378-415, March 2000.

[4] Paul Govereau. Denotational Translation Validation. Doctoral dissertation, Harvard University. `http://nrs.harvard.edu/urn-3:HUL.InstRepos:10121982` 2012.

[5] Robert Tarjan. Fast algorithms for solving path problems. *J. ACM* **28** (3) pages 594-614. 1981,

[6] LLVM. MergeFunctions pass, how it works. `http://llvm.org/docs/MergeFunctions.html`