One Representation to Rule Them All

Combining analyses on SSA with On-Demand SSA Construction

Paul Biggar *
Trinity College Dublin
pbiggar@cs.tcd.ie

David Gregg
Trinity College Dublin
david.gregg@cs.tcd.ie

1. The elephant in (SSA's) room

Static single assignment form (SSA) [5] is nearly ubiquitous in the compiler world. It is dearly loved by most compiler writers, and even more so by undergraduate compiler-class instructors. Its popularity comes from a number of powerful features:

- It fits neatly into a 45 minute exam question.
- It provides flow-sensitivity for free.
- It adds sparseness to analyses, which can greatly reduce their run-time (especially for propagation algorithms).
- The single assignment property greatly reduces memory usage, compared to traditional bit-vector analyses.
- Factored use-def chains mitigate the explosion in memory use of use-def chains in certain circumstances.

In an ideal world, every compiler could use an end-to-end SSA representation, from just after parsing [2], right the way through to code generation [7].

1.1 Problem

But a great big elephant sits in the room: you can't just go straight into SSA form. In *real* compilers/languages, some form of alias analysis must be performed before SSA construction. Even in representations that incorporate alias analysis results into SSA, such as HSSA[1] [4], the alias analysis runs first, as a separate pass. As a result, the alias analysis **cannot** run on SSA, and cannot benefit from its properties.

There have been a few nice ideas for more precise alias analysis recently. Our favourite, from Pioli and Hind [11], avoids analysing unrealizable paths by performing Conditional Constant Propagation (CCP) simultaneously with alias analysis. However, this requires moving another analysis out of SSA form, reducing further its benefits.

One of the most elegant features of SSA is that it allows a unified value propagation framework. Sparse Conditional Constant Propagation (SCCP) [14] provides a general framework for efficient value propagation through symbolic execution, which is strictly more powerful than a combination of unreachable-code elimination and simple constant propagation. This framework has also been shown to work for value-range propagation (VRP) [10] and type-inference [8]. In an ideal world, call-hierarchy analysis [6], string-range propagation [13], and many value propagation analyses would be performed within the same framework.

But here everything falls apart. If CCP is to be performed alongside alias analysis, the sparse form cannot be used. This can make it dangerous to use an analysis that uses a deep lattice, such as VRP. The efficient representation is also lost, as a value must be stored for each program point rather than for each unique definition in a function. For algorithms which store values, like CCP, this can be expensive.

Existing solutions to these problems are awful. As well as being less powerful than Pioli's solution, iterating over the analyses (i.e. SSA $\rightarrow$ alias analysis $\rightarrow$ SSA $\rightarrow$ ...), leads to, well, iteration. Worse still, there are languages like PHP which make it impossible to create a non-pessimistic conservative estimate of a program's aliasing, making an iterative solution impossible.

1.2 Solution

The dream is to somehow combine all of these analyses. SSA construction could use the results of an alias analysis which ran on SSA form. If the phase ordering problems went away, all of this would be possible.

What if we could combine SSA construction with the analyses that use it? Essentially, we would construct SSA form *on-demand*: simultaneously with alias analysis, SCCP, and other analyses which use the SCCP algorithm. In effect, SSA construction would be just another client of the SCCP algorithm.

This would achieve the precision of Pioli, the speed and memory efficiency of SCCP, and would not require iteration. Most importantly, it would allow a "real" compiler to have an end-to-end SSA representation.

To realize this, there are three requirements:

1. It must be possible to construct SSA using an SCCP framework.
2. We must show the results of alias analysis can be merged into the SSA construction algorithm as it runs.
3. It must be possible to run each of these simultaneously without error.

Below, we start this off by providing some of point 1, by describing an algorithm for SSA construction via symbolic execution. It is completely unproven, and barely tested, but it looks right. We prove points 2 and 3 by hand waving, since they sound like they might work.

* Thanks to the Irish Research Council for Science, Engineering and Technology funded by the National Development Plan, whose funding made this work possible.

[1] Hashed-SSA adds χ and μ nodes, in addition to ϕ nodes, to represent possible indirect definitions and uses due to aliasing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FIT (PLDI) '09 16 June 2009, Dublin, Ireland.

2. Algorithm

This section provides, quite seriously, our algorithm. It should probably be skipped on first, and possibly subsequent, readings.

2.1 SCCP algorithm

We present a generalization of the SCCP algorithm [14]:

1. Use two worklists, one for CFG edges (from one basic block to another), and one for SSA edges (from a definition to a use).
2. Starting at the entry node of the CFG worklist, analyse each statement s of a basic block bb, then:
   - For each variable definition d in s, add all uses of d to the SSA worklist,
   - Add each successor $succ$ of bb to the CFG worklist, only if $succ$ is executable according to the analysis so far.
3. Once the CFG worklist is exhausted, analyse each statement s which is the target of an edge in the SSA worklist, then:
   - As above, for each variable definition d in s, add all uses of d to the SSA worklist.

2.2 SSA construction algorithm

This generalized form of the SCCP algorithm can be used for many client analyses. We extend it to allow SSA construction via symbolic analysis, in the following way:

2.2.1 Assumptions

- Dominance information (the immediate dominator and dominance frontiers of each basic block) is available.
- All variables in our program begin out of SSA form, and ϕ nodes are not initially present anywhere in the program.
- When ϕ nodes are added, they are placed as the first statements in the basic block.

2.2.2 Algorithm

Following the SCCP algorithm, for each statement s in a basic block bb:

1. For each use u in s:
   - Fetch an SSA version for u by searching upwards through the dominance tree of bb for definitions of u:
     - If no definitions are found, the zero version if chosen,
     - If s is a ϕ node, we begin the search in the predecessor of bb associated with u, rather than bb itself,
   - If u already has a version, we must still recalculate it, since a definition[2] may have been inserted in between s and the previous definition of u.
2. For each def d in s:
   - Add an unversioned ϕ node p, named after d, to each dominance frontier of b, and add p to the SSA worklist,
   - Give a new version to d.
   - Add the following uses of d to the SSA worklist:
     - The unversioned d, from before SSA construction, as they might not all have been reached,
     - The version of d which holds just before the new definition, as this s may be a new statement inserted on an existing path between some definition of d and its use.

[2] Due to the addition of a ϕ node, or a χ node in HSSA form.

3. Future Work

While this is an exciting idea, there are a number of steps before this research can be considered to be completed:

- We don't really know if this works in general.
- We haven't thought about complexity.
- Searching for a version could do with some caching.
- We have not really considered any of pruned, semi-pruned or minimal forms.
- Pruned SSA form uses liveness information, which doesn't fit nicely into the SCCP framework.
- In some languages, exception conditions may need to be conservatively calculated. This might also be incorporated into the SCCP framework.

4. Related Work

There are a slew of other SSA construction algorithms [1, 2, 3], including one which looks like it might solve our problem [12]. In fact, none are suitable. However, we note the existence of an unpublished algorithm in the Firm compiler [9], which constructs a CFG simultaneously with SSA construction from Java bytecodes. Although our algorithms are similar (especially the *get_value* and *maturity* abstractions), their algorithm lacks the ordering which makes it suitable to be used with the SCCP algorithm. We believe that they might be usefully combined in the future.

5. Conclusion

It looks like it should work, it solves a great and pressing problem, and we have an algorithm that looks right.

References

[1] J. Aycock and R. N. Horspool. Simple generation of static single-assignment form. In *CC '00*.

[2] M. M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *TOPLAS*, 1994.

[3] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28(8), 1998.

[4] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, 1996.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4), 1991.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95*.

[7] S. Hack and G. Goos. Optimal register allocation for SSA-form programs in polynomial time. *Inf. Process. Lett.*, 98(4), 2006.

[8] A. Lenart, C. Sadler, and S. K. S. Gupta. SSA-based flow-sensitive type analysis: combining constant and type propagation. In *SAC '00*.

[9] G. Lindenmaier. libFIRM – A library for compiler optimization research implementing FIRM. Technical Report 2002-5, Sep 2002.

[10] J. R. C. Patterson. Accurate static branch prediction by value range propagation. In *PLDI '95*.

[11] A. Pioli. Conditional pointer aliasing and constant propagation. Technical report, Master's thesis, SUNY at New Paltz, 1999.

[12] H. Saito and C. D. Polychronopoulos. sigma-SSA and its construction through symbolic interpretation. In *LCPC '96*.

[13] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07*.

[14] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2), 1991.