Equality-Based Translation Validator for LLVM

Michael Stepp, Ross Tate, and Sorin Lerner

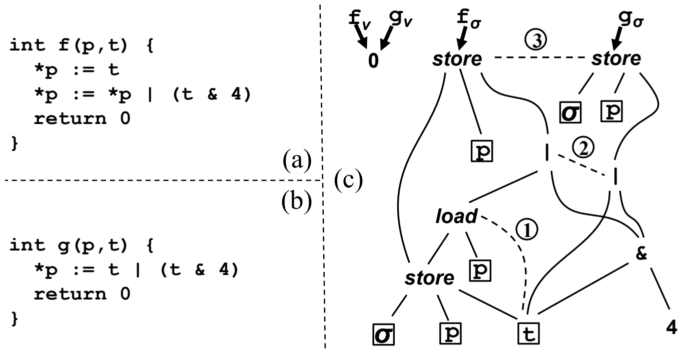
University of California, San Diego
{mstepp,rtate,lerner}@cs.ucsd.edu

Abstract. We updated our Peggy tool, previously presented in [7], to perform translation validation for the LLVM compiler using a technique called Equality Saturation. We present the tool, and illustrate its effectiveness at doing translation validation on SPEC 2006 benchmarks.

1 Introduction

Compiler optimizations have long played a crucial role in the software ecosystem, allowing programmers to express their programs at higher levels of abstraction without paying the performance cost. At the same time, however, programmers also expect compiler optimizations to be correct, in that they preserve the behavior of the programs they transform. Unfortunately, this seemingly simple requirement is hard to ensure in practice. One approach to improving the reliability of compiler optimizations is a technique called *Translation Validation* [4]. After each run of the optimizer, a separate tool called a translation validator tries to show that the optimized program is equivalent to the corresponding original program. Therefore, a translation validator is just a tool that tries to show two programs equivalent, one of the earliest examples of which can be found in [5]. In our own previous work [7], we developed a technique for reasoning about program equivalence called *Equality Saturation*, and a tool called Peggy that implements this technique. Although our paper focused mostly on performing compiler optimizations using Peggy, we also showed how Equality Saturation can be used to perform translation validation, and that Peggy is an effective translation validator for Soot, a Java bytecode-to-bytecode optimizer.

Inspired by the recent results of Tristan, Govereau and Morrisett [8] on translation validation for LLVM [1], we have updated our Peggy tool so that it can be used to perform translation validation for LLVM, a more aggressive and more widely used compiler than Soot. This updated version of Peggy reuses our previously developed equality saturation engine described in [7]. However, we had to develop several new, LLVM-specific components for Peggy: an LLVM frontend for the intermediate representation used by our equality-saturation engine; new axioms for our engine to reason about LLVM loads, stores and calls; and an updated constant folder that takes into account LLVM operators. Finally, we present new experimental results showing the effectiveness of Peggy at doing translation validation for LLVM on SPEC 2006 C benchmarks.



```
int f(p,t) {
  *p := t
  *p := *p | (t & 4)
  return 0
}
```

```
int g(p,t) {
  *p := t | (t & 4)
  return 0
}
```

Fig. 1. (a) Original code (b) Optimized code (c) Combined E-PEG.

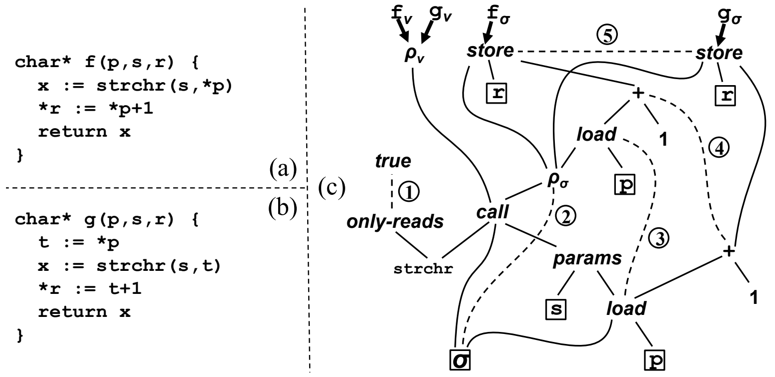
2 Overview

We first present several examples demonstrating how Peggy performs translation validation. These examples are distilled versions of real examples that we found while doing translation validation for LLVM on SPEC 2006.

Example 1. Consider the original code in Figure 1(a) and the optimized code in Figure 1(b). There are two optimizations that LLVM applied here. First, LLVM performed copy propagation through the location `*p`, thus replacing `*p` with `t`. Second, LLVM removed the now-useless store `*p := t`.

Our approach to translation validation uses a representation that we developed and presented previously called Program Expression Graphs [7] (PEGs). A PEG is a pure functional representation of the program which has nice properties for reasoning about equality. Figure 1(c) shows the PEGs for `f` and `g`. The labels $\mathtt{f}_v$ and $\mathtt{f}_\sigma$ point to the value and heap returned by `f` respectively, and likewise for `g` – for now, let us ignore the dashed lines. In general, a PEG is a (possibly cyclic) expression graph. The children of a node are shown graphically below the node. Constants such as 4 and 0 have no children, and nodes with a square around them represent parameters to the function and also have no children. PEGs encode the heap in a functional way using the standard operators *load* and *store*. In particular, given a heap σ and a pointer p, $load(\sigma, p)$ returns the value at address p; given a heap σ, a pointer p, and a value v, $store(\sigma, p, v)$ returns a new heap identical to σ, except that the value at address p is v. The PEG for `f` takes a heap σ as an input parameter (in addition to `p` and `t`), and produces a new heap, labeled $\mathtt{f}_\sigma$, and a return value, labeled $\mathtt{f}_v$.

Our approach to translation validation builds the PEGs for both the original and the optimized programs in the same PEG space, meaning that nodes are reused when possible. In particular note how `t & 4` is shared. Once this combined PEG has been constructed, we apply equality saturation, a process by which equality axioms are repeatedly applied to infer equalities between PEG nodes. If through this process Peggy infers that node $\mathtt{f}_\sigma$ is equal to node $\mathtt{g}_\sigma$ and that node



```
char* f(p,s,r) {
  x := strchr(s,*p)
  *r := *p+1
  return x
}
```

```
char* g(p,s,r) {
  t := *p
  x := strchr(s,t)
  *r := t+1
  return x
}
```

Fig. 2. (a) Original code (b) Optimized code (c) Combined E-PEG.

$\mathtt{f}_v$ is equal to node $\mathtt{g}_v$, then Peggy has shown that the original and optimized functions are equivalent. In the diagrams we use dashed lines to represent PEG node equality (in the implementation, we store equivalence classes of nodes using Tarjan's union-find data structure). Note that E-PEGs are similar to E-graphs from SMT solvers like Simplify [3] and Z3 [2], but specialized for representing programs and with algorithms specialized to handle cyclic expressions which arise much more frequently in E-PEGs than in typical SMT problems.

Peggy proves the equivalence of `f` and `g` in the following three steps:

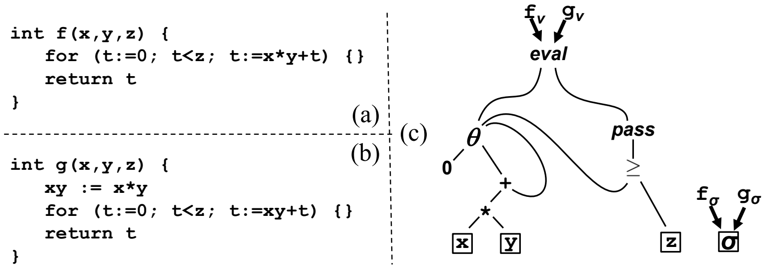
Peggy adds equality① using axiom: $\mathit{load}(\mathit{store}(\sigma, p, v), p) = v$
Peggy adds equality② by congruence closure: $a = b \Rightarrow f(a) = f(b)$
Peggy adds equality③ by axiom: $\mathit{store}(\mathit{store}(\sigma, p, v_1), p, v_2) = \mathit{store}(\sigma, p, v_2)$

By equality ③, Peggy has shown that `f` and `g` return the same heap, and are therefore equivalent since they are already known to return the same value 0.

Example 2. As a second example, consider the original function `f` in Figure 2(a) and the optimized version `g` in Figure 2(b). `p` is a pointer to an `int`, `s` is a pointer to a `char`, and `r` is a pointer to an `int`. The function `strchr` is part of the standard C library, and works as follows: given a string `s` (i.e., a pointer to a `char`), and an integer `c` representing a character[1], `strchr(s,c)` returns a pointer to the first occurrence of the character `c` in the string, or `null` otherwise. The optimization is correct because LLVM knows that `strchr` does not modify the heap, and the second load `*p` is redundant.

The combined PEGs are shown in Figure 2(c). The call to `strchr` is represented using a *call* node, which has three children: the name of the function, the incoming heap, and the parameters (which are passed as a tuple created by the *params* node). A *call* node returns a pair consisting of the return value and

[1] It may seem odd that `c` is not declared a `char`, but this is indeed the interface.



```
int f(x,y,z) {
   for (t:=0; t<z; t:=x*y+t) {}
   return t
}
```

```
int g(x,y,z) {
   xy := x*y
   for (t:=0; t<z; t:=xy+t) {}
   return t
}
```

Fig. 3. (a) Original code (b) Optimized code (c) Combined E-PEG.

the resulting heap. We use projection operators ρ_v and ρ_σ to extract the return value and the heap from the pair returned by a *call* node.

To give Peggy the knowledge that standard library functions like `strchr` do not modify the heap, we have annotated such standard library functions with an *only-reads* annotation. When Peggy sees a call to a function foo annotated with *only-reads*, it adds the equality *only-reads*(foo) = *true* in the PEG. Equality① in Figure 2(c) is added in this way.

Peggy adds equality② using: *only-reads*(n) = *true* $\Rightarrow \rho_\sigma(call(n, \sigma, p)) = \sigma$. This axiom encodes the fact that a read-only function call does not modify the heap. Equalities③,④, and⑤ are added by congruence closure.

In these 5 steps, Peggy has identified that the heaps $\texttt{f}_\sigma$ and $\texttt{g}_\sigma$ are equal, and since the returned values $\texttt{f}_v$ and $\texttt{g}_v$ are trivially equal, Peggy has shown that the original and optimized functions are equivalent.

Example 3. As a third example, consider the original code in Figure 3(a) and the optimized code in Figure 3(b). LLVM has pulled the loop-invariant code `x*y` outside of the loop. The combined PEG for the original function `f` and optimized function `g` is shown in Figure 3. As it turns out, `f` and `g` will produce the exact same PEG, so let us focus on understanding the PEG itself. The θ node represents the *sequence* of values that `t` takes throughout the loop. The left child of the θ node is the first element of the sequence (0 in this case) and the right child provides any element in the sequence after the first one in terms of the previous element. The *eval*/*pass* pair is used to extract the value of `t` after the loop. The $\geq$ node is a lifting of $\geq$ to sequences, and so it represents the sequence of values that `t` $\geq$ `z` takes throughout the loop. *pass* takes a sequence of booleans and returns the index of the first boolean in the sequence that is true. Therefore *pass* in this case returns the index of the last iteration of the loop. *eval* takes a sequence of values and an integer index, and returns the element of the sequence at that index. Therefore, *eval* returns the value of `t` after the last iteration of the loop (a denotational semantics of PEGs can be found in [7]).

As Figure 3 shows, the PEG for the optimized function `g` is the same as the original function `f`. Peggy has validated this example just by converting to PEGs, without even running equality saturation. One of the key advantages of PEGs is that they are agnostic to code-placement details, and so Peggy can validate code placement optimizations such as loop-invariant code motion, lazy code motion, and scheduling by just converting to PEGs and checking for syntactic equality.

3 Implementation

Axioms. Peggy uses a variety of axioms to infer equality information. Some of these axioms were previously developed and state properties of built-in PEG operators like θ, *eval*, *pass*, and ϕ (which are used for conditionals). We also implemented LLVM specific axioms to reason about *load* and *store*, some of which we have already seen. An additonal such axiom is important for moving unaliased loads/stores across each other: $p \neq q \Rightarrow load(store(\sigma, q, v), p) = load(\sigma, p)$.

Alias Analysis. The axiom above can only fire if $p \neq q$, requiring alias information. Our first attempt was to encode an alias analysis using axioms, and run it using the saturation engine. However, this added a significant run-time overhead, and so we instead took the approach from [8], which is to pre-compute alias information; we then used this information when applying axioms.

Generating Proofs. As described in our follow-up work [6], after Peggy validates a transformation it can use the resulting E-PEG to generate a proof of equivalence of the two programs. This proof has already helped us determine how often axioms are useful. In the future, we could also use this proof to improve the run-time of our validator: after a function f has been validated, we could record which axioms were useful for f, and enable only those axioms for subsequent validations of f (reverting back to all axioms if the validation fails).

4 Results

We used Peggy to perform translation validation for LLVM 2.8 on SPEC 2006 C benchmarks. We enabled the following optimizations: dead code elimination, global value numbering, partial redundancy elimination, sparse conditional constant propagation, loop-invariant code motion, loop deletion, loop unswitching, dead store elimination, constant propagation, and basic block placement.

Figure 4 shows the results: "#func" and "#instr" are the number of functions and instructions; "%success" is the percentage of functions whose compilation Peggy validated ("All" considers all functions; "OC", which stands for "Only Changed", ignores functions for which LLVM's output is identical to the input); "To PEG" is the average time per function to convert from CFG to PEG; "Avg Engine Time" is the average time per function to run the equality saturation engine ("Success" is over successful runs, and "Failure" over failed runs).

Overall our results are comparable to [8]. However, because of implementation differences (including the set of axioms), an in-depth and meaningful experimental comparison is difficult. Nonetheless, conceptually the main difference is that [8] uses axioms for destructive rewrites, whereas we use axioms

Benchmark	#func	#instr	%success		To	Avg Engine Time	
			All	OC	PEG	Success	Failure
`400.perlbench`	1,864	269,631	79.0%	73.3%	0.531s	1.028s	11s
`401.bzip2`	100	16,312	82.0%	76.9%	0.253s	0.733s	19s
`403.gcc`	5,577	828,962	80.8%	74.9%	0.558s	0.700s	19s
`429.mcf`	24	2,541	87.5%	87.0%	0.216s	0.500s	19s
`433.milc`	235	21,764	80.4%	75.0%	0.246s	0.188s	9s
`456.hmmer`	538	57,102	86.4%	84.6%	0.285s	0.900s	11s
`458.sjeng`	144	23,807	77.1%	72.5%	1.099s	0.253s	7s
`462.libquantum`	115	5,864	73.9%	64.3%	0.123s	0.167s	8s
`464.h264ref`	590	131,627	74.2%	70.5%	0.587s	0.846s	12s
`470.lbm`	19	3,616	78.9%	76.5%	0.335s	0.154s	3s
`482.sphinx3`	369	28,164	88.1%	86.0%	0.208s	0.480s	12s

Fig. 4. Results of running Peggy's translation validator on SPEC 2006 benchmarks.

to add equality information to the E-PEG, thus expressing multiple equivalent programs at once. Our approach has several benefits over [8]: (1) we simultaneously explore an exponential number of paths through the space of equivalent programs, whereas [8] explores a single linear path – hence we explore more of the search space; (2) we need not worry about axiom ordering, whereas [8] must pick a good ordering of rewrites for LLVM – hence it is easier to adapt our approach to new compilers, and a user can easily add/remove axioms (without worrying about ordering) to balance precision/speed or to specialize for a given code base; (3) our approach effectively reasons about loop-induction variables, which is more difficult using the techniques in [8]. However, the approach in [8] is faster, because it explores a single linear path through the space of programs.

Failures were caused by: (1) incomplete axioms for linear arithmetic; (2) insufficient alias information; (3) LLVM's use of pre-computed interprocedural information, even in intraprocedural optimizations. These limitations point to several directions for future work, including incorporating SMT solvers and better alias analyses, and investigating interprocedural translation validation.

References

1. The LLVM compiler infrastructure. `http://llvm.org`.
2. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
3. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
4. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.
5. H. Samet. Proving the correctness of heuristically optimized code. *Communications of the ACM*, 21(7):570–582, July 1978.
6. R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. In *POPL*, 2010.
7. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL*, 2009.
8. J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.