Semantic Equivalence Checking for HHVM Bytecode

Nick Benton
Facebook
London
pnb@fb.com

ABSTRACT

We describe a semantic differencing tool used to compare the bytecodes generated by two different compilers for Hack/PHP at Facebook. The tool is a prover for a simple relational Hoare logic for low-level code and is used in testing, allowing the developers to focus on semantically significant differences between the outputs of the two compilers.

CCS CONCEPTS

• **Theory of computation** → **Logic and verification**; **Hoare logic**; **Invariants**; **Pre- and post-conditions**; • **Software and its engineering** → **Software verification**; **Automated static analysis**; **Compilers**; **Software testing and debugging**; Object oriented languages;

ACM Reference Format:
Nick Benton. 2018. Semantic Equivalence Checking for HHVM Bytecode. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3236950.3236975

1 INTRODUCTION

HHVM, the HipHop Virtual Machine [14], is a high performance, JIT-compiled implementation of PHP. HHVM generates the Facebook site, as well as being used by Wikipedia, Slack, and others. Most of the code running on HHVM at Facebook is now written in Hack, a new language based on PHP, whose features include an expressive gradual type system, improved collections, lambdas, and support for asynchronous programming [16]. Hack was originally implemented using a combination of an independent typechecker plus some extensions to HHVM's PHP front-end. We have now built HackC, a new compiler for Hack. HackC emits HHAS, which is a textual form of HHBC, the HipHop ByteCode already used as an intermediate language inside HHVM.

Moving to an entirely new compiler for Hack code is not something to be undertaken lightly. The existing HHVM implementation is mature and well-tested, and we needed to have a similar level of confidence in the new compiler. A conservative decision was therefore taken that, in the first instance, the new compiler should produce the same target code as the old one for our existing codebase and tests.

But what do we mean by 'the same'? Just running `diff` on the output of the two compilers requires literal textual equality, which is too strict a requirement. Two compilers for the same language will naturally produce code that varies in inessential details, such as the use of local variables, ordering of basic blocks, etc. Trying to make a new code generator (written in OCaml) agree on all such details with an existing one (written in C++) would both be unnecessary work and create technical debt in the form of contorted code. The unappealing alternative is manually checking huge `diff` outputs and attempting to classify individual differences as significant or unimportant. What we really wanted was a mechanical way to check that the two compilers produce code that is behaviourally equivalent, i.e. produces the same observable behaviour in all contexts. (We also do not want performance regressions, but ignore that aspect of testing here.) This paper describes an automated tool, `semdiff`, that we built to check HHAS files for behavioural equivalence.

Behavioural equivalence is obviously undecidable, and writing a sound-but-incomplete fully automatic analysis that can cope with significant equivalence-preserving differences, such as those between optimized and unoptimized code, is generally extremely hard. However, in our case, the two compilers already produced (or were supposed to produce) 'essentially' the same code. Thus we were able to get good results with a fairly simple, fast analysis, that concentrates on 'trivial' differences.

`semdiff` is essentially a prover for a restricted version of Relational Hoare Logic (RHL) for low-level code. It was integrated into the test process for HackC, running over the compiled code from millions of lines of Hack and PHP, and freed the HackC team to concentrate on eliminating significant differences in the output of the two compilers.

1.1 `semdiff` architecture

`semdiff` takes two `hhas` files as input. These are each parsed in an unsurprising way into the representation of `hhbc` that is used by HackC. There are around 20 OCaml record types defining the nested structure of an bytecode file. For example, a `program` has a list of literal array constants, a list of top-level function definitions, a list of class definitions, a list of type definitions, and a distinguished main function. A function has a list of attributes, a name, a body, and a collection of boolean flags saying if it is `async`, a `generator`, and so on. Some of the record types have a non-trivial number of fields (e.g. a class currently has 24) and for nearly all of them, we just want to check for literal equality, set equality, or map equality. We reduce the pain of writing the associated boilerplate by using combinators:

```
type 'a compare = {
  comparer : 'a -> 'a -> int * (int * edit_sequence);
  size_of : 'a -> int;
  string_of : 'a -> string
}
```

An `'a compare` comprises a comparison function for values of type `'a`, a function that computes a rough indication of the size of values of type `'a`, and a `string_of` function. The comparison function returns an edit distance (which is zero if the two values are appropriately equal), a size, and an edit sequence, which can be thought of as a string representation of the delta between the two values. Values of type `'a compare` are built compositionally according to the structure of `'a` and the desired notion of equality. For example, there is a generic comparer for `'a lists` that takes a comparer for `'a` as argument, and computes the Levenshtein edit distance between the two lists. There is also a `join` combinator that takes two (independent) comparers for the same type and combines them into one that does both comparisons, and so on. The combinators are used to build up a value of type `program compare`, and it is the result of the associated `comparer` that is returned by `semdiff`.

The interesting comparer is that for the lists of instructions that are part of the bodies of functions and methods. Here we do *not* want to use our generic list comparer, as that would just yield a syntactic, rather than a semantic, comparison. Instead, we implement a comparer that tries to prove that two sequences of instructions have the same observable behaviour under the assumption that they are started in equivalent contexts (comprising the rest of the program, the parameters passed in, etc.). The intention is that this comparison is sound: if it reports that two bodies are equivalent then they really are according to the (intended) semantics of the bytecode. If the comparison fails, then it might be that there is an observable difference, or it might be that our analysis is too weak to establish equivalence. In such a case, we report the state of the prover when it got stuck and the syntactic difference between the two sequences, and hand over to a human being to decide whether this is a false positive or not.[1]

The comparer for instruction lists uses two different techniques. The first is a generic prover for a simple relational logic (Section 3). If that fails, then we try a collection of hardwired equivalence patterns (Section 4).

2 HHVM BYTECODE

`hhas` is code for a stack-based virtual machine with around three hundred opcodes, many of which are further parameterized by modes or operations. One could use a rather smaller instruction set but, for performance reasons, it is advantageous to have specialized versions for different contexts. For example, values that are to be used as the base for a later indexing operation are treated specially, as are values that are to be passed as function parameters. PHP's somewhat irregular semantics also calls for, for example, variants of instructions that differ in whether they throw exceptions or return default values in error cases. Some comparatively high-level features of `hhas` include direct support for asynchronous calls, iterators, and memo tables, two kinds of exception handler, and instructions for dynamically loading classes and functions.

As well as the stacks, `hhas` allows the use of named locals, such as `$x`, and 'unnamed' numbered temporaries, such as `_3`. If v is a variable, then the instruction `SetL` v copies the value on the top of the stack into v (without popping), `PopC` discards the value on the top of the stack, and `CGetL` v pushes the contents of v onto the evaluation stack. (We will describe other instructions as we encounter them.)

Figure 1 shows an example of some rather artificial Hack source code. Two different bytecode sequences that might correspond to that source are shown in Figure 2.

One can see that the two target files differ in their control flow, with simpler exception handling on the right, and different labels and jumps. For example, if `$pairs` is empty then the iterator created on the left on line 12 will jump to `L0` and then to `L2` before returning, whereas the corresponding one on the right will just jump to `L4` and return. The uses of local variables are also different. Local `_6` on the left, which stores the current value of the iterator, corresponds to local `_8` on the right. More subtly, local `_7` on the left, holding the result of the call to `$f`, also corresponds to local `_8` on the right. Other differences include the reversal of the tests on line 50 on the left and line 44 on the right, and the way `UnsetL` instructions on the left are not matched on the right.

Nevertheless, most of the instructions do correspond in a fairly simple way, and the two implementations are semantically equivalent.

3 RELATIONAL HOARE LOGIC FOR LOW-LEVEL CODE

Traditional Hoare logic for structured programs works with judgements $\vdash \{P\}\ C\ \{Q\}$, meaning that if command C is started in a state satisfying the precondition P then if it terminates, it does so in a state satisfying postcondition Q. Relational Hoare Logic [5] instead works with pairs of commands, and binary relations rather than unary predicates. The RHL judgement $\vdash \{\Phi\}\ C \sim C'\ \{\Psi\}$ means that if s and s' are states related by Φ, and we run C starting in s and C' starting in s', then either both executions diverge, or they both terminate, yielding final states that are related by Ψ. RHL can express many classic program analyses as well as the validity of the optimizing transformations they enable. It has also been profitably applied and extended in security, for verifying crypto primitives [3] and formalizing differential privacy [4].

To adapt RHL to unstructured, low-level code we move to a continuation-passing variant, in which we work only with pre-relations [6]. Suppose that l and l' are labels or program counters in low-level programs p and p', respectively, and that Φ is a binary relation on program states. Then we'll write $\models (l, l') : \Phi^{\top}$ to mean that for any states s,s' that are related by Φ, running p from program counter l in state s yields the same observable behaviour as running p' from program counter l' in state s', where observable behaviour encompasses termination and sequences of IO actions. We have previously formulated a relational version of separation logic in this style, and used it to specify a memory allocator and the semantics of high-level types as relations on low-level code [7, 10].

[1] We print a syntactic diff when the semantic equivalence check fails because it is hard to know what constitutes a 'semantic diff'. Nevertheless, there is room for improvement here: we could show symbolic traces leading up to the point of failure, for example.

```
function silly<Ta,Tb,Tc>(Traversable<(Ta, Tb)> $pairs, (function(Ta,Tb):Tc) $f) : vec<(Ta,Tc)> {
    $result = vec[];
    try {
      foreach ($pairs as list($a,$b)) {
        $result[] = $f($a,$b) |> tuple($a,$$);
      }
    }
    catch (Exception $e) {}
    return $result;
}
```

Figure 1: Example Hack source

The core of semdiff is a version of RHL in which the relations Φ are restricted to conjunctions of equalities between variables on the two sides. (This will be refined in subsequent sections.) We actually only consider equalities between *unnamed* locals, because PHP features 'variable variables', which allow runtime strings to be treated as variable names. Thus, for example,

```
$a = 'b';
$$a = 'x';
echo $b
```

prints 'x'. Variable variables introduce the possibility of aliasing for named locals, the static tracking of which would be complex.[2] The intended meaning of

$$(\_1 = \_1) \wedge (\_2 = \_3) \wedge (\_4 = \_3)$$

is the relation between program states that holds when local 1 on the left has the same value as local 1 on the right, locals 2 and 4 on the left both have the same value as local 3 on the right, *and* the unmentioned components of the state (named locals, stacks, heap, etc.) are the same.[3] We write *true* for the empty conjunction, which places no constraints on locals, but does still require the other components of the state to match. If we fix p and p' to be the bodies of two functions or methods with matching parameter lists, then we wish to show $\models (pc, pc') : true^{\top}$ for each matching pair of entry points pc and pc'. In other words, assuming that the parameters, etc. are equal on the two sides, then executions starting at pc and at pc' yield the same behaviour. These pairs of entry points will always include $pc = pc' = 0$, corresponding to an ordinary call, but there are others if the functions have optional arguments with default values. When some arguments are not supplied in a call, execution begins at a later point in the body, which initializes the omitted arguments and then jumps back to the top.

Rather than a traditional program logic, we choose to describe directly an abstract version of the analysis algorithm implemented in semdiff. This algorithm maintains two sets of assertions: Γ is the assumed assertions, while Θ is the assertions left to prove. Initially, Γ_0 is empty and

$$\Theta_0 = (pc_1, pc'_1) : true^{\top}, \ldots, (pc_n, pc'_n) : true^{\top}$$

for matching entry points pc_i, pc'_i. The algorithm takes steps $\Gamma_i \mid \Theta_i \rightarrow \Gamma_{i+1} \mid \Theta_{i+1}$ until there's nothing left to do (Θ_k is empty), in which case we report equivalence, or no transition is possible, in which case we report failure.

One transition rule allows us to use assumptions, with appropriate weakening of the relation:

$$\frac{\Phi_2 \leq \Phi_1}{\Gamma, (pc, pc') : {\Phi_1}^{\top} \mid \Theta, (pc, pc') : {\Phi_2}^{\top} \rightarrow \Gamma, (pc, pc') : {\Phi_1}^{\top} \mid \Theta}$$

another says that if we've reached a return instruction on both sides, then there's nothing more to prove. Recall that the interpretation of *any* Φ includes equivalence of the evaluation and call stacks on the two sides, so the values that are returned, and the contexts into which they'll be returned, will be equal:

$$\frac{p.pc = p'.pc' = \texttt{RetC}}{\Gamma \mid \Theta, (pc, pc') : \Phi^{\top} \rightarrow \Gamma, (pc, pc') : \Phi^{\top} \mid \Theta}$$

Note that we also add $(pc, pc') : \Phi^{\top}$ to the assumption set so that we can use it directly in future (though that doesn't achieve much in this particular case, as we could just as cheaply apply the rule again). Here's a more typical rule:

$$\frac{p.pc = p'.pc' = \texttt{Dup}}{\Gamma \mid \Theta, (pc, pc') : \Phi^{\top} \rightarrow \Gamma, (pc, pc') : \Phi^{\top} \mid \Theta, (pc + 1, pc' + 1) : \Phi^{\top}}$$

There is a Dup instruction on both sides, so we'll get the same behaviour in Φ-related states *provided* we get the same behaviour from the successor instructions in Φ-related states. The relation Φ does not change because duplicating the top of the stack does not affect any local variables. Again, what we were trying to show, $(pc, pc') : \Phi^{\top}$, is added to the assumptions in the new configuration. We discuss the soundness of this form of circular reasoning in Section 3.1.

For instructions that read (possibly different) local variables, we require the relation to imply that their values will be equal on the two sides:

$$\frac{p.pc = \texttt{CGetL}\ l \qquad p'.pc' = \texttt{CGetL}\ l' \qquad \Phi \leq l = l'}{\Gamma \mid \Theta, (pc, pc') : \Phi^{\top} \rightarrow \Gamma, (pc, pc') : \Phi^{\top} \mid \Theta, (pc + 1, pc' + 1) : \Phi^{\top}}$$

Whilst for those that write locals, we add an equation between variables to the relation associated with the successor states:

$$\frac{p.pc = \texttt{SetL}\ l \qquad p'.pc' = \texttt{SetL}\ l' \qquad \Phi_2 = \Phi[l = l']}{\Gamma \mid \Theta, (pc, pc') : \Phi^{\top} \rightarrow \Gamma, (pc, pc') : \Phi^{\top} \mid \Theta, (pc + 1, pc' + 1) : {\Phi_2}^{\top}}$$

where the notation $\Phi[l = l']$ means Φ with any existing equalities involving l on the left or l' on the right removed, and the equality $l = l'$ added. This relational treatment of reading and writing is

[2] Hack forbids variable variables, but semdiff has to work on legacy PHP code too.
[3] Actually, for compositionality, we want the assumed equivalence on the context to be a little weaker than literal equality. That can be made precise using biorthogonality [8], but we do not go into details here.

```
.function <"HH\\vec<(Ta, Tc)>">
 silly(<"HH\\Traversable<(Ta, Tb)>"> $pairs,
        <"(function (Ta, Tb): Tc)" N > $f) {
  .numiters 1;
  .declvars $result $a $b $e;
  VerifyParamType $pairs
  Vec @A_0
  SetL $result
  PopC
  .try {
    CGetL $pairs
    IterInit 0 L0 _6
    .try_fault F4 {
      .try_fault F5 {
      L1:
        BaseL _6 Warn
        QueryM 0 CGet EI:1
        SetL $b
        PopC
        BaseL _6 Warn
        QueryM 0 CGet EI:0
        SetL $a
        PopC
        UnsetL _6
      }
      CGetL $f
      FPushFunc 2
      FPassL 0 $a Cell
      FPassL 1 $b Cell
      FCall 2
      UnboxR
      SetL _7
      PopC
      .try_fault F6 {
        CGetL $a
        CGetL _7
        NewPackedArray 2
      }
      UnsetL _7
      BaseL $result Define
      SetM 0 W
      PopC
      IterNext 0 L1 _6
    }
  L0:
    Jmp L2
  } .catch {
    Dup
    InstanceOfD "Exception"
    JmpZ L3
    SetL $e
    PopC
    Jmp L2
  L3:
    Throw
  }
L2:
  CGetL $result
  VerifyRetTypeC
  RetC
F5:
  UnsetL _6
  Unwind
F6:
  UnsetL _7
  Unwind
F4:
  IterFree 0
  Unwind
}
```

```
.function <"HH\\vec<(Ta, Tc)>">
 silly(<"HH\\Traversable<(Ta, Tb)>"> $pairs,
        <"(function (Ta, Tb): Tc)" N > $f) {
  .numiters 1;
  .declvars $result $a $b $e;
  VerifyParamType $pairs
  Vec @A_0
  SetL $result
  PopC
  .try {
    CGetL $pairs
    IterInit 0 L4 _8
    .try_fault F1 {
    L0:
      BaseL _8 Warn
      QueryM 0 CGet EI:1
      SetL $b
      PopC
      BaseL _8 Warn
      QueryM 0 CGet EI:0
      SetL $a
      PopC
      CGetL $f
      FPushFunc 2
      FPassL 0 $a Cell
      FPassL 1 $b Cell
      FCall 2
      UnboxR
      SetL _8
      PopC
      CGetL $a
      CGetL _8
      NewPackedArray 2
      UnsetL _8
      BaseL $result Define
      SetM 0 W
      PopC
      IterNext 0 L0 _8
    }
    Jmp L4
  } .catch {
    Dup
    InstanceOfD "Exception"
    JmpNZ L1
    Throw
  L1:
    SetL $e
    PopC
  }
L4:
  CGetL $result
  VerifyRetTypeC
  RetC
F1:
  IterFree 0
  Unwind
}
```

Figure 2: Different hhas corresponding to Figure 1

essentially that of our earlier work with Hofmann on effect systems [9].

Some instructions both read and write:

$$\frac{p.pc = \mathtt{IncDecL}\ l\ op \qquad p'.pc' = \mathtt{IncDecL}\ l'\ op \qquad \Phi \le l = l' \qquad \Phi_2 = \Phi[l = l']}{\Gamma \mid \Theta, (pc, pc') : \Phi^\top \longrightarrow \Gamma, (pc, pc') : \Phi^\top \mid \Theta, (pc+1, pc'+1) : {\Phi_2}^\top}$$

For conditional branches and switches, our simple form of relation will ensure that the value being switched on will be the same, so we have to check both that the two successor instructions are related *and* that the two branched-to instructions are related:

$$\frac{p.pc = \mathtt{JmpZ}\ L \qquad p'.pc' = \mathtt{JmpZ}\ L'}{\Gamma \mid \Theta, (pc, pc') : \Phi^\top \rightarrow \Gamma, (pc, pc') : \Phi^\top \mid \Theta, (pc+1, pc'+1) : \Phi^\top, (L, L') : \Phi^\top}$$

where we identify labels with the program counters to which they refer.

So far, we have only considered cases where we have similar instructions on both sides. We also allow some instructions, such as unconditional branches, to make moves on one side only, which allows for some differences in control flow on the two sides. We define a left-move relation $(pc_0, pc'_1){:}\Phi_0{}^\top \xrightarrow{L} (pc_1, pc'_1){:}\Phi_1{}^\top$ using rules like this:

$$\frac{p.pc = \texttt{Jmp}\ L}{(pc, pc'){:}\Phi^\top \xrightarrow{L} (L, pc'){:}\Phi^\top}$$

with matching rules for the right-move relation $\xrightarrow{R}$. We then add a rule that allows steps on the left and steps on the right to combine to yield steps on both sides:

$$\frac{(pc_0, pc'_0){:}\Phi_0{}^\top \xrightarrow{L}{}^+ (pc_m, pc'_0){:}\Phi_1{}^\top \xrightarrow{R}{}^+ (pc_m, pc'_n){:}\Phi_2{}^\top}{\Gamma \mid \Theta, (pc_0, pc'_0){:}\Phi_0{}^\top \longrightarrow \Gamma, (pc_0, pc'_0){:}\Phi_0{}^\top \mid \Theta, (pc_m, pc'_n){:}\Phi_2{}^\top}$$

Note that this rule requires that the numbers of steps on each side (m and n) to be greater than 0, though they need not be the same. We can also allow steps on one side only, provided they are followed by a two-sided step:

$$\frac{\begin{array}{c}(pc, pc'){:}\Phi^\top \xrightarrow{L}{}^+ (pc_m, pc'){:}\Phi_1{}^\top \\ \Gamma \mid \Theta, (pc_m, pc'){:}\Phi_1{}^\top \longrightarrow \Gamma, (pc_m, pc'){:}\Phi_1{}^\top \mid \Theta_2\end{array}}{\Gamma \mid \Theta, (pc, pc'){:}\Phi^\top \longrightarrow \Gamma, (pc, pc'){:}\Phi^\top, (pc_m, pc'){:}\Phi_1{}^\top \mid \Theta_2}$$

and similarly for moves on the right. Note that we only introduce the assumption $(pc, pc'){:}\Phi^\top$ after we have checked the two-sided move without it: to ensure the soundness of our reasoning about loops, goals can only become assumptions once both sides have taken at least one step.

In the actual implementation, the non-determinism involved in choosing how many one-sided steps to take is resolved by making as many one-sided steps as possible, whilst checking explicitly for loops.

3.1 Soundness

We do not have a formal semantics of hhas with respect to which we could do a full proof of soundness. However, the semi-formal argument for the correctness of our treatment of loops is slightly delicate and deserves some comment. The argument can be seen as a translation into a step-indexed logic [1] with a Gödel-Löb style modality for recursion [2, 10]. Let the operational semantics of HHVM be given by a transition relation $\langle p, pc_0, s_0\rangle \mapsto \langle p, pc_1, s_1\rangle$, write $\langle p, pc, s\rangle \downarrow^k$ to mean that program p started at program counter pc and state $s \in S$ halts within k steps, and $\langle p, pc, s\rangle \uparrow$ to mean it diverges. (For simplicity, we just consider termination behaviour; the extension to IO and errors is not fundamentally different.)

Given an interpretation of our state relations $\Phi \subseteq S \times S$, we define our relations on program counters $\Phi^\top \in \mathbb{N} \to \mathbb{P}(\mathbb{N} \times \mathbb{N})$ in a step-indexed way:

$$\begin{aligned}\Phi^\top(k) = \{(pc, pc') \mid \forall (s, s') \in \Phi, \forall j \le k,\\ \langle p, pc, s\rangle \downarrow^k \implies \langle p', pc', s'\rangle \downarrow^\omega \wedge\\ \langle p', pc', s'\rangle \downarrow^k \implies \langle p, pc, s\rangle \downarrow^\omega\}.\end{aligned}$$

Note that if $\Phi_0 \subseteq \Phi_1$ then $\Phi_1{}^\top(k) \subseteq \Phi_0{}^\top(k)$ and that $\Phi^\top(k) \supseteq \Phi^\top(k+1)$. Now, given $R \in \mathbb{N} \to \mathbb{P}(\mathbb{N} \times \mathbb{N})$, define $\triangleright R$ by

$$\begin{aligned}\triangleright R(0) &= \mathbb{N} \times \mathbb{N}\\ \triangleright R(k+1) &= R(k)\end{aligned}$$

If R decreases with k, so does $\triangleright R$. We can define a logic that works with decreasing step-indexed relations, with conjunction defined pointwise and judgements being implicitly universally quantified over step indices. The $\triangleright$ modality then distributes over conjunction and satisfies $R \le \triangleright R$. It also validates a Löb rule, justifying guarded circular reasoning:

$$\frac{\Delta, (pc, pc') : \triangleright R \models (pc, pc') : R}{\Delta \models (pc, pc') : R}$$

which follows by simple mathematical induction on step indices.

We interpret configurations of the algorithm via (the semantics of) judgements in the modal logic

$$[\![\Gamma \mid \Theta]\!] = \Gamma, \triangleright\Theta \models \Theta$$

where the commas in Γ and Θ are treated as conjunctions. We then check that each transition is semantically valid, in the sense that $\Gamma_0 \mid \Theta_0 \longrightarrow \Gamma_1 \mid \Theta_1$ implies

$$\frac{[\![\Gamma_1 \mid \Theta_1]\!]}{[\![\Gamma_0 \mid \Theta_0]\!]}$$

If the algorithm succeeds, the semantics of the terminal configuration $[\![\Gamma_n \mid -]\!]$ is $\Gamma_n \models \mathbf{t}$, which is certainly valid, so by induction we get the validity of $\triangleright\Theta_0 \models \Theta_0$, which is the semantics of the initial configuration (where Θ_0 is the set of $(pc_i, pc'_i){:}\mathit{true}^\top$ for each pair of matching entry points). Finally, we can apply the Löb rule above to conclude $\models \Theta_0$ as required.

3.2 Unset variables

Separate from the presence of null values, variables in PHP can be *set* or *unset*. All variables are initially unset, assigning to a variable makes it set, and it is also possible to explicitly unset them, and to test their set status. Some code explicitly tests this status on unnamed variables that may not yet have been assigned to, for example to use the status of a local as a boolean flag that is implicitly set to false at the start of a function. In order to check equivalence of such code, we enhance our conjunction-of-equalities relations with two sets of variables, recording which variables on each side *may* be set. So now we have

$$\Phi \quad ::= \quad (l_1 = l'_1 \wedge \cdots \wedge l_m = l'_m; \{\bar{l}_1, \ldots, \bar{l}_n\}; \{\bar{l}'_1, \ldots, \bar{l}'_{n'}\})$$

which we'll sometimes abbreviate as $\Phi = (\phi; L; R)$. The initial relations that are used at entry points have both L and R empty, corresponding to the fact that on entry we know that all variables are unset on both sides. We refine entailment between relations accordingly; in particular, the $\Phi \le l = l'$ condition that applies when variables are accessed (for example in the CGetL rule above, and also in the rule for the Isset instruction) now means $(\exists i, l = l_i \wedge l' = l'_i) \vee (\forall j, l \ne \bar{l}_j \wedge \forall j', l' \ne \bar{l}'_{j'})$.

Corresponding assignments to variables now add to the two sets, so the SetL rule becomes

$$\frac{p.pc = \texttt{SetL}\ l \qquad p'.pc' = \texttt{SetL}\ l' \qquad \Phi = (\phi; L; R) \qquad \Phi_2 = (\phi[l = l']; L \cup \{l\}; R \cup \{l'\})}{\Gamma \mid \Theta, (pc, pc'){:}\Phi^\top \longrightarrow \Gamma, (pc, pc'){:}\Phi^\top \mid \Theta, (pc+1, pc'+1){:}{\Phi_2}^\top}$$

We remark that it is not the case that the sets L and R are always just the sets of variables appearing on the left and right hand sides of equalities in ϕ: the removal of equations that is part of the $\phi[l = l']$ operation can leave variables possibly set but not involved in any equation.

The two-sided rule for Unset removes any equations involving the corresponding variables and also removes them from the sets:

$$\frac{p.pc = \texttt{Unset}\ l \qquad p'.pc' = \texttt{Unset}\ l' \qquad \Phi = (\phi; L; R) \qquad \Phi_2 = (\phi[l, l']; L \setminus \{l\}; R \setminus \{l'\})}{\Gamma \mid \Theta, (pc, pc'){:}\Phi^\top \longrightarrow \Gamma, (pc, pc'){:}\Phi^\top \mid \Theta, (pc+1, pc'+1){:}{\Phi_2}^\top}$$

where we write $\phi[l, l']$ for the removal of equations involving l on the left or l' on the right.

Unsetting a variable also has the effect of removing a reference to its previous value, which may cause the associated storage to be freed. And indeed, locals frequently are explicitly unset prior to the end of a function. One might think (and we initially did think) that deallocation is unobservable, so one does not need to insist that Unsets on the two sides have to match up exactly. That led us to add a one-sided rule

$$\frac{p.pc = \texttt{Unset}\ l}{(l, l'){:}(\phi; L; R)^\top \xrightarrow{L} (l+1, l'){:}(\phi[l, -]; L \setminus \{l\}; R)^\top}$$

and similarly on the right. However, collecting an object can also cause a finalizer to run. When using a reference counting collector, the order in which finalizers execute is deterministic, and may be observable. The Hack semantics is moving away from deterministic finalization, but for the moment we have put the one-sided treatment of Unset behind a flag.

3.3 Exception handling

HHVM has a somewhat non-trivial model for exception handling. As one can see in Figure 2, blocks of code can be wrapped like this:

```
.try {
  // code...
} .catch {
  // handler...
}
```

or like this:

```
.try_fault Fn {
  // code...
}
...
Fn:
  // fault handler...
  Unwind
```

The first form is used in compiling `try {...} catch {...}` in the source, while the latter is used for `try {...} finally {...}` as well as to ensure that resources allocated during the compilation of other language constructs (such as iterators) are freed. The two forms can be nested, and what happens when an exception is thrown depends on a mix of static and dynamic information. Each program counter is mapped statically to a list of enclosing handlers, which may be either catch blocks or fault handlers. The runtime also maintains a dynamic stack of currently-active handlers. When an exception is thrown, the static handlers are added to the dynamic ones and the 'unwinder' is entered. If the topmost handler is a fault handler then control is passed to it until an Unwind is encountered, at which point we resume unwinding. If the topmost handler is a catch block then control is passed to it, with the exception object on the evaluation stack, maintaining the remainder of the dynamic stack. If the handler stack is empty, then the exception is re-raised in the calling frame.

To deal with this behaviour in semdiff, we refine the notion of program counter from a simple integer to an integer (for the actual program counter) paired with a list of handlers, each of which is an integer and an indication of which kind of handler it refers to. We allow one-sided transitions on Unwind instructions, which allows for some variation in exceptional control flow between the two sides.

As well as the explicit Throw instruction, a great many instructions potentially raise faults, including out of bounds accesses and dynamic type violations. Precisely when faults are raised can also depend on dynamic conditions. Rather than try to track the potential for throwing accurately, we make the very conservative assumption that most instructions can throw. Hence the two-sided rules for almost all instructions actually generate two, rather than one, subgoal in Θ: one for the next instruction and one for the exception case.

One side-effect of the refinement of program counters is that it adds a degree of 'polyvariance' to our analysis: we can associated different relations with the same location, provided the exception stacks differ.

3.4 Disjunction

For program points that occur on many paths, the relation that is assumed the first time the algorithm visits a point is quite likely to be too strong to be a global invariant. This is especially true for exception handlers, which (especially given our pessimistic assumption that most instructions can throw) can be entered from many different places, where different relations hold. Similar considerations apply to loops: it is not uncommon for a loop to be entered with an equation holding which the loop body invalidates, but which is not necessary to establish equivalence. To deal with this problem, we further enhance the form of our relations to include finite disjunctions of relations:

$$\Phi \quad := \quad (\phi_1; L_1; R_1) \vee \cdots \vee (\phi_n; L_n; R_n)$$

Formally, this is covered by our existing rules, if we treat the assumptions Γ and todos Θ as sets rather than partial maps from (pc, pc') pairs. The reason that is the case is that for any relations R_1, R_2 on states

$$(l, l'){:}(R_1 \vee R_2)^\top \quad\Longleftrightarrow\quad ((l, l'){:}{R_1}^\top) \wedge ((l, l'){:}{R_2}^\top)$$

but, in practice, we do use a partial map from pairs to sets of relations. The simple version of the algorithm would report failure when the $\Phi_2 \leq \Phi_1$ condition of the assumption rule was not satisfied. Now we can alternatively just analyze the code again with the prerelation Φ_2 unioned in:

$$\frac{}{\Gamma, (pc, pc'){:}\Phi_1{}^\top \mid \Theta, (pc, pc'){:}\Phi_2{}^\top \longrightarrow \Gamma, (pc, pc'){:}(\Phi_1 \vee \Phi_2)^\top \mid \Theta, (pc, pc'){:}\Phi_2{}^\top}$$

In theory, the finite nature of the set of relations that could be generated for a given function body means that this process should converge to a fixed point. However, we do not check for this. Instead, we simply limit the number of (semantically distinct) disjuncts that may be considered before we report failure to an empirically-determined hard bound, which is currently 7. (A much lower limit would suffice for nearly all the real code we have encountered.)

3.5 Dynamic loading

One question we have not yet addressed is how we decide which classes, and which functions/methods within those classes, should be compared by `semdiff`. This is not just a matter of matching up the names, because PHP is very dynamic language. For example, there can be more than one class with a given name in a file, and which one is actually loaded into the runtime can depend on dynamic control flow. Thus `semdiff` starts by comparing the top-level `.main` functions in each file, and dynamically schedules pairs of classes for comparison when it finds a matching pair of `DefCls` instructions (which refer to actual class definitions by integer index into the file). Those comparisons can schedule others, and so on. Similar considerations apply to functions (the `DefFunc` instruction) and closures (the `CreateCl` instruction). In the latter case we allow the two classes to store their free variables in a different order, so we compare up to a permutation.

4 PEEPHOLE EQUIVALENCE PATTERNS

As we said in the introduction, the 'logical' part of `semdiff` is backed up by a more ad hoc collection of equivalence patterns. The two compilers often produce code that could not be shown equivalent using the fairly simple rules we have described so far. One naturally thinks of increasing the power of the logic to track details of what's on the stack, and so on. But that would involve a detailed logical encoding of the semantics of each instruction, and probably have to rely on an external solver (which, apart from the complexity, would be comparatively slow). Since we were only interested in two specific compilers, whose correct output was likely to vary in a small number of predictable ways, we could get away with a much cruder approach. Whenever `semdiff` reported a discrepancy, we examined it and decided what to do:

- If we thought it was a real difference (which often involved lengthy discussions) then we changed the new compiler.
- If we thought the two bits of code really were equivalent then we chose between:
  - Increasing the power of `semdiff`'s analysis algorithm (for example, adding the treatment of unset variables),
  - Adding a custom equivalence pattern, or
  - Changing the new compiler anyway, as this was sometimes the simplest solution.

These custom equivalence patterns are written using another small set of combinators, very like parser combinators. There are about twenty patterns, ranging from trivial peephole equivalences to quite subtle patterns that can span many instructions. Here's an example of a simple equivalence, which should be self-explanatory:

```
Not
JmpZ L     =   JmpNZ L
```

and here's another, expressing the associativity of string concatenation (which shows up rather often in typical Hack code):

```
String "foo"     Concat
Concat        =  String "foo"
Concat           Concat
```

Here's a slightly more complex example, expressed as a special purpose rule:

$$\frac{p.pc = \mathsf{FPassL}\ n\ l \qquad p'.pc' = \mathsf{FPassL}\ n\ l' \qquad safe(p, pc+1) \qquad safe(p', pc'+1) \qquad \Phi \leq l = l' \qquad \Phi_2 = \Phi[l = l']}{\Gamma \mid \Theta, (pc, pc'){:}\Phi^\top \longrightarrow \Gamma, (pc, pc'){:}\Phi^\top \mid \Theta, (pc+1, pc'+1){:}\Phi_2{}^\top}$$

What's going on here is that we are pushing local l (resp. l') onto the stack as argument n of a function that will be called a bit later on. The problem is that whether that local is passed by value or by reference depends on the runtime signature of the function that gets called, and we don't always know statically what that will be. If the call is by reference then the function call could potentially invalidate some equations involving these locals. The predicate *safe* checks that the following instructions just involve pushing some more arguments, followed by a call that will consume them all. If that's the case then we can safely approximate the effect of the call by deleting any other equations involving l on the left or l' on the right (which we do earlier than strictly necessary).

An unexpected benefit of the pattern-matching combinators was they were easy to understand, so team members could extend `semdiff` without having to understand the details of the logical analysis.

5 DISCUSSION

The experience of using `semdiff` while developing HackC was very positive. We integrated the tool into our routine testing process whilst both it and HackC were under development, running it regularly over our test suites and, later on, over millions of lines of real code.

It should, of course, be stressed that we also test by actually running compiled code, but `semdiff` provided complimentary coverage. Although it still relies on a collection of tests, it provides coverage of all possible paths through each function, rather than just those that are executed in a test run. Indeed, it works well with tests that are not even runnable programs. We found and fixed numerous issues in HackC by using `semdiff`, and once we reached the point that `semdiff` validated the compilation of all our Hack code, reports of code generation bugs became conspicuous by their absence.

The choice of a simple, specialized prover combined with hard-wired patterns was certainly the right one for the immediate problem we faced. But it does make the tool rather inflexible. In particular, it would be good to continue to use an automated tool for regression testing between future versions of HackC, but if we were to introduce any non-trivial optimizations then we would probably need to move to using an external solver. It should also be noted that coding the analysis from scratch in a deterministic language, while resulting in a fast tool, was surprisingly awkward. Being able to express rules directly in a more declarative style would be much easier.

An early example of a semantic diff is that of Jackson and Ladd [11], which works by computing the dependence relation between the inputs and outputs of a procedure in a compositional way, and reporting and differences it finds between two versions. Notable amongst a number of other automated provers for program equivalence is SymDiff, by Lahiri et al. [12]. SymDiff is a tool that can compare programs in the verification language Boogie for partial (terminating) equivalence. There are translators from several languages, including C and C♯ into Boogie. SymDiff works by composing (appropriately renamed) versions of the two programs and generates verification conditions to be discharged by the Z3 solver. SymDiff can generate counterexample traces when verification fails. It can also deal with equivalences that rely on much more complex logical properties than `semdiff`, but is considerably slower and relies on a rather larger infrastructure. Closer in spirit to `semdiff` is the work of Rideau and Leroy on validating a register allocator [15] in the context of the CompCert verified compiler project [13]. Verifying advanced spilling algorithms once and for all turns out to be rather hard, so Rideau and Leroy present a translation validation approach that generates a proof of equivalence for specific runs of the allocator. Their dataflow analysis is very similar to that of `semdiff`, with the notable difference that it is a backwards analysis, which they found led to smaller verification conditions.

6 DEDICATION

This paper is dedicated to the memory of my dear friend and longterm collaborator, Martin Hofmann. I am only one of many people whom Martin taught much about life, beer, and mathematics. He is deeply missed.

7 ACKNOWLEDGEMENTS

I'd like to thank CJ Bell, Paul Bissonnette, Catherine Gasnier, Philip Hölzenspies, Andrew Kennedy, Arun Kumar, Vladimir Matveev, Greg Nisbet, Oguz Ulgen, and Hugo Venturini for their contributions to the development, testing, and deployment of `semdiff`.

REFERENCES

[1] A. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5), 2001.

[2] A.W. Appel, P.A. Melliès, C.D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, 2007.

[3] G. Barthe, B. Grégoire, and S. Zanella. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL '09)*. ACM, 2009.

[4] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3), November 2013.

[5] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL)*, January 2004.

[6] N. Benton. A typed, compositional logic for a stack-based abstract machine. In *Proc. 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, 2005.

[7] N. Benton. Abstracting allocation: The new new thing. In *Proceedings of Computer Science Logic (CSL '06)*, number 4207 in Lecture Notes in Computer Science. Springer-Verlag, September 2006.

[8] N. Benton and C-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ICFP*, 2009.

[9] N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations: Towards extensional semantics for effect analyses. In *Proc. 4th Asian Symposium on Programming Languages and Systems (APLAS '06)*, 2006.

[10] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *Proceedings of the Fourth ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, January 2009.

[11] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance (ICSM '94)*. IEEE, 1994.

[12] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12)*, number 7358 in Lecture Notes in Computer Science. Springer-Verlag, 2012.

[13] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

[14] G. Ottoni. HHVM JIT: A profile-guided, region-based compiler for PHP and Hack. In *Proceedings of the 39th ACM Conference on Programming Language Design and Implementation (PLDI '18)*. ACM, 2018.

[15] S. Rideau and X. Leroy. Validating register allocation and spilling. In *Proceedings of the 19th International Conference on Compiler Construction (CC '10)*, number 6011 in Lecture Notes in Computer Science. Springer-Verlag, 2010.

[16] O. Yamuachi. *Hack and HHVM: Programming Productivity Without Breaking Things.* O'Reilly, 2015.