Removing Dynamic Type Tests with Context-Driven Basic Block Versioning

Maxime Chevalier-Boisvert and Marc Feeley

DIRO, Université de Montréal, Montreal, QC, Canada
{chevalma,feeley}@iro.umontreal.ca

Abstract. Dynamic typing is an important feature of dynamic programming languages. Primitive operators such as those for performing arithmetic and comparisons typically operate on a wide variety of input value types, and as such, must internally implement some form of dynamic type dispatch and type checking. Removing such type tests is important for an efficient implementation.
In this paper, we examine the effectiveness of a novel approach to reducing the number of dynamically executed type tests called *context-driven basic block versioning*. This simple technique clones and specializes basic blocks in such a way as to allow the compiler to accumulate type information while machine code is generated, without a separate type analysis pass. The accumulated information allows the removal of some redundant type tests, particularly in performance-critical paths.
We have implemented intraprocedural context-driven basic block versioning in a JavaScript JIT compiler. For comparison, we have also implemented a classical flow-based type analysis operating on the same concrete types. Our results show that basic block versioning performs better on most benchmarks and removes a large fraction of type tests at the expense of a moderate code size increase. We believe that this technique offers a good tradeoff between implementation complexity and performance, and is suitable for integration in production JIT compilers.

1 Introduction

Dynamic programming languages make heavy use of late binding in their semantics. In essence this means doing at run time what can be done before run time in other programming languages, for example type checking, type dispatch, function redefinition, code linking, program evaluation (e.g. `eval`), and compilation (e.g. JIT compilation). In dynamic programming languages such as JavaScript, Python, Ruby and Scheme, there are no type annotations on variables and types are instead associated with values. Primitive operations, such as `+`, must verify that the operand values are of an acceptable type (type checking) and must use the types of the values to select the operation, such as integer addition, floating point addition, or string concatenation (type dispatching). We will use the generic term *type test* to mean a run time operation that determines if a value belongs to a given type. Type checking and dispatching are built with type tests.

VMs for dynamic programming languages must tackle the run time overhead caused by the dynamic features to achieve an efficient execution. Clever type representation and runtime system organization can help reduce the cost of the dynamic features. In this paper we focus on reducing the number of type tests executed, which is a complementary approach.

Static type analyses which infer a type for each variable can help remove and in some cases eliminate type test cost. However, such analyses are of limited applicability in dynamic languages because of the run time cost and the presence of generic operators. A whole program analysis provides good precision, compared to a more local analysis, but it is time consuming, which is an issue when compilation is done during program execution. Moreover, the results are generally invalidated when functions are redefined and code is dynamically loaded or evaluated, requiring a new program analysis. This often means that analysis precision must be traded for speed. Intraprocedural analyses are a good compromise when such dynamic features are used often, the program is large, the running time is short or a simple VM design is desired. The complexity of the type hierarchy for the numerical types may negatively impact the precision of the type analysis due to its conservative nature. To implement the numerical types of the language or for performance, a VM may use several concrete types for numbers (e.g. fixed precision integers, infinite precision integers, floating point numbers, complex numbers, etc). The VM automatically converts from one representation to another when an operator receives mixed-representation operands or limit cases are encountered (e.g. overflows). Variables containing numbers will often have to be assigned a type which is the union of some concrete numerical types (e.g. $\texttt{int} \cup \texttt{float}$) if they store the result of an arithmetic operator. This means that a type dispatch will have to be executed when this variable is used as an operand of another arithmetic operator. This is an important issue due to the frequent use of arithmetic in typical programs (for example an innocuous looking `i++` in a loop will typically require a type dispatch and overflow check).

We propose a new approach which reduces the number of type tests by eliminating those that are redundant within each function. Basic block versioning aims to be a simple and efficient technique mixing code generation, analysis and optimization. Section 2 explains the basic block versioning approach in more details. An implementation of our approach in a JavaScript compiler is described in Section 3 and evaluated in Section 4. Related work is presented in Section 5.

2 Basic Block Versioning

The basic block versioning approach generates one or more versions of each live basic block based on type information derived from the type tests executed by the code. The type analysis and code generation are performed together, generating on-demand new versions of blocks specialized to the typing context of predecessor blocks.

An important difference between this approach and traditional type analyses is that basic block versioning does not compute a fixed-point on types, but rather

```
function f(n) {
    if (n<0) n = n+1;
    return n+1;
}
```

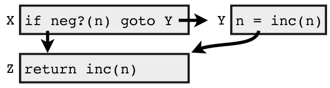

Fig. 1. Definition for function `f` and the corresponding high-level control flow graph.

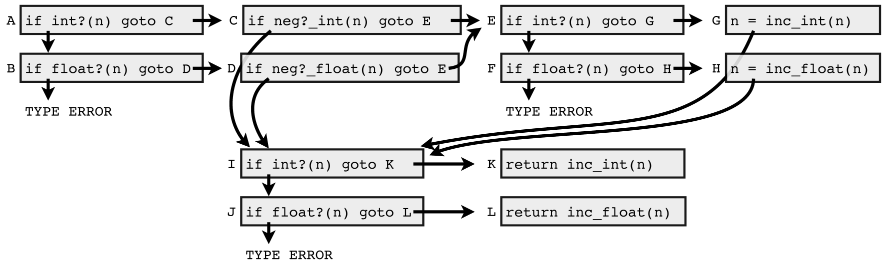

Fig. 2. Control flow graph after the inlining of the primitive operators `neg?` and `inc`.

computes a fixed-point on the generation of new block versions, each associated with a configuration of incoming types. Values which have different types at the same program point are handled more precisely with basic block versioning. In a traditional type analysis the union of the possible types would be assigned to the value, causing the analysis to be conservative. With basic block versioning, distinct basic blocks will be created for each type tested previously, allowing a more precise tracking of types. Because versions are created on demand, only versions for the relevant type combinations are created.

To illustrate this approach we will use a simple example in a hypothetical dynamically typed language similar only in syntax to JavaScript. Consider the function `f` whose definition and corresponding high-level control flow graph are shown in Figure 1. Lets assume that there are only two concrete types for numbers: `int`, a fixed precision integer, and `float`, a floating point number.[1] The value of parameter `n` must be one of these two types, otherwise it is a type error. The primitive operations `neg?(n)` and `inc(n)` must include a type dispatch to select the appropriate operation based on the concrete type of `n`. Inlining these primitive operations makes the type tests explicit as shown in the control flow graph in Figure 2. Note that basic block X has been expanded to basic blocks A-D, while Y has been expanded to E-H, and Z has been expanded to I-L. Note that for simplicity we will assume that the `inc_int(n)` operation yields an `int` (i.e. there is no overflow check).

Basic block versioning starts compiling basic block A with a context where value `n` is of an unknown type. This will generate the code for the `int?(n)` type test and will schedule the compilation of a version of block B, called $B_{\neg\texttt{int}}$,

[1] Note that JavaScript has a single type for numbers, which corresponds to IEEE 64-bit floating point numbers, but an implementation of JavaScript could implement numbers with these two concrete types to benefit from the performance of integer arithmetic for integer loop iteration variables and array indexing.

where the value `n` is known to not be an `int` and will schedule the compilation of a version of block C, called $C_{\mathtt{int}}$, where the value `n` is known to be an `int`. Our use of subscripts is a purely notational way of keeping track of the type context information, which only needs to give information on `n` in this example. When basic block $C_{\mathtt{int}}$ is compiled, code is generated for the `neg?_int(n)` test and this schedules the compilation of versions of blocks E and I, called $E_{\mathtt{int}}$ and $I_{\mathtt{int}}$ respectively, where the value `n` is known to be an `int`. Note that the compilation of $B_{\neg\mathtt{int}}$ will cause the compilation of $D_{\mathtt{float}}$, which will also schedule the compilation of versions of blocks E and I but in a different context, where the value `n` is known to be a `float` (blocks $E_{\mathtt{float}}$ and $I_{\mathtt{float}}$ respectively).

The type tests in the four blocks $E_{\mathtt{int}}$, $E_{\mathtt{float}}$, $I_{\mathtt{int}}$ and $I_{\mathtt{float}}$ can be removed and replaced by direct jumps to the appropriate destination blocks. For example $E_{\mathtt{int}}$ becomes a direct jump to $G_{\mathtt{int}}$ and $E_{\mathtt{float}}$ becomes a direct jump to $F_{\mathtt{float}}$. Because $G_{\mathtt{int}}$ and $H_{\mathtt{float}}$ jump respectively to $I_{\mathtt{int}}$ and $I_{\mathtt{float}}$, the type tests in those blocks are also removed. Note that the final generated code implements the same control flow graph as Figure 3. Two of the three type dispatch operations in the original code have been removed.

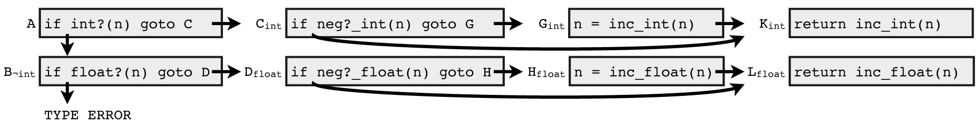

Fig. 3. Final control flow of function `f` after basic block versioning.

Let us now consider what would happen if the `inc_int(n)` operations detected integer overflow and yielded a `float` result in that case. Then the compilation of basic block $G_{\mathtt{int}}$ would schedule two versions of the successor basic block I: $I_{\mathtt{int}}$ for the case where there is no overflow, and $I_{\mathtt{float}}$ for the case where there is an overflow. Due to the normal removal of type tests, when `inc_int(n)` would overflow, a `float` would be stored in `n` followed by a direct jump to block $L_{\mathtt{float}}$. Thus the only change in the control flow of Figure 3 is that block $G_{\mathtt{int}}$ has an edge to $L_{\mathtt{float}}$ in addition to $K_{\mathtt{int}}$. So here too a single type dispatch is needed in function `f`.

In theory, the number of possible type configurations in a context grows combinatorially with the number of live values whose type is being accounted for and the number of types they can have. We believe that a combinatorial explosion is unlikely to be a problem in practice because typically the number of values live at a given program point is small and the number of possible types of a value is small.

There are pathological cases where a large number of block versions are needed to account for all the possible incoming type combinations. To prevent such occurrences, a simple approach is to place an arbitrary limit per block on the number of versions that are compiled. Once this limit is hit for a given block, a general version of the block will be compiled, which makes no assumptions about incoming types, i.e. all values are of unknown type.

If more versions of a block would be required than the block version limit allows, it is advantageous to compile the versions which will be executed most often. This can be done by monitoring the frequency of execution of each basic block (with a counter per block) prior to the JIT compilation. Generating linear machine code sequences along hot paths first has the beneficial effect that it will tend to prioritize the compilation of block versions for type combinations that occur more frequently at run time. This strategy is used in our experiments.

An indirect benefit of our basic block versioning approach is that it automatically unrolls some of the first iterations of loops in such a way that type tests are hoisted out of loop bodies. For example, if variables of unknown type are used unconditionnaly in a loop, their type will be tested inside the first iteration of the loop. Once this test is performed in the first loop iteration, the type information gained will allow the loop body to avoid the redundant type tests for the remaining iterations.

3 Implementation in Higgs

We have implemented basic block versioning inside a JavaScript virtual machine called Higgs. This virtual machine comprises an interpreter and a JIT compiler targeted at x86-64 POSIX platforms. The current implementation of Higgs supports most of the ECMAScript 5 specification [1], with the exception of the `with` statement, property attributes and getter-setter properties. Its runtime and standard libraries are self-hosted, written in an extended dialect of ECMAScript with low-level primitives. These low-level primitives are special IR instructions which allow us to express type tests as well as integer and floating point machine instructions in the implementation language.

In Higgs, the interpreter is used for profiling, and as a default, unoptimized mode of execution. Functions are parsed into an abstract syntax tree, and lazily compiled to an Static Single Assignment (SSA) Intermediate Representation (IR) when they are first called. The interpreter then executes code in SSA form directly. As code is executed by the interpreter, counters on basic blocks are incremented every time a given block is executed. Frequency counts for each potential callee are also collected at call sites.

The JIT compiler is triggered when the execution count for a function entry block or loop header reaches a fixed threshold (currently set to 800). Callees are first aggressively inlined into the function to be compiled. This is done by substituting the IR of callees at call sites. Calls are currently inlined only if profiling data indicates that they are monomorphic, and the callee is 30 basic blocks or less, which enables inlining of most runtime primitives. Call sites belonging to blocks with higher execution frequencies are prioritized for inlining. Once inlining is complete, the fused IR containing inlined callees is then optimized using simple subgraph substitution patterns before machine code generation proceeds.

Algorithm 1 Code generation with basic block versioning

```
procedure genFun(assembler, function)
    workList ← ∅                                        ▷ Stack of block versions to be compiled
    versionList ← ∅                                     ▷ List of existing block versions
    getLabel(function.entryBlock, ∅, workList, versionList)    ▷ Begin compilation
    while workList not empty do
        block, ctx, label ← workList.pop()
        assembler.addLabel(label)                       ▷ Insert the label for this block
        if block.execCount is 0 then
            genStub(assembler, block, ctx, label)
        else
            for instr in block.instrs do                ▷ Generate code for each instruction
                genInstr(assembler, instr, ctx, workList, versionList);
            end for
        end if
    end while
end procedure
procedure getLabel(block, ctx, workList, versionList)
    if numVersions(block) ≥ maxvers then    ▷ If the version limit for this block was reached
        bestMatch ← findBestMatch(block, ctx, versionList);
        if bestMatch ≠ null then                        ▷ If a compatible match was found
            return bestMatch
        else
            ctx ← ∅           ▷ Make a generic version accepting all incoming contexts
        end if
    end if
    label ← newLabel();
    workList.push(⟨block, ctx, label⟩);         ▷ Queue the new version to be compiled
    versionList.append(⟨block, ctx, label⟩);    ▷ Add the new block version to the list
    return label
end procedure
```

Algorithm 2 Code generation with basic block versioning

```
procedure AddInt32.genInstr(assembler, instr, ctx, workList, versionList)
    assembler.addInt32(instr.getArgs())    ▷ Generate the add machine instruction
    ctx.setOutType(instr, int32)    ▷ The output type of AddInt32 is always int32.
        If an overflow occurs, the result is recomputed using AddFloat64
end procedure
procedure IsInt32.genInstr(assembler, instr, ctx, workList, versionList)
    argType ← ctx.getType(instr.getArg(0))
    if argType is int32 then
        ctx.setOutType(instr, true)
    else if argType ≠ ⊤ then
        ctx.setOutType(instr, false)
    else
        assembler.isInt32(instr.getArgs())    ▷ Generate code for the type test
        ctx.setOutType(instr, const)
    end if
end procedure
procedure Jump.genInstr(assembler, instr, ctx, workList, versionList)
    label ← getLabel(instr.target, ctx, workList, versionList)
    assembler.jump(label)
end procedure
procedure IfTrue.genInstr(assembler, instr, ctx, workList, versionList)
    arg ← instr.getArg(0)
    argType ← ctx.getType(arg)
    trueCtx ← ctx.copy()    ▷ New context for the true branch
    if arg instanceof IsInt32 then
        trueCtx.setType(arg.getArg(0), int32)
    end if
    if argType is true then
        trueLabel ← getLabel(instr.trueTarget, trueCtx, workList, versionList)
        assembler.jump(trueLabel)
    else if argType is false then
        falseLabel ← getLabel(instr.falseTarget, ctx, workList, versionList)
        assembler.jump(falseLabel)
    else
        trueLabel ← getLabel(instr.trueTarget, trueCtx, workList, versionList)
        falseLabel ← getLabel(instr.falseTarget, ctx, workList, versionList)
        assembler.compare(arg, true)    ▷ Compare the argument to true
        assembler.jumpIfEqual(trueLabel)
        assembler.jump(falseLabel)
    end if
end procedure
```

Machine code generation (see Algorithm 1) begins with the function's entry block and entry context pair being pushed on top of a stack which serves as a work list. This stack is used to keep track of block versions to be compiled, and enable depth-first generation of hot code paths. Code generation proceeds by repeatedly popping a block and context pair to be compiled off the stack. If the block to be compiled has an execution count of 0, stub code is generated out of line, which spills live variables, invalidates the generated machine code for the function and exits to the interpreter. Otherwise, code is generated by calling code generation methods corresponding to each IR instruction to be compiled in the current block, in order.

As each IR instruction in a block is compiled, information is both retrieved from and inserted into the current context. Information retrieved may be used to optimize the compilation of the current instruction (e.g. eliminate type tests). Instructions will also write their own output type in the context if known. The last instruction of a block, which must be a branch instruction, may potentially push additional compilation requests on the work stack. More specifically, branch instructions can request an assembler label for a version of a block corresponding to the current context at the branch instruction. If such a version was already compiled, the label is returned immediately. Otherwise, a new label is generated, the block and the current context are pushed on the stack, to be compiled later.

To avoid pathological cases where a large number of versions could be generated for a given basic block, we limit the number of versions that may be compiled. This is done with the `maxvers` parameter, which specifies how many versions can be compiled for any single block. Once this limit is hit for a particular block, requests for new versions of this block will first try to find if an inexact but compatible match for the incoming context can be found. An existing version is compatible with the incoming context if the value types assumed by the existing version are the same as, or supertypes of, those specified in the incoming context. If a compatible match is found, this match will be returned. If not, a generic version of the block will be generated, which can accept all incoming type combinations. When the `maxvers` parameter is set to zero, basic block versioning is disabled, and only the generic version is generated.

3.1 Type tags and runtime primitives

The current version of Higgs segregates values into a few categories based on type tags [13]. These categories are: 32-bit integers (`int32`), 64-bit floating point values (`float64`), garbage-collected references inside the Higgs heap (`refptr`), raw pointers to C objects (`rawptr`) and miscellaneous JavaScript constants (`const`). These type tags form a simple, first-degree notion of types which we use to drive the basic block versioning approach. The current implementation of basic block versioning in Higgs does not differentiate between references to object, arrays and functions, but instead lumps all of these under the reference pointer category. We do, however, distinguish between the boolean `true` and `false` constants to enable the propagation of type test results.

We believe that this choice of a simple type representation is a worthwhile way to investigate the effectiveness and potential of basic block versioning. Higgs implements JavaScript operators as runtime library functions written in an extended dialect of JavaScript, and most of these functions use type tags to do dynamic dispatch. As such, eliminating this first level of type tests is crucial to improving the performance of the system as a whole. Extending the system to use a more precise representation of types is part of future work.

```
function $rt_add(x, y) {
    if ($ir_is_i32(x)) { // If x is integer
        if ($ir_is_i32(y)) {
            if (var r = $ir_add_i32_ovf(x, y))
                return r;
            else // Handle the overflow case
                return $ir_add_f64($ir_i32_to_f64(x),
                                   $ir_i32_to_f64(y));
        } else if ($ir_is_f64(y))
            return $ir_add_f64($ir_i32_to_f64(x), y);
    } else if ($ir_is_f64(x)) { // If x is floating point
        if ($ir_is_i32(y))
            return $ir_add_f64(x, $ir_i32_to_f64(y));
        else if ($ir_is_f64(y))
            return $ir_add_f64(x, y);
    }

    // Evaluate arguments as strings and concatenate them
    return $rt_strcat($rt_toString(x), $rt_toString(y));
}
```

Fig. 4. Implementation of the `+` operator

Figure 4 illustrates the implementation of the primitive `+` operator. As can be seen, this function makes extensive use of low-level type test primitives such as `$ir_is_i32` and `$ir_is_f64` to implement dynamic dispatch based on the type tags of the input arguments. All other arithmetic and comparison primitives implement a similar dispatch mechanism.

3.2 Flow-based representation analysis

To provide a point of comparison and contrast the capabilities of basic block versioning with that of more traditional type analysis approaches, we have implemented a forward flow-based representation analysis which computes a fixed-point on the types of SSA values. The analysis is an adaptation of Wegbreit's algorithm as described in [26]. It is an intraprocedural constant propagation analysis which propagates the types of SSA values in a flow-sensitive manner.

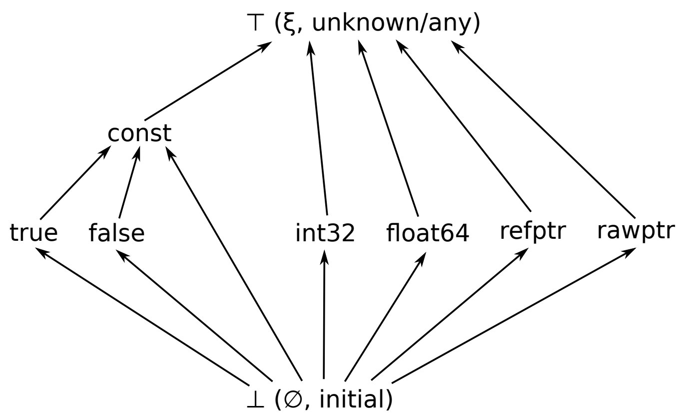

Fig. 5. Type lattice used by the representation analysis

Pseudocode for this analysis and some of its transfer functions is shown in Appendix A.

The representation analysis uses the same type representation (see Figure 5) as our basic block versioning implementation, and has similar type analysis capabilities. It is able to gain information from type tests and forward this information along branches. It is also able to deduce, in some cases, that specific branches will not be executed and ignore the effects of code that was determined dead.

We have also extended the flow-based algorithm to ignore basic blocks which are unexecuted (have an execution count of 0) at analysis time. This allows the analysis to ignore some code paths not executed up to now, which is useful in some cases, since primitive language operators often have multiple paths which can result in different output types. If presumed dead blocks turn out to be executed later, analysis results and associated compiled code will be invalidated at run time. This was done to make the analysis more competitive with basic block versioning which by construction ignores stubbed blocks, for which no compiled code was generated.

3.3 Limitations

There are a few important limitations to the current implementation of basic block versioning in Higgs. We do not, at this point, track the types of object properties. Global variables, which are properties of the global object in JavaScript, are also untracked. We do not account for interprocedural flow of type information either. That is, function parameter and return value types are assumed to be unknown. Finally, the current implementation of Higgs does not implement any kind of load-store forwarding optimization. These limitations are nontrivial to tackle due to factors such as the late-bound nature of JavaScript, the potential presence of the `eval` construct, dynamic addition and deletion of properties and the dynamic installation of getter-setter methods on object fields.

The results presented in this paper are entirely based on an intraprocedural implementation of basic block versioning which accounts only for the types of local variables and temporaries, in combination with aggressive inlining of library and method calls. Extending basic block versioning to take object identity, array and property type information into account constitutes future work.

4 Evaluation

To assess the effectiveness of basic block versioning, we have used a total of 24 benchmarks from the classic SunSpider and Google V8 suites. A handful of benchmarks from both suites were not included in our tests because the current Higgs implementation does not yet support them.

Figure 6 shows counts of dynamically executed type tests across all benchmarks for the representation analysis and for basic block versioning with various block version limits. These counts are relative to a baseline which has the version limit set to 0, and thus only generates a default, unoptimized version of each basic block, without attempting to eliminate any type tests. As can be seen from the counts, the analysis produces a reduction in the number of dynamically executed type tests over the unoptimized default on every benchmark. The basic block versioning approach does at least as well as the analysis, and almost always significantly better. Surprisingly, even with a version cap as low as 1 version per basic block, the versioning approach is often competitive with the representation analysis.

Raising the version cap reduces the number of tests performed with the versioning approach in a seemingly asymptotic manner as we get closer to the limit of what is achievable with our implementation. The versioning approach does remarkably well on the `bits-in-byte` benchmark, with a reduction in the number of type tests by a factor of over 50. This benchmark (see Figure 7) is an ideal use case for our versioning approach. It is a tight loop performing bitwise and arithmetic operations on integers which are all stored in local variables. The versioning approach performs noticeably better than the analysis on this test because it is able to hoist a type test on the function parameter `b` out of a critical loop. The type of this parameter is initially unknown when entering the function. The analysis on its own cannot achieve this, and so must repeat the test every loop iteration. Note that neither the analysis nor the basic block versioning approach need to test the type of `c` at run time because the variable is initialized to an integer value before loop entry, and integer overflow never occurs, so the overflow case remains a stub. The `bitwise-and` benchmark operates exclusively on global variables, for which our system cannot extract types, and so neither the type analysis nor the versioning approach show any improvement over baseline for this benchmark.

A breakdown of relative type test counts by kind, averaged accross all benchmarks (using the geometric mean) is shown in Figure 8. We see that the versioning approach is able to achieve better results than the representation analysis across each kind of type test. The `is_refptr` category shows the least improve-

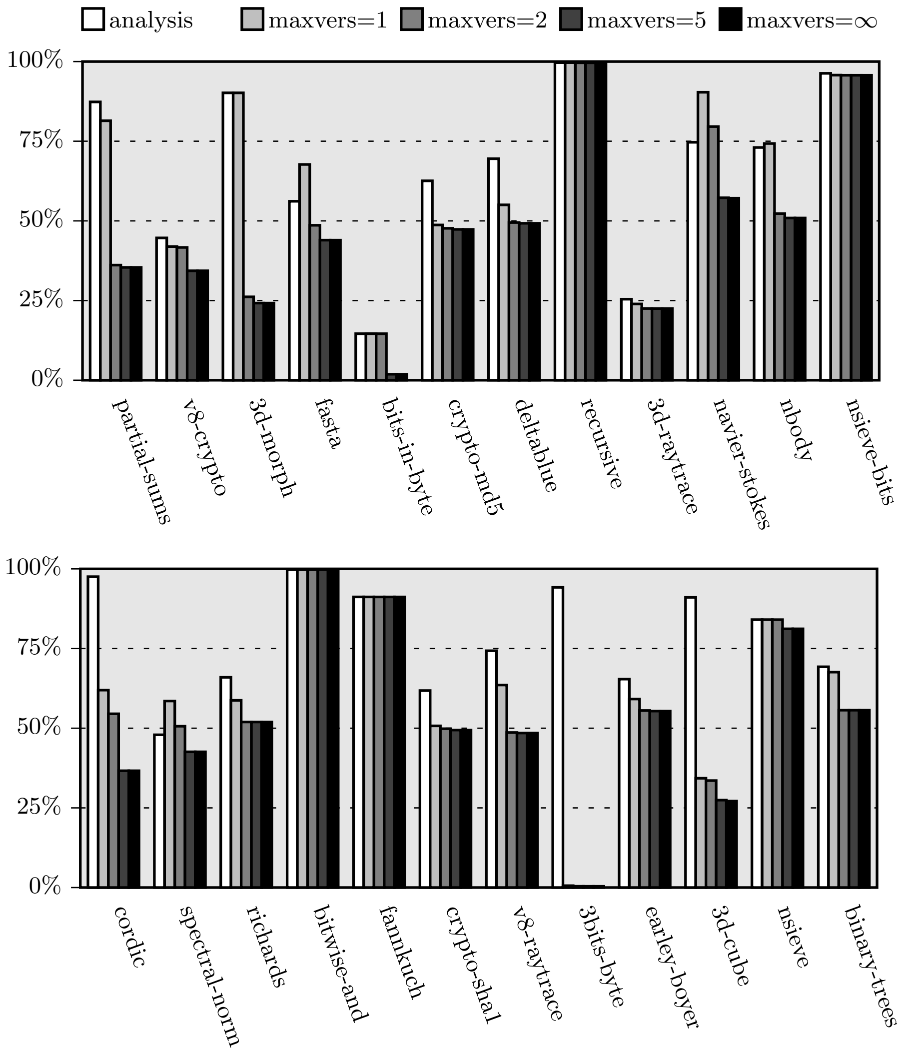

Fig. 6. Counts of dynamic type tests (relative to baseline)

```
function bitsinbyte(b) {
    var m = 1, c = 0;
    while(m < 0x100) {
        if(b & m) c++;
        m <<= 1;
    }
    return c;
}

function TimeFunc(func) {
    var x, y, t;
    for(var x = 0; x < 350; x++)
    for(var y = 0; y < 256; y++) func(y);
}
TimeFunc(bitsinbyte);
```

Fig. 7. SunSpider bits-in-byte benchmark

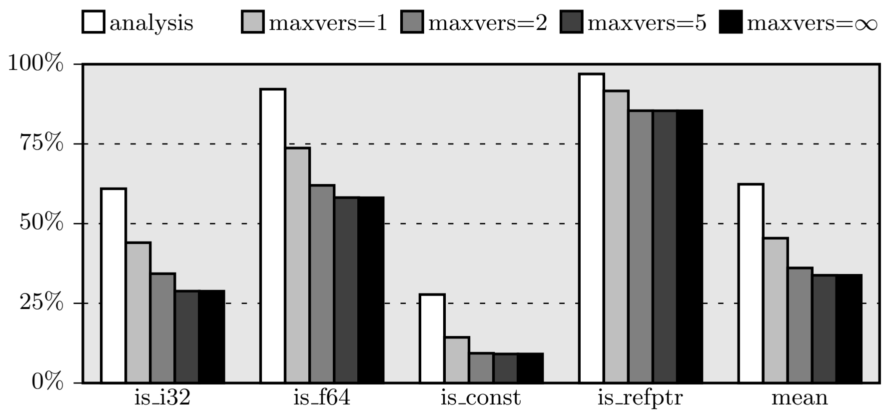

Fig. 8. Type test counts by kind of type test (relative to baseline)

ment. This is likely because property access primitives are very large, and thus seldom inlined, limiting the ability of both basic block versioning and the analysis to propagate type information for reference values. We note that versioning is much more effective than the analysis when it comes to eliminating `is_float64` type tests. This is probably because integer and floating point types often get intermixed, leading to cases where the analysis cannot eliminate such tests. The versioning approach has the advantage that it can replicate and detangle integer and floating point code paths. A limit of 5 versions per block eliminates 64% of type tests on average (geometric mean), compared to 33% for the analysis.

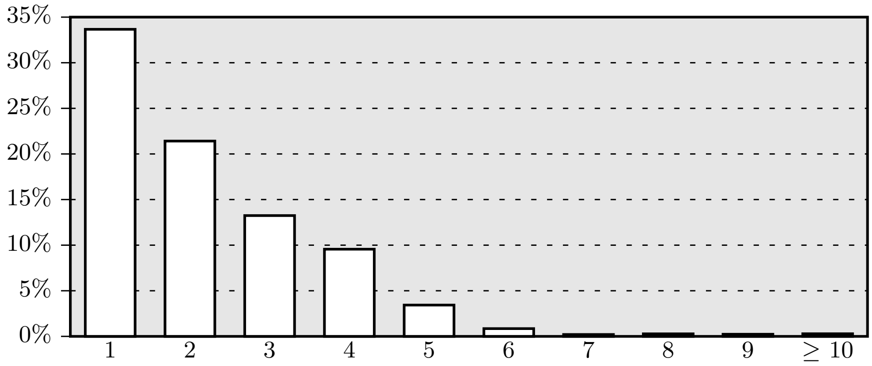

Fig. 9. Relative occurrence of block version counts

Figure 9 shows the relative proportion of blocks for which different numbers of versions were generated, averaged accross all benchmarks (geometric mean). As one might expect, the relative proportion of blocks tends to steadily decrease as the number of versions is increased. Most blocks only have one or two versions, and less than 9% have 5 versions or more. There are very few blocks which have 10 versions or more. These are a small minority, but such pathological cases do occur in practice.

The function generating the most block versions in our tests is `DrawLine` from the `3d-cube` benchmark, which produces 32 versions of one particular block. This function draws a line in screen space between point coordinates `x1,y1` and `x2,y2`. Multiple different values are computed inside `DrawLine` based on these points. Each of the coordinate values can be either integer or floating point, which results in a situation where there are several live variables, all of which can have two different types. This creates an explosion in the number of versions of blocks inside this function as basic block versioning tries to account for all possible type combinations of these values. In practice, the values are either all integer, or all floating point, but our implementation of basic block versioning is currently unable to take advantage of this helpful fact. We have experimentally verified that, in fact, only 17 of the 32 versions generated in `DrawLine` are actually executed. A strategy for addressing this problem is discussed in Section 6.

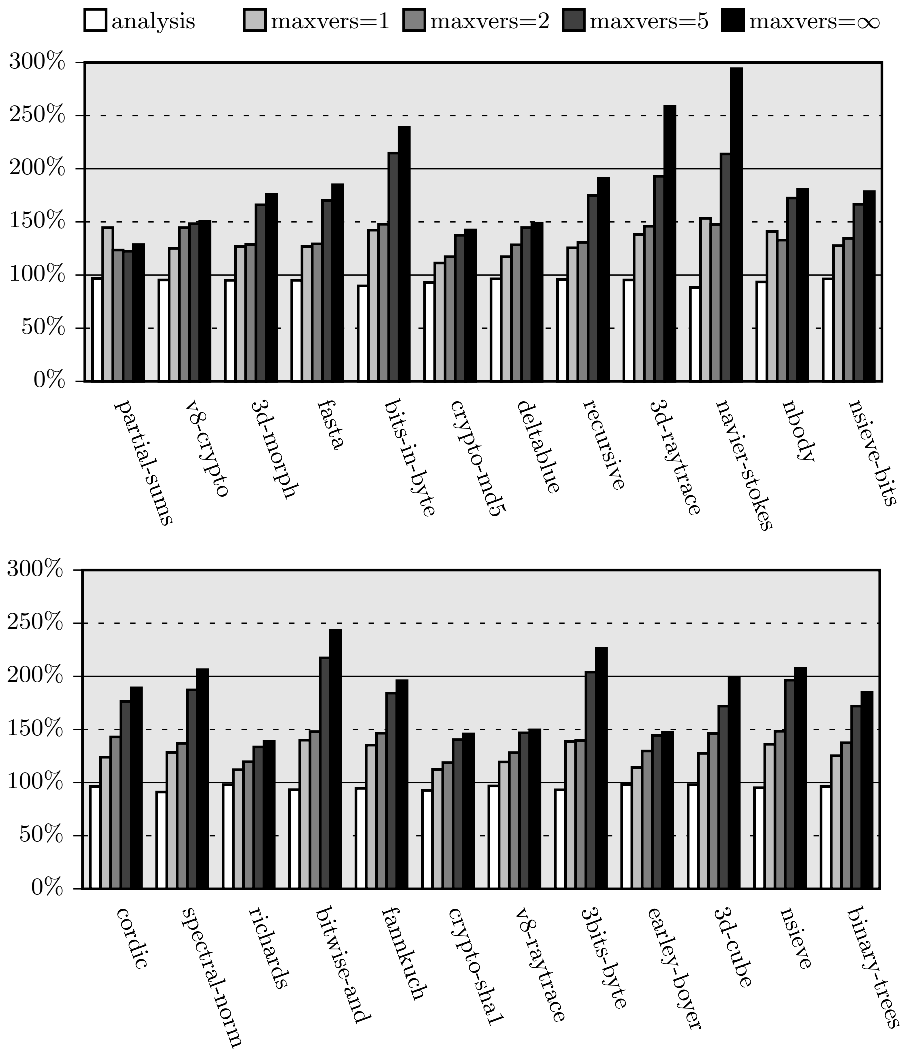

Fig. 10. Code size growth for different block version limits

The effects of basic block versioning on the total generated code size are shown in Figure 10. It is interesting to note that the representation analysis almost always results in a slight reduction in code size. This is because the analysis allows the elimination of type tests and the generation of more optimized code, which is often smaller. On the other hand, basic block versioning can generate multiple versions of basic blocks, which results in more generated code. The volume of generated code does not increase linearly with the block version cap. Rather, it tapers off as a limited number of versions tends to be generated for each block. Even without a block version limit, the code size is less than double that of the baseline in most cases. A limit of 5 versions per block results in a mean code size increase of 69%.

5 Related Work

There have been multiple efforts to devise type analyses for dynamic languages. The Rapid Atomic Type Analysis (RATA) [17] is an intraprocedural flow-sensitive analysis based on abstract interpretation which aims to assign unique types to each variable inside of a function. Attempts have also been made to define formal semantics for a subset of dynamic languages such as JavaScript [4], Ruby [10] and Python [3], sidestepping some of the complexity of these languages and making them more amenable to traditional type inference techniques. There are also flow-based interprocedural type analyses for JavaScript based on sophisticated type lattices [15][16]. Such analyses are usable in the context of static code analysis, but take too long to execute to be usable in compilation and do not deal with the complexities of dynamic code loading.

More recently, work done by Brian Hackett et al. resulted in an interprocedural hybrid type analysis for JavaScript suitable for use in production JIT compilers [14]. This analysis represents a great step forward for dynamic languages, but as with other type analyses, must assign one type to each value, which makes it vulnerable to imprecise type information polluting analysis results. Basic block versioning could potentially help improve on the results of such an analysis by hoisting tests out of loops and generating multiple optimized code paths where appropriate.

Trace compilation aims to record long sequences of instructions executed inside of hot loops [12]. Such sequences of instructions often make optimization easier. Type information can be accumulated along traces and used to specialize the code to remove type tests [11], overflow checks [23] and unnecessary allocations [6]. Tracing is similar to basic block versioning, in that context updating works on essentially linear code fragments and accumulates type information during compilation. However, trace compilation incurs several difficulties and corner cases in practice, such as the potential for trace explosion if there is a large number of control-flow paths going through a loop, and poor capability to deal with code that is not loop-based. Work on trace regions by Bebenita et al. [5] introduces traces with join nodes. These join nodes can potentially eliminate tail duplication among traces and avoid the problem of trace explosion, but also makes the compiler architecture more complex.

Basic block versioning bears some similarities to classic compiler optimizations such as loop unrolling [9], loop peeling [24], and tail duplication, in that it achieves some of the same results. Tail duplication and loop peeling are used in the formation of hyperblocks [18], which are sets of basic blocks grouped together, such that control-flow may enter only into one of the blocks, but may exit at multiple locations. This structure was designed to facilitate the optimization of large units of code for VLIW architectures. A parallel can be drawn between basic block versioning and Partial Redundancy Elimination (PRE) [19] in that the versioning approach seeks to eliminate and hoists out of loops a specific kind of redundant computation, that of dynamic type tests.

Basic block versioning is also similar to the idea of node splitting [25]. This technique is an analysis device designed to make control-flow graphs reducible and more amenable to analysis. The path splitting algorithm implemented in the SUIF compiler [22] aims at improving reaching definition information by replicating control-flow nodes in loops to eliminate joins. Unlike basic block versioning, these algorithm cannot gain information from type tests. The algorithms as presented are also specifically targeted at loops, while basic block versioning makes no special distinction. Similarly, a static analysis which replicates code to eliminate conditional branches has been developed [20]. This algorithm operates on a low-level intermediate representation, is intended to optimize loops and does not specifically eliminate type tests.

Customization is a technique developed to optimize the SELF programming language [7] which compiles multiple copies of methods, specialized based on the receiver object type. Similarly, type-directed cloning [21] clones methods based on argument types, which can produce more specialized code using richer type information. The work of Maxime Chevalier-Boisvert et al. on Just-In-Time (JIT) specialization for MATLAB [8] and similar work done for the MaJIC MATLAB compiler [2] tries to capture argument types to dynamically compile optimized copies of functions. All of these techniques are forms of type-driven code duplication aimed at enhancing type information. Basic block versioning operates at a lower level of granularity, which allows it to find optimization opportunities inside of method bodies by duplicating code paths.

6 Future Work

Our current implementation only tracks type information intraprocedurally. It would be desirable to extend basic block versioning in such a way that type information can cross function call boundaries. This could be accomplished by allowing functions to have multiple entry point blocks, specialized based on context information coming from callers. Similarly, call continuation blocks (return points) could also be versioned to allow information about return types to flow back into the caller.

Another obvious extension of basic block versioning would be to collect more detailed type information. For example, we may wish to propagate information about global variable types, object identity and object field types. It may also be desirable, in some cases, to know the exact value of some variable or object field, particularly if this value is likely to remain constant. Numerical range information could potentially be collected to help eliminate bound and overflow checks.

Basic block versioning, as we have implemented it, sometimes generates versions that account for type combinations that never occur in practice. This could potentially be addressed by generating stubs for the targets of cloned conditional branches. Higgs already produces stubs for unexecuted blocks, but generates all requested versions of a block if the block was ever executed in the past. Producing stubs for cloned branches would delay the generation of machine code for these branch targets until we know for a fact that they are executed, avoiding code generation for unnecessary code paths. The choice of where to generate stubs could potentially be guided by profiling data.

Some of the information accumulated and propagated by basic block versioning may not actually be useful for optimization. This is likely to become a bigger problem if the approach is extended to work accross function call boundaries, or if more precise type and constant information is accumulated. An interesting avenue may be to choose which information to propagate based on usefulness. That is, the most frequently executed type tests are probably the ones we should focus our resources on. These tests should be dynamically identified through profiling and used to decide which information to propagate.

7 Conclusion

We have introduced a novel compilation technique called context-driven basic block versioning. This technique combines code generation with type analysis to produce more optimized code through the accumulation of type information during compilation. The versioning approach is able to perform optimizations such as automatic hoisting of type tests and efficiently detangles code paths along which multiple numerical types can occur. Our experiments show that in most cases, basic block versioning eliminates significantly more dynamic type tests than is possible using a traditional flow-based type analysis. It eliminates 64% of type tests on average with a limit of 5 versions per block, compared to 33% for the analysis, and never performs worse than such an analysis.

Basic block versioning trades code size for performance. Such a tradeoff is often desirable, particularly for performance-critical application kernels. We have empirically demonstrated that although our implementation of basic block versioning does increase code size, the resulting increase is reasonably moderate, and can easily be limited with techniques as simple as a hard limit on the number of versions per basic block. In our experiments, a limit of 5 versions per block results in a mean code size increase of 69%. More sophisticated implementations that adjust the amount of code replication allowed based on the execution frequency of functions are certainly possible.

Basic block versioning is a simple and practical technique suitable for integration in real-world compilers. It requires little implementation effort and can offer important advantages in JIT-compiled environments where type analysis is often difficult. Dynamic languages, which perform a large number of dynamic type tests, stand to benefit the most.

Higgs is open source and the code used in preparing this publication is available on GitHub[2].

References

1. ECMA-262: ECMAScript Language Specification. European Association for Standardizing Information and Communication Systems (ECMA), Geneva, Switzerland, fifth edn. (2009)
2. Almási, G., Padua, D.: MaJIC: compiling MATLAB for speed and responsiveness. In: Proceedings of the 2002 conference on Programming Language Design and Implementation (PLDI). pp. 294–303. ACM New York (May 2002)
3. Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D.: RPython: a step towards reconciling dynamically and statically typed OO languages. In: Proceedings of the 2007 Dynamic Languages Symposium (DLS). pp. 53–64. ACM New York (2007)
4. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Proceedings of ECOOP 2005, pp. 428–452. Springer Berlin Heidelberg (2005)
5. Bebenita, M., Chang, M., Wagner, G., Gal, A., Wimmer, C., Franz, M.: Trace-based compilation in execution environments without interpreters. In: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ). pp. 59–68. ACM New York (2010)
6. Bolz, C.F., Cuni, A., FijaBkowski, M., Leuschel, M., Pedroni, S., Rigo, A.: Allocation removal by partial evaluation in a tracing jit. In: Proceedings of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM). pp. 43–52. ACM New York (2011)
7. Chambers, C., Ungar, D.: Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In: Proceedings of the 1989 conference on Programming Language Design and Implementation (PLDI). pp. 146–160. ACM New York (Jun 1989)
8. Chevalier-Boisvert, M., Hendren, L., Verbrugge, C.: Optimizing MATLAB through just-in-time specialization. In: Proceedings of the 2010 international conference on Compiler Construction (CC). pp. 46–65. Springer Berlin Heidelberg (2010)
9. Davidson, J.W., Jinturkar, S.: Aggressive loop unrolling in a retargetable, optimizing compiler. In: Proceedings of the 1996 international conference on Compiler Construction (CC). pp. 59–73. Springer Berlin Heidelberg (1996)
10. Furr, M., An, J.h.D., Foster, J.S., Hicks, M.: Static type inference for ruby. In: Proceedings of the 2009 ACM Symposium on Applied Computing (SAC). pp. 1859–1866. ACM New York (2009)

[2] https://github.com/maximecb/Higgs/tree/cc2014

11. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: Proceedings of the 2009 conference on Programming Language Design and Implementation (PLDI). pp. 465–478. ACM New York (2009)
12. Gal, A., Probst, C.W., Franz, M.: HotpathVM: an effective jit compiler for resource-constrained devices. In: Proceedings of the 2nd international conference on Virtual Execution Environments (VEE). pp. 144–153. ACM New York (2006)
13. Gudeman, D.: Representing type information in dynamically typed languages (1993)
14. Hackett, B., Guo, S.y.: Fast and precise hybrid type inference for JavaScript. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI). pp. 239–250. ACM New York (Jun 2012)
15. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Proceedings of the 16th International Symposium on Static Analysis (SAS). pp. 238–255. Springer Berlin Heidelberg (2009)
16. Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. In: Proceedings 17th International Static Analysis Symposium (SAS). Springer Berlin Heidelberg (September 2010)
17. Logozzo, F., Venter, H.: RATA: rapid atomic type analysis by abstract interpretation; application to JavaScript optimization. In: Proceedings of the 2010 international conference on Compiler Construction (CC). pp. 66–83. Springer Berlin Heidelberg (2010)
18. Mahlke, S.A., Lin, D.C., Chen, W.Y., Hank, R.E., Bringmann, R.A.: Effective compiler support for predicated execution using the hyperblock. In: ACM SIGMICRO Newsletter. vol. 23, pp. 45–54. IEEE Computer Society Press (1992)
19. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM 22(2), 96–103 (Feb 1979)
20. Mueller, F., Whalley, D.B.: Avoiding conditional branches by code replication. In: Proceedings of the 1995 conference on Programming Language Design and Implementation (PLDI). pp. 56–66. ACM New York (1995)
21. Plevyak, J., Chien, A.A.: Type directed cloning for object-oriented programs. In: Proceedings of the Workshop for Languages and Compilers for Parallel Computing (LCPC). pp. 566–580 (1995)
22. Poletto, M.A.: Path splitting: a technique for improving data flow analysis. Ph.D. thesis, MIT Laboratory for Computer Science (1995)
23. Sol, R., Guillon, C., Quinto Pereira, F., Bigonha, M.: Dynamic elimination of overflow tests in a trace compiler. In: Knoop, J. (ed.) Proceedings of the 2011 international conference on Compiler Construction (CC), pp. 2–21. Springer Berlin Heidelberg (2011)
24. Song, L., Kavi, K.M.: A technique for variable dependence driven loop peeling. In: Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on. pp. 390–395. IEEE (2002)
25. Unger, S., Mueller, F.: Handling irreducible loops: optimized node splitting versus dj-graphs. ACM Transactions on Programming Languages and Systems (TOPLAS) 24(4), 299–333 (Jul 2002)
26. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(2), 181–210 (Apr 1991)

A First Appendix

Algorithm 3 Type propagation analysis

```
procedure typeProp(function)
    outTypes ← ∅
    edgeTypes ← ∅
    visited ← ∅                                   ▷ Set of visited control-flow edges
    workList ← {⟨null, function.entryBlock⟩}
    while workList not empty do
        edge ← workList.dequeue()
        block ← edge.succ
        if block.execCount is 0 then
            continue                              ▷ Ignore yet unexecuted blocks (stubs)
        end if
        visited.add(edge)
        curTypes ← ∅                              ▷ Merge type info from predecessors
        for edge in block.incoming do
            if edge in visited then
                curTypes ← curTypes.merge(edgeTypes.get(edge))
            end if
        end for
        for phiNode in block.phis do
            t ← evalPhi(phiNode, block, visited)
            curTypes.set(phi, t)
            outTypes.set(phi, t)
        end for
        for instr in block.instrs do
            t ← evalInstr(instr, curTypes)
            curTypes.set(instr, t)
            outTypes.set(instr, t)
        end for
    end while
    return outTypes;
end procedure
```

Algorithm 4 Transfer functions for the type propagation analysis

```
procedure evalPhi(phiNode, block, visited)
    t ← ⊥
    for edge in block.incoming do
        if edge in visited then
            predType = getType(edgeTypes.get(edge), phiNode.getArg(edge))
            if predType is ⊥ then
                return ⊥
            end if
            t = t.merge(predType)
        end if
    end for
    return t
end procedure
procedure AddInt32.evalInstr(instr, curTypes)
    return int32    ▷ The output type of AddInt32 is always int32. If an overflow
  occurs, the result is recomputed using AddFloat64
end procedure
procedure IsInt32.evalInstr(instr, curTypes)
    argType ← getType(curTypes, instr.getArg(0))
    if argType is ⊥ then                    ▷ If the argument type is not yet evaluated
        return ⊥
    else if argType is ⊤ then                       ▷ If the argument type is unknown
        return const
    else if argType is int32 then
        return true
    else
        return false
    end if
end procedure
procedure IfTrue.evalInstr(instr, curTypes)
    arg ← instr.getArg(0)
    argType ← getType(curTypes, arg)
    if argType is ⊥ then
        return ⊥
    end if
    testVal ← null
    testType ← ⊤
    if arg instanceof IsInt32 then
        testVal ← arg.getArg(0)    ▷ Get the SSA value whose type is being tested
        testType ← int32
    end if
    if argType is true or argType is const or argType is ⊤ then
        queueSucc(instr.trueTarget, typeMap, testVal, testType) ▷ Queue the true
  branch, and propagate the test value's type (if applicable)
    end if
    if argType is false or argType is const or argType is ⊤ then
        queueSucc(instr.falseTarget, typeMap, null, ⊤)
    end if
    return ⊥
end procedure
```