Building JIT Compilers for Dynamic Languages with Low Development Effort

Baptiste Saleil
Université de Montréal
Montréal, Québec, Canada
baptiste.saleil@umontreal.ca

Marc Feeley
Université de Montréal
Montréal, Québec, Canada
feeley@iro.umontreal.ca

Abstract

Building high performance virtual machines for dynamic languages usually requires significant development effort. They may require an interpreter and one or more compilation phases to generate efficient code. In addition, they may require several static analyses using custom intermediate representation(s).

This paper presents techniques used to implement virtual machines for dynamic languages with relatively low development effort and good performance. These techniques allow compiling directly from the abstract syntax tree to target machine code while still enabling useful optimizations and without using any intermediate representation.

We have used these techniques to implement a JIT compiler for Scheme. We show that performance of the generated code competes with the code generated by mature Scheme implementations.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; *Compilers*;

Keywords Virtual Machine, Just-In-Time Compilation, Dynamic Language, Code Specialization, Scheme

ACM Reference Format:
Baptiste Saleil and Marc Feeley. 2018. Building JIT Compilers for Dynamic Languages with Low Development Effort. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '18), November 4, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3281287.3281294

VMIL '18, November 4, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6071-5/18/11…$15.00
https://doi.org/10.1145/3281287.3281294

1 Introduction

Building high performance Just-In-Time (JIT) compilers requires significant development effort. They typically have a complex architecture involving an interpretation phase followed by one or more compilation phases, each with its own set of optimizations. However, JIT compilers allow the efficient implementation of dynamic languages by adapting the generated code to the execution of the program at run time using some run time monitoring.

With dynamic languages, the dynamic properties of the source program are not generally statically known. This means that to implement the dynamic features of these languages, additional checks and operations are inserted in the generated code, hurting performance. To decrease this impact, compilers may use static analysis on low level intermediate representations (IR) coupled with profiling to infer the dynamic properties and decrease the number of checks and operations inserted in the generated code. Designing and maintaining an IR makes the architecture more complex and further increases development effort.

Previous work [2, 3, 16, 21, 22] suggests building interpreters to implement dynamic languages and to rely on other systems to optimize the interpreter itself instead of the executed programs. The implementation then relies on the optimizations of these systems to efficiently execute programs. These techniques significantly decrease development effort but they limit the control over the compilation process.

More recently, techniques have been designed to quickly build JIT compilers for dynamic languages with relatively good performance [5, 9, 15, 20]. This is particularly interesting to implement Domain Specific Languages, for prototyping or if development resources are limited.

One of these techniques, Basic Block Versioning [4, 5] (BBV) is based on a JIT lazy compilation design that allows the compiler to generate multiple versions of basic blocks, each specialized according to dynamic properties observed for the current execution of the program without requiring previous interpretation nor profiling phases. An interesting advantage of BBV is that it allows compilers to decrease the number of extra type checks and other operations using a single compilation pass by duplicating the basic blocks on-the-fly.

This paper presents simple techniques used on top of BBV allowing to further decrease development effort and thus

quickly implement JIT compilers for dynamic languages. We show that a compiler for dynamic languages can compile directly from the Abstract Syntax Tree (AST) to machine code without requiring the use of an IR, by limiting static analysis and without using a complex architecture, yet achieve relatively good performance.

We also present the implementation of LC[1]. LC is a research oriented JIT compiler for Scheme using these techniques and we evaluate the impact they have on the performance of the generated code.

The rest of the paper is structured as follows. Section 2 presents BBV and how we adapted it to a compiler that is not using any IR. Section 3 shows how constant propagation and constant folding can easily be applied in our implementation of BBV. Section 4 shows how we implemented register allocation with relatively good performance without using any static analysis. Section 5 presents how code specialization can be used to decrease the performance impact introduced by the operations necessary to implement dynamic typing. In section 6, we present the results of our experiments. Related work is presented in section 7 followed by a brief conclusion.

2 Basic Block Versioning

BBV is a JIT compilation technique that specializes the generated code according to the dynamic properties of the source program observed at run time. Instead of relying on profiling or analysis, BBV uses a design based on lazy compilation of basic blocks to discover new information. When generating code for a basic block *b*, if *b* has a single successor block, this block is generated immediately after *b* is generated. If *b* has several branches out, instead of generating the code for the successor blocks, each branch out of block *b* goes to a stub that calls back to the compiler to generate the corresponding successor block. Then the generated code is executed. Each branch may cause the compiler to discover new information about the dynamic properties of the program. For example, if *b* represents a type test on a variable *v*, it has two branches out. The first branch is taken if the test succeeds and the second is taken if the test fails. If the test succeeds, the stub associated with this branch is triggered and the compiler is called. The compiler now knows the type of *v* on this branch so it checks in a code cache if a version of the next basic block has already been specialized using this information. If the version exists, the test is patched to jump to this version. If the version does not exist, it is generated, added to the cache, and the test is patched to jump to this version.

Figure 1 shows a Scheme function computing the factorial of its argument `n` using tail recursion. This example is used throughout the paper.

Figure 2 shows the code generated for the `fact` function by a compiler using BBV. When the function `fact` is called for the first time, the compiler is triggered and compilation

[1]https://github.com/bsaleil/lc

```
(define (fact n r)
  (if (= n 0)
      r
      (fact (- n 1) (* n r))))

(fact (read) 1)
```

Figure 1. Scheme tail recursive function computing the factorial of its argument.

starts with no type information for `n` and `r`. The primitive `=` is overloaded over several operand types. For example, it can be used to test equality of *fixnums* (Scheme small integers) and *flonums* (Scheme floating point numbers). Because at this point the type of `n` is unknown, type tests must be generated to dispatch execution to the specialized `=` operator.

The compiler then first generates a code sequence to determine if `n` is a fixnum. Two stubs are created to handle both branches. When the function is executed, assuming `n` is a fixnum, the stub associated with the true branch is triggered, the type of `n` is discovered for this call, and compilation continues specializing code using this information.

The figure shows the compilation state after the execution finishes, when `fact` is called with `n` being a fixnum. If `fact` is later called with `n` being a flonum, the stub `Stub1` is triggered and using the same process, other versions of the same blocks are generated, specialized for `n` being a flonum.

This example shows how BBV is efficient at removing type checks. The type of `n` is determined once for the primitive `=` with one or two type tests and propagated for the rest of the function body's compilation. It is then not needed to determine its type dynamically for the primitives `-` and `*`. A second type dispatch is executed per recursive call to determine the type of `r`. This means that, without analyzing nor profiling code, only two type dispatches are inserted for this example whereas a naive compiler would insert four (three for *n* and one for *r*).

Other work has demonstrated that BBV is efficient at removing type checks to generate fast code [4, 5]. Chevalier-Boisvert and Feeley [6] later extended BBV to work interprocedurally and Saleil and Feeley [19] extended code specialization to work interprocedurally in the presence of higher order functions. These extensions allow BBV to remove even more type tests and generate even faster code.

In our example, if the compiler uses these extensions and is able to propagate discovered type information through function calls and returns, the type of `r` is never tested dynamically because the compiler knows that it is a fixnum (constant 1) for the first call, and its type remains the same after the multiplication and at the entry point of the recursive call. The type of `n` is tested in the first call of the function for the primitive `=`, its type is discovered and is also propagated

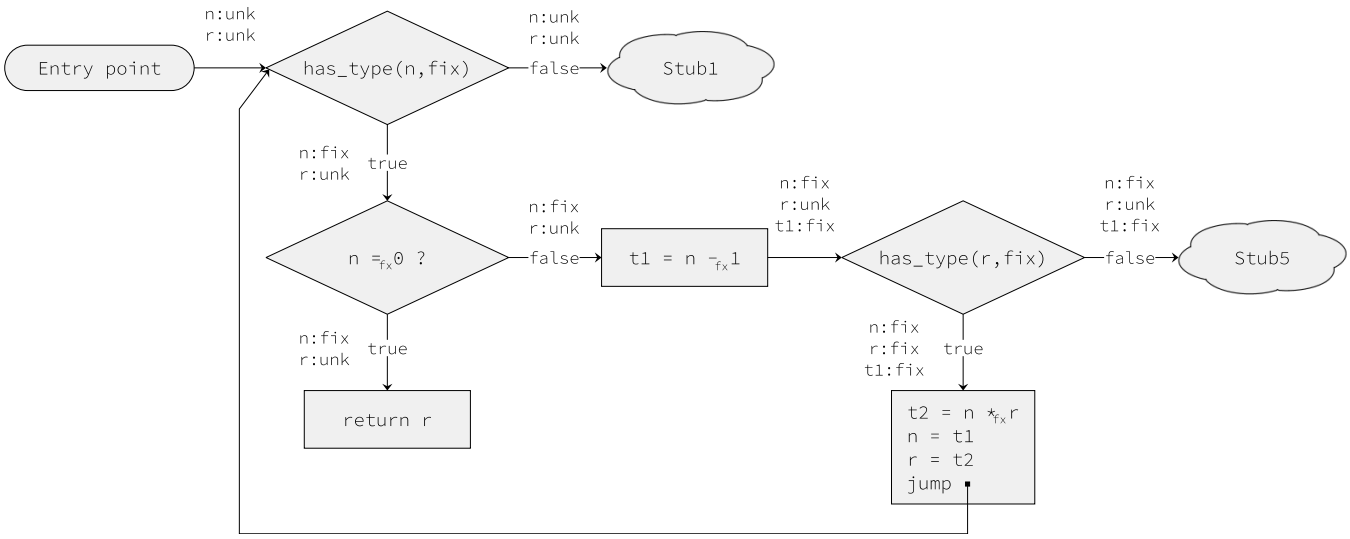

Figure 2. Code generated for the `fact` function of figure 1.

for the second call (in effect interprocedural BBV creates a second version of the function specialized for both `n` and `r` being fixnums). Consequently, the type of `n` is not tested in the recursive calls. This means that in this example, interprocedural BBV allows executing a single dynamic type test for the call to `fact`, regardless of the value of `n`.

2.1 Basic Block Versioning from AST

We adapted BBV to work in the absence of a low level IR. This allows building optimizing JIT compilers without requiring to design and implement an IR, simplifying the development process thus decreasing development effort. If the compiler does not use a low level IR, it has no concept of basic blocks thus BBV, as presented above, cannot be implemented. Instead, we extend BBV to work on AST nodes.

When the compiler needs to execute an expression, it first builds a chain of objects we call *lazy code objects* (LCO) from the AST representing the expression. The compiler stops building the chain if all nodes of the AST have been visited or if a node with several successors (i.e. a node that may cause compilation stubs to be generated) is visited. An LCO contains a reference to the AST node it is associated with and a *generator*. A generator is a piece of code (e.g. a function) that, given a compilation context, is able to generate a specialized version of the AST node it is associated with. When a generator is called, it *(i)* generates specialized code for the context, *(ii)* augments the context using information it just discovered and *(iii)* calls the generator of the successor node if it has a single successor (i.e. if it does not generate compilation stubs).

All the techniques presented in this paper are local to an LCO. When a generator is called, it generates code optimized for the AST node it is associated with using information propagated up to this node and information associated with the node itself, without considering extended control or data flow.

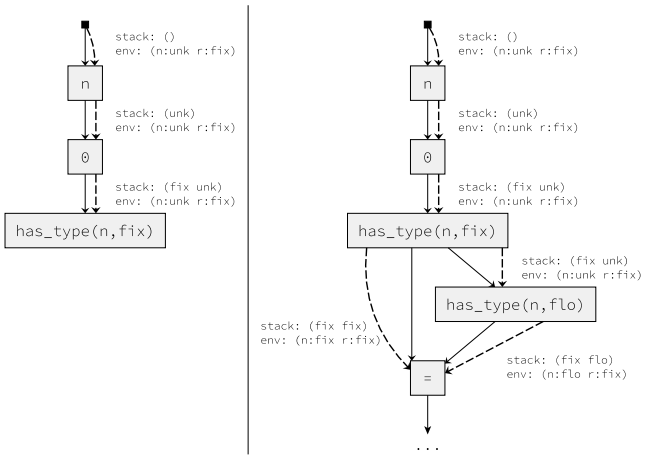

Figure 3. Two steps of the LCO chain creation process for the Scheme expression `(= n 0)` of the `fact` function. Solid arrows represent the successor links. Dashed arrows show how the generators are called and which compilation contexts are used.

In the case of LC, a compilation context contains a stack of types representing the types of the values in the current frame, and the local environment, including the types of the local variables. If type information is not known, the type unknown is used.

Figure 3 shows an example of code generation for the Scheme expression `(= n 0)` of the `fact` function. To execute the function, and because the chain does not exist yet, the

compiler first builds the chain associated with this expression. Because the Scheme primitive = is overloaded over several operand types, type checks LCO are created before creating the LCO associated with the = AST node. For this Scheme expression, the first node in the chain is the loading of the variable n, then the constant 1 then a type check for the primitive =.

Because this check has several successors (depending on the result of the check), the compiler stops building the chain. All the generators of the chain are then sequentially called (represented by dashed lines in the figure), starting with an initial context with an empty stack and no information on the type of n. Because interprocedural BBV is used, the compiler knows that r is a fixnum. When a generator is called, it generates the code corresponding to its node and updates the context before calling the generator of its successor. Once all generators have been called, the compiler jumps to the first instruction generated from the chain and the generated code is executed. If the type check succeeds, the next chain, containing the node for the primitive = is generated. If this piece of code is later executed with n being a flonum, the check fails, and using the same process, the rest of the chain is built and the generators are called. The right side of the figure shows the compilation state after this piece of code has been executed with n being a fixnum and a flonum. We can see that in both cases, the same LCO is used for the = node, but because the generator associated with this node has been called with two different contexts two different specialized versions of the equality operator are generated.

Using this design allows generation of code using a single pass on the AST, without visiting the nodes that are not executed, meaning that dead code is never executed nor generated.

2.2 Tail Position Detection

An interesting side effect of building a chain of LCO is that no additional pass on the AST is needed to determine if a function call is in tail position, which is required to implement the Scheme programming language. In LC, the LCO chain is built recursively meaning that each time an LCO is created, its successor LCO has already been created. When creating an LCO, if the AST node it is associated with represents a function return, the LCO can be flagged as *return LCO*. When a generator associated with a function call node is called, it can check if the successor LCO is flagged as *return LCO*. If it is the case, the call is in tail position. This is how LC detects that the recursive call of the `fact` function is in tail position.

3 Constant Propagation

Constant propagation and constant folding are classical optimizations and are required to generate efficient code but these optimizations require additional static analysis on IR,

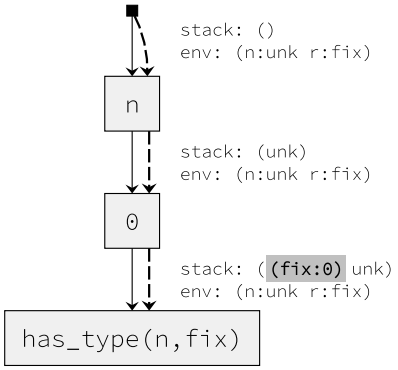

Figure 4. Contexts used for generators of the first chain of the fact function augmented with constant information.

or additional passes on the AST. However, BBV and the way we adapted it allows both optimizations without additional analyses or passes.

When a generator is called, it generates specialized code for the node it is associated with. It then updates the context and calls the generator of the successor with this updated context.

In the case the LCO is associated with a constant node (e.g. the node associated with `0` in figure 3), a naive stack-based compiler would generate code to push the value to the execution stack, add its type to the stack in the context and call the generator of the successor.

The type representation in the context can be augmented to include constant information. For a constant node, the compiler would generate no code, add the constant and its type to the context and call the generator of the successor LCO. In the same way, constants are propagated with the type of the local variables in addition to the type stack. This addition to the compilation context allows constant propagation.

As an example, figure 4 shows how the generators of the first chain of the `fact` function are called using a context augmented with constant information. When the generator associated with the AST node `0` is called, no code is generated and the type `(fix:0)` is added to the stack and propagated, meaning that the next nodes are specialized using this information.

When a generator associated with a primitive is called, it uses context information to generate optimized code. For example, in the case of a binary arithmetic operator, if the type of the two operands (i.e. the two types on top of the context stack) represent constants, no code is generated, the result of the operation, computed at compilation time, is added to the context and propagated as a constant. In the case the operator is a comparison operator the generator can directly call one of its successors depending on the result, without

```
(define (type-mask n m)
  (+ (if (fixnum? m) 1 0)
     (if (fixnum? n) 2 0)))

(type-mask 10 #f)
(type-mask (read) (read))
```

Figure 5. Scheme code of a function computing the type binary mask of two variables.

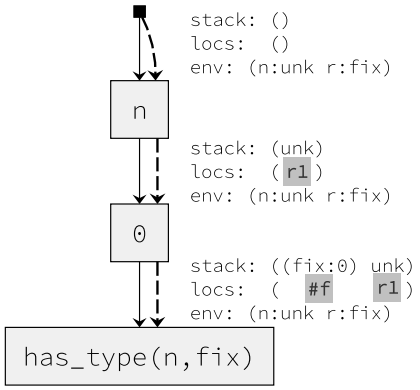

Figure 6. Contexts used for generators of the first chain of the fact function augmented with register allocation information.

generating stubs. This addition allows the compiler to do constant folding using propagated constant information.

Because BBV uses code duplication to generate specialized versions of the blocks, constant propagation and constant folding are available even if different constant combinations are observed when a generator is called.

Figure 5 shows an example in which constant propagation based on BBV allows generation of efficient code. The `type-mask` function computes a binary mask. A bit is set in the mask if the variable it is associated with is a fixnum. The `fixnum?` operator detects if the variable is a fixnum. No matter if the variable is a fixnum or not, the constant is propagated for the rest of the function. When the addition is reached, both operands are represented by constants in the compilation context meaning that the compiler can compute the final mask without generating code. In addition, if interprocedural extensions are used, it is possible that the type of the operands are known thus no code is generated for the `fixnum?` operators. It is the case for the first call to the `type-mask` function in the figure. This means that for this specialized version, no code at all is generated to compute the mask.

Using interprocedural extensions opens up the opportunity to interprocedurally propagate constants.

3.1 Loops

An important problem with this optimization is loops. If a loop counter is initialized using a constant and modified each iteration by a constant, its value is propagated through the context and generated code is specialized using its value resulting in a loop unrolled `n` times, `n` being the number of iterations. Although this is beneficial for loops with few iterations, it results in code explosion for loops with many iterations. To avoid this problem, simple heuristics can be used such as stopping constant propagation at loop nodes, or limiting variations of constants in versions. This is not a problem in LC because Scheme has no primitive loop constructs.

4 Register Allocation

Register allocation is a problem isomorphic to graph coloring. Graph coloring algorithms are then used to solve register allocation in Ahead-Of-Time (AOT) compilers allowing to generate efficient code. However, the allocation time of graph coloring is high which is problematic in a JIT compiler because compilation time impacts the total execution time.

Other techniques more adapted to JIT compilers such as Linear Scan [14] have been discovered. Linear Scan allows to reduce static work by generating code slightly slower than when graph coloring is used. However, Linear Scan requires liveness information gathered from dataflow analysis which is not compatible with our goals.

To avoid IR and static analysis, a greedy register allocation can be used. The compiler can greedily associate a machine register to each slot of the context virtual stack each time a type is added. It is possible that multiple slots of the stack are associated with the same register. For example, if a local variable is added multiple times to the stack, a single register can be used thus adding the variable to the stack requires no operation.

At its initial state, the virtual stack is empty meaning that all registers are available. When a type which is not representing a constant value is added to the virtual stack, the compiler associates an available register to this slot. If a register is available, this register is associated with the slot and it is removed from the set of available registers. If no register is available, a value associated with a slot in the virtual stack of the context must be spilled to free a register. Any spilling heuristic can be used in this situation. In LC, we decided to spill the value associated with the lowest slot in the context stack which is associated with a register.

As an example, figure 6 shows how the generators of the first chain of the `fact` function are called using contexts

augmented with register allocation information. When the generator of the first node is called, the type of `n` is added to the stack and the next available register (`r1` in this case) is associated with this new slot. This information is propagated and next generators use this information to generate code using registers. Because the node associated with `0` represents a constant, it is not necessary to associate a register to the new slot when its generator is called.

An important problem with greedy register allocation is that registers are allocated for temporary values. Because liveness analysis is not necessarily used, the compiler does not know when allocating a register, that a value is not live thus its associated register is available. Using a stack in the compilation context, as done in LC, solves this problem. Because a temporary value is represented in the compilation context by a slot in the virtual stack, the register it is associated with can be added to the available register set when the slot is popped from the stack and if no other slot is associated with the same register.

Using a virtual stack in the context then allows the compiler to free registers associated with temporary values as soon as they are used.

4.1 Code Specialization Based on Register Allocation

When a version must be generated, the compiler first checks if a version has already been generated for this context to reuse it. However, if register allocation is added to the compilation context, it is possible that a context used for an existing version differs in register allocation information only.

In this case, the compiler can use this version and generate extra moves between registers to conform to the register allocation used for the existing version. The other solution, used by LC, is to use register allocation information to specialize code. This means that, in this situation, a new version is generated for this exact context thus no extra move is generated nor executed at run time. Next blocks are then specialized using this register allocation information.

5 Boxing and Unboxing

To implement dynamic typing, type checks are inserted in the generated code to ensure safety of the language primitives. It has been shown that BBV is efficient at discovering types thus at removing type checks [5, 6, 19].

Another reason dynamic typing negatively impact performance is that because types are not necessarily known at compilation time and type checks must be able to retrieve the type of a value at run time, the run time representation of a value must include both the value and its type. This representation is commonly referred to as a *box*.

To allow primitives to handle values, additional boxing and unboxing operations are inserted in the generated code. Unboxing operations are generated before a primitive to extract values from the boxes representing its arguments. Boxing operations are generated after a primitive to create a box containing the result of the primitive along with the type of the result.

```
; compute n-1 and box the result
r1 = unbox(n)
r2 = r1 - 1
r2 = box(r2)
; compute n*r and box the result
r3 = unbox(n)
r4 = unbox(r)
r5 = r3 * r4
r5 = box(r5)
; recursive call
r6 = call fact(r2, r5)
```

Figure 7. Pseudo assembly code including boxing and unboxing operations generated by a naive compiler for the recursive call of the `fact` function.

In a compiler using naive boxing and unboxing, redundant boxing and unboxing operations are inserted in the generated code.

Figure 7 shows an example of pseudo assembly code generated for the recursive call of the `fact` function including boxing and unboxing operations by a naive compiler. Two unboxing operations are inserted, and executed for `n` for the operators `-` and `*` whereas a single unboxing operation could have been inserted and the unboxed value used for both operations.

Common Subexpression Elimination (CSE) is used to remove these redundant operations. Global CSE is usually computed on the IR to remove extra boxing and unboxing operations in a procedure, and local CSE is performed when generating code to remove boxing and unboxing operations inside a basic block. Because we decided to avoid the use of IR, and because the compiler has no concept of basic block, classical CSE cannot be applied.

5.1 Run Time Value Representation

Depending on the box representation the compiler uses, boxing and unboxing operations impact performance differently. Type tagging [10] is a technique widely used to represent boxes in dynamically typed languages. When type tagging is used, one or more of the least significant bits of a machine word representing a value are reserved to represent the type (type tag). The remaining bits are used to represent the value. In the case of LC, two bits are used for the type and the remaining 62 bits are used for the value. Fixnums are usually represented with a type tag containing only zeros allowing boxing and unboxing operations on fixnums at no cost. For example, the addition (e.g. the x86 `add` instruction) of two

boxed fixnums (i.e. two fixnums that are not unboxed before the addition), results in a fixnum representing the boxed result. In other words, adding two boxed fixnums is equivalent to unboxing the fixnums, adding both unboxed values, and boxing the result. However, some primitives still require unboxing. This is, for example, the case for multiplication. For multiplication, one of the operands must be unboxed before the multiplication is performed. The result already is boxed.

This shows that wisely choosing the representation of boxes allows to significantly decrease the performance impact of boxing and unboxing operations with no development effort.

On specific architectures, boxing and unboxing operations on heap-allocated objects can also be done at no cost. When type tagging is used, memory objects are aligned when allocated. This alignment ensures that the n least significant bits are set to 0. The type tag can then be inserted in these n bits. To unbox a heap-allocated object, the n bits are cleared to retrieve the object address.

In the case of x86, the addressing mode allows the use of a displacement constant when reading or writing a memory slot. This displacement allows the removal of the tag from the box at no cost thus to read or write a field of a boxed object using a single instruction. Boxing operations can also be executed at no cost. On x86, after an object is allocated, its address can be computed, and the tag can be inserted using a single `lea` instruction.

5.2 Flonums

Flonums are a specific case of heap-allocated objects. The double precision IEEE-754 standard [8], widely used to represent flonums, requires 64 bits to represent a number. This means that on 64 bits machines, if all the bits of a word are used to represent the value, the type tag cannot be inserted in the least significant bits thus flonums must be allocated in the heap.

Heap allocating flonums means that a memory allocation is executed for each boxing operation and a memory read is executed for each unboxing operation significantly impacting performance compared to other types. In addition, because more objects are allocated, garbage collector (GC) time also is impacted.

Other box representations such as NaN-boxing [10] solve this problem for flonums. NaN-boxing uses the unused range of NaN representations of the IEEE-754 standard to represent boxes of other types. The advantage of this representation is that boxed flonums can directly be represented by their IEEE-754 representation avoiding boxing and unboxing operations. However, NaN-boxing limits the fixnums to 32 bits and does not allow zero cost unboxing of heap-allocated objects on x86.

To decrease development effort, a compiler could then use type tagging to represent boxes, allowing a significant decrease in the cost of boxing and unboxing of fixnums and non-flonum heap-allocated objects. Development effort can then be focused on decreasing the impact of flonum boxing and unboxing.

5.3 Eager Unboxing of Flonums

Compilers can use type discovery of BBV to eagerly box and unbox values. Using BBV, each time a type check succeeds for the first time, a type is discovered and the compiler is called by the compilation stub created for this check to generate the next branch. Before generating the branch, the compiler can generate a code sequence to unbox the value for which the type has just been discovered meaning that this variable is handled unboxed for the rest of the execution. The compiler can use this approach on flonums to decrease boxing and unboxing impact.

Using eager unboxing, the compiler knows that if the type of a variable is known and the variable is a flonum, the value is unboxed (i.e. there is no need to unbox it before primitives). If the compiler does not know the the type of a variable it knows the value is boxed.

If the type of a value that is an argument of a flonum primitive is unknown, type checks LCO are necessarily inserted before the primitive meaning that using our implementation of BBV and eager unboxing, the compiler necessarily knows that this argument is an unboxed flonum when the generator of the LCO associated with the primitive is called.

Because BBV uses code duplication to generate specialized versions, polymorphic variables are successfully handled. In addition, by using interprocedural extensions, the compiler can propagate unboxed flonums through function calls and returns even in the presence of higher order functions.

In addition, if the target machine provides specialized registers (e.g. `xmm` registers on x86), each newly unboxed flonum can be moved to one of these registers. Register allocation of specialized registers can be propagated and used to specialize code using the same technique we presented in section 4.

5.3.1 Impact on the Garbage Collector

Even if unboxed flonums are moved to specialized registers, it is possible the compiler decides to spill an unboxed flonum at some point. This means that both boxed and unboxed values can be stored in the execution stack, meaning that the GC cannot do precise stack scanning to detect roots representing heap-allocated objects. Instead, the GC can use a conservative approach but such a system can retain objects that must actually be collected. Another solution, used by LC, is to indicate to the GC which slots of the execution stack actually contain heap-allocated objects using GC stack maps [11]. In the case of LC, a GC stack map uses a single bit per slot of the frame it is associated with. A bit is set if the compiler knows that the slot it is associated with contains a flonum (i.e. a context stack slot associated with a spilled

value contains the type `flonum`) allowing the GC to know that this slot does not contain a memory allocated object.

6 Results

LC is a research oriented JIT compiler for Scheme, written in Scheme. It implements a subset of the R5RS Scheme standard [12] (first-class continuations are not supported and *eval* implementation is limited). Its implementation is described in [17–19]. The compiler uses the techniques presented in this paper to generate efficient code and to decrease development effort. The compiler is based on Gambit [7], it uses its frontend, GC and x86 assembler.

LC interprocedurally specializes code according to type information. Supported types are *boolean*, *character*, *f64vector*, *fixnum*, *flonum*, *function*, *pair*, *string*, *symbol* and *vector*. Generated code is also intraprocedurally specialized using constants and register allocation information.

The 38 benchmarks used for these experiments are those typically used to benchmark Scheme implementations.

LC is written by a single person in less that 15 KLOC (physical SLOC) of Scheme and has been successfully used as a research tool for the past 4 years. This confirms that the techniques presented in this paper allow a reduction in development effort to build a JIT compiler with limited resources.

This section presents the results of the experiments showing what performance can be reached by a compiler based on these techniques.

The baseline used for the results is LC that is not using the optimizations we presented. In this configuration, a single generic version is allowed for each AST node, meaning that constants and types are not propagated, code is not specialized using type information nor register allocation information (i.e. merge code is generated) and flonums are not unboxed thus specialized registers are not used. The baseline then shows how code would be generated by a simple and naive JIT compiler for x86. For the rest of the section, this configuration is referred to as *LC in naive mode*. The configuration in which LC uses all the techniques is referred to as *LC in optimized mode*.

6.1 Type Checks

Chevalier-Boisvert and Feeley [5] showed that BBV is efficient at removing type checks for JavaScript. Other work has demonstrated that interprocedural extensions allow BBV to remove even more checks for JavaScript [6] and Scheme [19].

Figure 8 shows the number of type checks executed by LC in optimized mode relative to LC in naive mode. For 12 of the benchmarks, interprocedural BBV, as implemented in LC, allows the removal of essentially all the checks. For the other benchmarks, the number of executed type checks is significantly decreased. In the worst case with `takl`, around 7% fewer checks are executed. On average, 70% fewer checks are executed.

6.2 Eager Unboxing

We showed in section 5 that, used along with BBV, eager unboxing allows the removal of boxing and unboxing operations without using static analysis nor IR.

Table 1 shows the number of executed boxing and unboxing operations for LC in naive mode and LC in optimized mode for flonum intensive benchmarks. We can see that essentially all of the operations are removed for these benchmarks. In the worst case (`simplex`), around 97% fewer boxing operations are executed and around 95% fewer unboxing operations are executed. On average, more than 99% fewer operations are executed.

As a comparison, we measured the number of executed boxing and unboxing operations on flonums by the Gambit Scheme compiler for these benchmarks. Because LC uses the frontend of Gambit, the input AST of the two compilers is the same thus LC in naive mode represents the worst case (i.e. each argument is unboxed when generating a primitive and the result is boxed). Gambit, using local CSE in basic blocks, executes only 2% fewer boxing operations and 3% fewer unboxing operations than LC in naive mode on average.

6.3 Execution Time

The benchmarks are executed on a machine using an Intel Core i7-4870HQ CPU with 16GB DDR3 RAM running the GNU/Linux operating system. Each benchmark is executed 10 times, the minimum and maximum execution times are removed and the average of the 8 remaining values is taken. The number of iterations for each benchmark is taken from Gambit.

Figure 9 shows the execution time of the machine code generated by LC in optimized mode relative to LC in naive mode. The execution time presented in this figure allows the measurement of the impact of the techniques on the performance of the generated machine code. It then includes execution time of generated machine code and excludes compilation time and GC time. As expected, due to eager unboxing of flonums, the most important speedup for LC in optimized mode occurs for flonum intensive benchmarks (up to 8.7x faster with `fftrad4`). Because BBV allows to interprocedurally remove type checks, fixnum intensive micro-benchmarks such as `fib` and `sum` are also significantly faster. The biggest benchmark is `compiler` (~12 KLOC) which intensively uses data structures. For this benchmark, LC in optimized mode allows generating code that is 1.4x faster than the code generated by LC in naive mode. All the benchmarks are significantly faster with LC in optimized mode compared to LC in naive mode. The worst case occurs for `divrec` which is only 1.05x faster. This benchmark intensively uses lists. Because LC does not track the type of the values in compound data types such as lists or vectors, type information is lost when

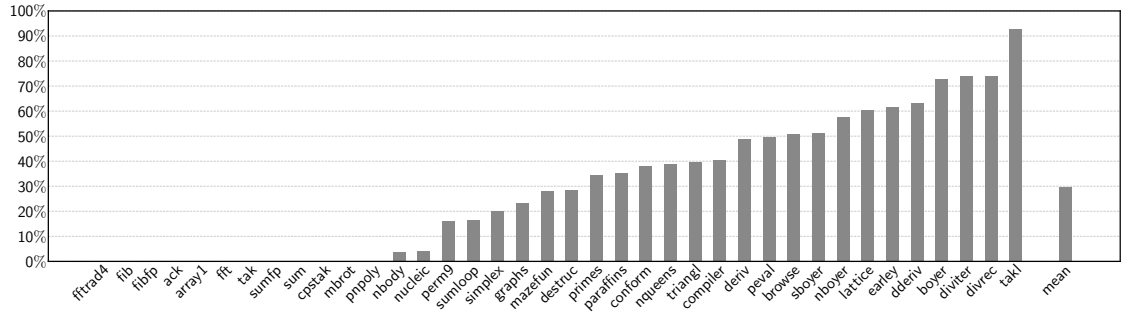

Figure 8. Number of executed type checks relative to LC in naive mode.

Table 1. Number of flonum boxing and unboxing operations with LC in optimized mode relative to LC in naive mode.

	LC - Naive unboxing		LC - Eager unboxing			
Benchmark	# boxing	# unboxing	# boxing	# unboxing	% boxing	% unboxing
fft	93168009	128990020	10	19	≈0.00	≈0.00
fftrad4	262133751	435134470	10	19	≈0.00	≈0.00
fibfp	89582115	179164234	11	21	≈0.00	≈0.00
mbrot	137762909	273654718	9	18	≈0.00	≈0.00
nbody	470000627	665000701	11	20	≈0.00	≈0.00
nucleic	65971745	74996176	189010	189019	0.29	0.25
pnpoly	112100009	194200018	9	18	≈0.00	≈0.00
simplex	48300009	52800018	1400009	2400018	2.90	4.55
sumfp	400040009	800100020	11	20019	≈0.00	≈0.00
Mean					**0.35**	**0.53**

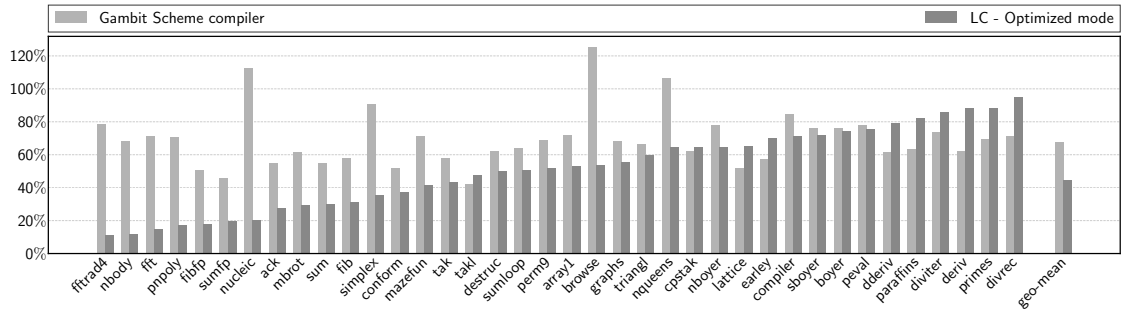

Figure 9. Execution time of the code generated by Gambit and by LC in optimized mode relative to the code generated by LC in naive mode.

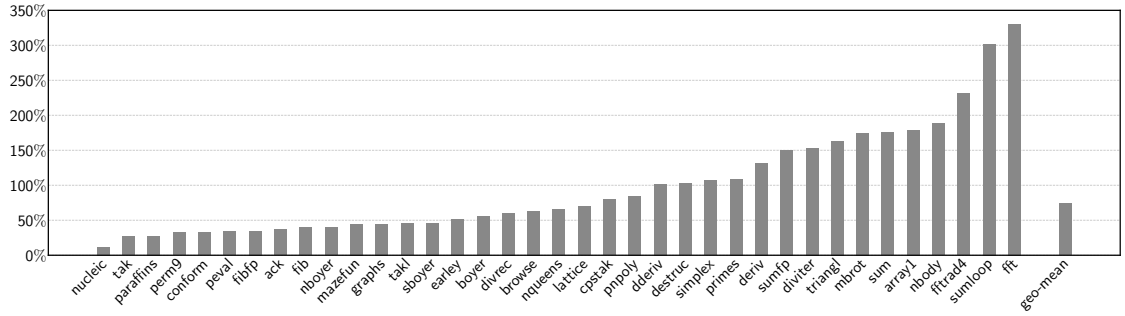

Figure 10. Execution time with LC in optimized mode relative to Pycket.

an element is inserted in a list or a vector meaning that the compiler is not able to remove type checks. On average, the code generated by LC in optimized mode is 2.25x faster than the code generated by LC in naive mode.

This figure also shows the execution time of the code generated by Gambit relative to LC in naive mode. Gambit is known to be an efficient AOT implementation of the Scheme programming language. For some benchmarks, Gambit generates faster code than LC in optimized mode. Because LC does not track the type of the values in compound data types, it does not remove type checks in some cases and because indirections at function returns are necessary to implement the interprocedural extensions, generated code is slower.

We can see in the figure that for 26 benchmarks out of 38, the code generated by LC in optimized mode is faster than the code generated by Gambit. Because eager unboxing allows the removal of the overhead of heap-allocated flonums, flonum intensive benchmarks are significantly faster with LC. On average, the code generated by LC in optimized mode is 1.47x faster than the code generated by Gambit.

Figure 10 shows the execution time of the benchmarks for LC relative to Pycket[2] [1]. Pycket is a Scheme JIT compiler based on meta-tracing built with the PyPy framework [16]. The execution time presented in this figure includes compilation time, GC time and execution time. The `compiler` benchmark is not included because Pycket fails to execute it. Because Pycket does not support homogeneous numeric vectors, heterogeneous vectors are instead used for both compilers. LC is significantly faster for 22 benchmarks out of 37. Thanks to BBV, fixnum intensive benchmarks are significantly faster. Because eager unboxing is used on flonums, several flonum intensive benchmarks are also significantly faster (up to 8.42x faster with `nucleic`). However, several flonum intensive benchmarks are significantly slower with LC (e.g. `fft` and `fftrad4`). In addition to flonums, these benchmarks intensively use vectors. We think these benchmarks are faster with Pycket because it optimizes vectors for flonums. When we execute these benchmarks with LC using homogeneous vectors (`f64vector`), they are as fast as with Pycket. On average, the benchmarks executed with LC in optimized mode are 1.35x faster than when executed with Pycket.

6.4 Summary

Our results show that these simple techniques allow to significantly decrease the performance impact necessary to implement dynamic programming languages. In the case of LC, the techniques allow to generate code that is, in many cases, faster than existing optimizing AOT and JIT compilers for Scheme.

[2]https://github.com/pycket/pycket, September 2018 build

7 Related Work

Several studies showed that JIT compilers are difficult to build compared to interpreters. The DynamoRIO [20] project showed that the negative performance of the interpretation of dynamic languages can be decreased using compilation *traces*. However, the technique requires the use of an interpreter thus makes the architecture more complex. Other research shows that development effort necessary to implement JIT compilers can be decreased by writing interpreters and optimizing the interpreter itself to increase performance. The PyPy project [2, 16] used by Pycket [1] uses trace based JIT compilation, typically used by JIT compilers on source programs, to the code of interpreters allowing the efficient building of virtual machines with low development effort. The Truffle [21, 22] project goal is similar to PyPy. It allows building interpreters for languages and optimizes the interpreters using type feedback and other profiling information. These projects allow to significantly decrease the development effort necessary to build language implementations. However, they give no control over the compilation phases and low level techniques limiting their use if full control is needed.

An alternative to build efficient language implementations is to use an existing bytecode based virtual machine such as the Java Virtual Machine [13]. These optimizing virtual machines execute programs previously compiled to the bytecode they support. A compiler from the source language to bytecode can then be built and rely on these virtual machines optimizations and execution environments to efficiently execute programs. Using bytecode based virtual machines allows more control on the compilation phases but gives no control on low level implementation.

8 Conclusion

We have presented techniques allowing to decrease the development effort necessary to build JIT compilers for dynamic languages. These techniques have been used to build LC, a JIT compiler for Scheme, written by a single person in less than 15 KLOC and successfully used as a research tool showing that they are suitable to be used in an environment with limited resources. We have evaluated performance of the code generated by LC to show what performance can be reached by a compiler based on these techniques. We have shown that they allow to reach significantly better performance than naive JIT compilers (up to 8.3x faster) allowing LC to reach better performance than mature AOT Scheme implementations and other optimizing JIT compilers.

This study shows that it is possible to build efficient JIT compilers for dynamic languages with relatively low development effort and, unlike frameworks designed to build language implementations, with full control over the system.

References

[1] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015. Pycket: A Tracing JIT for a Functional Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*.

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009*.

[3] Carl Friedrich Bolz and Armin Rigo. 2007. How to Not Write Virtual Machines for Dynamic Languages. In *3rd Workshop on Dynamic Languages and Applications*.

[4] Maxime Chevalier-Boisvert. 2015. *On the Fly Type Specialization Without Type Analysis*. Ph.D. Dissertation. Université de Montréal.

[5] Maxime Chevalier-Boisvert and Marc Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*.

[6] Maxime Chevalier-Boisvert and Marc Feeley. 2016. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*.

[7] Marc Feeley. 2018. Gambit Scheme Compiler v4.8.9. http://gambitscheme.org/

[8] IEEE Standard for Floating-Point Arithmetic. 2008. *IEEE Std 754-2008* (2008).

[9] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-In-Time Type Specialization For Dynamic Languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*.

[10] David Gudeman. 1993. Representing Type Information in Dynamically Typed Languages.

[11] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*.

[12] Richard Kelsey, William D. Clinger, and Jonathan Rees. 1998. Revised^5 Report on the Algorithmic Language Scheme. *SIGPLAN Notices* (1998).

[13] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. 2018. *The Java Virtual Machine Specification*.

[14] Massimiliano Poletto and Vivek Sarkar. 1999. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems, TOPLAS* (1999).

[15] Armin Rigo. 2004. Representation-Based Just-In-Time Specialization and the Psyco Prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2004*.

[16] Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*.

[17] Baptiste Saleil and Marc Feeley. 2014. Code Versioning and Extremely Lazy Compilation of Scheme. In *Scheme and Functional Programming Workshop, SFPW 2014*.

[18] Baptiste Saleil and Marc Feeley. 2015. Type Check Removal Using Lazy Interprocedural Code Versioning. In *Scheme and Functional Programming Workshop, SFPW 2015*.

[19] Baptiste Saleil and Marc Feeley. 2017. Interprocedural Specialization of Higher-Order Dynamic Languages Without Static Analysis. In *31st European Conference on Object-Oriented Programming, ECOOP 2017*.

[20] Gregory T Sullivan, Derek L Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. 2003. Dynamic Native Optimization of Interpreters. In *Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators*.

[21] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH 2012*.

[22] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages, DLS 2012*.