Code Versioning and Extremely Lazy Compilation of Scheme

Baptiste Saleil
Université de Montréal
baptiste.saleil@umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Abstract

Dynamically typed languages ensure safety through the use of type checks performed at run time. Static type inference has been used to remove type checks but this approach is limited in the absence of code duplication. This paper describes an alternate approach that generates multiple versions of the code to specialize it to the types that are observed at execution time by using an extremely lazy compiler. The laziness is important to limit the number of versions (limit code bloat) and to generate more specialized code (increase performance). We describe LC, a Scheme compiler which implements this code generation approach. The compiler is currently a prototype that cannot run large benchmarks, but an examination of the generated code shows that many dynamic type tests can be removed.

1. Introduction

Dynamic languages are widely used both for writing small scripts and more complex programs because they are expressive and easy to use, but often they suffer from a lack of performance. A main cause of this performance issue is that the code does administrative work at execution such as type checking, boxing and unboxing, etc.

As an example, consider this simple Scheme [1] expression:

```
(car (f 42))
```

In principle, a type test is performed by the `car` operation at run time to ensure that the result of `(f 42)` is a pair. If it isn't, an error will occur at run time when invoking `car`.

An approach is to do a type inference to determine types, if possible, before code generation [5]. Even if the information known about the types is partial (but conservative) the compiler can generate better machine code that is suitable for all executions. Performing an expensive and more precise type analysis is unfortunately not advisable in the context of *Just In Time* (JIT) compilers because compilation, which is done at run time, will negatively impact the execution time.

This paper describes *code versioning*, a code generation approach for JIT compilers that aims to use this compile time information to generate multiple versions of the same machine code, each one suitable for a particular execution context. This approach is illustrated by the following example:

```
(define (foo a)
  (car a))

(foo '(1 2 3))
(foo (read))
```

In this code, there are two executions of the primitive `car`. In the first one with `a='(1 2 3)` the compiler knows that the primitive is executed with a pair as argument. In the second call to `foo` with `a=(read)` it knows that `car` is invoked with a value of the same type as returned by `(read)`. Given that the type is unknown, the compiler will generate two versions of the primitive in this order:

Pair version On the first call to `foo`, the compiler knows that `a` is a pair, and knows that `car` expects a pair. It will then generate a version which directly accesses the `car` of the pair without performing any type test.

Generic version On the second call, the compiler has no information about the type of `a`. It will generate a version that contains a `pair?` type test on `a`.

While code versioning is extensible to other purposes, this paper focuses only on removing type checks. We also show how extremely lazy compilation can improve code versioning both by improving its action to remove more type tests and limiting the number of generated versions with the goal to balance the extra cost of generated code size.

Other existing techniques already generate multiple versions of code and some of them work at the procedure or loop level. Our technique aims to generate multiple versions of all pieces of code as soon as the execution reaches a point that the compiler can specialize with a recently discovered type information. A consequence of this is that each function can have multiple entry points which causes some implementation issues. We also present *LC*, our Scheme JIT compiler that aims to implement all mechanisms required to use code versioning and extreme laziness. LC avoids the use of an interpreter and compiles executed parts of source programs directly to machine code.

This paper is organized as follows. Section 2 explains some choices made in accordance with our goal. Section 3 presents the concept of code versioning and shows how it can be implemented. Section 4 explains how the code ver-

sioning approach can be improved by using extreme laziness and how it can reduce the size of the generated code. This section also explains implementation issues. Then, section 5 briefly explains current experimental results. In Section 6 we discuss the current limitations of our system and how they can be avoided. Finally, sections 7 and 8 present future and related work. Because LC is at an early stage of development we do not have extensive benchmark results yet.

2. Compiler specifications

LC is a JIT compiler developed to experiment with code versioning and lazy compilation for the Scheme language. We made some choices consistent with this main goal:

Representations The compiler uses no intermediate representation. It directly reads S-expressions and translates executed code into machine code. Therefore, the compiler does not lose time generating other code representations. Moreover, because we chose an extremely lazy compilation, the compiler will never run analysis such as *Data/Control Flow Analysis* so the use of a representation such as *Static Single Assignment* (SSA) [3] or *Three-Address Code* (TAC) [2] is discarded.

Compiler only Code versioning aims to generate faster code by generating specialized versions of machine code. Because our goal is to evaluate the benefits of this technique on generated machine code, we chose the strategy to only compile executed code and never execute it with an interpreter. So LC uses only a pure JIT compiler and never interprets code.

Target platform Because we focus our research on the benefits of the context on the generated machine code, the target platform is not critical. We arbitrarily chose to generate code for x86_64 family processors using only a stack machine (without any register allocation algorithm).

3. Code versioning

3.1 General principle

In statically typed languages, types are known at compile time and the compiler can generate a unique optimized version of the code suitable for all executions. In dynamically typed languages, type declarations are not explicit so the compiler embeds type tests on primitive operations to detect type errors at run time. Run time type tests are one of the main dynamically typed languages performance issues. A solution to limit this issue is to use type inference to determine types in expressions and generate code according to this information. Type inference often involves static analysis which is not suitable for JIT compilers. Other systems allow mixing dynamically typed languages with explicit type declarations to improve performance (e.g. Extempore [13]) but such solutions lose the main characteristic of dynamic languages. The JIT compilers, widely used in dynamic languages, allow to postpone the machine code generation of executed code. A benefit is that the compiler can use information newly discovered by execution to better optimize the future generated code. Code versioning uses this information to generate multiple versions of the same piece of code, each one associated to a particular entry context and optimized for this context. This involves that each piece of code is now accessible by as many points as versions generated. However, the coexistence of these versions will result in an increase of the generated code size, but we will show that this extra cost is attenuated by the use of an extremely lazy compilation.

```
PUSH a
PUSH b
ADD
JO  overflow
JMP next-generator
```

Figure 1. Generated version, in pseudo assembly, for context `((a . number) (b . number))`

```
PUSH a
PUSH b
CMP (type a), (number)
JNE type-error
ADD
JO  overflow
JMP next-generator
```

Figure 2. Generated version, in pseudo assembly, for context `((a . unknown) (b . number))`

3.2 Implementation

Each piece of code is represented by a *generator*. The generator is the object that manages the versions of this piece of code and acts as a code stub. A simple example of code versioning is for the Scheme expression `(+ a b)`. The compiler creates a generator for this expression. Let g be the generator of this expression. At the beginning of execution, no version is yet generated. As soon as an instruction `i` transfers control to this generator, and assuming that `ctx` is the current context at this point, the generator follows this algorithm:

1. if g does not contain a version associated to `ctx`,
   Generate a new version based on `ctx`*
2. Replace the destination of `i` by the address of the version
3. Transfer control to the newly generated version

*note that if `i` is the last instruction generated, we can just overwrite `i` with this new version.

Figures 1 and 2 show an example of two generated versions of the same expression `(+ a b)` based on two different contexts. Note that we use association lists to represent

contexts mapping identifiers to types. Figure 1 is based on a context in which we know that `a` and `b` are both numbers. So it is useless to generate dynamic type tests and we can directly use the two values. In figure 2, we know that `b` is a number but this time we do not have information about the type of `a`. We can then directly use the value of `b` and generate a dynamic type test for `a` only.

3.3 Improved closures

The traditional flat closure representation [7] is compatible with code versioning. The main problem of this representation is that the procedure has only one entry point. The compiler must generate code suitable for all executions and this looses information about the argument types. In order to take better advantage of code versioning, we decided to keep multiple entry points corresponding to versions by modifying this flat closure representation. The first field is now a reference to a table which contains the entry points of the procedure. Because of higher order functions, we must assign an index to a given context that is the same for all closures. We will call this table the *closure context table* (cc-table) for the following of this paper. For example, if a function is called with two arguments `a` and `b` and the context `ctx1 ((a . number) (b . number))`, the compiler will generate a code sequence similar to figure 4. Here the compiler associates `ctx1` to the index 3 in the cc-table. So the index for this specific context will now be the same for all closures. When the compiler creates the cc-table, it fills all the fields with the address of the function stub because there is currently no generated version. When the stub is called, the generator creates a new version associated to the context and patches the cc-table of the closure at the correct offset to jump directly to the generated version. Because our compiler does not have a garbage collector yet, it currently creates a fixed size table for each closure and stops execution on table overflows. A consequence of this approach is that each cc-table must be big enough to contain an entry for each call-site context of the program. This limitation is discussed in a dedicated section.

Figure 3 shows the changes made to classical flat closure representation. The first closure on top is an example of a closure just after its creation. This closure contains `n` non-local variables and a cc-table of size 4. The second one, at bottom, is the same closure after some executions with two generated versions, one at address V_1 for the context associated to index 1, and the other at address V_3 for the context associated to index 3.

4. Extremely lazy compilation

While many Scheme compilers use *Ahead Of Time* (AOT) compilation (e.g. Gambit [12], CHICKEN [11]), there are several strategies used by JIT compilers for dynamic languages. Some of them only compile hot spots to machine code to improve performance and use an interpreter for other

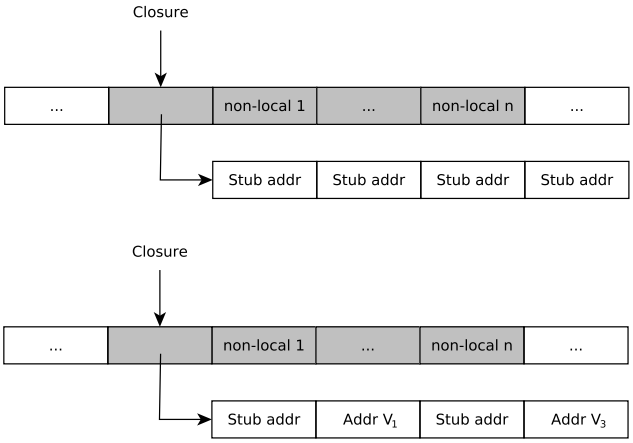

Figure 3. Closure example at creation, and with two generated versions

```
; pop closure
POP  R1
; return address is the
; continuation stub address
PUSH continuation
PUSH a
PUSH b
; get cc-table
MOV  R1 <- [R1]
; jump at offset 3
JMP  [R1+3*8]
```

Figure 4. Assembly code to jump to a procedure entry point associated to a context

executed code [6]. This can be done at multiple levels such as loops and functions. Another widely used strategy is to compile executed pieces of code just before their execution (e.g. Google V8 [10]). With this strategy, the more lazy the compiler is, the more it compiles only the executed code. For example, a compiler may compile only executed functions, but some branches of the function may not be executed, or only compile executed branches to be lazier. Using this strategy the compiler doe not lose time to compile code that is never executed.

4.1 An asset for code versioning

There are two advantages to using an extremely lazy compiler: Compile only executed code to save time and improve the efficiency of code versioning. If the compiler is lazier, it has more useful information in the current context:

```
(let ((v (foo)))
   (+ v 42))
```

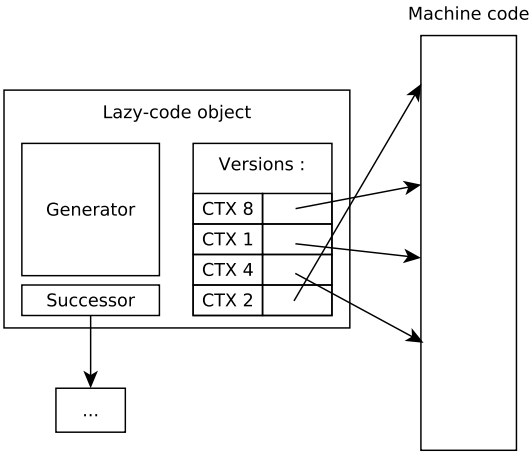

Figure 5. Example of lazy-code object

In this example, if the compiler is not lazy enough it does not have information about the type of `v` and generates a dynamic type test of `v` for the expression `(+ v 42)`. With an extreme laziness, the compiler generates the machine code after the `foo` function returns and it's possible that the type of `v` is known. If `v` is a number, the generator compiles a version without dynamic type test on `v`. At this point, the compiler does not care about the future executions. If `foo` always returns a number, the generator never compiles other versions and no type test is performed. On the other hand if `foo` returns other types, then other versions will be generated. In these two cases the extreme laziness removes the type test (at least in some cases) and improves performance.

One of the main weaknesses of code versioning is the coexistence of multiple versions of the same code which results in an increase of generated code size. Taking the expression `(+ a b)` as an example, we have to generate exactly 5 versions:

- No dynamic type test if `a` and `b` are numbers
- One dynamic type test on `a` if `b` is a number and `a` is unknown
- One dynamic type test on `b` if `a` is a number and `b` is unknown
- Two dynamic type tests if both are unknown
- An error if the type of `a` or `b` is known but not number

The extreme laziness allows the compiler to generate only executed versions of the code. Thus the number of versions is reduced which in turns reduces the size of the machine code.

4.2 Implementation

To keep the advantage of code versioning and be extremely lazy, LC uses *lazy-code objects*. Each piece of code of the source program is represented by one of these objects. Figure 5 shows an example of a lazy-code object. This object contains two main elements. The first one on the left is the generator presented in Section 3.2 which is able to generate a new version of the code that it represents. The second on the right is a table which contains all the addresses of generated versions in memory, each one associated to the entry context. All lazy-code objects are used similarly to Continuation Passing Style, when the compiler creates one object, it also gives the successor object (represented at the bottom of figure 5). Again with the example of expression `(+ a b)` if the compiler knows that both are numbers, and if it does not care about overflows, it creates exactly 4 lazy-code objects: (1) End of program object. This will clean stack, restore registers and return. This object is the last in execution flow so it does not have successor object. (2) Object for `+`. The code generated by this object performs the add operation: it will pop two values from the stack, add them, and push the result. When the compiler creates this object it gives object 1 as successor. (3) Object for `b`. This object generates the code to compute the value associated to identifier `b`. i.e. it pushes the value on the stack. For this object, the successor is object 2. (4) Object for `a`. This time the successor object is object 3. Figure 6 shows an example of the code which creates these 4 objects for this expression.

```
(define (gen-ast ast successor)
   ...
   (if (eq? (car ast) '+)
      (let* ((lazy-add
                (make-lazy-code
                   (lambda (ctx)
                      (pop r1)
                      (pop r2)
                      (add r1 r2)
                      (push r1)
                      (jump-to successor
                               ctx))))
             (lazy-arg1
               (gen-ast (caddr ast)
                        lazy-add)))
         (gen-ast (cadr ast) lazy-arg1)))
   ...)

(let ((obj
         (gen-ast '(+ a b)
                  (make-lazy-code
                     (lambda (ctx)
                        (pop r1)
                        (return))))))
   (execute obj))
```

Figure 6. Creation of lazy-code objects chain for expression `(+ a b)`

So the compiler creates a chain of lazy-code objects. At this point no machine code is yet generated for the expression. To execute the expression, the compiler transfers control to the generator of object 4. This generator follows the algorithm presented in Section 3.2 to generate an inlined version (or patch the jump). Because the context cannot be changed the compiler can trigger multiple generators to compile code as long as a branching expression is not yet encountered. This removes the useless jumps between versions in execution flow. In the previous example, as soon as the generator of `a` is called, the compiler will generate all machine code because there is no branching instruction until the end of this small program.

4.3 Procedure call problem

Because the compiler is lazy, it does not know the position of the entry point of the continuation when it generates the code to call a procedure. As shown in figure 7, our solution is to create a temporary code stub for this continuation. When the compiler generates the code for the call site, it pushes the address of this temporary stub as return address and jump, using closure, to the generator (or existing version associated to this context) of the procedure. When the compiler generates the code of the procedure return, it writes a classic `return` instruction which actually jumps to the continuation generator. As soon as the continuation is generated, the stub patches the call site to replace the current return address (continuation stub address) by the actual address (position of the machine code of continuation). Note that the context is a mandatory argument since the procedure stub cannot generate a version without this information. The `push closure` instruction is doubly useful here, first it is used to access non-local variables from generated code, but it is also used by the procedure stub to patch the cc-table as explained before. The right side of the figure shows the state after execution of the call site, procedure, and continuation.

4.4 Context construction

In order to take maximum advantage of code versioning and remove even more type checks, it is important to have as much information as possible within the context when compiling a piece of the code. There are two ways to build this context.

When the compiler is compiling constants, the type of the constant is known at compile time and we can extend the context to generate the next objects. Assuming the compiler uses a context that associates a type to each value on the stack, if it generates the code for a lazy code object containing a constant (`10` for example), then it will generate an instruction `push 10` and, as there are no branches, will start generating the next object by adding the type information `number` to the value on top of the stack. Therefore, if the next object uses this value (in a `+` for example) the dynamic type test on the value is then unecessary.

The second possibility to build up the context is to take advantage of the lazy compilation and the organization of the objects, that is similar to continuations, to have more information at execution. Let us take, for example, the expression `(+ (+ a 1) (+ a 2))`. In this case, a flow-sensitive static analysis [8] should detect that the second type test on `a` is not necessary. Even if such analysis can be performed in fast way, their cost is significant for a JIT compilation. The figure 8 shows, though simplified, the lazy code objects created for this expression. This figure also shows an example of execution where both the information about the type of each value on the stack and the types related to identifiers are contained within the context. As soon as the expression is executed the generation of the first block starts. The stack is then empty and the compiler doesn't know the type of `a` yet. The compiler will generate the code for `a` and then start the generation process of the next block with the new context within which the stack contains an only `unknown` element. When the code of the first primitive `+` is created for that version, the compiler doesn't know the type information regarding `a` and will then generate a dynamic test. The next block in the execution flow will then be created with the new context within which the compiler already knows that `a` is a number. When the second primitive `+` is reached, the compiler knows that the two operands are numbers and no dynamic test is then necessary. The process will be the same for the third primitive. Here, we notice that the compiler can take advantage of this design to improve the context and remove useless tests without influencing performance which static analysis would do.

5. Experimental results

As our implementation is still a prototype, it is then impossible to run large tests. However, current tests show that the compiler removes a lot of dynamic type tests. As an example, figure 5 shows the execution times needed to compute the 40th Fibonacci number in 3 ways.

The first one uses LC. This execution time also contains JIT compilation. The second one is to execute a binary compiled using Gambit in similar conditions (e.g. inline primitives) and the last one is the same than the previous though in not safe and fixnum mode. So the third case does not execute any type tests and can be taken as a reference.

Although the use of LC makes the computation slower, it's only slower by a factor of about 1.33. There are two explanations to this result: The compiler currently doesn't do any optimization on generated code and uses a stack as execution model.

In the example of Fibonacci recursive function, code versioning allows the compiler not to execute any test if $n < 2$ (instead of 1 without type information of code versioning) because the compiler knows the parameter type. It will execute exactly 2 tests for a call with $n > 1$ (instead of 5 without type information of code versioning) because the com-

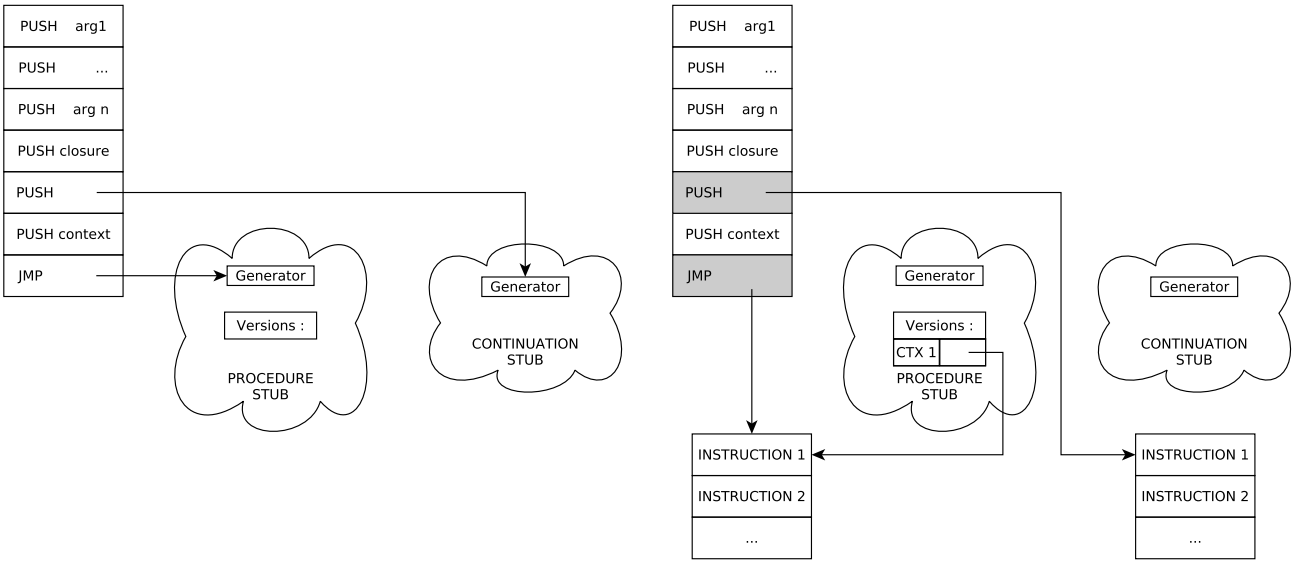

Figure 7. Procedure call before and after execution

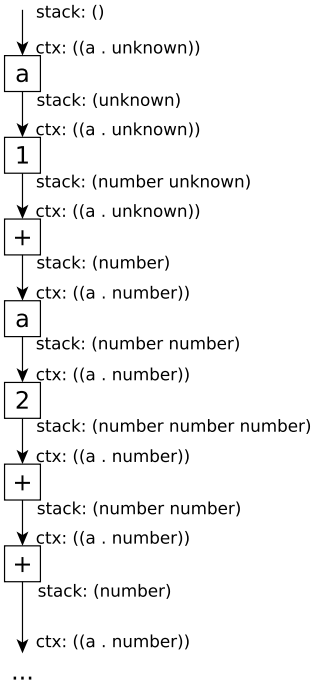

Figure 8. Simplified representation of lazy code objects chain for expression `(+ (+ a 1) (+ a 2))` with example of context during generation.

Implementation	Time (ms)
LC	2411
GSC	1810
GSC (fixnum and not safe)	757

Figure 9. Execution time to find the 40th Fibonacci number.

piler must test the type of returned values for the addition operands.

6. Limitations

6.1 Closure size

The combination of code versioning and extreme laziness has some limitations. In this section, we explain these limitations, and discuss potential solutions. The first major limitation is the construction of closures. As said in Section 3 all closures must follow the same mapping of context to entry point index in cc-table. Therefore all closures are the same size in heap whether it exists a lot of versions or none for a procedure. Because the actual number of used contexts itself depends on execution, thus the better way to measure the impact of this limitation in heap is by empirical way, but because LC is now at an early stage of development we are not yet able to measure this impact. Because we focus only on types, two contexts are equals only if they contain the same types:

```
ctx1 = (number number  number)
ctx2 = (number boolean number)
ctx3 = (number number  number)
```

In this example the 3 contexts represent a stack frame containing 3 values. Here *ctx1 = ctx3*, *ctx1 != ctx2* and *ctx2*

!= ctx3. So all procedure entry contexts could be represented by a list of types corresponding to the types of the arguments. We know that the maximum number of contexts for a procedure with `p` parameters is exponential and would have serious consequences on memory. Even though this is an extreme case wherein all possible contexts are actually used, it is a case we have to handle. There are several possibilities to limit this maximum:

Function curryfication could completely avoid the closure size problem at a price of performance.

Limit the number of contexts by simply stopping the generation of versions if a fixed number of contexts is reached.

Keep only *hot* contexts in cc-table. We can only generate versions of procedures for contexts frequently used.

Combination of previous points

6.2 Generated code size

The other important limitation is the size of the generated code. This problem is similar to the size of closures because again it's not possible to theoretically predict how many versions will be generated and the final code size. An empirical study could give us more information on its impact. We expect that lazy-compilation, which allows compiling only executed code will balance the size problem caused by the coexistence of multiple versions of the same code. Moreover, in most of current systems, memory is a less precious resource than performance. A possible consequence of code versioning could be that it is not a suitable technique for embedded systems and others memory limited platforms.

6.3 Return type

To be more efficient and have as much information as possible, the spread of the context is really important. We gave before the example of the call sites. If we have more information on arguments types in current context, we can generate more specialized versions of the function as long as this information is spread to the function stub. Another important spread of the context is all the information about returned value from function to the call continuation stub. LC currently loses this information by assuming that the returned type is `unknown` even if the type is known in function context. The reason is that we cannot patch the call site directly by replacing the stub address by the generated version address because another execution with the same context may cause a different execution and maybe a different return context.

```
(define (foo n)
   (if (even? n)
      42
      #f))

(foo m)
...
```

This example illustrates the problem. Assuming `m = 3` and we know that `m` is a number, the call `(foo m)` will cause the compiler to generate a version of `foo` with a number as entry context. When the function returns, we know that the returned value type is a boolean because `(even? 3)` is false. It's clear here that the same call with the same context, for example with `m = 4`, will use the same entry point of previously generated version of `foo` but results in a different entry context for the continuation. This entry context depends both on the call site and on the context of the return point of the function. So the only possible way to correctly spread the context from return point is to associate a return destination with both caller and context. LC does not currently use this kind of mechanism.

7. Future work

In its current state there is a lot of work to do on LC compiler to reach a decent implementation of Scheme which uses code versioning with extreme laziness. This section presents the most important work to accomplish.

First, the information in context is really important. If it has more information, the compiler can generate more specialized versions and remove more dynamic tests. An important way to reach this goal is to correctly spread the new information over execution. We said that one of the limitations of the current implementation of LC is that the spread of returned value information is not yet handled. This particular point is one of our future work.

Another important task is to perform some experiments to validate or invalidate both techniques. The first step will be to reach a more decent implementation, then we will execute some standard Scheme benchmarks and then study these results and compare them to other implementations.

8. Related work

Some works have been done on getting as much information as possible on types such as tagging optimization. Henglein [5] improved this optimization using type inference and apply it to Scheme language. Such optimizations require one or more passes on the representation and require additional calculations which goes against our goals and the extremely lazy compilation. Moreover, these optimizations are mainly used to increase the use of procedure inlining to generate an unique optimized version of the code which works regardless of the entry context.

Adams et al. [8] developed a flow-sensitive analysis to infer types based on the static analysis CFA. Although this algorithm reduces the cost of traditional CFA and is flow-sensitive, so it take cares of type predicates and others related operators, it performs analyses in $O(nlogn)$ which is significant for a JIT compiler and implies this technique to be more suitable for AOT compilation. As explained in sec-

tion 4.4, our technique removes type tests and take care of execution flow without additional cost.

A more close work than ours was done by Chambers and Ungar for *self* language [4]. Their technique, *code customization*, is used to generate multiple versions of the same procedure specialized depending on the type of the message's receiver. Moreover, this technique takes only advantage of type information while code versioning is extensible to other uses (e.g. register allocation).

Gal et al. [6] suggest to accumulate type information to specialize traces in order to remove some dynamic type tests. This technique is called *trace-based compilation*. This technique implies the use of a trace-based compiler and is made to specialize code at loop level. On the other side code versioning specialize each piece of code. While trace-based optimization is close to our approach, it implies the use of both a compiler and an interpreter and then rely on a more complex architecture than code versionning which only uses a compiler.

The work done on *Mercury* compiler [9] is also worth mentioning. This compiler uses a similar design than ours by using a lazy code generator for example to improve register allocation, but this compiler only uses lazy design to delay code generation for AOT compilation.

9. Conclusion

This paper has presented the technique of code versioning which allows the compiler to generate multiple versions of machine code based on compile-time known information. LC is our implementation of a Scheme compiler which uses code versioning coupled to an extremely lazy compilation design which improves its effect. LC currently is at an early stage of development and we are not yet able to measure the actual benefits of this technique as well as its impact on generated code size. There is a lot of remaining work to reach a decent implementation of the Scheme language which exploits code versioning / extreme laziness and correctly spreads context over execution.

The current tests show that LC compiler removes a lot of dynamic type tests on generated code. This is why we are expecting good results for this technique. The next step will be to validate the results by experiments.

References

[1] Gerald Jay Sussman and Guy L Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[2] Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. Compilers: Principles, techniques, and tools, 2006.

[3] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35. ACM, 1989.

[4] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN Notices*, volume 24, pages 146–160. ACM, 1989.

[5] Fritz Henglein. Global tagging optimization by type inference. *ACM SIGPLAN Lisp Pointers*, (1):205–215, 1992.

[6] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44, pages 465–478. ACM, 2009.

[7] R Kent Dybvig. *Three implementation models for scheme*. PhD thesis, University of North Carolina at Chapel Hill, 1987.

[8] Michael D Adams, Andrew W Keep, Jan Midtgaard, Matthew Might, Arun Chauhan, and R Kent Dybvig. Flow-sensitive type recovery in linear-log time. In *ACM SIGPLAN Notices*, volume 46, pages 483–498. ACM, 2011.

[9] Thomas C Conway, Fergus Henderson, and Zoltan Somogyi. Code generation for mercury. In *ILPS*, pages 242–256, 1995.

[10] Google v8 javascript engine. `http://code.google.com/p/v8/`.

[11] Chicken Scheme. `http://call-cc.org/`.

[12] M Feeley. Gambit Scheme. `http://gambitscheme.org`.

[13] Extempore language and environment. `http://benswift.me/extempore-docs/index.html`.