Type Check Removal Using Lazy Interprocedural Code Versioning

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

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

Abstract

Dynamically typed languages use runtime type checks to ensure safety. These checks are known to be a cause of performance issues. Several strategies are used to remove type checks but are expensive in a JIT compilation context or limited in the absence of code duplication. This paper presents an interprocedural approach based on Basic Block Versioning that allows the removal of many type checks without using an expensive analysis while simplifying the compilation process by avoiding the use of an intermediate representation. The experimentations made with our Scheme implementation of the technique show that more than 75% of type checks are removed in generated code.

1. Introduction

Dynamic typing lets the compiler verify the type safety at runtime through type checks directly inserted into the generated machine code. These operations are known to be a cause of performance issues of dynamically typed languages.

A lot of work has been done to reduce the cost of type checks. Type inference [5, 10] determines types, if possible, at compile time to avoid checks in the generated code. Tracing JIT compilation [9] interprets code to collect information, including types, during execution in order to generate optimized code using this information. Both techniques are not effective at removing type checks on polymorphic variables with known types. For example if an analysis shows that a variable `n` could only take the types `string` and `char`, then to be conservative the compiler will surround primitives using `n` with a type check. Another approach is to use Basic Block Versioning [3] (BBV) in a Just In Time (JIT) compiler to lazily specialize generated code depending on information gathered during previous executions by duplicating polymorphic code. In the same example as above, two different versions of the code will be generated, one for `string` and one for `char` as required by the actual type of `n` during multiple executions. In addition to this more precise context-dependent strategy, BBV doesn't need an expensive analysis or fixed point algorithm to infer types.

This paper presents a JIT compilation technique based on BBV which extends the original technique and addresses issues encountered in its implementation in a compiler for Scheme [13]. The first contribution is an extremely lazy compilation design which allows the compiler to directly translate s-expressions into stubs able to generate machine code. This allows the compiler to avoid the use of an intermediate representation such as Single Static Assignment form [4] and Three Address Code [11] and consequently save compilation time. This is particularly adapted in our context of JIT compilation in which compilation time directly impacts execution time. The other contribution is the use of multiple specialized function entry points allowing the compiler to propagate gathered typing information through function calls.

This paper is organized as follows. Section 3 presents the general approach and how types are discovered with the use of extremely lazy compilation. Section 4 explains how we extended code versioning to propagate accumulated information interprocedurally. Section 5 explains the problem introduced by free variables and how it is solved. Section 6 presents experimental results. Related work and future work are presented in sections 7 and 8.

2. Basic Block Versioning

Basic Block Versioning is an approach allowing to generate several specialized versions of a basic block. Each version is specialized according to the information available when compiling this block. The information is gathered from the compilation of the previous basic blocks in the execution flow therefore the technique does not require static analysis or profiling.

Gathered information could be the type of live variables. Because it is hard to predict all types used during execution, the compiler can't generate all versions ahead of time without a combinatorial explosion. JIT compilation allows to only generate versions actually

executed. Because BBV allows keeping several versions of the same code, the compiler can generate specialized versions based on variable types even if the code uses polymorphic variables.

Here is a simple example using type information to generate specialized versions of basic blocks:

```
(if (number? a)
   (< a 100)
   ...)
```

When compiling this code, if the compiler knows that `a` is a `fixnum`, it generates a specialized version of the true branch using comparison on fixnums, and without type check for primitive `<`. All the subsequent executions in which `a` is known to be a `fixnum` will use this specialized version. If, with another execution, `a` is known to be a `flonum`, the compiler generates a new version using a floating point number comparison and no type check. All the subsequent executions in which `a` is a `flonum` will use this version.

We then have two versions of the same code specialized for particular compilation context. Each time a version is generated, the compiler may discover new information that will possibly cause the generation of new more specialized versions of the successor basic blocks. Extremely lazy compilation aims to use code versioning to simplify the compilation process.

3. Extremely Lazy Compilation

3.1 Presentation

Typical compilers use an intermediate representation such as SSA or TAC. These representations are used to facilitate static analysis and code generation but they are expensive to generate. This compilation overhead is problematic if the implementation uses a JIT compiler because the compilation time impacts the execution time.

Extremely lazy compilation aims to simplify the implementation of code versioning by directly transforming the AST into code stubs with little overhead.

The idea is to do a more fine-grained JIT compilation by representing each not yet executed continuation of the program as a machine code stub. Then, a compilation context is associated with each expression and all information discovered during the compilation of this expression can directly be beneficial to the compilation and execution of the next expression in the execution flow.

3.2 Implementation

To implement BBV, a compiler must maintain a compilation context which associates type information to each live variable of the current basic block. Because our implementation is based on a stack machine, where temporary values are quickly consumed, we decided to maintain a context containing type information of all variables available in the current scope to avoid a more expensive liveness analysis. This allows the compiler to translate from s-expressions to code stubs with no prior code analysis other than those used to properly implement the Scheme language (such as mutation analysis and free variable analysis). However the use of a stack machine is not a requirement for the implementation of extremely lazy compilation.

In our implementation, we decided to keep only the information of simple (not compound) Scheme types. For example when creating a pair, the value is tagged as `pair` and we lose the possibly known information of its `car` and `cdr`. This allows avoiding compound type tracking that rapidly causes combinatorial explosion of the types and therefore an explosion in the number of versions.

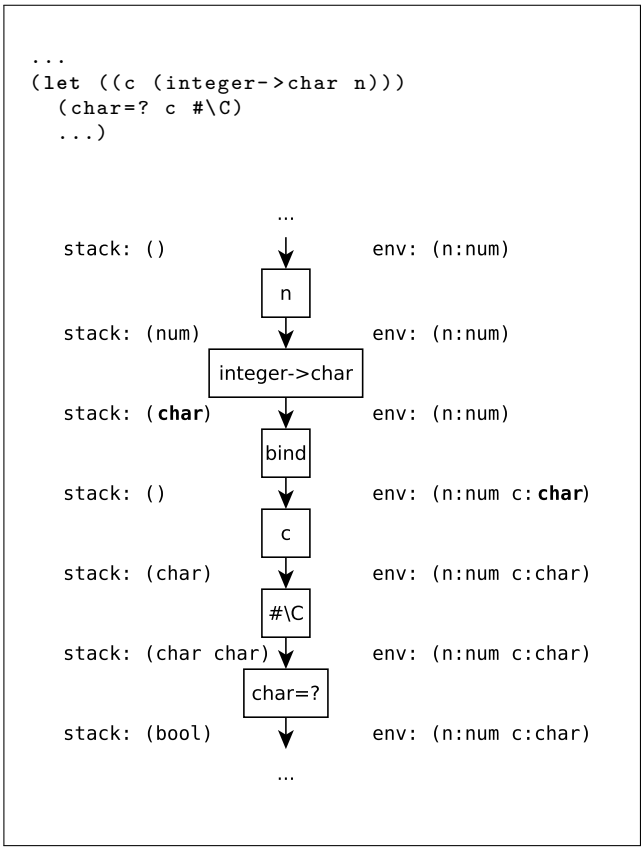

Figure 1. Example of a lazy code object chain

To implement extremely lazy compilation we create separate code stubs, which we call *lazy code objects*, for each piece of code and not only at a basic block level. Each object contains exactly three things *(i)* a code generator which, given a typing context, is able to generate a specialized version of the code associated to this stub *(ii)* a table which contains entry points of each already generated version, and *(iii)* a reference to the successor

object in the execution flow. Then these objects are organized in a similar way to Continuation Passing Style [13] using the successor reference to trigger the compilation of the continuation by giving it the newly discovered type information during compilation of the current expression. Figure 1 shows a simplified representation of the lazy code objects chain created from the associated Scheme code. If the compiler triggers the compilation of the first object with a context in which we know that `n` is a number and with an empty stack, the compiler successively triggers the compilation of the next object updating the context information at each step. We see that after the compilation of `integer->char` the compiler knows that a character is now on top of the stack. After binding `c` to the value on top of the stack, the compiler knows that `c` is a character for the rest of compilation and then compile a version of `char=?` in which it knows that both operands are characters. In this specific example, because no branching instruction is encountered, every object belongs to the same basic block thus all the chain is generated inline without extra jump instruction, exactly like the original approach of BBV. Thereby this extremely lazy design allows the compiler to keep type information of constants, or other newly discovered type for future compilation.

It is worth mentioning that using extremely lazy compilation, the compiler behaves like original BBV technique which allows it to also enrich the context with type information discovered from type checks previously executed in the flow (A type check is represented by a lazy code object, two successors and two distinct typing contexts associated to the two objects).

3.3 Chain construction

Figure 2 shows a simplified code of the function generating the lazy objects chain from a given s-expression. Similarly to CPS, the function also takes the successor lazy code object as a second parameter. If the function is called for the first time, an object with a generator able to generate the final return instructions sequence is given.

Each call to `make-lazy-code-stub` creates a lazy code object with the given code generator. A lazy code object is consumed by the function `jump-to` which is always called from within a generator. This function selects the version to jump to (the version associated to the current context) or generates a new inlined version if it does not exist yet. Each call to `gen-chain` creates a chain of lazy code objects ready to be consumed using the two functions `make-lazy-code-stub` and `jump-to` and returns the lazy code object representing the entry point of the chain.

Two cases are shown in the figure. In the first case the s-expression is a number then the compiler creates an object which, when triggered, generates a simple immediate push instruction and triggers the next object with an updated context. The second case shows an example of using the context. If the primitive `integer->char` is encountered the compiler generates a first object which, when triggered, is only used to trigger the right object depending on current type information, if the value is a number no check is needed, otherwise the object compiling a type check is triggered.

```
(define (gen-chain ast successor)
  (cond
    ...
    ((number? ast)
      (make-lazy-code-stub
        (lambda (ctx) ; Generator
          (x86-push ast)
          (jump-to successor
                   (ctx-push ctx CTX_NUM)))))
    ...
    ((eq? (car ast) 'integer->char)
      (let ((lazy-conv
              (make-lazy-code-stub
                (lambda (ctx) ; Generator
                  (x86-pop rax)
                  (x86-to-char rax)
                  (x86-push rax)
                  (jump-to
                    successor
                    (ctx-push (ctx-pop ctx)
                              CTX_CHAR)))))
            (lazy-check
              (make-lazy-code-stub
                (lambda (ctx) ; Generator
                  (x86-pop rax)
                  (x86-cmp tag_rax TAG_NUM)
                  (x86-jne label-error)
                  (x86-push rax)
                  (jump-to
                    lazy-conv
                    (ctx-push (ctx-pop ctx)
                              CTX_NUM))))))

        (gen-chain
          (cadr ast)
          (make-lazy-code-stub
            (lambda (ctx) ; Generator
              (if (eq? (type-top ctx) CTX_NUM)
                (jump-to lazy-conv  ctx)
                (jump-to lazy-check ctx)))))))
    ...))
```

Figure 2. Example of how to build a lazy code object chain from a s-expression and a successor lazy code object.

4. Interprocedural Type Propagation

4.1 Presentation

The approach presented in previous section aims to collect as much type information as possible during execution and compilation of previous lazy code objects in order to specialize the next objects in the execution

flow using this information. A limit is that this approach does not apply interprocedurally.

In order to transmit gathered information from function caller to callee the compiler needs to specialize functions entry points. This implies that each function possibly has several entry points depending on the type of actual parameters.

However commonly used closure representations such as flat-closure and others [7] only allow to store one entry point for the associated procedure. Because the arguments are possibly polymorphic in Scheme, and only one entry point is allowed, the compiler loses type information to use a generic entry point.

4.2 Implementation

Our solution to keep collected information is to extend the traditional flat closure representation by adding a reference to an external table which contains all entry points of the procedure, each one specialized according to the known types of parameters. This external table is associated to a procedure and therefore shared by every instance of this procedure. The initial entry point now represents the generic entry point without any assumptions on the type of parameters. This table is created at compile time, thus possibly in a dedicated memory area, and will live for the rest of the execution.

The problem with this external table is that with the higher-order functions of Scheme, the compiler doesn't necessarily know the identity of the callee function when compiling a call site, and is not able to determine the offset to use to get the right entry point from the table. Our solution to this problem is to keep a global layout shared by all the external tables which allows the compiler to associate a fixed offset to a specific context. Thereby the compiler is able to use this offset to retrieve the callee entry point regardless of the procedure identity.

When a procedure is first compiled, the compiler creates the external table and fills it with the function stub address. When compiling a call site, the compiler retrieves the offset associated to the calling context. If this context was never used before, a new offset is reserved to it. The generated code then gets the entry point (which is either the stub address or the address of a generated version) and jump to it. Note that the compiler adds the context as an additional argument to allow the stub to generate a version for this specific context. Whenever the stub is triggered, it generates the version and patches the external table entry which now contains the address of the newly generated version.

Figure 3 shows an example of a memory state after execution. At the top is the global layout in which we can see that each procedure call used one of the 5 contexts, regardless of the procedure called. Then we see that two procedures were compiled. The external table

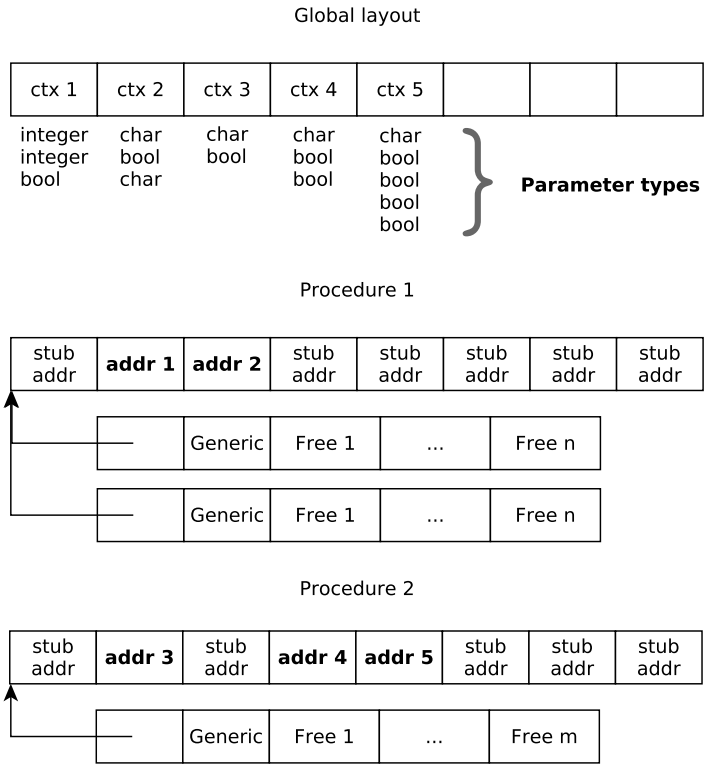

Figure 3. Extension of flat closure representation

of the first procedure contains two entry points which means that two specialized versions have been generated during execution. The first is associated to `ctx2` and the other to `ctx3`, all other slots contain procedure stub address. This procedure was instantiated two times and both instances share the same external table. Finally, three versions of the other procedure have been generated using a single instance. This time the three versions are specialized for contexts `ctx2`, `ctx4` and `ctx5`, and other slots contain the address of the code stub of the associated procedure. We can see in the figure that the offset associated to a context is actually invariant in all external tables.

4.3 External table limitation

This global layout could be a limitation if there is a combinatorial explosion on the types of parameters during execution. In this case, each external table must contain enough entries to store all of these contexts greatly increasing the memory used by the tables. Although this hypothetic explosion must be handled, our measures show that there is no such explosion in practice. Moreover, some simple heuristics can be used to reduce the size of the global table by removing the contexts in which we don't have *enough information*:

- If at a call site the compiler knows *nothing* about the type of parameters, it can simply use the fallback generic entry point. This eliminates all unnecessary entries from the global table and avoids the use of the indirection to retrieve the external table which is useless in this case.
- If the list of effective parameters in a calling context is *long* it probably means that they will be received in a rest parameter and the type information will be lost. In this case the compiler could use the fallback generic entry point.
- If the compiler doesn't know *enough* types on parameters, for example if there are 4 arguments and only one is known to be an integer, it could fall back to the generic entry point. In this precise case the cost of the indirection to get the offset from external table is more expensive than checking the type of an integer (as well as other non heap allocated objects) using tag types in callee function.
- Of course a better heuristic is probably a combination of heuristics.

A complementary aggressive solution could be to set a maximum allowed size for global layout and stop specializing entry points when the limit is reached. This can be done by using the generic entry point if a calling context, which doesn't exist in the global layout after reaching the limit, is used. This completely avoids the combinatorial explosion but potentially loses useful information. This is a technique to use as a last resort to prevent the hypothetical explosion.

The table presented in figure 4 shows the amount of memory (expressed in kilobytes) used by the entry point tables, the number of lines of code and the number of tables created for each benchmark. Because the standard library used by our implementation contains 110 functions, none of the benchmarks create less than 110 tables. The total size correspond to the perfect situation in which the size of the external tables is exactly equal to the minimum size required by the global layout. Our current implementation arbitrarily sets a constant size for the execution but there are two ways to avoid table overflows :

- Directly allocate a large amount of memory and stop specializing when the table is full. This can be coupled to the heuristics presented above.
- Use simple algorithm of dynamic reallocation to resize the external tables coupled to a garbage collector phase to update references.

The table shows that many benchmarks need less than 64 kilobytes to store the external tables. Only the benchmark `compiler` requires more (2.8 megabytes). The bigger memory footprint is not really significant considering the current amount of memory available on the devices.

Benchmark	Lines of code	Number of tables	Total tables size (*kb*)
compiler	11195	1561	2847
earley	647	187	64
conform	454	208	47
graphs	598	161	43
mazefun	202	149	37
peval	629	187	31
sboyer	778	149	23
browse	187	128	16
paraffins	172	133	14
boyer	565	134	13
nqueens	30	117	12
dderiv	74	121	8
string	24	113	5
deriv	34	112	4
destruc	45	113	4
perm9	97	117	4
triangl	54	112	4
array1	25	115	3
cpstak	24	116	3
primes	26	114	3
tak	10	111	3
ack	7	111	2
divrec	15	112	2
sum	8	112	2
cat	19	112	<1
diviter	16	112	<1
fib	8	111	<1
sumloop	22	113	<1
takl	26	113	<1
wc	38	112	<1

Figure 4. Space usage of the external tables

4.4 Impact on calling sequence

The technique presented in this section allows the compiler to propagate the collected information through the call sites using the external entry points table. This however requires changes in calling convention.

Figures 5 shows the additions made to common calling convention. This figure assumes that the called closure is in `r8`. The more expensive one is the indirection to retrieve the external table from the closure. In fact this cost is the same of the one introduced by virtual method table of object oriented programming using single inheritance [6]. But this indirection cost is compensated if the information in the context avoids at least one type check on a heap allocated object such as `string` or `pair` in Scheme because this check requires a memory access to retrieve the sub-tag representing

```
;Get external table location from closure
mov rax, [r8]
;Get entry point
mov rax, [rax+ctx_offset]
;Add context id as extra argument
mov rdi, ctx_id
;Call entry point
call rax
```

Figure 5. Calling sequence with interprocedural propagation (Intel syntax)

the type. The other is the extra `mov` used to give the context (the constant `ctx_id`) to the callee in case the call triggers a function stub. This time the move cost is directly compensated by the fact that the compiler doesn't need to give the number of actual parameters because the stub can retrieve this information directly from the context.

The interprocedural type propagation presented in this section only applies to the function entry points. Currently, our implementation does not track the type of returned values.

5. Free Variables

The presence of higher order functions means that in general, the compiler doesn't know the identity of the called function when compiling a call site. Thus, when compiling a call site it doesn't have any information on the type of the free variables so it is only able to specialize the entry point regarding the type of parameters. With specialized entry points, if two instances of the same closure but with different free variable types are called at the same call site, the same entry point is used, potentially resulting on an error. Lets take the well-known functional adder as an example:

```
(define (make-adder n)
   (lambda (x)
      (+ n x)))

(let ((add10 (make-adder 10))
      (add#f (make-adder #f)))

   (add10 1)
   (add#f 1))
```

In this code two adders are created. The first adds `10` to its argument. The second tries to add `#f` to its argument and causes an error. When calling both adders, the calling context is the same because in both cases there is only one argument which is known to be a number. It is then obvious that both instances can't share the same entry points table because the free variable `n` is polymorphic.

The easiest solution, which doesn't lose the gathered type information of free variables is to specialize the external table of a function according to the type combinations of its free variables. It is then possible to have several external tables shared between the instances, with same free variable types, of a function. In the example above, because the tables are specialized according to the type of free variables, both instances use a distinct entry points table. This handling of free variables allows to keep tracking their types, but slightly increases the number of external tables, and the amount of memory they use. The number of tables and the total size previously presented in figure 4 consider this approach.

Finally, if the compiled language allows variable mutation, the compiler is not able to specialize external tables regarding the type of mutable free variables because a type mutation could occurs at any time. The type of these variables can not be tracked.

6. Results

6.1 Number of tests removed

This section presents the results obtained with our Scheme implementation of lazy interprocedural code versioning. Figure 6 shows the number of runtime type checks executed with and without interprocedural propagation enabled without any maximum in the number of generated versions of the same lazy code object. The executed checks shown in this figure are percentages relative to an execution in which the maximum number of versions is set to 0 (i.e. only a generic version is used thus all type checks are executed). The extremely lazy compilation coupled to code versioning allow the compiler to remove a lot of type checks. For the benchmark `array1`, BBV removes almost all type tests. What is more interesting is that interprocedural propagation of type information allows the compiler to remove a lot more type checks. For the benchmarks `cpstak`, `string` and `sum`, the interprocedural propagation allows to remove almost all type checks. For the other benchmarks, the interprocedural propagation still removes a significant number of type checks. On average, around 63.7% of type checks are removed without interprocedural propagation and 77.2% with both BBV and interprocedural propagation.

6.2 Limiting the number of versions

We originally expected that the number of versions would grow faster than the original versioning for two reasons:

- The compiler specializes the versions according to the type information of all variables and not only live ones.
- Entry points are also versioned. Moreover the compiler specializes the entry points according to the type information of all actual parameters.

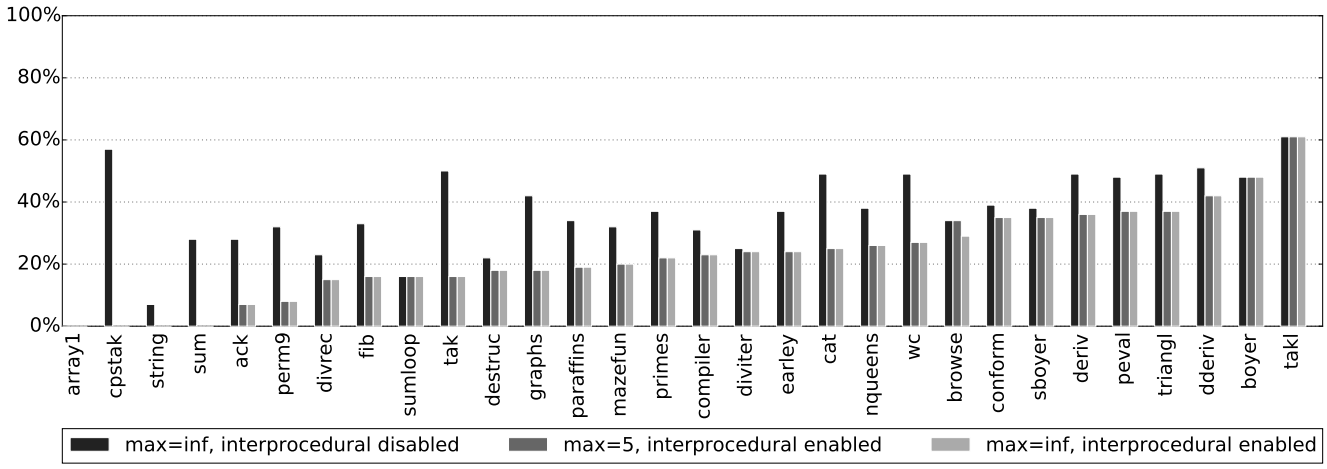

Figure 6. Percentage of executed check relative to generic versions

Figure 6 also shows the effect of changing the maximum number of versions on the number of type checks removed. We choose to show this result with a maximum of 5 versions to refer to the first presented BBV and to compare it to the result without limiting the number of versions. The benchmark `browse` is affected with a change of 4.5% which is not a huge increase in addition to being the only significantly affected benchmark. Moreover, our experiments showed that there is no pathological case causing an explosion on the number of versions as we would expect. However, our implementation currently doesn't support other number types than `fixnum` and because we think that a lot of type mutations occur with number-related operations such as integer overflow, it would be more interesting, once implemented, to study types evolution again so the effect of the number of maximum versions on the total amount of removed type checks.

A behavior worth mentioning appears in figure 7. This figure shows the percentage of removed type checks with a maximum of 3 versions and with and without interprocedural propagation enabled. We can see on benchmarks `browse`, `earley` and `nqueens` that when enabling interprocedural propagation more dynamic checks are executed. This is due to the fact that, because entry points are specialized according to the type of all actual parameters, a few versions among the limited number are *wasted* in the sense that a known type used to generate a new version is possibly attached to a variable which is not or little used in the rest of execution. When the limit is reached, all the subsequent versions use the fallback generic entry point whereas they are possibly based on type information attached to most used variables. This results in an increase of the number of executed type checks. Even if this behavior almost disappears starting from a limit of 4, it must be considered as a new parameter to consider when limiting the number of versions.

```
(define (fibcps n k)
  (if (< n 2)
    (k n)
    (fibcps (- n 1)
            (lambda (r1)
              (fibcps (- n 2)
                      (lambda (r2)
                        (k (+ r1 r2))))))))

(define (fib n)
  (fibcps n (lambda (r) r)))
```

Figure 8. CPS implementation of a function calculating the n^{th} Fibonacci number

6.3 Propagation of the returned value

Our implementation does not currently propagate the type of the return values. A mechanism close to the one used to specialize the entry points could be used, which would amount to using Continuation Passing Style. Let's use the code presented in figure 8 as an example. This code is a CPS program computing the n^{th} Fibonacci number. Each return site is transformed into a function call representing a call to the continuation. Because the compiler is able to propagate the type information through the function calls, the collected type information is propagated through the rest of the function. An interesting result with this example is that if the type of `n` is known to be a number when calling the `fib` function, and because of the CPS, absolutely no type checks are executed. If the compiler does not know the type of `n` its type is checked at the first ex-

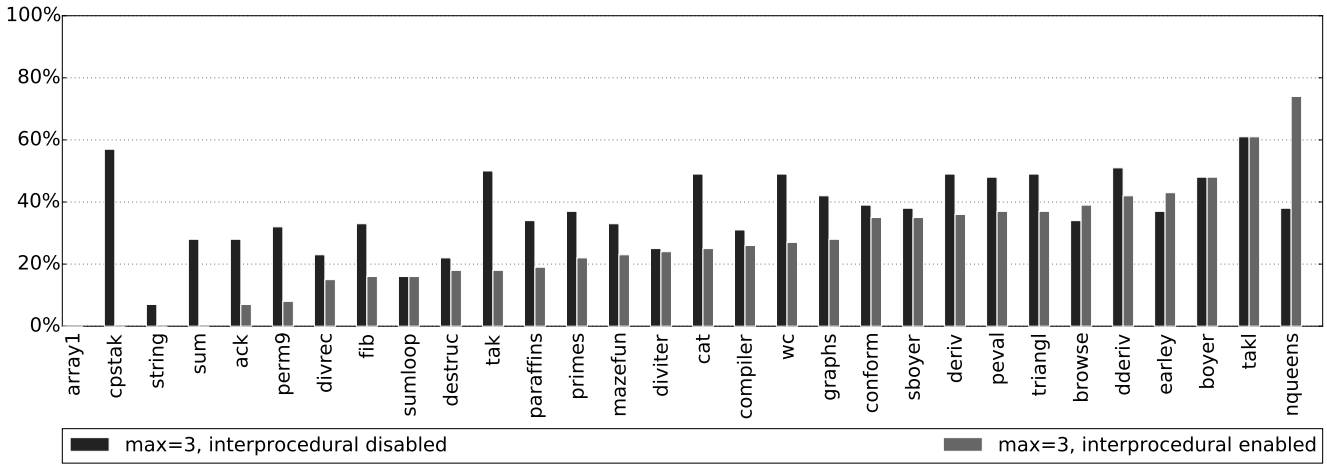

Figure 7. Percentage of executed check with limit on the number of versions set to 3

ecution of the expression `(< n 2)` and the information will be propagated to the rest of the program and this results in the execution of only one type test.

7. Related work

Several works have been done to remove dynamic type checks. Type inference [5] uses static analysis to recover type information from source program and allows to remove type checks in some cases. Henglein [10] also presented an interprocedural type inference in almost-linear time. Type inference performs expensive static work not necessarily suitable for a JIT compiler and is also often limited by the absence of code duplication.

Other approaches, such as Gradual Typing first presented by Siek [12], aim to remove dynamic type checks by explicitly writing type hints to the compiler. Ocurrence typing, improved by Logical Types, used in Typed Racket [14, 15], allows to infer more types and prevents the programmer from explicitly writing certain types. However, by letting the user explicitly write the type information, these approaches impact the simplicity of the language which is one of the main advantages of dynamically typed languages.

Other work attempts to remove type checks using code duplication. The well-known technique of Trace Compilation is often used in compilers to remove type checks [9]. Trace Compilation aims to specialize specific parts of the program according to the information gathered from profiling. But this technique requires the use of an interpreter to profile code and to record traces. Chang et al. presented a technique using Trace Compilation based on the observation of the actual types of variables at runtime to specialize code according to this information [2]. However this approach implies the compilation to a statically typed intermediate representation. In contrast with trace based techniques, Extremely Lazy Compilation aims to simplify the compilation process by using only a JIT compiler without any intermediate representation.

Finally, Bolz et al. presented a simple Scheme implementation based on Meta Tracing [1]. Because this simplicity is close to our goal to simplify the compilation process, it could be interesting to compare performance between both implementations.

8. Future work

First, we would like to improve the interprocedural propagation of context by keeping the type of returned values. As explained in section 6, CPS conversion is a good starting point to explore the effects on the generated code.

Another work should be to improve our implementation to better evaluate performance of the technique. A short term goal is to implement others data types such as `flonum` which we think to be responsible for more polymorphic data. Then we should reanalyze the space needed by external tables as well as the impact of changing the maximum number of versions on the number of type checks removed.

Another improvement should be to consider register allocation in our implementation, first to explore the integration of register allocation information into the context and to be able to evaluate the technique by comparing performance with state of the art Scheme JIT compilers such as Racket [8].

Finally, a future work is to explore some heuristics, among those presented in section 4, to use in order to reduce the memory footprint of the external tables without adding expensive dynamic type checks.

9. Conclusion

This paper presents the technique of *extremely lazy compilation* which allows the compiler to discover the type of variables from compilation and execution of previous code in the execution flow. According to this type information, the compiler uses *code versioning* to generate specialized versions of the code to remove a lot of type checks executed at runtime, even if a variable is polymorphic. This paper also presents an interprocedural extension of code versioning allowing to propagate the type information gathered from extremely lazy compilation through function calls by specializing the entry points using an external entry points table.

Our Scheme implementation shows that, in average, more than 75% of type checks are removed but they introduce two potential flaws. First, the external tables impact the amount of memory used, but we showed this amount stays low in practice. The other is an additional cost due to the indirection used at call sites. But again, we showed this cost is rapidly compensated.

Extremely lazy compilation and interprocedural propagation can be improved especially by using CPS or derived form to propagate the type of returned values using the same mechanism as the one used for entry points. It would also be good to improve our current implementation to be able to better evaluate performances.

Because the techniques don't need any intermediate representation or expensive static analysis, they allow to quickly implement a language with a simple JIT compiler with reasonable performance. Our current implementation is a good starting point to experiment on code versioning for example with the integration of register allocation information in compilation context.

References

[1] C. F. Bolz, T. Pape, J. Siek, and S. Tobin-Hochstadt. Meta-tracing makes a fast Racket. In *Workshop on Dynamic Languages and Applications*, 2014.

[2] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical report, Citeseer, 2007.

[3] M. Chevalier-Boisvert and M. Feeley. Simple and effective type check removal through lazy basic block versioning. In *Proceedings of the 2015 European Conference on Object-Oriented Programming (ECOOP)*. LIPIcs, 2015.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[6] K. Driesen. *Efficient Polymorphic Calls*, volume 596. Springer Science & Business Media, 2001.

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

[8] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

[9] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *ACM Sigplan Notices*, volume 44, pages 465–478. ACM, 2009.

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

[11] M. Lam, R. Sethi, J. Ullman, and A. Aho. Compilers: Principles, techniques, and tools, 2006.

[12] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

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

[14] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. *ACM SIGPLAN Notices*, 43(1):395–406, 2008.

[15] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. *ACM SIGPLAN Notices*, 45(9): 117–128, 2010.