Interprocedural Specialization of Higher-Order Dynamic Languages Without Static Analysis

Baptiste Saleil[1] **and Marc Feeley**[2]

1 **Université de Montréal**
Montreal, Quebec, Canada
baptiste.saleil@umontreal.ca

2 **Université de Montréal**
Montreal, Quebec, Canada
feeley@iro.umontreal.ca

Abstract

Function duplication is widely used by JIT compilers to efficiently implement dynamic languages. When the source language supports higher order functions, the called function's identity is not generally known when compiling a call site, thus limiting the use of function duplication.

This paper presents a JIT compilation technique enabling function duplication in the presence of higher order functions. Unlike existing techniques, our approach uses dynamic dispatch at call sites instead of relying on a conservative analysis to discover function identity.

We have implemented the technique in a JIT compiler for Scheme. Experiments show that it is efficient at removing type checks, allowing the removal of almost all the run time type checks for several benchmarks. This allows the compiler to generate code up to 50% faster.

We show that the technique can be used to duplicate functions using other run time information opening up new applications such as register allocation based duplication and aggressive inlining.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases Just-in-time compilation, Interprocedural optimization, Dynamic language, Higher-order function, Scheme

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.23

Supplementary Material ECOOP Artifact Evaluation approved artifact available at
http://dx.doi.org/10.4230/DARTS.3.2.14

1 Introduction

Dynamic languages typically have lower performance than static languages. A major reason for this is that the compiler has less information about the program at compile time and must postpone work to execution time.

Compilers generally use static analysis to predict execution time properties of the executed program. The collected information allows the compiler to generate specialized versions of functions according to the compilation context (i.e. the set of properties of the program).

In a higher order language with first-class functions, the specialization can depend on multiple sources of information. To generate efficient code, functions are specialized using *(i)* information available when compiling the call to the function and *(ii)* information captured when creating the lexical closure of the function. A lexical closure is a memory object used to implement first-class functions that stores the function and an environment containing the

© Baptiste Saleil and Marc Feeley;
licensed under Creative Commons License CC-BY

value of the free variables (i.e. the variables used by the function but defined in an enclosing scope).

Code specialization is generally used along with JIT compilation to only generate the actually executed versions.

A specialized function entry point is then uniquely identified by the triplet:

$$\mathcal{I}_{\text{lambda}}, \mathcal{P}_{\text{closure}}, \mathcal{P}_{\text{call}}$$

With $\mathcal{I}_{\text{lambda}}$ the identifier of the function, $\mathcal{P}_{\text{closure}}$ the captured properties at the closure creation site and $\mathcal{P}_{\text{call}}$ the properties at the call site. For example, if the compiler specializes the code according to type information, $\mathcal{P}_{\text{closure}}$ is the type of the closure's free variables and $\mathcal{P}_{\text{call}}$ is the type of the actual parameters. When a call site is executed, a new version of the callee is generated, specialized according to $\mathcal{P}_{\text{closure}}$ and $\mathcal{P}_{\text{call}}$ if such a version does not currently exist.

When a programming language allows the use of higher order functions, the identity of the callee is not generally known when compiling a call site thus only $\mathcal{P}_{\text{call}}$ is available. If we still want to use interprocedural specialization in the presence of higher order functions, two problems must be addressed.

The first problem is that if the callee is unknown, the function entry point is retrieved from the lexical closure. However, classical lexical closure representations store a single entry point. Because the functions are specialized, they possibly have several entry points that must be stored in the closure. These representations must then be extended to be used with function specialization. The second problem is that if $\mathcal{P}_{\text{call}}$ is the only information available, and if the closure allows the compiler to store several specialized entry points, a dynamic dispatch must be performed using the closure and $\mathcal{P}_{\text{call}}$ to branch to the corresponding specialized entry point.

An analogous problem exists for function returns. Because they are specialized, function continuations possibly have several entry points. Thus classical representations of return addresses must be extended. Because the continuation called at a return site may be unknown, a dispatch must be performed to branch to the right continuation entry point. Note however that function returns can be translated into function calls using Continuation-Passing Style (CPS) meaning that it can, in principle, be solved in the same way.

This paper presents a new JIT oriented technique allowing the compiler to lazily generate specialized versions of the functions according to $\mathcal{P}_{\text{closure}}$ and $\mathcal{P}_{\text{call}}$ in the presence of higher order functions. The technique is simple and does not require the use of static analysis nor a profiling phase. Unlike existing JIT techniques such as [9], our technique is based on dynamic dispatch instead of on the discovery of the function identities. This enables interprocedural specialization to be used at every call site. We also show that the technique can be used to solve the problem for function returns without requiring an explicit CPS conversion.

Contributions. The first contribution is an extension of the widely used flat lexical closure representation [12]. This extension allows the compiler to store several entry points in the lexical closure to address the first problem introduced by higher order functions. The second contribution is an efficient dynamic dispatch based on position invariance. This dynamic dispatch allows branching to the appropriate entry point using information available at the call site only, solving the second problem introduced by higher order functions.

The rest of the paper is structured as follow. Section 2 introduces Basic Block Versioning (BBV), an existing JIT compilation approach. We show how BBV can be used to intraprocedurally specialize the code to collect $\mathcal{P}_{\text{call}}$ and $\mathcal{P}_{\text{closure}}$. Section 3 presents our

```scheme
(define make-sumer
  (lambda (n)
    (letrec ((f (lambda (x)
                  (if (> x n)
                      0
                      (+ x (f (+ x 1)))))))
      f)))

(define sum-to-10 (make-sumer 10))
(define sum-to-pi (make-sumer 3.14))

(println (sum-to-10 6))     ; 6 + 7 + 8 + 9 + 10
(println (sum-to-10 7.5))   ; 7.5 + 8.5 + 9.5
(println (sum-to-pi 1.10)) ; 1.10 + 2.10 + 3.10
```

Figure 1 Scheme code of an arithmetic sequence generator with a common step of 1.

contributions. We first present the lexical closure extension and then the dynamic dispatch based on position invariance. In section 4 we present the implementation of the technique in a JIT compiler for Scheme [24], and how the implementation problems are solved. Section 5 presents the results of the experiments and the impact of the technique on the generated machine code and the execution time. Related and future work are presented in sections 6 and 7 followed by a brief conclusion.

2 Basic Block Versioning

Basic Block Versioning (BBV) [7, 8, 9, 20, 21], is a simple JIT compilation approach based on code duplication allowing the compiler to generate multiple specialized versions of the basic blocks according to compilation contexts observed during execution of the program. A lazy compilation design is used to only generate optimized versions that are actually executed. BBV duplicates and specializes basic blocks on-the-fly and it does not require the use of an expensive static analysis or profiling phase nor the use of interpretation or recompilation.

In the examples that follow, for simplicity we will consider a language with only two concrete numerical types: fixed precision integers (*fixnums*) and floating point numbers (*flonums*). Generalization to a richer set of numerical types is obvious.

The Scheme code presented in figure 1 is an example showing how lazy intraprocedural BBV can be use to generate specialized code. In this example, function `make-sumer` generates bounded arithmetic sequence calculators with common step of 1. Two sumers are created using respectively an upper bound of `10`, a *fixnum* and `3.14`, a *flonum*.

Figure 2 shows the CFG that is generated by BBV while executing function `f` presented in figure 1.

Initially, the function is represented by its compilation stub. A compilation stub is a piece of code calling back into the compiler from the generated code. The stub stores information such as the context the compiler uses to generate the code when the stub is triggered. When the stub associated to function `f` is triggered for the first time, the compiler has no information on the type of `x` and `n`. It must thus generate a run time type check to determine if `x` is a *fixnum* for the test `(> x n)`. Two stubs are then created to handle the

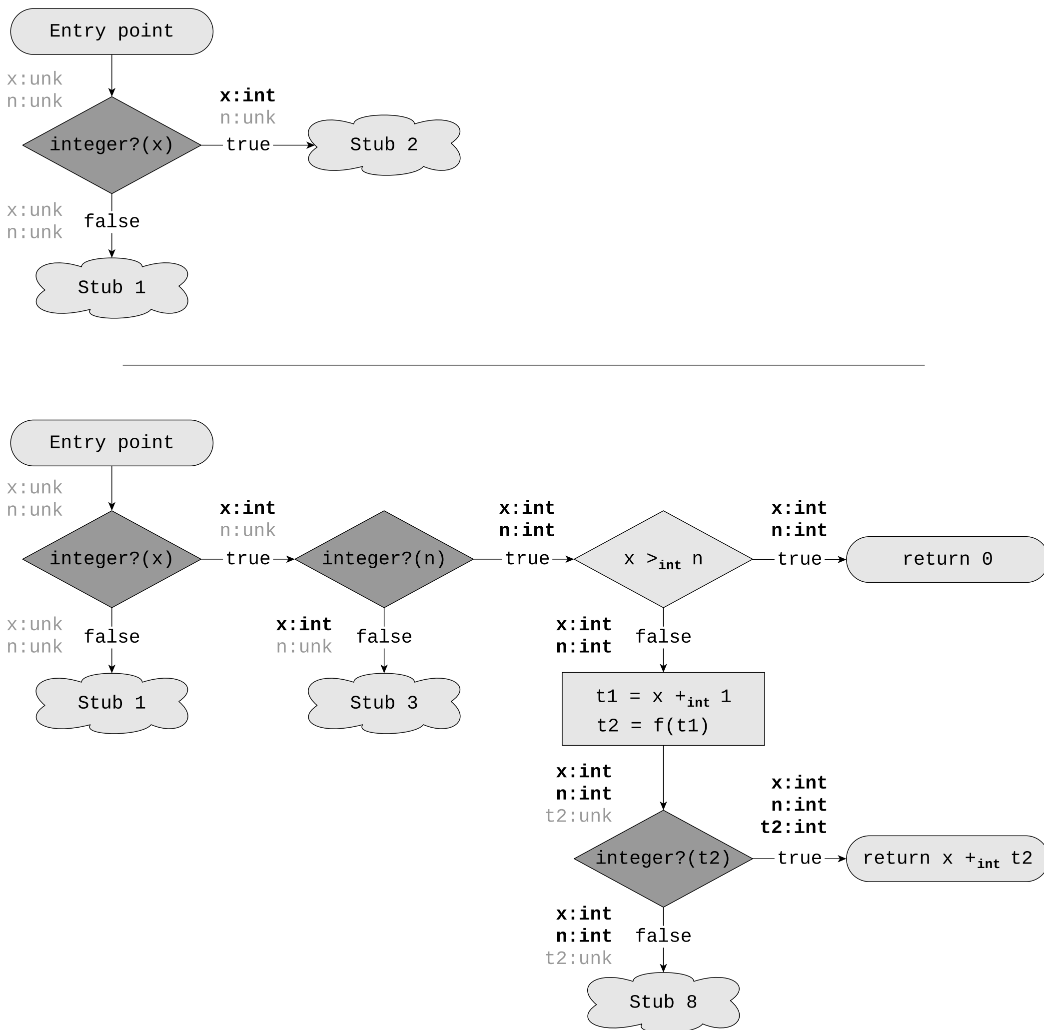

Figure 2 Progression of compilation of function `f` using Basic Block Versioning.

two outcomes. They are set up to compile the following code using an augmented context. This state is shown in the top of the figure.

When the check executes, if it succeeds, the compiler generates the following code using a context in which it knows that `x` is a *fixnum*. If the type check failed, the compiler would generate another check to determine the type of `x`. The bottom of the figure shows the resulting generated code of function `f` called with `(sum-to-10 6)` after all the executed stubs have been successively triggered. Initially, the type of the variables are unknown. When the first check executes and succeeds, the type of `x` is discovered and propagated through the stubs. The types of `n` and `t2` (the value returned by the recursive call) are discovered and propagated when the second and third type checks are executed. Because it is propagated, the type of `x` is known when compiling the two additions thus no more check on `x` is inserted. This allows the compiler to generate specialized versions of the basic blocks containing only three checks instead of five. Basic blocks corresponding to type checks are emphasized in figure 2.

If function `f` is later called with `x` not being a *fixnum*, the stub `Stub 1` is triggered and a new check is inserted to check if `x` is a *flonum*. If that is the case, new versions of the basic blocks are generated using this information.

This results in two different function bodies each optimized for a particular compilation context and accessible from a single function entry point. The purpose of interprocedural specialization is to add the possibility to access the two versions using two different entry points, and to branch to the appropriate entry point using information known at call sites to avoid the dynamic type checks of the function parameters.

3 Interprocedural Specialization

We illustrated that BBV can be effective at propagating information intraprocedurally. When applied to typing, figure 2 showed that in our example, two checks can be omitted. The three remaining checks are respectively used to check the type of *(i)* the function argument `x`, *(ii)* the free variable `n` and *(iii)* the value returned by the recursive call.

If the compiler knows $\mathcal{I}_{\text{lambda}}$ when compiling a call site, it can use $\mathcal{P}_{\text{call}}$ to generate a specialized version of the function, and branch to it. In our example, this means that the first type check can be omitted for the recursive call. The same applies to return points (i.e. calls to the continuations), so the third check can be omitted. Finally, if the compiler is able to specialize the code according to captured information, in this example the type of the free variable, all checks are avoided.

In the presence of higher order functions, compilers generally use static analysis such as *0-CFA* [22] to predict which functions can be called at a given call site and which continuations can be invoked at a given function return. Such static analyses are conservative, meaning that if the analysis is not able to determine that the inferred property is verified in all executions, the property is discarded resulting in a loss of precision. Furthermore, in the absence of code duplication, checks must be inserted if the analysis inferred multiple properties for a given site. Moreover, these analyses have high complexity, making them unsuitable for use in JIT compilers.

Other techniques based on dynamic dispatch such as Polymorphic Inline Caching [16] allow branching to the appropriate version using one or more guards degrading the performance of polymorphic call sites.

For the rest of the paper, we consider the general case where the compiler does not know the identity of the callee functions for the example presented in figure 1.

3.1 Function call

To use interprocedural code specialization in the presence of higher order functions, the compiler needs to be able to branch to the appropriate function entry point using the closure and $\mathcal{P}_{\text{call}}$ only. In this paper, this situation is referred to as *propagation through entry points*.

This means that the compiler needs to be able to store the entry point of all the function's versions in its lexical closure. Classical closure representations only store a single address to a function. These representations must then be extended to store a set of addresses instead of a single address. Using a JIT compiler allows generating actually executed versions only, limiting the number of entry points stored in each closure.

Given this new closure representation, the compiler must generate a dynamic dispatch for a call site to branch to the appropriate entry point using $\mathcal{P}_{\text{call}}$. This dispatch can be implemented in various ways including run time hashing or position invariant based dispatch. Ideally we want the dispatch mechanism to execute in constant time.

In the example of typing, $\mathcal{P}_{\text{call}}$ is the type of the actual parameters known when compiling the call. Using the function associated to `sum-to-10` presented in figure 1 as an example, the compiler generates a dynamic dispatch using the context `(x:fixnum)` for the call `(sum-to-10 6)`. No version exists for this $\mathcal{P}_{\text{call}}$. A version of the function is then generated and its entry point is stored in the lexical closure. Another version is generated for the call `(sum-to-10 7.5)` using the context `(x:flonum)`. The entry point is also stored in the lexical closure. Assuming that the compiler does not specialize the code according to $\mathcal{P}_{\text{closure}}$, it uses the previously generated version for the call `(sum-to-pi 1.10)`. When compiling the specialized versions, the type of `x` is known thus no check on `x` is inserted in the expression `(> x n)`.

Propagation through entry points is effective at removing dynamic type checks. However, it is possible to propagate more than type information to specialize function bodies. Such information includes the location of the arguments, to avoid the generation of extra move instructions at call sites, or the constant values associated to the arguments to do lazy interprocedural constant propagation.

3.2 Function return

As explained previously, an analogous problem exists for function returns that can be seen as calls to continuations. In this case $\mathcal{I}_{\text{lambda}}$ is the identity of the continuation and $\mathcal{P}_{\text{call}}$ is the information available when compiling the function return. In this paper, this situation is referred to as *propagation through return points*.

A single return address cannot be used to represent the continuation. As for lexical closures, the representation must be extended to store several continuation entry points. Using a JIT compiler allows generating actually executed versions only, limiting the number of entry points stored in the entry point set.

The dispatch is implemented in the same way. Ideally, the dispatch mechanism should also execute in constant time.

Using the function `sum-to-10` presented in figure 1 as an example, the compiler generates a dynamic dispatch using the context `(x:fixnum)` for the base case when the function is called with `(sum-to-10 6)`. A new version of the continuation of the recursive call is then generated and its entry point is stored in the continuation entry point set. The same version is used for the other return point because the compiler determines that the result of the addition is a `fixnum`. When compiling the specialized version of the continuation, the type of the returned value is known thus no check is inserted for the addition using the returned value.

Propagation through return points is effective at removing dynamic type checks. However, it is possible to propagate more than type information to specialize continuation bodies. Such information includes the location of the returned value, to avoid the generation of extra move instruction at return sites, or the constant value associated to the returned value to do lazy interprocedural constant propagation.

3.3 Captured information

The solution presented for function calls and returns allows the compiler to specialize function and continuation bodies using only $\mathcal{P}_{\text{call}}$ and the closure. However because $\mathcal{P}_{\text{closure}}$ may vary from one closure instance to another it is not safe to use it to specialize the code when compiling the function body.

We need to add to the compiler the ability to generate specialized code according to $\mathcal{P}_{\text{closure}}$ in addition to $\mathcal{P}_{\text{call}}$. Our solution is to use a different entry point set (i.e. a

specialized entry point set) in the closure each time a different $\mathcal{P}_{\text{closure}}$ is observed. $\mathcal{P}_{\text{closure}}$ is then retained and stored in the compilation stub. This way, each time a dynamic dispatch causes a new version to be generated using $\mathcal{P}_{\text{call}}$, the compiler uses the propagated $\mathcal{P}_{\text{call}}$ and the retained $\mathcal{P}_{\text{closure}}$ to generate the version.

In the example of typing, $\mathcal{P}_{\text{closure}}$ is the type of the free variables known when instantiating the closure. We showed that the compiler is able to propagate types through entry points. This means that when the compiler generates the code to create the closure associated to `f` for the call `(make-sumer 10)`, the compiler knows that `n` is a *fixnum*. This information is retained by the compilation stub. The first time the dynamic dispatch generated for the call `(sum-to-10 6)` is executed, a new version is generated using $\mathcal{P}_{\text{call}}$ and $\mathcal{P}_{\text{closure}}$ merged. The version is then generated with the context `(x:fixnum,n:fixnum)` and its entry point is stored in the closure associated to `sum-to-10`. The compiler then knows the type of `n` when compiling the body and no type check is inserted in the specialized versions of `sum-to-10` and `sum-to-pi`.

Entry point set specialization can also be used to specialize the continuations using information available when creating an object representing a continuation. In the example of typing, $\mathcal{P}_{\text{closure}}$ is the type of the local variables live across the function call.

Entry point set specialization is effective at removing dynamic type checks. However, it is possible to capture more information when creating the closure instance. Such information includes the constant values associated to the free variables to do lazy interprocedural constant propagation.

4 Implementation

We have implemented interprocedural specialization and entry point set specialization in LC [20, 21] (Lazy Compiler), a JIT compiler for Scheme using Basic Block Versioning as its compilation strategy. LC directly compiles Scheme s-expressions to x86 machine code without using an intermediate representation. LC uses BBV to specialize code according to the type of the variables. In this section, type information is taken as an example. Entry point set specialization is discussed later in the section. For now specialization according to $\mathcal{P}_{\text{call}}$ only is considered.

4.1 Entry point set

To allow the use of higher order functions with function duplication, the closure representation has been extended to store a pointer to a set of entry points (entry point table) instead of a single entry point address in the closure. Because the entry points are shared by the instances of a closure, only one table is created per function and is shared by the instances.

The compiler creates these tables at compilation time. Because the tables are live during the execution of the program, they are allocated as permanent objects thus they do not impact garbage collection time in LC.

The entry point table is initially filled with the function stub address. When a call site is executed, there are two possible situations. *(i)* The dispatch fails, there is no version associated to this $\mathcal{P}_{\text{call}}$. The stub is called, a new version is then generated using this $\mathcal{P}_{\text{call}}$, a new index of the table is associated to this $\mathcal{P}_{\text{call}}$, and the version address is written in the table at this index. *(ii)* The dispatch succeeds, an index already is associated to this $\mathcal{P}_{\text{call}}$ and the control flows to this version using the address stored at this index.

Figure 3 shows an example of flat closure extension to store multiple entry points for the instances of `f` (`sum-to-10` and `sum-to-pi`). The left side of the figure shows the state

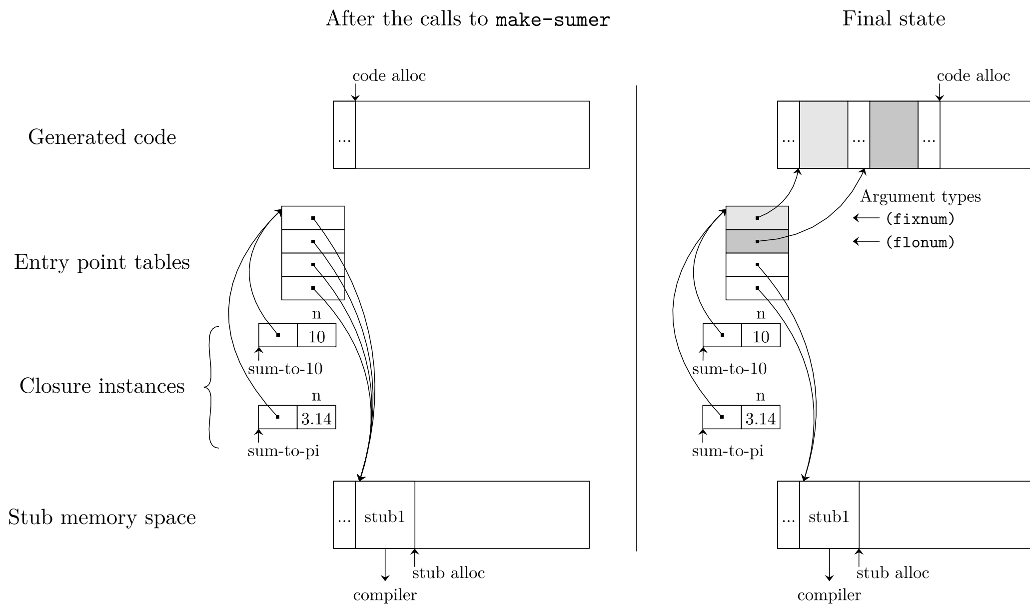

Figure 3 Extended flat closures using an entry point table.

after the two calls to the `make-sumer` function. A stub and two closures storing the free variables `10` and `3.14` are created. The two closures share the same entry point table which is initially filled with the stub address. The call `(sum-to-10 6)` causes a new version of `f` to be generated. The first slot in the table is then associated to the context `(fixnum)`. The call `(sum-to-10 7.5)` also causes a new version to be generated. The second slot in the table is then associated to the context `(flonum)`. The call `(sum-to-pi 1.10)` uses a $\mathcal{P}_{\text{call}}$ used by a previous call. The dispatch branches to the address stored in the second slot of the table. The right side of the figure shows the final state. The first two entries of the table now store the address of the two versions of the function.

The use of BBV allows the compiler to share code between multiple versions if the same compilation context is observed. For example if a new version of this function is generated for the context `(unknown)` the compiler generates a type check to discover the type of the argument `x` for the expression `(> x n)`. If `x` actually is a *fixnum*, the control simply flows to the existing version after this check.

4.2 Global layout

If the compiler knows the identity of the callee function when compiling a call site, it knows which slot is associated to a given $\mathcal{P}_{\text{call}}$. If it does not know the identity of the callee, a dynamic dispatch must be performed. We decided to use position invariance to efficiently implement this dispatch. This means that the compiler keeps a global layout shared by all the entry point tables. When a new $\mathcal{P}_{\text{call}}$ is associated to an index, this association is followed by the table of every function. Therefore, when the compiler compiles a call site, it determines the index associated to the current $\mathcal{P}_{\text{call}}$ from the global layout. The generated code uses this index to get the appropriate entry point from the table stored in the closure resulting in a fast dispatch. If no index is associated to this $\mathcal{P}_{\text{call}}$, the compiler uses the next available index and generates a jump to the address stored in the slot at this index. Because the table is initially filled with the stub address, the stub is triggered, a new version is generated and the table is patched.

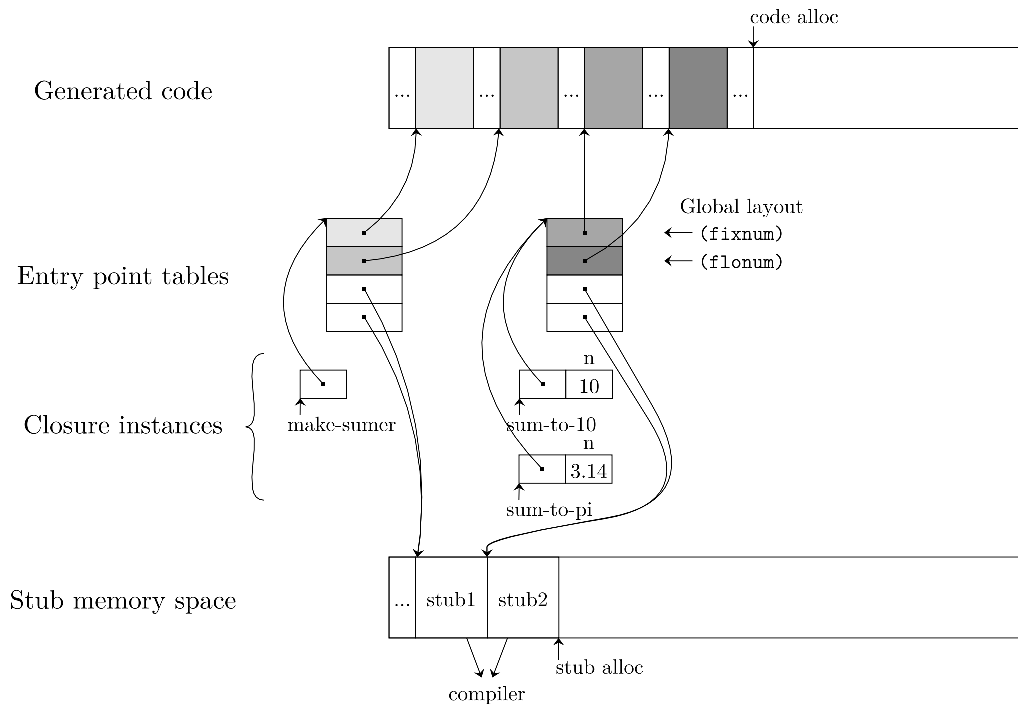

Figure 4 Extended flat closures using an entry point table and a global layout.

Figure 4 shows an example of using a global layout. In this figure all closure instances created during execution of our example are represented. After execution, we can see that two $\mathcal{P}_{\text{call}}$ have been used.

Two versions of function `make-sumer` are generated by the calls `(make-sumer 10)` and `(make-sumer 3.14)`. The compiler assigns the first two slots of the global layout to the contexts `(fixnum)` and `(flonum)`. Then, two versions of function `f` are generated by the calls `(sum-to-10 6)` and `(sum-to-10 7.5)`. The call `(sum-to-pi 1.10)` uses the version generated by the call `(sum-to-10 7.5)`. Because these three calls use contexts already associated to the first two global layout slots, the compiler reuses these slots.

A consequence of using a global layout is that the entry point tables may contain some *holes* associated to contexts for which the stub has not yet been triggered and will possibly never be.

4.3 Size of the tables

In the global layout, each entry is associated to a single context. If the only information used for entry point versioning is the type of the arguments, the compiler associates an entry to each combination of types observed when compiling a call site of the program. To avoid a combinatorial explosion, the compiler needs to limit the number of entries. We have identified three strategies regarding the size of the global layout:

- The size of the global layout is set ahead of time. One of the slots in the entry point table is initially associated to a fallback context representing a *generic* context. Each time a new $\mathcal{P}_{\text{call}}$ is used at a call site, the compiler assigns the next entry of the global

layout to this $\mathcal{P}_{\text{call}}$. When all entries are used, the compiler uses the fallback slot for all subsequent contexts and stops specializing function entry points.

- The tables are reallocated and copied when the global layout is full. This strategy implies that the compiler patches all live closures to update the table pointers.
- An additional level of indirection is used.

The compiler could use a combination of these strategies. It could resize the tables until a fixed size limit is reached. It could also apply some heuristics to reduce the number of entries used in the global layout. For example when specializing according to type information, a compiler could use the fallback generic entry point if:

- It does not know the type of a single argument. In this case, if the compiler assigns a slot for this context in the global layout, this slot is wasted.
- There are too many arguments. A large number of arguments probably means that a rest parameter is used by the callee function. If it is used, the arguments are stored in a compound data type thus type information is lost if the compiler does not propagate compound types (as is the case in LC).

4.4 Captured information

Using entry point tables, the compiler is able to propagate $\mathcal{P}_{\text{call}}$ to the callee function. When creating a closure, the compiler knows $\mathcal{P}_{\text{closure}}$. It can then easily keep this information to generate specialized function bodies. However, $\mathcal{P}_{\text{closure}}$ may vary from one closure instance to another. For example if $\mathcal{P}_{\text{closure}}$ is the type of the free variables, two instances of the same closure can hold data of different types. It is then not safe to use $\mathcal{P}_{\text{closure}}$ to specialize the body of the functions.

Our solution to this problem is to specialize the entry point table for each $\mathcal{P}_{\text{closure}}$ (in this case for each type combination of the free variables). Each time the compiler generates closure instantiation code, it checks if a specialized table exists for the current $\mathcal{P}_{\text{closure}}$. If the table exists, the compiler generates code to write the table address in the closure. If no table is associated to this $\mathcal{P}_{\text{closure}}$, it creates a new stub waiting for $\mathcal{P}_{\text{call}}$ and ready to compile using this $\mathcal{P}_{\text{closure}}$. A new entry point table is created and filled with this stub address. The address of the table is then written in the closure. Entry point table specialization allows the compiler to generate more efficient code using the $\mathcal{P}_{\text{closure}}$ it collected. However, the number of entry point tables and holes is increased.

In our example, because the compiler is able to propagate the type of the arguments through function calls, the type of `n` is known when generating the code to instantiate the closure associated to `f`. Figure 5 shows the closure representations at the end of the execution. We see that `sum-to-10` and `sum-to-pi` use a different entry point table. The table used in the closure associated to `sum-to-10` is specialized for $\mathcal{P}_{\text{closure}}$ =`(n:fixnum)` and the table used in the closure associated to `sum-to-pi` is specialized for $\mathcal{P}_{\text{closure}}$ =`(n:flonum)`.

Two versions of function `make-sumer` are generated by the calls `(make-sumer 10)` and `(make-sumer 3.14)`. The compiler assigns the first two slots of the global layout to the contexts `(fixnum)` and `(flonum)`. Then two versions of function `f` are generated by the calls `(sum-to-10 6)` and `(sum-to-10 7.5)`. Because these two calls use contexts already associated to the first two global layout slots, the compiler reuses these slots. Finally, a new version of function `f` is generated by the call `(sum-to-pi 1.10)`. This call uses the context `(flonum)` thus the compiler uses the second slot of the table associated to the closure `sum-to-pi`.

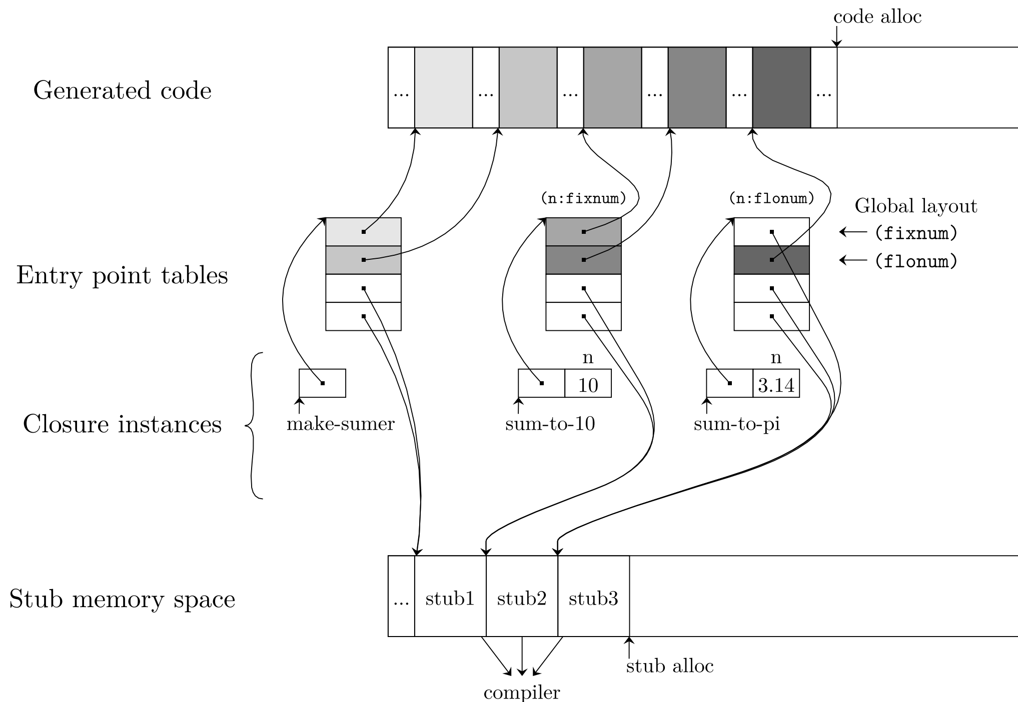

Figure 5 Extended flat closures using an entry point table, a global layout and table specialization.

To summarize, the following table shows the $\mathcal{P}_{\text{call}}$ and $\mathcal{P}_{\text{closure}}$ used for the different calls to the sumers:

Call	$\mathcal{P}_{\text{call}}$	$\mathcal{P}_{\text{closure}}$
`(sum-to-10 6)`	`(x:fixnum)`	`(n:fixnum)`
`(sum-to-10 7.5)`	`(x:flonum)`	`(n:fixnum)`
`(sum-to-pi 1.10)`	`(x:flonum)`	`(n:flonum)`

We see that the type of `x` and `n` are known for each call. Two specialized versions of `sum-to-10` and one of `sum-to-pi` are then generated and no dynamic type checks are executed.

4.5 Continuations

To allow propagation through return points, the compiler can convert the executed program to CPS. This way, each function return is translated into a function call. Specialization of the continuations is then directly handled by the specialization of the function bodies. However, the same technique can be implemented for function returns to avoid the CPS conversion.

Compilers typically use a single return address to represent a continuation. This address is written to the stack when executing the call and is used at the return point to jump to the continuation. Using interprocedural versioning, the continuation possibly has several entry points thus the continuation representation also needs to be extended. The compiler uses a set of continuation entry points (return point table) initially filled with the continuation stub address. The address of the table is written to the stack and a dispatch is performed at

Function call:

```
1  push 0203FF90h      ; setup return point table
2  mov  rbx, 40        ; setup first argument
3  mov  r11, 0         ; use first context index
4  mov  rdx, [rsi+7]   ; get entry point table from the closure
5  jmp  [rdx+16]       ; jump to entry point
```

Function return:

```
6  mov r11, 0          ; use first context index
7  mov rdx, [rbp]      ; get return point table from the stack
8  jmp [rdx+8]         ; jump to return point
```

Figure 6 Example of x86 function call and return sequences (Intel syntax).

return points to branch to the appropriate continuation entry point. Because the identity of the continuation may be unknown at function return, a global layout is used for the dispatch.

$\mathcal{P}_{\text{closure}}$ may also vary from one instance to another. Table specialization is then used to specialize the code according to $\mathcal{P}_{\text{closure}}$. In the example of type information, $\mathcal{P}_{\text{closure}}$ is the type of the live local variables known when instantiating the object representing the continuation.

Our implementation limits $\mathcal{P}_{\text{call}}$ to the type of the returned value (i.e. register allocation information is ignored). This choice simplifies the implementation. Each table has a fixed size equal to the number of types supported by the implementation with an additional entry for the *unknown* case. Each type is associated to an index in the global layout prior to execution forming the global layout.

$\mathcal{P}_{\text{call}}$ may not be limited to the type of the returned value. For example the compiler could propagate the register the value is assigned to. The same strategies as those presented for function entry points can be used to handle the tables.

4.6 Impact on generated code

This implementation has an effect on the code generated for call sites and return points. Figure 6 shows an example of generated code for a function call and a return site. In this figure, the code added to implement interprocedural specialization is highlighted.

4.6.1 Function call

At line 3, the compiler writes the index of the global layout associated to the current context in the register `r11`. In this example, the first index is used thus the compiler writes the constant `0`. If the call triggers the function stub (i.e. a version associated to this $\mathcal{P}_{\text{call}}$ has not yet been generated), the compiler uses this index to retrieve the $\mathcal{P}_{\text{call}}$ from the global layout. $\mathcal{P}_{\text{call}}$ is then used to generate the specialized version. This extra move may be omitted if a new stub entry point is created for each slot of the entry point table. We decided to keep this instruction to save the space occupied by these stub entries, and to use only one stub entry per table.

An extra move is generated at line 4. This instruction is used to retrieve the entry point table from the closure. The closure is represented by a tagged address in the register `rsi`.

An offset of 7 bytes is used to get the 64 bits (the entry point table address) following the closure header from the tagged address.

As shown at line 1, the continuation entry point table address is pushed to the stack instead of a single return address introducing no additional cost.

Because the first index is used, the compiler generates, at line 5, a `jmp` to the address stored in the first slot of the entry point table. An offset of 16 bytes is used because the first slot is located after the table header (64 bits) and the fallback generic entry point (64 bits).

If the compiler knows $\mathcal{I}_{\text{lambda}}$ (i.e. the identity of the callee) when compiling a call site, it retrieves the entry point from the table at compile time using $\mathcal{P}_{\text{call}}$. A single `jmp` instruction is then generated. If the compiler knows $\mathcal{I}_{\text{lambda}}$ but no version has yet been generated for the current $\mathcal{P}_{\text{call}}$, a single `jmp` instruction to the function stub can be generated and later patched. In those cases, no dispatch is inserted.

4.6.2 Function return

The impact on the code sequence generated for the function returns is the same. Compilers usually generate a single jump to the return address located in the stack (e.g. x86 `ret` instruction).

To use interprocedural specialization, when a function return jumps to a stub, the index associated to the current $\mathcal{P}_{\text{call}}$ must be passed to the stub (move at line 6). Here the first index is used (constant `0`). It also needs to get the continuation entry point table from the stack (line 7). A `jmp` is then generated to branch to the appropriate entry point. Here one slot of the table is assigned to the context `(unknown)` so no space is reserved for a generic continuation entry point after the table header. An offset of 8 bytes is used because the first slot is located right after the table header (64 bits).

If the compiler knows $\mathcal{I}_{\text{lambda}}$ (i.e. the identity of the continuation) when compiling a return site, no dispatch is inserted.

5 Experiments

LC implements a subset of the R5RS Scheme standard [17]. One of the goals of the compiler is to be simple yet generate efficient code. To reach this goal, it uses a simple and light JIT compiler directly translating Scheme s-expressions to x86 machine code using an extremely lazy compilation design [20] and no intermediate representation such as SSA [10] or TAC [18]. The compiler extends the Gambit Scheme compiler [13]; it uses its frontend, garbage collector, and x86 assembler.

LC uses Basic Block Versioning [21] to generate efficient code extended with interprocedural specialization using the implementation presented in the previous section. The compiler specializes basic blocks according to type information to reduce the number of dynamic type checks, and register allocation to avoid the generation of extra `mov` instructions at join points. However, only type information is interprocedurally propagated by our extensions.

Finally, it is worth mentioning that LC applies versioning on each subexpression of the s-expression instead of at basic block level.

This section presents the results of the experiments obtained using interprocedural code specialization to show its impact on the generated code. The 34 benchmarks used for the experiments are taken from Gambit. Some benchmarks are not used for the experiments because they use R5RS Scheme features not supported by LC. The number of iterations for each benchmark are also taken from Gambit.

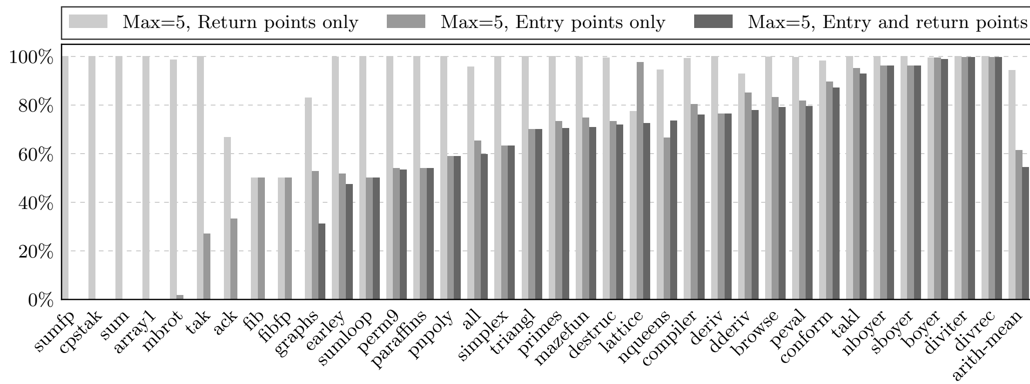

Figure 7 Number of executed type checks relative to pure intraprocedural specialization.

Many of the benchmarks are micro-benchmarks using a limited set of types. Large polymorphic real-world programs might present a challenge for our approach. To simulate these programs, we added the benchmark `all`. This benchmark contains all the others as a single program.

Experiments were conducted on a machine using an Intel Core i7-4870HQ CPU, with 16GB DDR3 RAM running the GNU/Linux operating system. To measure time, each benchmark was executed 10 times, the minimum and maximum values are removed and the average of the 8 remaining values is used.

5.1 Type checks

We consider a type check any code sequence inserted to dynamically retrieve the type of a value. This includes the checks inserted to ensure the safety of the primitives, and the code generated for the type predicate primitives such as `pair?`.

Chevalier-Boisvert and Feeley [8] showed that intraprocedural BBV is effective at removing dynamic type checks for JavaScript. They observed that most JavaScript code is monomorphic or slightly polymorphic and that the number of generated versions rarely exceeds 5. Thus, a hard limit of 5 in the number of generated versions allows the compiler to take advantage of context propagation and to avoid the explosion in the number of versions.

Our interprocedural extensions allow the compiler to collect more information by propagating compilation contexts interprocedurally. Figure 7 shows the number of executed type checks relative to purely intraprocedural BBV.

For each benchmark, the three bars represent executions with propagation enabled through return points only, entry points only and both entry and return points (full interprocedural BBV). For each execution, the maximum number of versions is set to 5.

We see that propagation through return points only has no effect in most cases, but allows the compiler to significantly remove type checks (up to 50%) for 4 of the benchmarks out of 35. On average with return point propagation only, 6% more checks are removed than pure intraprocedural BBV. Propagation through entry points has more impact on the number of removed type checks. Almost all benchmarks are significantly improved and the compiler is able to remove almost all checks for 5 of them. On average, 39% more checks are removed using propagation through entry points.

For 9 of the benchmarks out of 35, almost all checks are removed using full interprocedural propagation. This is explained by the fact that most of these benchmarks use recursive, com-

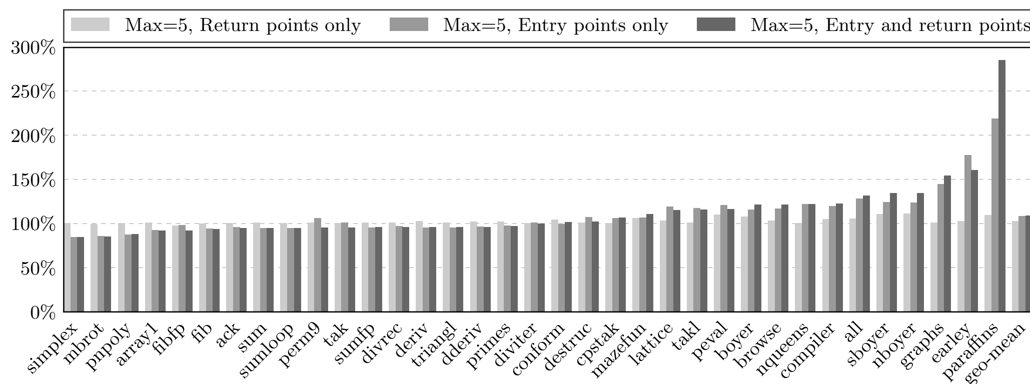

Figure 8 Generated code size relative to pure intraprocedural specialization.

putation intensive, and monomorphic functions in which the computed values and the value of the arguments depend on the result of the previous calls. On average, full interprocedural propagation allows the compiler to remove 46% more checks than intraprocedural BBV.

Our experiments show that the hard limit of 5 specialized versions to avoid code explosion for intraprocedural BBV is valid for Scheme programs. When using our interprocedural extensions, *nqueens* is the only significantly affected benchmark with 15% more checks executed if we limit the number of versions. On average, less than 1% more check are executed when we limit the number of versions. We conclude that this limit is still valid with our interprocedural extensions.

5.2 Generated code size

Figure 8 shows the size of the code generated using interprocedural specialization relative to the size of the code generated using pure intraprocedural specialization. The version limit is set to 5. For each benchmark, the figure shows the impact of propagating information through return points, entry points and both.

Propagation through return points does not significantly affect the size of the generated code.

Using propagation through entry points, we see that there is a slight decrease in the code size for about half of the benchmarks. This is due to the fact that these benchmarks are mostly monomorphic causing the compiler to generate only one version of the code. Because this version is specialized using the context, the compiler generates less code.

For most other benchmarks, we see an increase in the size of the generated machine code. Using full interprocedural specialization, more versions are generated leading to more code.

An interesting effect can be seen for some benchmarks such as *perm9*. For this benchmark, full interprocedural propagation causes a decrease in the generated code size compared to propagation through entry or return points only. This is due to the fact that when information is only propagated through entry or return points, information is lost at some point causing the compiler to insert more dynamic type checks. Using full interprocedural propagation, the compiler keeps the collected information and those checks are removed thus smaller versions are generated.

On average, propagation through return points causes an increase of 3%, propagation through entry points causes an increase of 8.5%, and if both are enabled, an increase of 9% is observed.

Benchmark	Entry point tables (kB)	Benchmark	Return point tables (kB)
all	14097	all	619
compiler	5098	compiler	398
earley	156	peval	32
graphs	101	conform	23
sboyer	88	paraffins	23
conform	88	nboyer	19
nboyer	88	sboyer	18
peval	77	earley	17
paraffins	73	mazefun	17
browse	65	graphs	13
mazefun	60	browse	13
lattice	53	boyer	12
simplex	38	lattice	11
others	< 32	*others*	< 6

Figure 9 Memory used by entry and return point tables to execute each benchmark.

Our experiments show that these Scheme benchmarks do not cause an explosion in the number of versions. Thus, setting a limit of 5 versions does not significantly decrease the size of the generated code. However, this limit ensures that the compiler avoids the potential explosion in the number of versions of highly polymorphic programs.

If there is no limit in the number of versions, the size of the code generated for the benchmark *nqueens* is significantly decreased (11%). For this benchmark, some blocks are generated exactly 6 times meaning that when using no limit, the compiler generates 6 optimized small versions. When the version limit is set to 5, the compiler generates 5 optimized versions and one more generic non-optimized version causing an increase in the code size.

Because some blocks are generated more than 5 times when no limit is set, the size of the code generated for the benchmark *mazefun* is significantly increased (17%).

On average, the size of the generated code is increased less than 1% if we do not limit the number of versions.

5.3 Memory occupied by the tables

As explained in the previous section, interprocedural specialization causes a slight increase in the size of generated code. The compiler must also create and store the entry and return point tables. The table presented in figure 9 shows, for each benchmark, the total memory space occupied by the entry and return point tables. The size is expressed in kilobytes and represents the actual space needed to execute the benchmark (i.e. considering that all the tables have a size that equals the final size of the global layout). It is computed using a limit in the number of versions set to 5.

The benchmarks *all* and *compiler* are the largest benchmarks (respectively ~18KLOC and ~11KLOC). These two benchmarks use about 14MB and 5MB for the entry point tables and 0.6MB and 0.4MB for the return point tables, which is not significant on current systems. Return point tables have a smaller impact because they are all of the same fixed small size.

Entry and return point tables are created at compile time and they live for all the execution. Thus the compiler allocates the tables as permanent objects and they do not

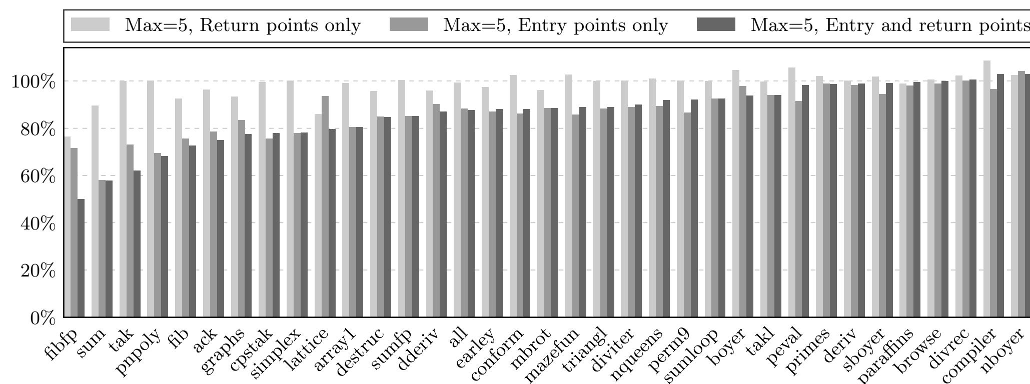

Figure 10 Execution time relative to pure intraprocedural specialization.

affect the garbage collector. We conclude that the tables required by the interprocedural extensions do not use significant memory space.

5.4 Execution time

This section presents execution times only. Compilation and garbage collection time is not taken into account.

Figure 10 presents the execution time using interprocedural specialization relative to the execution time using pure intraprocedural specialization. For each benchmark, we show the result using propagation through return points, propagation through entry points and full interprocedural propagation. The version limit is set to 5.

We see that propagation through return points positively affects a dozen benchmarks (up to 23% faster for *fibfp*). 8 benchmarks are negatively affected (up to 9% slower for *compiler*).

Propagation through entry points impacts execution time more significantly. All of the benchmarks except two are improved (up to 42% for *sum*). *divrec* is not affected and *nboyer* is slowed down by less than 5%.

Finally, when full interprocedural propagation is enabled all of the benchmarks except *divrec*, *compiler* and *nboyer* are improved. *divrec* is not affected and the two others are slowed down by less than 3%.

Propagation through both entry and return points allows the compiler to collect more information and to generate more optimized code. However, the compiler is not able to collect enough information to compensate for the negative impact of return point propagation for the benchmark *compiler* but the slowdown is reduced from 9% to less than 3%.

These results show the potential of interprocedural propagation. For programs using computation intensive recursive functions such as the *fibfp* benchmark, full interprocedural propagation allows the compiler to generate fast optimized code.

For bigger programs using polymorphic functions such as *compiler*, the positive effect of information propagation through return points does not compensate the extra indirection of the return point table. However, propagation through entry points allows the compiler to generate faster code for these benchmarks.

Finally, it is worth mentioning that we didn't identify a use case in which it is significantly better to propagate information through return points only.

Figure 11 shows the execution time relative to the code produced by the Gambit Scheme compiler configured in unsafe mode which is representative of a fully statically type checked

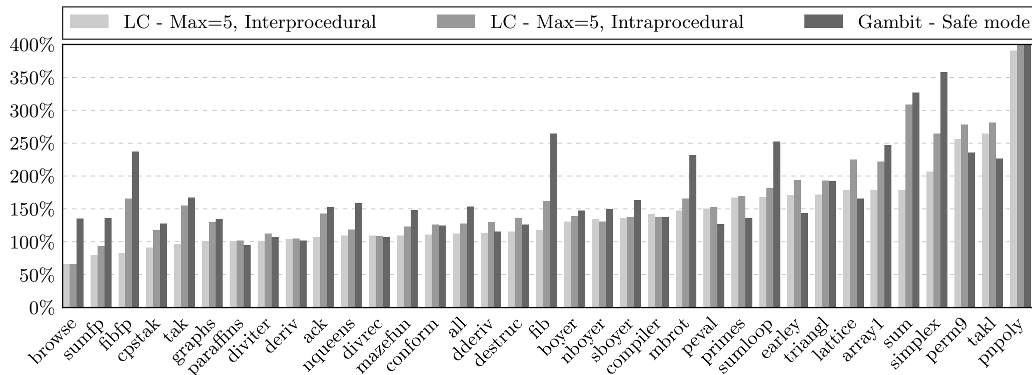

Figure 11 Execution time relative to the execution time of the code generated by the Gambit Scheme compiler configured in unsafe mode (capped at 400%).

situation. When configured in unsafe mode, Gambit does not insert run time type checks, overflow checks and index checks. For each benchmark, we show the execution time of the code generated by LC using full interprocedural propagation and a limit set to 5, LC using pure intraprocedural propagation and a limit set to 5 and Gambit configured in safe mode (i.e. the compiler inserts all the required run time checks and does not use BBV).

As expected, we see that full interprocedural propagation allows LC to significantly improve the execution time of the benchmarks that were greatly and positively affected regarding the number of type checks. For example, the code generated for *sumfp, fibfp* or *fib* using interprocedural propagation is significantly faster than the code generated by Gambit in safe mode.

Our approach allows generating code that is often as efficient as the code generated by Gambit in unsafe mode, and more efficient in some cases. This is in part due to the fact that LC does not use trampolines to implement tail call optimization.

The generated code ranges from 34% faster than the code generated by Gambit in unsafe mode for *browse*, to 390% slower for *pnpoly*. However, the code generated by Gambit for *pnpoly* in safe mode is 646% slower than Gambit in unsafe mode.

Theses experiments show that interprocedural specialization allows LC to generate code that competes with the code generated by current Ahead-Of-Time efficient Scheme implementations and, more generally, to write efficient interprocedurally optimizing compilers for languages supporting closures using relatively simple JIT compilation techniques.

5.5 Compilation time

Figure 12 shows the compilation time using interprocedural propagation relative to the compilation time using pure intraprocedural propagation. The version limit is set to 5.

Propagation through return points does not significantly impact the size of the generated code. As expected, it does not significantly impact the compilation time either.

Propagation through entry points impacts the generated code size more significantly. As expected, it also impacts the compilation time more significantly.

We see that *paraffins* is the most affected benchmark with 431% (70ms using intraprocedural propagation, and 302ms using interprocedural propagation).

The benchmarks *compiler* and *all* are the two largest benchmarks thus closer to real-world programs. *compiler* is 150% slower to compile (1573ms using interprocedural propagation

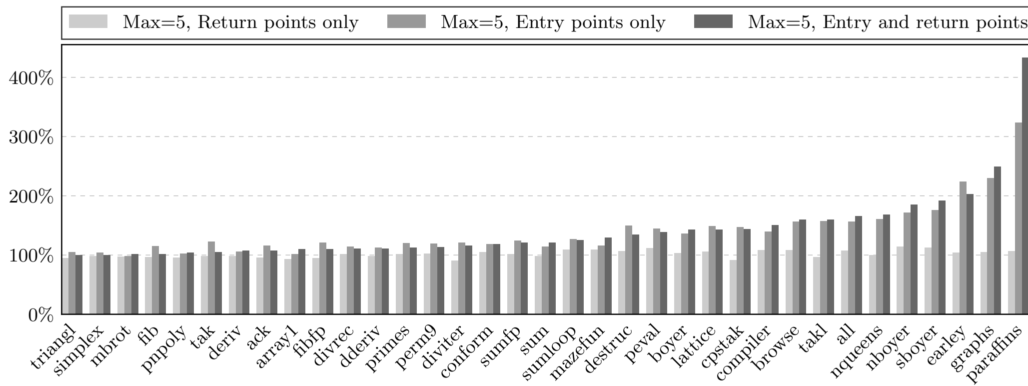

Figure 12 Compilation time relative to pure intraprocedural BBV.

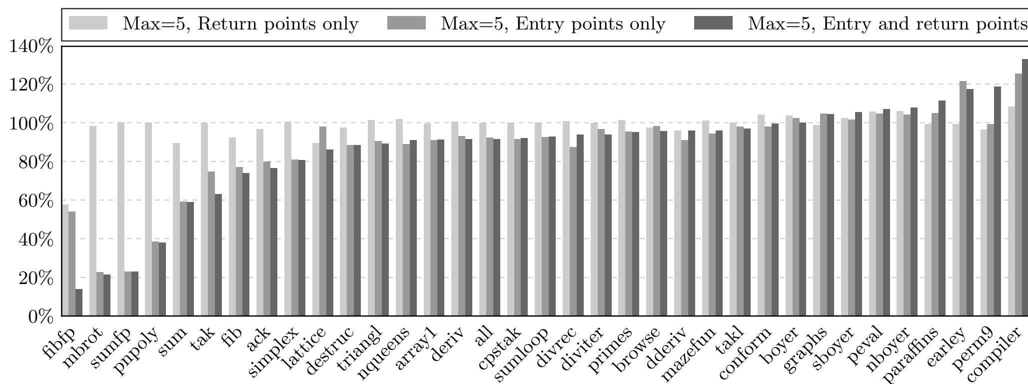

Figure 13 Total time relative to pure intraprocedural BBV.

and 2367ms using intraprocedural propagation). *all* is 166% slower to compile (2809ms using interprocedural propagation and 4657ms using intraprocedural propagation).

5.6 Total time

Figure 13 shows the total time using interprocedural propagation relative to the total time using pure intraprocedural propagation. This time is the total time spent to execute each benchmark including compilation, execution and garbage collection time. The version limit is set to 5.

When using propagation through return points, the execution time varies from 58% (*fibfp*) to 108% (*compiler*) of the total time required by intraprocedural specialization. Using propagation through entry points, the time varies from 23% (*mbrot*) to 125% (*compiler*). When the compiler uses full interprocedural propagation, the time varies from 14% (*fibfp*) to 133% (*compiler*).

The significant speedup observed for the benchmarks *fibfp*, *mbrot*, *sumfp* and *pnpoly* is due to the fact that they intensively use floating point arithmetic. Because the compiler is able to propagate the type of the values, it directly stores untagged double precision floating point numbers in registers. This allows avoiding the box allocations, the execution of boxing and unboxing code, tagging and untagging code, type checks, and data moves between

general purpose registers and floating point registers resulting in a significant decrease in the execution and garbage collection time.

Our JIT compiler significantly decreases the time required to execute Scheme programs both for micro benchmarks and real-world, more polymorphic programs such as the *all* benchmark. However, these results show that the technique may slow down the execution of programs, such as the *compiler* benchmark, using a more imperative style and fewer hot spots.

6 Related work

6.1 Interprocedural BBV

The work done by Chevalier-Boisvert and Feeley [9] is closely related to our own. They presented an extension of Basic Block Versioning enabling interprocedural code specialization based on function identity collection. That extension uses intraprocedural BBV and adds function identities to the propagated contexts. The identities are then propagated through the stubs and possibly known when compiling a call site.

One difference with our work is that their extension enables interprocedural specialization only if the identity of the callee is successfully propagated to the call sites whereas our technique can be used at every call site. Furthermore, because this BBV extension propagates function identities, more versions are generated especially for polymorphic call sites.

However, that approach can be combined with ours to propagate the function identities when possible to avoid the run time dispatch and use our dispatch when identities cannot be determined.

6.2 Inline Caching

Inline Caching (IC) is a technique used to efficiently implement dynamic languages first used in the Smalltalk-80 System [11] and the Self language [6]. IC can be used to implement the dynamic dispatch required by our interprocedural extensions. When a call site is executed for the first time, a dynamic lookup is executed. The compiler stores the result of the lookup at the call site and uses it to directly branch to the specialized version. A guard is inserted to check if the identity is the same for subsequent calls.

Polymorphic Inline Caching (PIC) [16] is an extension of Inline Caching storing several lookup results at call sites. A sequence of checks is inserted at call sites to dispatch according to the currently used function. Each time the call is executed using a new callee, a check is added to the sequence. Because PIC reveals the identity of the callee function, it can be used to branch to a specialized version of the callee [5].

Because of the check sequence, PIC has a bigger impact when used in polymorphic call sites. Furthermore, PIC is not suitable to be used for function return dispatch. Indeed, the continuation used at a given return site frequently varies which may cause the generation of several checks in the sequence.

6.3 Static analysis

Compilers usually use static analysis to determine the identity of the functions called at each call site. In particular, the k-CFA [22] family of algorithms has been popular to implement higher-order languages. The problem is that these analyses have high complexity (cubic in the case of 0-CFA [19]) making them not suitable to use in a JIT compiler. Furthermore,

these analyses are imprecise (the identity of the callee cannot be determined for all sites) limiting the reach of interprocedural specialization.

6.4 Tracing compilation

Tracing JIT compilation, first implemented in Dynamo [1] then adapted to JIT compilation of high-level languages in HotpathVM [15], is a compilation approach used to optimize the executed program at run time by optimizing frequently executed loops. This compilation approach has been used to remove type checks and to reduce run time work [14, 23, 3]. Tracing and Meta-Tracing [4] are often used to efficiently implement dynamic languages such as Javascript [14] and Scheme [2]. When a frequently executed loop is detected (profiling phase), the executed operations are recorded (tracing phase) following function calls. A specialized code sequence (a trace) is then generated and used for the next executions. A guard is inserted at each site where execution can diverge from the recorded path.

Because the tracing phase follows function calls, the function bodies recorded during that phase are inlined in the trace.

One difference with our work is that tracing compilation is based on function identities. When a trace is executed and other functions than those recorded are executed, the guards fail and the execution of the generated code is aborted. A tracing JIT also requires the use of a complex architecture with multiple phases, and the use of an interpreter in addition to a compiler.

7 Future work

The technique presented in this paper allows the compiler to propagate information collected during execution of the program through entry and return points. We showed that results are positive when propagating only type information. Future work includes extending the propagated context to include other information. The compiler could interprocedurally propagate register allocation information (i.e. the registers assigned to the actual parameters and the returned value) to avoid the generation of the extra instructions needed to satisfy the runtime calling convention. The compiler could also interprocedurally propagate the constants and the function identities to do interprocedural lazy aggressive inlining and lazy constant propagation in the presence of higher order functions.

8 Conclusion

This paper presents an approach allowing the compiler to interprocedurally specialize the code using information available at call sites, return points and when creating function closures. The approach is based on dynamic dispatch instead of trying to discover the function identities.

We showed that interprocedural specialization does not require static analysis or complex compiler architecture. Moreover, our approach can be used in the presence of higher order functions using an extended flat closure representation.

When applied to typing, the interprocedural specialization is effective at removing dynamic type checks. The compiler is able to remove almost all the checks for several benchmarks. There is a slight increase in the size of the generated code because more specialized versions of the code are generated. Because more code is generated, there is an increase in the compilation time. Our experiments using LC, a JIT compiler for Scheme, show the approach has a positive impact on the execution time. The code generated for the benchmarks is

executed up to 50% faster than the code generated by pure intraprocedural type specialization. These results indicate that a simple JIT compiler using interprocedural specialization, no complex representation, no complex register allocation algorithm, a simple compilation approach, and minimal static analysis can generate code competitive in performance with current Scheme implementations. This makes the approach a perfect candidate for baseline compilers to implement dynamic languages.

In addition to generating faster code when applied to typing, Our approach can be used to specialize the code using more than type information, opening up new applications such as register allocation based specialization and aggressive lazy inlining.

References

1. Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2000*, pages 1–12, 2000.
2. Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. Pycket: a tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 22–34, 2015.
3. Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pages 43–52, 2011.
4. Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. 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*, pages 18–25, 2009.
5. Solomon Boulos and Jeremy Sugerman. Optimized execution of dynamic languages, January 26 2016. US Patent 9,244,665.
6. Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of Self a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the 1989 Conference on Object-oriented Programming Systems, Languages and Applications*, OOPSLA 1989, pages 49–70, 1989.
7. Maxime Chevalier-Boisvert. *On the fly type specialization without type analysis*. PhD thesis, Université de Montréal, 2015.
8. Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*, pages 101–123, 2015.
9. Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural type specialization of JavaScript programs without type analysis. In *30th European Conference on Object-Oriented Programming, ECOOP 2016*, pages 7:1–7:24, 2016.
10. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth 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.
11. L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL 1984, pages 297–302, 1984.
12. R. Kent Dybvig. *Three Implementation Models for Scheme.* PhD thesis, University of North Carolina at Chapel Hill, 1987.

13 Marc Feeley. Gambit Scheme compiler v4.8.7, January 2017. URL: `http://gambitscheme.org/`.

14 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. 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*, pages 465–478, 2009.

15 Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE 2006*, pages 144–153, 2006.

16 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP 1991, pages 21–38, 1991.

17 Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

18 Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

19 Matthew Might. *Environment analysis of higher-order languages*. PhD thesis, Georgia Institute of Technology, 2007.

20 Baptiste Saleil and Marc Feeley. Code versioning and extremely lazy compilation of Scheme. In *Scheme and Functional Programming Workshop*, 2014.

21 Baptiste Saleil and Marc Feeley. Type check removal using lazy interprocedural code versioning. In *Scheme and Functional Programming Workshop*, 2015.

22 Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

23 Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza A. S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 2–21, 2011.

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