STRAF: A Scala Framework for Experiments in Trace-Based JIT Compilation

Maarten Vandercammen, Quentin Stiévenart,
Wolfgang De Meuter, and Coen De Roover

Software Languages Lab, Vrije Universiteit Brussel, Belgium
{mvdcamme,qstieven,wdmeuter,cderoove}@vub.ac.be

Abstract. We introduce STRAF, a Scala framework for recording and optimizing execution traces of an interpreter it is composed with. For interpreters that satisfy the requirements detailed in this paper, this composition requires but a small effort from the implementer to result in a trace-based JIT compiler. We describe the framework, and illustrate its composition with a Scheme interpreter that satisfies the aforementioned requirements. We benchmark the resulting trace-based JIT compiler on a set of Scheme programs. Finally, we implement an optimization to demonstrate that STRAF enables further experimentation in the domain.

Keywords: tracing compilation, JIT compilation, execution traces, Scala

1 Introduction

Trace-based just-in-time (JIT) compilers do not compile an entire program upfront, but rather start by interpreting the program and identifying its frequently executed loops at run-time. Instructions of these loops are recorded into a *trace*. Once one iteration of such a loop has been traced, the compiler compiles and optimizes the resulting trace. Subsequent iterations will execute the optimized trace rather than interpret the original loop, resulting in speed-ups.

Most trace-based JIT compilers are constructed specifically for one particular programming language. As a result, few efforts are shared between implementations. The RPython framework for implementing trace-based JIT compilers [1] addresses this problem. Its runtime is capable of tracing various interpreters. However, while RPython enables constructing performant language runtimes, its focus on maximizing performance may hinder its comprehensibility and adaptability. In contrast to RPython our framework does not focus on performance. Instead, it aims at being minimalistic, comprehensible, and extensible. This way, our framework should facilitate further experimentation in the domain of trace recording and optimization.

This paper reports on the integration of our earlier ideas [12] in SCALA-AM [10], a framework for implementing interpreters from abstract machine formalizations and using these abstract machines as static program analyzers. We call the resulting Scala framework for developing trace-based JIT compilers

STRAF. Its integration into the SCALA-AM static analysis framework specifically, though not uniquely, enables experimenting with employing static analysis to improve optimization of traces. The complete implementation of STRAF is available at `https://github.com/mvdcamme/scala-am`.

2 Trace-based JIT Compilation

Trace-based JIT compilers build on two assumptions: most of a program's execution time is spent in loops, and several iterations of the same loop are likely to take the same path through the program [1]. They therefore optimize frequently executed loops, whereas method-based JIT compilers optimize methods only.

Trace-based JIT compilers are generally conceived as a mixed-mode execution involving an interpreter and a compiler. The interpreter executes the program and simultaneously profiles loops to identify the frequently executed ones. When a "hot" loop is detected, the interpreter starts *tracing* its execution: every operation performed by the interpreter is recorded. Tracing continues until a full loop iteration is complete. The compiler optimizes the recorded trace next. Subsequent iterations of the loop then execute the compiled trace directly. Conditions that held when a trace was recorded might no longer hold when the trace is executed. Trace-based JIT compilers therefore add *guards* to the trace to verify these conditions. When a guard fails, trace execution is aborted and regular interpretation of the program is resumed.

Figure 1 depicts a Scheme function `fact` that computes the factorial of 5. The recursive calls implement a loop that, when recorded, results in the trace of operations depicted on the right. If the condition `(= n 0)` evaluated to `false` while recording, the trace will feature a guard `ActionGuardFalse` verifying that this condition still evaluates to `false` when the trace is executed. As such, the trace corresponds to the operations performed by the interpreter in the `false`-branch of the `if`-expression. Should this guard fail at run-time, trace execution is aborted and interpretation will resume from the other branch.

```
(letrec ((fact (lambda (n)
                 (if (= n 0)
                     1
                     (* n (fact (- n 1)))))))
  (fact 5))
```

```
...
ActionEvalPush("=", FrameFunCallFunction(List("n", 0)))
ActionLookupVar("=")
ActionPushValue
ActionPopKont
ActionEvalPush("n", FrameFunCallFunction(List(0)))
ActionLookupVar("n")
ActionPushValue
ActionPopKont
ActionEvalPush(0, FrameFunCallArgs(List()))
ActionLiteralValue(0)
...
ActionGuardFalse(...)
ActionEvalTraced((* n (fact (- n 1))))
...
```

Fig. 1: A Scheme program containing a loop and part of the corresponding trace.

3 The STRAF framework for building JIT compilers

STRAF decouples tracing mechanisms from language semantics through a fixed *tracing machine* (or tracer) that can be composed with a developer-provided *abstract machine* (or interpreter) [5]. The abstract machine handles regular program execution while the tracing machine is responsible for trace recording and execution. Any abstract machine can be used, on the condition that it satisfies the requirements outlined in Section 3.1. Section 3.2 describes an example of such an abstract machine for the Scheme programming language. The tracing machine of our framework is detailed in Section 3.3.

3.1 Requirements on the Abstract Machine

Program State The interpreter must be conceived as an abstract machine that transitions between program states. This way, the tracing machine can easily resume program evaluation from a particular state. Instructions recorded into a trace then correspond to state transitions. Interpreters modeled after the ubiquitous CESK machine [6] trivially satisfy this requirement. However, we do not impose any constraints on the actual state representation used by the abstract machine.

Tracing Signals The tracing machine is to record "hot" loops, but their form is language-specific. For instance, loops are typically implemented using recursive functions in Scheme. For STRAF to remain language-agnostic, the abstract machine it is composed with must signal when it has started one loop iteration by using a `SignalStart` instance. It must also label each loop-expression in the program. This enables the tracing machine to associate traces with loops.

Guards Traces include guard instructions verifying that their control flow remains valid for a later execution. These too have to be provided by the abstract machine STRAF is composed with. Guards need to provide a *restart point* from which the abstract machine can resume interpretation when the guard fails during trace execution. No other constraints are imposed on their implementation.

Hooks Finally, the abstract machine must provide the following functions to the tracing machine:

- A function `step` which, given a program state, returns a `Step` instance encapsulating the actions to be applied on this state. A `Step` can also include a `SignalStart`.
- An `applyActions` function which consecutively applies the actions from a `Step` to a given program state, and returns the new resulting program state.
- A `restart` function which takes a program state and the *restart point* of a failed guard as input and returns a new program state.

– An `optimize` function which, given a previously recorded trace and the program state observed at the start of the recording of this trace, returns an optimized version of this trace. Implementing this function in the abstract machine ensures that STRAF itself remains language-agnostic.

3.2 A CESK-based Interpreter for Scheme

Section 4 will compose STRAF with a Scheme interpreter to evaluate the resulting trace-based JIT compiler. Being modeled after a CESK-machine [6], this interpreter trivially satisfies the requirements of Section 3.1. Listing 1 defines its representation of program states. Their first component `control` is either an expression to be evaluated or a continuation frame to be followed. In addition, their environment component `env` maps variables to addresses and their store component `sto` maps these addresses to Scheme values. The remaining components are a continuation stack `kstack`, a value register `v` containing the value of the last expression that was evaluated, and a value stack `vstack` which is used to save lexical environments and argument values while evaluating a function call.

```
type Storable = Either[Value, Environment]
case class Step(actions: List[Action], signal: Signal)
case class ProgramState(control: Control, env: Environment, sto: Store,
                        kstack: Stack[Frame], v: Value, vStack: Stack[Storable])
```

Listing 1: Representation of program states.

To evaluate a composite expression, the interpreter pushes a specific *continuation frame* onto the continuation stack before evaluating its subexpressions. This frame is later popped and continued with when the interpreter has finished evaluating the subexpressions. States corresponding to the latter case feature the popped continuation frame as their `control` component instead of an expression. For those states, the interpreter's `step` function (cf. the hooks defined above) applies a function `stepKont` on the continuation frame and the contents of the value register `v`.

```
def stepEval(exp: SchemeExp): Step = exp match {
  case SchemeVarRef(varName) =>
       Step(List(ActionLookupVar(varName), ActionPopKont),
            SignalFalse)
  case SchemeFuncall(function, args) =>
       Step(List(..., ActionEvalPush(function, FrameFunCallFunction(args)))),
            SignalFalse)
  ...
}
```

Listing 2: Handling function application and variable lookup in `stepEval`.

stepEval Listing 2 illustrates how function `stepEval` evaluates atomic expressions such as variable references and composite expressions such as function calls. For variable references, the interpreter returns a list of actions `ActionLookupVar` and `ActionPopKont` which respectively perform the variable lookup, placing the

resulting value in `v`, and pop the topmost continuation from the stack. For a function call, `stepEval` returns an `ActionEvalPush` among its actions which pushes a `FrameFunCallFunction` onto the continuation stack before proceeding to evaluate the `function` subexpression. The pushed continuation encapsulates the function arguments that need to be evaluated next. For neither expression a loop is entered. This is communicated to the tracer using the `SignalFalse` argument to `Step`.

stepKont Listing 3 details how `stepKont` handles the remainder of function call evaluation. `stepKont` takes as input the value `v` that was just computed and the `frame` that was popped from the continuation stack. The former either corresponds to the invoked procedure (i.e., for `FrameFunCallFunction` frames) or to one of the argument values (i.e., for `FrameFunCallArg` frames). Function `evalFunctionCall` is delegated to in either case. If no more arguments remain to be evaluated, evaluation proceeds to the body of the called procedure (`ActionStepIn`). Otherwise, the newly computed value is saved on the value stack (`ActionPushVal`) and evaluation proceeds to a new argument by pushing `FrameFunCallArgs` onto the continuation stack. As loops are typically implemented through recursion in Scheme, any call can potentially start a loop. The interpreter therefore sends a `SignalStart` whenever a procedure is stepped into. The body of the invoked procedure functions as loop label.

```
def evalFunctionCall(functionValue: FunctionValue, args: List[SchemeExp]): Step = args match {
   case Nil =>
      Step(List(... , ActionStepIn),
           SignalStart(functionValue.body))
   case e :: rest =>
      Step(List(... , ActionPushVal, ActionPushTraced(e, FrameFunCallArgs(functionValue, rest))),
           SignalFalse))
}

def stepKont(v: Value, frame: Frame): Step = frame match {
   case FrameFunCallFunction(args) =>
      evalFunctionCall(v, args)
   case FrameFunCallArg(functionValue, args) =>
      evalFunctionCall(functionValue, args)
   ...
}
```

Listing 3: Continuation of function call evaluation in `stepKont`.

Applying Actions Listing 4 illustrates how the `applyActions` hook applies a single action to a given program state. In the case of an `ActionEvalPush`, the interpreter retrieves the expression `exp` to be evaluated and places a corresponding `ControlExp` in the control component of the program state. The given frame is also pushed onto the continuation stack.

Guards Listing 5 illustrates how the interpreter communicates guard instruction to the tracer for `(if pred cons alt)` expressions. An `ActionGuardTrue`

```
def applyAction(state: ProgramState, action: Action): ProgramState = action match {
   case ActionEvalPush(exp, frame) =>
        state.copy(control = ControlExp(e), kstack = state.kstack.push(frame))
   ...
}
```

Listing 4: Applying a single `ActionPush` action to a given program state.

with a restart point that refers to the `alt`-expression is emitted when `pred` evaluates to `true`. Like other actions, this guard will be executed by the `applyAction` function. Should it then find that the expression last evaluated (i.e., `pred`) did *not* evaluate to `true`, the function calls `restart` with the current program state and the restart point encapsulated in the guard. As depicted in the same listing, `restart` only has to replace the old control component of the state by this restart point.

```
case class ActionGuardTrue(restart: RestartPoint) extends Action
case class RestartGuardIfFailed(exp: SchemeExp) extends RestartPoint

def stepKont(v: Value, frame: Frame, sto: Store): Step = frame match {
   case FrameIf(cons, alt) =>
      if (v.isTrue()) {
           Step(List(ActionGuardTrue(RestartGuardIfFailed(alt)) ...),
                SignalFalse) }
      else { ... }
   ...
}

def restart(state: ProgramState, restart: RestartPoint): ProgramState = restart match {
   case RestartGuardIfFailed(exp) =>
      state.copy(control = ControlExp(exp))
   ...
}
```

Listing 5: Emitting guards for an `if`-expression.

3.3 Tracing Machine

The tracing machine controls the mixed-mode execution of the program. Figure 2 depicts the transitions between its three modes: 1) *normal interpretation*, in which the interpreter executes the program without interference from the tracing machine; 2) *trace recording*, in which the tracer records all actions undertaken by the interpreter; and 3) *trace execution* in which the tracing machine executes a previously recorded trace.

Normal Interpretation In this mode, the tracer repeatedly asks the abstract machine to perform a single interpretation step. The tracer updates the current program state by applying the actions returned by the interpreter. If these actions do not include a tracing signal, the tracer continues running in normal interpretation mode. Upon encountering a `SignalStart`, the tracing machine either starts recording a new trace for unseen loops or starts executing a previously recorded trace for seen loops. Note that, in contrast to the basic scheme described here, STRAF does wait for a loop to become hot before tracing it, by counting how many times a `SignalStart` was sent for a particular procedure, and tracing it once a threshold has been reached.

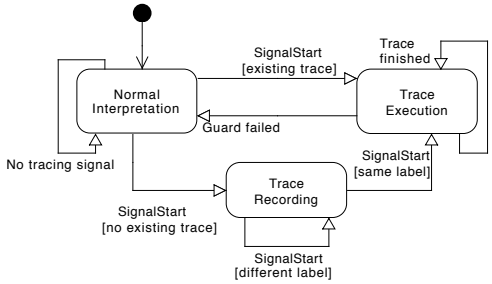

Fig. 2: Transitions between execution modes of the tracing machine.

Trace Recording This mode is similar to the previous one, but the tracing machine records all actions communicated by the interpreter into a trace. The tracer keeps recording these actions until the interpreter sends a new `SignalStart` *for the loop currently being recorded*, as this indicates that one full iteration of the loop has been completed. The recorded trace is then optimized via the `optimize` hook of the interpreter and subsequently stored. Note that if the interpreter executes an inner loop in the loop being traced, and therefore sends another `SignalStart` for that inner loop, this entire loop will be unrolled in the trace.

Trace Execution A previously recorded trace is executed by consecutively applying the actions it is composed of. When the end of the trace has been reached, the tracer loops back to its beginning. At some point, a guard will fail and execution of the trace will be aborted. The interpreter's `restart` hook is then called with the restart point of the guard that failed and the current program state. Execution of the program is resumed under the normal interpretation mode with the program state that is returned.

4 Evaluation

STRAF does not aim for top performance, but strives to facilitate experimentation in the domain through the qualities of minimalism, comprehensibility and extensibility. Our evaluation therefore focuses on whether it is possible to easily extend the framework, e.g., with new trace optimizations or tracing mechanisms. To this end, we compose STRAF with the Scheme interpreter described in Section 3.2 and implement several optimizations as well as extensions to the previously described tracing mechanism. Section 4.1 gives a high-level overview of some trace optimizations and extensions to STRAF's tracing mechanism. Section 4.2 describes and evaluates one of these optimizations in detail.

4.1 Extensions to STRAF

We have designed and implemented several trace optimizations, including a constant folding [4], a type specialization [2] and a variable folding optimization.

These optimizations together span around 400 lines of code[1]. Additionally, we have also extended STRAF with a hot loop detection and a guard tracing mechanism. The former enables STRAF to detect hot loops by counting the number of `SignalStart`s sent for each procedure and only tracing procedures for which the number of `SignalStart`s that were sent has crossed some threshold. The latter makes it possible to not only trace procedures, but also to start tracing from the point of a guard failure. When the guard fails again at some later point in the execution, execution jumps to the trace that was recorded for this guard, instead of resuming normal interpretation. This reduces the performance penalty incurred for a guard failure, as execution can jump from one optimized trace to another instead of returning to normal interpretation. These two additional mechanisms were completed in only 100 lines of code[2].

4.2 Continuation Stack Optimization

We now describe and evaluate an additional trace optimization, the *continuation stack optimization*, which eliminates all pairs of actions from a trace that push and pop a continuation frame. This is sound because continuation frames only affect the control flow, which is fixed for a particular trace. Care must be taken, however, that no guard instruction is located between these actions. The continuation stack *must* be kept up-to-date if normal interpretation might be resumed after a guard failure. In practice, applying this continuation stack optimization often reduces the length of a trace by up to 25%. The implementation of this optimization spans about 50 lines of code, and was completed in about an hour's effort. We evaluate this optimization on a set of several programs ranging from just two lines of code to around 240. These stem from the benchmark suite included with the SCALA-AM framework on top of which STRAF is implemented.

Evaluation Figure 3 depicts the effectiveness of the continuation stack optimization. It shows the number of continuation stack operations that are applied throughout the execution of a benchmark as a fraction of the number of continuation stack operations that are applied when the optimization is not applied on the collected traces. This optimization drastically reduces the number of such applications, by up to 95% in some cases.

We also evaluate the optimization in terms of the performance improvement it brings to the compiler. We conducted this evaluation on an Intel I7-4870HQ CPU at 2.50GHz with 6MB cache and 16GB of RAM. The machine ran 64bit OS X 10.11.6 and Scala 2.11.7. Each program was executed thirty times, with each run on a separate JVM; measurements only started after JVM warm-up was completed. Figure 4 shows the median execution times, along with its 95% confidence interval, of the programs when traces were collected and executed,

[1] `https://github.com/mvdcamme/scala-am/blob/master/src/main/scala/tracing/SchemeTraceOptimizer.scala`

[2] `https://github.com/mvdcamme/scala-am/blob/master/src/main/scala/tracing/SchemeTracer.scala`

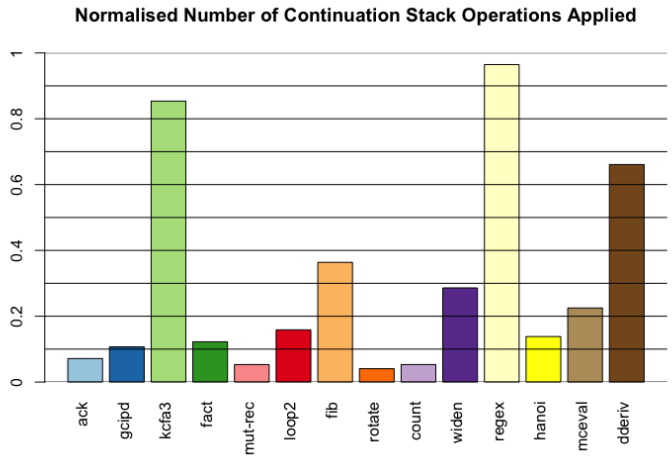

Fig. 3: Number of continuation stack operations executed, normalized with respect to the unoptimized execution.

but not optimized. These numbers serve as the baseline with respect to which the effectiveness of the continuation stack optimization is compared. Figure 5 shows the execution time of these same benchmarks, normalized to the baseline execution time and with the 95% confidence interval included.

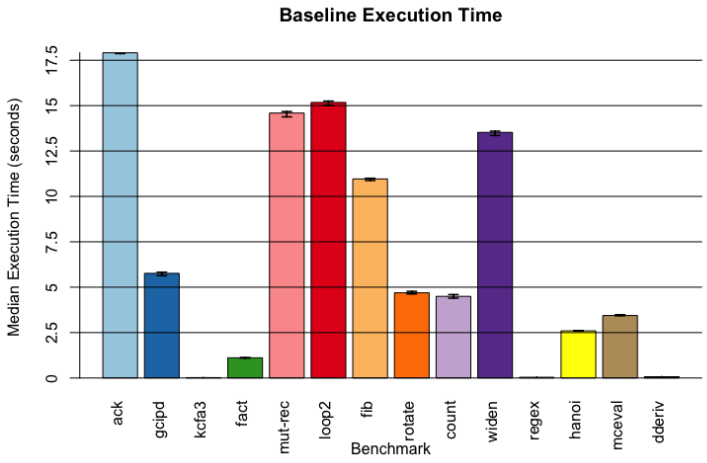

Fig. 4: Median execution time of the benchmarks when traces are not optimized.

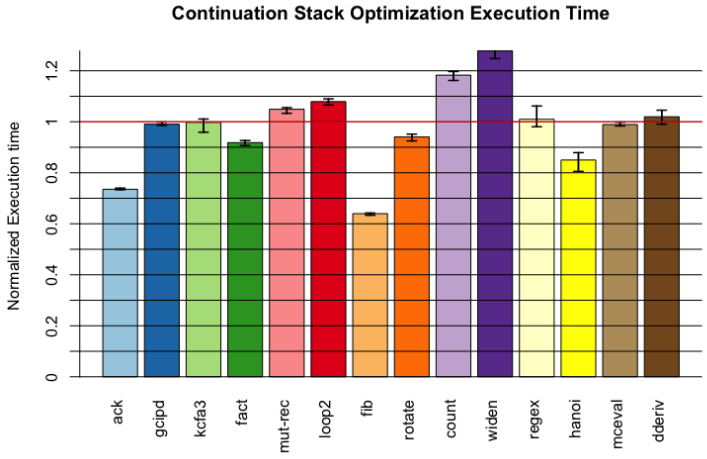

Fig. 5: Median execution time of the benchmarks with just the continuation stack optimization enabled, normalized with respect to the baseline results.

When the continuation stack optimization is applied, performance conclusively improves in 5 out of 14 cases. In the remaining cases, it is likely that the traces are either too short or the hot loop detection mechanism prioritized tracing a loop which was afterwards not executed often enough. In both cases, the overhead of tracing and optimizing negates any improvement made by the optimization.

5 Related work

We reported an earlier version of the core principles behind the separation of the tracer and the interpreter in prior work [12]. That work relied on a formalization and a Scheme implementation of the framework. This paper extends that work by transposing the described ideas to the SCALA-AM framework [10], by specifying its implementation instead of offering a formal model, by describing optimization strategies for traces and by evaluating STRAF via a set of benchmarks detailing its performance.

Several widely-used trace-based JIT compilers have been deployed, such as HotPath [8], TraceMonkey [7], Tamarin-Tracing [3]. However, these compilers all execute one particular language and cannot be composed with a variety of different interpreters.

The RPython framework is a meta-compilation framework that applies the technique of *meta-tracing* [1]: instead of tracing the execution of a program directly, a meta-tracer traces the execution of an interpreter *while this interpreter executes the program*. Similarly to STRAF, RPython thus enables language implementers to provide a regular interpreter, annotated with certain hints to guide tracing and optimization, to benefit from the advantages of trace-based

compilation without having to construct a dedicated JIT compiler for the language. RPython greatly reduces the engineering effort required by language implementers and is also successful in lifting the performance of the meta-traced interpreter to the same order of magnitude than a dedicated JIT compiler [9]. However, the complexity of RPython and its focus on performance makes it less suited for experimenting with novel trace recording or trace optimization strategies. In contrast, STRAF focuses exclusively on providing a minimalistic yet extensible framework that facilitates studying of and experimenting with trace-based compilation strategies.

6 Conclusion and Future Work

We have introduced the STRAF framework for recording and optimizing execution traces of an interpreter it is composed with. For interpreters that satisfy a limited set of requirements, this composition results in a trace-based JIT compiler. STRAF does not aim to generate trace-based JIT compilers that outperform existing ones, but to facilitate further experimentation with trace recording and trace optimization. To this end, it achieves low coupling between tracing mechanisms and language semantics.

STRAF is the embodiment of our earlier ideas on JIT compilation [12] into the SCALA-AM framework [10] for implementing interpreters as abstract machines and for deriving static analyses from these interpreters. We are currently investigating whether trace-based JIT compilation can benefit from whole-program static analysis, by providing information about the program that lies beyond the boundaries of the trace. We have recently described an approach [11] for using a whole-program static analysis to find constant variables in a program and using this information to improve trace optimization: if the compiler knows that a variable will remain constant throughout the program's execution, it can replace a lookup of the variable in the trace by its value as it was observed during trace recording. Our approach enables detecting more constants than would be found by other trace-based compilers because these compilers only consider the local part of the program that is actually traced and do not look beyond the boundaries of this trace. By integrating STRAF into the SCALA-AM framework, we can construct an abstract machine based interpreter for a language, derive a static analysis from it by using the SCALA-AM framework and couple the interpreter to the tracing machine. Using the same abstract machine for both functions makes it possible to easily alternate between these functions, enabling us to perform static analysis over parts of the program *at run time*. This in turn increases precision of the static analysis, as we can include observed runtime values in the program analysis instead of having to predict these statically. The minimalistic but extensible implementation of STRAF facilitates these kinds of experiments in hybrid trace optimizations.

References

1. Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the meta-level: Pypy's tracing jit compiler. In: Proc. of the 4th ICOOOLPS Workshop (2009)
2. Chang, M., Bebenita, M., Yermolovich, A., Gal, A., Franz, M.: Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Tech. Rep. ICS-TR-07-10, University of Irvine, Department of Computer Science (2007)
3. Chang, M., Smith, E., Reitmaier, R., Bebenita, M., Gal, A., Wimmer, C., Eich, B., Franz, M.: Tracing for web 3.0: Trace compilation for the next generation web applications. In: Proc. of the 2009 ACM SIGPLAN/SIGOPS Int. VEE Conf. (2009)
4. Corporation, N.: Constant folding. `http://www.compileroptimizations.com/category/constant_folding.htm`, accessed: 2016-05-24
5. Felleisen, M., Friedman, D.P.: Control Operators, the SECD-machine, and the λ-calculus. Indiana University, Computer Science Department (1986)
6. Felleisen, M., Friedman, D.P.: A calculus for assignments in higher-order languages. In: Proc. of the 14th ACM SIGACT-SIGPLAN POPL Symp. (1987)
7. Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M.R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E.W., Reitmaier, R., Bebenita, M., Chang, M., Franz, M.: Trace-based just-in-time type specialization for dynamic languages. In: Proc. of the 30th ACM SIGPLAN PLDI Conf. (2009)
8. Gal, A., Probst, C.W., Franz, M.: Hotpathvm: An effective jit compiler for resource-constrained devices. In: Proc. of the 2nd Int. VEE Conf. (2006)
9. Marr, S., Ducasse, S.: Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In: Proc. of the 2015 ACM Int. OOPSLA Conf. (2015)
10. Stiévenart, Q., Nicolay, J., De Meuter, W., De Roover, C.: Building a modular static analysis framework in scala (tool paper). In: Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala. pp. 105–109. SCALA 2016, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2998392.3001579`
11. Vandercammen, M., De Roover, C.: Improving trace-based jit optimisation using whole-program information. In: Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages. pp. 16–23. VMIL 2016, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2998415.2998418`
12. Vandercammen, M., Nicolay, J., Marr, S., De Koster, J., D'Hondt, T., De Roover, C.: A formal foundation for trace-based jit compilers. In: Proc. of the 13th WODA (2015)