Optimization Coaching [*]

Optimizers Learn to Communicate with Programmers

Vincent St-Amour   Sam Tobin-Hochstadt   Matthias Felleisen
PLT @ Northeastern University
{stamourv,samth,matthias}@ccs.neu.edu

Abstract

Optimizing compilers map programs in high-level languages to high-performance target language code. To most programmers, such a compiler constitutes an impenetrable black box whose inner workings are beyond their understanding. Since programmers often must understand the workings of their compilers to achieve their desired performance goals, they typically resort to various forms of reverse engineering, such as examining compiled code or intermediate forms.

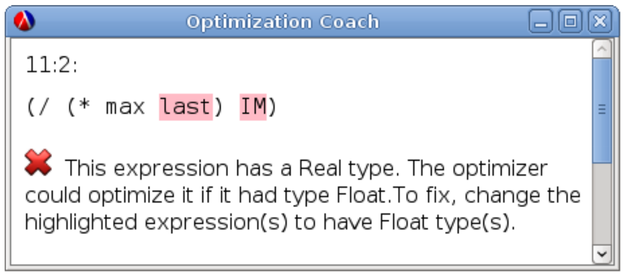
Instead, optimizing compilers should engage programmers in a dialog. This paper introduces one such possible form of dialog: *optimization coaching*. An optimization coach watches while a program is compiled, analyzes the results, generates suggestions for enabling further compiler optimization in the source program, and presents a suitable synthesis of its results to the programmer. We present an evaluation based on case studies, which illustrate how an optimization coach can help programmers achieve optimizations resulting in substantial performance improvements.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Integrated Programming Environments; D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Optimization Coaching, Visualization

1. Compilers: A Dialog with Programmers

With optimizing compilers programmers can create fast executables from high-level code. As Knuth (1971) observed, however, "[p]rogrammers should be strongly influenced by what their *compilers* do; a compiler writer ... may in fact know what is really good for the programmer and would like to steer him towards a proper course. This viewpoint has some merit, although it has often been carried to extremes in which programmers have to work harder and make unnatural constructions just so the compiler writer has an easier job." Sadly, the communication between compilers and programmers has not improved in the intervening 40 years.



```
11:2:
(/ (* max last) IM)
```

This expression has a Real type. The optimizer could optimize it if it had type Float.To fix, change the highlighted expression(s) to have Float type(s).

Figure 1: Our optimization coach in action

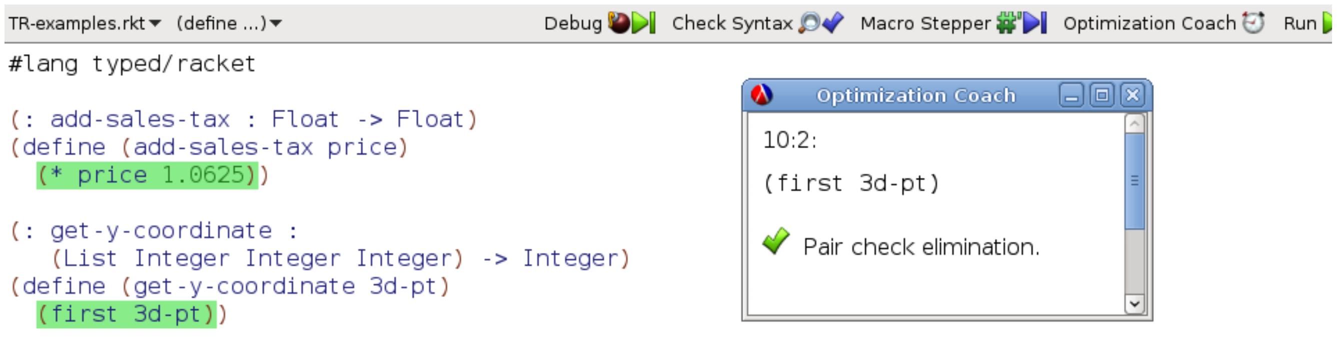
To achieve high quality results, expert programmers learn to reverse-engineer the compiler's approach to optimization from object code generated for some source programs. With an appropriate model of the optimizer, they can then write their programs to take advantage of compiler optimizations. Other programmers remain at the mercy of the compiler, which may or may not optimize their code properly. Worse, if the compiler fails to apply an optimization rule, it fails silently, and the programmer may never know. Sometimes even experts cannot reliably predict the compiler's behavior. For example, during a recent discussion of[1] about the performance of a ray tracer, the authors of the compiler publically disagreed on whether an inlining optimization had been performed, eventually resorting to a disassembling tool.

Currently, programmers seeking maximal optimization from their compilers turn to style guides (Fog 2012; Hagen 2006; Zakas 2010) on how to write programs that play nicely with the optimizer. Naturally, the advice of such guides is limited to generic, program-agnostic advice. They cannot offer programmers targeted advice about individual programs.

In this paper, we propose an alternative solution to this problem. Specifically, we introduce the idea of *optimiza-*

[*] Supported in part by NSF grants, the DARPA CRASH program, a grant from the Mozilla Foundation, and an NSERC scholarship for V. St-Amour.

[1] See Racket bug report `http://bugs.racket-lang.org/old/12518`



```
#lang typed/racket

(: add-sales-tax : Float -> Float)
(define (add-sales-tax price)
  (* price 1.0625))

(: get-y-coordinate :
   (List Integer Integer Integer) -> Integer)
(define (get-y-coordinate 3d-pt)
  (first 3d-pt))
```

```
10:2:
(first 3d-pt)
Pair check elimination.
```

Figure 2: Optimization Coach's analysis of our example functions (with focus on `get-y-coordinate`'s body)

tion coaching and illustrate it with an implementation in the Racket ecosystem (Flatt and PLT 2010). A compiler with an optimization coach "talks back" to the programmer. It explains which optimizations it performs, which optimizations it misses, and suggests changes to the source that should trigger additional optimization. Figure 1 presents our tool, named Optimization Coach, in action. Here the compiler points to a specific expression where the optimizer could improve the performance of the program, along with a particular recommendation for how to achieve the improvement. As the figure shows, a compiler with an optimization coach is no longer a capricious master but a programmer's assistant in search of optimizations.

2. Goals and Overview

An optimization coach engages the programmer in a dialog. The objective is to gather information during compilation, to analyze the information, and to present it in an easily accessible manner. More concretely, an optimization coach should report two kinds of information:

- *successes*: optimizations that the compiler performed on the current program.
- *near misses*: optimizations that the compiler could perform if it had additional information. Since an optimization coach may report many such near misses for a large program, it should also rank changes according to how they may affect the overall status of optimization.

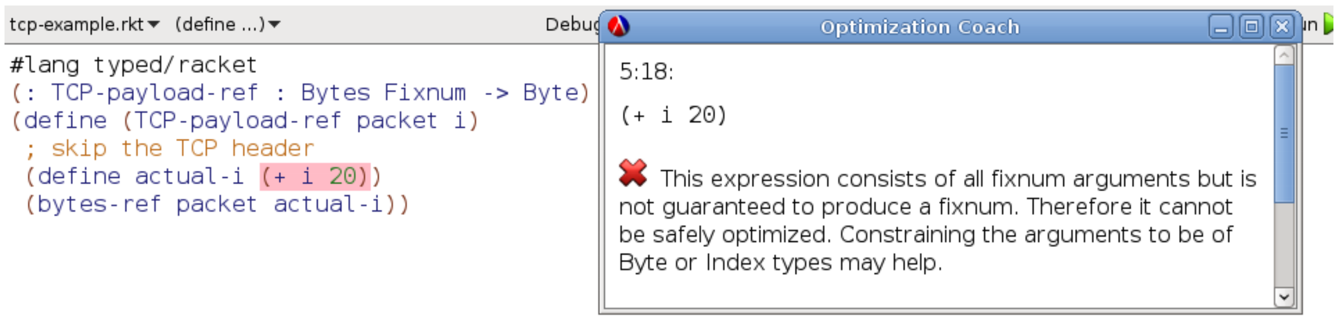
Throughout, the programmer remains in charge; the optimization coach merely provides information and advice. It remains the programmer's responsibility to act on these recommendations.

We propose the following architecture for optimization coaches, consisting of four main phases:

1. An instrumented compiler logs optimization decisions.
2. The optimization coach analyzes the resulting logs to detect successes and near-misses.
3. From the results of the analysis, it generates advice and recommends changes to the program.
4. The optimization coach presents this information in a comprehensible manner, but only on-demand.

Designing and implementing an optimization coach poses challenges at every stage, e.g. how to accurately reconstruct the optimization process, how to avoid the false positives problem, how to present complex optimization information in a comprehensible fashion. In response, we identify several general concepts underlying optimizations, such as *optimization failures*, *optimization proximity* and *irritants*, making it possible to discuss optimization decisions abstractly. Based on these concepts, we present optimization- and compiler-agnostic solutions to the identified challenges.

To validate the usefulness of optimization coaching, we have implemented an optimization coach, named Optimization Coach, that instantiates our framework on two different compilers that reflect two different approaches to optimization. The first, for Typed Racket (Tobin-Hochstadt et al. 2011), handles various local type-driven code specialization optimizations. The second, for the Racket production optimizer, handles inlining optimizations. Typed Racket programs are also Racket programs and Optimization Coach seamlessly composes the results from the two compilers. Optimization coach presents its results via syntax coloring and tool-tips in DrRacket (Findler et al. 2002), an integrated development environment for Racket. We then used this prototype on existing Racket code bases and uncovered optimizations with minimal effort, greatly improving the performance of several benchmarks and programs. The rest of the paper starts with brief sections on the Racket and Typed Racket compilers. The following sections discuss each of the four phases of our proposed architecture by first outlining the challenges faced by optimization coaches during that phase, then explaining our solutions to these challenges, and finally providing concrete examples drawn from our two prototype instantiations. The result portion of the paper ends with an empirical evaluation of the effectiveness of Optimization



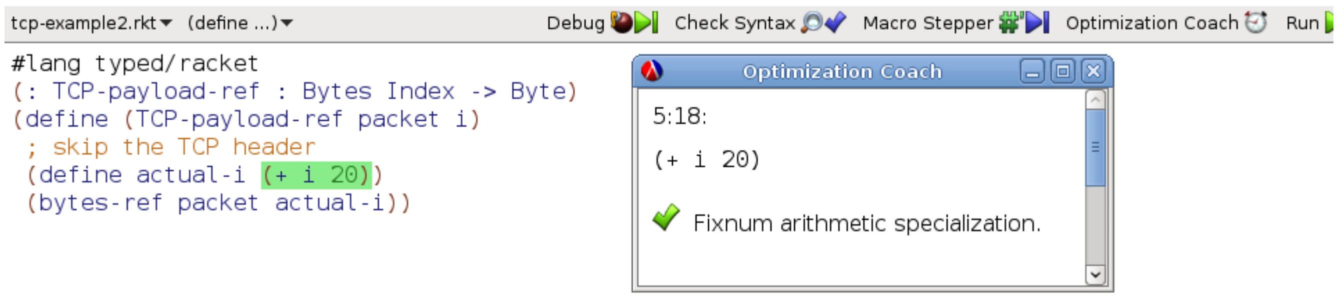
```
#lang typed/racket
(: TCP-payload-ref : Bytes Fixnum -> Byte)
(define (TCP-payload-ref packet i)
 ; skip the TCP header
 (define actual-i (+ i 20))
 (bytes-ref packet actual-i))
```

5:18:

```
(+ i 20)
```

✗ This expression consists of all fixnum arguments but is not guaranteed to produce a fixnum. Therefore it cannot be safely optimized. Constraining the arguments to be of Byte or Index types may help.

Figure 3: Optimization near miss involving fixnum arithmetic



```
#lang typed/racket
(: TCP-payload-ref : Bytes Index -> Byte)
(define (TCP-payload-ref packet i)
 ; skip the TCP header
 (define actual-i (+ i 20))
 (bytes-ref packet actual-i))
```

5:18:

```
(+ i 20)
```

✓ Fixnum arithmetic specialization.

Figure 4: Confirming that the optimization failure is now fixed

Coach. The paper concludes with a comparison with other techniques with goals similar to our own and a discussion of future work.

3. The Typed Racket Compiler

The Typed Racket compiler is a research compiler that compiles Typed Racket programs to Racket programs (Tobin-Hochstadt et al. 2011). It uses core Racket programs as a high-level intermediate representation. This representation is close to actual source programs, and most source-level information is still present. Typed Racket performs source-to-source transformations to implement optimizations. The most important transformations are type-driven code specializations of generic operations. For example, a generic use of the multiplication function can be specialized if both its arguments are floating-point numbers, e.g. in definitions such as this one:

```
(: add-sales-tax : Float -> Float)
(define (add-sales-tax price)
  (* price 1.0625))
```

Since Typed Racket's type system validates that this mutliplication always receives floating-point numbers as its arguments, the compiler may specialize it thus:

```
(: add-sales-tax : Float -> Float)
(define (add-sales-tax price)
  (unsafe-fl* price 1.0625))
```

Similarly, if type information provides guarantees that a list is non-empty, the compiler may elide checks for null:

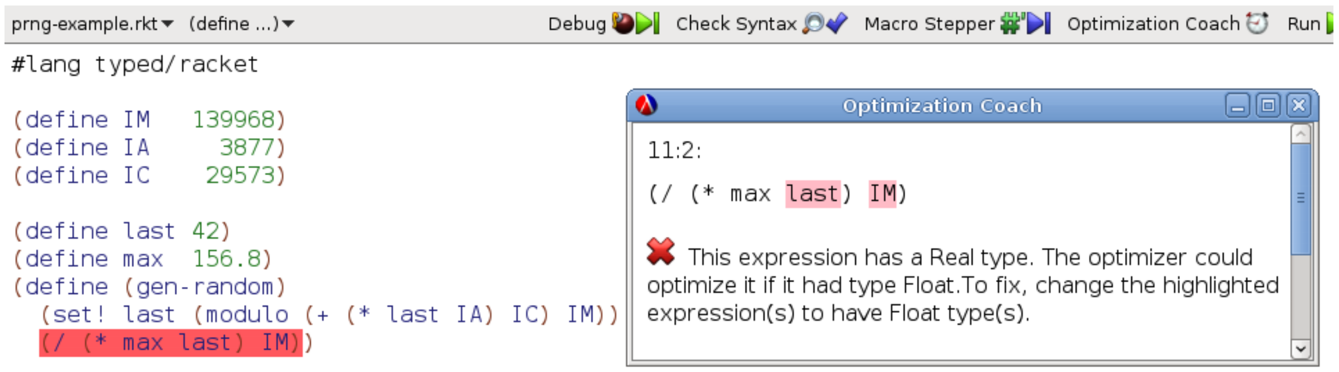
```
(: get-y-coordinate :
   (List Integer Integer Integer) -> Integer)
(define (get-y-coordinate 3d-pt)
  (first 3d-pt))
```

In this case, the type specifies that the input is a three-element list. Hence taking its first element is always safe:

```
(: get-y-coordinate :
   (List Integer Integer Integer) -> Integer)
(define (get-y-coordinate 3d-pt)
  (unsafe-first 3d-pt))
```

Figure 2 shows how Optimization Coach informs the programmer that the type specialization succeeds for such functions. When the highlight surrounding an optimized region is clicked, the tool brings up a new window with extra information about that region. Hints also become available when the optimizer cannot exploit some type information. Consider the following function, which indexes into a TCP packet's payload, skipping the headers:

```
(: TCP-payload-ref : Bytes Fixnum -> Byte)
(define (TCP-payload-ref packet i)
  ; skip the TCP header
  (define actual-i (+ i 20))
  (bytes-ref packet actual-i))
```



```
#lang typed/racket

(define IM   139968)
(define IA     3877)
(define IC    29573)

(define last 42)
(define max  156.8)
(define (gen-random)
  (set! last (modulo (+ (* last IA) IC) IM))
  (/ (* max last) IM))
```

11:2:

`(/ (* max last) IM)`

This expression has a Real type. The optimizer could optimize it if it had type Float.To fix, change the highlighted expression(s) to have Float type(s).

Figure 5: Surprising optimization failure involving mixed-type arithmetic

This program works, but the optimizer cannot eliminate the genericity overhead from the addition. Racket's addition function implicitly promotes results to bignums on overflow, which may happen for the addition of `20` to a fixnum. Therefore, the Typed Racket compiler cannot safely specialize the addition to fixnum-only addition. Optimization Coach detects this near miss and reports it, as shown in figure 3. In addition, Optimization Coach suggests a potential solution, namely, to restrict the argument type further, ensuring that the result of the addition stays within fixnum range.

In figure 4 we show the result of following Optimization Coach's recommendation. Once the argument type is `Index`, the optimizer inserts a fixnum addition for `+`, and Optimization Coach confirms the optimization.

Next we consider a surprising optimization failure:

```
(define IM   139968)
(define IA     3877)
(define IC    29573)

(define last 42)
(define max  156.8)
(define (gen-random)
  (set! last (modulo (+ (* last IA) IC) IM))
  (/ (* max last) IM))
```

This code implements Lewis et al. (1969)'s pseudo-random number generator, using mixed-type arithmetic in the process. In Racket, mixing integers and floating-point numbers in arithmetic operations usually results in the coercion of the integer argument to a floating-point value and the return of a floating-point number. Therefore, the author of this code expected the last expression of `gen-random` to be specialized for floating-point numbers.

Unbeknownst to the programmer, however, this code suffers from a special case in Racket's treatment of mixed-type arithmetic. Integer-float multiplication produces a floating point number, unless the integer is `0`, in which case the result is the integer `0`. Thus the result of the above multiplication is a floating-point number most of the time, but not always, making floating-point specialization unsafe.

The Typed Racket optimizer knows this fact (St-Amour et al. 2012) but most programmers fail to think of it when they program. Hence, this optimization failure may surprise them and is thus worth reporting. Figure 5 shows how Optimization Coach explains this failure.

Again, the programmer can respond to this recommendation with the insertion of coercions:

```
(define (gen-random)
  (set! last (modulo (+ (* last IA) IC) IM))
  (/ (* max (exact->inexact last))
     (exact->inexact IM)))
```

Optimization Coach confirms that this change enables further optimization.

4. The Racket Compiler

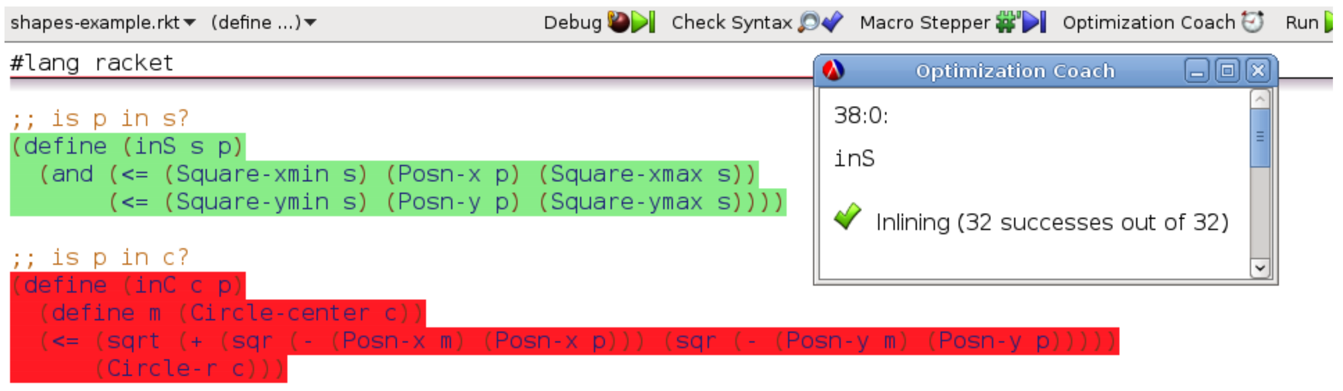
The Racket compiler is a mature ahead-of-time compiler that has been in development for 17 years and comes with a sophisticated optimizer. It compiles Racket programs to bytecode, which the Racket virtual machine then translates to machine code just in time, at the first call of each function. This runtime code-generation does not perform significant additional optimization. The compiler is written in C and consists of several passes.

The first pass, which we focus on in this paper, features a large number of optimizations, including inlining as well as constant and copy propagation plus constant folding. Subsequent passes perform closure conversion (Appel and Jim 1989) and lambda lifting.

Optimization Coach provides two kinds of information concerning inlining transformations:

- which functions are inlined and how often.
- which functions are not inlined, and why.

Both kinds of information are associated with *function definitions*. This is no accident; while inlining is an optimization that happens at call sites, the only way programmers can control the process is through the definition. Therefore, any



```
#lang racket

;; is p in s?
(define (inS s p)
  (and (<= (Square-xmin s) (Posn-x p) (Square-xmax s))
       (<= (Square-ymin s) (Posn-y p) (Square-ymax s))))

;; is p in c?
(define (inC c p)
  (define m (Circle-center c))
  (<= (sqrt (+ (sqr (- (Posn-x m) (Posn-x p))) (sqr (- (Posn-y m) (Posn-y p)))))
      (Circle-r c)))
```

38:0:

inS

✓ Inlining (32 successes out of 32)

Figure 6: Optimization Coach confirming that inlining is happening (with focus on `inS`)

information that would help the user get the most out of the inliner has to be explained in terms of the definition sites.

To illustrate the kind of information provided by Optimization Coach, let us consider two functions implementing geometric formulas: `inS` checks whether a point is inside a square, and `inC` checks whether a point is inside a circle. The programmer introduced these helper functions to make the surrounding program readable, but it would be unfortunate if this refactoring were to result in extra function calls, especially if they were to hurt the program's performance. As shown in figure 6, the Racket inliner does inline `inS` at all of its call sites.

In contrast, the `inC` function is not inlined to a satisfactory level, as evidenced by the red highlight in figure 7. This may be indicative of low-hanging optimization fruit. One way to resolve the issue is to use a Racket macro to force inlining. When we follow this advice, Optimization Coach confirms that the manual inlining is successful. Breaking up the function into smaller pieces might also work.

5. Optimizer Instrumentation

In order to explain an optimizer's results to programmers, we must discover what happens during optimization. One option is to reconstruct the optimizations via analysis of the compiler's output. Doing so would mimic the actions of highly-expert programmers with a thorough understanding of the compiler—with all their obvious disadvantages. In addition, it would forgo all the information that the compiler generates during the optimization phase. Our proposed alternative is to equip the optimizer with instrumentation that gathers information as optimizations are performed or rejected. Debugging tools that explain the behavior of compilers or runtime systems (Clements et al. 2001; Culpepper and Felleisen 2010) have successfully used similar techniques in the past.

The goal of the instrumentation is to generate a complete log of the optimization decisions that the compiler makes as it compiles a given program. In particular, Optimization coaching is concerned with reporting near misses and unexpected optimization failures. For this reason, merely logging optimization successes is insufficient. We also need to log cases where optimizations were not applied, i.e. *optimization failures*. While most of these failures are not interesting to the programmer, pruning the log is left to a separate phase.

The following two subsections explain the instrumentation for the Racket and Typed Racket compilers.

5.1 Instrumentation of Typed Racket

As mentioned, the Typed Racket compiler mostly performs source-to-source optimizations. These optimizations are implemented using pattern matching and templating, e.g.:

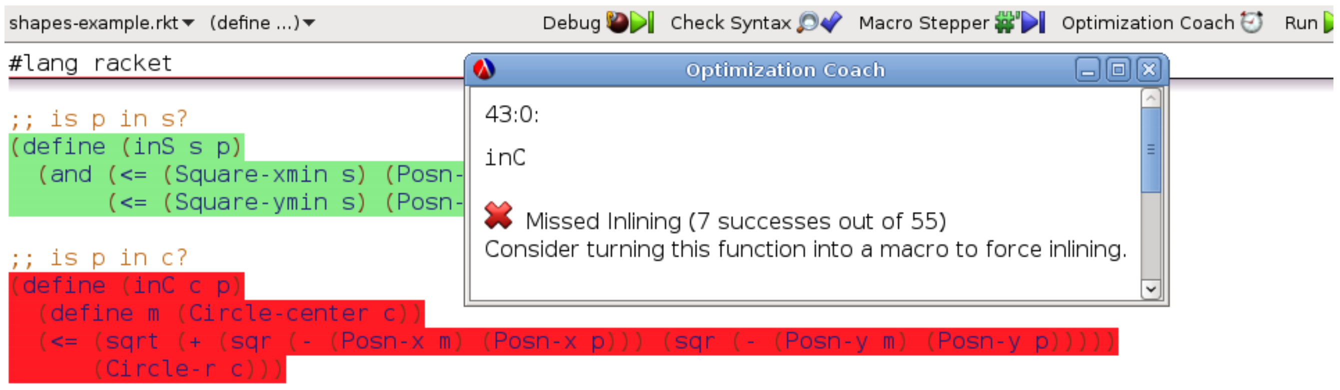
```
(pattern
  (+ f1:float-expr f2:float-expr)
  #:with opt #'(unsafe-fl+ f1.opt f2.opt))
```

Each optimization is specified with a similar `pattern` construct. The only factors that affect whether a term is optimized are its shape—in this case, an AST that represents the application of the addition function—its type, and the type of its subterms.

Instrumenting the optimizer to log optimization successes is straightforward; we add a logging statement to each `pattern` describing which optimization happened, the code involved, its source location, and the information that affected the optimization decision (the shape of the term, its type and the type of its subterms):

```
(pattern
  (+ f1:float-expr f2:float-expr)
  #:with opt
  (begin (log-optimization
           "binary float addition"
           this-syntax)
         #'(unsafe-fl+ f1.opt f2.opt)))
```

When we compile the `add-sales-tax` and `get-y-coordinates` functions from figure 2, the instrumented optimizer generates this log, confirming that it applies the optimizations mentioned above:



```
#lang racket

;; is p in s?
(define (inS s p)
  (and (<= (Square-xmin s) (Posn-
       (<= (Square-ymin s) (Posn-

;; is p in c?
(define (inC c p)
  (define m (Circle-center c))
  (<= (sqrt (+ (sqr (- (Posn-x m) (Posn-x p))) (sqr (- (Posn-y m) (Posn-y p)))))
      (Circle-r c)))
```

43:0:

inC

✖ Missed Inlining (7 successes out of 55)
Consider turning this function into a macro to force inlining.

Figure 7: The `inC` function, failing to be inlined (with focus on `inC`)

```
TR opt: TR-examples.rkt 5:2 (* price 1.0625)
  -- Float Float -- binary float multiplication
TR opt: TR-examples.rkt 10:2 (first 3d-pt)
  -- (List Integer Integer Integer)
  -- pair check elimination
```

To log optimization failures, we add an additional `pattern` form that catches all non-optimized additions and does not perform any optimizations:

```
(pattern
  (+ n1:expr n2:expr)
  #:with opt
  (begin (log-optimization-failure
           "generic addition"
           this-syntax)
         this-syntax)) ; no change
```

This pattern is general enough to produce superfluous logging information: to avoid generating excessive amounts of such information, we restrict failure logging to terms that could at least conceivably be optimized. For instance, we log additions that are not specialized, but we do not log all non-optimized function applications.

As in the optimization success case, the logs contain the kind of optimization considered, the term involved, its type, and the types of its subterms. The following log entry shows evidence for the failed optimization in the `TCP-payload-ref` function of figure 3:

```
TR opt failure: tcp-example.rkt 5:18 (+ i 20)
-- Fixnum Positive-Byte -- generic addition
```

Overall, the instrumentation is fairly lightweight. Each optimization clause is extended to perform logging in addition to optimizing; a few catch-all clauses log optimization failures. The structure of the optimizer is unchanged.

5.2 Instrumentation of the Racket Inliner

The Racket inliner is based on a design by Serrano (1997) and is a much more sophisticated optimizer than the Typed Racket optimizer. Its decision process is guided by multiple factors, including the size of the inlining candidate and the amount of inlining that has already been performed in the context of the candidate call site. This makes the Racket inliner similar to other production compiler optimizers, and it also makes it unrealistic to log all the factors that contributed to a given optimization decision.

An additional constraint is that the inliner is a complex program, and instrumentation should not increase its complexity. Specifically, instrumentation should not add new paths to the inliner. Concretely, this rules out the use of catch-all clauses.

Finally, the Racket inliner is deep enough in the compiler pipeline that most source-level information is not available in the internal representation. As a result, it becomes difficult to correlate optimization decisions with the original program. Again, these constraints on the available information are representative of production optimizers.

To record optimization successes, we identify all code paths that trigger an inlining transformation and, on each of them, log the name of the function being inlined and the location of the original call site. It is worth noting that precise information about the original call site may not be available at this point in the compiler; the call site may also have been introduced by a previous optimization pass, in which case source location would be meaningless. In general, though, it is possible to locate the call site with at least function-level granularity; that is, we can usually determine the function body where the call site is located.

Inlining is a sufficiently general optimization that it could conceivably apply almost anywhere. This makes defining and logging optimization failures challenging. Since our goal is to ultimately enumerate a list of optimization near misses and surprising failures, we need to consider only optimization failures that directly contribute to near misses. For example, the failure of a large function to be inlined is an optimization failure that is unlikely to be linked to a near miss or a surprising optimization failure.

Consequently we consider as optimization failures only cases where inlining was considered by the compiler, but ultimately decided against. We identify the code paths where inlining is decided against and add logging to them. As in the case of inlining successes, we log the name of the candidate and the call site. In addition, we also log the cause of failure to provide an explanation to the user. The most likely cause of failure is the inliner running out of *fuel*. To avoid code size explosions, each call site is given a limited amount of fuel, and inlining a function inside this call site consumes a quantity of fuel proportional to the size of the function being inlined. If the inliner runs out of fuel for a specific instance, it is not performed and the optimization fails. For these kinds of failures, we also log the size of the function being considered for inlining, as well as the remaining fuel.

Here is an excerpt from the inliner log produced when compiling the `binarytrees` benchmark from section 9:

```
mzc: no inlining, out of fuel #(for-loop 41:6)
   in #(main 34:0) size=77 fuel=8
mzc: inlining #(loop 25:2) in #(check 24:0)
mzc: inlining #(loop 25:2) in #(check 24:0)
mzc: no inlining, out of fuel #(loop 25:2)
  in #(check 24:0) size=28 fuel=16
mzc: inlining #(check 24:0) in #(for-loop 46:1)
mzc: no inlining, out of fuel #(check 24:0)
  in #(for-loop 46:18) size=31 fuel=24
```

6. Optimization Analysis

Compiling a program with an instrumented compiler generates a log of optimization-related decisions. We use this log as a starting point to generate a high-level optimization report for the programmer.

Optimization logs are not directly suitable for user consumption due to several reasons. First, the sheer amount of data in these logs makes it hard to visualize the optimization process. Indeed, large portions of that information turn out to be irrelevant.

Second, some optimizations are related by causality: one optimization can open up further optimization opportunities, or an optimization failure may prevent other optimizations from happening. In these cases, presenting a combined report of the related optimizations yields more useful information than reporting them individually. Otherwise, Optimization Coach would shift the burden of establishing causality to the programmer, which we consider unacceptable.

Finally, some optimizations are related by locality: multiple optimizations are applied to one piece of code or the same piece of code triggers optimizations in different places. Again, aggregated information about these optimization events helps summarize information.

In the remainder of this section, we explain the three main phases of the optimization analysis in general terms. The last two subsections illustrate them with concrete examples.

6.1 Log Pruning

The sheer amount of log data calls for criteria that discriminate good opportunities from irrelevant information. For the former, we introduce the notion of optimization proximity; for the latter, we use three ground rules.

Optimization proximity is an optimization-specific metric that measures how close an optimization is from happening. For a particular optimization, it might be derived from the number of program changes that would be necessary to trigger the optimization of interest. Log entries with close proximity are retained, others are pruned from the log.

Some log entries are considered *incomprehensible* because they are about code that is not present in the original program. Such code might be introduced by macro-expansion or by previous optimization passes. Since the main goal of optimization coaching is to help programmers adapt their code to enable further compiler optimization, it is of limited usefulness to report about code that is not present in source programs.

Other optimization failures are considered *irrelevant* because the non-optimized semantics is likely to be desired by design. For example, reporting that an addition could not be specialized to floating-point numbers is irrelevant if the addition is used in a generic way. Presenting recommendations that programmers are most likely to reject is unhelpful, so we prune these kinds of reports.

Finally, we prune *harmless* optimization failures, a kind that we associate with opportunities due to actual optimizations. Consider a loop that is unrolled several times—eventually, unrolling must stop. This final decision is still reported in the log as an optimization failure, because the optimizer considered unrolling further but decided against it. It is a harmless failure, however, because loop unrolling cannot go on forever. These failures and others like them are therefore pruned from the logs.

6.2 Causality Merging

Once the log has been pruned, the optimization analyzer tries to detect clusters of optimization events that are related by causality. It consolidates these clusters into a single optimization event. We call this step *causality merging*. Our notion of causality heavily relies on the concept of *irritant*. In the case of some optimizations, we can relate an optimization failure for a term to one or several of its subterms. These subterms may have the wrong shape or the wrong type.

Since irritants are terms, they can themselves be the subjects of optimization failures, meaning they may contain irritants. All these related optimization failures are grouped into a tree. Each failure becomes a node, and its irritants become children. Irritants that do not contain irritants form the leaves of the tree; they are the initial causes of the optimization failure. The optimization failure that is caused by all the others becomes the root of the tree. The entire tree can be merged into a single entry, with the root of the tree as its subject and the leaves as its irritants. The intermediate nodes of the tree can be safely ignored since they are not responsible for the failure; they only propagate it.

Causality merging reduces the size of the logs and helps pinpoint the program points that cause optimization failures. Furthermore, grouping these optimization events enables a ranking of missed optimizations according to their impact on the overall optimization status of the program. If we know a subexpression heads a cascade of missed optimizations, fixing it will have a significant impact. Thus, the size of the irritant tree is used in the merged log entry as a *badness* measure to formulate the presentation of the analysis results.

In addition to being useful for causality merging, the notion of irritant is also useful to help explain optimization failures to the programmer and to synthesize recommendations.

6.3 Locality Merging

Next, the optimization coach synthesizes a single log entry for clusters of optimization reports that affect the same piece of code. Doing so helps formulate a coherent diagnosis about all the optimizations that affect a given piece of code. We call this phase *locality merging*.

Reports that are related by locality but concern unrelated optimizations are still grouped together, but the grouping process unions the reports without further analysis. These groupings, while they do not infer new information, improve the user interface by providing all the relevant information about a piece of code in a single location.

This step also aggregates log entries that provide information too low-level to be of use to the programmer directly. New log entries report on these patterns and replace the low-level entries in the log.

6.4 Analysis of Typed Racket Optimizations

Log pruning and causality merging are particularly relevant in the context of the Typed Racket optimizer. In this section we illustrate these concepts with concrete examples. For instance, the following function

```
(: sum : (Listof Number) -> Number)
(define (sum list-of-numbers)
  (if (null? list-of-numbers)
      0
      (+ (first list-of-numbers)
         (sum (rest list-of-numbers)))))
```

uses `+` in a generic way. This code is not specialized by the Typed Racket optimizer, creating the following log entry:

```
TR opt failure: sum-example.rkt 5:6
(+ (first list-of-numbers)
   (sum (rest list-of-numbers)))
-- Number Number -- generic addition
```

Since the generic behavior is actually desired for `sum`, it should not be eliminated by optimization. Optimization analysis should therefore detect these cases and remove them from the logs.

To trigger a specialization optimization, all the arguments to a generic operation must be convertible to the same type, and that type needs to be one for which a specialized version of the operation is available. We define optimization proximity for this optimization to be the number of arguments that would need to have a different type in order to reach a state where optimization could happen.

For example, the combination of argument types

```
(+ Float Real)
```

is 1-close to being optimized, while this one

```
(+ Float Real Real)
```

is 2-close, and thus further from being optimized, since addition cannot be specialized for `Real` numbers, but can be for `Float`s. Only optimization failures within a specific proximity threshold are kept. Our prototype uses a threshold of 1 for this optimization, which works well in practice. The generic addition in the `sum` function above has a 2-close measure and is therefore ignored by Optimization Coach.

Our Typed Racket prototype also provides examples of causality merging. Let us consider the last part of the pseudo-random number generator from section 3:

```
(/ (* max last) IM)
```

While `max` is of type `Float`, `last` and `IM` have type `Integer`. Therefore, the multiplication cannot be specialized, which causes the type of the multiplication to be `Real`. This type, in turn, causes the division to remain generic. Here is the relevant excerpt from the logs:

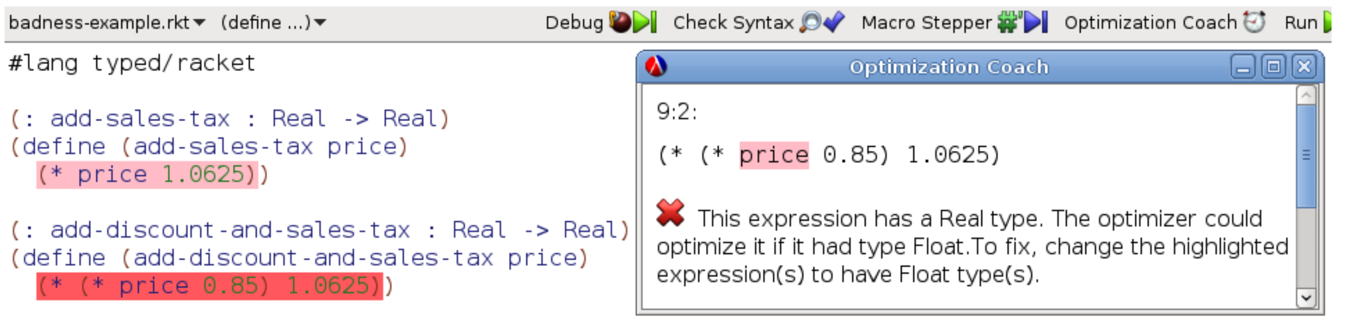
```
TR opt failure: prng-example.rkt 11:5
  (* max last)
  -- Float Integer -- generic multiplication
TR opt failure: prng-example.rkt 11:2
  (/ (* max last) IM)
  -- Real Integer -- generic division
```

The subject of the multiplication entry is an irritant of the division entry. Optimization Coach joins the two entries in a new one: the entire division becomes the subject of the new entry with `last` and `IM` as irritants.

6.5 Analysis of Racket Inlining Optimizations

Optimization log entries from the Racket inliner provide two main pieces of information: which function is inlined, and at which call site it is inlined. On one hand, this information is not especially enlightening to users. On the other hand, for widely used functions, it is likely that there is a large number of such reports; the number of inlining reports grows with the number of call sites of a function.

Clusters of log entries related to inlining the same function are a prime target for locality merging. Locality merging provides an aggregate view of a function's inlining behavior. Based on the ratio of inlining successes and failures, Optimization Coach decides whether the function as a whole is successful with regards to inlining or not.



```
#lang typed/racket

(: add-sales-tax : Real -> Real)
(define (add-sales-tax price)
  (* price 1.0625))

(: add-discount-and-sales-tax : Real -> Real)
(define (add-discount-and-sales-tax price)
  (* (* price 0.85) 1.0625))
```

```
9:2:

(* (* price 0.85) 1.0625)
```

This expression has a Real type. The optimizer could optimize it if it had type Float.To fix, change the highlighted expression(s) to have Float type(s).

Figure 8: Optimization Coach ordering near misses by predicted importance (with focus on `add-discount-and-sales-tax`); the most important near miss is in a darker shade of red and the least important one is in a lighter shade of red

A function with a high success to failure ratio is reported as an inlining success. In contrast, a function for which inlining failures outnumber sucesses is reported as a potential point for improvement. This ratio is also used to compute the badness measure with which Optimization Coach ranks the importance of optimization events. We determined the threshold ratios for successes and failures empirically; thus far, they have been accurate in practice.

In the case of failures the instrumented compiler also logs the size of the function and the amount of fuel left. This information can be used to assign a weight to specific inlining failures before computing the above ratio. When little fuel is left, only small functions can be inlined. In these cases, the blame for the inlining failure lays more with the call site than with the function being called. Therefore, these failures should count for less than other failures; applying such discount avoids penalizing the relevant function.

Since Racket is a mostly functional language, loops are expressed as tail recursive functions; therefore, function inlining also expresses loop unrolling. An important role of optimization analysis for the Racket inliner is to determine which log entries were produced as part of loop unrolling and which were produced as part of traditional inlining. This analysis can only be performed post-facto because the same code paths apply to both forms of inlining.

Considering unrolling and inlining separately has two benefits. First, as previously mentioned, unrolling failures are not usually problematic; any series of unrollings needs to end in a failure. These failures should not be confused with inlining failures, which may actually be problematic. Treating failed unrollings separately avoids false positives, which would lead to overly pessimistic reports. Second, absence of inlining successes for a given function is usually more alarming than absence of unrollings. Confusing inlining and unrolling successes can lead to false negatives. A function that is never inlined, but does get unrolled several times, would be reported as an overall success. Fortunately, considering an unrolling of a recursive function is a simple and highly effective heuristic to identify loop unrollings. It is accurate in practice and leads to improved analysis results.

7. Generating Recommendations

After optimization analysis, reports of success and failure are at the appropriate level of abstraction. In this shape, they are suitable for presentation to the programmer. Furthermore, it is now possible to generate concrete advice from these reports—when possible—to help programmers eliminate optimization failures.

To identify the expressions for which to recommend changes, the optimization coach reuses the concept of irritant. Recall that irritants are terms whose structure caused optimization to fail. If these terms were changed to have an appropriate structure, the optimizer would be able to apply transformations. Irritants are thus ideal candidates for recommendation targets.

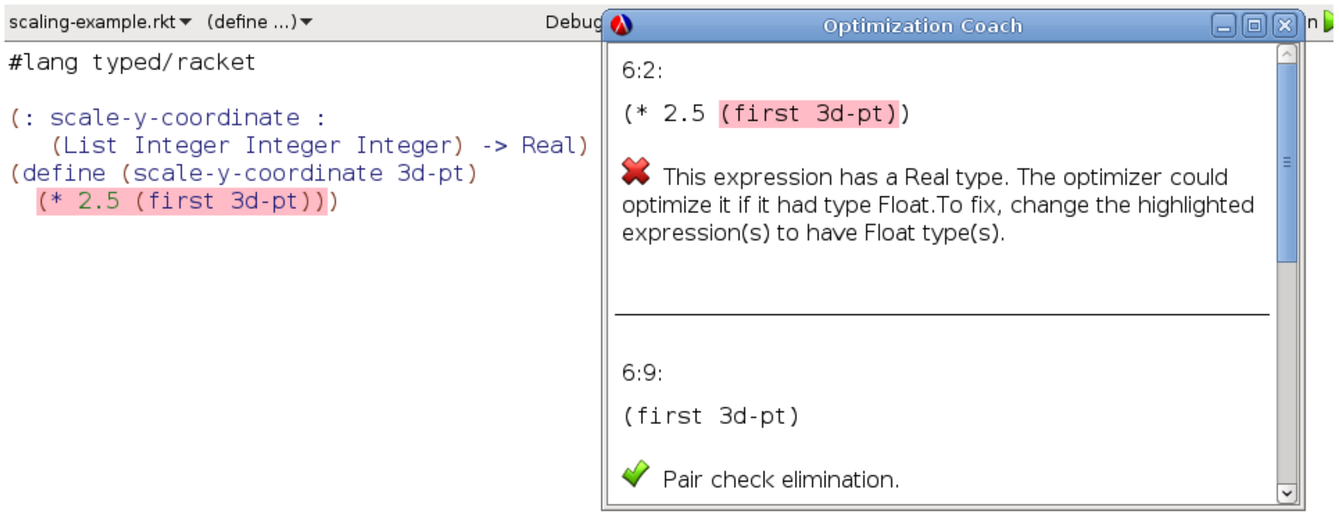
Determining the best change to a given irritant relies on optimization-specific logic. Since each optimization has its own failure modes, general rules do not apply here.

In some cases, generating recommendations is impossible, mostly due to the nature of an optimization or to the particular term. Irritants are reported nonetheless; knowing the *cause* of a failure may still help the programmer.

7.1 Recommendations for Typed Racket

In the context of Typed Racket, Optimization Coach's recommendations suggest changes to the types of irritants. The fixes are determined as part of the optimization analysis phase. Recommendation generation for Typed Racket is therefore a straightforward matter of connecting the facts and presenting the result in an informative manner.

For example, when presented with the fixnum arithmetic example from figure 3, Optimization Coach recommends changing the type of `i` to `Byte` or `Index`. In the case of the pseudo-random number generator from figure 5, Optimization Coach recommends changing the types of the irritants to `Float` to conform with `max`.



```
#lang typed/racket

(: scale-y-coordinate :
   (List Integer Integer Integer) -> Real)
(define (scale-y-coordinate 3d-pt)
  (* 2.5 (first 3d-pt)))
```

6:2:

```
(* 2.5 (first 3d-pt))
```

✗ This expression has a Real type. The optimizer could optimize it if it had type Float.To fix, change the highlighted expression(s) to have Float type(s).

6:9:

```
(first 3d-pt)
```

✓ Pair check elimination.

Figure 9: A function affected by both an optimization success and an optimization near miss

7.2 Recommendations for Inlining

Inlining is not as easy to control through changes in the source program as Typed Racket's type-driven optimizations. Therefore, recommendations relating to inlining optimizations are less precise than those for Typed Racket.

Since lack of fuel is the main reason for failed inlinings, reducing the size of functions is the simplest recommendation. In some cases, it is possible to give the programmer an estimate of how much to reduce the size of the function, using its current size and the remaining fuel.

For languages with macros, an optimization coach can also recommend turning a function into a macro, i.e., inlining it manually at all call sites. To avoid infinite expansion, Optimization Coach recommends this action only for non-recursive functions. Figure 7 shows examples of Optimization Coach's recommendations related to inlining.

8. User Interface

The integration of an optimization coach into the programmer's workflow requires a carefully designed tool for presenting relevant information. While the tool must present some of the information immediately, it must hide other pieces until there is a demand for it. In both cases, the presentation must remain easily comprehensible.

Optimization Coach is a plug-in tool for DrRacket (Findler et al. 2002). As such a tool, Optimization Coach has access to both the Racket and Typed Racket compilers and can easily collect instrumentation output in a non-intrusive fashion. At the press of a button, Optimization Coach compiles the program, analyzes the logs, and presents the results.

As our screenshots show, Optimization Coach highlights regions that are affected by either optimizations or near misses. To distinguish the two, green boxes highlight optimized areas and red boxes pinpoint areas affected by near misses. If a region is affected by both optimizations and missed optimizations, a red highlight is used; missed optimizations are more cause for concern than optimizations and should not be hidden. The `scale-y-coordinate` function from figure 9 contains both an optimization success—taking the first element of its input list—and an optimization near miss—scaling it by a floating-point factor.

The user interface uses different shades of red to express an ordering of near misses according to the number of optimization failures involved. Optimization Coach uses the *badness* measure introduced in section 6.2 to generate this ordering. Figure 8 shows ordering in action; the body of `add-sales-tax` contains a single near miss and is therefore highlighted with a lighter shader of red, distinct from the body of `add-discount-and-sales-tax`, which contains two near misses.

Clicking on a region brings up a tool-tip that enumerates and describes the optimization events that occurred in the region. The description includes the relevant code, which may be a subset of the region in the case of nested optimizations. It especially highlights the irritants and, with these, explains the event. Finally, the window also comes with recommendations when available.

9. Evaluation

To validate our optimization coach empirically, we conducted two experiments to confirm that our optimization coach can find critical optimizations and that following its recommendation improves the performance of programs. First, we ran Optimization Coach on non-optimized versions of five small benchmarks from the Computer Language Benchmark Game,[2] followed the recommendations, measured the performance of the resulting programs, and

[2] `http://shootout.alioth.debian.org`

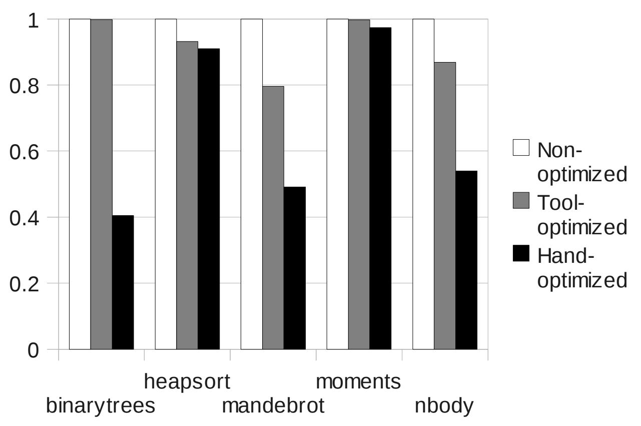


Figure 10: Optimization Coach's impact on benchmarks (smaller is better)

compared it to versions of the programs that had been hand-optimized by experts. Second, we identified two small but performance-intensive Racket applications and used Optimization Coach to identify missed optimizations.[3]

9.1 Benchmarks

The selected benchmark programs were highly tuned by expert Racket programmers. For each program, we also benchmark the non-optimized version from which the highly-optimized version is derived. Then we used the optimization coach to construct a third version of each benchmark. Specifically, we ran Optimization Coach on the non-optimized version and followed all of its recommendations.

Figure 10 compares the running time of the three versions, and illustrates how close we get to the highly-tuned, hand-optimized version of each benchmark just by following Optimization Coach's recommendations. On three of the benchmarks, Optimization Coach's recommendations translate to significant performance improvements. On two of these three, the Optimization Coach-optimized version is almost as fast as the hand-optimized version.

In addition to this quantitative evaluation, we performed a qualitative evaluation by comparing the specific optimizations recommended by Optimization Coach to those performed by experts. For all the benchmarks, the hand-optimized version includes all the optimizations recommended by Optimization Coach. No false positives and no novel optimizations were found. Experts also performed optimizations outside the domains covered by Optimization Coach, suggesting potential areas for future improvements.

Here are specific comments on the benchmark programs:

[3] See `http://github.com/stamourv/oopsla2012-benchmarks` for the complete code for the experiments. All our benchmarking timings report the average of 10 runs on a 2.50GHz Intel Core 2 Quad 8300 processor with 2 GB of memory running Ubuntu 11.10.

binarytrees Optimization Coach does not provide useful recommendations for improving optmization. However, it does confirm that all helper functions are inlined as expected. The performance gains observed in the hand-optimized versions come almost entirely from a single optimization: all the data structures were changed from structures to vectors, which have a lower allocation overhead. This transformation is not worth recommending in general because structure semantics is often desirable. However, an optimization coach with access to profiling information could recommend changing structures to vectors if structures are allocated in a hot loop. This integration is a key area of future work.

heapsort Optimization Coach finds most of the optimizations present in the hand-optimized version. The `heapsort` implementation is written in Typed Racket and most optimizations are type-related. Optimization Coach successfully recommends changing types and inserting coercions in all the places where the expert programmer inserted them. Manual vector bounds-check elimination explains the remaining performance gains in the hand-optimized version.

mandelbrot The hand-optimized version benefits from two major optimizations: the manual inlining of a helper function too large to be inlined by the optimizer and the manual unrolling/specialization of an inner loop. Optimization Coach detects that the helper function is not being inlined and recommends using a macro to force inlining, which is exactly what the expert did. Optimization Coach does not recommend manually unrolling the loop; its recommendation generation process does not extend to unrolling.

moments The expert-optimized version of the `moments` benchmark, which also uses Typed Racket, includes replacing types with low optimization potential with types with greater potential. In places, this transformation involves adding dynamic coercions to the desired type when the type system cannot guarantee the desired type bound. Optimization Coach recommends the same type changes performed by the expert. However, Optimization Coach's recommendations do not place the coercions in the same locations as the expert's. The expert placed the coercions in locations where their runtime overhead is low, which is responsible for the performance gains of the hand-optimized version over the Optimization Coach-optimized version.

nbody Optimization Coach identifies expressions where the Typed Racket compiler must conservatively assume that `sqrt` may produce complex numbers, leading to optimization failures. Following Optimization Coach's recommendations, we replaced uses of `sqrt` by `flsqrt`, which is guaranteed to return floating-point numbers, leading to another optimization visible in the hand-optimized version. The rest of the performance improvements in the hand-optimized version are due to replacing structures and lists with vectors, as in `binarytrees`.

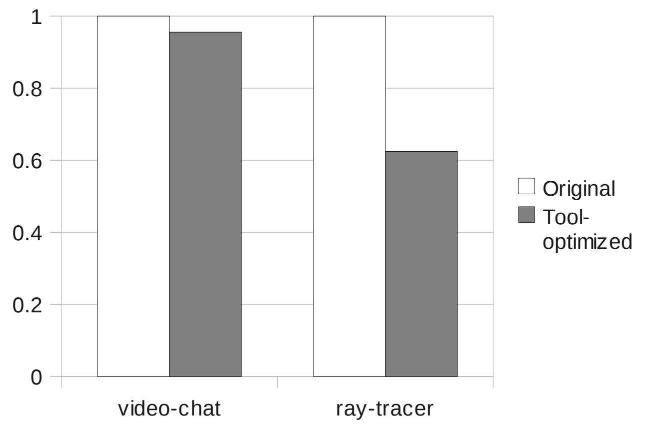


Figure 11: Optimization Coach's impact on full applications (smaller is better)

9.2 Full Applications

To measure Optimization Coach's effectiveness in a realistic setting, we applied it to two Racket applications: a simple video chat client written by Tony Garnock-Jones (Northeastern University Programming Research Lab) and a ray tracer for rendering icons written by Neil Toronto (Brigham Young University). Messrs. Garnock-Jones and Toronto are highly experienced Racket programmers, and both worked hard to make their applications performant.

Our experiment proceeded as follows. First, we ran Optimization Coach on each code base. Second, we modified the programs following the recommendations that Optimization Coach considered most important. Finally, we measured the performance impact of the changes using the same setup as the benchmark measurements. Figure 11 shows our performance measurements, in the same format as figure 10.

Video Chat The first application we studied is a simple video chat client. For scale, the chat client is 860 lines of Racket code. We focused our effort on the simple differential video coder-decoder (codec) in the client. To measure the performance of each version of the codec, we timed the decoding of 600 pre-recorded video frames. The code mostly consists of bitwise operations on pixel values. Optimization Coach uncovered additional opportunities for the Racket optimizer.

The decoding computation is spread across several helper functions. Optimization Coach confirmed that Racket inlines most of these helper functions, avoiding extra function call overhead and enabling other optimizations. However, the three largest helper functions (`kernel-decode`, `clamp-pixel` and `integrate-delta`) were either not inlined to a satisfactory level or not inlined at all. Optimization Coach predicted that inlining `kernel-decode` and `clamp-pixel` would have the biggest impact on performance and highlighted them in a darker shade of red than `integrate-delta`, as decribed in section 8.

We followed these two recommendations, turning both helper functions into macros. Each change was local, required only a change to a single line of code and did not require understanding the behavior of the function or its role in the larger computation. These two changes are solely responsible for the speedup shown in figure 11.

Afterwards, we followed the last, weaker, recommendation: inlining `integrate-delta`. The impact on performance was negligible. This suggests that, at least for this program, Optimization Coach's recommendation ordering heuristics are accurate and useful in practice. However, the tool can order recommendations only relative to each other. After the important recommendations were followed, it became impossible to tell whether the impact of the last one would be significant. This points out an interesting limitation. Running Optimization Coach on programs with little potential for optimization is likely to yield only low-impact recommendations, but programmers cannot learn this by looking at the output of the tool alone. Addressing this limitation calls for future work, possibly the integration of a profiling tool.

Ray Tracer The second application we studied is a ray tracer that is used to render the logos and icons used by DrRacket and the Racket website. For scale, the ray tracer is 3,199 lines of code. It supports a wide variety of features, such as refraction, multiple highlight and reflection modes, plus text rendering.

Over time, most icons in the Racket codebase have been ported to use this ray tracer, and it eventually became a bottleneck in the Racket build process. Its author spent significant time and effort[4] to improve its performance.

To determine whether an optimization coach would have helped him, we attempted to optimize the original version of the ray tracer ourselves using Optimization Coach. For our measurements, we rendered a 600×600 pixel logo using each version of the ray tracer.

Starting from the original version of the ray tracer, we ran Optimization Coach on the files containing the core of the computation. Optimization coach identified three helper functions that Racket failed to inline. We followed its recommendations by turning the three functions into macros. As with the video codec, the changes were local and did not require knowledge of the code base. Furthermore, diagnosis was entirely automatic; the tool pinpointed the exact location of the recommended change.

The ray tracer's author had independently performed the same changes, which were responsible for most of the speedups over his original ray tracer. Optimization Coach successfully identified the same sources of the performance

[4] Leading to the discussion mentioned in the introduction.

bugs as a Racket expert and provided solutions, making a strong case for the effectiveness of optimization coaching.

10. Related Work

Optimization coaching assists programmers with finding missed optimizations and gives suggestions on how to enable them. We briefly survey previous efforts that pursue related goals.

10.1 Profilers

Programmers use profilers (Altman et al. 2010; Ammons et al. 2004; Jovic et al. 2011) heavily when diagnosing performance issues. Profilers answer the question

Which pieces of the program take a long time?

but programmers still need to determine which parts of a program take an *abnormally* long time before they can address performance issues.

Optimization coaches, in contrast, answer a different question, namely

Which pieces could still be optimized?

and the answers identify the location of potential code improvements. Optimization coaches are unaware of which parts of the program are heavily executed, though, and may suggest recommendations about infrequently used regions. Such recommendations are unlikely to lead to significant performance improvements. Similarly, optimization coaches may recommend performing optimizations that, despite applying to hot code, have only minimal effect on performance.

It is up to programmers to judge which recommendations are likely to have a significant impact on the resulting performance of the program and to focus on those. This duality suggests that profilers and optimization coaches are complementary tools and should be used together. Combining the two into a single, unified tool is a direction for future work.

Furthermore, for profilers to produce meaningful output, they need to run programs with representative input, but performance-heavy inputs may appear only after deployment, at which point fixing performance bugs becomes significantly more expensive than during development. In contrast, optimization coaching operates statically, in the absence of program input. Then again, an optimization coach reports observations about the optimizer's omissions, and a fix may or may not impact the performance of the program.

Finally, profilers also do not interpret their results to explain what causes specific program regions to be slow nor do they provide recommendations for improvement. The Zoom[5] system-wide profiler is an exception, providing hints to programmers about possibly slow operations. However, Zoom describes their operations at the assembly instruction level, which makes it challenging for programmers—especially for non-experts—to act on these recommendations using a high-level language.

10.2 Analysis Visualization

A large number of tools exist for visualizing analysis results, of which MrSpidey (Flanagan et al. 1996) is an early example. Some of these tools focus on helping programmers understand and debug their programs; others help compiler writers understand and debug their analyses.

Two recent efforts additionally aim to help programmers optimize their programs. Lhoták's work (Lhoták et al. 2004; Shaw 2005) introduces a plug-in for the Eclipse IDE that displays the results of static analyses computed by Soot (Vallée-Rai et al. 2000), an optimization framework for Java bytecode. While most of these visualizations are targeted at compiler researchers or students learning about compilers, their visualization (Shaw (2005), page 87) of Soot's array bounds check analysis (Qian et al. 2002) informs programmers about provably safe array accesses. Similarly, their visualization of loop invariants (Shaw (2005), page 99) highlights expressions that, according to Soot's analysis, can safely be hoisted out of the loop by the compiler.

Although these visualizations are relevant for programmers concerned with optimization, they differ from those of an optimization coach in two major ways. First, they report the results of an *analysis*, not those of the *optimizer*. The two are closely related,[6] but analysis information is only a proxy for the decisions of the optimizer.

Second, while Lhoták's tool reports potentially unsafe array accesses—and explains which bounds are to blame—it does not attempt to distinguish between expected failures and near misses. In contrast to Optimization Coach, it also fails to issue recommendations concerning which piece of code needs to change to eliminate the array bounds check.

JIT Inspector[7] is a Firefox extension that provides optimization metrics for JIT-compiled Javascript code. Most importantly, it provides information about whether operations are executed in JIT code or whether they require calls to the runtime system.

In addition, JIT Inspector gives programmers some insight into the type inference (Hackett and Guo 2012) process that the SpiderMonkey JIT compiler uses to guide optimization. Like the Soot-Eclipse plug-in, however, JIT Inspector presents the results of an *analysis* not those of the optimizer.

10.3 Compiler Logging

Several compilers implement logging in their optimizers, e.g., GCC (The Free Software Foundation 2012) supports the `-ftree-vectorizer-verbose` flag for its vectorizer. Similarly, GHC (The GHC Team 2011) provides the

[5] `http://www.rotateright.com/zoom.html`

[6] We ignore the fact that, as implemented, the plugin uses its own instance of Soot (Shaw (2005), page 15), that may not reflect the analyses performed by the compiler.

[7] `http://addons.mozilla.org/firefox/addon/jit-inspector/`

`-ddump-rule-firings` flag that enumerates the rewriting rules that it applies to its input program. SBCL (The SBCL Team 2012) goes one step further and also logs some optimization failures, such as failures to specialize generic operations or to allocate some objects on the stack.

The logged information of these compilers is similar to the result of the logging phase of our optimization coach. This information can be useful to expert programmers with a solid understanding of the compiler internals, but without the interpretive phases of an optimization coach, it is not suitable for non-expert programmers or for casual use.

The FeedBack compiler (Binkley et al. 1998), based on `lcc` (Fraser and Hanson 1995), improves on compiler logging by visualizing the optimizations it applies to programs. It was primarily designed to help students and compiler writers understand how specific optimizations work. Implicitly, it also informs programmers of the optimizations applied to their programs. The Feedback Compiler's visualizations illustrate two optimizations: a stepper-like interface that replays common subexpression elimination events and a graphical representation of array traversals affected by *iteration space reshaping* (Wolfe 1986).

While the output of the FeedBack compiler is easier to understand than a textual log, the tool suffers from most of the same problems as optimization logging. It directly reports optimization events, without post-processing or filtering them, leaving the interpretation step to programmers. It also does not detect or report near misses; optimization failures must be inferred by programmers from the absence of optimization reports.

10.4 Other Tools

Jin et al. (2012) extract source-based rules from known performance bugs in various applications, then use these rules to detect previously unknown performance bugs in other applications. When their rules uncover matches, their tools recommend the change that fixed the original bug as a potential solution for the newly discovered bug.

Their corpus of rules successfully discovers potential performance bugs, which their suggestions can fix. Their work focuses on algorithmic and API usage-related performance bugs and is complementary to optimization coaching.

Kremlin (Garcia et al. 2011) is a tool that analyses program executions and generates recommendations concerning parallelization efforts. Like an optimization coach, the Kremlin tool issues program-specific recommendations but, in contrast to an optimization coach, it works dynamically. Like a profiler, it requires representative input to produce meaningful results, which exposes it to the same limitations and criticisms as profilers. Finally, Kremlin is a special-purpose tool; it is unclear whether its techniques would apply to other optimization domains.

11. Future Work

Optimization coaching should apply beyond the optimizations covered here. In this section, we discuss several standard optimizations and describe how optimization coaching may apply. For each optimization, we explain its purpose and prerequisites. We then propose a way of detecting near misses of this kind of optimization, which is usually but not always based on optimization proximity. Finally, we sketch ideas for how a programmer could react to a near miss.

Common Subexpression Elimination (CSE) CSE (Muchnick (1997), §13.1) is only valid if the candidate expressions do not depend on variables that may be mutated.

An optimization coach can detect cases where an expression is computed multiple times—and is a candidate for CSE—but a reference to a mutated variable prevents the optimization from happening. The optimization coach could recommend that the programmer reconsider mutating the relevant variable.

Test Reordering Conditionals with multiple conjuncts can be optimized by performing the cheaper tests (e.g. integer comparisons) before the more expensive ones (e.g. unknown function calls). This optimization (Muchnick (1997), §12.3) is only valid if the reordered tests are pure.

Counting the number of impure tests should identify near misses; specifically, the lower the ratio of impure tests to the total number of tests, the closer the optimizer is to triggering the reordering. To further rank missed optimizations, an optimization coach can take into account the size of the body of the conditional because the cost of the condition matters more for small conditionals. When reporting near misses, the optimization coach can recommend making the problematic tests pure or reordering them manually.

Scalar Replacement A scalar replacement optimization (Muchnick (1997), §20.3) breaks up aggregates, e.g., structures or tuples, and stores each component separately. Like inlining, scalar replacement is mostly useful because it opens up further optimization possibilities and reduces allocation. Scalar replacement is performed only when the target aggregate does not escape.

Typed Racket performs scalar replacement on complex numbers. Optimization Coach reports when and where the optimization triggers, but it does not currently detect near misses or provide recommendations.

An optimization coach could use the ratio of escaping use sites to non-escaping use sites of the aggregate as a possible optimization proximity metric. This metric can be refined by considering the size of the aggregate: breaking up larger aggregates may enable more optimizations than breaking up smaller ones. When it does discover near misses, the optimization coach can recommend eliminating escaping uses or manually breaking up the aggregate.

Loop-Invariant Code Motion (LICM) A piece of code is loop-invariant (Muchnick (1997), §13.2) if all reaching definitions of variables in the piece of code come from outside the loop. If one or more relevant definitions come from *inside* the loop, the optimization is inapplicable.

A potential proximity metric for LICM would measure how many problematic assignments in the body of the loop could potentially be avoided. This can be refined by also considering the ratio of assignments and references inside the loop for problematic variables. Variables that are used often and mutated rarely are more easily made loop-invariant than those that are mutated often. When presenting such near misses, the optimization coach could recommend avoiding the problematic assignments or performing the code motion manually. Assignments to loop index variables cannot be avoided, however; attempting to make them loop-invariant would defeat the purpose of using a loop. Such necessary assignments can be recognized and should not count towards the optimization proximity metric.

Devirtualization In object-oriented languages, a method call can be turned into a direct procedure call if its receivers belong to one single class. The transformation avoids the cost of runtime dispatch (Dean et al. 1995). If the receivers belong to a small set of classes, the compiler can inline a small type-case at the call site, which is still cheaper than full-fledged method dispatch. If the set of classes is too large, the compiler uses general dispatch.

An obvious proximity metric for devirtualization would be the number of receiver classes minus the maximum for which the optimizer generates a type-case. In cases where this number is small, the optimization coach can recommend using more precise type annotations to narrow down the type of the receiver or writing the type-case manually.

Reducing Closure Allocation When a compiler decides whether—and where—to allocate closures, they take into account a number of factors (Adams et al. 1986). A closed function corresponds to just a code pointer. A non-escaping function can have its environment allocated on the stack, avoiding heap allocation. Efficient treatment of closures is a key optimization for functional languages.

An optimization coach could warn programmers when the compiler needs to allocate closures on the heap. For instance, storing a function in a data structure almost always forces it to be allocated on the heap. An optimization coach can remind programmers of that fact, and encourage them to avoid storing functions in performance-sensitive code.

To reduce the number of false positives, the optimization coach can discard reports that correspond to common patterns where heap allocation is desired, such as functions that mutate variables from their lexical context.

Specialization of Polymorphic Containers Polymorphic containers, such as lists, usually require a uniform representation for their contents, in order for operations such as `map` to operate generically. In some cases, however, a specialized representation e.g., a list of unboxed floating-point numbers, can be more efficient. Leroy (1992) and Shao and Appel (1995) propose optimizations that allow specialized representations to be used where possible and to fall back on uniform representations when not. To avoid excessive copying when converting between representations, these optimizations are typically applied only when the element types are sufficiently small. For example, a list of 4-tuples may be specialized, but not a list of 5-tuples.

An optimization coach could communicate to the programmer which datatypes are not specialized and report how much they need to be shrunk to enable optimization.

Case-of-Case Transformation The case-of-case transformation (Peyton Jones 1996) rewrites nested pattern matching of the form (in Haskell syntax)

```
case (case E of {P1 -> R1; P2 -> R2}) of
 {Q1 -> S1; Q2 -> S2}
```

to this form

```
case E of {P1 -> case R1 of {Q1 -> S1; Q2 -> S2};
           P2 -> case R2 of {Q1 -> S1; Q2 -> S2}}
```

which potentially exposes more optimization opportunities.

This transformation may lead to code duplication—just as with inlining—and may generate extra closures for join points. Whether it is worth performing depends on the optimizations it enables, and compilers must resort to heuristics. As with inlining, an optimization coach can report cases where the case-of-case transformation is possible but ultimately not performed, along with the cause of the failure.

12. Conclusion

In this paper, we identify an unfilled niche in the programming ecosystem—feedback from the compiler's optimizer concerning successes and failures of specific optimizing transformations. Currently, if a programmer wishes to understand how the compiler views a program, the best option is to study the compiler's generated code. We propose an alternative technique: optimization coaching. Our first optimization coach operates at compile time and provides detailed, program-specific information about the optimizer's actions and omissions.

In addition to reporting optimization successes and failures, our tool generates recommendations as to what changes would allow optimizations. By applying these suggestions to a variety of programs, we show that optimization coaching can automatically identify crucial optimization opportunities. Reacting to its recommendations can often recover many of the performance gains available to manual optimization by an expert.

Acknowledgments We gratefully acknowledge Matthew Flatt's help with Racket's optimizer. Robby Findler suggested working with the Racket inliner. The OOPSLA reviewers, including James Noble, helped us hone our ideas.

Software Optimization Coach is available in version 5.3 of DrRacket (August 2012) at `http://racket-lang.org`.

Bibliography

Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. ORBIT: an optimizing compiler for Scheme. In *Proc. Symp. on Compiler Construction*, 1986.

Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 739–753, 2010.

Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In *Proc. European Conf. on Object-Oriented Programming*, pp. 172–196, 2004.

Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Proc. Symp. on Principles of Programming Languages*, pp. 293–302, 1989.

David Binkley, Bruce Duncan, Brennan Jubb, and April Wielgosz. The FeedBack compiler. In *Proc. International Works. on Program Comprehension*, pp. 198–206, 1998.

John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proc. European Symp. on Programming*, pp. 320–334, 2001.

Ryan Culpepper and Matthias Felleisen. Debugging hygienic macros. *Science of Computer Programming* 75(7), pp. 496–515, 2010.

Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. European Conf. on Object-Oriented Programming*, pp. 77–101, 1995.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *J. of Functional Programming* 12(2), pp. 159–182, 2002.

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 23–32, 1996.

Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

Agner Fog. Software optimization resources. 2012. `http://www.agner.org/optimize/`

Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

Saturnino Garcia, Donghwan Jeon, Chris Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *Proc. Programming Language Design and Implementation*, pp. 458–469, 2011.

Brian Hackett and Shu-Yu Guo. Fast and precise type inference for JavaScript. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 239–250, 2012.

William von Hagen. *The Definitive Guide to GCC*. Apress, 2006.

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 77–88, 2012.

Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me of you can: performance bug detection in the wild. In *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 155–170, 2011.

Donald E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience* 1, pp. 105–133, 1971.

Xavier Leroy. Unboxed objects and polymorphic typing. In *Proc. Symp. on Principles of Programming Languages*, pp. 177–188, 1992.

Peter A. W. Lewis, A. S. Goodman, and J. M. Miller. A pseudo-random number generator for the System/360. *IBM Systems Journal* 8(2), pp. 136–146, 1969.

Jennifer Lhoták, Ondřej Lhoták, and Laurie J. Hendren. Integrating the Soot compiler infrastructure into an IDE. In *Proc. Symp. on Compiler Construction*, pp. 281–297, 2004.

Stephen S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann, 1997.

Simon L Peyton Jones. Compiling Haskell by program transformation a report from the trenches. In *Proc. European Symp. on Programming*, pp. 18–44, 1996.

Feng Qian, Laurie J. Hendren, and Clark Verbrugge. A comprehensive approach to array bounds check elimination for Java. In *Proc. Symp. on Compiler Construction*, pp. 325–342, 2002.

Manuel Serrano. Inline expansion: when and how. In *Proc. International Symp. on Programming Language Implementation and Logic Programming*, pp. 143–147, 1997.

Zhong Shao and Andrew Appel. A type-based compiler for standard ML. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 116–129, 1995.

Jennifer Elizabeth Shaw. *Visualisation Tools for Optimizing Compilers*. MS dissertation, McGill University, 2005.

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *Proc. International Symp. on Practical Aspects of Declarative Languages*, pp. 289–303, 2012.

The Free Software Foundation. *GCC 4.7.0 Manual*. 2012.

The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.4.1*. 2011.

The SBCL Team. *SBCL 1.0.55 User Manual*. 2012.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. Programming Language Design and Implementation*, pp. 132–141, 2011.

Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *Proc. Symp. on Compiler Construction*, pp. 18–34, 2000.

Michael Wolfe. Loop skewing: the wavefront method revisited. *J. of Parallel Programming* 15(4), pp. 279–293, 1986.

Nicholas C. Zakas. *High Performance JavaScript*. O'Reilly, 2010.