Profile-Guided Meta-Programming

Anonymous

Abstract

Modern compilers such as GCC, .NET, and LLVM incorporate profile-guided optimizations (PGOs) on low-level intermediate code and basic blocks, with impressive results over purely static heuristics. Recent work shows that profile information is also useful for performing source-to-source optimizations via meta-programming. For example, using profiling information to inform decisions about data structures and algorithms can potentially lead to asymptotic improvements in performance. General-purpose meta-programming systems should makes profile information available to to enable meta-programmers to write their own, potentially domain-specific optimizations. Currently, each existing profile-guided meta-program comes with its own special-purpose toolkit, specific to a single optmization, creating barriers to use and development of new profile-guided meta-programs.

We propose an approach for supporting profile-guided meta-programs in a general-purpose meta-programming system. Our approach is parametric over particular profilings technique and meta-programming systems. We demonstrate our approach by implementing it in two different (albeit similar) meta-programming system: the syntactic extensions systems of Chez Scheme and Racket.

1. Introduction

Profile-guided optimization (PGO) is a compiler technique in which the compiler uses profile information, e.g., execution counts, gathered from test runs on representative sets of inputs to generate more efficient code. The profile information acts as an oracle for runtime behavior, and is used to inform low-level optimization decisions about, e.g., reordering basic blocks, inlining, reordering conditional branches, and function layout (Gupta et al. 2002). The resulting code usually exhibits improved performance, at least on the represented class of inputs, than code generated with static optimization heuristics. For example, using profiling information to inform inlining decisions in Java resulted in up to 59% improvement over static heuristics (Arnold et al. 2000). Modern compilers that support PGO include .NET (Profile-Guided Optimizations 2013), GCC (Optimize Options - Using the GNU Compiler Collection 2013), and LLVM (Lattner 2002).

Profile information has also proven useful to implement profile-guided meta-programs. Meta-programs, i.e., programs that operate on programs, are used to implement high-level abstractions such as abstract libraries like Boost (Dawes and Abrahams 2009), compiler generators (W. Keep and Kent Dybvig 2013), domain specific languages (Flatt et al. 2009; K. Sujeeth et al. 2013), and even whole general purpose languages (Barzilay and Clements 2005; Rafkind and Flatt 2012; Tobin-Hochstadt and Felleisen 2008; Tobin-Hochstadt et al. 2011). Languages with existing meta-programming systems include C, C++, Haskell (Sheard and Peyton Jones 2002), Java (Erdweg et al. 2011), ML (Sheard and Jones 2002; Taha and Sheard 2000), Racket (Flatt and PLT 2010), Scheme (Dybvig et al. 1993), and Scala. (Burmako 2013).

TODO: Should add some references to Delite, Scala work. Maybe in the related work

Using profile-guided meta-programming, Chen et. al. implement a profile-guided meta-program for performing process placement for SMP clusters (Chen et al. 2006). Liu and Rus provide a tools that uses profile information to identify suboptimal usage of the C++ STL (Liu and Rus 2009). Hawkins et. al. implement a compiler for a language that generates C++ implementations of data structures based on high-level specifications (Hawkins et al. 2011; Hawkins et al. 2012).

Existing general-purpose meta-programming systems do not provide profile information about the input programs on which the meta-program is operating. Therefore, existing profile-guided meta-programs introduce new special-purpose toolkits for profiling and meta-programming. Instead, meta-programmers need an approach that gives existing general-purpose meta-programming systems access to profile information. Meta-programmers could then implement the profile-guided meta-programs, reusing the meta-programming and profiling tools of an existing, familiar, system. Programmers could then take advantage of all the optimizations implemented in that system.

We propose an approach for supporting profile-guided meta-programming in a single general-purpose system. The approach provides a simple API through which meta-programs can access fine-grain source-level profile information, and does not interfere with traditional, i.e., "low-level" PGOs.

TODO: Say something about profiling information being available for run-time decisions, in later sections

We implement this approach in Chez Scheme using standard and efficient block-level profiling techniques (Ball and R. Larus 1994) (Burger and Dybvig 1998). We also implement this approach in Racket (Flatt and PLT 2010) purely as a library, using preexisting profiling tools.

The remainder of the paper is organized as follows. In section 2 we present a running example and introduce meta-programming in Scheme. In section 3 we present the design of our system at a high-level, and the implementation for both Chez Scheme and Racket. In section 4 we demonstrate that our approach is general enough to implement and extend existing profile-guided optimizations and profile-guided meta-programs. In section 5 we related to existing work on PGOs and profile-guided meta-programming. We conclude in section 6 a discussion of how our approach could be implemented in other meta-programming systems.

The main contributions of the paper are:

- A general approach for profile-guided meta-programming.
- An evaluation of this approach based on implementing existing profile-guided meta-programs, using our approach.

TODO: Don't like the phrasing of that second bullet

2. A running example

We first introduce a Scheme syntax extension which we use as a running example and to familiarize readers with Scheme and Racket flavor meta-programming.

```
(define-syntax (if-r stx)
  (syntax-case stx ()
    [(if-r test t-branch f-branch)
     ; Rewrites to
     (let ([t-prof (profile-query #'t-branch)]
           [f-prof (profile-query #'f-branch)])
       (if (< t-prof f-prof)
           ; if false branch is more frequent:
           #'(if (not test) f-branch t-branch)
           ; if true branch is more frequent:
           #'(if test t-branch f-branch)))]))
```

Figure 1: Example syntax extension

In figure 1, `define-syntax` introduces a new syntax extension `if-r`. Any uses of `if-r` in the source will be rewritten using the code in the body of the extension. For example, the function in figure 2 uses `if-r`. The extension will receive the argument:

```
#'(if-r (subject-contains-ci email "PLDI")
        (flag email 'important)
        (flag email 'spam))
```

This is a data representation of a term called a syntax object. The forms `#'`, `#`` , and `#,` implement Lisp's quote, quasiquote, and unquote but on syntax instead of lists. `syntax-case` performs pattern matching on syntax objects.

```
; Source code
(define (classify email)
  (if-r (subject-contains email "PLDI")
        (flag email 'important) ; Run 5 times
        (flag email 'spam)))    ; Run 10 times

; Code generated from expanding syntax extension
(define (classify email)
  (if (not (subject-contains email "PLDI"))
      (flag email 'spam)
      (flag email 'important)))
```

Figure 2: Using `if-r`

At compile time, `if-r` it looks up the profile information attached to each branch, reorders the branches based on which is more likely to be executed. It uses `profile-query` to access the profile information for each branch and generate an `if` expression. This transformation is not a meaninful optimizaion and is only used for illustrative purposes. The syntax extension is expanded at compile-time, while the resulting `if` will be run at run-time.

3. Approach and API

This section presents the high-level design of our approach, discusses design decisions, and presents a sketch of the API provided to meta-programmers. We conclude section with a brief description of our implementations of this approach in Chez Scheme and Racket.

3.1 Profile Information

While our approach is not specific to a particular profile technique, we refer to counter-based profile information to simplify explanations. We consider that profile point identified by the meta-program has a unique counter associated with it

Absolute profile information such as exact counts are difficult to compare across different data sets. Multiple data sets are important to ensure PGOs can optimize for multiple classes of inputs expected in production. Instead, our API provides profile *weights*. The profile weight of a profile point in a given data set is the ratio of the absolute profile information for that profile point to the maximum of all other profile point, represented as a number in the range [0,1]. In the case of counters, the profile weight for a given profile point is count for that point divided by the the count for the most frequently executed profile in the data set. This provides a single value identifying the relative importance of an expression and simplifies the combination of multiple profile data sets.

TODO: Still want to say something about percent-of-max vs percent-of-

To demonstrate profile weights, consider the running example from figure 2. Suppose in the first data set, `(flag email 'important)` runs 5 times and `(flag email 'spam)` runs 10 times. In the second data set, `(flag email 'important)` runs 100 times and `(flag email 'spam)` run 10 times. Figure 3 shows the profile weights computed after each data set.

```
;; After loading data from data set 1
(flag email 'important)→ 5/10              ;; 0.5
(flag email 'spam)     → 10/10             ;; 1

;; After loading data from data sets 1 and 2
(flag email 'important)→ (.5 + 100/100)/2 ;; 0.75
(flag email 'spam)     → (1 + 10/100)/2   ;; 0.55
```

Figure 3: Sample profile weight computations

3.2 Source objects

TODO: I don't like 'profile points'

To store and access profile information, we need to associate profile points with profile information. We use *source objects* (Dybvig et al. 1993) to uniquely identify profile points. Source objects are typically introduced by the lexer and parser for a source language and maintained throughout the compiler to correlate source expressions with intermediate or object code. This enables both error messages and debuggers to refer to source expressions instead of target or intermediate representations.

We reuse source objects in our approach to uniquely identify profile points. As source objects uniquely identify every source expression, the provide very fine-grain profile information. If two expressions are associated with the same source object, then they both increment the same profile counter when executed. Conversely, if two expressions are associated with different source objects, then they increment different profile counters when executed.

While source objects are typically introduced by the lexer and parser, we also require the ability to create new source objects in meta-programs. This is useful, for instance, when implementing a DSL. You may want to profile generated expressions separately from any other expression in the source language.

In the case of our running example, the lexer and parser introduce source objects for each expression (and subexpression). That is, separate source objects are created for `#'(if ...)`, `#'(subject-contains-ci email "PLDI")`, `#'subject-contains-ci`, `#'email`, `#'"PLDI"`, `#'(flag email 'spam)`, and so on. Note that `#'flag` and `#'email` appear multiple times, and will have a unique source object for each occurrence.

When loading profile information from a previous run, we compute profile weights and store them in an associative map from source objects to profile weights. The meta-programmer can access this information using an API call, such a `profile-query` in our running example.

3.3 Efficient Instrumentation

Adding a profile point for every single source expression requires care to instrument correctly and efficiently. This section describes efficient instrumentation techniques that preserve profile points.

As expressions are duplicated or thrown out during optimization, the profile points must not be duplicated or lost. The language implementation should consider profile points as effectful expressions, such as an assignment to an external variable, that should never be lost or duplicated. For example, for every profile point the language could generate an expression `(profile src)`, where `src` is the source object for that profile point, and never lose or duplicate `(profile src)` expressions.

To instrument profiling efficiently, `profile` expressions can be preserved until generating basic blocks. While generating basic blocks, all the source objects from the profile expressions can be attached to the basic block in which they appear. After associating profile points with basic blocks, we can analyze the blocks to determine which profile points can be calculated in terms of other profile points. Instead of emitting counters for each profile points, some counters are computed as the sum of a list of other counters. This technique reduces the number of counter increments to at most one per block, and fewer in practice, using techniques from Burger and Dybvig (Burger and Dybvig 1998).

The above infrastructure is also useful for instrumenting block-level profiling. Recall that we can generate new source objects. When generating basic blocks, we attach a newly generated source objects for that block. The source objects need to be generated deterministically to remain stable across different runs.

TODO: Not sure how much of this is actually relevant to profiling techniques other than counter based. I can imagine timing the entry to a block, updating timers based on these profile forms...

3.4 Source and Block PGO

One goal of our approach is to complement rather than interfere with traditional, e.g., basic block-level PGO. However, since meta-programs may generate different source code after optimization, the low-level representation will change after meta-programs perform optimizations. Therefore we need to instrument and perform source and basic block-level optimizations separately. We describe a workflow for our approach via the running example from figure 2.

TODO: Lots of 'we'

To get both source-level and block-level optimizations, we first instrument the program for source profiling. After running it on representative inputs, we get the profile weights such as in figure 3. Next we recompile using that source profile information, and instrument profiling for basic blocks. The generated source code, figure 2, will remain stable as long as we continue to optimize using the source profile information. Since the generated source code remains stable, so do the generated basic blocks. Now we profile the basic blocks generated from the optimized source program. Fi-

```
(run (instrument-source "spam-filter.rkt")
     input1 input2)
(load-source-profile-info)
(run (instrument-blocks
      (optimize-source "spam-filter.rkt"))
     input1 input2)
(load-source-profile-info)
(load-block-profile-info)
(compile (optimize-source "spam-filter.rkt"))
```

Figure 4: This workflow in code

nally, we use both the source profile information and the basic block profile information to do profile-guided optimizations via meta-programming and traditional low-level optimizations

3.5 Complete API sketch

Here we sketch a specification for a complete API.

```
; Takes:   A syntax or source object
; Returns: A profile weight
(define (profile-query-weight x) ...)

; Takes: The name of a file contains profile
;   information
; Returns: Nothing. Updates an associative map
;   accessible by profile-query-weight
(define (load-profile-info filename) ...)

; Takes: A file name
; Returns: Nothing. Saves information from the
;   profiling system to filename
(define (save-profile-info filename) ...)

; Takes: A syntax object and a source object
; Returns: The given syntax object modified to
;   be associated with the given source object
(define (annotate-syntax syn src) ...)

; Takes: A syntax object
; Returns: A deterministically generated but
;   fresh source object, possibly based on the
;   source of the given syntax object
(define (make-source-obj syn) ...)
```

3.6 Chez Scheme implementation

In Chez Scheme, a source object contains a file name and starting and ending character positions. The Chez Scheme reader automatically creates and attaches these to each syntax object read from a file. Chez Scheme also provides an API to programmatically manipulate source objects and attach them to syntax (Dybvig 2011 Chapter 11).

We generate a new source objects by adding a suffix to the file name of a base source object. We generated the suffix from a user provided string and an internally incremented counter. By basing generated source object on source objects from the original source program, errors in generated code are easier to debug as the generated code contains source file information from the meta-program that generated the code.

Chez Scheme implements exact counter based profiling using the instrumentation techniques described in section 3.3, including the basic block instrumentation techniques to perform traditional PGOs.

The meta-programming system's runtime maintains an associative map of source objects to profile weights which is updated by API calls. The API provide by Chez Scheme nearly identical to the one sketched in section 3.5.

3.7 Racket implementation

Racket does not attach separate source objects to syntax. Instead, the reader attaches the file name, line number, column number, position, and span are all attached directly to the syntax object. Racket provides functions for attaching source information when building a new syntax object. We provide wrappers to extract source information into separate source objects, and to merge source objects into Racket syntax objects. We then generate source objects in essentially the same way as in Chez Scheme.

We use one of the pre-existing Racket profiling implementation. The `errortrace` library provides exact profile counters, like the Chez Scheme profiler.

We implement a library which provides a similar API to the one sketched in section 3.5. This library maintains the map from source objects to profile information and computers profile weights. This library is implemented as standard Racket library that can be called at by meta-program, and requires no changes to either the Racket implementation or the `errortrace` library.

4. Case Studies

In this section we evaluate our approach. We show it is general enough to implement and improve upon existing profile-guided meta-programs. We first demonstrate optimizing Scheme's `case` construct, a multi-way branching construct similar to C's `switch`. Then we then demonstrates profile-guided receiver class prediction (Grove et al. 1995) for an object-oriented DSL. Finally we demonstrate that our approach is powerful enough to reimplement and improve upon Perflint (Liu and Rus 2009) by providing a list and vector libraries that warn programmers when they may be using a less than optimal data structure, based on profile information. We provide implementations of all case studies in both Chez Scheme and Racket.

4.1 Profile-guided conditional branch optimization

The .NET compiler features value probes, which enable profile-guided reordering of if/else and `switch` statements. As our first case study, we optimize Scheme's `cond` and `case` constructs, which are similar to if/else and `switch` in other languages. This demonstrates that our approach can be used to easily implement this optimization without the specialized support of value probes. It also demonstrates that our approach allows programmers to use meta-programming to

encode their knowledge of the program, enabling optimizations that may have been otherwise impossible.

The Scheme `cond` construct is analogous to a series of if/else-if statements. To execute a `cond`, we run the left-hand side of each clause until some left-hand side evaluates to true. When we find the first true clause, we run the right-hand side of that clause, and ignore further clauses. If there is an `else` clause, we run the right-hand side of the `else` clause only if no other clause's left-hand side is true. Figure 5 shows an example program using `cond`.

```
(define (lex char)
 (cond
  [(is-whitespace? char) e1]
  [(is-open-parn? char) e1]
  [else (* n (fact (sub1 n)))]))
```

Figure 5: An example using `cond`

We introduce the `exclusive-cond` construct, figure 6, as a condition branch construct similar to `cond`, but one that expects all branches to be mutually exclusive. When the branches are mutually exclusive we can safely reorder the clauses to execute the most likely clauses first. While the compiler cannot prove such a property in general, meta-programming allows the programmer to encode this knowledge in their program and take advantage of optimizations that would have otherwise been impossible.

The `exclusive-cond` macro rearranges clauses based on the profiling information of the right-hand sides. Since the left-hand sides are executed depending on the order, profiling information from the left-hand side cannot be used to determine which clause is executed most often. The `clause` structure stores the original syntax for `exclusive-cond` clause and the weighted profile count for that clause. Recall that `profile-query-weight` may return `#f`. We do not care to distinguish between 0 and `#f`, so we use 0 when if we there is no profile information. Since a valid `exclusive-cond` clause is also a valid `cond` clause, we copy the syntax and generate a new `cond` in which the clauses are sorted according to profile weights. The `#,@` splices a list of syntax objects into a syntax object. Of course the `else` clause is always last and is not included when sorting the other clauses.

The `case` construct takes an expression `key-expr` and an arbitrary number of clauses, followed by an optional `else` clause. The left-hand side of each clause is a list of constants. `case` executes the right-hand side of the first clause in which `key-expr` is `eqv?` to some element of the left-hand. If `key-expr` is not `eqv?` to any element of any left-hand side and an `else` clause exists then the right-hand side of the `else` clause is executed. Figure 7 shows an example `case` expression. In this example, the programmer has a spurious `0` in the second clause which should never be matched against, since the first clause will always match `0`.

```
(define (fact n)
 (case n
  [(0) 1]
  [(0 5) 120]
  [else (* n (fact (sub1 n)))]))
```

Figure 7: An example using `case`

Figure 8 shows the full profile-guided implementation of `case` that sorts clauses by which is executed most often. The majority of the work is in `trim-keys!`, which removes duplicate keys to ensure mutually exclusive clauses. We omit its definition for brevity. Since `case` permits clauses to have overlapping elements and uses order to determine which branch to take, we must remove overlapping elements before reordering clauses. We parse each clause into the set of left-hand side keys and right-hand side bodies. We remove overlapping keys by keeping only the first instance of each key when processing the clauses in the original order. After removing overlapping keys, we generate an `exclusive-cond` expression.

Figure 9 shows how the example `case` expression from figure 7 expands into `exclusive-cond`. Note the duplicate `0` in the second clause is dropped to preserve ordering constraints from `case`.

```
(define (fact n)
 (let ([x n])
  (exclusive-cond x
   [(memv x '(0)) 1]
   [(memv x '(5)) 120]
   [else (* n (fact (sub1 n)))])))
```

Figure 9: The expansion of figure 7

Finally, figure 10 show the result of expanding `exclusive-cond` in figure 9. In the final generated program, the most common case is checked first.

```
(define (fact n)
 (let ([x n])
  (cond x
   [(memv x '(5)) 120]
   [(memv x '(0)) 1]
   [else (* n (fact (sub1 n)))])))
```

Figure 10: The expansion of figure 9

```
(define-syntax (exclusive-cond x)
  (define-record-type clause (fields syn weight))
  (define (parse-clause clause)
    (syntax-case clause ()
      [(e0 e1 e2 ...) (make-clause clause (or (profile-query-weight #'e1) 0))]
      [_ (syntax-error clause "invalid clause")]))
  (define (sort-clauses clause*)
    (sort (lambda (cl1 cl2) (> (clause-weight cl1) (clause-weight cl2)))
     (map parse-clause clause*)))
  (define (reorder-cond clause* els?)
    #`(cond #,@(map clause-syn (sort-clauses clause*)) . #,els?))
  (syntax-case x (else)
    [(_ m1 ... (else e1 e2 ...)) (reorder-cond #'(m1 ...) #'([else e1 e2 ...]))]
    [(_ m1 ...) (reorder-cond #'(m1 ...) #'())]))
```

Figure 6: Implementation of `exclusive-cond`

```
(define-syntax (case x)
  (define (helper key-expr clause* els?)
    (define-record-type clause (fields (mutable keys) body))
    (define (parse-clause clause)
      (syntax-case clause ()
        [((k ...) e1 e2 ...) (make-clause #'(k ...) #'(e1 e2 ...))]
        [_ (syntax-error "invalid case clause" clause)]))
    (define (emit clause*)
      #`(let ([t #,key-expr])
          (exclusive-cond
            #,@(map (λ (cl) #`[(memv t '#,(clause-keys cl)) #,@(clause-body cl)]) clause*)
            . #,els?)))
    (let ([clause* (map parse-clause clause*)])
      (for-each trim-keys! clause*) (emit clause*)))
    (syntax-case x (else)
      [(_ e clause ... [else e1 e2 ...]) (helper #'e #'(clause ...) #'([else e1 e2 ...]))]
      [(_ e clause ...) (helper #'e #'(clause ...) #'())]))
```

Figure 8: Implementation of `case` using `exclusive-cond`

4.2 Profile-guided receiver class prediction

We provide a meta-program that implements profile-guided receiver class prediction (Grove et al. 1995; Hölzle and Ungar 1994) for a simplfied object-oriented DSL implemented as a syntax extension. This case study demonstrates that our mechanism is both general enough to implement well-known profile-guided optimizations, and powerful enough to provide DSL writers with standard PGOs.

Figure 11 shows the key parts of our implementation of recevier class prediction. A method call such as `(method shape area)` will generate code as follows. First, we generate a new source object for each class in the system. Then we instrument a call to the dynamic dispatch routine for each of the newly generated source objects. When there is no profile data, we expand into a `cond`[1] that calls the instrumented version of the dynamic dispatch depending on the class of the object `shape`. When there is profiling information, we expand into a `cond` that tests for the most frequently used classes at this method call site, and inlines thost methods. Otherwise we fall back to dynamic dispatch.

TODO: Maybe implement the instrumented hash table later

The entire implementation of profile-guided receiver class prediction is 44 lines of code. The rest of the OO DSL implementation requires an additional 82 lines.

Figure 12 shows an example method call, the resulting code after instrumentation, and the resulting code after optimization. Note that the each occurences of `(instrumented-dispatch x area)` has a different source objects, so they are each profiled separately.

We have demonstrated that our approach can easily implement a well known profile-guided optimization as a meta-program, but our approach provides one additional advantage. We can reuse `exclusive-cond` to test for the most likely class first.

[1] A production implementation would create a table of instrumented dynamic dispatch calls and dynamically dispatch through this table, instead of instrumenting code with `cond`.

```
(begin-for-syntax
  (define make-fresh-source-obj! (make-fresh-source-obj-factory! "method-call")))
(define-syntax (method syn)
  (define source-objs (map (λ (x) (make-fresh-source-obj! syn)) class-list))
...
  (syntax-case syn ()
    [(_ obj m val* ...)
    (let ([instrumented-dispatchs (for/list ([source-obj source-objs])
                                    (instrument-dispatch source-obj #'m #'(val* ...)))]
          [sorted-classes (drop-zero-weight (sort-by-weight source-objs class-list))]))
...
     #`(let* ([x obj])
         (cond
           #,@(if no-profile-data?
                  (for/list ([d instrumented-dispatchs] [class class-list])
                    #`((class-equal? x #,class) (#,d x)))
                  (for/list ([class (take sorted-classes inline-limit)])
                    #`((class-equal? x #,class)
                       #,(inline-method class #'x #'m #'(val* ...)))))
           [else (dynamic-dispatch obj m val* ...)])))))
```

Figure 11: Implementation of profile-guided receiver class prediction

```
(class Square
  ((length 0))
  (define-method (area this)
    (sqr (field this length))))
(class Circle
  ((radius 0))
  (define-method (area this)
    (* pi (sqr (field this radius)))))
(class Triangle
  ((base 0) (height 0))
  (define-method (area this)
    (* 1/2 base height)))
...
(for/list ([s (circle-and-squares)])
  (method s area))

; ------------------
; After instrumentation
...
(let* ([x c])
  (cond
    [(class-equal? x 'Square) (instrumented-dispatch x area)]
    [(class-equal? x 'Circle) (instrumented-dispatch x area)]
    [(class-equal? x 'Triangle) (instrumented-dispatch x area)]))

; ------------------
; After optimization
...
(let* ([x c])
  (cond
    [(class-equal? x 'Square) (let ([this x]) (sqr (field x length)))]
    [(class-equal? x 'Circle) (let ([this x]) (* pi (sqr (field x radius))))]
    [else (dynamic-dispatch x area)]))
```

Figure 12: Example of profile-guided receiver class prediction

```
; ------------------
; After optimization
...
(let* ([x c])
  (exclusive-cond
    [(class-equal? x 'Square) (let ([this x]) (sqr (field x length)))]
    [(class-equal? x 'Circle) (let ([this x]) (* pi (sqr (field x radius))))]
    [else (dynamic-dispatch x area)]))

; ------------------
; After more optimization
...
(let* ([x c])
  (cond
    [(class-equal? x 'Circle) (let ([this x]) (* pi (sqr (field x radius))))]
    [(class-equal? x 'Square) (let ([this x]) (sqr (field x length)))]
    [else (dynamic-dispatch x area)]))
```

Figure 13: Profile-guided recevier class prediction, sorted.

4.3 Data Structure Specialization

While profile-guided optimizations can provide important speeds up by optimizing code paths, programmers can us profile information to identify much higher level performance issues. For instance, profile information can be used to figure out that a different algorithms or data structures might be cause an asymptotic speed up. (Liu and Rus 2009) In this case study we show our mechanism is general enough to implement this kind of profiling tool, and even go beyond it by automating the recommendations.

We provide implementations of lists and vectors (array) that warn the programmer when they may be using a less optimal data structure. The implementations provide wrappers around the standard list and vector functions that introduce new source objects to profile the uses of each new list and vector separately. Finally, we provide an implementation of a sequence datatype that will automatically specialize to a list or vector based on profiling information. Complete versions of both Chez Scheme and Racket implementations of this code are freely available at (??? ???).

Figure 14 shows the implementation of the profiled list constructor. This constructor has the same interface as the standard Scheme list constructor—it takes an arbitrary number of elements and returns a representation of a linked list. We represent a list as a pair of the underlying linked list and a hash table of profiled list operations. We generate these profiled operations by simply wrapping calls to underlying, 'real', list operations with freshly generated source objects. We generate two source objects for each list. One is used to profile operations that are fast for lists and the other is used to profile operations that are fast for vectors. Finally, we export new versions of all the list operations that work on our new list representation. For instance, `car` takes our profiled list representations, and calls the profiled version of `car` from the hash table of the profiled list on the underlying list. When profiling information already exists, for instance, after a profiled run, this list constructor emits a warning (at compile time) if the list fast vector operations are more common than fast list operations.

We also provide an analogous implementation of vectors. While we implement only two data structures here, this technique should scale to the many other data structures analyzed by Perflint. However, we can do one step better. Since our meta programs are integrated into the language, rather than existing as a separate tool in front of the compiler, we can provide libraries to the programmer that automatically follow these recommendations rather than asking the programmer to change their code. To demonstrate this, we implement a profiled sequence data type that will automatically specialize to a list or vector, at compile time, based on profile information.

Figure 15 shows the implementation of the profiled sequence constructor. The code follows exactly the same pattern as the profiled list. The key difference is we conditionally generate wrapped versions of the list *or* vector operations, and represent the underlying data using a list *or* vector, depending on the profile information.

This implementation of an automatically specializing data structure is not ideal. The extra indirects through a hashtable and wrapped operations introduce constant overhead to constructing a sequence, and to every operation on the sequence. This case study does, however, demonstrate that our mechanism is general and powerful enough to implement novel profile directed optimizations.

5. Related and Future Work

5.1 Low-level PGO

TODO: Add links for all references.

Modern systems such as GCC, .NET, and LLVM use profile directed optimizations (Lattner 2002; Optimize Options - Using the GNU Compiler Collection 2013). These systems

```
(define-record list-rep (op-table ls))
(define (car ls)
  (make-list-rep (list-rep-op-table ls)
    ((hashtable-ref (list-rep-op-table ls) 'car #f)
     (list-rep-ls ls))))
...
(meta define make-fresh-source-obj! (make-fresh-source-obj-factory! "profiled-list"))
(define-syntax (list x)
    ; Create fresh source object. list-src profiles operations that are
    ; fast on lists, and vector-src profiles operations that are fast on
    ; vectors.
    (define list-src (make-fresh-source-obj! x))
    (define vector-src (make-fresh-source-obj! x))
    ; Defines all the sequences operations, giving profiled implementations
    (define op-name* '(list? map car cdr cons list-ref length))
    (define op*
      (real:map
        (lambda (v src)
          (datum->annotated-syntax x `(lambda args (apply ,v args)) src))
        '(real:list? real:map real:car real:cdr real:cons real:list-ref real:length)
        (real:list #f #f #f list-src list-src vector-src vector-src)))
    (syntax-case x ()
      [(_ init* ...)
       (unless (>= (or (profile-query-weight list-src) 0)
                   (or (profile-query-weight vector-src) 0))
         (printf "WARNING: You should probably reimplement this list as a vector: ~a\n" x))
       #`(let ()
           (make-list-rep
             (let ([ht (make-eq-hashtable)])
               #,@(real:map (lambda (op op-name) #`(hashtable-set! ht #,op-name #,op))
                    (syntax->list op*) (syntax->list op-name*))
               ht)
             (real:list init* ...)))]))
```

Figure 14: Implementation of profiled list

uses profile information to guide decisions about code positioning, register allocation, inlining, and branch optimizations.

GCC profiles an internal control-flow graph (CFG). To maintain a consistent CFGs across instrumented and optimization builds, GCC requires similar optimization decisions across builds (Chen et al. 2010). In addition to the common optimizations noted previously, .NET extends their profiling system to probe values in `switch` statements. They can use this value information to optimize `switch` branches, similar to the implementation of `case` we presented in section 4.1.

Our system supports all these optimizations and has several advantages. While .NET extends their profiling system to get additional optimizations, we can support all the above optimizations in a single general-purpose system. By using profile information associated with source expressions, we reduce reliance specific internal compiler decisions and make profile information more reusable. When there is no substitute for block-level information, such as when reordering basic blocks, we support both source and block profiling in the same system.

5.2 Dynamic Recompilation

The standard model for PGO requires the instrument-profile-optimize workflow. LLVM has a different model for PGO. LLVM uses a runtime reoptimizer that monitors the running program. The runtime can profile the program as it runs "in the field" and perform simple optimizations to the machine code, or call to an offline optimizer for more complex optimizations on the LLVM bytecode.

While not currently enabled, our mechanism supports this kind of reoptimization. We build on the work of Burger and Dybvig, who present an infrastructure for profile-directed dynamic reoptimization (Burger and Dybvig 1998). Their work shows just 14% run-time overhead for instrumented code, but they express concerns that dynamic recompilation will not overcome this cost. Our internal microbenchmarks show similar overhead. To enable dynamic PGO, we would need to modify our mechanism to automatically reload profile information, such as whenever `profile-query-weight` is called, instead of manually loading information from a file. This is a trivial change to our system, but we have no optimizations in mind that make use of profile-guided at runtime. It may also increase overhead, since we compute pro-

```
(define-record seq-rep (op-table s))
...
(meta define make-fresh-source-obj! (make-fresh-source-obj-factory!  "profiled-seq"))
(define-syntax (seq x)
   (define list-src (make-fresh-source-obj! x))
   (define vector-src (make-fresh-source-obj! x))
   (define previous-list-usage (or (profile-query-weight list-src) 0))
   (define previous-vector-usage (or (profile-query-weight vector-src) 0))
   (define list>=vector (>= previous-list-usage previous-vector-usage))
   (define op-name* '(seq? seq-map seq-first seq-rest seq-cons seq-append
     seq-copy seq-ref seq-set! seq-length))
   (define op*
     (map
       (lambda (v src)
         (datum->annotated-syntax x '(lambda args (apply ,v args)) src))
       (if list>=vector
           '(list? map first rest cons append list-copy list-ref
             list-set! length)
           '(vector? vector-map vector-first vector-rest vector-cons
             vector-append vector-copy vector-ref vector-set!
             vector-length))
       (list #f #f #f list-src list-src list-src #f vector-src vector-src
             vector-src)))
   (syntax-case x ()
     [(_ init* ...)
      #'(let ()
          (make-seq-rep
            (let ([ht (make-eq-hashtable)])
                #,@(map (lambda (op op-name) #'(hashtable-set! ht #,op-name #,op))
                        (syntax->list op*) (syntax->list op-name*))
                ht)
            (#,(if list>=vector #'list #'vector) init* ...)))]))
```

Figure 15: Implementation of profiled sequence

file weights and many counters when loading new profile data.

5.3 Meta-program optimizations

TODO: Revisit Hermit references for more examples of meta-programming??

Meta-programming has proven successful at providing higher levels of abstraction while still producing efficient code. Meta-programming has been used to implement abstract libraries (Dawes and Abrahams 2009)
, domain specific languages (Flatt et al. 2009; K. Sujeeth et al. 2013), and even whole general purpose languages (Barzilay and Clements 2005; Rafkind and Flatt 2012; Tobin-Hochstadt and Felleisen 2008; Tobin-Hochstadt et al. 2011). The HERMIT toolkit provides an API for performing program transformations on Haskell intermediate code before compiling, even allowing interactive experimentation (Farmer et al. 2012). These meta-programs can lose or obscure information during the translation into target-language code.

TODO: Add two related works sitting on my desk for profile-guided meta-programming.

We're not the first to realize this. Many meta-program optimizations exist. Tobin-Hochstadt et. al. implement the optimizer for Typed Racket as a meta-program (Tobin-Hochstadt et al. 2011). Sujeeth et. al. provide a framework for generated optimized code from DSLs (K. Sujeeth et al. 2013). Hawkins et. al. implement a compiler for a language that generates C++ implementations of data structures based on high-level specifications (Hawkins et al. 2011; Hawkins et al. 2012).

Even using profile information to perform optimizations in meta-programs is not new. Chen et. al. implement their own profile and meta-program tools to provide a profile-guided meta-program for performing process placement for SMP clusters (Chen et al. 2006). Liu and Rus provide a tools that uses profile information to identify suboptimial usage of the C++ STL (Liu and Rus 2009).

We support these works by providing a single, general-purpose mechanism in which we can implement new languages, DSLs, abstract libraries, and arbitrary meta-programs, all taking advantage of profile-guided optimizations. Further, we our mechanism reuses existing meta-programming and profiling facilities, rather than implementing new tools in front of the compiler.

5.4 More PGO

We have referred to past work on both low-level PGOs and profile-guided meta-programs. But the use of profile information is still an active area of research. Furr et. al. present a system for inferring types in dynamic languages to assist in debugging (Furr et al. 2009). Chen et. al. use profile information to reorganize the heap and optimize garbage col-

lection (Chen et al. 2006). Luk et. al. use profile information to guide data prefetching (Luk et al. 2002). Debray and Evans use profile information to compress infrequently executed code on memory constrained systems (Debray and Evans 2002).

With so many profile-guided optimizations, we need a general-purpose mechanism in which to implement them without reimplementing profiling, compiling, and meta-programming tools.

6. Conclusion

We have presented a general mechanism for profile-guided meta-program optimizations implemented in Scheme. While our mechanism should easily extend to other meta-programming facilities, we conclude by discussing precisely how other common meta-programming facilities need to be extended to use our mechanism.

Template Haskell, MetaOcaml, and Scala all feature powerful meta-programming facilities similar to Scheme's (Burmako 2013; Czarnecki et al. 2004; Dybvig et al. 1993; Sheard and Jones 2002; Taha and Sheard 2000). They allow executing arbitrary code at compile-time, provide quoting and unquoting of syntax, and provide direct representations of the source AST. Source objects could be attached to the AST, and `profile-query-weight` could access the source objects given an AST. These languages all appear to lack source profilers, however.

C++ template meta-programming does not support running arbitrary programs at compile time. This might limit the kinds of optimizations that could be implemented using C++ template meta-programming as it exists today. Many source level profilers already exist for C++, so the challenge is in implementing source objects and `profile-query-weight`. C++ templates offers no way to directly access and manipulate syntax, so it is not clear where to attach source objects.

C preprocessor macros do support using syntax as input and output to macros, but are very limited in what can be done at compile time. Adding directives to create, instrument, and read source profile points might be enough to support limited profile-guided meta-programming using C preprocessor macros.

Meta-programming is being used to implement high-level optimizations, generate code from high-level specifications, and create DSLs. Each of these can take advantage of PGO to optimize before information is lost or constraints are imposed. Until now, such optimizations have been implemented via toolchains designed for a specific meta-program or optimization. We have described a general mechanism for implementing arbitrary profile-guided meta-program optimizations, and demonstrated its use by implementing several optimizations previously implemented in seperate, specialized toolchains.

Bibliography

??? Code repository.

Matthew Arnold, Stephen Fink, Vivek Sarkar, and Peter F. Sweeney. A Comparative Study of Static and Profile-based Heuristics for Inlining. In *Proc. Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000. http://doi.acm.org/10.1145/351397.351416

Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16(4), pp. 1319–1360, 1994.

Eli Barzilay and John Clements. Laziness Without All the Hard Work: Combining Lazy and Strict Languages for Teaching. In *Proc. The 2005 Workshop on Functional and Declarative Programming in Education*, 2005. http://doi.acm.org/10.1145/1085114.1085118

Robert G. Burger and R. Kent Dybvig. An infrastructure for profile-driven dynamic recompilation. In *Proc. International Conference on Computer Languages, 1998.*, pp. 240–249, 1998. http://pdf.aminer.org/000/289/483/an_infrastructure_for_profile_driven_dynamic_recompilation.pdf

Eugene Burmako. Scala Macros: Let Our Powers Combine! In *Proc. of the 4th Annual Scala Workshop*, 2013.

Deheo Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming Hardware Event Samples for FDO Compilation. In *Proc. Annual IEEE/ACM international symposium on Code generation and optimization*, 8, pp. 42–52, 2010. http://hpc.cs.tsinghua.edu.cn/research/cluster/papers_cwg/tamingsample.pdf

Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided Proactive Garbage Collection for Locality Optimization. In *Proc. The 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006. http://doi.acm.org/10.1145/1133981.1134021

Krzysztof Czarnecki, John T O'Donnell, Jörg Striegntiz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Proc. Domain-Specific Program Generation* volume Springer Berlin Heidelberg., pp. 51–72, 2004. http://camlunity.ru/swap/Library/ComputerScience/Metaprogramming/Domain-SpecificLanguages/DSLImplementationinMetaOCaml,TemplateHaskellandC++.pdf

B. Dawes and D. Abrahams. Boost C++ Libraries. 2009. http://www.boost.org

Saumya Debray and William Evans. Profile-guided Code Compression. In *Proc. The ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002. http://doi.acm.org/10.1145/512529.512542

R. Kent Dybvig. Chez Scheme Version 8 User's Guide. 8.4 edition. Cadence Research Systems, 2011. http://www.scheme.com/csug8

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and symbolic computation* 5(4), pp. 295–326, 1993. http://pdf.aminer.org/001/006/789/syntactic_abstraction_in_scheme.pdf

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based Syntactic Language Extensibility. In *Proc. of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 391–406, 2011. http://www.informatik.uni-marburg.de/~seba/publications/sugarj.pdf

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Proc. ACM SIGPLAN Notices*, 2012.

Matthew Flatt, Eli Barzilay, and Robert Bruce Findler. Scribble: Closing the Book on Ad Hoc Documentation Tools. In *Proc. The 14th ACM SIGPLAN International Conference on Functional Programming*, 2009. http://doi.acm.org/10.1145/1596550.1596569

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

Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided Static Typing for Dynamic Scripting Languages. In *Proc. The 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009. http://doi.acm.org/10.1145/1640089.1640110

David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-guided receiver class prediction. In *Proc. The tenth annual conference on Object-oriented programming systems, languages, and applications*, 1995. http://doi.acm.org/10.1145/217838.217848

R. Gupta, E. Mehofer, and Y. Zhang. *Profile Guided Code Optimization*. 2002.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Data representation synthesis. In *Proc. ACM SIGPLAN Notices*, 2011.

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard,, and Mooly Sagiv. Concurrent data representation synthesis. In *Proc. The 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012. http://doi.acm.org/10.1145/2254064.2254114

Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proc. ACM SIGPLAN Notices*, 1994.

Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proc. The 12th International Conference on Generative Programming: Concepts \&\#38; Experiences*, 2013. http://doi.acm.org/10.1145/2517208.2517220

Chris Authors Lattner. LLVM: An infrastructure for multi-stage optimization. Master dissertation, University of Illinois, 2002.

Lixia Liu and Silvius Rus. Perflint: A context sensitive performance advisor for c++ programs. In *Proc. The 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss, P. Geoffrey Lowney, and Robert Cohn. Profile-guided Post-link Stride Prefetching. In *Proc. The 16th International Conference on Supercomputing*, 2002. http://doi.acm.org/10.1145/514191.514217

Optimize Options - Using the GNU Compiler Collection. 2013. http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html#index-fprofile_002duse-867

Profile-Guided Optimizations. 2013. http://msdn.microsoft.com/en-us/library/e7k32f4k(v=vs.90).aspx

Jon Rafkind and Matthew Flatt. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Proc. The 11th International Conference on Generative Programming and Component Engineering*, 2012. http://doi.acm.org/10.1145/2371401.2371420

Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002.

Time Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. ACM SIGPLAN workshop on Haskell*, 2002. http://research.microsoft.com/en-us/um/people/simonpj/Papers/meta-haskell/meta-haskell.pdf

Walid Taha and Time Sheard. MetaML and multi-stage programming with explicit annotations . *Theoretical Computer Science* 248((1 2)), pp. 211–242, 2000. http://www.cs.rice.edu/~taha/publications/journal/tcs00.pdf

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. The 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008. http://doi.acm.org/10.1145/1328438.1328486

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages As Libraries. In *Proc. The 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011. http://doi.acm.org/10.1145/1993498.1993514

Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proc. The 18th ACM SIGPLAN international conference on Functional programming*, 2013.