# Profile-Guided Meta-Program Optimization

Anonymous

## Abstract

Many contemporary compilers allow the use of profile information to guide various low-level optimizations. This is not the case for contemporary meta-programming systems, although profile information can have an even greater impact on the high-level optimizations performed by meta-programs. For example, a meta-program sometimes has control over the data structures and algorithms used by the generated code, and use of profiling information to select appropriate data structures and algorithms can potentially lead even to asymptotic improvements in performance.

This paper describes a general-purpose mechanism for supporting arbitrary profile-guided meta-program optimization. It makes profile information available at the granularity of arbitrary source points identified by the meta-program, while making use of standard and efficient block-level profile-instrumentation techniques. We have implemented the mechanism as part of Chez Scheme and Racket, with profile information made available via the syntactic abstraction facility through which each languages supports meta-programming. Our mechanism can be adapted to most meta-programming systems with compilers that support profiling.

## 1. Introduction

Meta-programs, or programs that operate on programs, are often used to implement high-level abstractions ranging from simple syntactic abstractions, to compiler generators, to domain-specific languages (DSLs). While meta-programs can be written for virtually any language, some languages have built-in support for meta-programming, including C, C++, Haskell, Scheme, ML, and Scala. (Burmako 2013; Czarnecki et al. 2004; Dybvig et al. 1993; Erdweg et al. 2011; Sheard and Jones 2002; Taha and Sheard 2000). Ideally, meta-programs would not be concerned with generating optimized code but instead leave that to the target-language compiler. However, information is sometimes unavoidably lost or obscured during the translation into the target-language program. For example, constraints on types, ranges, and effects can be lost, as can the lack of constraints on data representation, algorithms, and evaluation order. Optimizations that depend on the lost information cannot be performed by the target-language compiler and thus must be performed by the meta-program, if at all.

Profile-guided optimization (PGO) is a compiler technique in which profile information, e.g., execution counts, from test runs on representative sets of inputs is fed into the compiler to enable it to generate more efficient code. The resulting code usually exhibits improved performance, at least on the represented class of inputs, than code generated with static optimization heuristics. Compilers that support PGO include .NET, GCC, and LLVM (Lattner 2002; Optimize Options - Using the GNU Compiler Collection 2013) The profile information used by these compilers, such as execution counts of basic blocks or control flow graph nodes, is low-level compared to the source-language operated on by meta-programs, so the optimizations that use the profile information are also performed on low-level constructs. Common optimizations include reordering basic blocks, inlining decisions, conditional branch optimization, and function layout decisions (Gupta et al. 2002).

Profile information can have an even greater impact on meta-program optimizations. For example, a meta-program might select data structures and algorithms based on the frequency with which certain operations are performed, potentially even leading to improvements in asymptotic performance.

Existing techniques that use profile information for these kinds of meta-program optimizations introduce a custom toolchain, or expect the programmer to optimize code by hand. 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, but leaves it up to the programmer to take corrective action (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). These works implement

highly specific meta-programming or profiling systems to provide advanced optimizations. Yet no general-purpose mechanism has been proposed to date that makes profile information available to meta-programing systems for arbitrary optimizations.

This paper describes such a general-purpose mechanism. Our mechanism makes profile information available at the granularity of arbitrary source points identified by the meta-program. In the case of a meta-program implementing an embedded DSL, these could correspond to source expressions already present in the source-language program. In a manner similar to standard profile-guided optimization mechanisms, making use of our mechanism involves running the meta-program and compiler once to instrument the code, running the resulting executable one or more times on representative data to gather profile data, and running the meta-program and compiler a second time to generate the optimized code. During the second run of the meta-program, the meta-program retrieves the profile information associated with source points, and can use this information to inform transformations and optimizations. The profile information is also available to the target-language compiler to support the further optimizations at the target-language level.

Our implementation of this mechanism in Chez Scheme uses standard and efficient block-level profiling techniques and is potentially suitable for dynamic optimization of a running program in systems that support dynamic recompilation (Burger and Dybvig 1998). It enables using data sets from multiple executions of the instrumented program, and does not interfere with traditional ("low-level") PGO. We implement this mechanism as part of a high performance Scheme system, with profile information made available via an API accessible from the high-level syntactic abstraction facility through which Scheme supports meta-programming, and even accessible at run-time. It should be straightforward to adapt to most meta-programming systems with compilers that already support profiling.

We also reimplement this mechanism, and all our examples, in the Racket programming langauge (Flatt and PLT 2010). While the meta-programming facilities provided in Racket are similar to those of Chez Scheme, the language implementation and profiling systems are entirely different. We're able to reimplement our work by reusing the existing Racket profiling and meta-programming infrastructure without making any changes to the language implementation.

The remainder of the paper is organized as follows. Section 2 presents the design of our system at a high level. Section 3 demonstrates how to use our mechanism to implement several optimizations as meta-programs. These examples demonstrate how our work can be used to implement and build on past work in a single, general system. In particular, we show how our work could be used to automate the recommendataions produced by Liu and Rus by automatically specialize an abstract sequence datatype (Liu and Rus 2009). We also demonstrate how to implement profile-guided receiver class (Grove et al. 1995) prediction using our mechanism. Section 4 discusses our implementation and how it works with traditional PGOs. Section 5 discusses PGO and meta-programming in more detail. We conclude in section 6 with a discussion of how our work could be implemented in other meta-programming systems.

## 2. Design

This section presents the essential concepts required to understand examples, and design decisions of our mechanism. We first discuss how source points are identified and manufactured. We then discuss what profile information we use and how we handle multiple data sets. We delay giving implementation details until section 4.

In a typical meta-programming situation, a meta-program takes as input a *source program* in a high-level domain-specific language (DSL) and produces a *target program* in some other language, e.g., C, Haskell, or Scheme. To perform arbitrary meta-program optimizations, we might require profile information for arbitrary points in the source program, arbitrary points in the target program, or both. We use *source objects* (Dybvig et al. 1993) to uniquely identify these points, and maintains a database associating source objects with profile information, whenever profile information from earlier program runs has been supplied.

### 2.1 Source objects

Source objects are typically introduced by the lexer and parser for a source language and maintained throughout the compiler to correlate source with intermediate or object code, enabling both compile-time source-error messages and source-level debugging, among other things. While the source objects created by the lexer and parser might encapsulate, e.g., a source file descriptor and character range for a specific source expression, source objects can contain other or different information. Meta-programs can make use of this to manufacture new source objects representing unique points in the target program, perhaps based on corresponding points in the source program.

We use source objects in our mechanism to uniquely identify profile counters. 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. We also use the ability to manufacture new source objects to introduce new profile counters. For instance, in section 3.4, we use this to generate new profile counters for each instance of a data structure.

### 2.2 Profile weights

Instead of using exact counts in the profile database, we use *weights*. The profile weight of a source point in a given

data set is the ratio of the exact count for the source point and the maximum count for any source point, represented as a floating-point number in the range [0,1]. That is, the profile weight for a given source object is profile count for that source object divided by the the profile count for the most frequently executed source object in the database. This provides a single value identifying the relative importance of an expression and simplifies the combination of multiple profile data sets.

We considered using absolute counts, but this complicates the combination of multiple data sets, since absolute counts from one run to the next are not directly comparable. We also considered using ratios of individual counts to total or average counts. In both cases, the results are distorted when there are a few heavily executed expressions, potentially leading to difficulty distinguishing profile weights for two less frequently executed expressions. We also considered using fixed-precision rather than floating point, but the floating-point representation makes it easy to determine the importance of a particular expression overall while still providing substantial precision when comparing the counts for source points with similar importance.

To understand how we compute profile weights, consider a program with two loops, **A** and **B**. If **A** is executed 5 times, and **B** is executed 10 times, we store **A** → `5/10 = 0.5` and **B** → `10/10 = 1`. To support multiple data sets, we simply compute the average of these weights. For instance, if in a second data set **A** is executed 100 times and **B** is executed 10 times, then **A** → `((5/10) + (100/100))/2 = 0.75` and **B** → `((10/10) + (10/100))/2 = 0.55`.[1] Multiple data sets enable reuse and help the developer collect representative profile data. This is important to ensure our PGOs can optimize for multiple classes of inputs expected in production.

## 3. Examples

This section demonstrates how to use our mechanism, and how it generalizes and advances past work on 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 how our mechanism is powerful enough to reimplement Perflint (Liu and Rus 2009). We provide list and vector libraries that warn programmers when they may be using a less than optimal data structure, and even provide a version that makes the choice automatically, based on profile information. Complete versions of all examples are freely available at (??? ???). Racket implementations exist for all examples for those without access to Chez Scheme.

[1] TODO: Diagram

### 3.1 Scheme macro example

Our mechanism and examples are implemented in Scheme, so we give below a simple example to introduce Scheme meta-programming and its syntax.

```
; Defines a macro (meta-program) 'do-n-
times'
; Example:
; (do-n-times 3 (display "*")) expands into
; (begin (display "*")
;        (display "*")
;        (display "*"))
(define-syntax (do-n-times stx)
  ; pattern matches on the inputs syntax
  (syntax-case stx ()
    [(do-n-times n body)
     ; Start generating code
     #`(begin
         ; Runs at compile time then
         ; splices the result into the
         ; generated code
         #,@(let l [(i (syntax->datum n))]
              ; Loops from n to 0
              (if (zero? i)
                  '()
                  ; Create a list #'body
                  (cons #'body (l (sub1 i)))))))]))
```

Figure 1: Sample macro

The meta-program in figure 1 expects a number **n** and an expression **body** and duplicates the expression **n** times. Each meta-program, created by **define-syntax**, takes a single piece of syntax as its argument. We use **syntax-case** to access the subforms of the syntax via pattern matching. **#'**, **#`**, and **#,** implement Lisp's quote, quasiquote, and unquote but on syntax instead of lists. In the example, we run a loop at compile-time that generates a list with **n** copies of the syntax **body**, and then splice (**#,@**) the copies into the generated program.

### 3.2 Profile-guided conditional branch optimization

The .NET compiler feature value probes, which enable profile-guided reordering of if/else and **switch** statements (Profile-Guided Optimizations 2013). As our first example, we optimize Scheme's **cond** and **case** constructs, which are similar to if/else and **switch** in other languages. This example demonstrates that our mechanism can be used to easily implement this optimization without the specialized support of value probes. It also demonstrates that our mechanism allows programmers 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. The clauses of **cond** are executed in

order until the left-hand side of a clause is true. If there is an **else** clause, the right-hand side of the **else** clause is taken only if no other clause's left-hand side is true. Figure 2 shows and example program using **cond**.

```
(define (fact n)
 (cond
  [(zero? n) 1]
  [(eq? n 5) 120] ; A very common case
  [else (* n (fact (sub1 n)))]))
```

Figure 2: An example using **cond**

We introduce the **exclusive-cond** construct, figure 3, as a similar conditional branching construct, 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, metaprogramming 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 is not enough 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. 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. Of course the **else** clause it is always last and is not included when sorting the other clauses.

We use the function **profile-query-weight** to access the profile information. Given a source object or piece of syntax, it returns the associated profile weight. [2]

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 4 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.

Figure 5 shows the full profile-guided implementation of **case** that sorts clauses by which is executed most often.

[2] TODO: Ensure this is runnable

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

Figure 4: An example using **case**

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 6 shows how the example **case** expression from figure 4 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 6: The expansion of figure 4

Finally, figure 7 show the result of expanding **exclusive-cond** in figure 6. 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 7: The expansion of figure 6

### 3.3 Profile-guided receiver class prediction

In this example implement profile-guided receiver class prediction (Grove et al. 1995) for an object-oriented DSL implemented in Scheme. We perform this optimization by tak-

```
(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 3: 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 5: Implementation of **case** using **exclusive-cond**

ing advantage of the **exclusive-cond** construct we developed in the last section. This example demonstrates that our mechanism is both general enough to implement well-known profile-guided optimizations, and powerful enough to provide DSL writers optimizations traditionally left to compiler writers.

We borrow the following example from Grove et. al. (Grove et al. 1995). The classes **Square** and **Circle** implement the method **area**. The naïve DSL compiler simply expands every method call into a conditional checks for known instances of classes and inlines the correct method bodies, as in figure 8. We would like to inline the common cases, but if there are many known classes the conditional tests may be too expensive to make this worthwhile. Furthermore, we would like to perform the tests in order according to which is most likely to succeed.

```
(cond
 [(class-equal? obj Square)
  (* (field obj length) (field obj width))]
 [(class-equal? obj Circle)
  (* pi (sqr (field obj r)))]
 [else (method obj area)])
```

Figure 8: Generated receiver class prediction code.

As we saw in the previous section, **exclusive-cond** provides a way to encode our high level knowledge of the program. In particular, we know class equality tests are mutually exclusive and safe to reorder. We can simply reimplement method calls using **exclusive-cond** instead of **cond** to get profile-guided receiver class prediction. To eliminate uncommon cases altogether and more quickly fall

```
; method call expands to ==>
(exclusive-cond
 [(class-equal? obj Square)
  ; executed 2 times
  (* (field obj length) (field obj width))]
 [(class-equal? obj Circle)
  ; executed 5 times
  (* pi (sqr (field obj r)))]
 [else (method obj "area")])
; expands to ==>
(cond
 [(class-equal? obj Circle)
  ; executed 5 times
  (* pi (sqr (field obj r)))]
 [(class-equal? obj Square)
  ; executed 2 times
  (* (field obj length) (field obj width))]
 [else (method obj "area")])
```

Figure 10: Profile-guided Generated code and expansion

back to dynamic dispatch, we can even use the profile information to stop inlining after a certain threshold. This implementation is shown in figure 9. In this example, we arbitrarily choose to inline only methods that take up more than 20% of the computation.

Figure 10 shows how our receiver class prediction example is optimized through **`exclusive-cond`**. Again, the generated **`cond`** will test for the common case first.

### 3.4 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 example we should our mechanism is general enough to implement this kind of profiling tool, and even go beyond it by automating the recommendations.

In this example, 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 11 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 12 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 example does, however, demonstrate that our mechanism is general and powerful enough to implement novel profile directed optimizations.

## 4. Implementation

This section describes the details of our mechanism in Chez Scheme and in Racket. We discuss representations of source objects and profile weights, how we instrument code, and how we ensure source-level and block-level profile-guided optimizations work together in our system.

### 4.1 Source objects

In the previous sections we assumed that source objects can be created arbitrarily, attached to source points in the surface

```
; Programmer calls to obj.m(val* ...) expand to (method-inline obj m val* ...)
; Inline likely classes in most likely order
(define-syntax (method-inline syn)
 (syntax-case syn ()
  [(_ obj m val* ...)
  (with-syntax ([(this-val* ...) #'(obj val* ...)])
  ; Create an exclusive-cond, since it knows how to optimize clauses
   #`(exclusive-cond
       #,@(filter values
            (map (lambda (class)
                   (let* ([method-ht (cdr (hashtable-ref classes class &undefined))]
                          [method-info (hashtable-ref method-ht (syntax-
>datum #'m) &undefined)])
                     (with-syntax
                       ([(arg* ...) (cadr method-info)] [(body body* ...) (cddr method-info)])
                       ; Inline only methods that use more than 20% of the computation.
                       (if (> (profile-query-weight #'body) 0.2)
                           #`[(class-equal? obj #,(datum->syntax #'obj class))
                              (let ([arg* this-val*] ...) body body* ...)]
                           #f))))
                 (vector->list (hashtable-keys classes))))
       ; Fall back to dynamic dispatch
       [else (method obj m val* ...)]))]))
```

Figure 9: The implementation of method inlining.

syntax and be used as keys. Here we describe the Chez and Racket implementations of source objects.

In Chez Scheme, a source object usually contains a file name and starting and ending character positions. The Chez Scheme reader automatically creates and attaches these to each piece of syntax read from a file. Chez Scheme also provides an API to programmatically manipulate source objects (Dybvig 2011 Chapter 11). This is useful when using Chez Scheme as a target language for a DSL with a different surface syntax. Custom source objects can be attached to target syntax as well to support profile-guided meta-programming.

To create custom source objects for fresh profile counters, we can use arbitrary file names and positions. For instance, in section 3.4 we create custom source objects to profile list and vector operations. We use the Chez API to attach these to generated syntax objects. In our Chez implementation, these are created using the function in figure 13. This function takes a string prefix and uses a counter to generate a fake file names and character positions. The generated file name is partly based on the input syntax in case they show up in error messages.

Racket does not attach separate source objects to syntax. Instead, the file name, line number, column number, position, and span are all attached directly to the syntax object. We provide wrappers to extract these into separate source objects to implement the profile database, and to attach our source objects to Racket syntax objects. Figure 14 shows the

```
(define (make-fresh-source-obj-
factory! prefix)
    (let ([n 0])
      (lambda (syn)
        (let* ([sfd (make-source-file-
descriptor
                     (format "~a:~a:~a" (syntax-
>filename syn) prefix n) #f)]
               [src (make-source-
object n n sfd)])
          (set! n (add1 n))
          src))))
```

Figure 13: Chez implementation for generating source objects

Racket implementation of the previous function. Again we generate a fake file names, but simply copies the other information from the input syntax object.

### 4.2 Profile weights

We represent a profile weight as a set of floating point numbers between 0 and 1. We store `#f` (false) when there is no profile information, and 0 when the counter was never executed. Our examples all ignore this distinction and assume `profile-query-weight` always returns a number. As mentioned in section 2.2, while the profiler counters track the exact number of times a source point is reached, the pro-

```
(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 (>= (profile-query-weight list-src) (profile-query-weight vector-src))
         (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 11: Implementation of profiled list

```
(define (make-fresh-source-obj-
factory! prefix)
  (let ([n 0])
    (lambda (syn)
      (let ([src (struct-
copy srcloc (syntax->srcloc syn)
                   [source (format "~a:~a:~a" (syntax-
source syn) prefix n)])])
        (set! n (add1 n))
        src))))
```

Figure 14: Racket implementation for generating source objects

file information is converted to weights when being stored in the database. We considered using Scheme fixnums (integers) for additional speed, but fixnums quickly loose precision, particularly when working with multiple data sets.

In Chez, we store profile weights by creating a hash table from source file names to hash tables. Each second level hash table maps the starting character position to a profile weight. These tables are not updated in real time, only when a new data set is manually loaded by an API call in a program or meta-program.

In Racket, we use one of the pre-existing profiling systems. The **errortrace** library provides exact profile counters, like the Chez Scheme profiler. We implement several wrappers to provide an API similar to the API provided by Chez Scheme. All these wrappers are implemented simply as Racket functions that can be called at compile time, requiring no change to either the Racket language implementation or the **errortrace** library.

The Racket implementation does not maintain a database in the way the Chez Scheme implementation does. Profile information is stored as an association list mapping source objects to profile counts. Profile weights are computed on each call to **profile-query-weight**.

```
(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 (profile-query-weight list-src))
   (define previous-vector-usage (profile-query-weight vector-src))
   (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 12: Implementation of profiled sequence

### 4.3 Instrumenting code

In this section we discuss how we instrument code to collect profiling information efficiently. As we use a preexisting profiling library for Racket, we only discuss how we instrument Chez Scheme.

The naïve method for instrumenting code to collect source profile information would be to add a counter for each source expression. However this method can easily distort the profile counts. As expressions are duplicated or thrown out during optimizations, the source information is also duplicated or lost.

Instead we create a separate profile form that is created after macro expansion. Each expression ***e*** that has a source object attached is expanded internally to **(begin (profile *src*) *e*)**, where ***src*** is the source object attached to ***e***. The profile form is considered an effectful expression and should never be thrown out or duplicated, even if ***e*** is. This has the side-effect of allowing profile information to be used for checking code-coverage of test suites. While the separate profile form has benefits, it can interfere with optimizations based on pattern-matching on the structure of expressions, such as those implemented in a nanopass framework (W Keep and Kent Dybvig 2013).

We keep profile forms until generating basic blocks. While generating basic blocks, the source objects from the profile forms are gathered up and attached to the basic block in which they appear. When a basic block is entered, every instruction in that block will be executed. For every instruction in the block, the profile counter must be incremented. So it is safe to increment the counters for all the instructions in the block at the top of the block.

In our implementation, we minimize the number of counters incremented at runtime. After generating basic blocks and attaching the counters to blocks, we analyze the blocks to determine which counters can be calculated in terms of other counters. If possible, a counter is computed as the sum of a list of other counters. This complicates the internal representation of counters and the generation of counters, but

decreases the overhead of profiling. These techniques are based on the work of Burger and Dybvig (Burger and Dybvig 1998). We generate at most one increment per block, and fewer in practice.

To instrument block-level profiling, we reuse the above infrastructure by creating fake source objects. Before compiling a file, we reset global initial block number to 0, and create a fake source file based on the filename. We give each block a source object using the fake filename and using the blocks number as the starting and ending file position.

### 4.4 Source and block PGO

In this section we discuss how we use source and block-level PGO in our mechanism. Again this section is only relevant to our Chez Scheme implementation.

When designing our source level profiling system, we wanted to continue using prior work on low level profile-guided optimizations (Gupta et al. 2002; Pettis and C. Hansen 1990; W. Hwu and P. Chang 1989). However, optimizations based on source-level profile information may result in a different set of blocks, so the block-level profile information will be stale. Therefore optimization using source profile information and those using block profile information cannot be done after a single profiled run of a program.

To take advantage of both source and block-level PGO, first we compile and instrument a program to collect source-level information. We run this program and collect only source-level information. Next we recompile and optimize the program using the source-level information only, and instrument the program to collect block-level information. From this point on, source-level optimizations should run and the blocks should remain stable. We run this program and collect only the block-level information. Finally, we recompile the program with both source-level and block-level information. Since the source information has not changed, the meta-programs generate the same source code, and thus the compiler generates the same blocks. The blocks are then optimized with the correct profile information.

## 5. Related and Future Work

### 5.1 Low-level PGO

Modern systems such as GCC, .NET, and LLVM use profile directed optimizations (Lattner 2002; Optimize Options - Using the GNU Compiler Collection 2013). These systems 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 3.2.

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 profile weights and many counters when loading new profile data.

### 5.3 Meta-program optimizations

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)[3], 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). These meta-programs can lose or obscure information during the translation into target-language code.

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

[3] TODO: STL?

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 collection (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.

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

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

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

Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning. In *Proc. The ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990. http://doi.acm.org/10.1145/93542.93550

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

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.

W. W. Hwu and P. P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proc. The 16th Annual International Symposium on Computer Architecture*, 1989. http://doi.acm.org/10.1145/74925.74953