Keyword and Optional Arguments in PLT Scheme

Matthew Flatt
University of Utah and PLT
mflatt@cs.utah.edu

Eli Barzilay
Northeastern University and PLT
eli@ccs.neu.edu

Abstract

The **lambda** and procedure-application forms in PLT Scheme support arguments that are tagged with keywords, instead of identified by position, as well as optional arguments with default values. Unlike previous keyword-argument systems for Scheme, a keyword is not self-quoting as an expression, and keyword arguments use a different calling convention than non-keyword arguments. Consequently, a keyword serves more reliably (e.g., in terms of error reporting) as a lightweight syntactic delimiter on procedure arguments. Our design requires no changes to the PLT Scheme core compiler, because **lambda** and application forms that support keywords are implemented by macros over conventional core forms that lack keyword support.

1. Using Keyword and Optional Arguments

A rich programming language offers many ways to abstract and parameterize code. In Scheme, first-class procedures are the primary means of abstraction, and procedures are unquestionably the right vehicle for parameterizing code with respect to a few run-time values. For parameterization over larger sets of values, however, Scheme procedures quickly become inconvenient.

Keyword and optional arguments support tasks that need more arguments than fit comfortably into procedures, but where radically different forms—such as **unit** or **class** in PLT Scheme—are too heavyweight conceptually and notationally. At the same time, keyword and optional arguments offer a smooth extension path for existing procedure-based APIs. Keyword arguments can be added to a procedure to extend its functionality without binding a new identifier (which always carries the danger of colliding with other bindings) and in a way that composes with other such extensions.

Keyword arguments in PLT Scheme are supported through a straightforward extension of the **lambda**, **define**, and application forms. Lexically, a keyword starts with `#:` and continues in the same way as an identifier; for example, `#:color` is a keyword.[1]

A keyword is associated with a formal or actual argument by placing the keyword before the argument name or expression. For example, a **rectangle** procedure that accepts two by-position arguments and one argument with the `#:color` keyword can be written as

```
(define rectangle
  (lambda (width height #:color color)
    ....))
```

or

```
(define (rectangle width height #:color color)
  ....)
```

This **rectangle** procedure could be called as

```
(rectangle 10 20 #:color "blue")
```

A keyword argument can be in any position relative to other arguments, so the following two calls are equivalent to the preceding one:

```
(rectangle #:color "blue" 10 20)
(rectangle 10 #:color "blue" 20)
```

The `#:color` formal argument could have been in any position among the arguments in the definition of **rectangle**, as well. In general, keyword arguments are designed to look the same in both the declaration and application of a procedure.

In a procedure declaration, a formal argument can be paired with a default-value expression using a set of parentheses—or, by convention, square brackets. The notation for a default-value expression is the same whether the argument is by-position or by-keyword. For example, a **rectangle**'s height might default to its width and its color default to pink:

```
(define (rectangle width
                   [height width]
                   #:color [color "pink"])
  ....)
```

This revised **rectangle** procedure could be called in any of the following ways:

```
(rectangle 10)
(rectangle 10 20)
(rectangle 10 20 #:color "blue")
(rectangle 10 #:color "blue")
(rectangle #:color "blue" 10 20)
(rectangle #:color "blue" 10)
(rectangle 10 #:color "blue" 20)
```

Our goals in a design for keyword and optional arguments include providing especially clear error messages and enforcing a consistent syntax for keyword arguments. Toward these goals, two aspects of our design set it apart from previous approaches in Lisp and Scheme:

- Keywords are distinct from symbols, and they are not self-quoting as expressions.

  For example, the form

  ```
  #:color
  ```

  in an expression position is a syntax error, while

  ```
  (rectangle #:color "blue")
  ```

[1] See Section 7.7 for a discussion on this choice of keyword syntax.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

is a call to `rectangle` with the `#:color` argument `"blue"`. In the latter case, the procedure-application form treats the `#:color` keyword as an argument tag, and not as an expression. Every keyword in an application must be followed by a value expression, so the form

```
(rectangle #:color #:filled? #f)
```

is rejected as a syntax error, because `#:color` lacks an argument expression; if keywords could be expressions, the call would be ambiguous, because `#:filled?` might be intended as the `#:color` argument to `rectangle`.

- Keywords are not passed as normal arguments to arbitrary procedures, where they might be confused with regular procedure arguments. Instead, a different calling convention is used for keyword arguments.

  For example,

  ```
  (cons #:color "blue")
  ```

  does not create a pair whose first element is a keyword and second element is a string. Evaluating this expression instead reports a run-time error that `cons` does not expect keyword arguments.

Although a keyword is not self-quoting as an expression, a keyword is a first-class value in PLT Scheme. A keyword can be quoted to produce a valid expression, as in `'#:color` and `(cons '#:color "blue")`, where the latter creates a pair whose first element is a keyword. Keyword values and quoted-keyword expressions are useful for creating a procedure that accepts arbitrary keyword arguments and processes them explicitly. Keyword values are also useful in reflective operations that inspect the keyword requirements of a procedure. By convention, PLT Scheme programmers do not use keywords for run-time enumerations and flags, leaving those roles to symbols and reserving keywords for syntactic roles.

The rest of the paper proceeds as follows. Section 2 describes the syntax and semantics of keyword and optional arguments in PLT Scheme. Section 4 describes our implementation of keyword arguments. Section 5 provides some information on the performance of keyword and optional arguments. Section 6 reports on our experience using keywords in PLT Scheme. Section 7 describes previous designs for keywords in Lisp and Scheme and relates them to our design.

2. Syntax and Semantics

Figure 1 shows the full syntax of PLT Scheme's **`lambda`**. In a ⟨kw-formals⟩, all ⟨id⟩s must be distinct, including ⟨rest-id⟩, and all ⟨keyword⟩s must be distinct. A required non-keyword argument (i.e., the first case of ⟨formal-arg⟩) must not follow an optional non-keyword argument (i.e., the second case of ⟨formal-arg⟩).

A **`lambda`** form that is constructed using only the ⟨id⟩ form of ⟨fomal-arg⟩ has the same meaning as in standard Scheme (Sperber 2007). A **`lambda`** form that uses only the ⟨id⟩ and `[`⟨id⟩ ⟨default-expr⟩`]` forms of ⟨formal-arg⟩ can be converted to an equivalent **`case-lambda`** form; the appendix shows the conversion precisely in terms of **`syntax-rules`**. For each optional argument that is not supplied in an application of the procedure, the corresponding ⟨default-expr⟩ is evaluated just before the procedure body is evaluated. The environment of each ⟨default-expr⟩ includes the preceding arguments, and if multiple ⟨default-expr⟩s are evaluated, then they are evaluated in the order that they are declared. When a "rest argument" is declared after optional arguments, arguments in an application are first consumed by the optional-argument positions, so the rest argument is non-empty only when more arguments are provided that the total number of required and optional arguments.

`(lambda` ⟨kw-formals⟩ ⟨body⟩ ...+**`)`**

⟨kw-formals⟩ = **`(`**⟨formal-arg⟩ ...**`)`**
| **`(`**⟨formal-arg⟩ ...+ **`.`** ⟨rest-id⟩**`)`**
| ⟨rest-id⟩

⟨formal-arg⟩ = ⟨id⟩
| **`[`**⟨id⟩ ⟨default-expr⟩**`]`**
| ⟨keyword⟩ ⟨id⟩
| ⟨keyword⟩ **`[`**⟨id⟩ ⟨default-expr⟩**`]`**

... means "zero or more," ...+ means "one or more," ⟨id⟩ or ⟨rest-id⟩ matches an identifier, ⟨expr⟩ or ⟨default-expr⟩ matches an expression, ⟨keyword⟩ matches a keyword, and ⟨body⟩ matches a definition or expression in an internal-definition context

Figure 1: Extended grammar for **`lambda`**

`(`⟨proc-expr⟩ ⟨actual-arg⟩ ...+**`)`**
⟨actual-arg⟩ = ⟨expr⟩
| ⟨keyword⟩ ⟨expr⟩

Figure 2: Extended grammar for procedure application

```
> (define polygon
    (lambda (n [side-len (/ 12 n)] . options)
      (list n side-len options)))
> (polygon)
procedure polygon: no clause matching 0 arguments
> (polygon 3)
(3 4 ())
> (polygon 3 7)
(3 7 ())
> (polygon 3 7 'solid 'smooth)
(3 7 (solid smooth))
```

When the ⟨keyword⟩ ⟨id⟩ or ⟨keyword⟩ `[`⟨id⟩ ⟨default-expr⟩`]` forms of ⟨formal-arg⟩ are used to construct a **`lambda`** expression, the resulting procedure accepts keyword-tagged arguments in addition to the arguments that would be accepted without the keyword-tagged arguments. Arguments using the ⟨keyword⟩ ⟨id⟩ form are required, while arguments using the ⟨keyword⟩ `[`⟨id⟩ ⟨default-expr⟩`]` form are optional. As with the keywordless `[`⟨id⟩ ⟨default-expr⟩`]` form, each keyword-tagged ⟨default-expr⟩ is evaluated for a given application of the procedure if no actual argument is tagged with the corresponding ⟨keyword⟩, and the preceding argument ⟨id⟩s are in the environment of each ⟨default-expr⟩. When ⟨default-expr⟩s are evaluated for multiple arguments, they are evaluated in the order declared in the **`lambda`** expression, independent of whether the arguments have a keyword tag or the order of keyword tags on actual arguments. Actual arguments that are tagged with a keyword can be supplied in any order with respect to each other and with respect to by-position arguments.

```
> (define polygon
    (lambda (n [side-len (/ 12 n)]
              #:color [color "blue"]
              #:rotate theta
              . options)
      (list n side-len color theta options)))
> (polygon 4)
polygon: requires an argument with keyword #:rotate, not
supplied; arguments were: 4
> (polygon 4 #:rotate 0)
(4 3 "blue" 0 ())
> (polygon 4 7 #:rotate 0 #:color "red" 'solid)
(4 7 "red" 0 (solid))
```

The above examples use the extended syntax of procedure applications shown in Figure 2, which allows arguments tagged with keywords. Each ⟨keyword⟩ in an application must be distinct. Crucially, the grammar of ⟨expr⟩ in PLT Scheme (not shown here) does not include an unquoted ⟨keyword⟩, so the grammar for procedure application is unambiguous.

Naturally, the result of ⟨proc-expr⟩ in an application must be a procedure. For each keywordless argument ⟨expr⟩, the result is delivered to the procedure as a by-position argument, while each other ⟨expr⟩ is provided with the associated ⟨keyword⟩. PLT Scheme always evaluates the sub-expressions of a procedure application left-to-right, independent of whether the argument is tagged with a keyword. If the applied procedure evaluates ⟨default-expr⟩s for unsupplied arguments, it does so only after all of the ⟨expr⟩s in the procedure application are evaluated. Similarly, the expected and supplied arguments (in terms of arity and keywords) are checked after all of the argument ⟨expr⟩s are evaluated but before the any ⟨default-expr⟩s would be evaluated (so no ⟨default-expr⟩s are evaluated if the number of supplied by-position arguments is wrong, if a required keyword argument is missing, or if an unexpected keyword is supplied).

The **define** shorthand for procedure is extended in the obvious way to support keyword and optional arguments. PLT Scheme also supports the MIT curried-function shorthand, which composes seamlessly with keyword and optional arguments.

```
> (define ((rect w [h w] #:color [c "pink"])
           canvas x y)
    (set-pen-color! canvas c)
    (draw-rectangle! canvas x y w h))
> ((rect 10 #:color "blue") screen 0 0)
```

In addition to the **lambda**, **define**, and application syntactic forms, our design extends and adds a few procedures. An extended **apply** procedure accepts arbitrary keyword arguments, and it propagates them to the given procedure.

```
> (apply polygon 4 7 #:rotate 0 '(solid smooth))
(4 7 "blue" 0 (solid smooth))
```

Keyword arguments to **apply** are analogous to arguments between the procedure and list argument in the standard **apply**; that is, they are propagated directly as provided. The **keyword-apply** procedure generalizes **apply** to accept a list of keywords and a parallel list of values, which are analogous to the last argument of **apply**.

```
> (keyword-apply polygon
                 '(#:color #:rotate)
                 '("blue" 0)
                 4 7
                 '(solid smooth))
(4 7 "blue" 0 (solid smooth))
```

The list of keywords supplied to **keyword-apply** must be sorted alphabetically, for reasons explained in Section 4.

The **make-keyword-procedure** procedure constructs a procedure like **apply** that accepts arbitrary keyword arguments. The argument to **make-keyword-procedure** is a procedure that accepts a list of keywords for supplied arguments, a parallel list of values for the supplied keywords, and then any number of by-position arguments.

```
> (define trace-call
    (make-keyword-procedure
     (lambda (kws kw-vals proc . args)
       (printf ">>~s ~s ~s<<\n" kws kw-vals args)
       (keyword-apply proc kws kw-vals args))))
> (trace-call polygon 6 #:rotate 0)
>>(#:rotate) (0) (6)<<
(6 2 "blue" 0 ())
```

Finally, the reflection operations **procedure-arity** and **procedure-reduce-arity** in PLT Scheme inspect or restrict the arity of a procedure. The additional procedures **procedure-keywords** and **procedure-reduce-keyword-arity** extend the set of reflection operators to support keywords. The **procedure-keywords** procedure reports the keywords that are required and allowed by a given procedure. The **procedure-reduce-keyword-arity** procedure converts a given procedure with optional keyword arguments to one that allows fewer of the optional arguments and/or makes some of them required. A typical use of **procedure-reduce-keyword-arity** adjusts the result of **make-keyword-procedure** (for which all keywords are optional) to give it a more specific interface.

PLT Scheme does not extend **case-lambda** to support keyword or optional arguments; the extension would be straightforward, but there has been no demand. Similarly, continuations in PLT Scheme do not support keyword arguments. Extended variants of **call-with-values**, **values**, and **call/cc** procedures could support keyword results and continuations that accept keyword arguments. We have not tried that generalization, but an implementation could use continuation marks (Clements and Felleisen 2004) that are installed by **call-with-values** and used by **values** and **call/cc** to connect a keyword-accepting continuation with its application or capture.

3. Keywords in Other Syntactic Forms

The PLT Scheme macro system treats keywords in the same way as a number or a boolean. For example, a pattern for a macro can match a literal keyword:

```
(define-syntax show
  (syntax-rules ()
    [(_ #:canvas c expr ...)
     (call-with-canvas c (lambda () expr ...))]
    [(_ expr ...)
     (show #:canvas default-canvas expr ...)]))
```

This macro recognizes an optional **#:canvas** specification before a sequence of drawing expressions to select the target of the drawing operations. For example, the first pattern in the **syntax-rules** form matches

```
(show #:canvas my-canvas (draw-point! 0 0))
```

while the second clause matches

```
(show (draw-point! 0 0))
```

The second clause also matches

```
(show #:dest my-canvas (draw-point! 0 0))
```

in which case **#:dest** is used as an expression, and a syntax error after expansion reports the misuse of **#:dest**. That is, the pattern matcher for macros does not constrain arbitrary pattern variables against matching literal keywords. The error message "**#:dest** is not an expression" is less clear than "the **show** form expects **#:canvas** and does not recognize **#:dest**," and a **syntax-case** implementation of **#:draw** could more thoroughly check its sub-forms. Similarly, the first clause in the **show** macro does not match

```
(show (draw-point! 0 0) #:canvas my-canvas)
```

since it recognizes **#:canvas** only at the beginning of the form. Again, a **syntax-case** implementation of **show** could allow **#:canvas** in later positions, if desired.[2]

[2] A better solution would be a variant of **syntax-rules** that handles keyword constraints and ordering automatically—along with related constraints, such as requiring an identifier.

```
(define-struct ⟨id⟩ (⟨field⟩ ...) ⟨struct-option⟩ ...)

      ⟨field⟩ = ⟨field-id⟩
              | [⟨field-id⟩ ⟨field-option⟩ ...]

⟨struct-option⟩ = #:super ⟨super-expr⟩
                | #:auto-value ⟨auto-expr⟩
                | #:property ⟨prop-expr⟩ ⟨val-exr⟩
                | #:transparent

⟨field-option⟩ = #:mutable
               | #:auto
```

Figure 3: Partial grammar for PLT Scheme's **define-struct**

Syntactic forms in PLT Scheme that use keywords include the **define-struct** form and the **->*** contract constructor. Both are typical in that they allow keywords only in specific places (instead of anywhere between the form's parentheses). For example, the syntax of **define-struct** is shown in Figure 3, where keyword-tagged options appear only within ⟨field⟩s and after the ⟨field⟩ sequence. In the allowed positions, however, keywords are used in a more flexible way than in an application form; the **#:transparent**, **#:mutable**, and **#:auto** keywords need no corresponding argument expression, while the **#:property** keyword is followed by two expressions. This combination of constraints (i.e., requiring keywords in certain positions) and generalizations (i.e., allowing different numbers of expressions associated with a keyword) compared to procedure application is the prerogative of a syntactic form.

At the same time, **define-struct** relies on the prohibition of unquoted keywords as expressions to provide good error messages when parsing a set of ⟨struct-option⟩s, such as when the **#:super** keyword lacks a corresponding expression before the next keyword. The consistent role of keywords as non-expression delimiters has encouraged the use of keywords within syntactic forms for PLT Scheme.

4. Implementation

Although **lambda** and the procedure-application form in PLT Scheme support keyword and optional arguments, the core compiler does not directly support them. Instead, support for keyword and optional arguments is implemented as a macro in a library, in the same way that **unit** and **class** are implemented as macros over the core **lambda** form. The only core support for keywords is a keyword datatype and reader syntax.

The library is implemented so that the keyword-supporting application form is equivalent to the core application form when no keywords are supplied, and a **lambda** form with no keyword or optional arguments is equivalent to the core **lambda** form. Furthermore, a procedure with only optional keyword arguments can be called through the core application form. These constraints on the design preserve the performance of keywordless procedure applications and provide good interoperability between libraries that use and do not use keyword-supporting syntactic forms.

A PLT Scheme library can implement an extended application form, because an application form implicitly uses the **#%app** binding in its lexical environment. For example, in

```
(require (rename-in scheme [#%app orig-#%app]))
(define-syntax-rule (#%app expr ...)
  (begin
    (orig-#%app printf "at ~s\n" '(expr ...))
    (orig-#%app expr ...)))
(+ 1 (+ 2 3))
```

each application of **+** prints debugging information before evaluating the application:

```
at (+ 1 (+ 2 3))
at (+ 2 3)
6
```

The library that implements keyword and optional arguments supplies an **#%app** macro in addition to **lambda** and **define** to replace the core bindings. The replacement macros expand a keyword-supporting **lambda**, **#%app**, or **define** into a combination of primitive forms and run-time functions (such as **make-keyword-procedure**) that implement keyword arguments.

To allow procedures with optional keywords to be applied through the core application form, the implementation relies on a second PLT Scheme facility that predates support for keywords: applicable structure types. When the core application form encounters a value to apply that is not a procedure, it checks whether the value is an instance of a structure type that has an associated application operation (which is itself represented as a procedure). If so, it uses the associated operation to apply the structure to the given arguments. For example, another way to create noisy procedure applications is to wrap the base procedure in a **traced** structure:

```
(define-struct traced (f)
  #:property prop:procedure ; => applicable
             (lambda (t . args)
               (let ([f (traced-f t)])
                 (printf "~s\n" (cons f args))
                 (apply f args))))
(define traced-cons (make-traced cons))
(traced-cons 1 2)
```

Internally, the keyword-handling part of a procedure is represented by a core procedure that accepts a list of keywords, a list of corresponding values, and then the by-position arguments—just like a procedure given to **make-keyword-procedure**. This internal representation is wrapped in an applicable structure, where the application operation (which is used by a non-keyword application form) calls the internal procedure with empty keyword and keyword-value lists. The application form with keywords, meanwhile, extracts the internal procedure and applies it to non-empty keyword and value lists. The list of keywords is always sorted alphabetically, so that the supplied keywords can be checked against an expected set without sorting or searching when the internal procedure is called. The internal procedure is not directly accessible, since it is wrapped in an opaque structure.

The keyword-supporting application form sorts a set of supplied keywords at compile time. Compile-time sorting is possible because keywords in an application are statically apparent; keywords that act as argument tags are syntactic literals, while expressions that produce keyword values are never treated as argument tags. The list of keywords also can be allocated once per call site (as a quoted list of keywords), while the list of corresponding values must be allocated for each call. This detail explains why the internal representation of a keyword-accepting procedure accepts a list of keywords separate from the list of arguments.

Finally, an applicable structure that represents a keyword procedure has an associated property to generates a string description of the procedure's arity and expected keywords. This property is used when a procedure that accepts only optional keyword arguments is applied to the wrong number of by-position arguments. In that case, the arity-mismatch error not only describes the expected number of by-position arguments, but also the optional keyword arguments. This arity-description property is built into the run-time system, since it must be used when reporting an arity mismatch from the core application form.

5. Performance

In PLT Scheme, application of a keyword-accepting procedure is somewhat slower than a keywordless procedure, but the design presented here significantly outperforms our earlier, more conventional implementation. The performance cost relative to plain procedures has several causes: applications without optional keywords must extract a procedure from an applicable structure; keyword arguments are always collected into a list; keyword arguments must be checked against the expected set of keywords; and the compiler currently cannot inline keyword-accepting procedures. Procedures with optional (but no keyword) arguments expand to **case-lambda**, in which case the relative cost is lower (no applicable structure, no keyword checking, and not collecting arguments into a list), but the compiler currently does not inline multi-clause **case-lambda** procedures.

The following loops serve as rough micro-benchmarks:

```
; A plain procedure
(define (sub1 n) (- n 1))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1 n))))

; With an optional argument
(define (sub1/opt [n 0]) (- n 1))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/opt n))))

; With unsupplied keyword argument
(define (sub1/kw/unused n #:m [m 1]) (- n m))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/kw/unused n))))

; Pass the argument in a list
(define (sub1/list nl) (- (car nl) 1))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/list (list n)))))

; With a required keyword argument
(define (sub1/kw #:n n) (- n 1))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/kw #:n n))))

; Required and unsupplied optional
(define (sub1/kw2 #:n n #:m [m 1]) (- n m))
(let loop ([n 10000000])
  (unless (zero? n) (loop (sub1/kw2 #:n n))))

; Many optional keywords
(define (sub1/kws #:a [a 0] #:n [n 5]
                  #:q [q 0] #:z [z 1])
  (- n z))
(let loop ([n 10000000])
  (unless (zero? n)
    (loop (sub1/kw4 #:n n #:z 1))))
```

Since the variations of **sub1** merely perform a fixnum subtraction that will be inlined by the compiler, the micro-benchmarks compare just the overhead of different forms of procedure application. The run times for these versions are shown in Figure 4. For those runs, the benchmarks are executed outside of a module, where the compiler cannot inline definitions (but it can still inline the subtraction operation).

The "optional" case demonstrates the cost of **case-lambda** versus **lambda**, while the "unused keyword" case demonstrates the overhead of an applicable structure. The "list" case demonstrates the overhead of putting a single argument into a list and extracting it in the called function, as happens to a keyword argument in our implementation. The "keyword" case demonstrates the additional overhead of checking provided keyword arguments against the expected set. The "keywords and unused" case demonstrates how the checking overhead grows with both required and optional keywords, and the "many optional" case demonstrates how the overhead grows as additional optional keywords are added.

program	*CPU time (msec)*	*relative*
plain	327	1.0
optional	348	1.0
unused keyword	784	2.3
list	503	1.5
keyword	1115	3.4
required plus optional	1470	4.4
many optional	1999	6.1

Figure 4: Micro-benchmark results for PLT Scheme 4.2.1, on an 2GHz Core Duo MackBook running Mac OS X 10.5.7; results are median run times over three runs as measured using the **time** form

implementation	*plain*	*...hide*	*keywords*	*...hide*	*many kws*
PLT	48	327	1115	1103	1999
Old PLT	48	327	2869	2866	7891
Gambit-C, default	221	222	989	992	1437
Gambit-C, fast	43	118	897	930	1287
Chicken, default	1066	1079	2478	2502	7881
Chicken, fast	8	353	1203	1430	4529
R6RS, Ikarus	100	100	130*	1237	2054
R6RS, Larceny	66	143	66*	2237	4752
R6RS, PLT	50	361	50*	4644	9001
SBCL	126	217	232	332	473
Allegro CL	150	230	450	520	780

Figure 5: Micro-benchmark results on an 2GHz Core Duo MackBook running Mac OS X 10.5.7; PLT Scheme version 4.2.1; Gambit-C version 4.4.0, with **(declare (standard-bindings) (block) (fixnum) (not safe))** for the "fast" variant; Chicken version 4.0.0 with the **-Ob** compiler flag for the "fast" variant; Ikarus version 0.0.4-rc1+ revision 1827; Larceny version 0.97b1; SBCL version 1.0.23; Allegro CL express edition version 8.1; both SBCL and Alllegro CL use **(declaim (optimize (speed 3) (safety 1) (space 0) (debug 0)))**; results are median run times over three runs as measured using a **time** form

As a further check on the performance of our keyword implementation, we provide a comparison to several other implementations:

- An older and more conventional implementation of keyword arguments in PLT Scheme, where keywords are self-quoting and keywords are passed as normal procedure arguments.
- Keyword-argument support as provided by Gambit-C (Feeley 2009) and Chicken (Winkelmann et al. 2009), first with the default compiler settings, and then with settings for faster performance.
- Eddington's R6RS library for keyword procedures[3] as run in Ikarus (Ghuloum 2009), Larceny (Clinger et al. 2009), and PLT Scheme.
- SBCL (SBCL 2009) and Allego CL (Franz, Inc. 2009) using Common Lisp standard keyword functions (Steele 1990).

For each implementation, Figure 5 reports run times for the plain, single-keyword, and many-keyword micro-benchmarks. The plain and single-keyword benchmarks are each run in two ways: one

[3] `http://bazaar.launchpad.net/~derick-eddington/scheme-libraries/xitomatl/files`, revision 180

with a direct use of a **define**d function within a compilation unit (e.g., within a module), and another where the function name is **define**d as **#f** and then **set!**ed to the function (or, in the case of Common Lisp, **setf**ed and then called via **funcall**). The latter corresponds to the *...hide* column, and the intent is to defeat inlining and other static analyses. We measure this difference because our approach to implementing keywords, if ported to other systems, might discourage static analysis. For similar reasons, we check the effect of different compiler settings in Gambit-C and Chicken. In the R6RS cases, the non-*...hide* case for keywords is special, because it works in the opposite direction: it uses a **define/kw** form that binds a macro to statically convert keyword arguments to by-position arguments at the call site.

Not surprisingly, Common Lisp implementations perform keyword applications with the lowest overhead relative to plain applications. Keywords in Common Lisp are standard and widely used, so implementors are motivated to tune their compilers for keyword arguments. Along similar lines, the result for the new keyword system in PLT Scheme reflects a 30% speed boost from a JIT-specialized primitive that fits the structure-unpacking needs of a keyword application (although the JIT is oblivious to the use of this primitive for keyword applications). Overall, the results illustrate that keywords in PLT Scheme have a typical overhead, even while providing a better separation of keyword arguments from by-position arguments and providing more flexibility in the placement of keyword arguments relative to by-position arguments.

As the first row in Figure 5 shows, the PLT Scheme compiler can greatly improve performance through procedure inlining, but inlining is not currently available for procedures that accept keywords. The performance of inlining could be recovered with a form analogous to **define/kw** in Eddington's R6RS library, which binds the name of a keyword-accepting procedure to an identifier macro. The macro expands direct applications of the keyword-accepting procedure to call a plain procedure—statically converting keyword arguments into by-position arguments, and thus enabling the usual inlining optimizations for plain procedures. Note that keyword arguments always can be detected statically by such a macro with our design, since keywords are a syntactic part of a keyword application, instead of dynamically detected as expression results. The performance of keyword applications without macros has been good enough, however, that we have not yet explored this approach.

6. Experience

Support for optional arguments was one of the first macros that we included in PLT Scheme. We managed, however, to avoid supporting keyword arguments for over a decade. The protocol for keyword arguments seemed inherently complex, so we tried to live without them.

Eventually, however, we ended up with too many functions consuming too many optional arguments, where supplying the *n*th optional argument required supplying also the *n*-1 preceding optional arguments. We also created many functions that were small, non-composable variations of each other. For example, Slideshow (Findler and Flatt 2004) provided a **slide** function for generating a slide, a **slide/title** function for generating a slide with a title, a **slide/center** for generating a slide with centered content, and a **slide/title/center** function for generating a slide with a title and centered content. Further variations of **slide** included three slashes.

Our initial design for keywords in PLT Scheme was based on Common Lisp, but with even more extensions and with some attempts to clean up the mingling of keywords and rest arguments. For example, while an argument tagged with **#:rest** includes any supplied keyword arguments (analogous to Common Lisp), an argument tagged with **#:body** includes only extra arguments that follow keyword arguments—and those extra arguments need not have keywords. We used **#:body** frequently; for example, a keyword-based **slide** procedure must accept keywords for configuration but arbitrary rest arguments for the content of the slide.

Many other beautiful generalizations in our initial keyword system, such as the ability to nest optional and keyword syntax in place of a **#:body** identifier, went completely unused. Worse, concerns with error messages and with accidental consumption of keywords as arguments lead to a relatively restrained use of keywords in our libraries. To some degree, the complexity of the syntax for defining keyword-accepting procedures (and notably its lack of connection to the application syntax) also limited adoption. Finally, having to import an extra library to obtain the keyword-supporting **lambda** form was a significant obstacle.

The design presented here arose from an effort to make keyword arguments more widely acceptable in PLT Scheme: to simplify their semantics, to streamline their syntax, and to integrate them into our main dialect of Scheme. Subjectively, the design feels right, and we now use keyword arguments in many more functions and in parts of the language that are closer to the core. For example, **call-with-output-file** used to accept optional arguments to select text versus binary mode and to indicate handling for a file that exists already. Since the arguments were optional, they were placed at the end of the argument list, which is after the callback procedure that is often a **lambda** expression:

```
(call-with-output-file
 dest
 (lambda (out)
   ....) ; many lines
 'truncate
 'text)
```

The distance between the file name and the mode flags made the code difficult to read and write, and the specification of the extra arguments was awkward to document (i.e., up to two extra arguments that are distinct symbols from certain sets). Using keyword arguments, we write the above expression as

```
(call-with-output-file
 dest #:exists 'truncate #:mode 'text
 (lambda (out)
   ....))
```

In this form, the callback procedure regains its place at the end, where it belongs. The file name is still the first argument, where it belongs. The extra optional arguments are more clearly tagged via keywords, and they can be placed in the middle of the by-position arguments, which is where they work best. The specification of the optional arguments (i.e., keyword-tagged with simple defaults) is straightforward and easy to document.

Before deploying our current design for keyword argument, we anticipated problems with the pattern **(lambda args ...)** to accept arbitrary arguments or **(lambda args (apply ... args))** to propagate all arguments. Those patterns work only for by-position arguments; generalizing any use of those patterns requires a switch to **make-keyword-procedure** and **keyword-apply**, which is more verbose and more difficult to remember. For example, the **traced** example of an applicable structure in Section 4 does not support tracing of keyword arguments, and it should be generalized as follows:

```
(define-struct traced (f)
  #:property prop:procedure
  (make-keyword-procedure
   (lambda (kws vals t . args)
     (let ([f (traced-f t)])
       (printf "~s\n" (list* f kws vals args))
       (keyword-apply f kws vals args)))))
```

For similar reasons, some PLT Scheme library procedures have not automatically worked with keywords on a first iteration, such as the `const` function to produce another function that accepts any arguments and returns a constant. Such problems are easy to fix, and occasional missing support for keywords has not been a significant problem so far, but we expect to provide syntactic support for the `make-keyword-procedure` and `keyword-apply` pattern.

The initial implementation of our design for keywords did not include the extra property for arity reporting that is described at the end of Section 4. As a result, if the keyword-based `call-with-output-file` was applied to four by-position arguments, the error message simply reported that the procedure expects one to two arguments without mentioning that the procedure also accepts optional `#:exists` and `#:mode` arguments. Indeed, such an error message often appeared as a result of a call to `call-with-output-file` using old-style optional symbols instead of the new keyword arguments. PLT Scheme users immediately requested improvement in the error message, which reflects the demand for clear error reporting that our design was created to satisfy.

7. Related Work

We know of three major designs for keywords in Lisp and Scheme: keywords in Common Lisp (Steele 1990), keywords in DSSSL (ISO 1996), and SRFI-89 (Feeley 2007). At least one other design has been implemented through portable Scheme macros. Ada, Python, and OCaml, support keyword arguments, while keyword arguments in Smalltalk are fundamentally different. We take each of these in turn in the following sections, and we end with a brief discussion of the syntax of keywords in Scheme.

7.1 Common Lisp

In Common Lisp, keywords are the same datatype as symbols, but they are written with a `:` prefix and they are self-quoting as an expression. (This is actually a trick related to packages; see Section 7.7.)

A Lisp procedure definition can include the special identifiers `&optional` or `&key` before a set of arguments to declare them as optional or by-keyword. In the latter case, the local name of the argument effectively doubles as the keyword. Keyword arguments are always optional, and the default value for optional and keyword arguments is `nil` if none is declared. An `&allow-other-keys` declaration suppresses rejection of keywords for actual arguments that have no corresponding `&key` formal argument. A "rest" argument can be specified with the `&rest` declaration, which must appear before any `&key` declarations. (The full syntax is somewhat more complex, but those are the main points.)

For example, a `rectangle` procedure that accepts a width, an optional height that default to the width, and and an optional keyword-tagged color argument that defaults to `"pink"` is written and called as

```
(defproc (rectangle width
                    &optional (height width)
                    &key (color "pink"))
  ....)

(rectangle 10 20 :color "blue")
```

The semantics of `&optional` and `&key` declarations is essentially to extend the number of arguments accepted by the procedure, and then post-process the list of extra by-position arguments to match them with optional and keyword arguments. When the function consumes keyword arguments, the total number of arguments after the by-position arguments must be even, and the keywords that tag arguments are interleaved with the argument values–i.e., the argument list is used as a *plist*. When both `&key` and `&rest` are used, arguments that are candidates for keyword arguments (including the keywords themselves) are collected into a `&rest` argument, and the number of arguments must be even.

With keywords as part of the standard, many standard procedures in Common Lisp can exploit keyword arguments. For example, the `member` function accepts a comparison procedure as a `test:` argument, in contrast to Scheme's proliferation of separate `member`, `memv`, `memq`, and `memp` procedures.

An advantage of implementing keyword-argument passing as normal arguments, as in Common Lisp, is that procedures like `apply` work with keywords automatically, and the `&rest`-argument convention accommodates arbitrary keywords (at least when `&allow-other-keys` is declared). Separate `keyword-apply` and `make-keyword-procedure` procedures are unnecessary.

Compared to our design, however, the Common Lisp design suffers several drawbacks:

- Since optional- and keyword-argument values are drawn from the same set of actual arguments, and since the keywords that are meant as tags are passed the same as ordinary arguments, keywords can be accidentally consumed as optional arguments. As noted by Seibel (2005), "Combining `&optional` and `&key` parameters yields surprising enough results that you should probably avoid it altogether."
- Although folding keyword arguments into a `&rest` arguments makes sense in combination with `&allow-other-keys`, it means that a procedure cannot generally accept both by-position rest arguments and keyword arguments. Instead, using keywords forces the rest argument to be a plist.
- Keyword arguments must be placed last in a procedure application. That is, keywords can be in any order relative to each other, but they must appear after all required and optional by-position arguments.

The first two drawbacks, in particular, inhibit the use of keywords to extend existing procedures that already use optional or rest arguments. Our design accommodates such extensions, while producing more consistent error messages and being simpler to explain overall.

Dylan (Shalit 1996) supports keyword arguments in much the same way as Common Lisp, except that only keyword arguments can be optional. Furthermore, Dylan distinguishes keyword tags in applications from argument expressions, so that a keyword intended as a tag is never accepted as an argument value. Dylan thus achieves many of the goals in our design of providing a better separation between keyword and by-position arguments, but it does so by restricting the Common Lisp model. A remaining drawback is that keyword arguments cannot be mixed with by-position arguments.

7.2 DSSSL

DSSSL includes an expression language that is based on Scheme, but it includes keyword arguments similar to those of Common Lisp. Keywords in DSSSL are a separate datatype from symbols; they are written like symbols, but with a trailing `:`. Instead of identifiers like `&key` that are treated specially in argument lists, DSSSL uses the special constants `#!key`, `#!optional`, and `#!rest` (and it omits the other declarations of Common Lisp). The semantics of procedure calls and argument processing are as in Common Lisp.

The `rectangle` example in DSSSL syntax looks like the Common Lisp version, but with `&` changed to `#!` and a colon in the application moved to the end of the keyword:

```
(define (rectangle width
                   #!optional (height width)
                   #!key (color "pink"))
  ....)

(rectangle 10 20 color: "blue")
```

DSSSL-style keyword and optional arguments is implemented by several Scheme implementations, including Bigloo (Serrano 2009), Chicken (Winkelmann et al. 2009), and Gambit (Feeley 2009), though details vary slightly. For example, **`#!key`** is a symbol in Chicken. Compared to our design, keyword and optional arguments in DSSSL have the same advantages and drawbacks as in Common Lisp.

7.3 SRFI-89

Like DSSSL, SRFI-89 distinguishes keyword values from symbols, uses a trailing `:` for the syntax of keywords, and keywords are self-quoting. Unlike DSSSL, SRFI-89 regularizes the syntax of procedures with keyword and optional arguments by making the procedure syntax more closely match the application syntax.

A keyword is associated with an argument in a procedure expression by placing the keyword before the formal argument; a small difference to our syntax is that the keyword and argument identifier are grouped by parentheses. An optional argument is declared by placing a default-value expression after the formal argument, and then grouping the two with parentheses. A keyword argument can be required, or it can be made optional by adding a default-value expression after identifier, within the parentheses that group it with the keyword.

The **`rectangle`** example could be written with SRFI-89 as follows:

```
(define (rectangle width
                   (height width)
                   (color: color "pink"))
  ....)

(rectangle 10 20 color: "blue")
```

Our design mostly imitates the SRFI-89 syntax, because we value the syntactic similarly of declarations and applications. We depart from SRFI-89 syntax in not grouping a keyword with a formal argument in a procedure declaration, because that change further strengthens the similarity to applications (where a keyword and its argument expression are not grouped with parentheses).

SRFI-89 separates a rest argument from keyword arguments; an argument is consumed either as a keyword argument or collected into the rest argument, but never both. SRFI-89 also generalizes keyword support by allowing keyword arguments to appear before by-position arguments. Unlike our design, however, keywords are either grouped together before by-position arguments or together after by-position arguments, and the order for a given procedure is determined by the procedure declaration. A drawback of this approach is that callers of a procedure must remember which order is used for a given procedure. Our design more completely separates by-position and by-keyword arguments, so that keyword arguments can always appear in any order relative to by-position arguments.

As in Common Lisp and DSSSL, optional- and keyword- argument handling is defined in terms of post-processing a sequence of by-position arguments, where keyword tags are mingled with argument values. As a result, it suffers from the many of the same problems in terms of accidental treatment of a keyword tag as a direct argument.

7.4 Implementation via Macros

Scheme macros support a portable implementation of optional and keyword arguments, although no such implementation has become widely used. One recent effort is Eddington's implementation for R6RS, which we used for performance measurements in Section 5. A lack of documentation for the library makes a detailed comparison difficult, but as we noted in Section 5, the library supplies a **`define/kw`** form for binding names that resolve keyword arguments statically. Having no syntactic distinction between keywords as expressions and keywords as argument tags, however, makes the library's static resolution inconsistent with its dynamic resolution.[4]

A variant of our design appears to be possible as a portable implementation using macros. Keywords could be identified through a **`keyword`** form that signals a syntax error when used as an expression, while an explicit **`with-keyword`** form would serve the role of a keyword-allowing application form that detects **`keyword`** tags. The combination of **`keyword`** and **`with-keyword`** enables the distinction between keyword tags and argument expressions, though it is syntactically more verbose than a built-in syntax of keywords or allowing **`#%app`** to be refined. To allow a procedure with optional keywords to be called through a normal application form, keyword-accepting procedures would be represented as plain procedures (since Scheme standards do not include applicable structure types); the protocol for supplying keyword arguments could use a special value as a regular argument to indicate that certain other arguments provide lists of keywords and associated values.

7.5 Ada, Python, and OCaml

Every function in Ada or Python supports keyword arguments, where the name of each formal argument doubles as the keyword for the argument. In a function call, by-position arguments are provided first and matched to formal arguments in order, and then keyword arguments can appear (in any order) to supply values for the remaining arguments. As in our design, keyword arguments are syntactically distinguished from by-position arguments in a function call. Unlike our design, by-position arguments must be supplied first.

Ada's double role for every formal argument as both a by-position and by-keyword argument is different from Lisp and Scheme systems, where formal argument names are purely local. Exposing all argument names as keywords in Scheme conflicts with other important aspects of the language, such as alpha renaming. A workable syntax might have the programmer annotate identifiers that should double as by-position and by-keyword arguments.

OCaml supports *labels* on function arguments that are similar to Ada's keyword arguments. A programmer explicitly designates labeled formal arguments using $\sim$ on the argument (and, optionally, a label that is separate from the argument's local identifier). The label of an argument becomes part of the function's type, which means that the compiler can always statically adjust the order of labeled arguments in a function call—even changing the order of curried applications to match the declaration order. Labelled arguments also can be optional (which, again, is exposed in the type of the function).

Finally, the PLT Scheme class system behaves much like Ada, in that class initialization arguments (i.e., constructor arguments) are usually supplied by name, but they can also be supplied by position. If arguments are supplied by position, the order of the names in the class declaration is used to match them with arguments values. This design pre-dates general keyword support in PLT Scheme, and it mainly provided backward compatibility with a previous iteration of the class system that supported only by-position initialization arguments.

Allowing keyword arguments to be supplied by position, as in Ada, conflicts somewhat with allowing keyword arguments in

[4] `http://groups.google.com/group/ikarus-users/browse_thread/thread/fb3a813c198311ff`

any order relative to by-position arguments; perhaps sensible rules could be specified to govern a mixed order of arguments with and without keywords. Ada-style argument handling also conflicts with combining keyword arguments and a by-position rest argument. More generally, we have not found much need for passing keyword arguments by position.

7.6 Smalltalk

In Smalltalk, most methods arguments are tagged with names, but the tags are not keywords in the sense of this paper. The tag on a Smalltalk method argument is simply part of the method name that is interleaved with the arguments; the tags and arguments cannot be reordered, and individual arguments are not optional.

7.7 Keyword Lexical Syntax

A keyword in Common Lisp is prefixed with `:`. This choice of syntax is related to Common Lisp's notion of package-specific symbols, where the empty package name corresponds to the "keyword" package. Conceptually, keywords are self-quoting because all symbols in the keyword package are bound to themselves.

In DSSSL and many Scheme systems, a keyword is *suffixed* with `:`, instead of prefixed with `:`. To many programmers, the suffix better connects the keyword with its argument, while others argue that a prefix is more appropriate for a prefix-oriented language like Scheme.

PLT Scheme uses a `#:` prefix. Chicken also supports a `#:` prefix in addition to a `:` suffix, though the keyword in both cases is equivalent to a symbol. The `#:` choice is natural for Scheme, since a `#` is normally used to extend the reader syntax, and `:` is normally allowed in symbols (i.e., some symbols and identifiers in existing code might break if a `:` prefix or suffix becomes the syntax of keywords). Many argue, however, that `#:` looks too heavy, while the whole point of keywords is arguably to add a lightweight grouping syntax to the language (i.e., lighter weight than parentheses). Also, Common Lisp uses the prefix `#:` for uninterned symbols.

We can offer no rationale that will resolve the debate. We chose `#:` because it broke no existing code and because at least one author likes how it stands out. An informal poll among PLT Scheme users suggested roughly equal support for all three choices (prefix `#:`, prefix `:`, and suffix `:`) with a slightly higher preference for `#:`—possibly reflecting the syntax that is already in place. In any case, PLT Scheme's `#lang` notation would allow future modules to be written using a different syntax without affecting old modules.

8. Conclusion

Scheme's "rest" arguments and **`case-lambda`** allow flexible handling of procedure arguments, and they easily accommodate keyword-like patterns using symbols and lists. When a pattern is used widely enough, however, converting the pattern to a language construct offers many advantages: better readability, clearer documentation, better error messages, easier composition of libraries, and a central point of control for implementation details of the pattern. For all of these reasons, we believe that specific constructs for keyword and optional arguments are appropriate for dialects of Scheme.

The essential elements of our design are (1) keywords that are distinct from symbols, as in many Scheme systems, (2) a form for creating keyword-based procedures that matches the application syntax, similar to SRFI-89, (3) disallowing unquoted keywords as literal expressions, which is novel in our design, and (4) passing keyword arguments to a procedure in a way that reliably separates them from by-position arguments, which is also novel.

Bibliography

John Clements and Matthias Felleisen. A Tail-Recursive Machine with Stack Inspection. *ACM Trans. Programming Languages and Systems* 26(6), pp. 1029–1052, 2004.

William D. Clinger et al. Larceny. 2009. `http://www.ccs.neu.edu/home/will/Larceny/`

Marc Feeley. SRFI-89: Optional Positional and Named Parameters. 2007.

Marc Feeley. Gambit v4.4.3. 2009. `http://www.iro.umontreal.ca/~gambit/`

Robert Bruce Findler and Matthew Flatt. Slideshow: Functional Presentations. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 224–235, 2004.

Franz, Inc. Allegro CL. 2009. `http://www.franz.com/`

Abdulaziz Ghuloum. Ikarus Scheme v3.0+. 2009. `http://ikarus-scheme.org/`

ISO. Document Style Semantics and Specification Language (DSSSL). ISO/IEC 10179:1996, 1996.

SBCL. 2009. `http://sbcl.sourceforge.net/`

Peter Seibel. *Practical Common Lisp*. Apress, 2005.

Manuel Serrano. Bigloo v3.2b-2. 2009. `http://www-sop.inria.fr/mimosa/fp/Bigloo/`

Andrew Shalit. *The Dylan Reference Manual*. Addison-Wesley, 1996.

Michael Sperber (Ed.). The Revised ^{6} Report on the Algorithmic Language Scheme. 2007.

Guy L. Steele Jr. *Common Lisp: The Language*. Second edition. Digital Press, 1990.

Felix Winkelmann, Kon Lovett, and Leonard Frank (elf). Chicken v4.0.0. 2009. `http://www.call-with-current-continuation.org/`

Appendix

Implementation of optional arguments in terms of **`case-lambda`**:

```
(define-syntax lambda
  (syntax-rules ()
    [(lambda (arg ... . rest) . body)
     (letrec ([f (case-lambda* f (arg ...) () ()
                               rest body)])
       f)]))

(define-syntax case-lambda*
  (syntax-rules ()
    [(case-lambda* f () (id ...) (clause ...)
                   rest body)
     (case-lambda clause ...
                  [(id ... . rest) . body])]
    [(case-lambda* f ([opt-id default-expr]
                      . rest-args)
                   (id ...) clauses rest body)
     (case-lambda* f rest-args (id ... opt-id)
                   ([(id ...)
                     (f id ... default-expr)]
                    . clauses)
                   rest body)]
    [(case-lambda* f (req-id . rest-args)
                   (id ...) clauses rest body)
     (case-lambda* f rest-args (id ... req-id)
                   clauses rest body)]))
```