λ* : Beyond Currying

Jason Hemann Daniel P. Friedman
Indiana University
{jhemann,dfried}@cs.indiana.edu

Abstract

Though the currying of functions has a long and storied history, the technique has found less acceptance in Scheme than one might at first imagine. We present a function-creation mechanism λ^* that, while maintaining the functionality of Scheme's λ, provides enhanced features of function abstraction and application, including automatic currying of functions, automatic nesting of applications, and reasonable behavior when over- or under-supplied with arguments.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Procedures, functions, and subroutines

Keywords currying, partial application, macro, Scheme

1. Introduction

The concept of currying functions is not new. The term "currying" was coined by Christopher Strachey in 1967 for Haskell Curry who used it extensively [4, 23], though the idea was known to Moses Schönfinkel by at least 1920. In his honor, the name "schönfinkeling" has also been used [2]. Quine, writing of Schönfinkel's work [20], describes the concept thusly:

> All functions, propositional and otherwise, are for Schönfinkel one-place functions, thanks to the following ingenious device (which was anticipated by Frege (1893, §36)). Where F is what we would ordinarily call a two-place function, Schönfinkel reconstrues it by treating Fxy not as $F(x, y)$ but as $(Fx)y$. Thus F becomes a one-place function whose value is a one-place function. The same trick applies to functions of three or more places; thus "$Fxyz$" is taken as "$((Fx)y)z$".

The technique is deeply rooted in category theory [1] and is widely used in languages that support "functions as values". In both Haskell and ML, for instance, all functions are implicitly (automatically) curried.

Currying is, however, not so ubiquitous in Scheme. The documentation for SRFI-26, which provides operators for a form of partial application distinct from that possible with traditional currying, suggests reasons this might be the case. The Design Rationale states that Scheme is "not prepared to deal with curried procedures in a convenient way" and that "[t]he primary relevance of currying/uncurrying in Scheme is to teach concepts of combinatory logic" [10]. We explore these arguments further in Section 2.

In this paper, we suggest that there may yet be a reasonable way to work with curried procedures in Scheme and that they may have some practical use. To do so, we develop systematically a mechanism for implicitly currying Scheme functions. This mechanism also provides partial application of functions, consistent behavior for an excessive quantity of arguments, and consistent behavior for variadic and nullary functions that we believe is consonant with such functions created with Scheme's λ.

2. Currying, `curry`, and `$`

Currying began as a technique of mathematical logic, but has found use in programming. Although there are $n!$ ways to curry an n argument function, here when we refer to currying we always mean currying from the left. Following the terminology of Moreno et. al [12], we describe the application of a curried function as *partial*, *exact*, or *exceeding* depending whether the arguments provided to it are less than, equal to, or greater than the number of bindings in the list of bindings in the uncurried form.

We use the following conventions to describe function arity. Formals may be proper or improper. Proper formals denote functions of fixed arity. Functions with the empty formals list are called *nullary*. We use the term *unary* for functions of precisely one argument, and reserve *polyadic* to mean functions of a fixed arity at least two. We describe those functions with a fixed, positive arity (unary and polyadic functions collectively) as *posary*. A single variable or an improper list of formals denote functions of variable arity. We call functions whose formals is a single symbol *variadic*, and use the term *polyvariadic* to mean functions of variable arity mandated to have at least one element. This discussion of the five non-overlapping formal parameter structures is summarized in Table 1.

By always currying from the leftmost argument, curried functions provide, for free, partial application in the left argument. Curried polyadic functions present opportunities to η-reduce that would have been otherwise impossible due

Proper		Improper	
nullary	`()`	variadic	`y`
unary	`(x)`	polyvariadic	`(x+ . y)`
polyadic	`(x+ y)`		

Table 1. Formals of functions

to arity mismatch. Where available, taking advantage of opportunities to η-reduce may make for shorter, clearer code. The verification of programs, either formally or informally, may also be aided by the code that currying can help create.

A straightforward implementation of an explicit mechanism for currying functions in Scheme is of limited value though.

```
(define-syntax curry
  (syntax-rules ()
    ((_ (x . x*) e) (λ (x) (curry x* e)))
    ((_ () e) e)))
```

Figure 1. A currying macro that takes a list of bindings and returns nested functions of one binding each.

The macro in Figure 1 takes as arguments a non-empty list of bindings and a body, akin to a subset of the functionality of Scheme's λ. This macro fails to function as one might hope when passed the empty list of bindings. Here, and for the remainder of this discussion, we presume valid data, though the need for this assumption is diminished as the discussion continues.

It transforms the expression into one with the same body, but nested within unary λs in the number of bindings. This provides the ability to write functions as usual, but to ensure that they now take arguments one at a time. This is similar to a number of other implementations of currying [15, 18]. The resultant functions must be invoked a single argument at a time as in the following example, whose value is 5.

```
((((((curry (a b c d e) e) 1) 2) 3) 4) 5)
```

Due to the highly parenthetical syntax of Lisp-like languages, currying in Scheme is typically viewed as somewhat inelegant. While in some instances it might be desirable to pass arguments one at a time, it is often more readable and more edifying to pass all arguments at once, or in a couple of distinct groupings. Passing arguments one at a time causes the code to take up more room than it otherwise would. Too, this verbosity may obscure understanding of the code's operation.

A "nested application" macro, here named `$`, relieves the burden of applying curried functions manually. The macro `$` applies arguments one at a time to a chain of nested functions until the list of arguments is exhausted. As with `curry` above, there are many implementations of `$`.

```
(define-syntax $
  (syntax-rules ()
    ((_ f e) (f e))
    ((_ f e e* ...) ($ (f e) e* ...))))
```

Figure 2. A nested application macro, which passes each of a list of arguments to a function until the list is exhausted.

While code written using both `curry` and `$` is more compact than with `curry` alone, it is unfortunate that we now require a macro for both abstraction and application.

```
($ (curry (a b c d e) e) 1 2 3 4 5)
```

Moreover, while this suffices as an implementation of currying in Scheme, this is far from ideal. For one, these mechanisms offer no support for nullary or variadic λs. Secondly, the above example suggests that with this mechanism, currying is primarily useful only as part of a "curry, specialize, uncurry" pattern: precisely the behavior SRFI-26 provides, only more generally.

All of this seems to explain the discussions that took place prior to ratification of SRFI-26, or at least shed more light on certain comments. From the above, one might indeed be led to conclude "Scheme is not prepared to deal with nested single-argument functions", and that, apart from the "curry, specialize, uncurry" pattern, "[c]urrying is pointless in Scheme" [9].

3. Development of λ^*

It may be that the identified problems are not endemic to currying in Scheme *per se*, but perhaps just to particular implementations. In this section we develop iteratively, through eight increasingly more general variants, a mechanism that meets the above objections.

A mechanism for currying functions does not require an operator like `$`. The λ^* macro in Variant 1, with the associated function `app*` below, provides curried function definitions while allowing arguments to be passed as usual.

```
(define-syntax λ*
  (syntax-rules ()
    ((_ () e) e)
    ((_ (x . x*) e)
     (λ (x . a*) (app* (λ* x* e) a*)))))

(define app*
  (λ (f a*)
    (if (null? a*)
      f
      (app* (f (car a*)) (cdr a*)))))
```

Variant 1: This definition of λ^* uses `app*` which is recursively defined over `f` and `a*`.

This allows for the added flexibility that comes with curried functions, while at the same time functions can be created and applied using the relatively more succinct syntax of polyadic functions. From this follows short, flexible code and advantages that come with η-reducing.

λ^* is not, strictly speaking, *actually* currying the bindings. Instead, it returns a function which, when supplied a list of arguments, *behaves* as if it were curried. When supplied a list of bindings, which must be non-empty, the second clause is matched and λ^* expands to a function expecting one or more arguments. The first argument it is provided is bound to the first variable in the bindings. Any remaining arguments are successively passed to nested functions. When there are no more bindings, the first clause is matched, and the body itself is returned. The first clause must only be reached in the base of a recursion, and any remaining arguments are passed to the body. In that case the body must be a unary function in order to avoid signaling an exception.

The use of `x` as both a pattern variable and as the first argument to the function returned in the macro is deliberate. The argument to the function is bound to `x`, which is scoped over the entirety of the body, including nested functions. The first argument is bound to `x` in the body by virtue of hygiene in the syntactic rules system. The application of additional arguments is performed by `app*` and the remaining arguments are successively applied to nested functions. If the function is partially applied, the value is a function expecting the remainder of its arguments; if the function is exactly applied, the result is the value of the body; and if it is excessively applied, the excessive arguments are passed to the value of the body.

```
> ((λ* (a b c) c) 1 2 3)
3
> (((λ* (a b c) c) 1) 2 3)
3
```

These examples demonstrate uses of λ^*. Arguments can be supplied at once as with functions defined by λ, or in groups of one or more in whatever manner is most convenient. Unlike the earlier implementation, we make no use of a special application form like `$`.

Because functions created with λ^* apply any remaining arguments to the body one at a time, we can supply arguments to functions in what seems to be an arity mismatch, but in fact produces the correct answer. Moreover, since excessive arguments are passed to the body one at a time, we can nest unary Scheme λs inside a λ^*.

```
> (((λ* (a b) (λ* (c) c)) 1) 2 3)
3
> ((λ* (a) (λ (b) (λ (c) c))) 1 2 3)
3
```

This behavior is only available for Scheme λs of a single argument. In the example below, for instance, attempting to provide arguments to the body one at a time results in an exception, as the Scheme λ must have exactly two arguments.

```
> ((λ* (a b) (λ (d e) e)) 1 2 3 4)
exception
```

Worse, this definition of λ^* is afflicted with a subtle, more insidious flaw: the code generated is not tail recursive. The following is an example of a program that, with Scheme's λ loops indefinitely, but when using λ^* instead consumes all available memory.

```
> (letrec ((fn (λ* (x) (fn x)))) (fn 1))
out of memory
```

Expanding `(λ* (x) (fn x))`, the definition of `fn`, reveals the mistake. With the λ^* macro this will be transformed into `(λ (x . a*) (app* (fn x) a*))` where, unlike with Scheme's λ, the invocation of `(fn x)` has become a non-tail call. When invoked as above, those waiting calls accumulate and lead to unbounded memory consumption.

The implementation of λ^* in Variant 2 remedies both of those problems. By using Scheme's `apply` in the definition of `app*th` below, we gain the ability to use polyadic Scheme functions as the body of an excessively-applied λ^*. In order to ensure our programs are tail recursive, we delay the recursive calls by *thunking* them, and invoke the thunks in `app*th`.

```
(define-syntax λ*
  (syntax-rules ()
    ((_ () e) e)
    ((_ (x . x*) e)
     (λ (x . a*)
       (app*th (λ () (λ* x* e)) a*)))))

(define app*th
  (λ (th a*)
    (if (null? a*) (th) (apply (th) a*))))
```

Variant 2: A redefinition of λ^* and a new definition `app*th`. Instead of calling itself recursively, `app*th` uses Scheme's `apply`, and λ^* thunks the recursive calls to ensure tail recursion.

The list of bindings is processed one at a time, creating a series of nested functions, the body of each being a call to `app*th`. The first argument of all but the innermost is a thunk whose body is the recursive call processing the remaining elements of the list of bindings. In the base case, where there are no more bindings, the body of the thunk is simply the expression passed as the body of λ^*. When the list is exhausted, the result is simply the body. Because we thunk recursive calls, calls to λ^* are guaranteed to be tail-recursive. And because Scheme's `apply` applies a function to a list of arguments, and the functions created by λ^* are prepared to take a list of arguments of at least size one, we retain the earlier behavior of nested λ^*s. At the same time, a Scheme λ of two arguments that is the body of a λ^* can now

be supplied two arguments directly, and the earlier example loops indefinitely. This implementation can no longer continually supply the excess arguments of a λ^* successively to nested unary Scheme λs. This behavior, remains for nested unary functions constructed with λ^*, so effectively we lose no functionality by using the `app*th` of Variant 2.

Variant 2 now has the ability to pass arguments to posary functions, and has the ability to partially apply λ^* functions to one or more arguments. However, this ability has come at a steep price. In this implementation, for a series of functions built from λ^* with n total formal parameters to become exactly applied, Scheme's `apply` will have been called n times. This is an unnecessary cost. In Variant 3 `apply` is called in the number of formal parameters only in the worst case.

```
(define-syntax λ*
  (syntax-rules ()
    ((_ (a a* ...) e)
     (letrec
       ((rec
          (case-λ
            ((a a* ...) e)
            ((a a* ... . rest)
             (apply (rec a a* ...) rest))
            (some (get-more rec some)))))
        rec))))

(define get-more
  (λ (rec some)
    (λ more
      (apply rec (append some more)))))
```

Variant 3: This definition of λ^* invokes a function capable of taking its entire list of arguments for a partial application, which often decreases the number of calls to `apply`.

The number of formal parameters required is clear at the time of function definition. Using `case-`λ [8, 21] we in effect case on the number of actual arguments. In the first case, the quantity is exactly that demanded by the function. In the second the function is exceedingly applied, and the third case of necessity matches argument lists shorter than that demanded by the function. In the first case, the formal parameters are paired with the arguments by virtue of hygene; we simply return the body of the function. In the second case, we invoke the function on exactly the quantity of arguments which saturate it, and since there are more arguments to be invoked, we apply the remainder to the value of that invocation, which ought to return a procedure. In the third case, the number of arguments is fewer than the number of formal parameters, and so we return the function that results from a call to `get-more`. This function awaits more arguments, and when receiving some will append them to the list of arguments already received before applying `rec` to this newly augmented list of arguments.

This has in addition endowed it the ability to partially apply a function to *no* arguments. It seems that, just as partial application of one or more arguments is meaningful, the partial application of a function with no arguments ought to return a function expecting exactly as many arguments as before. However, the cost here is also greater than one might wish. A slight modification to the prior variant gives that capability without requiring the expensive `append` by taking advantage of the fact `rec` is already recursively defined (Variant 4). We add a fourth case to `rec`, where it receives no arguments, and simply returns the definition of `rec`.

```
(define-syntax λ*
  (syntax-rules ()
    ((_ (a a* ...) e)
     (letrec
       ((rec
          (case-λ
            (() rec)
            ((a a* ...) e)
            ((a a* ... . rest)
             (apply (rec a a* ...) rest))
            (some (get-more rec some)))))
        rec))))
```

Variant 4: We special-case the application of no arguments, so as to avoid the expense of an `append`.

This definition of λ^* builds functions of one or more parameters that provide what we believe to be reasonable behavior when supplied zero or more arguments. It does so regardless of whether the number of arguments is less than, more than, or exactly as many arguments as the number of parameters the function appears to expect. In implementing this version of λ^*, we use polyvariadic formals-lists in `case-`λ, which is the lynchpin of the macro's operation.

Our λ^* macro itself, though, provides no support for improper lists of formals. Although we can use Scheme's λ for this purpose and λ^* otherwise, we instead augment λ^* with support for polyvariadic functions.

Here, the definition of λ^* from Variant 4 is renamed `posary-h`. When λ^* in Variant 5 receives the arguments for a function definition with a non-empty proper list of formals, it passes them to `posary-h`. The second clause of the new λ^* accounts for improper lists of formal parameters, and simply passes them along to the newly-defined `polyvariadic-h`. The `polyvariadic-h` macro is similar to `posary-h`. In the case of an exact match (that is, when enough actual arguments supplied to saturate `a a*`..., and perhaps more), then the body of the function is simply applied. The other two cases are the same as the corresponding `case-`λ clauses in the `posary-h` macro.

With this definition we provide support for posary and polyvariadic functions. The following examples indicate that the addition of this clause does not diminsh the ability to

```
(define-syntax λ*
  (syntax-rules ()
    ((_ (a a* ...) e) (posary-h (a a* ...) e))
    ((_ (a a* ... . rest) e)
     (polyvariadic-h (a a* ... . rest) e))))

(define-syntax posary-h
  (syntax-rules ()
    ((_ (a a* ...) e)
     (letrec
       ((rec
         (case-λ
           (() rec)
           ((a a* ...) e)
           ((a a* ... . rest)
            (apply (rec a a* ...) rest))
           (some (get-more rec some)))))
       rec))))

(define-syntax polyvariadic-h
  (syntax-rules ()
    ((_ (a a* ... . rest) e)
     (letrec
       ((rec
         (case-λ
           (() rec)
           ((a a* ... . rest) e)
           (some (get-more rec some)))))
       rec))))
```

Variant 5: The prior version is renamed `posary-h`, and a new macro `polyvariadic-h`, which is similar in spirit, but intended to account for functions with improper formals lists, is added. Both are called from the new λ^*, which merely chooses between these two cases.

use Scheme's variadic λs with λ^*, but show that we now no longer need to use them.

```
> ((λ* (a) (λ (b . c) c)) 1 2 3 4)
(3 4)
> ((λ* (a b . c) c) 1 2 3 4)
(3 4)
```

Nullary and variadic functions must still be built with λ, however. Different behavior is required for the completion of processing a list of bindings and building a function with an empty list of bindings or a symbol instead of a list. Nullary and variadic functions built with Scheme's λ can be intermixed with those built by λ^*. It is more desireable to have a single function-creation mechanism, one by which nullary and variadic functions can be built as well. In order to do so, we add two more clauses to the λ^* macro from above.

This variant allows us to construct from within λ^* nullary functions identical in behavior to those built by Scheme's λ. Upon reflection, though, we would like nullary functions

```
(define-syntax λ*
  (syntax-rules ()
    ((_ () e) (λ () e))
    ((_ (a a* ...) e) (posary-h (a a* ...) e))
    ((_ (a a* ... . rest) e)
     (polyvariadic-h (a a* ... . rest) e))
    ((_ a* e) (λ a* e))))
```

Variant 6: The λ^* from Variant 5 is extended with clauses which match nullary and variadic functions.

to also, when given arguments, pass those arguments to the body. We believe this is a natural extension of the behavior of λ^* for functions of other arities, and maintains the behavior of Scheme's λ when provided no arguments. With a slight modification of the first clause of Variant 6 of λ^*, we make this possible as well.

```
(define-syntax λ*
  (syntax-rules ()
    ((_ () e)
     (λ a* (if (null? a*) e (apply (e) a*))))
    ((_ (a a* ...) e) (posary-h (a a* ...) e))
    ((_ (a a* ... . rest) e)
     (polyvariadic-h (a a* ... . rest) e))
    ((_ a* e) (λ a* e))))
```

Variant 7: When invoked with arguments, nullary functions pass those arguments to the body.

This new definition of λ^* in Variant 7 ensures that the following examples now run.

```
> (((λ* () (λ* (a b c) c))) 1 2 3)
3
> ((λ* (a) (λ* () (λ* (d . e) e))) 1 2 3)
(3)
```

Heretofore we have restricted the λ^* macro's behavior to that of generating functions of a single body. Nothing is lost, however, by generalizing to functions of one or more bodies as in Variant 8. To do so, we treat the penultimate definition of λ^* as a helper macro, renamed λ^*-`h`, which is called from λ^*. In doing so, we equip λ^* with the capabilities of Scheme's λ, and more. The complete definition of λ^* is presented in Variant 8.

This definition of λ^* acts as a near drop-in replacement for Scheme's λ. We conjecture that, modulo exception-handlers, previously functioning programs should still work, and future programs can be written with the additional power provided by λ^* of Variant 8.

4. All that glitters ...

We have spent some time describing the benefits of this tool, and we would be remiss to continue without describing examples of its use that might give one pause. The examples

```
(define-syntax λ*
  (syntax-rules ()
    ((_ a* e* ...)
     (λ*-h a* (let () e* ...)))))

(define-syntax λ*-h
  (syntax-rules ()
    ((_ () e)
     (λ a* (if (null? a*) e (apply (e) a*))))
    ((_ (a a* ...) e) (posary-h (a a* ...) e))
    ((_ (a a* ... . rest) e)
     (polyvariadic-h (a a* ... . rest) e))
    ((_ a* e) (λ a* e))))

(define-syntax posary-h
  (syntax-rules ()
    ((_ (a a* ...) e)
     (letrec
       ((rec
         (case-λ
           (() rec)
           ((a a* ...) e)
           ((a a* ... . rest)
            (apply (rec a a* ...) rest))
           (some (get-more rec some)))))
       rec))))

(define-syntax polyvariadic-h
  (syntax-rules ()
    ((_ (a a* ... . rest) e)
     (letrec
       ((rec
         (case-λ
           (() rec)
           ((a a* ... . rest) e)
           (some (get-more rec some)))))
       rec))))

(define get-more
  (λ (rec some)
    (λ more
      (apply rec (append some more)))))
```

Variant 8: λ^*

below demonstrate uses of λ^* that, while reasonable upon reflection, seems slightly odd at first glance.

```
> ((λ* (a) (λ* () (λ* () (λ* (b) b)))) 1 2)
2
> (((((((((λ* (a b c) 4)))))))) 1 2 3)
4
```

The application of four arguments to the function proceeds right past the two `(λ* () ...)`. Since any function can now be partially applied to no arguments, and any arguments provided to expressions of the form `(λ* () ...)` are passed to the body, nullary λ^* functions seem to be of little to no use. Likewise, passing no arguments to a function demanding some returns the original function, and this action can be repeated over and over again. While these behaviors are decidedly not *wrong*, they are perhaps mildly unsettling.

Programs which rely on arity mismatches to signal an exception in the course of their correct functioning, for instance, might not signal an exception as before. Programs relying on such behavior may no longer work as expected.

Equally troubling, perhaps, this more general notion of function definition and application may make small mistakes more difficult to uncover. When reading code, the bindings of a function definition no longer indicate how many arguments it could or should be provided. That level of obscurity demonstrates the utility of static typing and an associated typechecker, such as in Typed Racket [22].

5. Related Works

The technique and practice of currying itself has been known since before the advent of computers. Those interested in early works on techniques of currying itself should look to the works of Schönfinkel [20] and Curry [5]. An overview is presented by Rosser [19], and a thorough history of the lambda-calculus and combinatory logic is provided in Cardone and Hindley [3]. Those interested in the mathematical background of partial application, including the *s-m-n* theorem, are referred to Kleene [13] and Rogers [6].

We were surprised to find so many interesting implementations as regards currying in a language like Scheme. Many of those are limited to transforming functions of n arguments into n nested unary functions, in the style of `curry` above. A number of currying macros similar in respects to those developed herein are mentioned below. Meunier, in addition to an explicit currying macro similar to `curry`, also presents an implicit currying macro [15]. His curry macro checks that the application is partial or exact; if partial, it returns a function whose arity is equal to the number of arguments yet to be received. One has the implicit currying and returns the same procedure when invoked with no arguments. Piet Delport's version processes the bindings as in λ^*, but does not handle variadic functions or excessive application [7, 16]. Kmett extends Delport's version to account for the partial application of no arguments and excessive applications [14]. Kmett's is similar to λ^*, though λ^* also handles variadic functions. The Racket function `curry` allows currying of previously-built functions, and provides the option to use keyword arguments [11].

The currying macros developed here, along with those just described, provide for partial application in the leftmost argument. SRFI-26 provides the `cut` and `cute` forms that allow for partial application of any of a function's arguments, rather than just the leftmost one [10]. In instances where we might like to at different times partially apply different arguments to the same function, we cannot rely on using the leftmost argument position. With an exponentiation function, for instance, at different times one might wish to partially

apply to it the base or the index. The forms `cut` and `cute` provide the ability to the same function for both of those partial applications, which currying and λ^* cannot match. Haskell's sections provide an operation similar to `cute` for infix binary operations [17].

6. Conclusions and Future Work

The examples of the use of λ^* seem to suggest that there are in fact uses of a form of currying in Scheme, and that λ^* may be a practical tool.

This mechanism allows for the behavior one would gain from currying Scheme functions. It does so without having to perform currying in the strictest sense and yet it provides exactly the sort of partial application of functions that currying allows. Functions defined with λ^* can also be called as though they were polyadic Scheme functions built with λ, or as some mixture of the two. In addition, λ^* performs a similar sort of curry-like behavior for polyvariadic functions, and provides meaning for excessively-applied functions as well as partially-applied functions.

One goal of future research is practical programming experience in a language with λ^*-style functions as the primary function definition construct. We also seek to investigate λ^* in the context of types. Too, we hope to mitigate the performance penalty of Scheme's `apply`, either by implementing λ^* in a language without the side-effects that induce the copying of arguments, or special-casing certain common arities so as to not incur the cost as frequently.

We believe that λ^* should be considered a useful mechanism that provides additional tools to the programmer's toolbox. Moreover, it should be taken as evidence of the importance of currying in Scheme, and beyond.

Acknowledgments

We are grateful to Andre Kuhlenschmidt, Chung-chieh Shan, Cameron Swords, Sam Tobin-Hochstadt, and Mitch Wand for their discussions and suggestions, which in several places were critical to the development of the present work.

References

[1] T. Bartels. Currying, May 2010. URL `http://ncatlab.org/nlab/show/currying`.

[2] C. Bozsahin. *Combinatory Linguistics*. Walter de Gruyter, 2012.

[3] F. Cardone and J. R. Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5, 2006.

[4] H. Curry. Some philosophical aspects of combinatory logic. In *The Kleene Symposium*, pages 85–101. North Holland, Amsterdam, 1980.

[5] H. B. Curry and R. Feys. *Combinatory logic, Vol. I*. Studies in Logic and The Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[6] M. Davis. *Computability & unsolvability*. McGraw-Hill (New York), 1958.

[7] P. Delport. curried.scm, September 2013. URL `https://gist.github.com/pjdelport/6535964`.

[8] R. K. Dybvig and R. Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation*, 3(3):229–244, 1990.

[9] S. Egner. Changes to the design of srfi-26, February 2002. URL `http://srfi.schemers.org/srfi-26/mail-archive/msg00016.html`.

[10] S. Egner. Notation for specializing parameters without currying, June 2002. URL `http://srfi.schemers.org/srfi-26/srfi-26.html`.

[11] M. Flatt and PLT. The racket reference, 2013. URL `http://docs.racket-lang.org/reference/`.

[12] J. C. González-Moreno, M. T. Hortal-González, and M. Rodríguez-Artalejo. *Denotational Versus Declarative Semantics For Functional Programming*, pages 134–148. Springer Berlin Heidelberg, 1992.

[13] S. C. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[14] E. Kmett. Curried scheme, August 2009. URL `http://comonad.com/reader/2009/curried-scheme/`.

[15] J. A. Meunier. Function currying in scheme, March 1997. URL `http://www.engr.uconn.edu/~jeffm/Papers/curry.html`.

[16] J. A. Ortega-Ruiz. Scheme code capsule: currying, February 2007. URL `http://programming-musings.org/2007/02/03/scheme-code-capsule-currying/`.

[17] S. L. Peyton-Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[18] Programming Praxis. Standard prelude, 2013. URL `http://programmingpraxis.com/contents/standard-prelude/#higher-order-functions`.

[19] J. B. Rosser. Highlights of the history of the lambda-calculus. *Annals of the History of Computing*, 6(4):337–349, 1984.

[20] M. Schönfinkel. On the building blocks of mathematical logic. In *From Frege to Gödel*, pages 355–366. Harvard University Press, 1924.

[21] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[22] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. *Proc. Symposium on Principles of Programming Languanges*, 2008.

[23] D. Turner. Currying, or schonfinkeling?, 1997. URL `http://computer-programming-forum.com/26-programming-language/976f118bb90d8b15.htm`.