Typed Scheme: Bringing Types to Untyped Languages

Sam Tobin-Hochstadt

September 1, 2008

Abstract

Programming languages has recently experienced a renaissance, especially in the field of untyped scripting languages. But when scripts written in untyped languages grow large, they may also grow difficult to maintain. To improve the maintainability of untyped language programs, we propose porting portions of them into typed languages. To validate the feasability of this approach, we have developed Typed Scheme, a typed sister language to PLT Scheme. Typed Scheme provides smooth and sound interoperability with untyped PLT Scheme. It features a type system that supports idiomatic Scheme programming, so that the porting process is relatively straightforward.

1 The Challenge of Untyped Languages

Under the heading of "scripting languages", a variety of new, untyped, programming languages has become popular, and even pervasive, in web- and systems-related fields [3, 9, 25, 28, 41, 43]. As a result, programmers often create scripts that then grow into large applications. Programmers are also beginning to notice, however, that untyped scripts are difficult to maintain over the long run. The lack of types means a loss of design information that programmers must recover every time they wish to change existing code.

One possible solution is to rewrite the program in a typed language. This requires vast investment of time and resources. It also imposes heavy transition costs. Maintenance must be performed on two systems during the transition, and successfully reimplementing all features in a new language is extremely difficult. Also, the programmers must adjust to a new language and style of programming. Instead of this, I propose a gradual migration from untyped to typed code.

This brings me to my thesis:

Module-by-module porting of code from an untyped language to a typed sister language allows for an easy transition from untyped scripts to typed programs.

In support of this thesis, I have developed Typed Scheme [7, 36, 38, 39], a typed sister language of PLT Scheme. In the remainder of this document, I describe Typed Scheme as well as the challenges that remain. First, I describe how Typed Scheme interoperates with untyped code, and the current design of the type system. Following that, I outline additional challenges for the type system that I plan to address, and I discuss empirical validation. The proposal concludes with related work and a schedule.

1.1 A Brief Introduction to PLT Scheme

Since Typed Scheme is built on top of and alongside PLT Scheme, I begin with a review of the basic outlines of PLT Scheme. Fundamentally, it is a dialect of Scheme [34], enriched with a sophisticated module and macro system, as well as numerous primitive operations including file system access, network transmission, and GUI construction.

PLT Scheme programs are written in modules, as follows:

```
#lang scheme ; book-module
(define-struct book (author title))
(provide (struct book))

#lang scheme/gui ; print-module
(require media-module)
; Displays information about books
(define (print m)
  (cond [(book? m)
         (message-box (book-author m) (book-title m))]
        [else (error 'bad-input)]))
```

This small examples demonstrates many of the most important features. The first line of each module specifies that what language the module is written in. Here, the first module is written in the `scheme` language, which provides a wide variety of functionality by default. The second is in the `scheme/gui` language, which includes graphical elements. The flexibility of specifying the language on a module-by-module basis is key for Typed Scheme.

The first module defines a `book` structure with two fields, using the `define-struct` form. It also `provide`s this structure to other modules.

The second module `require`s the first, gaining access to the earlier definition. It then uses the `match` macro, which provides pattern matching facilities, to destructure the input `book`. The GUI is then used to display the data. All of these features fit together to make a powerful and expressive language.

2 Interoperability between Typed and Untyped Modules

A key component of incremental migration from typed to untyped code is interoperbility between typed and untyped code. Programmers should be able to freely intermix typed and untyped modules, without needing to concern themselves with the ordering of dependencies. Typed Scheme therefore supports interoperation across the typed/untyped boundary, in both directions.

2.1 Importing from the Untyped World

When a typed module imports functions from an untyped module—say PLT Scheme's extensive standard library—Typed Scheme requires dynamic checks at the module boundary. Those checks are the means to enforce type soundness. In order to determine the correct checks and in keeping with our decision that only binding positions in typed modules come with type annotations, we have designed a typed import facility. For example,

```
(require/typed scheme/base add1 (Number -> Number))
```

imports the `add1` function from the `scheme/base` library with the given type. The `require/typed` facility expands into dynamically checked contracts, which are enforced as values cross module boundaries [11].

For a more complete example, consider the untyped module

```
#lang scheme
(define (sq x) (* x x))
(provide sq)
```

and the typed module

```
#lang typed-scheme
(require/typed sq (Number -> Number) square-lib)
(sq 5)
```

The second, typed, module declares the type that it expects the `sq` function to have, and a contract enforcing that type is applied at the boundary between modules. If we add the expression `(sq "five")` to the typed code, Typed Scheme signals a *static* type error. If we instead change the definition of `sq` to

```
(define (sq x) (if (= x 5) "whoops" (* x x)))
```

Then when running the typed module, we get a *dynamic* contract error, which blames the untyped module for delivering a string where a number is promised.

An additional complication arises when an untyped module provides an opaque data structure, i.e., when a module exports constructors and operators on data without exporting the structure definition itself. In these cases, we do not wish to expose the structure merely for the purposes of type checking. Still, we must have a way to dynamically check this type at the boundary between the typed and the untyped code and to check the typed module.

To support such situations, Typed Scheme provides a facility for creating opaque types, which require nothing but the predicate for testing membership. This predicate can be trivially turned into a contract, but no operations on the type are allowed, other than those imported with the appropriate type from the untyped portion of the program.

Here is a sample usage of the special form for importing a predicate and thus defining an opaque type:

```
(require/opaque-type Doc document? xml)
```

It imports the `document?` function from the `xml` library and uses it to define the `Doc` type. The rest of the module can now import functions with `require/typed` that use `Doc` as a type.

2.2 Exporting to the Untyped World

When a typed module is required by untyped code, other considerations come into play. Again, the typed code must be protected, but we already know the necessary types. Therefore, in untyped contexts, typed exports are automatically guarded by contracts, without additional annotation effort by the programmer.

Consider the above example, reversed:

```
#lang typed-scheme
(: sq (Number -> Number))
(define (sq x) (* x x))
(provide sq)

#lang scheme
(require square-lib)
(sq 5)
(sq "five")
```

The type of `sq`, `(Number -> Number)` is automatically converted to a contract (the same contract created in the earlier example) when `square-lib` is required into an untyped context. This contract allows the first use of `sq`, but `(sq "five")` produces a contract error, which blames the untyped module for applying a function to a string, even though its contract requires numbers.

Unfortunately, because macros allow unchecked access to the internals of a module, macros defined in a typed module cannot currently be imported into an untyped context.

2.3 Contracts and Type Soundness

Systems that consist of both typed and untyped code may, of course, signal errors concerning primitive operations at runtime. Our type system still provides guarantees about the typed portion of the code, however. These guarantees can be expressed using the notion of blame [11]. This insight gives the following generalization of the type soundness theorem.[1]

> *If e is a program with typed modules $\bar{t}$ and untyped modules $\bar{u}$, then if execution of e signals an error concerning primitive operations, that error blames some untyped module u_i.*

This theorem can be made fully formal and proved for a calculus representing the core of the Typed Scheme's interoperability features [38].

3 A Type System for Idiomatic Scheme

Typed Scheme ought to support typed refactoring with a type system that directly accommodates many of the standard Scheme programming idioms. In principle, Scheme programmers need only annotate structure and function headers with types to move a module into the Typed Scheme world; on occasion, they may also have to define a type alias to keep type expressions concise. On rare occasions, the must change the code of functions so that the type checker can verify the types. For this purpose, the type system combines true union types, recursive types, first-class polymorphic functions, and the novel discipline of occurrence typing. Additionally, Typed Scheme infers types for instantiations of polymorphic functions, based on locally-available type information.

3.1 Basic Typed Scheme

Disciplined Scheme programmers typically describe the structure of their data in comments. For example, a media data type, extending the `book` from earlier, might be represented as:

```
; Media is either a book or a cd
(define-struct book    (author title))
(define-struct cd      (artist tracks))
```

[1]The usual provisions to type soundness theorems apply. See footnote 2.

To accommodate this style in Typed Scheme, programmers can specify true, untagged unions of types:

```
(define-type-alias media  (U book cd))
(define-struct: book ([author : String] [title : String]))
(define-struct: cd   ([artist : String] [tracks : (Listof String)]))
```

Typed Scheme also supports explicit recursive types, which are necessary for typing uses of `cons` pairs in Scheme programs. This allows the specification of both fixed-length heterogeneous lists and arbitrary-length homogeneous lists, or even combinations of the two.

3.2 Occurence Typing

Typed Scheme introduces *occurrence typing*, which allows the types of variable occurrences to depend on their position in the control flow graph.

Consider the following illustrative program fragment, using the data defined above, which extends the earlier `print` function

```
; media -> Void
; Displays information about media
(define (print m)
  (cond [(book? m)
         (message-box (book-author m) (book-title m))]
        [(cd? m) (message-box "cd" (cd-artist m))]
        [else (error 'bad-input)]))
```

As the informal original data definition states, a `media` item is either a `book` or a `cd`.

The definition illustrates several key elements of the way that Scheme programmers reason about their programs: ad-hoc type specifications, true union types, and predicates for type testing. No datatype specification is needed to introduce the sum type on which the function operates. Instead there is just an “informal” data definition and contract [10], which gives a name to a set of pre-existing data, without introducing new constructors. Further, the function does not use pattern matching to dispatch on the union type. All it uses is a predicate that distinguishes the two cases: the first `cond` clause, which deals with the parameter `m` as a `book` and the second one, which treats it as a `cd`.

Here is the corresponding Typed Scheme code:

```
(: print (media -> Void))
(define (print m)
  (cond [(book? m)
         (message-box (book-author m) (book-title m))]
```

```
          [(cd? m) (message-box "cd" (cd-artist m))]
          [else (error 'bad-input)]))
```

This version explicates both aspects of our informal reasoning. The type `media` is an abbreviation for the true union intended by the programmer; naturally, it is unnecessary to introduce type abbreviations like this one; we do so for convenience. Furthermore, the defintion of `print` is not modified at all; Typed Scheme type-checks each branch of the conditional appropriately. In short, only minimal type annotations are required to obtain a typed version of the original code, in which the informal, unchecked comments become statically-checked design elements.

More complex reasoning about the flow of values in Scheme programs is also accomodated in our design:

```
(foldl add-book-to-library empty-library
   (filter book? list-of-media))
```

This code selects all the `book`s from a list of media, and then adds them one by one to an initially empty library, perhaps being prepared for display on a web page. Even though the initial `list-of-media` may contain media that are not `book`s, those are removed by the `filter` function. The resulting list contains only `book`s, and is an appropriate argument to `add-book-to-library`.

This example demonstrates a different mode of reasoning than the first; here, the Scheme programmer uses polymorphism and the argument-dependent invariants of `filter` to ensure correctness.

No changes to this code are required for it to typecheck in Typed Scheme. The type system is able to integrate the two modes of reasoning the programmer uses with polymorphic functions and occurrence typing. In contrast, a more conventional type system would require the use of an intermediate data structure, classified with an option type, to ensure conformance.

3.3 Local Type Inference

Typed Scheme supports first-class polymorphic functions, a mechanism that Scheme programmers often implicitly assume when reasoning about many library functions. For example, `list-ref` has the type

```
(All (α) ((Listof α) Integer -> α))
```

It can be defined in Typed Scheme as follows:

```
(: list-ref (All (α) ((Listof α) Integer -> α)))
(define (list-ref l i)
   (cond [(not (pair? l)) (error "empty list")]
            [(= 0 i) (car l)]
            [else (list-ref (cdr l) (- i 1))]))
```

The example illustrates two important aspects of polymorphism in Typed Scheme. First, the abstraction over types is explicit in the polymorphic type of `list-ref` but implicit in the function definition. Second, typical uses of polymorphic functions, e.g., `car` and `list-ref`, do not require explicitly type instantiation. Instead, the required type instantiations are synthesized from the types of the arguments.

Argument type synthesis uses the local type inference algorithm of Pierce and Turner [30]. It greatly facilitates the use of polymorphic functions and makes conversions from Scheme to Typed Scheme convenient, while dealing with the subtyping present in the rest of the type system in an elegant manner. Furthermore, it ensures that type inference errors are always locally confined, rendering them reasonably comprehensible to programmers.

4 Research

Preliminary evaluations of Typed Scheme suggest that it can cope with a large amount of the existing PLT Scheme code base, but also reveal numerous challenges. This section addresses two of these, and then presents a research plan.

4.1 Variable-arity Functions

Like many other untyped languages, PLT Scheme provides a rich variety of methods for defining functions which have multiple arities. The simplest of these is `case-lambda`, which allows functions with multiple, fixed arities. But Scheme, and therefore Typed Scheme, also allow functions with variable arity, allowing for power of abstraction not found in other typed languages.

4.1.1 Uniform Variable-Arity Functions

Uniform variable-arity functions expect their rest parameter to be a homogeneous list. Consider the following four examples:

```
  (: + (Integer * -> Integer))
  (: - (Integer Integer * -> Integer))
  (: string-append (String * -> String))
  (: list (All (α) (α * -> (Listof α))))
```

The syntax `SomeType *` for the type of rest parameters alludes to the Kleene star for regular expressions. It signals that in addition to the other arguments, the function takes an arbitrary number of arguments of the given base type. The form `SomeType *` is dubbed a *starred pre-type*, because it is not a full-fledged type and may appear only as the last element of a function's domain.

Here is a possible definition of variable-arity `+` in Typed Scheme, using `binary-+`, a hypothetical binary addition operator:

```
(define (+ . xs)
  (if (null? xs)
      0
      (binary-+ (car xs)
                (apply + (cdr xs)))))
```

Typing this definition is straightforward. The type assigned to the rest parameter of starred pre-type `tau *` in the body of the function is `(Listof tau)`, and it maps to an already-existing type in Typed Scheme. In the above example, `xs` is has the starred pre-type `Integer *`, and so `xs` has the type `(Listof Integer)` Thus, no further work is needed to handle uses of such rest parameters.

4.1.2 Non-Uniform Variable-Arity Functions

For more sophisticated uses of rest parameters, more sophisticated types are needed. Consider the function `map`. The PLT Scheme documentation describes `(map proc lst ...)` as follows:

> *Applies `proc` to the elements of the `lst`s from the first elements to the last. The `proc` argument must accept the same number of arguments as the number of supplied `lst`s, and all `lst`s must have the same number of elements. The result is a list containing each result of `proc`.*

Note that there are potentially many list arguments, and the function is applied to as many arguments as there are lists.

In Typed Scheme, `map` has the type

```
(: map
 (All (γ α β ...)
  ((α β ... β -> γ) (Listof α) (Listof β) ... β  -> (Listof γ))))
```

This introduces several new elements into the type system of Typed Scheme. First, the binding of type variables indicates with `...` that the type variable β can be instantiated with any number of types. The further uses of β indicate that the first argument, the function, takes precisely as many arguments as the number of lists provided as additional arguments.

Instantiating the map function with the types Integer Boolean Integer Boolean results in the type

```
((Integer Boolean Integer -> Boolean)
 (Listof Integer) (Listof Boolean) (Listof Integer)
 ->
 (Listof Boolean))
```

With this type, we can use map in all the ways that Scheme programmers expect. For example, all of these applications typecheck correctly in Typed Scheme:

```
(map not (list #t #f #t))
(map = (list 1 20 300) (list 10 20 30))
(map make-book (list "Flatland") (list "A. Square") (list 1884))
```

Note that map is used at 3 different arities, and in the final example, the list elements are both strings and numbers. [2]

Typed Scheme also allows the definition of functions that use variable-arity polymorphism. Here is the definition of fold-left from the R^6RS [34]:

```
(: fold-left
   (All (γ α β ... β)
      ((γ α β ... β -> γ) γ (Listof α) (Listof β) ... β -> γ)))
(define (fold-left f c as . bss)
  (cond [(and (null? as) (andmap null? bss)) c]
        [(or (null? as) (ormap null? bss))
         (error 'fold-left "wrong length lists")]
        [else (apply fold-left
                     (apply f c (car as) (map car bss))
                     (cdr as)
                     (map cdr bss))]))
```

This generalizes the familiar list fold to work over arbitrary numbers of lists. Typed Scheme is able to express and check the constraint that fold-left must be supplied a procedure with the appropriate arity for the number of list arguments. It is also able to verify that the body of fold-left correctly implements these constraints. To accomplish this, Typed Scheme must, for example, understand that the list produced by (map cdr bss) produces a list of lists precisely as long as bss.

Finally, dotted pre-types are not merely useful in function types. The multiple-value return feature of Scheme also integrates nicely with dotted pre-types. For example, the values function, which is the fundamental constructor for multiple-value return, has the type

[2] The examples also demonstrate how typechecking can eliminate the need for some but not all run-time checks. A map that is applied to two lists must still dynamically enforce the equal-length constraint.

```
(: values
   (All (α ...) (α ... α -> (Values α ... α))))
```

and the `call-with-values` function has the type

```
(: call-with-values
   (All (β α ...) ((-> (Values α ... α)) (α ... α -> β) -> β)))
```

4.2 Keyword Arguments

PLT Scheme provides programmers the ability to define and use functions that take keyword arguments. For example, the `open-output-file` procedure takes a filename as a fixed argument, as well as a `#:mode` keyword argument. The following are all valid applications of `open-output-file`.

```
(open-output-file "foo.txt")
(open-output-file "foo.txt" #:mode 'binary)
(open-output-file #:mode 'text "foo.txt")
```

Since keyword arguments are used pervasively in PLT Scheme, Typed Scheme must also support them. The type of `open-output-file` is therefore

```
(: open-output-file
   (String #:mode (U 'binary 'text) -> Output-Port))
```

here the presence of keywords in the type of `open-output-file` indicates that the function accepts keyword arguments. Given this type for `open-output-file`, all of the above examples typecheck successfully.

Additionally, the handling of the `apply` function must take into account the presence of keyword arguments, and the `keyword-apply` function must be handled in Typed Scheme.

4.3 Practical Evaluation

To determine whether Typed Scheme is practical and whether converting PLT Scheme programs is feasible, I conducted a series of small experiments in porting existing Scheme programs of varying complexity to Typed Scheme. I plan to continue this on a larger scale to validate the overall design an implementation of Typed Scheme.

4.3.1 Small Experiments

I have ported several thousand lines of small Scheme programs from a wide variety of sources to Typed Scheme. This included code from *How to Design Programs* [10] as well as *The Little Schemer* [15] and *The Seasoned Schemer* [14], and also numerous assignments from an undergraduate programming language class. These experiments indicate that Typed Scheme can handle pedagogical Scheme code quite easily, with the exception of complex invariants on the structure of S-expressions.

I also ported[3] several larger programs, including a multi-thousand line multi-player game implementation. This effort again demonstrated that Typed Scheme can handle most idiomatic PLT Scheme code. The only changes required are in places where the original program makes potentially-unsafe assumptions.

4.3.2 Larger Experiments

My preliminary experiments suggest that Typed Scheme is potentially effective. But to truly validate its usefullness, more extensive study is required. I will therefore port portions of the existing PLT Scheme infrastructure, including portions of DrScheme [12], to Typed Scheme. This effort will determine whether Typed Scheme scales to large programs, consisting of tens of thousands of lines and hundreds of modules.

Additionally, metrics are needed to determine the effectiveness of Typed Scheme in finding bugs, documenting code, and the effort required to do so. I will therefore develop such numerical measurements and apply them all of the porting efforts conducted.

4.4 Research Plan

My schedule for completing the work outlined in this proposal is as follows:

- September-October 2008: Port portions of DrScheme to Typed Scheme
- November 2008-January 2009: Implement keyword arguments and variable-arity polymorphism
- February-March 2009: Journal paper on Typed Scheme design
- April-May 2009: Paper on experience
- June-July 2009: Writing
- August 2009: Defense

[3]Ivan Gazeau, a student from ENS, provided valuable assistance with the porting effort.

5 Related Work

The integration of typed and untyped languages has a long history, and Typed Scheme borrows from many systems to develop its unique type system. In this section, I survey the most directly related work, both in typechecking Scheme, and in the predecessors of the features of Typed Scheme.

5.1 Types and Untyped Languages

The history of programming languages knows many attempts to add or to use type information in conjunction with untyped languages. Starting with LISP [35], language designers have tried to include type declarations in such languages, often to help compilers, sometimes to assist programmers.

Beginning with Cartwright's Typed Lisp [4], several projects aimed to provide a typed version of Scheme or Lisp. Other examples include Wand's Semantic Prototyping System [44], Haynes' Infer system [19] and the system of Leavens et. al. [22]. None of these systems, however, attempted to accommodate Scheme programming idioms, nor did they provide sound interoperation with untyped code.

Several systems have attempted to add type inference directly to existing Scheme programs. From the late 1980s until recently, people have studied soft typing [5, 1, 45, 20, 13, 26], a form of type inference to assist programmers debug their programs statically. This work has mainly been in the context of Scheme but has also been applied to Erlang [24] and Python [32]. Static analysis has also been applied to finding bugs via "program comprehension" [27]. Finally, the slogan of "gradual typing" has resurrected the LISP-style annotation mechanisms and has had a first impact with its tentative inclusion in Perl6 [37].

The goal of the soft typing research is to provide an optional type checker for programs in untyped languages. One key premise is that programmers shouldn't have to write down type definitions or type declarations. Soft typing should work via type inference only, just like ML. Another premise is that soft type systems should never prevent programmers from running any program. If the type checker encounters such an ill-typed program, it should insert run-time checks that restore typability and ensure that the type system remains sound.

In contrast, Typed Scheme is a explicit and static type system for a language that can interoperate with Scheme. We believe that this reduces the complexity of the overall system, makes it computationally more tractable, and easier for programmers to use and understand. Most importantly, types that must be inferred by an analysis cannot play the role of mandatory and checked documentation. However, Typed Scheme has learned from these earlier efforts. In particular, both Wright [45] and Flanagan [13] provided a form of syntactic `if`-splitting which foreshadowed the occurrence typing of Typed Scheme.

The recent gradual typing research [21, 33, 42] has mostly consisted of theoretical investigations of the properties of small calculi in which typed and untyped code can be freely intermixed. Unfortunately, this strategy prevents programmers from drawing clear boundaries between typed and untyped code. Additionally, it is not yet clear how to scale a gradually-typed system up to a full-fledged programming language without efforts such as our purpose-built type system with occurence typing, non-uniform variable-arity functions, keyword arguments and other features neccessary to accomodate programs from existing untyped languages.

5.2 Type System Features

Many of the type system features we have incorporated into Typed Scheme have been extensively studied in isolation. Polymorphism in type systems dates to Reynolds [31]. Recursive types were studied by Amadio and Cardelli [2], and union types by Pierce [29], among many others. Intensional polymorphism appears in calculi by Harper and Morrisett [18], among others. Some of the design of the type system for occurrence typing was inspired by prior work on effect systems [17]. Typing variables differently in different portions of a program was discussed by Weirich et. al. [6]. However, occurrence typing as presented here has not been previously considered.

Several other attempts have been made to handle non-uniform variable-arity polymorphism, but no typed language supports them in a systematic an principled manner. Dzeng and Haynes [8] come close to our goal of providing a practical type system for variable-arity functions as apart of the Infer system for type-checking Scheme [19]. However, their system does not handle subtyping, and cannot typecheck the defintions of several key examples. Tullsen [40] attempts to bring non-uniform variable-arity functions to Haskell via the Zip Calculus. This work uses a restricted form of dependent types. It again does not seem to be able to handle several of the most important functions without further extension, and has not been tried on an actual Haskell code base.

Keyword arguments are already present in some typed languages [23] and have also been studied in a theoretical framework [16]. However, these systems are complicated by the need to support auto-currying.

6 Conclusion

The recent explosion of scripting languages, and thus of scripts, has led to a need for migration paths to maintainable programs. I have proposed Typed Scheme, a typed sister language of PLT Scheme, as both a strategy and an example. Typed Scheme's interoperability features and type system will allow it to integrate smoothly and painlessly with existing PLT Scheme code, facilitating easy migration.

In the future, migration from untyped scripts to typed programs will not just happen for

Scheme. The specific features I have proposed for Typed Scheme may not carry over to Perl or Javascript, but the fundamental ideas of sound interoperability using contracts, and type systems that accomodate idiomatic practice can.

References

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173, New York, NY, USA, 1994. ACM Press.

[2] Roberto Amadio and Luca Cardelli. Subtyping recursive types. In *ACM Transactions on Programming Languages and Systems*, volume 15, pages 575–631, 1993.

[3] Stig Sæther Bakken, Alexander Aulbach, Egon Schmid, Jim Winstead, Lars Torben Wilson, Rasmus Lerdorf, Andrei Zmievski, and Jouni Ahto, January 2002. `http://www.php.net/manual/`.

[4] R. Cartwright. User-defined data types as an aid to verifying lisp programs. In *Third International Colloquium on Automata, Languages and Programming*, pages 228–256, 1976.

[5] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 278–292, New York, NY, USA, 1991. ACM Press.

[6] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 301–312, New York, NY, USA, 1998. ACM Press.

[7] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced Macrology and the Implementation of Typed Scheme. In *Proceedings of the Eighth Workshop on Scheme and Functional Programming*, 2007.

[8] Hsianlin Dzeng and Christopher T. Haynes. Type Reconstruction for Variable-Arity Procedures. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and Functional Programming*, pages 239–249, New York, NY, USA, 1994. ACM Press.

[9] ECMA International. ECMAScript Edition 4 group wiki, 2007.

[10] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[11] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, 2002.

[12] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In Hugh Glaser, Pieter Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 369–388, Southampton, UK, September 1997. Springer.

[13] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 21(2):370–416, 1999.

[14] Daniel P. Friedman and Matthias Felleisen. *The Seasoned Schemer*. MIT Press, Cambridge, 1996.

[15] Daniel P. Friedman and Matthias Felleisen. *The Little Schemer, Fourth Edition*. MIT Press, Cambridge, 1997.

[16] Jacques Garrigue and Hassan Aït-Kaci. The typed polymorphic label-selective λ-calculus. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 35–47, New York, NY, USA, 1994. ACM.

[17] David Gifford, Pierre Jouvelot, John Lucassen, and Mark Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.

[18] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, 1995.

[19] Christopher T. Haynes. Infer: A Statically-typed Dialect of Scheme. Technical Report 367, Indiana University, 1995.

[20] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *FPCA '95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 192–203, 1995.

[21] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *8th Symposium on Trends in Functional Programming*, April 2007.

[22] Gary T. Leavens, Curtis Clifton, , and Brian Dorn. A Type Notation for Scheme. Technical Report 05-18a, Iowa State University, August 2005.

[23] Xavier Leroy. *The Objective Caml system, Documentation and User's guide*, 1997.

[24] Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 136–149, New York, NY, USA, 1997. ACM Press.

[25] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly, 2001.

[26] Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *POPL '06: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 218–231, New York, NY, USA, 2006. ACM Press.

[27] Robert O'Callahan and Daniel Jackson. Lackwit: a program understanding tool based on type inference. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 338–348, New York, NY, USA, 1997. ACM.

[28] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[29] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[30] Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[31] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Paris, France*, pages 513–523. Elsevier, 1983.

[32] Michael Salib. Starkiller: A static type inferencer and compiler for Python. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 2004.

[33] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop, University of Chicago, Technical Report TR-2006-06*, pages 81–92, September 2006.

[34] Michael Sperberr, Kent Dybvig, Matthew Flatt, and Anton van Straaten (Eds.). Revised^6 report of the algorithmic language Scheme, 26 September 2007. `http://www.r6rs.org`.

[35] Guy Lewis Steele Jr. *Common Lisp—The Language*. Digital Press, 1984.

[36] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Variable-arity polymorphism, 2008. Manuscript in preparation.

[37] Audrey Tang. Perl 6: reconciling the irreconcilable. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–1, New York, NY, USA, 2007. ACM Press. http://pugscode.org.

[38] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: From Scripts to Programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 964–974. Companion (Dynamic Languages Symposium), 2006.

[39] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 395–406, 2008.

[40] Mark Tullsen. The Zip Calculus. In Roland Backhouse and Jose Nuno Oliveira, editors, *Mathematics of Program Construction, 5th International Conference, MPC 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 28–44. Springer-Verlag, July 2000.

[41] G. van Rossum. An introduction to Python for UNIX/C programmers. *Proc. NLUUG - Dutch Unix User Group Conference*, 1993.

[42] Philip Wadler and Robert B. Findler. Well-typed programs can't be blamed. In *Workshop on Scheme and Functional Programming*, September 2007.

[43] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1996.

[44] Mitchell Wand. A semantic prototyping system. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 213–221, New York, NY, USA, 1984. ACM Press.

[45] Andrew K. Wright and Robert Cartwright. A practical soft type system for scheme. *ACM Trans. Program. Lang. Syst.*, 19(1):87–152, 1997.