Higher-Order and Symbolic Computation manuscript No.
(will be inserted by the editor)

The Design and Implementation of Typed Scheme: From Scripts to Programs*

Sam Tobin-Hochstadt · **Matthias Felleisen**

Received: date / Accepted: date

Abstract When scripts in untyped languages grow into large programs, maintaining them becomes difficult. A lack of explicit type annotations in typical scripting languages forces programmers to must (re)discover critical pieces of design information every time they wish to change a program. This analysis step both slows down the maintenance process and may even introduce mistakes due to the violation of undiscovered invariants.

This paper presents Typed Scheme, an explicitly typed extension of PLT Scheme, an untyped scripting language. Its type system is based on the novel notion of *occurrence typing*, which we formalize and mechanically prove sound. The implementation of Typed Scheme additionally borrows elements from a range of approaches, including recursive types, true unions and subtyping, plus polymorphism combined with a modicum of local inference.

The formulation of occurrence typing naturally leads to a simple and expressive version of predicates to describe *refinement types*. A Typed Scheme program can use these refinement types to keep track of arbitrary classes of values via the type system. Further, we show how the Typed Scheme type system, in conjunction with simple recursive types, is able to encode refinements of existing datatypes, thus expressing *both* proposed variations of refinement types.

Keywords Scheme · Type Systems · Refinement types

1 Type Refactoring: From Scripts to Programs

Recently, under the heading of "scripting languages", a variety of new languages have become popular, and even pervasive, in web- and systems-related fields. Due to their popularity, programmers often create scripts that then grow into large applications.

* This is a revised and extended version of a paper presented at the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2008. The authors were supported by the NSF, the USAF, and the Mozilla Foundation.

S. Tobin-Hochstadt
Northeastern University
E-mail: samth@ccs.neu.edu

M. Felleisen
Northeastern University

Most scripting languages are untyped and provide primitives with flexible semantics to make programs concise. Many programmers find these attributes appealing and use scripting languages for these reasons. 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. Both the Perl community (Tang, 2007) and the JavaScript community (ECMA, 2007) are implicitly acknowledging this problem by considering the addition of Common Lisp-style (Steele Jr., 1984) typing constructs to the upcoming releases of their respective languages. Additionally, type system proposals have been made for Ruby (Furr et al, 2009a) and for Dylan (Mehnert, 2009)

In the meantime, industry faces the problem of porting existing application systems from untyped scripting languages to the typed world. In response, we have proposed a theoretical model for this conversion process and have shown that partial conversions can benefit from type-safety properties to the desired extent (Tobin-Hochstadt and Felleisen, 2006). This problem has also sparked significant research interest in the evolution of scripts to programs (Wrigstad et al, 2009). The key assumption behind our work is the existence of an explicitly typed version of the scripting language, with the same semantics as the original language, so that values can freely flow back and forth between typed and untyped modules. In other words, we imagine that programmers can simply add type annotations to a module and thus introduce a certain amount of type-safety into the program.

At first glance, our assumption of such a typed sister language may seem unrealistic. Programmers in untyped languages often loosely mix and match reasoning from various type disciplines when they write scripts. Worse, an inspection of code suggests they also include flow-oriented reasoning, distinguishing types for variables depending on prior operations. In short, untyped scripting languages permit programs that appear difficult to type-check with existing type systems.

To demonstrate the feasibility of our approach, we have designed and implemented Typed Scheme, an explicitly typed version of PLT Scheme. We have chosen PLT Scheme for two reasons. On one hand, PLT Scheme is used as a scripting language by a large number of users. It also comes with a large body of code, with contributions ranging from scripts to libraries to large operating-system like programs. On the other hand, the language comes with macros, a powerful extension mechanism (Flatt, 2002). Macros place a significant constraint on the design and implementation of Typed Scheme, since supporting macros requires type-checking a language with a user-defined set of syntactic forms. We are able to overcome this difficulty by integrating the type checker with the macro expander. Indeed, this approach ends up greatly facilitating the integration of typed and untyped modules. As envisioned (Tobin-Hochstadt and Felleisen, 2006), this integration makes it mostly straightforward to turn portions of a multi-module program into a partially typed yet still executable program.

Developing Typed Scheme requires not just integration with the underlying PLT Scheme system, but also a type system that works well with the idioms used by PLT Scheme programmers when developing scripts. It would be an undue burden if the programmer needed to rewrite idiomatic PLT Scheme code to make it typeable in Typed Scheme. For this purpose, we have developed a novel type system, combining the idea of *occurrence* typing with subtyping, recursive types, polymorphism and a modicum of inference.

The design of Typed Scheme and its type system also allows for simple additions of sophisticated type system features. In particular, the treatment of predicates in Typed Scheme lends itself naturally to treating predicates such as *even?* as defining refinements of existing types, such a integers. This allows for a lightweight form of refinement types, without any need for implication or inclusion checking.

We first present a formal model of the key aspects of occurrence typing and prove it to be type-sound. We then describe how refinement types can be added to this system, and how they can be used effectively in Typed Scheme. Later we describe how to scale this calculus into a full-fledged, typed version of PLT Scheme and how to implement it. Finally, we give an account of our preliminary experience, adding types to thousands of lines of untyped Scheme code. Our experiments seem promising and suggest that converting untyped scripts into well-typed programs is feasible.

2 Overview of Typed Scheme

The goal of the Typed Scheme project is to develop an explicit type system that easily accommodates a conventional Scheme programming style. Ideally, programming in Typed Scheme should feel like programming in PLT Scheme, except for typed function and structure signatures plus type definitions. Few other changes should be required when going from a Scheme program to a Typed Scheme program. Furthermore, the addition of types should require a relatively small effort, compared to the original program. This requires that macros, both those used and defined in the typed program, must be supported as much as possible.

Supporting this style of programming demands a significant rethinking of type systems. Scheme programmers reason about their programs, but not with any conventional type system in mind. They superimpose on their untyped syntax whatever type (or analysis) discipline is convenient. No existing type system could cover all of these varieties of reasoning.

2.1 Occurrence Typing

Consider the following function definition:[1]

```
;; data definition: a Complex is either
;; - a Number or
;; - (cons Number Number)

;; Complex → Number
(define (creal x)
  (cond [(number? x) x]
        [else (car x)]))
```

As the informal data definition states, complex numbers are represented as either a single number, or a pair of numbers (cons).

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 a sum type on which the function operates. Instead there is just an "informal" data definition and contract (Felleisen et al, 2001), 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. Instead, it uses a predicate that distinguishes the two cases: the first **cond** clause, which treats *x* as a number and the second one, which treats it as a pair.

Here is the corresponding Typed Scheme code:

[1] Standards-conforming Scheme implementations provide a complex number datatype directly. This example serves only expository purposes.

```
(define-type-alias Cplx (⋃ Number (cons Number Number)))

(define: (creal [x : Cplx]) : Number
  (cond [(number? x) x]
        [else (car x)]))
```

This version explicates both aspects of our informal reasoning. The type *Cplx* is an abbreviation for the true union intended by the programmer; naturally, it is unnecessary to introduce type abbreviations like this one. Furthermore, the body of *creal* 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.

Our design also accommodates more complex reasoning about the flow of values in Scheme programs.

```
(foldl scene+rectangle empty-scene (filter rectangle? list-of-shapes))
```

This code selects all the *rectangle*s from a list of shapes, and then adds them one by one to an initially-empty scene, perhaps in preparation for rendering to the screen. Even though the initial *list-of-shapes* may contain shapes that are not *rectangle*s, those are removed by the *filter* function. The resulting list contains only *rectangle*s, and is an appropriate argument to *scene+rectangle*. No additional coercions are needed.

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 accommodate both modes of reasoning the programmer uses with polymorphic functions and occurrence typing. In contrast, a more conventional type system such as SML (Milner et al, 1997) would require the use of an intermediate data type, such as an option type, to ensure conformance.

2.2 Refinement Types

Refinement types, introduced originally by Freeman and Pfenning (1991), are types which describe subsets of conventional types. For example, the type of even integers is a refinement of the type of integers. Many different systems have proposed distinct ways of specifying these subsets (Rondon et al, 2008; Wadler and Findler, 2009). In Typed Scheme, we describe a set of values with a simple Scheme predicate.

The fundamental idea is that a boolean-valued function, such as *even?*, can be treated as defining a type, which is a subtype of the input type of *even?*. This type has no constructors, but it is trivial to determine if a value is a member by using the predicate *even?*. For example, this function produces solely even numbers:[2]

```
(: just-even (Number → (Refinement even?)))
(define (just-even n)
  (if (even? n) n (error 'not-even)))
```

This technique harnesses occurrence typing to work with arbitrary predicates, and not just those that correspond to Scheme data types.

[2] In the subsequent formal development, we require a slightly more verbose syntax for refinement types.

2.3 Other Type System Features

In order to support Scheme idioms and programming styles, Typed Scheme supports a number of type system features that have been studied previously, but are rarely found in a single, full-fledged implementation. Specifically, Typed Scheme supports true union types (Pierce, 1991), as seen above. It also provides first-class polymorphic functions, known as impredicative polymorphism, a feature of the Glasgow Haskell Compiler (Vytiniotis et al, 2006). In addition, Typed Scheme allows programmers to explicitly specify recursive types, as well as constructors and accessors that manage the recursive types automatically. Finally, Typed Scheme provides a rich set of base types to match those of PLT Scheme.

2.4 S-expressions

One of the primary Scheme data structures is the S-expression. We have already seen an example of this in the preceding section, where we used pairs of numbers to represent complex numbers. Other uses of S-expressions abound in real Scheme code, including using lists as tuples, records, trees, etc. Typed Scheme handles these features by representing lists explicitly as sequences of *cons* cells. Therefore, we can give an S-expression as precise a type as desired. For example, the expression (*list* 1 2 3) is given the type (*cons* **Number** (*cons* **Number** (*cons* **Number** '()))), which is a subtype of (**Listof Number**).

Lists, of course, are recursive structures, and we exploit Typed Scheme's support for explicit recursive types to make **Listof** a simple type definition over *cons*. Thus, the subtyping relationship for fixed-length lists is simply a consequence of the more general rules for recursive types.

Sometimes, however, Scheme programmers rely on invariants too subtle to be captured in our type system. For example, S-expressions are often used to represent XML data, without first imposing any structure on that data. In these cases, Typed Scheme allows programmers to leave the code dealing with XML in the untyped world, communicating with the typed portions of the program just as other untyped code does.

2.5 Other Important Scheme Features

Scheme programmers also use numerous programming-language features that are not present in typical typed languages. Examples of these include the *apply* function, which applies a function to a heterogeneous list of arguments; the multiple value return mechanism in Scheme; the use of arbitrary non-false values in conditionals; the use of variable-arity and multiple-arity functions; and many others. Some variable-arity functions, such as *map* and *foldl*, require special care in the type system (Strickland et al, 2009). All of these features are widely used in existing PLT Scheme programs, and supported by Typed Scheme.

2.6 Macros

Handling macros well is key for any system that claims to allow typical Scheme practice. This involves handling macros defined in libraries or by the base language as well as macros defined in modules that are converted to Typed Scheme. Further, since macros can be imported from arbitrary libraries, we cannot specify the typing rules for all macros ahead of

time. Therefore, we must expand macros before typechecking. This allows us to handle almost all simple macros, and many existing complex macros without change, i.e., those for which we can infer the types of the generated variables. Further, macros defined in typed code require no changes. Unfortunately, this approach does not scale to the largest and most complex macros, such as those defining a class system (Flatt et al, 2006), which rely on and enforce their own invariants that are not understood by the type system. Handling such macros remains future work.

3 Two Examples of Refinements

To demonstrate the utility of refinement types as provided by Typed Scheme, as well as the other features of the language, we present two extended examples. The first tackles the problem of form validation, demonstrating the use of predicate-based refinements. The second encodes the syntax of the continuation-passing-style λ-calculus in the type system.

3.1 Form Validation

One important problem in form validation is avoiding SQL injection attacks, where a piece of user input is allowed to contain an SQL statement and passed directly to the database. A simple example is the query

(*string-append* "SELECT ∗ FROM users WHERE name = '" *user-name* "';")

If *user-name* is taken directly from user input, then it might contain the string "a' or 't'='t", resulting in an query that returns the entire contents of the `users` table. More damaging queries can be constructed, with data loss a significant possibility (Munroe, 2007).

One common solution for avoiding this problem is sanitizing user input with escape characters. Unfortunately, sanitized input, like unsanitized input, is simply a string. Therefore, we use refinement types to statically verify that only validated input is passed through to the database. This requires two key pieces: the predicate, and the final consumer.

The predicate is a Typed Scheme function that determines if a string is acceptable as input to the database:

```
(: sql-safe? (String → Boolean))
(define (sql-safe? s) omitted)
```

No special type system machinery is required to write and use such a predicate. One more step is needed, however, to turn this predicate into a refinement type:

(**declare-refinement** *sql-safe?*)

This declaration changes the type of *sql-safe?* to be a predicate for (**Refinement** *sql-safe?* **String**).[3]

With this refinement type, we can specify the desired type of our query function:

```
(: query ((Refinement sql-safe? String) → (Listof Result)))
(define (query user-name)
  (run-query
   (string-append "SELECT ∗ FROM users WHERE name = '" user-name "';")))
```

[3] It is similar to the function *sql-safe?* being in the environment Δ in the formalization of refinement types, see section 5.

Since (**Refinement** *sql-safe?* **String**) is a subtype of **String**, *user-name* can be used directly as an argument to *string-append*.

We can also write a *sanitize* function that performs the necessary escaping, and use the *sql-safe?* function and refinement types for static and dynamic verification:

```
(: sanitize (String → (Refinement sql-safe? String)))
(define (sanitize s)
  (define s* (string-map escape-char s))
  (if (sql-safe? s) s (error "escape failed")))
```

The only function that is added to the trusted computing base is the definition of *sql-safe?*, which can be provided by the database vendor. Everything else can be entirely user-written.

Alternative Solutions Another solution to this problem, common in other languages, would have *sanitize* be defined in a different module, with *SQLSafeString* as an opaque exported type. Unfortunately, this requires using an accessor whenever a *SQLSafeString* is used in a context that expects a string (such as *string-append*). The use of refinement types avoids both the dynamic cost of wrapping in a new type, as well as the programmer burden of managing these wrappers and their corresponding accessors.

3.2 Restricted Grammars

Given a recursive data type, it is common to describe subsets of such data that are valid in a particular context. Non-empty lists are a paradigmatic case, and are the original motivating example for Freeman and Pfenning (1991) in their work on refinement types.

In Typed Scheme, the type for a list of **Integer**s would be

```
(define-type IntList (Rec L (⋃ '() (Pair Integer L))))
```

where **Rec** is the constructor for recursive types. Non-empty lists are just a single unfolding of this type, without the initial '() case:

```
(define-type NonEmpty (Pair Integer (Rec L (⋃ '() (Pair Integer L)))))
```

Of course, *NonEmpty* is a subtype of *IntList*.

Using this technique, we can encode other interesting examples of refinement types. To demonstrate its expressiveness, we show how to encode the *partitioned CPS* of Sabry and Felleisen (1993, Definition 8). We begin with encodings of variables, which distinguishes variables ranging over user values (V_u) from variables ranging over continuations (V_k):

```
(define-type V_u Symbol)
(define-struct: V_k ([v : Symbol]))
(define-type V (⋃ V_u V_k))
```

The λ and *app* constructors are both parameterized over their two field types. A λ contains a variable and a body, while an *app* has an operator and an operand:

```
(define-struct: (V A) λ ([x : V] [b : A]))
(define-struct: (A B) app ([rator : A] [rand : B]))

(define-type (λ_U A) (λ V_u A))
(define-type (λ_K A) (λ V_k A))
(define-type (λ_A A) (λ V A))
```

λ_U and λ_K are abbreviations for user-level and transformation-introduced abstractions. λ_A allows any kind of variable.

With these preliminaries in place, we can define λ-terms:

(**define-type** Λ (**Rec** L ($\bigcup$ V (λ_A L) (*app* L L))))

A term is either a variable (V), an abstraction whose body is a term, or an application of two terms.

We now transform the original definition of partitioned CPS terms:

$$\begin{array}{l} P ::= (K\ W) \\ W ::= x \mid \lambda k.K \\ K ::= k \mid (W\ K) \mid \lambda k.K \end{array}$$

We can write these definitions directly as Typed Scheme types:[4]

(**define-type** P (*app* K W))
(**define-type** W ($\bigcup$ V_u (λ_K K)))
(**define-type** K ($\bigcup$ V_k (*app* W K) (λ_K K))

All of the types are of course subtypes of Λ. Thus, compiler writers can write typeful functions that manipulate CPS terms, while also using more general functions that accept arbitrary terms (such as evaluators) on them.

4 A Formal Model of Typed Scheme

Following precedent, we have distilled the novelty of our type system into a typed lambda calculus, λ_{TS}. While Typed Scheme incorporates many aspects of modern type systems, the calculus serves only as a model of occurrence typing, the primary novel aspect of the type system, in conjunction with true union types and subtyping. The latter directly interact with the former; other features of the type system are mostly orthogonal to occurrence typing. This section first presents the syntax and dynamic semantics of the calculus, followed by the typing rules and a (mechanically verified) soundness result.

4.1 Syntax and Operational Semantics

Figure 1 specifies the syntax of λ_{TS} programs. An expression is either a value, a variable, an application, or a conditional. The set of values consists of abstractions, numbers, booleans, and constants. Binding occurrences of variables are explicitly annotated with types. Types are either $\top$, function types, base types, or unions of some finite collection of types. We refer to the decorations on function types as *latent predicates* and explain them, along with *visible predicates*, below in conjunction with the typing rules. For brevity, we abbreviate ($\bigcup$ **true false**) as **Boolean** and ($\bigcup$) as $\bot$.

The operational semantics is standard: see figure 7. Following Scheme and Lisp tradition, any non-**false** value is treated as **true**.

[4] Unfortunately, Typed Scheme currently requires the equivalent, but more verbose definition of P and K, in which the other definitions are inlined:

(**define-type** P (*app* K ($\bigcup$ V_u (λ_K K))))
(**define-type** K (**Rec** K ($\bigcup$ V_k (*app* ($\bigcup$ V_u (λ_K K)) K) (λ_U (*app* K ($\bigcup$ V_u (λ_K K)))))))

We are investigating how to admit the shorter syntax directly.

$d, e, \ldots$	$::= x \mid (e_1\ e_2) \mid (\textbf{if}\ e_1\ e_2\ e_3) \mid v$	Expressions
v	$::= c \mid b \mid n \mid lx : \tau.e$	Values
c	$::= add1 \mid number? \mid boolean? \mid procedure? \mid not$	Primitive Operations
E	$::= [] \mid (E\ e) \mid (v\ E) \mid (\textbf{if}\ E\ e_2\ e_3)$	Evaluation Contexts
ϕ	$::= \tau \mid \bullet$	Latent Predicates
ψ	$::= \tau_x \mid x \mid \textbf{true} \mid \textbf{false} \mid \bullet$	Visible Predicate
σ, τ	$::= \top \mid \textbf{Number} \mid \textbf{true} \mid \textbf{false} \mid (\sigma \xrightarrow{\phi} \tau) \mid (\bigcup\ \tau \ldots)$	Types

Fig. 1 Syntax

$$\text{T-VAR}\quad \Gamma \vdash x : \Gamma(x); x \qquad \text{T-NUM}\quad \Gamma \vdash n : \textbf{Number}; \textbf{true} \qquad \text{T-CONST}\quad \Gamma \vdash c : \delta_\tau(c); \textbf{true}$$

$$\text{T-TRUE}\quad \Gamma \vdash \textbf{true} : \textbf{Boolean}; \textbf{true} \qquad \text{T-FALSE}\quad \Gamma \vdash \textbf{false} : \textbf{Boolean}; \textbf{false}$$

$$\text{T-ABSPRED}\quad \frac{\Gamma, x : \sigma \vdash e : \tau; \sigma'_x}{\Gamma \vdash lx : \sigma.e : (\sigma \xrightarrow{\sigma'} \tau); \textbf{true}} \qquad \text{T-ABS}\quad \frac{\Gamma, x : \sigma \vdash e : \tau; \psi}{\Gamma \vdash lx : \sigma.e : (\sigma \xrightarrow{\bullet} \tau); \textbf{true}}$$

$$\text{T-APP}\quad \frac{\Gamma \vdash e_1 : \tau'; \psi \quad \Gamma \vdash e_2 : \tau; \psi' \quad \vdash \tau <: \tau_0 \quad \vdash \tau' <: (\tau_0 \xrightarrow{\phi} \tau_1)}{\Gamma \vdash (e_1\ e_2) : \tau_1; \bullet}$$

$$\text{T-APPPRED}\quad \frac{\Gamma \vdash e_1 : \tau'; \psi \quad \Gamma \vdash e_2 : \tau; x \quad \vdash \tau <: \tau_0 \quad \vdash \tau' <: (\tau_0 \xrightarrow{\sigma} \tau_1)}{\Gamma \vdash (e_1\ e_2) : \tau_1; \sigma_x}$$

$$\text{T-IF}\quad \frac{\Gamma \vdash e_1 : \tau_1; \psi_1 \quad \Gamma + \psi_1 \vdash e_2 : \tau_2; \psi_2 \quad \Gamma - \psi_1 \vdash e_3 : \tau_3; \psi_3 \quad \vdash \tau_2 <: \tau \quad \vdash \tau_3 <: \tau \quad \psi = \mathsf{combpred}(\psi_1, \psi_2, \psi_3)}{\Gamma \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \tau; \psi}$$

Fig. 2 Primary Typing Rules

$$\mathsf{combpred}(\psi', \psi, \psi) = \psi \qquad \mathsf{combpred}(\tau_x, \textbf{true}, \sigma_x) = (\bigcup\ \tau\ \sigma)_x$$
$$\mathsf{combpred}(\textbf{true}, \psi_1, \psi_2) = \psi_1 \qquad \mathsf{combpred}(\textbf{false}, \psi_1, \psi_2) = \psi_2$$
$$\mathsf{combpred}(\psi, \textbf{true}, \textbf{false}) = \psi \qquad \mathsf{combpred}(\psi_1, \psi_2, \psi_3) = \bullet$$

$$\delta_\tau(add1) = (\textbf{Number} \xrightarrow{\bullet} \textbf{Number}) \qquad \delta_\tau(not) = (\top \xrightarrow{\bullet} \textbf{Boolean})$$
$$\delta_\tau(procedure?) = (\top \xrightarrow{(\bot \xrightarrow{\bullet} \top)} \textbf{Boolean}) \qquad \delta_\tau(number?) = (\top \xrightarrow{\textbf{Number}} \textbf{Boolean})$$
$$\delta_\tau(boolean?) = (\top \xrightarrow{\textbf{Boolean}} \textbf{Boolean})$$

Fig. 3 Auxiliary Operations

$$\Gamma + \tau_x = \Gamma[x : \mathsf{restrict}(\Gamma(x), \tau)\ ] \qquad \Gamma + x = \Gamma[x : \mathsf{remove}(\Gamma(x), \textbf{false})\ ] \qquad \Gamma + \bullet = \Gamma$$
$$\Gamma - \tau_x = \Gamma[x : \mathsf{remove}(\Gamma(x), \tau)\ ] \qquad \Gamma - x = \Gamma[x : \textbf{false}\ ] \qquad \Gamma - \bullet = \Gamma$$

$$\mathsf{restrict}(\sigma, \tau) = \sigma \text{ when} \vdash \sigma <: \tau \qquad \mathsf{restrict}(\sigma, (\bigcup\ \tau \ldots)) = (\bigcup\ \mathsf{restrict}(\sigma, \tau) \ldots)$$
$$\mathsf{restrict}(\sigma, \tau) = \tau \text{ otherwise}$$
$$\mathsf{remove}(\sigma, \tau) = \bot \text{ when} \vdash \sigma <: \tau \qquad \mathsf{remove}(\sigma, (\bigcup\ \tau \ldots)) = (\bigcup\ \mathsf{remove}(\sigma, \tau) \ldots)$$
$$\mathsf{remove}(\sigma, \tau) = \sigma \text{ otherwise}$$

Fig. 4 Environment Operations

4.2 Preliminaries

The key feature of λ_{TS} is its support for assigning distinct types to distinct occurrences of a variable based on control flow criteria. For example, to type the expression

(ł(*x* **:** (⋃ **Number Boolean**))
  (**if** (*number? x*) (= *x* 1) (*not x*)))

the type system must use **Number** for *x* in the *then* branch of the conditional and **Boolean** in the *else* branch. If it can distinguish these occurrences and project out the proper component of the declared type (⋃ **Number Boolean**), the computed type of the function is

$$((\bigcup \textbf{Number Boolean}) \rightarrow \textbf{Boolean}).$$

The type system for λ_{TS} shows how to distinguish these occurrences; its presentation consists of two parts. The first are those rules that the programmer must know and that are used in the implementation of Typed Scheme. The second set of rules are needed only to establish type soundness; these rules are unnecessary outside of the proof of the main theorem.

Visible Predicates Judgments of λ_{TS} involve both types and visible predicates (see the production for ψ in figure 1). The former are standard. The latter are used to accumulate information about expressions that affect the flow of control and thus demand a split for different branches of a conditional. Of course, a syntactic match would help little, because programmers of scripts tend to write their own predicates and compose logical expressions with combinators. Also, programmer-defined datatypes extend the set of predicates.

Latent Predicates In order to accommodate programmer-defined functions that are used as predicates, the type system of λ_{TS} uses latent predicates (see ϕ in figure 1) to annotate function types. Syntactically speaking, a latent predicate is a single type ϕ atop the arrow-type constructor that identifies the function as a predicate for ϕ. This latent predicate-annotation allows a uniform treatment of built-in and user-defined predicates. For example,

$$\textit{number?} : (\top \overset{\textbf{Number}}{\rightarrow} \textbf{Boolean})$$

says that *number*? is a discriminator for numbers. An eta-expansion preserves this property:

$$(\lambda\ (x{:}\top)\ (\textit{number?}\ x)) : (\top \overset{\textbf{Number}}{\rightarrow} \textbf{Boolean}).$$

Thus far, *higher-order* latent predicates are useful in just one case: *procedure?*. For uniformity, the syntax accommodates the general case. We intend to study an integration of latent predicates with higher-order contracts (Findler and Felleisen, 2002) and expect to find additional uses.

The λ_{TS} calculus also accommodates logical combinations of predicates. Thus, if a program contains a test expression such as:

(**if** (*number? x*) #t (*boolean? x*))

then Typed Scheme computes the appropriate visible predicate for this union, which is $(\bigcup \textbf{Number Boolean})_x$. This information is propagated so that a programmer-defined function receives a corresponding latent predicate. That is, the *bool-or-number* function:

(ł(*x* **: Any**) (**if** (*number? x*) #t (*boolean? x*)))

acts like a predicate of type (**Any** $\overset{(\bigcup \textbf{Number Boolean})}{\rightarrow}$ **Boolean**) and is used to split types in different branches of a conditional.

4.3 Typing Rules

Equipped with types and predicates, we turn to the typing rules. They derive judgements of the form

$$\Gamma \vdash e : \tau; \psi.$$

It states that in type environment Γ, expression e has type τ and visible predicate ψ. The latter is used to change the type environment in conjunction with **if** expressions.[5] The type system proper comprises the ten rules in figure 2.

The rule T-IF is the key part of the system, and shows how *visible* predicates are treated. To accommodate Scheme style, we allow expressions with *any* type as tests. Most importantly, though, the rule uses the visible predicate of the test to modify the type environment for the verification of the types in the two conditional branches. When a variable is used as the test, we know that it cannot be **false** in the *then* branch, and must be in the *else* branch.

While many of the type-checking rules appear familiar, the presence of visible predicate distinguishes them from ordinary rules:

- T-VAR assigns a variable its type from the type environment and names the variable itself as the visible predicate.
- Boolean constants have Boolean type and a visible predicate that depends on their truth value. Since numbers and primitive functions are always treated as true values, they have visible predicate **true**.
- When we abstract over a predicate, the abstraction should reflect the test being performed. This is accomplished with the T-ABSPRED rule, which gives an abstraction a latent predicate if the body of the abstraction has a visible predicate referring to the abstracted variable, as in the *bool-or-number* example.
  Otherwise, abstractions have their usual type; the visible predicate of their body is ignored. The visible predicate of an abstraction is **true**, since abstractions are treated that way by **if**.
- Checking plain applications proceeds as normal. The antecedents include latent predicates and visible predicates but those are ignored in the consequent.
- The T-APPPRED rule shows how the type system exploits latent predicates. The application of a function with latent predicate to a variable turns the latent predicate into a visible predicate on the variable (σ_x). The proper interpretation of this visible predicate is that the application produces **true** if and only if x has a value of type σ.

Figure 3 defines a number of auxiliary typing operations. The mapping from constants to types is standard. The ternary $\mathsf{combpred}(-,-,-)$ metafunction combines the effects of the test, then and else branches of an **if** expression. The most interesting case is the second one, which handles expressions such as this:

(**if** (*number? x*) #t (*boolean? x*))

the equivalent of an **or** expression. The combined effect is $(\bigcup \mathbf{Number}\,\mathbf{Boolean})_x$, as expected.

The environment operations, specified in figure 4, combine a visible predicate with a type environment, updating the type of the appropriate variable. Thus, $\mathsf{restrict}(\sigma,\tau)$ is σ restricted to be a subtype of τ, and $\mathsf{remove}(\sigma,\tau)$ is σ without the portions that are subtypes of τ. The only non-trivial cases are for union types.

[5] Other control flow constructs in Scheme are almost always macros that expand into **if**, and that the typechecker can properly check.

$$\text{S-REFL}\quad \vdash \tau <: \tau$$

$$\text{S-FUN}\quad \frac{\vdash \sigma_1 <: \tau_1 \qquad \vdash \tau_2 <: \sigma_2 \qquad \phi = \phi' \text{ or } \phi' = \bullet}{\vdash (\tau_1 \xrightarrow{\phi} \tau_2) <: (\sigma_1 \xrightarrow{\phi'} \sigma_2)}$$

$$\text{S-UNIONSUPER}\quad \frac{\vdash \tau <: \sigma_i \qquad 1 \le i \le n}{\vdash \tau <: (\bigcup\ \sigma_1 \cdots \sigma_n)}$$

$$\text{S-UNIONSUB}\quad \frac{\vdash \tau_i <: \sigma \text{ for all } 1 \le i \le n}{\vdash (\bigcup\ \tau_1 \cdots \tau_n) <: \sigma}$$

Fig. 5 Subtyping Relation

$$\text{T-APPPREDTRUE}\quad \frac{\Gamma \vdash e_1 : \tau'; \psi \qquad \Gamma \vdash e_2 : \tau; \psi' \qquad \vdash \tau <: \tau_0 \qquad \vdash \tau <: \sigma \qquad \vdash \tau' <: (\tau_0 \xrightarrow{\sigma} \tau_1)}{\Gamma \vdash (e_1\ e_2) : \tau_1; \mathbf{true}}$$

$$\text{T-APPPREDFALSE}\quad \frac{\Gamma \vdash e_1 : \tau'; \psi \qquad \Gamma \vdash v : \tau; \psi' \qquad \vdash \tau <: \tau_0 \qquad \vdash \tau \not<: \sigma \qquad v \text{ closed} \qquad \vdash \tau' <: (\tau_0 \xrightarrow{\sigma} \tau_1)}{\Gamma \vdash (e_1\ v) : \tau_1; \mathbf{false}}$$

$$\text{T-IFTRUE}\quad \frac{\Gamma \vdash e_1 : \tau_1; \mathbf{true} \qquad \Gamma \vdash e_2 : \tau_2; \psi_2 \qquad \vdash \tau_2 <: \tau}{\Gamma \vdash (\mathbf{if}\ e_1\ e_2\ e_3) : \tau; \bullet}$$

$$\text{T-IFFALSE}\quad \frac{\Gamma \vdash e_1 : \tau_1; \mathbf{false} \qquad \Gamma \vdash e_3 : \tau_3; \psi_3 \qquad \vdash \tau_3 <: \tau}{\Gamma \vdash (\mathbf{if}\ e_1\ e_2\ e_3) : \tau; \bullet}$$

$$\text{SE-REFL}\quad \vdash \psi <:_? \psi \qquad \text{SE-NONE}\quad \vdash \psi <:_? \bullet$$

$$\text{SE-TRUE}\quad \frac{\psi \neq \mathbf{false}}{\vdash \mathbf{true} <:_? \psi} \qquad \text{SE-FALSE}\quad \frac{\psi \neq \mathbf{true}}{\vdash \mathbf{false} <:_? \psi}$$

Fig. 6 Auxiliary Typing Rules

For the motivating example from the beginning of this section,

(ɬ(x **:** ($\bigcup$ **Number Boolean**)) (**if** (*number? x*) (= x 1) (*not x*)))

we can now see that the test of the **if** expression has type **Boolean** and visible predicate $\mathbf{Number}_x$. As a consequence, the *then* branch is type-checked in an environment where x has type **Number**; in the *else* branch, x is assigned **Boolean**.

Subtyping The definition of subtyping is given in figure 5. The rules are for the most part standard, with the rules for union types adapted from Pierce's (Pierce, 1991). One important consequence of these rules is that $\bot$ is below all other types. This type is useful for typing functions that do not return, as well as for defining a supertype of all function types.

We do not include a transitivity rule for the subtyping relation, but instead prove that the subtyping relation as given is transitive. This choice simplifies the proof in a few key places.

The rules for subtyping allow function types with latent predicates to be used in a context that expects a function that is not a predicate. This is especially important for *procedure?*, which handles functions regardless of latent predicate.

$$\text{E-DELTA}\ \frac{\delta(c,v) = v'}{(c\ v) \hookrightarrow v'} \qquad \text{E-BETA}\ (lx : \tau.e\ v) \hookrightarrow e[x/v]$$

$$\text{E-IFFALSE}\ \frac{v = \textbf{false}}{(\textbf{if}\ v\ e_2\ e_3) \hookrightarrow e_3} \qquad \text{E-IFTRUE}\ \frac{v \neq \textbf{false}}{(\textbf{if}\ v\ e_2\ e_3) \hookrightarrow e_2}$$

$$\frac{L \hookrightarrow R}{E[L] \to E[R]}$$

$$\delta(add1, n) = n + 1$$
$$\delta(not, \textbf{false}) = \textbf{true} \qquad \delta(not, v) = \textbf{false}\ \ v \neq \textbf{false}$$
$$\delta(number?, n) = \textbf{true} \qquad \delta(number?, v) = \textbf{false}$$
$$\delta(boolean?, b) = \textbf{true} \qquad \delta(boolean?, v) = \textbf{false}$$
$$\delta(procedure?, lx : \tau.e) = \textbf{true} \qquad \delta(procedure?, c) = \textbf{true}$$
$$\delta(procedure?, v) = \textbf{false}\ \text{otherwise}$$

Fig. 7 Operational Semantics

4.4 Proof-Technical Typing Rules

The typing rules in figure 2 do not suffice for the soundness proof. To see why, consider the function from above, applied to the argument #f. By the E-BETA rule, this reduces to

(**if** (*number?* #f) (= #f 1) (*not* #f))

Unfortunately, this program is not well-typed according the primary typing rules, since = requires numeric arguments. Of course, this program reduces in just a few steps to #t, which is an appropriate value for the original type. To prove type soundness in the style of Wright and Felleisen (Wright and Felleisen, 1994), however, every intermediate term must be typeable. So our types system must know to ignore the *then* branch of our reduced term.

To this end, we extend the type system with the rules in figure 6. This extension assigns the desired type to our reduced expression, because (*number?* #f) has visible predicate **false**. Put differently, we can disregard the *then* branch, using rule T-IFFALSE.[6]

In order to properly state the subject reduction lemma, we need to relate the visible predicates of terms in a reduction sequence. To this end, we define a sub-predicate relation, written $\vdash \psi <:_? \psi'$. The relation is defined in figure 6; it is not used in the subtyping or typing rules, being needed only for the soundness proof.

We can now prove the traditional lemmas. We work only with closed terms, since it simplifies the possible predicates of the expression.

Lemma 1 (Preservation) *If* $\vdash e : \tau; \psi$ *(with* e *closed) and* $e \to e'$*, then* $\vdash e' : \tau'; \psi'$ *where* $\vdash \tau' <: \tau$ *and* $\vdash \psi' <:_? \psi$*.*

Proof Sketch This is a corollary of two other lemmas: that plugging a well typed term into the hole of an evaluation preserves the type of the resulting term, and that the $e_1 \hookrightarrow e_2$

[6] The rules in figure 6 are similar to rules used for the same purpose in systems with a `typecase` construct, such as Crary et al (1998).

preserves type when e_1 is closed. These two lemmas are both proved by induction on the relevant typing derivations. □

Lemma 2 (Progress) *If* $\vdash e : \tau; \psi$ *(with* e *closed) then either* e *is a value or* $e \to e'$ *for some* e'.

Proof Sketch By induction on the derivation of $\Gamma \vdash e : \tau; \psi$. □

From these, soundness for the extended type system follows. Programs with untypable subexpressions, however, are not useful in real programs. We only needed to consider them, as well as our additional rules, for our proof of soundness. Fortunately, we can also show that the additional, proof-theoretic, rules are needed only for the type soundness proof, not the result. Therefore, we obtain the desired type soundness result.

Theorem 1 (Soundness) *If* $\Gamma \vdash e : \tau; \psi$*, with* e *closed, using only the rules in figure 2, and* τ *is a base type, one of the following holds*

1. e *reduces forever, or*
2. $e \to^* v$ *where* $\vdash v : \sigma; \psi'$ *and* $\vdash \sigma <: \tau$ *and* $\vdash \psi' <:_? \psi$.

Proof Sketch First, this is a corollary of soundness if the requirement is only that v typechecks in the extended system, since it types strictly more terms. Second, the extended system agrees with the non-extended system on all values of ground type (numbers and booleans). Thus, v has the appropriate type even in the original system. □

4.5 Mechanized Support

We employed two mechanical systems for the exploration of the model and the proof of the soundness theorem: Isabelle/HOL (Nipkow et al, 2002) and PLT Redex (Matthews et al, 2004). Indeed, we freely moved back and forth between the two, and without doing so, we would not have been able to formalize the type system and verify its soundness in an adequate and timely manner.

For the proof of type soundness, we used Isabelle/HOL together with the nominal-isabelle package (Urban, 2008). Expressing a type system in Isabelle/HOL is almost as easy as writing down the typing rules of figures 2 and 6 (our formalization runs to 5000 lines). To represent the reduction semantics (from figure 7) we turn evaluation contexts into functions from expressions to expressions, which makes it relatively straightforward to state and prove lemmas about the connection between the type system and the semantics. Unfortunately, this design choice prevents us from evaluating sample programs in Isabelle/HOL, which is especially important when a proof attempt fails.

Since we experienced such failures, we also used the PLT Redex system (Matthews et al, 2004) to explore the semantics and the type system of Typed Scheme. PLT Redex programmers can write down a reduction semantics as easily as Isabelle/HOL programmers can write down typing rules. That is, each line in figures 1 and 7 corresponds to one line in a Redex model. Our entire Redex model, with examples, is less than 500 lines. Redex comes with visualization tools for exploring the reduction of individual programs in the object language. In support of subject reduction proofs, language designers can request the execution of a predicate for each "node" in the reduction sequences (or graphs). Nodes and transitions that violate a subject reduction property are painted in distinct colors, facilitating example-based exploration of type soundness proofs.

Every time we were stuck in our Isabelle/HOL proof, we would turn to Redex to develop more intuition about the type system and semantics. We would then change the type system of the Redex model until the violations of subject reduction disappeared. At that point, we would translate the changes in the Redex model into changes in our Isabelle/HOL model and restart our proof attempt. Switching back and forth in this manner helped us improve the primary typing rules and determine the shape of the auxiliary typing rules in figure 6. Once we had those, pushing the proof through Isabelle/HOL was a labor-intensive mechanization of the standard proof technique for type soundness.

5 Formalizing Refinements

It is straightforward to add refinement types to the λ_{TS} calculus. We extend the grammar with the new type constructor $(\mathbf{R}\ c\ \tau)$, which is the refinement defined by the built-in function c, which has argument type τ.[7] We restrict refinements to built-in functions so that refinement types can be given to closed expressions and values such as 0. We then add two new constants, *even?*, with type

$$(\mathbf{Number}\ {}^{(\mathbf{R}\ \mathit{even?}\ \mathbf{Number})}\!\!\rightarrow\ \mathbf{Boolean})$$

and *odd?*, with type

$$(\mathbf{Number}\ {}^{(\mathbf{R}\ \mathit{odd?}\ \mathbf{Number})}\!\!\rightarrow\ \mathbf{Boolean})$$

and the obvious semantics.

The subtyping rules for refinements require an additional environment Δ, which specifies which built-ins may be used as refinements. Extending the existing subtyping rules with this environment is straightforward, giving a new judgement of the form $\Delta \vdash_r \tau_1 <: \tau_2$, with the subscript r distinguishing this judgement from the earlier subtyping judgement. As an example, the extended version of the S-REFL rule is

$$\Delta \vdash_r \tau <: \tau$$

The new rule for refinement types is

$$\frac{c \in \Delta \qquad \delta_\tau(c) = (\tau_1 \xrightarrow{\phi} \tau_2) \qquad \Delta \vdash_r \tau_1 <: \tau}{\Delta \vdash_r (\mathbf{R}\ c\ \tau_1) <: \tau}$$

This rule states that a refinement of type τ_1 is a subtype of any type of which τ_1 is a subtype. As expected, this means that $\Delta \vdash_r (\mathbf{R}\ c\ \tau) <: \tau$.

The addition of this environment to the subtyping judgement requires a similar addition to the typing judgement, which now has the form $\Delta, \Gamma \vdash_r e : \tau; \psi$.

This subtyping rule, along with the constants *even?* and *odd?*, are sufficient to write useful examples. For example, this function consumes an even-consuming function and a number, and uses the function if and only if the number is even.

```
(ł([f : ((Refinement even? Number) → Number)] [n : Number])
  (if (even? n) (f n) n))
```

No additional type rules are necessary for this extension. Additionally, any expression of type $(\mathbf{R}\ c\ \tau)$ can be used as if it has type τ, meaning that standard arithmetic operations still work on even and odd numbers.

[7] τ is inferred from the type of c in the implemented system, as demonstrated in section 3.

$$
\begin{array}{lcl}
\mathsf{erase}_\tau((\mathbf{R}\ c\ \tau)) & = & \mathsf{erase}_\tau(\tau) \\
\mathsf{erase}_\tau((\tau \xrightarrow{\tau'} \sigma)) & = & (\mathsf{erase}_\tau(\tau) \xrightarrow{\mathsf{erase}_\tau(\tau')} \mathsf{erase}_\tau(\sigma)) \\
\mathsf{erase}_\tau((\tau \xrightarrow{\bullet} \sigma)) & = & (\mathsf{erase}_\tau(\tau) \xrightarrow{\bullet} \mathsf{erase}_\tau(\sigma)) \\
\mathsf{erase}_\tau(\mathbf{Number}) & = & \mathbf{Number} \\
\mathsf{erase}_\tau(\mathbf{true}) & = & \mathbf{true} \\
\mathsf{erase}_\tau(\mathbf{false}) & = & \mathbf{false} \\
\mathsf{erase}_\tau(\top) & = & \top \\
\mathsf{erase}_\tau((\bigcup\ \tau \ldots)) & = & (\bigcup\ \mathsf{erase}_\tau(\tau) \ldots) \\
\\
\mathsf{erase}_\lambda(lx : \tau.e) & = & lx : \mathsf{erase}_\tau(\tau).\mathsf{erase}_\lambda(e) \\
\mathsf{erase}_\lambda((e_1\ e_2)) & = & (\mathsf{erase}_\lambda(e_1)\ \mathsf{erase}_\lambda(e_2)) \\
\mathsf{erase}_\lambda((\mathbf{if}\ e_1\ e_2\ e_3)) & = & (\mathbf{if}\ \mathsf{erase}_\lambda(e_1)\ \mathsf{erase}_\lambda(e_2)\ \mathsf{erase}_\lambda(e_3)) \\
\mathsf{erase}_\lambda(n) & = & n \\
\mathsf{erase}_\lambda(c) & = & c \\
\mathsf{erase}_\lambda(b) & = & b \\
\mathsf{erase}_\lambda(x) & = & x \\
\\
\mathsf{erase}_\psi(\tau_x) & = & \mathsf{erase}_\tau(\tau)_x \\
\mathsf{erase}_\psi(x) & = & x \\
\mathsf{erase}_\psi(\bullet) & = & \bullet \\
\mathsf{erase}_\psi(\mathbf{true}) & = & \mathbf{true} \\
\mathsf{erase}_\psi(\mathbf{false}) & = & \mathbf{false} \\
\\
\mathsf{erase}_\Gamma(x : \tau, \ldots) & = & x : \mathsf{erase}_\tau(\tau), \ldots \\
\\
\mathsf{erase}_\vdash(\Gamma \vdash e : \tau; \psi) & = & \mathsf{erase}_\Gamma(\Gamma) \vdash \mathsf{erase}_\lambda(e) : \mathsf{erase}_\tau(\tau); \mathsf{erase}_\psi(\psi)
\end{array}
$$

Fig. 8 Erasure Metafunctions

5.1 Soundness

Proving soundness for the extended system with refinements raises the interesting question of what additional errors are prevented by the refinement type extension. The answer is none; no additional behavior is ruled out. This is unsurprising, of course, since the soundness theorem from section 4.4 does not allow the possibility of any errors. But even if errors were added to the operational semantics, such as division by zero, none of these errors would be prevented by the refinement type system. Instead, refinement types allow the specification and enforcement of types that do not have any necessary correspondence to the operational semantics of the language.

We therefore adopt a different proof strategy. Specifically, we erase the refinement types and are left with a typeable term, which reduces appropriately. Given a type in the extended language, we can compute a type without refinement types, simply by erasing all occurrences of $(\mathbf{R}\ c\ \tau)$ to τ. The definition of this function, erase_τ is given in figure 8, along with its extension to terms (erase_λ), predicates (erase_ψ), environments (erase_Γ) and judgments ($\mathsf{erase}_\vdash$). We also assume the obvious modifications to δ_τ.

With these definitions in hand, we can conclude the necessary lemmas for proving soundness.

Lemma 3 (Typing Erased Terms) *If* $\Delta, \Gamma \vdash_r e : \tau; \psi$, *then* $\mathsf{erase}_\vdash(\Gamma \vdash e : \tau; \psi)$.

Proof By induction on the derivation of $\Delta, \Gamma \vdash_r e : \tau; \psi$. □

Lemma 4 (Reducing Erased Terms) *If* $e_1 \to e_2$, *then* $\mathsf{erase}_\lambda(e_1) \to \mathsf{erase}_\lambda(e_2)$.

Proof By induction on the derivation of $e_1 \to e_2$. □

We can combine these lemmas with our earlier preservation and progress lemmas to conclude soundness.

Theorem 2 (Soundness with Refinement Types) *If $\Delta, \Gamma \vdash_r e : \tau; \psi$, with e closed, using only the rules in figure 2, and τ is a base type or a refinement of a base type, one of the following holds*

1. *e reduces forever, or*
2. *$e \rightarrow^* v$ where $\mathsf{erase}_{\vdash}(\vdash v : \sigma; \psi')$ and $\vdash \mathsf{erase}_{\tau}(\sigma) <: \mathsf{erase}_{\tau}(\tau)$ and $\vdash \mathsf{erase}_{\psi}(\psi') <:_? \mathsf{erase}_{\psi}(\psi)$.*

6 From λ_{TS} To Typed Scheme

It is easy to design a type system, and it is reasonably straightforward to validate some theoretical property. However, the true proof of a type system is a pragmatic evaluation. To this end, it is imperative to integrate the novel ideas with an existing programming language. Otherwise it is difficult to demonstrate that the type system accommodates the kind of programming style that people find natural and that it serves its intended purpose.

To evaluate occurrence typing rigorously, we have implemented Typed Scheme. Naturally, occurrence typing with refinements, in the spirit of λ_{TS} makes up only the core of this language; we have supplemented it with a number of important ingredients, both at the level of types and at the level of large-scale programming.

6.1 Type System Extensions

As argued in the introduction, Scheme programmers borrow a number of ideas from type systems to reason about their programs. Chief among them is parametric polymorphism. Typed Scheme therefore allows programmers to define and use polymorphic functions. For example, the *map* function is defined as follows:

```
(define: (a b) (map [f : (a → b)] [l : (Listof a)]) : (Listof b)
  (if (null? l) l
      (cons (f (car l)) (map f (cdr l)))))
```

The definition explicitly quantifies over type variables *a* and *b* and then uses these variables in the type signature. The body of the definition, however, is identical to the one for untyped *map*; in particular, no type application is required for the recursive call to *map*. Instead, the type system infers appropriate instantiations for *a* and *b* for the recursive call.

In addition to parametric polymorphism, Scheme programmers also exploit recursive subtypes of S-expressions to encode a wide range of information as data. To support arbitrary regular types over S-expressions as well as conventional structures, Typed Scheme provides explicit recursive types, though the programmer need not manually fold and unfold instances of these types.

Consider the type of binary trees over *cons* cells:

```
(define-type-alias STree (μ t (⋃ Number (cons t t))))
```

A function for summing the leaves of such a tree is straightforward:

```
(define: (sum-tree [s : STree]) : Number
  (cond [(number? s) s]
        [else (+ (sum-tree (car s)) (sum-tree (cdr s)))]))
```

In this function, occurrence typing allows us to discriminate between the different branches of the union; the (un)folding of the recursive (tree) type happens automatically.

Finally, Typed Scheme supports a rich set of base types, including vectors, boxes, parameters, ports, and many others. It also provides type aliasing, which greatly facilitates type readability.

6.2 Local Type Inference

In order to further relieve the annotation burden on programmers, Typed Scheme provides two simple instances of what has been called "local" type inference (Pierce and Turner, 2000).[8] First, local non-recursive bindings do not require type annotations. For example, the following fragment typechecks without annotations on the local bindings:

```
(define: (m [z : Number]) : Number
  (let* ([x z]
         [y (* x x)])
    (- y 1)))
```

By examining the right-hand sides of the **let**$*$, the typechecker can determine that both x and y should have type **Number**.

The use of internal definitions can complicate this inference process. For example, the above code could be written as follows:

```
(define: (m [z : Number]) : Number
  (define x z)
  (define y (* x x))
  (- y 1))
```

This fragment is macro-expanded into a **letrec**; however, recursive binding is not required for typechecking this code. Therefore, the typechecker analyzes the **letrec** expression and determines if all of the bindings can be treated non-recursively. If so, the above inference method is applied.

Second, local inference also allows the type arguments to polymorphic functions to be omitted. For example, the following use of *map* does not require explicit type instantiation:

```
(map (lambda: ([x : Number]) (+ x 1)) '(1 2 3))
```

To accommodate this form of inference, the typechecker first determines the type of the argument expressions, in this case (**Number** $\rightarrow$ **Number**) and (**Listof Number**), as well as the operator, here (**All** (a b) (($a \rightarrow b$) (**Listof** a) $\rightarrow$ (**Listof** b))). Then it matches the argument types against the body of the operator type, generating a substitution. Finally, the substitution is applied to the function result type to determine the type of the entire expression.

For cases such as the above, this process is quite straightforward. When subtyping is involved, however, the process is complex. Consider this, seemingly similar, example:

```
(map (lambda: ([x : Any]) x) '(1 2 3))
```

Again, the second operand has type (**Listof Number**), suggesting that *map*'s type variable b should substituted with **Number**, the first operand has type (**Any** $\rightarrow$ **Any**), suggesting that both a and b should be **Any**. The solution is to find a common supertype of **Number** and **Any**, and use that to substitute for a.

[8] This modicum of inference is similar to that in recent releases of Java (Gosling et al, 2005).

Unfortunately, this process does not always succeed. Therefore, the programmer must sometimes annotate the arguments or the function to enable the typechecker to find the correct substitution. For example, this annotation instantiates *foldl* at **Number** and **Any**:

#{*foldl* @ **Number Any**}

In practice, we have rarely needed these annotations; local inference almost always succeeds.

6.3 Adapting Scheme Features

PLT Scheme comes with numerous constructs that need explicit support from the type system. We describe several of the more important ones here.

- The most important one is the *structure* system. A **define-struct** definition is *the* fundamental method for constructing new varieties of data in PLT Scheme. This form of definition introduces constructors, predicates, field selectors, and field mutators. Typed Scheme includes a matching **define-struct:** form. Thus the untyped definition

  (**define-struct** *A* (*x y*))

  which defines a structure *A*, with fields *x* and *y*, becomes the following in Typed Scheme:

  (**define-struct:** *A* ([*x* **: Number**] [*y* **: String**]))

  Unsurprisingly, all fields have type annotations.
  The **define-struct:** form, like **define-struct**, introduces the predicate *A?*. Scheme programmers use this predicate to discriminate instances of *A* from other values, and the occurrence typing system must therefore be aware of it. The **define-struct:** definition facility can also automatically introduce recursive types, similar to those introduced via ML's datatype construct.
  Programmers may define structures as extensions of an existing structure, similar to extensions of classes in object-oriented languages. An extended structure inherits all the fields of its parent structure. Furthermore, its parent predicate cannot discriminate instances of the parent structure from instances of the child structure. Hence, it is imperative to integrate structures with the type system at a fundamental level.
- PLT Scheme encourages placing all code in modules, but the top level still provides valuable interactivity. Typed Scheme supports both definitions and expression at the top-level, but support is necessarily limited by the restrictions of typechecking a form at a time. For example, mutually recursive top-level functions cannot be defined, since type checking of the first happens before the second is entered.
- Variable-arity functions also demand special attention from the type perspective. PLT Scheme supports two forms of variable-arity functions: rest parameters, which bundle up extra arguments into a list; and **case-lambda** (Dybvig and Hieb, 1990), which, roughly speaking, introduces dynamic overloading by arity. A careful adaptation of the solutions employed for mainstream languages such as Java and C# suffices for some of these features; for others, we have developed additional type system extensions to handle the unique features of PLT Scheme (Strickland et al, 2009).
- Dually, Scheme supports multiple-value returns, meaning a procedure may return multiple values simultaneously without first bundling them up in a tuple (or other compound values). Multiple values are given special treatment in the type checker because the construct for returning multiple values is a primitive function (**values**), which can be used

in higher-order contexts. Such higher-order uses of **values** benefit from extensions to handle variable-arity polymorphism, as described above (Strickland et al, 2009).

– Finally, Scheme programmers use the *apply* function, especially in conjunction with variable-arity functions. The *apply* function consumes a function, a number of values, plus a list of additional values; it then applies the function to all these values.
  Because of its use in conjunction with variable-arity functions, we type-check the application of *apply* specially and allow its use with variable-arity functions of the appropriate type.
  For example, the common Scheme idiom of *apply*ing the function + to a list of numbers to sum them works in Typed Scheme: (*apply* + (*list* 1 2 3 4)).

6.4 Special Scheme Functions

A number of Scheme functions, either because of their special semantics or their particular roles in the reasoning process of Scheme programmers, are assigned types that demand some explanation. Here we cover just two interesting examples: *filter* and *call/cc*.

An important Scheme function, as we saw in section 2, is *filter*.

When *filter* is used with predicate *p?*, the programmer knows that every element of the resulting list satisfies *p?*. The type system should have this knowledge as well, and in Typed Scheme it does:

filter **:** (**All** (*a b*) ((*a* $\xrightarrow{b}$ **Boolean**) (**Listof** *a*) $\rightarrow$ (**Listof** *b*))

Here we write (*a* $\xrightarrow{b}$ **Boolean**) for the type of functions from *a* to **Boolean** that are predicates for type *b*. Note how the latent predicate of filter becomes the type of the resulting elements. In a setting without occurrence typing, this effect has only been achieved with dependent types or with explicit casting operations.

For an example, consider the following definition:

```
(define: the-numbers (Listof Number)
  (let ([lst (list 'a 1 'b 2 'c 3)])
    (map add1 (filter number? lst))))
```

Here *the-numbers* has type (**Listof Number**) even though it is the result of filtering numbers from a list that contains both symbols and numbers. Using Typed Scheme's type for *filter*, type-checking this expression is now straightforward. *filter* can of course be user-defined, the straightforward implementation is accepted with the above type. The example again demonstrates type inference for local non-recursive bindings.

The type of *call/cc* must reflect the fact that invoking a continuation aborts the local computation in progress:

call/cc **:** (**All** (*a*) (((*a* $\rightarrow \bot$) $\rightarrow$ *a*) $\rightarrow$ *a*))

where $\bot$ is the empty type, expressing the fact that the function cannot produce values. This type has the same logical interpretation as Peirce's law, the conventional type for *call/cc* (Griffin, 1990) but works better with our type inference system.

6.5 Programming in the Large

PLT Scheme has a first-order module system (Flatt, 2002) that allows us to support multi-module typed programs with no extra effort. In untyped PLT Scheme programs, a module

```
#lang typed-scheme
(provide LoN sum)
(define-type-alias LoN (Listof Number))
(define: (sum [l : LoN]) : Number
  (if (null? l) 0 (+ (car l) (sum (cdr l)))))
```

```
#lang typed-scheme
(require m1)
(define: l : LoN (list 1 2 3 4 5))
(display (sum l))
```

Fig. 9 A Multi-Module Typed Scheme Program

consists of definitions and expressions, along with declarations of dependencies on other modules, and of export specifications for identifiers. In Typed Scheme, the same module system is available, without changes. Both defined values and types can be imported or provided from other Typed Scheme modules, with no syntactic overhead. The types of provided identifiers is taken from their initial definition. In the example in figure 9, the type *LoN* and the function *sum* are provided by module *m1* and can therefore be used in module *m2* at their declared types.

Additionally, a Typed Scheme module, like a PLT Scheme module, may contain and export macro definitions that refer to identifiers or types defined in the typed module.

6.6 Interoperating with Untyped Code

Importing from the Untyped World When a typed module must import 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 (Tobin-Hochstadt and Felleisen, 2006). 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* [*add1* (**Number** → **Number**)])

imports the *add1* function from the *scheme* library, with the given type. The **require/typed** facility expands into contracts, which are enforced as values cross module boundaries (Findler and Felleisen, 2002). In this example, the use of **require/typed** is automatically rewritten to a plain *require* along with a contract application using the contract (*number? . → . number?*).

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. 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.

For these situations, Typed Scheme supports *opaque types*, in which only the predicate for testing membership is specified. 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. Of course, the predicate is naturally integrated into the occurrence type system, allowing modules to discriminate precisely the elements of the opaque type.

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

(**require/typed** [**opaque** *xml Doc document?*])

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 refer to *Doc*.

Exporting to the Untyped World When a typed module is required by untyped code, the typed code must be protected (Tobin-Hochstadt and Felleisen, 2006). Since exports from typed code come equipped with a type, they are automatically guarded by contracts, without additional effort or annotation by the programmer. 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.

7 Implementation

We have implemented Typed Scheme as a language for the PLT Scheme environment, and it is available in the standard PLT Scheme distribution (Culpepper et al, 2007). [9] The implementation is available from `http://www.plt-scheme.org` .

Since Typed Scheme is intended for use by programmers developing real applications, a toy implementation was not an option. Fortunately, we were able to implement all of Typed Scheme as a layer on top of PLT Scheme, giving us a full-featured language and standard library. In order to integrate with PLT Scheme, all of Typed Scheme is implemented using the PLT Scheme macro system (Culpepper et al, 2007). When the macro expander finishes successfully, the program has been typechecked, and all traces of Typed Scheme have been compiled away, leaving only executable PLT Scheme code remaining. The module can then be run just as any other Scheme program, or linked with existing modules.

7.1 Changing the Language

Our chosen implementation strategy requires an integration of the type checking and macro expansion processes.

The PLT Scheme macro system allows language designers to control the macro expansion process from the top-most abstract syntax node. Every PLT Scheme module takes the following form:

(**module** *m language*
  ...)

where *language* can specify any library. The library is then used to provide all of the core Scheme forms. For our purposes, the key form is **#%module-begin**, which is wrapped around the entire contents of the module, and expanded before any other expansion or evaluation occurs. Redefining this form gives us complete control over the expansion of a Typed Scheme program. At this point, we can typecheck the module and signal an error at macro-expansion time if it is ill-typed.

[9] The implementation consists of approximately 10000 lines of code and 6800 lines of tests.

7.2 Handling Macros

One consequence of PLT Scheme's powerful macro system is that a large number of constructs that might be part of the core language are instead implemented as macros. This includes pattern matching (Wright and Duba, 1995), class systems (Flatt et al, 2006) and component systems (Flatt and Felleisen, 1998), as well as numerous varieties of conditionals and even boolean operations such as **and**. Faced with this bewildering array of syntactic forms, we could not hope to add each one to our type system, especially since new ones can be added by programmers in libraries or application code. Further, we cannot abandon macros—they are used in virtually every PLT Scheme program, and we do not want to require such changes. Instead, we transform them into simpler code.

In support of such situations, the PLT Scheme macro system provides the *local-expand* primitive, which expands a form in the current syntactic environment. This allows us to fully expand the original program in our macro implementation of Typed Scheme, prior to type checking. We are then left with only the PLT Scheme core forms, of which there are approximately a dozen.

7.3 Cross-Module Typing

In PLT Scheme programs are divided up into first-order modules. Each module explicitly specifies the other modules it imports and the bindings it exports. In order for Typed Scheme to work with actual PLT Scheme programs, it must be possible for programmers to split up their Typed Scheme programs into multiple modules.

Our type-checking strategy requires that all type-checking take place during the expansion of a particular module. Therefore, the type environment constructed during the typechecking of one module disappears before any other module is considered.

Instead, we turn the type environments into persistent code using Flatt's reification strategy (Flatt, 2002). After typechecking each module, the type environment is reified in the code of the module as instructions for recreating that type environment when that module is expanded. Since every dependency of a module is visited during the expansion of that module, the appropriate type environment is recreated for each module that is typechecked. This implementation technique has the significant benefit that it provides separate compilation and typechecking of modules for free.

Further, our type environments are keyed by PLT Scheme identifiers, which maintain information on which module they were defined in, providing several advantages. First, the technique described by Flatt (2002) and adapted for Typed Scheme by Culpepper et al (2007) allows the use of one typed module from another without having to redeclare types. Second, standard tools for operating on PLT Scheme programs, such as those provided by DrScheme (Findler et al, 2002) work properly with typed programs and binding of types.

7.4 Performance

There are three important aspects to the performance of Typed Scheme: the performance of the typechecker itself, the overhead of contracts generated for interoperation, and the overhead that Typed Scheme's runtime support imposes on purely typed program. We address each in turn.

The typechecker is currently notably slower than macro expansion without typechecking, but is not problematically slow. Even large files typecheck in just a few seconds. We have optimized the typechecker significantly over the development of Typed Scheme; the most significant optimization is interning of all type representations, allowing constant-time type comparison and substantially reducing memory use.

The overhead of contracts can be substantial, depending on the particular contracts generated. In some cases, contracts can change the asymptotic complexity of existing programs. We hope to investigate techniques for lazy checking of contracts (Findler et al, 2007) to alleviate this problem. However, this overhead is only imposed when crossing the typed-untyped boundary, which we predict will be rare in inner loops and other performance critical code. Adding types to selected portions of the DrScheme (Findler et al, 2002) implementation resulted in no measurable slowdown.

Finally, the implementation of Typed Scheme imposes no runtime overhead on programs, with the exception of the need to load the code associated with the library. Thus typed code executes at full speed. We are investigating optimization opportunities based on the type information (St-Amour et al, 2010).

7.5 Limitations

Our implementation has two significant limitations at present. First, we are unable to dynamically enforce some types using the PLT contract system. For example, although checking polymorphic types (Guha et al, 2007; Ahmed et al, 2009) is supported, variable-arity polymorphism is not. Additionally, mutable data continues to present problems for contracts. As solutions for these limitations are integrated into the PLT Scheme contract system, more of Typed Scheme's types will be dynamically enforceable.

The second major limitation is that we cannot typecheck code that uses the most complex PLT Scheme macros, such as the *unit* and *class* systems. These macros maintain their own invariants, which must be understood by the typechecker in order to sensibly type the program. For example, the *class* macro maintains a *vtable* for each class, which in this implementation is a set of methods indexed by symbols. Typing such an implementation would require either a significant increase in the complexity of the type system or special handling of such macros. Since these macros are widely used by PLT Scheme programmers, we plan to investigate both possibilities.

8 Practical Experience

To determine whether Typed Scheme is practical and whether converting PLT Scheme programs is feasible, we conducted a series of experiments in porting existing Scheme programs of varying complexity to Typed Scheme.

Educational Code For small programs, which we expected to be written in a disciplined style that would be easy to type-check, we turned to educational code. Our preliminary investigations and type system design indicated that programs in the style of *How to Design Programs* (Felleisen et al, 2001) would type-check successfully with our system, with only type annotations required.

To see how more traditional educational Scheme code would fare, we rewrote most programs from two additional text books: *The Little Schemer* (Friedman and Felleisen, 1997) and *The Seasoned Schemer* (Friedman and Felleisen, 1996). Converting these 500 lines of

code usually required nothing but the declaration of types for function headers. The only difficulty encountered was an inability to express in our type system some invariants on S-expressions that the code relied on.

Second, we ported 1,000 lines of educational code, which consisted of the solutions to a number of exercises for an undergraduate programming languages course. Again, handing S-expressions proved the greatest challenge, since the code used tests of the form (*pair?* (*car x*)), which does not provide useful information to the type system (formally, the visible predicate of this expression is •). Typing such tests required adding new local bindings. This code also made use of a non-standard datatype definition facility, which required adaptation to work with Typed Scheme.

Libraries We ported 500 lines of code implementing a variety of data structures from Søgaard's *galore.plt* library package. While these data structures were originally designed for a typed functional language, the implementations were not written with typing in mind. Two sorts of changes were required for typing this library. First, in several places the library failed to check for erroneous input, resulting in potentially surprising behavior. Correcting this required adding tests for the erroneous cases. Second, in about a dozen places throughout the code, polymorphic functions needed to be explicitly instantiated in order for typechecking to proceed. These changes were, again, in addition to the annotation of bound variables.

Applications A research intern ported two sizable applications under the direction of the first author. The first was a 2,700 line implementation of a game, written in 2007, and the second was a 500 line checkbook managing script, maintained for 12 years.

The game is a version of the multi-player card game Squadron Scramble.[10] The original implementation consists of 10 PLT Scheme modules, totaling 2,700 lines of implementation code, including 500 lines of unit tests.

A representative function definition from the game is given in figure 10. This function creates a *turn* object, and hands it to the appropriate *player*. It then checks whether the game is over and if necessary, constructs the new state of the game and returns it.

The changes to this complex function are confined to the function header. We have converted the original **define** to **define:** and provided type annotations for each of the formal parameters as well as the return type. This function returns multiple values, as is indicated by the return type. Other than the header, no changes are required. The types of all the locally bound variables are inferred from the bodies of the individual definitions.

Structure types are used extensively in this example, as well as in the entire implementation. In the definition of the variables *the-end* and *the-return-card*, occurrence typing is used to distinguish between the *res* and *end* structures.

Some portions of the implementation required more effort to port to Typed Scheme. For example, portions of the data used for the game is stored in external XML files with a fixed format, and the program relies upon the details of that format. However, since this invariant is neither checked nor specified in the program, the type system cannot verify it. Therefore, we moved the code handling the XML file into a separate, untyped module of fewer than 50 lines that the typed portion uses via **require/typed**.

Scripts The second application ported required similarly few changes. This script maintained financial records recorded in an S-expression stored in a file. The major change made to the program was the addition of checks to ensure that data read from the file was in the correct format before using it to create the relevant internal data structures. This was similar to the issue encountered with the Squadron Scramble game, but since the problem concerned

[10] Squadron Scramble resembles Rummy; it is available from US Game Systems.

```
(: play-one-turn (Player Cards Cards Hand →
                  (values Boolean RCard Hand Attacks From)))
(define (play-one-turn player deck stck fst:discs)
  (define trn (create-turn (player-name player) deck stck fst:discs))
  ;; — go play
  (define res (player-take-turn player trn))
  ;; the-return-card could be false
  (define-values (the-end the-return-card)
    (cond
      [(ret? res) (values #f (ret-card res))]
      [(end? res) (values #t (end-card res))]))
  (define discards:squadrons  (done-discards res))
  (define attacks  (done-attacks res))
  (define et (turn-end trn))
  (values the-end the-return-card discards:squadrons attacks et))
```

Fig. 10 A Excerpt from the Squadron Scramble Game

a single function, we added the necessary checks rather than creating a new module. The other semantic change to the program was to maintain a typing invariant of a data structure by construction, rather than after-the-fact mutation. As in the case of the Galore library, we consider this typechecker-mandated change an improvement to the original program, even though it has already been used successfully for many years.

9 Related Work

The history of programming languages knows many attempts to add or to use type information in conjunction with untyped languages. Starting with LISP (Steele Jr., 1984), language designers have tried to include type declarations in such languages, often to help compilers, sometimes to assist programmers. From the late 1980s until recently, people have studied soft typing (Cartwright and Fagan, 1991; Aiken et al, 1994; Wright and Cartwright, 1997; Henglein and Rehof, 1995; Flanagan and Felleisen, 1999; Meunier et al, 2006), 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 Python (Salib, 2004). Recently, the slogan of "gradual typing" has resurrected the LISP-style annotation mechanisms and has had a first impact with its tentative inclusion in Perl6 (Tang, 2007).

In this section, we survey this body of work, starting with the soft-typing strand, because it is the closest relative of Typed Scheme. We conclude with a discussion of refinement types.

9.1 Types for Scheme

The goal of the soft typing research agenda 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 an ill-typed program, it should insert run-time checks that restore typability and ensure that the type system remains sound. Naturally, a soft type system should minimize these insertions of run-time checks. Furthermore, since these insertions represent potential failures of type invariants, a good soft type

system must allow programmer to inspect the sites of these run-time checks to determine whether they represent genuine errors or weaknesses of the type system.

Based on the experiences of the second author, soft type systems are complex and brittle. On one hand, these systems may infer extremely large types for seemingly simple expressions, greatly confusing the original programmer or the programmer who has taken on old code. On the other hand, a small syntactic change to a program without semantic consequences can introduce vast changes into the types of both nearby and remote expressions. Experiments with undergraduates—representative of average programmers—suggest that only the very best understood the tools well enough to make sense of the inferred types and to exploit them for the assigned tasks. For the others, these tools turned into time sinks with little benefit.

Roughly speaking, soft typing systems fall into one of two classes, depending on the kind of underlying inference system. The first soft type systems (Cartwright and Fagan, 1991; Wright and Cartwright, 1997; Henglein and Rehof, 1995; Henglein, 1994) used inference engines based on Hindley-Milner though with extensible record types. These systems are able to type many actual Scheme programs, including those using outlandish-looking recursive datatypes. Unfortunately, these systems severely suffer from the general Hindley-Milner error-recovery problem. That is, when the type system signals a type error, it is extremely difficult—often impossible—to decipher its meaning and to fix it.

In response to this error-recovery problem, others built inference systems based on Shiver's control-flow analyses (1991) and Aiken's and Heintze's set-based analyses (Aiken et al, 1994; Heintze, 1994). Roughly speaking, these soft typing systems introduce sets-of-values constraints for atomic expressions and propagate them via a generalized transitive-closure propagation (Aiken et al, 1994; Flanagan and Felleisen, 1999). In this world, it is easy to communicate to a programmer how a values might flow into a particular operation and violate a type invariant, thus eliminating one of the major problems of Hindley-Milner based soft typing (Flanagan et al, 1996).

Our experience and evaluation suggest that Typed Scheme works well compared to soft typing. First, programmers can easily convert entire modules with just a few type declarations and annotations to function headers. Second, assigning explicit types and rejecting programs actually pinpoints errors better than soft typing systems, where programmers must always keep in mind that the type inference system is conservative. Third, soft typing systems do not support type abstractions well. Starting from an explicit, static type system for an untyped language should help introduce these features and deploy them as needed.

The Rice University soft typing research inspired occurrence typing. These systems employed **if**-splitting rules that performed a case analysis for types based on the syntactic predicates in the test expression. This idea was derived from Cartwright (1976)'s `typecase` construct (also see below) and—due to its usefulness—is generalized by our framework. The major advantage of soft typing over an explicitly typed Scheme is that it does not require any assistance from the programmer. In the future, we expect to borrow techniques from soft typing for automating some of the conversion process from untyped modules to typed modules.

Shivers (1991) presented 0CFA, which also uses flow analysis for Scheme programs. He describes a possible extension to account for occurrence-typing like behavior for literal applications of the predicate *number?*, but did not discuss more general aspects of the issue.

Henglein and Rehof (1995) used a flow analysis to convert Scheme programs to ML programs, while minimizing runtime checks. While this is also converting Scheme programs to typed programs, it is intended as a compilation step, not a refactoring, and the ML code is

not intended to be maintained as the primary form of the program. Additionally, their system does not take predicate tests into account, which is the primary focus of occurrence typing.

Aiken et al (1994) describe a type inference system using *conditional types*, which refine the types of variables based on patterns in a `case` expression. Since this system is built on the use of patterns, abstracting over tests as the T-ABSPRED rule does, or combining them, as with **or** is impossible.

9.2 Gradual Typing

Under the name "gradual typing", several other researchers have experimented with the integration of typed and untyped code (Siek and Taha, 2006; Herman et al, 2008; Wadler and Findler, 2009; Wrigstad et al, 2009). This work has been pursued in two directions. First, theoretical investigations have considered integration of typed and untyped code at a much finer granularity than we present, providing soundness theorems which prove that only the untyped portions of the program can go wrong. This is analogous to our earlier work on Typed Scheme (Tobin-Hochstadt and Felleisen, 2006), which provides such a soundness theorem, which we believe scales to full Typed Scheme and PLT Scheme. These gradual typing systems have not been scaled to full implementations.

Second, Furr et al (2009a,b) have implemented a system for Ruby which is similar to Typed Scheme. They have also designed a type system which matches the idioms of the underlying language, and insert dynamic checks at the borders between typed and untyped code. Their work does not yet have a published soundness theorem, and requires the use of a new Ruby interpreter, whereas Typed Scheme runs purely as a library for PLT Scheme.

Bracha (2004) suggests pluggable typing systems, in which a programmer can choose from a variety of type systems for each piece of code. Although Typed Scheme requires some annotation, it can be thought of as a step toward such a pluggable system, in which programmers can choose between the standard PLT Scheme type system and Typed Scheme on a module-by-module basis.

9.3 Type System Features

Many of the type system features we have incorporated into Typed Scheme have been extensively studied. Polymorphism in type systems dates to Reynolds (1983). Recursive types are considered by Amadio and Cardelli (1993), and union types by Pierce (1991), among many others. Intensional polymorphism appears in calculi by Harper and Morrisett (1995), among others. Our use of visible predicates and especially latent predicates is inspired by prior work on effect systems (Gifford et al, 1987).

9.4 Other Type Systems

Cartwright (1976) describes Typed Lisp, which includes `typecase` expression that refines the type of a variable in the various cases; Crary et al (1998) re-invent this construct in the context of a typed lambda calculus with intensional polymorphism. The `typecase` statement specifies the variable to be refined, and that variable is typed differently on the right-hand sides of the `typecase` expression. While this system is superficially similar to our type system, the use of latent and visible predicates allows us to handle cases other than simple

uses of `typecase`. This is important in type-checking existing Scheme code, which is not written with `typecase` constructs.

Visible predicates can also be seen as a kind of dependent type, in that (*number? e*) could be thought of as having type **true** when *e* has a value that is a number. In a system with singleton types, this relationship could be expressed as a dependent type. This kind of combination typing would not cover the use of **if** to refine the types of variables in the branches, however.

The term "occurrence typing" was coined independently by Komondoor et al (2005), in the context of a static analysis system for Cobol. That system considers a specific syntactic form of **if** tests: the comparison of variables with character literals. This accommodates a common encoding of datatypes in Cobol programs. It does not allow for abstraction over tests or any other form of predicates.

9.5 Type Systems for Untyped Languages

Multiple previous efforts have attempted to typecheck Scheme programs. Wand (1984), Haynes (1995), and Leavens et al (2005) developed typecheckers for an ML-style type system, each of which handle polymorphism, structure definition and a number of Scheme features. Wand's macro-based system integrated with untyped Scheme code via unchecked assertions. Haynes' system also handles variable-arity functions (Dzeng and Haynes, 1994). However, none attempts to accommodate a traditional Scheme programming style.

Bracha and Griswold's Strongtalk (1993), like Typed Scheme, presents a type system designed for the needs of an untyped language, in their case Smalltalk. Reflecting the differing underlying languages, the Strongtalk type system differs radically from ours and does not describe a mechanism for integrating with untyped code.

9.6 Refinement Types

Refinement types were originally introduced by Freeman and Pfenning (1991). Since then, refinement types have been used in a wide variety of systems (Rondon et al, 2008; Wadler and Findler, 2009; Flanagan, 2006). Previous refinement type systems come in two varieties. Freeman and Pfenning's original system used the underlying language of ML types to specify subsets of the existing types, such as non-empty lists, defined by recursive datatype-like specifications. Most other systems have paired predicates in some potentially-restricted language with a base type, meaning the set of values of that base type accepted by that predicate. Typically, this requires some algorithm for deciding implication between predicates for subtyping. In some languages, this can be an external and almost always incomplete theorem prover, as in the Liquid Typing and Hybrid Typing approaches.

Typed Scheme provides support for both of these approaches, as seen in section 3. To support Freeman and Pfenning's style, such data types can be directly encoded via recursive types. Typed Scheme is able to handle all of Freeman and Pfenning's examples in this fashion. To support predicate style-refinement, Typed Scheme takes a different approach. First, refinements are not specified using a special language of predicates or formulae but as in-language predicates. This allows any computable set to be a refinement. Second, no attempt is made to decide implication between predicates. Two distinct functions might be extensionally equivalent, but the associated refinement types have no subtyping relationship. This

frees both the programmer and the implementor from the burden of depending on a theorem prover.

10 Conclusion

Migrating programs from untyped languages to typed languages is an important problem. In this paper we have demonstrated one successful approach, based on the development of a type system that accommodates the idioms and programming styles of our scripting language of choice.

Our type system combines a simple new idea, occurrence typing, with a range of previously studied type system features with some widely used and some only studied in theory. Occurrence typing assigns distinct subtypes of a parameter to distinct occurrences, depending on the control flow of the program. We introduced occurrence typing because our past experience suggests that Scheme programmers combine flow-oriented reasoning with typed-based reasoning. Occurrence typing also allows us to naturally extend the type system with a simple and expressive form of refinement types, allowing for static verification of arbitrary property checking.

Building upon this design, we have implemented and distributed Typed Scheme as a package for the PLT Scheme system. This implementation supports the key type system features discussed here, as well as integration features necessary for interoperation with the rest of the PLT Scheme system.

Using Typed Scheme, we have evaluated our type system. We consider the experiments of section 8 illustrative of existing code and believe that their success is a good predictor for future experiments. We plan on continuing to port PLT Scheme libraries to Typed Scheme and on exploring the theory of occurrence typing in more depth.

For a close look at Typed Scheme, including documentation and sources for its Isabelle/HOL and PLT Redex models, visit the Typed Scheme web page:

`http://www.ccs.neu.edu/~samth/typed-scheme`

Acknowledgements We thank Ryan Culpepper for invaluable assistance with the implementation of Typed Scheme, Matthew Flatt for implementation advice, Ivan Gazeau for his porting of existing PLT Scheme code, and members of Northeastern PRL as well as several anonymous reviewers for their comments.

References

Ahmed A, Findler RB, Matthews J, Wadler P (2009) Blame for all. In: (Wrigstad et al, 2009), pp 1–13

Aiken A, Wimmers EL, Lakshman TK (1994) Soft typing with conditional types. In: POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 163–173

Amadio RM, Cardelli L (1993) Subtyping recursive types. ACM Transactions on Programming Languages and Systems 15(4):575–631

Bracha G (2004) Pluggable type systems. In: OOPSLA Workshop on the Revival of Dynamic Languages

Bracha G, Griswold D (1993) Strongtalk: typechecking Smalltalk in a production environment. In: OOPSLA '93: Proceedings of the 8th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, pp 215–230

Cartwright R (1976) User-defined data types as an aid to verifying LISP programs. In: International Conference on Automata, Languages and Programming, pp 228–256

Cartwright R, Fagan M (1991) Soft typing. In: PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, pp 278–292

Crary K, Weirich S, Morrisett G (1998) Intensional polymorphism in type-erasure semantics. In: ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, pp 301–312

Culpepper R, Tobin-Hochstadt S, Flatt M (2007) Advanced Macrology and the Implementation of Typed Scheme. In: Proceedings of the 2007 Workshop on Scheme and Functional Programming, Université Laval Technical Report DIUL-RT-0701, pp 1–13

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

Dzeng H, Haynes CT (1994) Type reconstruction for variable-arity procedures. In: Proceedings of the 1994 ACM Conference on LISP and Functional Programming, pp 239–249

ECMA (2007) ECMAScript Edition 4 group wiki. URL `http://wiki.ecmascript.org/`

Felleisen M, Findler RB, Flatt M, Krishnamurthi S (2001) How to Design Programs. MIT Press, URL `http://www.htdp.org/`

Findler RB, Felleisen M (2002) Contracts for higher-order functions. In: ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, pp 48–59

Findler RB, Clements J, Flanagan C, Flatt M, Krishnamurthi S, Steckler P, Felleisen M (2002) DrScheme: A programming environment for Scheme. Journal of Functional Programming 12(2):159–182

Findler RB, Guo S, Rogers A (2007) Lazy contract checking for immutable data structures. In: Proceedings of the International Symposium on Implemenation and Application of Functional Languages (IFL)

Flanagan C (2006) Hybrid type checking. In: Conference Record of POPL '06: The 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 245–256

Flanagan C, Felleisen M (1999) Componential set-based analysis. ACM Transactions on Programming Languages and Systems 21(2):370–416

Flanagan C, Flatt M, Krishnamurthi S, Weirich S, Felleisen M (1996) Catching bugs in the web of program invariants. In: PLDI '96: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, pp 23–32

Flatt M (2002) Composable and compilable macros: You want it *when?* In: ICFP '02: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming, pp 72–83

Flatt M, Felleisen M (1998) Units: Cool modules for HOT languages. In: PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pp 236–248

Flatt M, Findler RB, Felleisen M (2006) Scheme with classes, mixins, and traits. In: Asian Symposium on Programming Languages and Systems (APLAS) 2006, Lecture Notes in Computer Science, vol 4279, pp 270–289

Freeman T, Pfenning F (1991) Refinement types for ML. In: PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, pp 268–277

Friedman DP, Felleisen M (1996) The Seasoned Schemer. MIT Press, Cambridge

Friedman DP, Felleisen M (1997) The Little Schemer, Fourth Edition. MIT Press, Cambridge

Furr M, An J, Foster JS, Hicks M (2009a) Static type inference for ruby. In: SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing, pp 1859–1866

Furr M, An J, Foster JS, Hicks M (2009b) Tests to the left of me, types to the right: how not to get stuck in the middle of a Ruby execution. In: (Wrigstad et al, 2009), pp 14–16

Gifford D, Jouvelot P, Lucassen J, Sheldon M (1987) FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science

Gosling J, Joy B, Steele Jr GL, Bracha G (2005) The Java Language Specification, 3rd edn. Addison-Welsley

Griffin TG (1990) A formulae-as-type notion of control. In: POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 47–58

Guha A, Matthews J, Findler RB, Krishnamurthi S (2007) Relationally-parametric polymorphic contracts. In: Proceedings of the 2007 Symposium on Dynamic languages, pp 29–40

Harper R, Morrisett G (1995) Compiling polymorphism using intensional type analysis. In: POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 130–141

Haynes CT (1995) Infer: A statically-typed dialect of Scheme. Technical Report 367, Indiana University

Heintze N (1994) Set based analysis of ML programs. In: LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming, pp 306–317

Henglein F (1994) Dynamic typing: syntax and proof theory. In: Selected papers of the symposium on Fourth European symposium on programming, pp 197–230

Henglein F, Rehof J (1995) 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, pp 192–203

Herman D, Tomb A, Flanagan C (2008) Space-efficient gradual typing. In: Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, pp 1–18

Komondoor R, Ramalingam G, Chandra S, Field J (2005) Dependent types for program understanding. In: Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol 3440, pp 157–173

Leavens GT, Clifton C, Dorn B (2005) A Type Notation for Scheme. Tech. Rep. 05-18a, Iowa State University

Matthews J, Findler R, Flatt M, Felleisen M (2004) A visual environment for developing context-sensitive term rewriting systems. In: Rewriting Techniques and Applications, 15th International Conference, Lecture Notes in Computer Science, vol 3091, pp 2–16

Mehnert H (2009) Extending Dylan's type system for better type inference and error detection. Diplom Arbeit, Technische Universität Berlin, Berlin, Germany

Meunier P, Findler RB, Felleisen M (2006) Modular set-based analysis from contracts. In: Conference Record of POPL '06: The 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 218–231

Milner R, Tofte M, Harper R, MacQueen D (1997) The Definition of Standard ML (Revised). MIT Press

Munroe R (2007) Exploits of a mom. `http://xkcd.com/327/`

Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol 2283. Springer-Verlag

Pierce BC (1991) Programming with intersection types, union types, and polymorphism. Tech. Rep. CMU-CS-91-106, Carnegie Mellon University

Pierce BC, Turner DN (2000) Local type inference. ACM Transactions on Programming Languages and Systems 22(1):1–44

Reynolds JC (1983) Types, abstraction, and parametric polymorphism. In: Mason REA (ed) Information Processing 83, Paris, France, pp 513–523

Rondon PM, Kawaguci M, Jhala R (2008) Liquid types. In: PLDI '08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, pp 159–169

Sabry A, Felleisen M (1993) Reasoning about programs in continuation-passing style. Lisp and Symbolic Computation 6:289–360

Salib M (2004) Starkiller: A static type inferencer and compiler for Python. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts

Shivers O (1991) Control-flow analysis of higher-order languages or taming lambda. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania

Siek JG, Taha W (2006) Gradual typing for functional languages. In: Seventh Workshop on Scheme and Functional Programming, University of Chicago Technical Report TR-2006-06, pp 81–92

Søgaard JA (2006) Galore. URL `http://planet.plt-scheme.org/`

St-Amour V, Tobin-Hochstadt S, Flatt M, Felleisen M (2010) Where are you going with those types? In: Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010), Utrecht University Technical Report UU-CS-2010-020

Steele Jr GL (1984) Common Lisp—The Language. Digital Press, Bedford, MA

Strickland TS, Tobin-Hochstadt S, Felleisen M (2009) Practical variable-arity polymorphism. In: ESOP '09: Proceedings of the Eighteenth European Symposium on Programming, Lecture Notes in Computer Science, vol 5502, pp 32–46

Tang A (2007) Perl 6: reconciling the irreconcilable. In: Conference Record of POPL '07: The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, p 1, `http://pugscode.org`

Tobin-Hochstadt S, Felleisen M (2006) Interlanguage migration: from scripts to programs. In: OOPSLA '06: Companion to the 21st Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, pp 964–974

Urban C (2008) Nominal techniques in Isabelle/HOL. Journal of Automated Reasoning 40(4):327–356

Vytiniotis D, Weirich S, Jones SP (2006) Boxy types: inference for higher-rank types and impredicativity. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, pp 251–262

Wadler P, Findler RB (2009) Well-typed programs can't be blamed. In: ESOP '09: Proceedings of the Eighteenth European Symposium on Programming, Lecture Notes in Computer Science, vol 5502, pp 1–16

Wand M (1984) A semantic prototyping system. In: SIGPLAN '84: Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, pp 213–221

Wright A, Duba B (1995) Pattern matching for Scheme. Unpublished note, Rice University

Wright A, Felleisen M (1994) A syntactic approach to type soundness. Information & Computation 115(1):38–94

Wright AK, Cartwright R (1997) A practical soft type system for Scheme. ACM Transactions on Programming Languages and Systems 19(1):87–152
Wrigstad T, Nystrom N, Vitek J (eds) (2009) STOP '09: Proceedings for the 1st Workshop on Script to Program Evolution, ACM Press, New York, NY, USA