The Design and Implementation of Typed Scheme

Sam Tobin-Hochstadt Matthias Felleisen

PLT, Northeastern University
Boston, MA 02115

Abstract

When scripts in untyped languages grow into large programs, maintaining them becomes difficult. A lack of types in typical scripting languages means that programmers 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 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. Initial experiments with the implementation suggest that Typed Scheme naturally accommodates the programming style of the underlying scripting language, at least for the first few thousand lines of ported code.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Language Constructs and Features*]: Modules, Packages; D.3.m [*Miscellaneous*]: Cartesian Closed

General Terms Languages, Design

Keywords Type Systems, Scheme

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.

Most scripting languages are untyped and have a flexible semantics that makes 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 International 2007) are implicitly acknowledging this problem with the addition of Common Lisp-style (Steele Jr. 1984) typing constructs to the upcoming releases of their respective languages.

In the meantime, industry faces the problem of porting existing application systems from untyped scripting languages to the typed world. Based on our own experience, 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). 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, such an assumption seems 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 easy to turn portions of a multi-module program into a partially typed yet still executable program.

Here we report on the novel type system, which combines the idea of *occurrence* typing with subtyping, recursive types, polymorphism and a modicum of inference. We first present a formal model of the key aspects of occurrence typing and prove it to be type-sound. 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.

POPL'08, January 7–12, 2008, San Francisco, California, USA.

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.

Consider the following function defintion:[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. All it uses is a predicate that distinguishes the two cases: the first **cond** clause, which deals with *x* as a number and the second one, which treats it as a pair.

Here is the corresponding Typed Scheme code:[2]

```
(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.

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

```
(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 being prepared 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*.

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 would require the use of an intermediate data structure, such as an option type, to ensure conformance.

2.1 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 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.

2.2 S-expressions

One of the primary Scheme data structures is the S-expression. We have already seen an example of this in the foregoing 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**).

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 module dealing with XML in the untyped world, communicating with the typed portions of the program just as other untyped libraries do.

2.3 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 variable-arity and multiple-arity functions; and many others. All of these features are widely used in existing PLT Scheme programs, and supported by Typed Scheme.

2.4 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 the majority

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

[2] In this paper, we typeset Typed Scheme code in a manner that differs slightly from what programmers enter into an editor.

$$
\begin{array}{lll}
d, e, \ldots ::= x \mid (e_1\ e_2) \mid (\textbf{if}\ e_1\ e_2\ e_3) \mid v & & \text{Expressions} \\
v ::= c \mid b \mid n \mid \lambda x : \tau.e & & \text{Values} \\
c ::= \mathit{add1} \mid \mathit{number?} \mid \mathit{boolean?} \mid \mathit{procedure?} \mid \mathit{not} & & \text{Primitive Operations} \\
E ::= [] \mid (E\ e) \mid (v\ E) \mid (\textbf{if}\ E\ e_2\ e_3) & & \text{Evaluation Contexts} \\
\phi ::= \tau \mid \bullet & & \text{Latent Predicates} \\
\psi ::= \tau_x \mid x \mid \textbf{true} \mid \textbf{false} \mid \bullet & & \text{Visible Predicate} \\
\sigma, \tau ::= \top \mid \textbf{Number} \mid \textbf{true} \mid \textbf{false} \mid (\sigma \xrightarrow{\phi} \tau) \mid (\bigcup\ \tau \ldots) & & \text{Types}
\end{array}
$$

Figure 1. Syntax

$$
\begin{array}{llll}
\text{T-VAR} & \text{T-NUM} & \text{T-CONST} & \text{T-TRUE} \\
\Gamma \vdash x : \Gamma(x); x & \Gamma \vdash n : \textbf{Number}; \textbf{true} & \Gamma \vdash c : \delta_\tau(c); \textbf{true} & \Gamma \vdash \textbf{true} : \textbf{Boolean}; \textbf{true}
\end{array}
$$

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

$$
\text{T-ABS}\quad \frac{\Gamma, x : \sigma \vdash e : \tau; \psi}{\Gamma \vdash \lambda x : \sigma.e : (\sigma \xrightarrow{\bullet} \tau); \textbf{true}}
$$

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

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

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

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

Figure 2. Primary Typing Rules

$$
\begin{array}{l}
\text{combpred}(\psi', \psi, \psi) = \psi \\
\text{combpred}(\tau_x, \textbf{true}, \sigma_x) = (\bigcup\ \tau\ \sigma)_x \\
\text{combpred}(\textbf{true}, \psi_1, \psi_2) = \psi_1 \\
\text{combpred}(\textbf{false}, \psi_1, \psi_2) = \psi_2 \\
\text{combpred}(\psi, \textbf{true}, \textbf{false}) = \psi \\
\text{combpred}(\psi_1, \psi_2, \psi_3) = \bullet
\end{array}
$$

$$
\begin{array}{l}
\delta_\tau(\mathit{add1}) = (\textbf{Number} \xrightarrow{\bullet} \textbf{Number}) \\
\delta_\tau(\mathit{not}) = (\top \xrightarrow{\bullet} \textbf{Boolean}) \\
\delta_\tau(\mathit{procedure?}) = (\top \xrightarrow{(\bot \xrightarrow{\bullet} \top)} \textbf{Boolean}) \\
\delta_\tau(\mathit{number?}) = (\top \xrightarrow{\textbf{Number}} \textbf{Boolean}) \\
\delta_\tau(\mathit{boolean?}) = (\top \xrightarrow{\textbf{Boolean}} \textbf{Boolean})
\end{array}
$$

Figure 3. Auxiliary Operations

$$
\begin{array}{l}
\Gamma + \tau_x = \Gamma[x : \text{restrict}(\Gamma(x), \tau)\ ] \\
\Gamma + x = \Gamma[x : \text{remove}(\Gamma(x), \textbf{false})\ ] \\
\Gamma + \bullet = \Gamma \\
\Gamma - \tau_x = \Gamma[x : \text{remove}(\Gamma(x), \tau)\ ] \\
\Gamma - x = \Gamma[x : \textbf{false}\ ] \\
\Gamma - \bullet = \Gamma
\end{array}
$$

$$
\begin{array}{l}
\text{restrict}(\sigma, \tau) = \sigma \ \text{ when} \vdash \sigma <: \tau \\
\text{restrict}(\sigma, (\bigcup\ \tau \ldots)) = (\bigcup \text{restrict}(\sigma, \tau) \ldots) \\
\text{restrict}(\sigma, \tau) = \tau \ \text{ otherwise} \\
\text{remove}(\sigma, \tau) = \bot \ \text{ when} \vdash \sigma <: \tau \\
\text{remove}(\sigma, (\bigcup\ \tau \ldots)) = (\bigcup \text{remove}(\sigma, \tau) \ldots) \\
\text{remove}(\sigma, \tau) = \sigma \ \text{ otherwise}
\end{array}
$$

Figure 4. Environment Operations

of existing 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. 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 novel aspect of the type system, 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.

3.1 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**.

3.2 Typing Rules

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

```
(λ (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 ($\bigcup$ **Number Boolean**), then the computed type of the function is confirmed as

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

The type system for λ_{TS} accomplishes this; 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; indeed, we can prove that those 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 and need little explanation. 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 predicate (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,

$$number? : (\textbf{Number} \overset{\textbf{Number}}{\to} \textbf{Boolean})$$

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

$$(\lambda\ (x{:}\,\textbf{Number})\ (number?\ x)) : (\textbf{Number} \overset{\textbf{Number}}{\to} \textbf{Boolean}).$$

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:

(**or** (*number? x*) (*boolean? x*))

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

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

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

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.[3] The type system proper comprises the ten rules in figure 2.

[3] Other control flow constructs in Scheme are almost always macros that expand into **if**.

$$\text{S-Refl}\ \ \vdash \tau <: \tau \qquad \text{S-Fun}\ \ \frac{\vdash \sigma_1 <: \tau_1 \qquad \vdash \tau_2 <: \sigma_2 \qquad \phi = \phi' \text{ or } \phi' = \bullet}{\vdash (\tau_1 \overset{\phi}{\to} \tau_2) <: (\sigma_1 \overset{\phi'}{\to} \sigma_2)}$$

$$\text{S-UnionSuper}\ \ \frac{\vdash \tau <: \sigma_i \qquad 1 \le i \le n}{\vdash \tau <: (\bigcup\ \sigma_1 \cdots \sigma_n)} \qquad \text{S-UnionSub}\ \ \frac{\vdash \tau_i <: \sigma \text{ for all } 1 \le i \le n}{\vdash (\bigcup\ \tau_1 \cdots \tau_n) <: \sigma}$$

Figure 5. Subtyping Relation

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 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 the application of an expression to another expression that is not a variable 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 CombPred metafunction combines the effects of the test, then and else branches of an **if** expression. The most interesting case is the second, which handles expressions such as this:

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

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

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

For the motivating example from the beginning of this section,

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

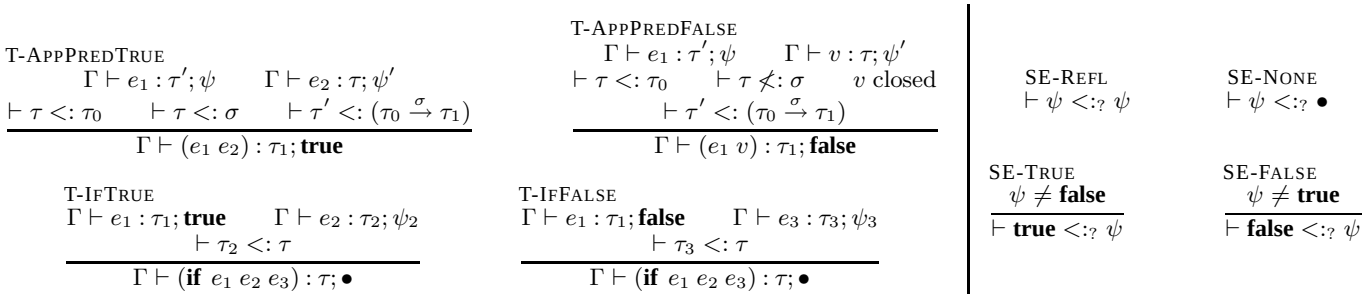

Figure 6. Auxiliary Typing Rules

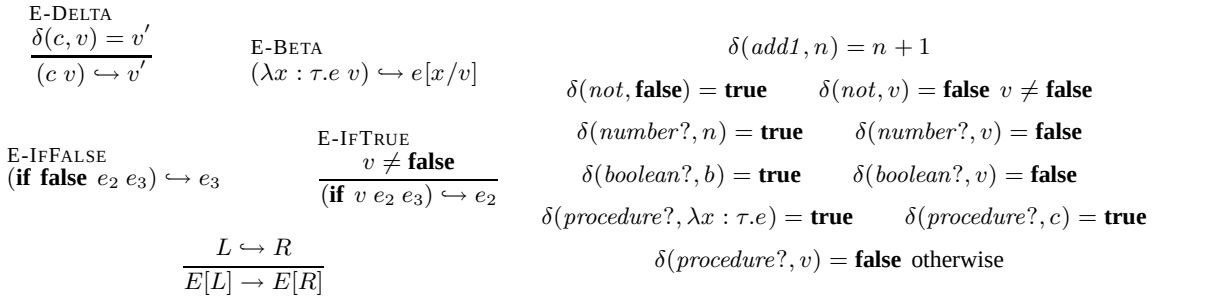

Figure 7. Operational Semantics

we can now see that the test of the **if** expression has type **Boolean** and visible predicate $\mathbf{Number}_x$. Thus, 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. The rules for union types are adapted from Pierce's (Pierce 1991). One important consequence of these rules is that $\perp$ is below all other types. This type is useful for typing functions that do not return to their continuation, 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.

3.3 Proof-Theoretic 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.[4]

In order to properly state the subject reduction lemma, we need to relate the visible predicates of terms in a reduction sequence. Therefore, we define a sub-predicate relation, written $\vdash \psi <:_? \psi'$, for this purpose. The relation is defined in figure 5. This relation is not used in the subtyping or typing rules, and is only necessary for the soundness proof.

We can now prove the traditional lemmas.

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

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

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 $\vdash \psi' <:_? \psi$.*

3.4 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,

[4] 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).

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 and Tasson 2005). 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]

4. From λ_{TS} To Typed Scheme

In this day and age, it is easy to design a type system and it is reasonably straightforward to validate some theoretical property. The true proof of a type system is a pragmatic evaluation, however. 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 in the spirit of λ_{TS} makes up the core of this language, but we have also supplemented it with a number of important ingredients, both at the level of types and at the level of large-structure programming.

4.1 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 explicitly 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)))))
```

[5] The mechanised calculus does not currently handle **or**.

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 more conventional structures, Typed Scheme provides explicit recursive types though the programmer need not manually fold and unfold instances of these types.

For example, here is the type of a binary tree 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, and 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.

4.2 Type Inference

In order to further relieve the annotation burden on programmers, Typed Scheme provides two simple forms what has been called "local" type inference (Pierce and Turner 2000).[6] First, local non-recursive bindings do not require annotation. 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)))
```

since by examining the right-hand sides of the **let***, the typechecker can determine that both *x* and *y* should have type **Number**. Note that the inference mechanism does not take into account the uses of these variables, only their initializing expressions.

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 3)
  (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 any of the bindings can be treated non-recursively. If so, the above inference method is applied.

The second form of inference allows the type arguments to polymorphic functions to be omitted. For example, the following use of *map* does not require explicit annotation.

```
(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

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

(**Number** → **Number**) and (**Listof Number**), as well as the operator, here (**All** (*a b*) ((*a* → *b*) (**Listof** *a*) → (**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 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** → **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*.

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.

4.3 Adapting Scheme Features

Scheme in general, and PLT Scheme in particular, comes with numerous constructs that need explicit support from the type system:

- 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 definitions introduces constructors, predicates, field selectors, and field mutators. Typed Scheme includes a matching **define-typed-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-typed-struct** *A* ([*x* **:** **Number**] [*y* **:** **String**]))

  Unsurprisingly, all fields have type annotations.

  The **define-typed-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-typed-struct** definition facility can also automatically introduce recursive types, similar to those introduced via ML's datatype construct.

  The PLT Scheme system also allows programmers to 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 strucuture. 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.

- 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 is roughly speaking numeric and dynamic overloading. Fortunately, these constructs do not present new challenges to type system designers. A careful adaptation of the solutions employed for mainstream languages such as Java and C# suffices.

- 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 require 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.

- 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 argument 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)).

4.4 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, the programmer knows that every element of the resulting list answers **true** to the supplied predicate. 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*) → (**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 its latent predicate becomes the type of the resulting elements. In the conventional type world, this effect can only be achieved with dependent types.

For an example, consider the following expression:

```
(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. 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* → ⊥) → *a*) → *a*))

where ⊥ 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.

4.5 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 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. No declaration of the types of provided identifiers is required. In the example in figure 8, the type *LoN* and the function *sum* are provided by module *m1* and can therefore be used in module *m2*.

```
(module m1 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))))))

(module m2 typed-scheme
  (require m1)
  (define: l : LoN (list 1 2 3 4 5))
  (display (sum l)))
```

Figure 8. A Multi-Module Typed Scheme Program

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.

4.6 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 mzscheme add1 (Number → Number))
```

imports the *add1* function from the *mzscheme* library, with the given type. The **require/typed** facility expands into contracts, which are enforced as values cross module boundaries (Findler and Felleisen 2002).

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.

In support of such situations, Typed Scheme includes support for 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/opaque-type (lib "xml.ss" "xml") Doc document?)
```

It imports the *document?* function from the (**lib** "xml.ss" "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, other considerations come into play. Again, the typed code must be protected (Tobin-Hochstadt and Felleisen 2006), but we already know the necessary types. Therefore, in untyped contexts, typed exports 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.

5. Implementation

We have implemented Typed Scheme as a language for the DrScheme programming environment (Findler et al. 2002). It is available from the PLaneT software repository (Matthews 2006). [7]

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 sucessfully, 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.

5.1 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 AST 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 type-check the program and signal an error at macro-expansion time if it is ill-typed.

5.2 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 MzScheme core forms, of which there are approximately a dozen.

5.3 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

[7] The implementation consists of approximately 6000 lines of code and 2800 lines of tests.

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 MzScheme identifiers, which maintain information on which module they were defined in. Therefore, no special effort is required to use one typed module from another and standard DrScheme (Findler et al. 2002) tools for presenting binding structure work properly.

5.4 Limitations

Our implementation has two significant limitations at present. First, we are unable to dynamically enforce polymorphic types using the PLT contract system. Therefore, programmers cannot require polymorphically-typed functions from untyped libraries as such (only in instantiated form). We plan to address this difficulty by providing a comprehensive standard library of typed functions, based upon the PLT Scheme standard library.

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 radically alter the binding structure of the program. In order to typecheck them, our system would need to be extended to either infer this type structure from the results of macro expansion, or to understand units and classes natively. Since these macros are widely used by PLT Scheme programmers, we plan to investigate both possibilities.

6. 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 smaller 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 *The Little Schemer* (Friedman and Felleisen 1997) and *The Seasoned Schemer* (Friedman and Felleisen 1996) in Typed Scheme. 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.

```
(define: (play-one-turn [player : Player]
                        [deck : Cards]
                        [stck : Cards]
                        [fst:discs : Hand])
  : (values Boolean RCard Hand Attacks From)
  (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))
```

Figure 9. A Excerpt from the Squadron Scramble Game

Applications Finally, a student 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.[8] 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 9. 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 their body of the individual definitions.

Structure types are used extensively in this example, as they are 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 that the typed portion uses via **require/typed**.

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

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 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.

7. 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.

7.1 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 such 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 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 type 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) 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 when compared to soft typing systems. 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 simply do not support type abstractions. 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—inspired our generalization. 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, and extended it to account for the use of predicates and to distinguish different occurences of variables based on these predicates, as occurrence typing does.

7.2 Gradual Typing

Recently, proposals for what is called "gradual typing" have become popular (Siek and Taha 2006; Herman et al. 2007). This work also intends to integrate typed and untyped programs, but at a much finer granularity than we present. So far, this has prevented the proof of a type soundness theorem for such calculi that properly assign blame for failures to the typed and untyped pieces of a system.[9] In contrast, our earlier work on Typed Scheme (Tobin-Hochstadt and Felleisen 2006) provides such a soundness theorem, which we believe scales to full Typed Scheme and PLT Scheme.

The gradual typing proposals have also failed describe the embedding of such systems into realistic type systems that are suitable for writing significant applications, whereas Typed Scheme has been used for the porting of thousands of lines of code.

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] Wadler and Findler (2007)'s formulation eliminates this objection.

7.3 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 were studied 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 was inspired by prior work on effect systems (Gifford et al. 1987). Typing variables differently in different portions of a program was discussed by Crary et al. (1998). However, occurrence typing as presented here has not been previously considered.

7.4 Dependent Types

Some features similar to those we describe have appeared in the dependent type literature. 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 specified the variable to be refined, and that variable was 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 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.

7.5 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 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 from ours, and does not describe a mechanism for integrating with untyped code.

8. Conclusion

Migrating programs from untyped language to typed ones 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, some used widely, some just 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.

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 6 illustrative of existing code and believe that their success is a good predictor for future experiments. We plan on porting 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, are available from the Typed Scheme web page:

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

Acknowledgments

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 the anonymous reviewers for their comments.

References

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

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

Gilad Bracha. Pluggable type systems. In *Revival of Dynamic Languages Workshop at OOPSLA 2004*, October 2004.

Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.

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

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

Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 301–312, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-024-4. doi: http://doi.acm.org/10.1145/289423.289459.

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

R. Kent Dybvig and Robert Hieb. A new approach to procedures with variable arity. *Lisp and Symbolic Computation: An International Journal*, 3 (3), 1990.

Hsianlin Dzeng and Christopher T. Haynes. Type reconstruction for variable-arity procedures. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and Functional Programming*, pages 239–249, New York, NY, USA, 1994. ACM Press.

ECMA International. ECMAScript Edition 4 group wiki, 2007.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. MIT Press, 2001. URL `http://www.htdp.org/`.

Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, 2002.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen.

DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

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

Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *PLDI '96: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

Matthew Flatt. Composable and compilable macros: You want it *when?* In *ICFP '02: ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *PLDI '98: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems (APLAS) 2006*, pages 270–289, November 2006.

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

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

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

J. Gosling, B. Joy, G. L.Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Welsley, 2005.

Timothy G. Griffin. The formulae-as-types notion of control. In *POPL '90: Conf. Record 17th Annual ACM Symp. on Principles of Programming Languages*, Jan 1990.

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

Christopher T. Haynes. Infer: A statically-typed dialect of scheme. Technical Report 367, Indiana University, 1995.

Nevin Heintze. Set based analysis of ML programs. In *LFP '94: ACM Symposium on Lisp and Functional Programming*, pages 306–317, 1994.

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

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

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

Jacob Matthews. Component deployment with PLaneT: You want it *where?* In *Proceedings of the Seventh Workshop on Scheme and Functional Programming, University of Chicago, Technical Report TR-2006-06*, 2006.

Jacob Matthews, Robert Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *RTA 2004: Rewriting Techniques and Applications, 15th International Conference, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 2–16. Springer-Verlag, 2004.

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

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

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

Benjamin C. Pierce and David N. Turner. Local type inference. volume 22, pages 1–44, New York, NY, USA, 2000. ACM.

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

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

Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991.

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

Jens Axel Søgaard. Galore, 2006. http://planet.plt-scheme.org/.

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

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

Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 964–974, New York, NY, USA, 2006. ACM Press.

Christian Urban and Christine Tasson. Nominal Reasoning Techniques in Isabelle/HOL. In *Proceedings of the 20th Conference on Automated Deduction (CADE 2005)*, volume 3632 of *Lecture Notes in Artificial Intelligence*. Springer, 2005.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 251–262, New York, NY, USA, 2006. ACM Press.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the Eighth Workshop on Scheme and Functional Programming*, 2007.

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

A. Wright and B. Duba. Pattern matching for Scheme. Technical report, Rice University, May 1995.

Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.

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