Typed Scheme:

From Scripts to Programs

Sam Tobin-Hochstadt

Draft date: October 28, 2009

Abstract

The field of programming languages has recently experienced a renaissance, especially in the field of untyped scripting languages. But when scripts written in untyped languages grow into large programs, they may also become difficult to maintain. To improve the maintainability of programs in untyped languages, I propose porting portions into typed sister languages. To demonstrate the feasibility of this approach, I have developed Typed Scheme, a typed variant of PLT Scheme. Typed Scheme provides smooth and sound interoperability with untyped PLT Scheme; it also features a novel type system that supports idiomatic Scheme programming, so that the porting process is relatively straightforward. I have validated the effectiveness of Typed Scheme by porting thousands of lines of untyped PLT Scheme code.

Why, anybody can have a brain. That's a very mediocre commodity. Every pusillanimous creature that crawls on the Earth or slinks through slimy seas has a brain. Back where I come from, we have universities, seats of great learning, where men go to become great thinkers. And when they come out, they think deep thoughts and with no more brains than you have. But they have one thing you haven't got: a diploma.

— The Wizard of Oz

Acknowledgments

My original debt is to James D. Jungbauer Jr. and James E. Hamblin, who first taught me Scheme at the Johns Hopkins University Center for Talented Youth. But to the extent that I am more than merely someone who knows some Scheme, the credit is due to Matthias Felleisen, the advisor that I needed, and that made me the researcher that I am today. Matthias has been wise and impetuous, understanding and demanding, and has consistently expected and obtained better from me than I myself did.

In between, I have been assisted by numerous people. Robby Findler and Laszlo Babai encouraged me to go to Northeastern. Once there, my colleagues have been wonderful resources: Dave Herman, Carl Eastlund, Felix Klock, Jesse Tov, Christos Dimoulas, David van Horn, Aaron Turon, Dan Brown and many others have pushed me to make my ideas clearer and my thinking more precise. Ryan Culpepper, Stevie Strickland and Ivan Gazeau have been productive collaborators as well as fellow students. My colleagues at Sun Labs: Eric Allen, David Chase, Victor Luchancgo, Christine Flood, and Jan-Willem Maessen helped give me a wider perspective and fresh ideas. I have also been fortunate to work with Robby Finder and Matthew Flatt, both as co-authors and as co-developers. My committee, Mitch Wand, Olin Shivers and Guy Steele, have helped improve this dissertation.

All my life, my parents have provided models of what it means to be an academic, and they have encouraged me at every step as I have pursued that life. One day I hope to live up to their example.

Finally, my greatest debt is to my wife, Katie Edmonds, for which no thanks are enough.

Contents

List of Figures

CHAPTER 1

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 [Lerdorf, Tatroe, and MacIntyre 2006; ECMA 1999; Matsumoto 2001; Ousterhout 1994; Van Rossum and Drake 2009; Wall, Christiansen, and Schwartz 1996]. Due to their popularity, programmers often create scripts that then grow into large applications.

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; Herman and Flanagan 2007] are implicitly acknowledging this problem with the addition of Common Lisp-style [Steele Jr. 1990] 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.

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

This brings me to my thesis:

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

In support of this thesis, I have developed Typed Scheme, a typed sister language of PLT Scheme [Tobin-Hochstadt and Felleisen 2006; Culpepper, Tobin-Hochstadt, and Flatt 2007; Tobin-Hochstadt and Felleisen 2008; Strickland, Tobin-Hochstadt, and Felleisen 2009]. The choice of PLT Scheme (a dialect of Scheme [Sperber, Dybvig, Flatt, Van Straaten, Findler, and Matthews 2009; Sussman and Steele Jr. 1975]) is important 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 typechecking a language with a user-defined set of syntactic forms. This difficulty can be mostly overcome 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, Typed Scheme comes with a novel type system. This type system combines the concept of *occurrence typing*[1] with additional novel features for handling variable-arity polymorphism and refinement types.

The Structure of this Thesis

In this thesis, I describe the Typed Scheme experiment both formally and informally, as well as its relation to past and present work by others. In chapter 2, I give an example-driven overview of the major features of Typed Scheme, and then describe at a high level how the typechecker is able to handle various representative examples. Chapter 3 describes five key choices made in the design of Typed Scheme, as well as the rationale for each. In chapter 4, I describe the extensive previous work on which Typed Scheme is based. This includes both previous type systems for Scheme and LISP and prior work on interlanguage interoperability, as well as key language technologies on which the design of Typed Scheme is based. In the subsequent five chapters, I describe the formal systems underlying the design of Typed Scheme:

- Chapter 5 describes how software contracts mediate between typed and untyped code, leading to a soundness proof for a mixed system. This work previously appeared in the Dynamic Languages Symposium at OOPSLA 2006 [Tobin-Hochstadt and Felleisen 2006].

- Chapter 6 describes occurrence typing, the primary novel feature of the Typed Scheme type system. This work previously appeared at POPL

[1] The term “occurrence typing” for this idea in the context of a type system was also coined independently of my work by Komondoor, Ramalingam, Chandra, and Field [2005].

2008 [Tobin-Hochstadt and Felleisen 2008], and is under submission in its present form.

- Chapter 7 presents two extensions to occurrence typing, allowing it to faithfully capture more Scheme idioms. This work is currently under submission.

- Chapter 8 describes how refinement types can be added to an occurrence typing system in a lightweight yet expressive fashion, and presents an extended example. This work was discussed at the Workshop on Script to Program Evolution, 2009 [Tobin-Hochstadt and Findler 2009] and is under submission in its current form.

- Chapter 9 describes the many ways that Scheme programmers use variable-arity functions, and the technique of *variable-arity polymorphism* for typechecking them. This work previously appeared at the European Symposium on Programming, 2009 [Strickland et al. 2009].

Building upon the PLT Scheme module and macro system, Typed Scheme is implemented entirely as a library. This implementation, and the techniques on which it is based, are described in chapter 10. The implementation was previously presented at the Workshop on Scheme and Functional Programming, 2007 [Culpepper et al. 2007]. Using this implementation, a number of programmers have ported existing PLT Scheme code to Typed Scheme. This experience, along with quantitative measurements of the changes required, is described in chapter 11.

Chapter 12 compares Typed Scheme to other related work and explains additional connections to the literature. I conclude in chapter 13.

CHAPTER 2

Typed Scheme through Examples

In this chapter, I describe the general outlines of Typed Scheme via a series of examples. This includes simple typed programs, the integration between typed and untyped code, occurrence typing, and variable-arity polymorphism. In sections 2.1-2.6 these are described informally, and section 2.7 provides more detail to give an overview of how the system works.

2.1 Simple Typed Scheme

The obligatory *Hello, World* Typed Scheme program is:

Example 1

```
#lang typed-scheme
(display "Hello, World!\n")
```

This example demonstrates several salient features of Typed Scheme. First, a Typed Scheme program is a PLT Scheme module [Flatt 2002], which begins with the token **#lang**. Second, the *typed-scheme* token after **#lang** specifies that the module is written in the Typed Scheme language. The program is otherwise identical to the corresponding untyped Scheme program.

When more complex programs are required, the type system requires slightly more help from the programmer:

Example 2

```
#lang typed-scheme
(: collatz (Number → Number))
(define (collatz n)
  (if (even? n)
      (collatz (/ n 2))
      (collatz (add1 (∗ 3 n)))))
(collatz 17)
```

In this example, we define the function *collatz*, and provide an annotation for its input and output types. These annotations are required for every top-level definition.

Typed Scheme programs can also define and manipulate new data structures:

Example 3

```
#lang typed-scheme
(define-struct: person ([first : String] [last : String]))
(: greeting (person → String))
(define (greeting n)
  (format "~a ~a" (person-first n) (person-last n)))
(greeting (make-person "Bob" "Smith"))
```

The **define-struct:** form defines a new structure, with selectors, predicate, and constructor. The name of the structure (here *person*) is used as the name of the type. The fields must be given types by the programmer, but the selectors (*person-first* and *person-last*), predicate (*person?*) and constructors (*make-person*) are given types automatically from the field types.

2.2 Polymorphism and Local Type Inference

Typed Scheme supports first-class polymorphic functions.[1] For example, *list-ref* has the type (∀ (α) ((**Listof** α) *Integer* → α)). It can be defined in Typed Scheme as follows:

[1] Such functions are not always parametric, because occurrence typing can be used to examine the arguments.

Example 4

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

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

2.3 Integration with Untyped Scheme

Typed Scheme integrates smoothly and safely with existing untyped Scheme code. This mean both that Typed Scheme modules can depend on untyped Scheme modules, and that untyped Scheme modules can depend on Typed Scheme modules. Further, the untyped code can never violate the invariants of the Typed Scheme type system. This allows free choice of which modules are refactored and ported to Typed Scheme. The simplest example of typed/untyped integration merely uses a value from Typed Scheme in untyped code.

Example 5

```
#lang typed-scheme  ;; module m1
(: x Number)
(define x 42)
(provide x)

#lang scheme
(require m1)
(add1 x)
```

Since the value is being used by untyped code, no new types must be specified. In contrast, when importing values *from* untyped code, the type must be specified:

Example 6

```
#lang scheme ;; module m2
(define x 42)
(provide x)

#lang typed-scheme
(require/typed m2 [x Number])
(add1 x)
```

The **require/typed** form allows the programmer to specify the type of an imported binding (here *x*). The type is dynamically turned into a runtime contract [Meyer 1992], which ensures that the value is appropriate to the type, and raises an error if not. In the case of an error, for example if the value of *x* was `"42"`, *blame* is assigned to the series of modules involved. The Typed Scheme type system ensures (see chapter 5) that the typed portion of the program is never blamed for such runtime contract errors—instead, the blame falls on one of the untyped modules.

The contract system also ensures appropriate blame for higher-order values [Findler and Felleisen 2002]. Here, errors are possible in both directions. Several examples are given in figure 2.1. In example 7, the untyped module uses the *add5* procedure incorrectly, resulting in a runtime contract error blaming the untyped module. In example 8, an incorrect use of the *add5* procedure would result in a static type error. Finally, in example 9, the implementation of the untyped *add5* procedure is incorrect, and thus a runtime contract error is signaled, again blaming the untyped module.

2.4 Occurrence Typing

Here is the simplest example of occurrence typing:[2]

Example 11

```
... (if (number? x) (add1 x) 0) ...
```

[2] In the remainder of this chapter, module declarations will be omitted where unnecessary.

Example 7

```
#lang typed-scheme ;; module ho1
(: add5 (Number → Number))
(define (add5 n) (+ 5 n))
(provide add5)

#lang scheme
(require ho1)
(add5 7)
(add5 "seven") ;; contract error
```

Example 8

```
#lang scheme ;; module ho2
(define (add5 n) (+ 5 n))
(provide add5)

#lang typed-scheme
(require ho2 [add5 (Number → Number)])
(add5 7)
;;(add5 ”seven”) - static type error
```

Example 9

```
#lang scheme ;; module ho3
(define (add5 n) (number->string (+ 5 n)))
(provide add5)

#lang typed-scheme
(require ho3 [add5 (Number → Number)])
(add5 7) ;; contract error
```

Example 10

```
#lang typed-scheme ;; module ho4
(: add-blaster (Number → (Number → Number)))
(define ((add-blaster x) y)
  (+ x y))

#lang scheme
(require ho4)
((add-blaster 1) "wrong") ;; contract error
```

Figure 2.1: Higher-order Contract Examples

Regardless of the value of *x*, this program fragment always produces a number. Thus, our type system should accept this fragment, regardless of the type assigned to *x*, even if it is a type such as **String** or **Any**, which are not legitimate argument types for *add1*.

The key to typing example 11 is to assign the second occurrence of *x* a different, more precise, type than it has in the outer context. Fortunately, we know that for any value of type **Number**, *number?* returns #t (otherwise, it returns #f). Therefore, it is safe to use **Number** as the type of *x* in the then branch.

Example 12

```
(define f
  (λ ([x : (∪ String Number)])
    (if (number? x) (add1 x) (string-length x))))
```

The function *f* in example 12 always produces a number. If (*number? x*) produces #t, *x* is an appropriate input to *add1*. If it produces #f, *x* must be a **String** by process of elimination, and it is therefore an acceptable input to *string-length*. Handling this program means that the type system must take into account not only the consequences when predicates hold, but also when they do not.

2.4.1 More Complex Tests

Of course, simple applications of predicates such as (*number? x*) are not the only kind of test that Scheme programmers write. For example, it is possible to use logical connectives to combine the results of predicates:

Example 13

```
... (if (or (number? x) (string? x))
        (f x) ;; f from example 12
        0) ...
```

For the fragment in example 13 to typecheck, the type system must recognize that the expression (**or** (*number? x*) (*string? x*)) ensures that *x* has type (∪ **String Number**) in the then branch, the domain of *f* from above.

For **and**, there is no such neat connection:

Example 14

```
... (if (and (number? x) (string? y))
        (+ x (string-length y))
        0) ...
```

Example 14 is perfectly safe, regardless of the types of *x* and *y*.

Example 15

```
;; x is either a Number or a String
... (if (and (number? x) (string? y))
        (+ x (string-length y))
        (string-length x)) ...
```

In example 15, the programmer falsely assumed *x* to be a **String** when the test fails. But the test may produce #f because *x* is actually a **String**, in which case the program would succeed, or because *y* is not a **String**, but *x* *is* a number, which would cause (*string-length x*) to fail. In general, when a conjunction is false, we do not know which conjunct is false.

2.4.2 Abstraction over Predicates

So far, we have seen how programmers can use predefined predicates. It is important, however, that programmers can also abstract over existing predicates and create new ones.

Example 16

```
(define strnum?
  (λ ([x : Any])
    (or (string? x) (number? x))))
```

Taking our previous example of a test for the type (∪ **String Number**), we can create the function *strnum?*, which behaves as a predicate for that type. This means that the type system must be able to represent the fact that *strnum?* is a predicate for this type, so that it can exploit it for conditionals.

2.4.3 Variables, Predicates and Selectors

An important feature of Scheme that Typed Scheme must also handle is the ability to use arbitrary non-#f values as true and to use #f as a marker for missing results, analogous to ML's `NONE`.[3]

Example 17

```
... (let ([x (member v l)])
      (if x
          — compute with x —
          (error 'fail))) ...
```

Example 17 represents the essence of a common Scheme idiom. The *member* procedure yields either the desired result, if *v* is found in *l* or #f if *v* is not found. Therefore, in the then branch we know *x* is the desired result, otherwise the else branch is taken.

All of the tests thus far have involved variables such as *x* and *y*. It is also useful to test arbitrary expressions. For example, we can test that the *car* of a pair is a **Number** with the expression (*number?* (*car p*)). Integrating this form of reasoning into the type system requires further modifications, as we will see.

Example 18

```
... (if (number? (car p)) ;; p : (Pair Any Any)
        (add1 (car p))
        7) ...
```

If *p* has the type (**Pair Any Any**), then example 18 should produce a number.[4] Of course, simply accommodating repeated applications of *car* is insufficient for real programs. Instead, the relevant portions of the type of *p* must be refined in the then and else branches of the if. In example 19, we assume that *g* has argument type (**Pair Number Number**):

[3] Like other dynamically typed languages, Scheme treats all values that are not #f as true.

[4] This relies on the recent change to PLT Scheme to make pairs immutable by default.

Example 19

```
... (if (and (number? (car p)) (number? (cdr p)))
        (g p)
        'nope) ...
```

Thus, the test expression must refine the type of *p* to (**Pair Number Number**), which is the expected result of the conjunction of tests on the *car* and *cdr*.

As example 20 shows, programmers can abstract the use of predicates and selectors together.

Example 20

```
(define carnum?
  (λ ([x : (Pair Any Any)]) (number? (car x))))
```

The *carnum?* predicate tests if the *car* of its argument is a **Number**.

2.4.4 Reasoning Logically

Of course, we *do* learn something when conjunctions such as those in examples 14 and 15 are false. When a conjunction is false, we know that one of the conjuncts is false, and thus when all but one are true, the remaining one must be false. In Scheme form, this reasoning principle is found quite often in nested conditionals.

Example 21

```
... (cond [(and (number? x) (string? y))  — 1 —]
          [(number? x)                    — 2 —]
          [else                           — 3 —]) ...
```

This program represents a common idiom.[5] In clause 1, we obviously know that *x* is a **Number** and *y* is a **String**. In clause 2, *x* is again a **Number**. But we also know that *y* cannot be a **String**. To make sense of the type discipline employed in such programs, the Typed Scheme type system must be able to follow this reasoning.

[5] The introductory textbook *How to Design Programs* [Felleisen, Findler, Flatt, and Krishnamurthi 2001] devotes an entire section to this idiom.

2.4.5 Putting it all Together

Finally, we can combine all of these features into a single example that demonstrates all aspects of occurrence typing.

Example 22

```
(λ ([input : (∪ Number String)] [extra : Any])
  (cond
    [(and (number? input) (number? (car extra)))
     (+ input (car extra))]
    [(number? (car extra))
     (+ (string→number input) (car extra))]
    [else 0]))
```

2.5 Variable-arity Functions

PLT Scheme allows programmers to define functions that do not take a fixed number of arguments. Some functions can accept any one of a set of numbers of arguments, or a variable number. To accommodate these programming patterns, Typed Scheme supports a variety of forms of variable-arity function definitions.

2.5.1 Uniform Variable-Arity Functions

Uniform variable-arity functions are those that expect their rest parameter to be a homogeneous list. Consider the following three examples of type signatures:

```
(: + (Integer* → Integer))
(: string-append (String* → String))
(: list (∀ (α) (α* → (Listof α))))
```

The syntax *Type** indicates that 0 or more arguments of type *Type* are required.

Here is a definition of variable-arity + in Typed Scheme:

Example 23

```
(: + (Integer* → Integer))

;; assumes binary-+, a binary addition operator
(define (+ . xs)
  (if (null? xs) 0 (binary-+ (car xs) (apply + (cdr xs)))))
```

2.5.2 Non-uniform Variable-Arity

Not all variable-arity functions assume that their rest parameter is a homogeneous list of values, which allows simple typechecking. Typechecking heterogeneous rest parameters requires analyzing other relationships between types. For example, the length of the list assigned to the rest parameter may be connected to the types of other parameters or the returned value.

For example, Scheme's *map* function maps a n-ary function over n lists, unlike its counter-parts in ML or Haskell. When *map* receives a function *f* and n lists, it expects *f* to accept n arguments. Also, the type of the kth function parameter must match the element type of the kth list.

The following example is taken from the PLT Scheme code base:

Example 24

```
;; implements a wrapper that prints f's arguments
(: verbose (∀ (β α ...) ((α ... → β) → (α ... → β))))
(define (verbose f)
  (if quiet? f (λ a (printf "xform-cpp: ~a\n" a) (apply f a))))
```

The intent of the programmer is clear—the result of applying *verbose* to a function *f* should have the same type as *f* for *any* function type. Typed Scheme represents this by allowing *verbose* to be instantiated with a *sequence* of type arguments, one for each of the arguments to *f*.

Below are the types for some additional example functions:

Example 25

```
;; map is the standard Scheme map
(: map
   (∀ (γ α β ...)
      ((α β ... → γ) (Listof α) (Listof β) ... → (Listof γ))))
```

Example 26

```
;; map-with-funcs takes any number of functions,
;; and then an appropriate set of arguments, and then produces
;; the results of applying all the functions to the arguments
(: map-with-funcs
   (∀ (β α . . . ) ((α . . . → β)* → (α . . . → (Listof β)))))
```

When a variable-arity polymorphic type is instantiated, the dotted sequence is replaced with the provided sequence of type arguments. For *map-with-funcs*, this works as follows:

```
(inst map-with-funcs Number Integer Boolean String)
```

results in a value with the type:

```
((Integer Boolean String → Number)* →
(Integer Boolean String → (Listof Number)))
```

Typed Scheme also provides local inference of the appropriate type arguments for dotted polymorphic functions, so explicit type instantiation is rarely needed [Strickland, Tobin-Hochstadt, and Felleisen 2008].

A more substantial definition of a variable-arity function is *fold-left*.

Example 27

```
(: fold-left
   (∀ (γ α β . . . ) ((γ α β . . . → γ) γ (Listof α) (Listof β) . . . → γ)))
(define (fold-left f c as . bss)
  (if (or (null? as) (ormap null? bss))
      c
      (apply fold-left (apply f c (car as) (map car bss)) (cdr as)
             (map cdr bss))))
```

Its type shows that it accepts at least three arguments: a function *f*; an initial element *c*; and at least one list *as*. Optionally, *fold-left* consumes another sequence *bss* of lists. For this combination to work out, *f* must consume as many arguments as there are lists plus one; in addition, the types of these lists must match the types of *f*'s parameters because each list item becomes an argument.

2.6 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. In Typed Scheme, we can 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 member by using the predicate *even?*. For example, this example includes the *just-even* function, which produces solely even numbers, and the *halve* function, which consumes only even numbers.

Example 28

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

(: halve ((Refinement even?) → Number))
(define (halve n) (/ n 2))
```

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

2.7 How to Check the Examples

This section describes in more detail how Typed Scheme's type system handles some of the examples of chapter 2 in order to motivate and explain the basic ideas of the formal systems described in the subsequent chapters.

2.7.1 Simple Examples

In examples 1 and 2, typechecking is straightforward. Structures, as in example 3, require only slightly more effort—Typed Scheme must understand the relationships between the types of the fields and the type of the constructor, here *make-person*, in order to check the application of the constructor.

To facilitate easier programming, type arguments to polymorphic functions are automatically synthesized where possible. Argument type synthesis uses the local type inference algorithm of Pierce and Turner [2000]. It greatly facilitates the use of polymorphic functions and makes conversions from Scheme to Typed Scheme convenient, while dealing with the subtyping present in the rest of the type system in an elegant manner. Furthermore, it ensures that type inference errors are always locally confined, rendering them reasonably comprehensible to programmers.

2.7.2 Typed/Untyped Integration

The fundamental solution for sound integration between typed and untyped code is runtime contracts [Findler and Felleisen 2002]. In example 5, a contract checking that *x* is a number is automatically added to module *m1*, and used to protect *m1* against any malicious use of *x* by untyped code. A similar contract is automatically generated for example 6, checking that *m1*, the untyped module in this example, actually produces a number.

For higher-order types, higher-order contracts are generated. Here, protection in both directions is vital, as in examples 7 and 8.

In chapter 5, I describe these ideas formally, and prove that only untyped modules can receive blame for violating the automatically-generated contracts.

2.7.3 Occurrence Typing

For occurrence typing, we begin by returning to example 11:

(**if** (*number? x*) (*add1 x*) 0) ;; *x* : **Any**

In this example, we must propagate information from the test, (*number? x*), to the then branch. Therefore, in addition to determining that the test has the type **Boolean**, the typechecker also proves the proposition that "if the test evaluates to #t, then *x* is a number". We write the second part of this proposition as $\mathbf{N}_x$, and call it a *filter* (in the formal system, we abbreviate **Number** as **N**). We can combine *x* to **Any**, to get a new environment for checking the then branch where *x* has type **Number**. This new environment is used to check the then branch.

The type system must then compute this proposition (and thus the new type environment) from the expression (*number? x*). First, we need information in the type of *number?* that it is a predicate for the **Number** type. Second, we need information from the operand *x*, that testing it produces information about the variable *x*.

We accomplish our first goal with *latent* information on the function type of the *number?* procedure. Such latent information describes not only what sorts of values it accepts and produces, but also what filter it produces when applied. We take this notion of latent information from work on effect systems Lucassen and Gifford [1988] and say that *number?* has a *latent filter*.

We accomplish the second by adding an additional piece of information to the type and the propositions, describing which bit of the environment a given expression accesses. We refer to this information as the *object* of an expression, in this case *x*

Given these two pieces of information—the latent filter from *number?* and the object from *x*—we obtain the filter $\mathbf{N}_x$ as desired. This, in turn, is enough to typecheck the rest of the **if** expression.

2.7.3.1 Handling Complex Tests

In example 12, we have

(**if** (*number? x*) (*add1 x*) (*string-length x*))

where *x* has the type ($\bigcup$ **String Number**). To typecheck the else branch, we also need the information that *x* is not a **Number** i.e., that it is a **String**.

To accomplish this, the type checker must propagate the proposition "*x* is some element of its original type, but not a **Number**" from the test expression to the else branch. This proposition is written $\overline{\mathbf{N}}_x$, and it is straightforward to compute it from the latent filter of the operator and the object of the operand from the test.

With more complex tests, we must combine the filters of different subexpressions. If (**or** (*number? x*) (*string? x*)) is true, then the value of *x* must be a member of the type ($\bigcup$ **Number String**), and if it is false then *x* cannot be either of those types. Using our notation, these can be expressed with the filters $(\bigcup \mathbf{N}\,\mathbf{S})_x$ and $\overline{(\bigcup \mathbf{N}\,\mathbf{S})}_x$, which apply in the then and else branches, respectively.

For all of the examples we have seen so far, the filters for the then and else branches had a simple relationship. This is not always the case, however. Consider (**and** (*number? x*) (*string? y*)). If this expression evaluates to #t, then *x* must be a **Number** and *y* a **String**. We represent this new information with two filters that apply to the then branch, $\mathbf{N}_x$ and $\mathbf{S}_y$. If the test evaluates to #f, we do not know which of the two individual tests fails, so we can't create a new type environment for the else branch. Therefore, no filters apply to the else branch. This is why the first example using **and** (example 14) is safe, but the second (example 15) is not; the type of *x* cannot be refined appropriately.

2.7.3.2 Abstracting over Predicates

The next challenge is how to include filter information in function types for user-defined predicates (see example 16):

(λ ([*x* : **Any**]) (**or** (*string? x*) (*number? x*)))

We must first abstract the filter away from the choice of parameter, in this case *x*. Also, the resulting filter is not immediately available after evaluating the entire expression, because the evaluation of a λ-expression obviously provides no information before it is applied. Therefore, the information in the filter, once abstracted, forms the latent filter of the function. This function has the latent filter $(\bigcup \mathbf{S}\,\mathbf{N})$ when the result of applying the function is true, and $\overline{(\bigcup \mathbf{S}\,\mathbf{N})}$ when it is false, which are abstractions of the original filters for this disjunction. These latent filters implicitly refer to whatever the actual argument to the function might be.

Latent filters are also the underlying mechanism for specifying the types of built-in predicates such as *string?* and *number?*. Examples of such latent filters are shown in figure 6.6 on page 75.

2.7.4 Variables, Predicates and Selectors

Scheme programmers often use predicates on selector expressions such as (*car p*). Our type system represents such expressions as complex *objects*. For example, (*number?* (*car p*)) involves a predicate with a latent filter applied to an expression whose object indicates that it accesses the *car* field of *p*. We write this object as $\mathbf{car}(p)$. Thus, the entire expression has filter $\mathbf{N}_{\mathbf{car}(p)}$ (for the then branch) and $\overline{\mathbf{N}_{\mathbf{car}(p)}}$ (for the else branch).

It is also important to abstract over selector expressions. Therefore, in addition to having latent filters, function types also have *latent objects*, specifying their access pattern on their arguments. So, just as the abstraction of the filter $\mathbf{N}_x$ is the latent filter $\mathbf{N}$, the abstraction of the object $\mathbf{car}(p)$ is the latent object **car**. We refer to a sequence of selectors (here $\mathbf{car}$) as a *path*.

Of course, a filter involving a non-trivial object can also be abstracted into a latent filter, turning $\mathbf{N}_{\mathbf{car}(p)}$ into $\mathbf{N}_{\mathbf{car}}$. In all of these cases, the key element of abstraction is removing the parameter.

Finally, the Scheme's notion of truth can also be expressed using the language of filters. When an expression such as *x* is used as a test, in the then branch *x* cannot be #f, and in the else branch it must be #f. This can be expressed with the filter $\overline{\#\mathbf{f}}_x$ for the then branch and $\#\mathbf{f}_x$ for the else branch.

2.7.5 Reasoning Logically

Let us now revisit our assumptions concerning expressions such as (**and** (*number? x*) (*string? y*)). If this expression evaluates to #f, we do know something about the values of *x* and *y*. In particular, if we think of the filter for the original expression as $\mathbf{N}_x \wedge \mathbf{S}_y$, then its negation is, by simple classical logic, $\mathbf{N}_x \supset \neg\,\mathbf{S}_y$. We can reinterpret $\neg\,\mathbf{S}_y$ as $\overline{\mathbf{S}}_y$, giving us $\mathbf{N}_x \supset \overline{\mathbf{S}}_y$, meaning that if *x* is a number, *y* cannot be a string. If we take this proposition to be a filter, then it can be one of the propositions that we learn upon evaluation of the original expression, in the case that it evaluates to #f. If we were to further learn $\mathbf{N}_x$, as we might from a test like (*number? x*), we would also learn that $\overline{\mathbf{S}}_y$. This allows the type system to reason in the same way that the programmer does and to keep track of (some of) the myriad facts available for deducing the partial correctness of a program fragment.

2.7.5.1 The Form of the Type System

Our examples suggest that four elements are central to the operation of the Typed Scheme type system:

- Typechecking an expression computes two sets of *filters*, which are propositions that hold when the expression evaluates to true or false, respectively.

- Typechecking an expression also determines an *object* of inquiry, describing the particular piece of the environment pointed to by that expression. This piece of the environment may also be a portion of a larger data structure, accessed via a *path*.

- Filters can be combined via logical connectives to form more complex filters.

- *Latent* filters and objects abstract over a function's parameters. They are included in function types.

In chapters 6 and 7, I translate these ideas into a calculus, λ_{TS} , and its type system.

2.7.6 Variable-arity Polymorphism

To typecheck variable-arity functions, we consider uniform variable-arity functions separately from non-uniform variable-arity.

2.7.6.1 Uniform Variable-arity

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

Returning to example 23, typing this definition is straightforward. The type assigned to the rest parameter of starred pre-type τ^* in the body of the function is (**Listof** τ), a pre-existing type in Typed Scheme.

2.7.6.2 Non-uniform Variable-arity

In contrast, consider the types of *map* and *map-with-funcs* from examples 25 and 26. Our first key innovation is the possibility to attach ... to the last type variable in the binding position of a $\forall$ type constructor. Such type variables are dubbed *dotted type variables*. Dotted type variables signal that this polymorphic type can be instantiated with an arbitrary number of types.

Next, the body of $\forall$ types with dotted type variables may contain expressions of the form $\tau\ \ldots_{\alpha}$ for some type τ and a dotted type variable α. These are *dotted pre-types*; they classify non-uniform rest parameters just like starred pre-types classify uniform rest parameters. A dotted pre-type has two parts: the base τ and the bound α. Only dotted type variables can be used as the bound of a dotted pre-type. Since $\forall$-types are nestable and thus multiple dotted type variables may be in scope, dotted pre-types must specify the bound.

Finally, in example 27, we have a substantial definition of a variable-arity polymorphic function. Its type shows that it accepts at least three arguments: a function *f*; an initial element *c*; and at least one list *as*. Optionally, *fold-left* consumes another sequence *bss* of lists. For this combination to work out, *f* must consume as many arguments as there are lists plus one; in addition, the types of these lists must match the types of *f*'s parameters because each list item becomes an argument.

Beyond this, the example illustrates that the rest parameter is treated as if it were a place-holder for a plain list parameter. In this particular case, *bss* is thrice subjected to *map*-style processing. In general, variable-arity functions should be free to process their rest parameters with existing list-processing functions.

The challenge is to assign types to such expressions. On the one hand, list-processing functions expect lists, but the rest parameter has a dotted pre-type. On the other hand, the result of list-processing a rest parameter may flow again into a rest-argument position. While the first obstacle is simple to overcome with a conversion from dotted pre-types to list types, the second one is onerous. Since list-processing functions do not return dotted pre-types but list types, we cannot possibly expect that such list types come with enough information for an automatic conversion.

Thus we use special type rules for the list processing of rest parameters with *map*, *andmap*, and *ormap*. Consider *map*, which returns a list of the same length as the given one and whose component types are in a predictable order. If *xs* is classified by the dotted pre-type $\tau \ldots_{\alpha}$, and *f* has type $(\tau \rightarrow \sigma)$, we classify (*map f xs*) with the dotted pre-type $\sigma \ldots_{\alpha}$. Thus, in the definition of *fold-left* (*map car bss*) is classified as the dotted pre-type $\beta \ldots_{\beta}$ because *car* is instantiated at $((\textbf{Listof}\ \beta) \rightarrow \beta)$ and *bss* is classified as the dotted pre-type $(\textbf{Listof}\ \beta) \ldots_{\beta}$.

One way to use such processed rest parameters is in conjunction with *apply*. Specifically, if *apply* is passed a variable-arity function *f*, then its final argument *l*, which must be a list, must match up with the rest parameter of *f*. If the function is a uniform variable-arity procedure and the final argument is a list, typing the use of *apply* is straightforward. If it is a *non-uniform* variable-arity function, the number and types of parameters must match the elements and types of *l*.

Here is an illustrative example from the definition of *fold-left*:

(*apply f c* (*car as*) (*map car bss*))

By the type of *fold-left*, *f* has type $(\gamma\ \alpha\ \beta\ \ldots_{\beta} \rightarrow \gamma)$. The types of *c* and (*car as*) match the types of the initial parameters to *f*. Since the *map* application has dotted pre-type (**Listof** β) $\ldots_{\beta}$ and since the rest parameter position of *f* is bounded by β, we are guaranteed that the length of the list produced by (*map car bss*) matches *f*'s expectations about its rest argument. In short, we use the type system to show that we cannot have an arity mismatch, even in the case of *apply*.

In chapter 9, I describe how to make these ideas precise, and prove that such arity mismatches are statically rejected.[6]

[6] Of course, not all dynamic errors raised by variable-arity functions are statically detected—when *map* is applied to lists of varying length, the error is only reported dynamically.

CHAPTER 3

Design Choices

In this chapter, I describe several key design decisions and their rationale.

3.1 Reject Ill-typed Programs

Most previous approaches to static checking for untyped languages have not rejected programs. Instead, they validated all programs, and either issued warnings or added new runtime checks to programs that could not be proved safe. In contrast, Typed Scheme issues an error at compile-time for programs that do not typecheck.

This simplifies the design and implementation of Typed Scheme in a number of ways. First, Typed Scheme does not have to transform the program to insert dynamic checks. Second, it allows the Typed Scheme type system to be more expressive than the dynamic checks that can be easily implemented with the PLT Scheme contract system. For example, dynamic checks for types with filters is an open problem. Third, it makes the programming model much simpler for users, by comparison to systems which automatically insert dynamic checks. Almost all users of Typed Scheme have used typed languages before, and thus no explanation is required, and the program's behavior doesn't change based on the types. Fourth, and most importantly, all the errors detected by the type system are reported, meaning that the the programmer can rely on types in the program for reasoning about the program. Even in sound external tools, the program can still be run when ill-typed, whereas Typed Scheme statically rejects such programs as a part of the PLT Scheme execution process.

3.2 Explicit Typing

With few exceptions, every bound variable in Typed Scheme must be annotated with its type. This makes programming more inconvenient by comparison either to idiomatic untyped Scheme programming, or to other typed functional languages, especially when anonymous functions are used. But it has a number of key advantages.

First, complete inference for the Typed Scheme type system is very complex, since it combines polymorphism and subtyping. Second, type inference is well-known to cause hard-to-decipher type errors with non-local behavior. Third, experience with systems that performed complete type inference for untyped Scheme code suggested that such systems are extremely brittle, with minor code changes radically changing the inference results. Fourth, type inference allows programmers to omit type annotations in many cases where the type would serve as valuable statically-checked documentation.

Instead, Typed Scheme provides local type inference in certain cases: for type arguments to polymorphic functions, for non-recursive local bindings, and for some λ-bound variables. This local inference removes much of the most significant burden from the programmer, while simplifying the implementation and preserving valuable checked documentation.

3.3 Module-level Granularity

When using Typed Scheme, a module must be either wholly typed, or wholly untyped. In contrast, many other approaches to “gradual typing” have allowed mixing typed and untyped code at the expression level.

Both approaches obviously have the same expressiveness, since any piece of code can be factored out into its own module. However, the module-by-module approach has several advantages. First, modules are a natural level to think about the organization of code. Second, since the contracts generated at typed/untyped boundaries have runtime cost, module level granularity makes reasoning about the performance impact of typed/untyped boundaries easier. Third, the PLT module system allows individual modules to specify their language, making the module level the most natural granularity for integration with the rest of PLT.

3.4 Pre-expanding Macros

In Typed Scheme, all macros in the source program are expanded before typechecking.[1] This has several vital advantages but also disadvantages that must be overcome. First, since untyped Scheme code may use arbitrary macros, which do not come pre-equipped with type rules, this allows Typed Scheme to handle the full range of existing programs. Second, it allows the implementation to deal with just the few core forms of PLT Scheme.

Since almost all control structures in PLT Scheme are defined in terms of **if**, occurrence typing works for them without additional modification. For example, the **match** library provides a sophisticated pattern matcher, which compiles to plain PLT Scheme. Typed Scheme works properly for almost all uses of **match**, without any special implementation effort. Since PLT programmers use thousands of different macros, this is the only way to make the implementation manageable.

However, some macros introduce invariants that must be understood by the type system for proper typechecking. These must be handled specially. For example, the **define-struct** macro must communicate with the typechecker to introduce new types, and thus Typed Scheme introduces the

[1] This is also the approach taken by the ACL2 theorem prover [Kaufmann, Moore, and Manolios 2000].

new **define-struct:**. This means that further complex macros of this sort require special handling, preventing their use in Typed Scheme currently. The problem of sound and simple macro-extensibility of typed languages is still an open problem.

3.5 No New Idioms

For Typed Scheme, I took the idioms and styles of PLT Scheme programmers as they exist. The addition of a type system allows for many new styles of programming, and a type system designer is always tempted to add these. I have attempted to resist this temptation, and only add features to typecheck PLT Scheme programs that already exist.[2] This has provided a rigorous bound on what features to add to Typed Scheme, and on what the goals of the Typed Scheme project are. As Typed Scheme becomes more widely used, and is used as the initial language for programs, this decision may need to be revisited.

[2] Refinement types are somewhat of an exception to this rule.

CHAPTER 4

Prior Work

The design of Typed Scheme builds on extensive prior work. This chapter surveys the most important strands. The first section discusses the extensive history of static checking for untyped languages, in particular LISP and Scheme. While Typed Scheme differs significantly from all of these prior efforts, their design choices and experience informed the choices made in creating Typed Scheme. The second section reviews prior work on designing multi-language systems, especially those that use contracts at the boundaries. The third section describes previous solutions to the problem of typechecking a language with macros, as it has been explored both in the context of static checkers and soft typing. The final section covers two key innovative technologies that Typed Scheme builds on–runtime software contracts and module systems.

4.1 Static Checking for Scheme and LISP

The history of programming languages knows many attempts to add or to use type information in conjunction with untyped languages. In 1976, Cartwright [1976] proposed a type system to simplify the operation and use of Boyer-Moore-style theorem provers [Boyer and Moore 1997]. Since then, the designers of Common Lisp [Steele Jr. 1990] as well as many other languages have included 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, Wimmers, and Lakshman 1994; Wright and Cartwright 1997; Henglein and Rehof 1995; Flanagan and Felleisen 1999; Meunier, Findler, and Felleisen 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] and Erlang [Marlow and Wadler 1997].

This section surveys this body of work, starting with previous static type systems for Scheme and LISP.

4.1.1 Static Type Systems

Many researchers have developed systems that brought together both static type systems and LISP or Scheme programming, usually to make it easier to reason about programs. In 1976, Cartwright presented TYPED LISP [1976]. However, in his paper (section 5) he describes having to abandon the policy of rejecting type-incorrect programs because the types of variables in conditionals had overly broad types, which is precisely the issue that Typed Scheme and occurrence typing address. TYPED LISP also did not consider the problem of interoperation with untyped code.

Wand's Semantic Prototyping System [Wand 1984] included a type system for Scheme programs, designed for implementing denotational models of languages. The type system did not include any features for handling typical Scheme idioms, nor for interoperating soundly with untyped code.

Haynes [Haynes 1995] designed a type system called Infer based on row types for Scheme. His system did not accommodate Scheme programming idioms in the way that Typed Scheme does, and aimed for complete type inference for all programs, a non-goal of the Typed Scheme project. However, his system does support restricted forms of variable-arity polymorphism [Dzeng and Haynes 1994]. This system is discussed in detail in section 12.6.

McDermott [2004] developed Nisp, an unsound type system for Common LISP. It allows the description and checking of many types similar to those in Typed Scheme, but does not provide checking for typical LISP idioms. It also does not provide sound interaction with untyped code.

Leavens [Leavens, Clifton, and Dorn 2005] designed an ML-like type system for Scheme, but made no attempt to accommodate Scheme idioms, or to integrate with untyped code.

4.1.2 Soft Typing

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 any type definitions or type declarations. Soft typing should work via type inference only. 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 invariants of the type system are not violated. 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 programmers to inspect the sites of these run-time checks to determine whether they represent genuine errors or weaknesses of the type system.

Such soft typing systems are complex and brittle, however. On one hand, these systems may infer extremely large types for seemingly simple expressions, greatly confusing the programmer, either the original or the maintenance 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; Aiken and Murphy 1991] 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 Shivers'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 value 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, Flatt, Krishnamurthi, Weirich, and Felleisen 1996].

My experience and evaluation, as well as that of other users of soft typing systems, 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 prior soft typing research inspired this work on occurrence typing. These systems employed simplistic **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 the generalization to occurrence typing. 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, I hope to borrow techniques from soft typing for automating some of the conversion process from untyped modules to typed modules.

4.2 Interlanguage Interoperability

Smooth operation between typed and untyped code is essential to Typed Scheme. The design of this interoperation relies heavily on prior work on language interoperation. Gray, Findler, and Flatt [2005] developed ProfessorJ, which combines PLT Scheme and Java. Their interoperability strategy relies on custom *mirrors*, generated specifically for each class, which dynamically check the invariants of methods. They also extend each language with the features of the other, using the Java-like class system of PLT Scheme [Flatt, Findler, and Felleisen 2006], and adding first-class functions to their implementation of Java. Their system also automatically generates contracts as part of the mirrors for each class.

Matthews and Findler [2009] consider interlanguage interoperability between a typed and untyped lambda calculus, and demonstrate the such interoperability corresponds to contracts, although their work does not generate contracts. They claim, but do not prove, that only untyped portions of the program can be blamed. However, their system has only two parties.

4.3 Implementing Types in Scheme

The implementation of typecheckers for Scheme poses several challenges. First, the typechecker must somehow cope with user-written macros. Second, the typechecker must accommodate pre-existing code in a sound manner, preferably without requiring changes to the untyped Scheme implementation. Third, several systems, like Typed Scheme, have implemented the type system via the underlying macro system.

The SPS system mentioned above used unhygenic macros to implement the typechecker. Typed Scheme scales up the approach of SPS to a modern macro and module system, enabling the seamless integration between typed and untyped code.

Flanagan's static analyzer for DrScheme [Flanagan et al. 1996] analyzed expanded programs and used syntax source information to display the analysis results in the program editor. The analyzer included a macro protocol that let programmers annotate their programs with hints for the analyzer.

The Ziggurat project [Fisher and Shivers 2008] has investigated alternate approaches to static analysis and other program observations in the presence of macros. Analyses in Ziggurat are implemented as methods on expression nodes, and derived expression forms (that is, macros) may either override analysis methods with special behavior or defer to the default analysis of the macro's expansion. Ziggurat represents a new approach to defining macro protocols, and it is as yet unclear how the Ziggurat approach compares with the mechanisms described here.

4.4 Contracts and Modules

The design and implementation of Typed Scheme builds heavily on prior work in contracts and module systems.

Runtime software contracts have a long history, but traditionally did not support two features necessary for the design of Typed Scheme. First, Scheme, and thus Typed Scheme, makes extensive use of higher-order language constructs, which must therefore be accommodated by the contract system. Second, Typed Scheme isolates typed from untyped code, and thus needs a mechanism for determining which portion of the program violated a contract.

These two shortcomings are addressed by Finder's contract system [Findler and Felleisen 2002], as implemented in PLT Scheme [Flatt and PLT 2009]. This system provides guarantees that ensure that the appropriate portion of code is blamed for violating a contract, even when the violation is of a higher-order contract and takes place arbitrarily later in the execution of the program. Typed Scheme builds directly on this system, automatically generating contracts from types.

Typed Scheme also relies on a module system that provides strong abstraction barriers, in order to prevent unauthorized access by the untyped code to unprotected portions of the typed program. For Typed Scheme, this module system is the PLT module system [Flatt 2002; Flatt and PLT 2009]. This system provides strong abstraction barriers for both values and macros. It also allows Typed Scheme to be implemented entirely as a library, using the techniques described in chapter 10.

CHAPTER 5

Integrating Typed and Untyped Code

This chapter presents a formal account of the integration of typed and untyped code in a single program. For simplicity, we consider only a simple type system and module system. Further, we restrict the language to have only one typed module.

5.1 Relationship to Typed Scheme

This chapter presents a formalism that admits only simple types. Further, each module may contain exactly one expression, and other modules are referred to implicitly. Finally, there may only be one typed module in the system. This formalism, initially designed and published prior to the implementation of Typed Scheme, is thus inadequate for direct use in Typed Scheme.

Therefore, in the full Typed Scheme language, several changes are made. First, a program may consist of an arbitrary combination of arbitrarily many typed and untyped modules. Second, each module explicitly specifies which modules it depends upon. Third, and most significantly, the importing typed module must specify the types to be assigned to any imports from untyped modules.

The implemented system has a number of advantages. First, the type specifications for imports provide checked documentation of the expected behavior of the untyped code. Second, as in the wrapper modules presented in this chapter, they provide a localized target of blame. Therefore, when the untyped module fails to live up to the contract assigned to it, instead of blaming the untyped module, which may not be at fault, the type specification is blamed instead. Third, it does not require the complex contract inference process from this chapter for determining the desired type of imported untyped identifiers.

Of course, the crucial result of this chapter, Theorem 5.4.12, is still applicable to the resulting system, despite these changes. Developing a formalism that more accurately captures the design of the Typed Scheme implementation remains future work.

5.2 An Informal Tour

To make things simple, we assume that a program is a sequence of modules followed by a "main expression."

The evaluation of such a program starts with the main expression. Evaluation proceeds as usual in an expression-oriented language. When the evaluation process encounters a reference to a module, it imports the appropriate expression from there and evaluates it.

To keep things as simple as possible, we work with a typed variant of the untyped language where all binding positions come with type declarations. In short, we migrate to an explicitly typed language that is otherwise syntactically and semantically equivalent to the untyped one.

In a program that mixes typed and untyped modules, evaluation proceeds as before. This implies that the values of the typed language are those of the untyped language (and vice versa). Figures 5.1, 5.2, 5.3 and 5.4 present the formal syntax and semantics of our model, though this section liberally adds features to provide intuition; section 5.3 explains these figures in detail.

$$\begin{array}{lll}
P & ::= e \mid M P & \text{Programs} \\
M & ::= (\textbf{module}\ f\ v) & \text{Modules} \\
v & ::= n \mid (\lambda x.e) & \text{Values} \\
e & ::= v \mid x \mid f \mid (e\ e) \mid (\textbf{if0}\ e\ e\ e) & \text{Expressions}
\end{array}$$

Figure 5.1: Scripting Language Syntax

$$\begin{array}{lll}
P & ::= e_m \mid M P & \text{Programs} \\
M & ::= M_c \mid M_u \mid M_t & \text{Modules} \\
M_u & ::= (\textbf{module}\ f\ v) & \text{Untyped Modules} \\
M_c & ::= (\textbf{module}\ f\ c\ v) & \text{Contracted Modules} \\
M_t & ::= (\textbf{module}\ f\ t\ v_t) & \text{Typed Modules} \\
c & ::= \textbf{int} \mid (c \rightarrow c) \mid \textbf{int} \vee (c \rightarrow c) & \text{Contracts} \\
t & ::= \textbf{int} \mid (t \rightarrow t) & \text{Types} \subseteq \text{Contracts} \\
v & ::= n \mid (\lambda x.e) & \text{Untyped Values} \\
v_t & ::= n \mid (\lambda x : t.e_t) & \text{Typed Values} \\
v_m & ::= n \mid (\lambda x : t.e_m) \mid (\lambda x.e_m) & \text{Mixed Values} \\
e & ::= v \mid x \mid f \mid (e\ e) \mid (\textbf{if0}\ e\ e\ e) & \text{Untyped Expressions} \\
e_t & ::= v_t \mid x \mid f \mid (e_t\ e_t) \mid (\textbf{if0}\ e_t\ e_t\ e_t) & \text{Typed Expressions} \\
e_m & ::= v_m \mid x \mid f \mid (e_m\ e_m) \mid (\textbf{if0}\ e_m\ e_m\ e_m) & \text{Mixed Expressions}
\end{array}$$

Figure 5.2: Typed Language Syntax

$$\begin{array}{ll}
v ::= n \mid (\lambda x : t.e_t) \mid (\lambda x.e) \mid \{(c \dashrightarrow c) \Leftarrow^f v\} & \text{Values} \\
e ::= v \mid x \mid f \mid (e\ e) \mid (\textbf{if0}\ e\ e\ e) \mid \{c \Leftarrow^f e_t\} & \text{Expressions}
\end{array}$$

Figure 5.3: Runtime Syntax

Given an untyped modular program, the first step of interlanguage migration is to turn one module into a typed module. We assume that this step simply adds types to all binding positions of the module, including the module exports. While porting to Typed Scheme is not always this easy, the additional changes sometimes required do not affect the modeling process.

After the chosen module has been rewritten in the typed version of the language, we need to check the types and infer from them how the typed module is going to interact with the others, which remain untyped. Consider the following simplistic program:

$$\begin{array}{llll}
((\lambda x.e)\ v) & \longrightarrow & [v/x]e & \textsc{Subst} \\
((\lambda x:t.e)\ v) & \longrightarrow & [v/x]e & \textsc{TypedSubst} \\
(n\ v)^f & \longrightarrow & (\textbf{blame}\ f) & \textsc{App-Error} \\
(\textbf{if0}\ 0\ e_1\ e_2) & \longrightarrow & e_1 & \textsc{if0-True} \\
(\textbf{if0}\ v\ e_1\ e_2) & \longrightarrow & e_2 & \textsc{if0-False} \\
\{\textbf{int} \Leftarrow^g n\} & \longrightarrow & n & \textsc{Int-Int} \\
\{\textbf{int} \vee c \Leftarrow^g n\} & \longrightarrow & n & \textsc{Int-IntOr} \\
\{\textbf{int} \vee c \Leftarrow^g v\} & \longrightarrow & \{c \Leftarrow^g v\} & \textsc{Int-LamOr} \\
\{\textbf{int} \Leftarrow^g v\} & \longrightarrow & (\textbf{blame}\ g) & \textsc{Int-Lam} \\
\{(c_1 \rightarrow c_2) \Leftarrow^g n\} & \longrightarrow & (\textbf{blame}\ g) & \textsc{Lam-Int} \\
\{(c_1 \rightarrow c_2) \Leftarrow^g v\} & \longrightarrow & & \textsc{Lam-Lam} \\
\qquad \{(c_1 \dashrightarrow c_2) \Leftarrow^g v\} & & & \\
(\{(c_1 \dashrightarrow c_2) \Leftarrow^g v\}\ w)^f & \longrightarrow & & \textsc{Split} \\
\qquad \{c_2 \Leftarrow^g (v\ \{c_1 \Leftarrow^f w\})\} & & & \\
\ldots(\textbf{module}\ f\ v)\ldots E[f] & \longrightarrow & & \textsc{Lookup} \\
\qquad \ldots(\textbf{module}\ f\ v)\ldots E[v] & & & \\
\ldots(\textbf{module}\ f\ c\ v)\ldots E[f] & \longrightarrow & & \textsc{Lookup-Contract} \\
\qquad \ldots(\textbf{module}\ f\ c\ v)\ldots E[\{c \Leftarrow^f v\}] & & & \\
\ldots(\textbf{module}\ f\ t\ v)\ldots E[f^g] \text{ where } g \neq f & \longrightarrow & & \textsc{Lookup-Type} \\
\qquad \ldots(\textbf{module}\ f\ t\ v)\ldots E[\{t \Leftarrow^g v\}] & & & \\
\ldots(\textbf{module}\ f\ t\ v)\ldots E[f^f] & \longrightarrow & & \textsc{Lookup-Type-Self} \\
\qquad \ldots(\textbf{module}\ f\ t\ v)\ldots E[v] & & &
\end{array}$$

Figure 5.4: Reduction Rules

Example 29

(**module** *f* (*int* → *int*) (λ ([*x* **:** *int*]) (*g x*)))
(**module** *g* 999)
(*f* 5)

It consists of two modules: the first is presumably a module that has been rewritten in the typed language, the second one is still in the untyped language. Also, the first one exports a function from integers to integers; the second one exports a simple integer.

If we were to evaluate this program as is, it would eventually attempt to apply 999 to 5 via the application $(g\ x)$ in the typed module. In other words, the typed portion of the program would trigger a run-time error, which, assuming proper source tracking, would tell the programmer that the typed module went wrong.

A different view of the problem is that when one module changes, the rest of the program has to play by new rules, too. In this case, the very fact that the export from *g*, the second module, is used as a function in the typed module establishes an agreement between the two modules. This agreement, however, is informal (and unuttered) and is neither checked nor monitored during run-time. The evaluation therefore results in a run-time error seemingly due to the typed module.

Thus our first lesson is that informal agreements don't jive with the goal of introducing types. To reap the benefits of types, we must not only have agreements, we must state them and enforce them. This line of reasoning naturally suggests the use of behavioral contracts in the spirit of Findler and Felleisen [2002]. More precisely, we assume that an interlanguage migration process has access to the interfaces of the remaining modules and that it is possible to add contracts to these interfaces.[1]

For our running example we would expect that migration changes the program as follows:

Example 30

```
(module f (int → int) (λ ([x : int]) (g x)))
(module g (integer? → integer?) 999)
(f 5)
```

Put differently, we can infer from the types of the first module that the second module must always export a function from integers to integers. In our framework, we express this fact with a contract to the module interface.

The example has two implications for interlanguage migration and its formal model. First, the language must also include optional contracts at module boundaries. Second, a type checker for the typed variant of the language must not only enforce the rules of the type system, it must also infer the contracts that these type annotations imply for the remaining modules.

[1] In the implementation of Typed Scheme, these contracts are inserted at the point where the untyped module is *required*, as described in chapter 10.5.

Unfortunately, there are yet more problems. Consider this second program, which applies an (**int** → **int**) function to *false*, a boolean.

Example 31

```
(module h (int → int)
  (λ ([y : int])
    (let ((g (λ ([x : int]) (+ x 10))))
      (+ (g y) (g 10)))))
(h false)
```

Since our evaluator ignores types, the boolean value flows into the typed module without any objections, only to cause havoc there. Again, the typed module appears to have gone wrong.

In this case, the solution is to interpret the types on the module exports as contracts so that the evaluator monitors how the other modules use functional exports from the typed module. For flat types such as **int**, the values that flow into typed functions are checked immediately; for functional values, the contracts are distributed over the domain and range of the function until flat values show up [Findler and Felleisen 2002]. Technically, the types become contracts on external references to the module *f*, and are interpreted as runtime checks, or casts, which specify the party to be blamed if they fail:

(*h false*)

steps to

({(*integer?* → *integer?*) <= *h* **:** *Main*} *false*)

steps to

{*integer?* <= (*h* {*integer?* <= *false* **:** *Main*}) **:** *Main*}

At this point it has become clear that *false* is a bad value, and the evaluator can abort the execution blaming the main expression for supplying bad values to the typed module.

Simply adding contracts to existing modules doesn't solve all problems, though:

Example 32

```
(module f int
  (if (not (m 5))
      (m true)
      7))
(module m ((or/c boolean? integer?)
             →
           (or/c boolean? integer?))
  (λ (x)
    (if (boolean? x) true (+ x 1))))
f
```

The first, typed module contains two references to *m*, the second, untyped module. From the types of the subexpressions we can infer that *m* must export a function that can consume both booleans and integers and that can also produce both kinds of values. The resulting contract uses *or/c* in the domain and range part to state this fact but it still means that the evaluation of *f* raises a run-time error because (*m true*) produces a boolean rather than a number.

Our solution is to add new wrapper modules with more specific contracts based on the existing type and contract information, as shown in figure 5.5. The new wrapper modules *m-int* and *m-bool* provide additional guarantees about the behavior of *m*, by placing stricter runtime contracts on the body of *m* than the contracts that were originally inferred. The new contracts allow the original module *f* to typecheck. When the program is evaluated, the contracts show that *m-bool* misrepresented *m*, and is thus blamed for the runtime violation.

This strategy raises the question: why not add wrapper modules everywhere. Doing so would add safety checks for each variable occurrence. This solution, while conceptually simple, fails to generate maintainable code. Each time this migration step is applied (for example, for each module ported from being untyped to typed), a slew of new wrapper modules would be created. Soon, the system would be an incomprehensible mess.

This points to a design requirement we consider fundamental, namely,

```
                                                   Example 33
(module f int
  (if (not (m-bool 5)) (m-int true) 7))

(module m-int
      ((or/c boolean? integer?) → integer?)
      m)

(module m-bool
      ((or/c boolean? integer?) → boolean?)
      m)

(module m ((or/c boolean? integer?) → (or/c boolean? integer?))
  (λ (x)
    (if (boolean? x) true (+ x 1))))

f
```

Figure 5.5: Wrapper Modules

that the resulting program be maintainable. Our transformation is not a compilation strategy to ensure safety, it is a migration that is part of the development of a system. As with a semantics-preserving refactoring, it must respect the surrounding code as much as possible.

The rest of the chapter presents a formal model of this migration process, drawing on our experience of implementing the model as a prototype. The focus of our presentation concerns the derivation of constraints via type checking; the translation of these constraints into contracts; the addition of wrapper modules based on types and contracts; and last but not least a theorem that proves that typed modules can't go wrong in this setting.

5.3 The Formal Framework

The objective of this section is to formally describe our interlanguage migration framework, from the syntax and semantics of the programming languages all the way to the linguistic aspects of the process itself.

5.3.1 Syntax

Our scripting language is a simplified version of the language of Meunier et al. [2006], augmented with types and typed modules. It consists of the lambda calculus enriched with numeric constants and a conditional, as well as casts, modules and contracts. The initial syntax used in the original program is specified in figure 5.1. After the migration step, the syntax is more complicated: see figure 5.2. The runtime system collapses some distinctions and adds casts, which are specified in figure 5.3.

Modules Our language has a simple first-order module system, in which each module consists of a name and a value. The module exports its value via its name. Two other module forms are provided: modules with contracts and modules with types. Contracted modules are identical to their untyped counterparts, except that a contract is added to the (simplified) module interface. When the value of the module is used, that value is checked against the contract. Typed modules have a top-level type, and contain only typed values, v_t in figure 5.2.

Contracts and Casts The contracts allow the base **int** contract, as well as function contracts and disjunction. Function contracts have the Findler–Felleisen semantics [Findler and Felleisen 2002], and disjunction allows either of the two branches to be satisfied. The disjunction of two function contracts is syntactically prohibited. This restriction significantly reduces the complexity of the reduction rules for $\vee$-contracts. At run-time, contracts turn into checks, which we express with casts. Syntactically, a cast $\{c \Leftarrow^{e} m\}$ combines a contract with an expression and a label for a module, which it blames for the contract violation if the check fails.

Casts are not part of the source language, but the state space of the runtime system; programmers cannot write them and our transformation does not insert them.

Types The types of the typed fragment of the language are just the base type **int** and function types. Importantly, every type is syntactically also a contract.

Expressions The language contains three kinds of expressions: typed, untyped and mixed. Typed expressions occur only in the typed module. Untyped expressions occur in all other modules. Mixed expressions can contain typed and untyped subexpressions and appear only in the main expression. The main expression is initially untyped.

5.3.2 Semantics

For the dynamic semantics, we assume that every expression has been labeled with the name of its original source module. The main expression is labeled with m. This labeling is necessary for appropriate blame assignment when a dynamic error occurs. It corresponds to an annotation pass for source location tracking. For clarity, we omit these labels wherever they are not needed.

The dynamic semantics is defined in figure 5.4 as a reduction semantics, and again follows Meunier et al., with additions for types and $\vee$-contracts. Reduction takes place in the context of modules, which are not altered during reduction. The relation $\rightarrow$ is the one-step (standard) reduction relation, with $\rightarrow^*$ as its reflexive, transitive closure, and the set of evaluation contexts is defined as:

$$E = [] \mid (E\ e) \mid (v\ E) \mid (\mathbf{if0}\ E\ e\ e) \mid \{c \Leftarrow^f E\}$$

Rules that do not refer explicitly to the context are implicitly wrapped in $E[-]$ on both sides, with the exception of the rules that reduce to ($\mathbf{blame}\ f$), which discard the evaluation context.

The reduction rules fall into the following categories:

- The rules that lookup module references all refer explicitly to the module context. The LOOKUP rule refers to untyped modules, and simply substitutes the body of the module for the reference. The LOOKUPCONTRACT and LOOKUPTYPE rules retrieve the appropriate expression and wrap it in a contract. The contract wrapped around typed module bodies is necessary so that typed expressions are never used in incorrect ways, even when the untyped modules refer to the typed module. This check is not necessary when the typed module refers to itself, and is thus omitted in the LOOKUPTYPESELF rule [Findler and Felleisen 2002].

- The rules for the core are straightforward. These include SUBST and TYPEDSUBST, which perform β_v-reduction on untyped and typed abstraction respectively. IF0-TRUE and IF0-FALSE are also simple.
- APP-ERROR is the one runtime error that does not involve a contract or a cast. If a number is in the application position of an application, clearly the invariants of the language have been violated. We blame the source of the application for this error.

  The remaining rules handle contracts and casts.
- INT-INT and INT-INTOR both pass numbers through **int** contracts unchanged. INT-LAMOR discards an unsatisfiable disjunct. We do not need rules for disjunction with an arrow contract on the left, since such contracts are syntactically prohibited.
- INT-LAM and LAM-INT represent contract failures and blame the appropriate party, as labeled on the cast.
- LAM-LAM blesses an arrow contract applied to an abstraction, turning it into a blessed arrow contract. In contrast to INT-INT, we must keep the contract around for later use. The resulting expression, while still a cast, is also a syntactic value.
- The SPLIT rule breaks a blessed arrow contract into its positive and negative halves, and places them around the argument and the entire

application. The creation of two new casts, with appropriate blame assignment, is the key to proper contract checking for higher-order functions [Findler and Felleisen 2002].

5.3.3 Adding Type Declarations

The first step in interlanguage migration requires the programmer to change one module from the untyped language to the typed one. In our system, this involves adding types to every variable binding and to the module export as a whole.

Once this module is annotated, the new program is referred to as P^{M}, where *M* is the name of the now-typed module.

Since the simply-typed λ-calculus has a straightforward type-soundness theorem, we might expect a similar one to hold for migrated programs, provided the type annotations are self-consistent. Sadly, this is not the case. For example, the typed module *M* might refer to some other module, which could provide an arbitrary value or raise a run-time error. The other modules may contain outright errors, such as $(3\ 4)$, as well as untypeable expressions such as $(\lambda x.(\textbf{if0}\ x\ 1\ (\lambda y.y)))$; we do not rule these out, since the programmer is only adding types to one module. Furthermore, since we do not typecheck the other modules, they may use the typed module in ways that do not accord with its type. Because of these possibilities, the migration process must protect the typed module from its untyped brethren.

5.3.4 Inferring Constraints from Types

In order to protect the typed module when it refers to untyped ones, we apply a transformation to the program that expresses the implicit agreements between the typed and untyped modules as contracts. Our transformation examines the references to other modules in *M* and from these uses infers contracts that become obligations of those other modules. For example, in

MT-VAR

$$\Gamma \vdash^{RT} x : \Gamma(x); \emptyset$$

MT-MODVAR

$$\Gamma \vdash^{RT} f : t; \{f \triangleleft t\} \quad f \neq M$$

MT-MODVARSELF

$$\Gamma \vdash^{RT} M : t; \emptyset \text{ if module } M \text{ has type } t.$$

MT-INT

$$\Gamma \vdash^{RT} n : \mathbf{int}; \emptyset$$

MT-ABS

$$\frac{\Gamma, x : t \vdash^{RT} e : s; \phi}{\Gamma \vdash^{RT} (\lambda x : t.e) : (t \to s); \phi}$$

MT-APP

$$\frac{\Gamma \vdash^{RT} e_1 : (M_1 \to M_2); \phi_1 \qquad \Gamma \vdash^{RT} e_2 : M_1; \phi_2}{\Gamma \vdash^{RT} (e_1\ e_2) : M_2; \phi_1 \cup \phi_2}$$

MT-IF0

$$\frac{\Gamma \vdash^{RT} e_1 : M_1; \phi_1 \qquad \Gamma \vdash^{RT} e_2 : M_2; \phi_2 \qquad \Gamma \vdash^{RT} e_3 : M_2; \phi_3}{\Gamma \vdash^{RT} (\mathbf{if0}\ e_1\ e_2\ e_3) : M_2; \phi_1 \cup \phi_2 \cup \phi_3}$$

Figure 5.6: Constraint Generation

the following program, the context makes it obvious that g must have type ($\mathbf{int} \to \mathbf{int}$):

(**module** *f int* (*g* 5))

The transformation would therefore add the contract ($\mathbf{int} \to \mathbf{int}$) to g.

The type system in figure 5.6 formalizes this intuition. Its rules define the judgement

$$\Gamma \vdash^{RT} e : t; \Phi$$

which states that in type environment Γ, expression e has type t under the constraint set Φ. The rules are similar to those of the simply typed λ-calculus, propagating and combining the constraints from their constituents, except for the module variable reference rules: MT-MODVARSELF and MT-MODVAR. The former checks references to the typed module itself. The latter allows a module variable to be assigned any type and adds a matching constraint to the constraint set. Constraints are of the form $g \triangleleft c$, which states that g must have contract c.

Because of the non-determinism of the MT-MODVAR rule, these rules do not naturally map onto a syntax-directed type checker. They do define a logic program, however, which potentially produces many solutions, each satisfying the desired type, and including a matching set of constraints. Each of these solutions gives rise to a potentially different set of contracts imposed on the other modules in the program. For example, consider the following program:

(**module** *f int* (*g h*))

There are many possible sets of constraints that could be generated by our system. A simple one requires that h be an **int** and that g have the contract (**int** → **int**). Of course, there are infinitely many possibilities. While a real system for migration would use programmer input and static analysis to choose one solution, our soundness theorem holds for all of them.

$$\begin{array}{rcl} merge(c, c) & = & c \\ merge(\textbf{int}, c) & = & \textbf{int} \vee c \\ merge(c, \textbf{int}) & = & \textbf{int} \vee c \\ merge((c_1 \rightarrow c_2), (c_3 \rightarrow c_4)) & = & \\ \multicolumn{3}{r}{(merge(c_1, c_3) \rightarrow merge(c_2, c_4))} \end{array}$$

Figure 5.7: From Constraints to Contracts

5.3.5 From Constraints to Contracts

The next step is to turn the set of constraints into an actual contract. For example, the constraint set for our first example is $\{g \lhd (\textbf{int} \rightarrow \textbf{int})\}$. The obvious contract is then (**int** → **int**) for module g. When there are multiple references to a module variable within the typed module, however, there are necessarily multiple constraints on that module, which we must somehow combine into a single contract.

Two obvious approaches present themselves. Consider the following program:

```
(module f (int → int)
         (λ ([x : int]) ((g 1) (g 0))))
(module g (λ (x) (if0 x 1 (λ (y) y))))
f
```

Given our evaluation rules for **if0**, this is a perfectly reasonable untyped program. Furthermore, the untypeable expression is in g, to which we are not adding types.

The constraints generated for g are

$$\{g \triangleleft (\mathbf{int} \rightarrow \mathbf{int}), g \triangleleft (\mathbf{int} \rightarrow (\mathbf{int} \rightarrow \mathbf{int}))\}$$

First, if the language of contracts supported conjunction, the merge operation could just compute the conjunction of all constraints. This would give

$$(\mathbf{int} \rightarrow ((\mathbf{int} \rightarrow \mathbf{int}) \wedge \mathbf{int}))$$

as the contract for g. But no value is both a function and an integer, so this contract cannot possibly be satisfied. Since we want to allow this program, conjunction is not the correct solution.

Second, the merge can choose the disjunction of the constraints. In our example, we would get

$$(\mathbf{int} \rightarrow (\mathbf{int} \vee (\mathbf{int} \rightarrow \mathbf{int})))$$

which is legal and acceptable

The process of combining constraints into contracts is specified in figure 5.7. This process ensures that contracts are tidy in arrows: there is only one arrow contract in any disjunction. This invariant is required by our contract syntax, and simplifies the reduction rules for contracts with disjunction.

5.3.6 Adding Wrapper Modules

Unfortunately, disjunction in contracts introduces new problems. The contract we assign to g is not sufficient to establish that $(g\ 1)$ produces an abstraction, which is required for the body of f to execute without type errors.

$$\textsc{AC-ModVarContract}\quad \frac{(\textbf{module}\ f\ c\ e) \in P \qquad \vdash c \Rightarrow t}{P, \Gamma \vdash^{\Rightarrow} f : t; f; \emptyset} \qquad \textsc{AC-ModVarWrap}\quad \frac{(\textbf{module}\ f\ c\ e) \in P \qquad \vdash c \not\Rightarrow t}{P, \Gamma \vdash^{\Rightarrow} f : t; \textit{f-t}; \{(f, t)\}}$$

$$\textsc{AC-ModVarSelf}\quad \frac{(\textbf{module}\ M\ t\ e) \in P}{P, \Gamma \vdash^{\Rightarrow} M : t; M; \emptyset} \qquad \textsc{AC-Var}\quad P, \Gamma \vdash^{\Rightarrow} x : \Gamma(x); x; \emptyset \qquad \textsc{AC-Int}\quad P, \Gamma \vdash^{\Rightarrow} n : \textbf{int}; n; \emptyset$$

$$\textsc{AC-App}\quad \frac{P, \Gamma \vdash^{\Rightarrow} e_1 : (M_1 \rightarrow M_2); e_1'; C_1 \qquad P, \Gamma \vdash^{\Rightarrow} e_2 : M_1; e_2'; C_2}{P, \Gamma \vdash^{\Rightarrow} (e_1\ e_2) : M_2; (e_1'\ e_2'); C_1 \cup C_2}$$

$$\textsc{AC-If0}\quad \frac{P, \Gamma \vdash^{\Rightarrow} e_1 : M_1; e_1'; C_1 \qquad P, \Gamma \vdash^{\Rightarrow} e_2 : M_2; e_2'; C_2 \qquad P, \Gamma \vdash^{\Rightarrow} e_3 : M_2; e_3'; C_3}{P, \Gamma \vdash^{\Rightarrow} (\textbf{if0}\ e_1\ e_2\ e_3) : M_2; (\textbf{if0}\ e_1'\ e_2'\ e_3'); C_1 \cup C_2 \cup C_3}$$

$$\textsc{AC-Abs}\quad \frac{P, \Gamma, x : t \vdash^{\Rightarrow} e : s; e'; C}{P, \Gamma \vdash^{\Rightarrow} (\lambda x : t.e) : (t \rightarrow s); (\lambda x : t.e'); C}$$

Figure 5.8: Generating Wrapper Modules

Therefore, we must place some additional constraint on $(g\ 1)$. To implement these constraints, we add a wrapper module to the program, with a more precise contract for g. To accomplish this, we perform a second pass over the typed module, accumulating unsatisfied constraints and changing module references to point to the new wrapper modules. Then the transformation adds the required wrappers to the program.

These rules have three interesting cases:

- AC-ModVarContract applies for module variables where the contract is sufficient to ensure that the type is always satisfied.
- AC-ModVarWrap requires a new wrapper module and changes the module reference to the new wrapper module, whose name is formed from the original name and the type, when the contract is insufficient.
- AC-ModVarSelf handles self-reference to the typed module *M*.

The other rules merely recur structurally.

Once we have a collection of constraints C, it is trivial to construct the appropriate wrapper modules. For every element $(f, t) \in C$, simply add a new module of the form

(**module** *f-t t f*)

to the original program.

This transformation relies on a relationship between contracts and types. We write $c \Rightarrow t$ when contract c establishes the preconditions for type t. Then we insert casts precisely at those module references f in *M* when $c \not\Rightarrow t$, where c is the contract on f and t is the desired type. We formalize this relation with two new judgements

$$\vdash c \Rightarrow t$$

and

$$\vdash c \Leftarrow t$$

These judgements are defined in figure 5.9. The first judgement states that contract c is sufficient to establish the preconditions of type t. The second is the converse, namely that t establishes the preconditions for c. The second judgement is not used in the rest of the transformation; it is only needed for the definition of the first.

$$\vdash \textbf{int} \Rightarrow \textbf{int} \qquad \frac{\vdash c_1 \Leftarrow t \qquad \vdash c_2 \Rightarrow s}{\vdash (c_1 \rightarrow c_2) \Rightarrow (t \rightarrow s)}$$

$$\vdash \textbf{int} \Leftarrow \textbf{int} \qquad \frac{\vdash c_1 \Leftarrow t}{\vdash c_1 \vee c_2 \Leftarrow t} \qquad \frac{\vdash c_2 \Leftarrow t}{\vdash c_1 \vee c_2 \Leftarrow t} \qquad \frac{\vdash c_1 \Rightarrow t \qquad \vdash c_2 \Leftarrow s}{\vdash (c_1 \rightarrow c_2) \Leftarrow (t \rightarrow s)}$$

Figure 5.9: Contracts imply types

With this definition, we can now define a further transformation on the typed module, which adds wrappers at the places we have just described. This transformation is presented in figure 5.8, which defines the judgement

$$P, \Gamma \vdash^{\Rightarrow} e : t; e'; c$$

This states that Γ proves that e can be transformed to e', which then has type t with wrapper modules generated from constraints c.

5.3.7 Summary of the transformation

The transformation we have just described is MT, for migration transformation. $MT(P)$ transforms P as follows:

1. The programmer chooses one module *M* from P and adds types to this module, so that is is now a typed module according to the grammar and type system, producing *M*′.

2. Using the type system described in figure 5.6, generate constraints from module *M*′.

3. Merge these constraints according to figure 5.7 to produce contracts, which are added to the other modules.

4. Transform *M*′ into *M*″ to accommodate weak contracts.

5. Add wrapper modules as required by the transformation of *M*′ to *M*″.

This gives a new program $P^{\mathbf{M}}$.

Since step 2 in this process is nondeterministic, $MT(P)$ produces a set of programs. We prove in the next section that every element $P^{\mathbf{M}}$ of this set implements P.

5.4 Soundness

Before we can prove that our migration transformation is sound, we must first define what soundness means for a partially typed system. It cannot mean absence of runtime errors, since not all modules are necessarily typed. All we can say instead is that the *typed* modules do not go wrong.

$$(\textbf{module}\ f\ e) \rhd (\textbf{module}\ f\ e)$$

$$(\textbf{module}\ f\ e) \rhd (\textbf{module}\ f\ c\ e)$$

$$\frac{e \rhd e'}{(\textbf{module}\ f\ e) \rhd (\textbf{module}\ f\ t\ e')}$$

$$x \rhd x \qquad f \rhd f \qquad n \rhd n$$

$$\frac{e \rhd e'}{e \rhd \{t \Leftarrow^{f} e'\}}$$

$$\frac{e_1 \rhd e_1' \qquad e_2 \rhd e_2'}{(e_1\ e_2) \rhd (e_1'\ e_2')} \qquad \frac{e \rhd e'}{(\lambda x.e) \rhd (\lambda x : t.e')}$$

$$\frac{e_1 \rhd e_1' \qquad e_2 \rhd e_2' \qquad e_3 \rhd e_3'}{(\textbf{if0}\ e_1\ e_2\ e_3) \rhd (\textbf{if0}\ e_1'\ e_2'\ e_3')}$$

Figure 5.10: Similarity

5.4.1 Soundness for Mixed Programs

Soundness for interlanguage migration is a relation between the program before and after migration. Intuitively, this relation states that the two programs agree when they both produce values and that the typed module never produces a type error at runtime. We say that a program that has been migrated is *partially typed*.

Definition 5.4.1 (Soundness). *$P \unrhd P^{\tau}$, where P^{τ} is partially typed with typed module τ iff:*

1. *If $P \rightarrow^{*} v$ then there exists v' where $P^{\tau} \rightarrow^{*} v'$ with $v \rhd v'$ or $P^{\tau} \rightarrow^{*}$ $(\textbf{blame}\ g)$ with $g \neq \tau$.*

2. *If $P \rightarrow^{*} (\textbf{blame}\ h)$ then there exists g where $P^{\tau} \rightarrow^{*} (\textbf{blame}\ g)$ with $g \neq \tau$.*

3. *If P reduces forever, then P^{τ} reduces forever or there exists g where $P^{\tau} \rightarrow^{*}$ $(\textbf{blame}\ g)$ with $g \neq \tau$.*

This definition relies on the *similarity* relation $v \rhd v'$, which states that v' is the same as v, with the possible addition of types and casts. In figure 5.10, this relation is defined formally and extended to modules and to programs. For programs, $P \triangleright P^\tau$ states that P and P^τ are syntactically identical, ignoring casts, contracts and types.

We say that a system for typed migration is *sound* if the migrated program is always in the $\trianglerighteq$ relation to the original program.

This captures our intuition as to how typed migration should work: that once we have migrated, we have proven the absence of errors in the typed module. Further, if we get an answer, it is related to the original answer. Since our reduction system tracks where errors occur, we are able to express this statement formally.

5.4.2 Soundness of our system

Proving soundness for our system is a multi-step process. First, we establish that the migrated system agrees with the original one, when errors are ignored. This is established through a simple simulation relation between programs. Second, we define, and prove the correctness of, a transformation called ST that is simpler than MT. Finally, we prove that MT is appropriately related to ST.

For the first of these steps, we make use of the similarity property mentioned above and defined in figure 5.10. This relation between an untyped program (respectively, module or expression) and a partially typed one, states syntactically that the two programs P and P^τ are similar, written $P \triangleright P^\tau$, if they are identical ignoring contracts, types and casts.

Similarity satisfies three lemmas:

Lemma 5.4.2. *If $P \triangleright P^\tau$ and $P \rightarrow^* w$ and $P^\tau \rightarrow^* w'$ then $w \triangleright w'$.*

Lemma 5.4.3. *If $P \triangleright P^\tau$ and P reduces forever then P^τ reduces forever or $P^\tau \rightarrow^*$ $(\textbf{blame}\ f)$ for some f.*

Lemma 5.4.4. *If $P \triangleright P^\tau$ and $P \rightarrow^*$ $(\textbf{blame}\ f)$ for some f then $P^\tau \rightarrow^*$ $(\textbf{blame}\ g)$ for some g.*

Proof Sketch These three lemmas all follow from similar bisimulation arguments. If $e_1 \triangleright e_2$, then there are three possibilities:

1. $e_2 = E[\{c \Leftarrow v\}]$ Then either $e_2 \rightarrow (\textbf{blame}\ f)$ or $e_2 \rightarrow e_2'$ where $e_1 \triangleright e_2'$. This can be seen by simple inspection of the reduction rules for casts.

2. $e_2 = E[(\{(c_1 \dashrightarrow c_2) \Leftarrow^f v\}\ w)]$. Then $e_2 \rightarrow e_2'$ and $e_1 \triangleright e_2'$

3. $e_1 = E[r_1]$ and $e_2 = E[r_2]$ where $r_1 \triangleright r_2$. Then $r_1 \rightarrow r_1'$ and $r_2 \rightarrow r_2'$ where $r_1' \triangleright r_2'$ or $\rho_1 \rightarrow (\textbf{blame}\ f)$ and $r_2 \rightarrow (\textbf{blame}\ g)$. That the hypothesis holds is true from the definition of similarity and the grammar for $E[]$. The fact that the redexes reduce to similar terms or to errors can be seen from inspection of the reduction rules where the redex is not a cast or the application of a (blessed arrow) cast to a value.

Given this, similarity is consistently maintained by reduction, which is all we need for three three lemmas. □

These lemmas are insufficient to establish soundness, since they make no claim about who is blamed for an error. To prove that the typed module is not the one blamed for errors, we introduce a different transformation on typed modules, for which it is possible to prove soundness with conventional techniques. This new transformation, ST, is defined as a sequence of four steps.

1. Choose one module τ from P.

2. Add types to this module, so that is is now a typed module according to the grammar. Call this new module τ'.

3. Apply the transformation and typechecking pass defined in figure 5.11 to the body of τ'. That is, if

   $$\tau' = (\textbf{module}\ \tau\ t\ e)$$

   and $\Gamma \vdash^{ST} \emptyset : e; te'$ then

   $$\tau'' = (\textbf{module}\ \tau\ t\ e')$$

4. Replace τ in P with τ'', producing P^{τ}.

$$\textsc{ST-Var}\quad \Gamma \vdash^{ST} x : \Gamma(x); x \qquad\qquad \textsc{ST-ModVar}\quad \Gamma \vdash^{ST} f : t; \{t \Leftarrow^{f} f\}$$

$$\textsc{ST-ModVarSelf}\quad \Gamma \vdash^{ST} f : t; f \text{ if module } f \text{ has type } t \text{ in } P \qquad\qquad \textsc{ST-Int}\quad \Gamma \vdash^{ST} n : \textbf{int}; n$$

$$\textsc{ST-App}\quad \frac{\Gamma \vdash^{ST} e_1 : (\tau_1 \rightarrow \tau_2); e_1' \qquad \Gamma \vdash^{ST} e_2 : \tau_1; e_2'}{\Gamma \vdash^{ST} (e_1\ e_2) : \tau_2; (e_1'\ e_2')}$$

$$\textsc{ST-If0}\quad \frac{\Gamma \vdash^{ST} e_1 : \tau_1; e_1' \qquad \Gamma \vdash^{ST} e_2 : \tau_2; e_2' \qquad \Gamma \vdash^{ST} e_3 : \tau_2; e_3'}{\Gamma \vdash^{ST} (\textbf{if0}\ \ e_1\ e_2\ e_3) : \tau_2; (\textbf{if0}\ \ e_1'\ e_2'\ e_3')}$$

$$\textsc{ST-Abs}\quad \frac{\Gamma, x : t \vdash^{ST} e : s; e'}{\Gamma \vdash^{ST} (\lambda x : t.e) : (t \rightarrow s); (\lambda x : t.e')}$$

Figure 5.11: Simple Transformation

The transformation of figure 5.11 annotates every module reference in the typed module with a cast to the appropriate type. The key type rules are ST-MODVAR, which adds a cast around a module reference, and ST-MODVARSELF, which handles self-reference to the typed module and does not insert a cast. In the program that results, no module has a contract. These invariants simplify the proof of soundness.

This transformation, like the original, is non-deterministic, thus $ST(P)$, like $MT(P)$, is a set. We must therefore prove that it is sound for *any* set of casts that it might generate.

Lemma 5.4.5 (Soundness of ST)**.** *If $P^{\tau} \in ST(P)$ then $P \trianglerighteq P^{\tau}$.*

Proof Sketch By similarity, if $P \rightarrow^{*} v$ and $P^{\tau} \rightarrow^{*} v'$ then $v \triangleright v'$. Similarly, if P reduces forever, then $P^{\tau} \not\rightarrow^{*} v$ for any v. Therefore, the following lemma, stating that if an error occurs, the blame is assigned to one of the untyped modules, suffices for the proof. □

Lemma 5.4.6 (ST never blames the typed module)**.** *If* $P^\tau \in ST(P)$*, and* $P^\tau \rightarrow^* (\textbf{blame}\ g)$ *then* $g \neq \tau$*.*

Proof Sketch The only way we could ever reduce to (**blame** τ) is if $(n\ v)^\tau$ is the redex or if a cast fails and blames τ. To prove that neither of these happens, we show that the main expression is always *valid*, using the type system of Figure 5.12. Soundness is then implied by lemma 5.4.10. □

Validity implies that there are no applications of numbers where the application is labeled with τ, and also that every cast blaming τ is applied to a term that satisfies that cast.

T-Var

$$\Gamma \vdash^M_{PM} x : \Gamma(x)x$$

T-Cast

$$\Gamma \vdash^M_{PM} \{t \Leftarrow^f e\} : t$$

T-BlessedCast

$$\Gamma \vdash^M_{PM} \{(c_1 \dashrightarrow c_2) \Leftarrow^f v\} : (c_1 \rightarrow c_2)$$

T-Int

$$\Gamma \vdash^M_{PM} n : \textbf{int}$$

T-App

$$\frac{\Gamma \vdash^M_{PM} e_1 : (\tau_1 \rightarrow \tau_2) \qquad \Gamma \vdash^M_{PM} e_2 : \tau_1}{\Gamma \vdash^M_{PM} (e_1\ e_2) : \tau_2}$$

T-If0

$$\frac{\Gamma \vdash^M_{PM} e_1 : \tau_1 \qquad \Gamma \vdash^M_{PM} e_2 : \tau_2 \qquad \Gamma \vdash^M_{PM} e_3 : \tau_2}{\Gamma \vdash^M_{PM} (\textbf{if0}\ e_1\ e_2\ e_3) : \tau_2}$$

T-Abs

$$\frac{\Gamma, x : t \vdash^M_{PM} e : s}{\Gamma \vdash^M_{PM} (\lambda x : t.e) : (t \rightarrow s)}$$

T-TypeMod

$$\Gamma \vdash^M_{PM} f : t \text{ if } (\textbf{module}\ \tau\ t\ e) \in P^\tau$$

Figure 5.12: Mixed Type System

This type system, with judgement $\Gamma \vdash^M_{PM} e_m : t$ for mixed term e_m, allows us to type mixed terms even if they are not originally from the typed module τ. This is key to the subsequent proofs, since we need to verify that both numbers and casted terms have the appropriate types, even if they arose from untyped sources.

There a several rules to note in the type system of figure 5.12. First, the rule T-Cast does not ensure that its argument is well-typed. Therefore, it applies even where the argument is an untyped term, and the cast is protecting the context of the cast from its argument. Second, the T-BLESSEDCAST rule is necessary so that blessed casts can appear during reduction, even though they are not part of the syntax of types. Third, we allow the typed module to be used without a cast in rule T-TYPEMOD. Such module references are still protected from the untyped world, because they are within a typed expression.

We now define two important properties of mixed terms.

Definition 5.4.7. *A mixed term e is* consistent in P^τ *iff* $\emptyset \vdash^M_{PM} e : t$ *for some type* t.

Terms may be consistent even if they do not originate in a typed module, or even if some of their subterms are not consistent. For example, $\emptyset \vdash^M_{PM} \{\textbf{int} \Leftarrow^f (\lambda x.(3\ x))\} : \textbf{int}$ for any f, even though the expression is patently erroneous.

Based on this definition, only some kinds of terms can be consistent: typed abstractions, numbers, casts and applications of two consistent terms.

Definition 5.4.8. *A mixed term e^f is* typed in P^τ *iff f is the name of module τ.*

This definition gives us a handle on those terms that came from the original typed module. These are the terms that must not trigger errors during the execution of the program.

With these definitions, we can define validity, the property that we use for our central lemma. This property ensures both that numbers are not in the operator position of a typed application, and that typed terms satisfy any immediately surrounding casts. Maintenance of these two properties is sufficient to ensure that τ is never blamed. The third portion of the definition states that there is always a syntactic barrier between consistent and inconsistent portions of the expression, with the exception of numbers. This is the mechanism that is central to maintaining the other two during reduction.

Definition 5.4.9. *A mixed term e_m is* valid in program P^M *iff all of the following hold:*

1. *every typed subexpression $d^\tau \in e_m$ is either consistent or or the form $((\lambda x.e)\ e')^\tau$*

2. *for every expression of the form $\{t \Leftarrow^\tau e^\tau\}$, $\emptyset \vdash^M_{PM} e^\tau : t$.*

3. *every consistent term $d \in e_m$ is either a number or the immediate subterm of a consistent term*

Lemma 5.4.10. *If a mixed term e_m is valid in program $P^\tau \in ST(P)$ for some P, and $e_m \rightarrow e'_m$ then e'_m is valid in program P^τ.*

Proof Sketch This proof proceeds by cases on the reduction rule that takes e_m to e'_m. The important cases are SPLIT, SUBST, TYPEDSUBST and LOOKUP-TYPED. We explain how to prove one of these here.

Consider case SPLIT. Then $r = (\{(c_1 \dashrightarrow c_2) \Leftarrow^g v\}\ w)^f$ and $r' = \{c_2 \Leftarrow^g (v\ \{c_1 \Leftarrow^f w\})\}$. We consider the three components of validity in turn.

1. Since the casts and application in r' are new, they cannot be labeled τ. Therefore, typed subexpressions of r' occur only in v and w. If v or w contain typed subexpression, these subexpressions must have been typed in r, and so by hypothesis they are still consistent. If the redex is part of a larger typed expression, then r must have been consistent, and must have had type c_2. But $\vdash^M_{PM} r' : c_2$, so this property is maintained.

2. The application expression in r' is new, and so cannot have label τ. Therefore, we only need to consider the inner cast. If w is typed, then it must have been consistent (since the only other possibility is a redex). Therefore, the whole application must have been consistent, and thus $\vdash^M_{PM} w : c_1$, which is precisely the desired type.

3. Both casts trivially satisfy this case. Thus we have to consider v, w, and the application. Both the application and w are immediate arguments to a cast. If v is consistent, then it must be been a typed abstraction, since it is the argument of a blessed arrow contract, and untyped abstractions are not consistent. If it is a type-annotated abstraction, it must have label τ, as required by the grammar. Thus, by hypothesis, it must satisfy its cast, and have type $(c_1 \rightarrow c_2)$. Therefore, since the operand is a cast to c_2, $\vdash^M_{PM} (v\ \{c_1 \Leftarrow^f w\}) : c_2$.

This concludes the case. The others are proved in a similar way. □

Given the soundness of ST, we can turn to proving soundness for MT. This relies on a relationship between contracts, as stated in the following lemma.

Lemma 5.4.11 (Soundness of the $\Rightarrow$ relation). *If* $\{t \Leftarrow^f v\} \rightarrow^* v'$ *and* $c \Rightarrow t$ *then* $\{c \Leftarrow^f v\} \rightarrow^* v'$.

Proof Sketch By induction on the derivation of $c \Rightarrow t$, either $c = t$ or c contains some disjunction in negative position, where t does not. If $c = t$ then the conclusion trivially follows. Further, by examination of the reduction rules INT-INTOR and INT-LAMOR we note that if $\{c_1 \Leftarrow v\} \rightarrow v$ then $\{c_1 \vee c_2 \Leftarrow v\} \rightarrow v$. Therefore, these additional disjunctions will not introduce new failures that did not exist previously. □

With this, we can now conclude the main theorem of this chapter.

Theorem 5.4.12 (Soundness of the transformation). *If* $P' \in MT(P)$ *then* $P \trianglerighteq P'$.

Proof Sketch Given the soundness of the simple transformation, all we need to prove is that every module variable reference that is not wrapped in a cast reduces to an appropriate value. By the definition of the MT transformation, however, every module reference is to a module to which we have added a contract. And by the lookup rule, that contract is turned into a cast at the point of reference. Therefore, by lemma 5.4.11, and the rules in figure 5.8 by which we add casts to the typed module, the new program still cannot blame the typed module. □

CHAPTER 6

Occurrence Typing

Occurrence typing, originally developed by Komondoor et al. [2005], is the idea of assigning differing types to different occurrences of variables based on the control flow of the program. In Typed Scheme, we use applications of type predicates such *number?* to determine the types of variables, as show in chapter 2. This chapter and the next formalize the ideas of occurrence typing, as well as extensions to paths (section 7.1) and complex logical propositions (section 7.2), in the calculus λ_{TS} .

We begin our presentation of λ_{TS} with the base system. The fundamental judgement of the type system is:

$$\Gamma \vdash e : \tau \; ; \; \phi \; ; \; o$$

It states that in type environment Γ, the expressions e has type τ, following standard practice. Additionally, the judgement describes the filters ϕ of e, i.e., the propositions known to be true depending on whether e evaluates to true or false. Finally, the object o describes the portion of the environment for which e in an accessor.

6.1 Syntax and Operational Semantics

The syntax and semantics of λ_{TS} are standard, except for the types; see figure 6.1. Terms are either boolean constants, numbers, function constants,

$$\begin{array}{lll} d, e, \ldots & ::= \; x \mid (e\ e) \mid (\textbf{if}\ e\ e\ e) \mid v & \text{Expressions} \\ v & ::= \; c \mid \#t \mid \#f \mid n \mid \lambda x : \tau . e & \text{Values} \\ c & ::= \; \textit{add1} \mid \textit{zero?} \mid \textit{number?} & \\ & \quad \mid \textit{boolean?} \mid \textit{procedure?} \mid \textit{not} & \text{Primitive Operations} \\ E & ::= \; [] \mid (E\ e) \mid (v\ E) \mid (\textbf{if}\ E\ e\ e) & \text{Evaluation Contexts} \end{array}$$

$$\begin{array}{lll} \sigma, \tau & ::= \top \mid \mathbf{N} \mid \mathbf{\#t} \mid \mathbf{\#f} \mid \tau \xrightarrow[O]{\upphi} \tau \mid (\bigcup\ \vec{\tau}) & \text{Types} \\ \psi & ::= \tau_x \mid \overline{\tau}_x \mid \bot & \text{Filters} \\ \uppsi & ::= \tau \mid \overline{\tau} \mid \perp\!\!\!\perp & \text{Latent Filters} \\ \phi & ::= \vec{\psi} | \vec{\psi} & \text{Filter Sets} \\ \upphi & ::= \vec{\uppsi} | \vec{\uppsi} & \text{Latent Filter Sets} \\ o & ::= x \mid \emptyset & \text{Objects} \\ O & ::= \bullet \mid \emptyset & \text{Latent Objects} \end{array}$$

Figure 6.1: Syntax of Terms and Types

variables, λ-abstractions or **if** expressions. All of these have the usual operational semantics, which we therefore omit. One distinctive feature of the operational semantics is that values other than $\#f$ are treated as true.

The distinctive features of λ_{TS} are found in the types. A type is either the top type $\top$, the number type **N**, a type representing either of the two constant booleans, a (true) union type or a function type.

There are two distinctive aspects of these types. First, separating the booleans into particular types for true (**#t**) and false (**#f**) is necessary since many common Scheme idioms treat $\#f$ specially. The more typical boolean type is ($\bigcup$ **#t** **#f**), which is abbreviated **B**. Second, function types come with two annotations: a *latent filter set* $\upphi$ and a *latent object* O.

6.1.1 Filters, Latent Filters, and Filter Sets

Filters ψ are propositions about types of variables. The filters proved by a particular expression form part of the type judgement for that expression. A filter of the form τ_x states that x has type τ. The filter $\overline{\tau}_x$ states that x does

$$\textsc{T-Var}\quad \Gamma \vdash x : \Gamma(x)\ ;\ \overline{\mathbf{\#f}_x}|\mathbf{\#f}_x\ ;\ x \qquad\qquad \textsc{T-Const}\quad \Gamma \vdash c : \delta_\tau(c)\ ;\ \epsilon|\bot\ ;\ \emptyset$$

$$\textsc{T-True}\quad \Gamma \vdash \#t : \mathbf{\#t}\ ;\ \epsilon|\bot\ ;\ \emptyset \qquad \textsc{T-False}\quad \Gamma \vdash \#f : \mathbf{\#f}\ ;\ \bot|\epsilon\ ;\ \emptyset \qquad \textsc{T-Num}\quad \Gamma \vdash n : \mathbf{N}\ ;\ \epsilon|\bot\ ;\ \emptyset$$

$$\textsc{T-Abs}\quad \dfrac{\begin{array}{c}\Gamma, x : \sigma \vdash e : \tau\ ;\ \phi\ ;\ o \\ \varphi = \mathit{abstractfilter}(x, \phi) \\ O = \begin{cases} \bullet & \text{if } o = x \\ \emptyset & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash \lambda x : \sigma . e : \sigma \xrightarrow[O]{\varphi} \tau\ ;\ \epsilon|\bot\ ;\ \emptyset}$$

$$\textsc{T-App}\quad \dfrac{\begin{array}{c}\Gamma \vdash e_{op} : \tau_{op}\ ;\ \phi_{op}\ ;\ o_{op} \\ \Gamma \vdash e_a : \tau_a\ ;\ \phi_a\ ;\ o_a \\ \vdash \tau_a <: \tau_f \qquad \vdash \tau_{op} <: \tau_f \xrightarrow[O]{\varphi_f} \tau_r \\ \phi_r = \mathit{applyfilter}(\varphi_f, \tau_a, o_a) \\ o_r = \begin{cases} x & \text{if } O = \bullet \text{ and } o_a = x \\ \emptyset & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash (e_{op}\ e_a) : \tau_r\ ;\ \phi_r\ ;\ o_r}$$

$$\textsc{T-If}\quad \dfrac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1\ ;\ \overrightarrow{\psi_+}|\overrightarrow{\psi_-}\ ;\ o_1 \\ \Gamma + \overrightarrow{\psi_+} \vdash e_2 : \tau_2\ ;\ \phi_2\ ;\ o_2 \\ \Gamma + \overrightarrow{\psi_-} \vdash e_3 : \tau_3\ ;\ \phi_3\ ;\ o_3 \\ \vdash \tau_2 <: \tau \qquad \vdash \tau_3 <: \tau \\ \phi = \mathit{combfilter}(\overrightarrow{\psi_+}|\overrightarrow{\psi_-}, \phi_2, \phi_3)\end{array}}{\Gamma \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \tau\ ;\ \phi\ ;\ \emptyset}$$

Figure 6.2: Core Type Rules

$$\textsc{S-Refl}\quad \dfrac{}{\vdash \tau <: \tau} \qquad \textsc{S-Top}\quad \dfrac{}{\vdash \tau <: \top} \qquad \textsc{S-Fun}\quad \dfrac{\begin{array}{c}\vdash \sigma_a <: \tau_a \qquad \vdash \tau_r <: \sigma_r \\ \varphi' \subseteq \varphi \qquad O = O' \text{ or } O' = \emptyset\end{array}}{\vdash \tau_a \xrightarrow[O]{\varphi} \tau_r <: \sigma_a \xrightarrow[O']{\varphi'} \sigma_r}$$

$$\textsc{S-UnionSuper}\quad \dfrac{\exists i. \vdash \tau <: \sigma_i}{\vdash \tau <: (\bigcup\ \overrightarrow{\sigma}^{i})} \qquad \textsc{S-UnionSub}\quad \dfrac{\overrightarrow{\vdash \tau_i <: \sigma}^{i}}{\vdash (\bigcup\ \overrightarrow{\tau}^{i}) <: \sigma}$$

Figure 6.3: Subtyping

not have type τ. The filter $\bot$ represents impossibility. Latent filters ψ take on analogous forms, but without attached variables, and can be thought of as uninstantiated propositions.

A filter set ϕ is a pair of sequences of filters, which is written as $\overrightarrow{\psi_+}|\overrightarrow{\psi_-}$. The first component, $\overrightarrow{\psi_+}$, collects the propositions that are true if the ex-

$$\begin{aligned}
&\mathit{abstractfilter}(x, \overrightarrow{\psi_+}|\overrightarrow{\psi_-}) = \overrightarrow{\mathit{abo}(x,\psi_+)}|\overrightarrow{\mathit{abo}(x,\psi_-)}\\
&\mathit{abo}(x,\perp) && = \perp\!\!\!\perp\\
&\mathit{abo}(x,\tau_x) && = \tau\\
&\mathit{abo}(x,\overline{\tau}_x) && = \overline{\tau}\\
&\mathit{abo}(x,\tau_y) && = \epsilon \text{ where } x \neq y\\
&\mathit{abo}(x,\overline{\tau}_y) && = \epsilon \text{ where } x \neq y
\end{aligned}$$

$$\begin{aligned}
&\mathit{applyfilter}(\overrightarrow{\psi_+}|\overrightarrow{\psi_-}, \sigma, o) = \overrightarrow{\mathit{apo}(\psi_+,\sigma,o)}|\overrightarrow{\mathit{apo}(\psi_-,\sigma,o)}\\
&\mathit{apo}(\perp\!\!\!\perp,\sigma,o) && = \perp\\
&\mathit{apo}(\overline{\tau},\sigma,o) && = \perp \text{ where } \vdash \sigma <: \tau\\
&\mathit{apo}(\tau,\sigma,o) && = \perp \text{ where } \mathit{no\text{-}overlap}(\sigma,\tau)\\
&\mathit{apo}(\psi,\sigma,\emptyset) && = \epsilon\\
&\mathit{apo}(\tau,\sigma,x) && = \tau_x\\
&\mathit{apo}(\overline{\tau},\sigma,x) && = \overline{\tau}_x
\end{aligned}$$

Figure 6.4: Filter Metafunctions

$$\begin{aligned}
\Gamma + \tau_x, \overrightarrow{\psi} &= (\Gamma, x : \mathit{update}(\Gamma(x),\tau)) + \overrightarrow{\psi}\\
\Gamma + \overline{\tau}_x, \overrightarrow{\psi} &= (\Gamma, x : \mathit{update}(\Gamma(x),\overline{\tau})) + \overrightarrow{\psi}\\
\Gamma + \perp, \overrightarrow{\psi} &= \Gamma' \qquad \text{where } \forall x \in \mathrm{dom}(\Gamma).\Gamma'(x) = \perp\\
\Gamma + \epsilon &= \Gamma
\end{aligned}$$

$$\begin{aligned}
\mathit{update}(\tau,\upsilon) &= \mathit{restrict}(\tau,\upsilon)\\
\mathit{update}(\tau,\overline{\upsilon}) &= \mathit{remove}(\tau,\upsilon)
\end{aligned}$$

Figure 6.5: Environment Operations

pression evaluates to true. The second, $\overrightarrow{\psi_-}$, collects the propositions that are true when the expression evaluates to false. Latent filter sets ϕ are the analogous construct with latent filters as the underlying elements. Empty sequences are written ϵ. When $\perp$ (or the latent form $\perp\!\!\!\perp$) is an element of a sequence of filters, we omit the other elements, since they are irrelevant.

Consider the expression (*number? x*). It has the filter set $\mathbf{N}_x\,|\overline{\mathbf{N}}_x$, which states that if it evaluates to #t, then x is a number, otherwise x is not a number. The function *number?* has the latent filter $\mathbf{N}\,|\overline{\mathbf{N}}$ which states that when applied to a value, if the result is true the value is a number, and if the result is false, the value is not a number.

6.1.2 Objects and Latent Objects

The second piece of additional information computed by the type system is the *object*. It indicates for which part of the dynamic environment the expression is an accessor. In our base calculus, the only such expressions we consider are variable references, which are given the variable itself as the object. Thus, the expression x has object x. The object $\emptyset$ indicates that no information is available about what is accessed.

Abstracting from an object results in a *latent object*, which represents the piece of the environment accessed when a function is applied. The latent object $\bullet$ indicates that whatever the object of the argument to the function is, it is also the object of the result. The function $\lambda x : \tau . x$ has the latent object $\bullet$. The latent object $\emptyset$ indicates the absence of information concerning the result. The constant function $\lambda x : \tau . 1$ has latent object $\emptyset$, as does the function *number?*, whose full type is

$$\top \xrightarrow[\emptyset]{\mathbf{N}|\overline{\mathbf{N}}} \mathbf{B}$$

6.2 Typing Rules

The typing rules for the base system of λ_{TS} are given in figure 6.2 (the *combfilter* metafunction is defined in section 6.4). The simplest rule is T-NUM, which gives all numbers the type **N**. Since numbers are treated as true by the evaluation rules for **if**, numbers have the filter set $\epsilon|\bot$, indicating that no new information is acquired when the number evaluates to a true value, and that if it evaluates to a false value, a contradiction is obtained. The rule for function constants, T-CONST, is similar, but their types are given by the δ_τ meta-function, whose definition is given in figure 6.6. The boolean constants are given singleton types in T-TRUE and T-FALSE, along with filter sets that reflect that #t is always true, and #f is always false.

The rule for typing variables involves additional aspects of the system.

The type of a variable is looked up in the type environment. The object for a variable is itself. Finally, the filter for a variable indicates that if x evaluates to a true value, then x cannot have type **#f**. Similarly, if x evaluates to $\#$f, its type should also be **#f**.

Typechecking an abstraction is straightforward. The body is typechecked in an appropriately-extended environment, producing a type, filter set, and object. The type is the result type of the function, the filter set is abstracted into a latent filter set with the *abstractfilter* metafunction, defined in figure 6.4, and the object is abstracted if it is the formal parameter of the abstraction. The filter set and object of the overall expression is as for any other true value.

In the application rule, T-APP, all of the elements of the type system are combined. In addition to the standard checking of the types of the formal and actual argument, information is propagated from the operator's latent filter set and object and combined with the object of the operand. The *applyfilter* metafunction is (roughly) the inverse operation of *abstractfilter*, and explained further in section 6.4. Similarly, the latent object of the operator is combined with the object of the operand.

Finally, in the T-IF rule, the filter set of the test is divided into its two components, and the first is used for the then branch, the second for the else branch. The operation $\Gamma + \overrightarrow{\psi}$, which combines a type environment with filters to produce a new, refined type environment, is defined in figure 6.5. This makes use of the metafunctions *restrict*(τ, σ) and *remove*(τ, σ), which intuitively compute $\tau \cap \sigma$ and $\tau - \sigma$, respectively. The filter set for the entire expression is computed from the filter sets of the three components by the *combfilter* metafunction, which is the subject of section 6.4.

Subtyping The subtyping relation on types is given in figure 6.3. Most of the rules are straightforward. All types are subtypes of $\top$, and function subtyping proceeds as usual, although strengthening of the filter sets is permitted. The rules for unions are simple, and demonstrate that the union of no types, $(\bigcup)$, is below all other types.

6.3 A Small Example

Returning to example 1, we consider how the type system proves it type safe. The example is (**if** (*number? x*) (*add1 x*) 0), and the challenge is that the second occurrence of x must be given type **N**. We assume that the type environment is $\Gamma = x : \top$. First, we consider the typing of the application (*number? x*). The type of x is $\top$, and its object is just x. Second, the latent filter set of the operand *number?* is $\mathbf{N}\,|\overline{\mathbf{N}}$, and the filter set of the application is thus

$$\textit{applyfilter}(\mathbf{N}\,|\overline{\mathbf{N}}, \top, x) = \mathbf{N}_x\,|\overline{\mathbf{N}}_x$$

To typecheck the whole **if** expression, we take the relevant filter set and add the first part to the type environment for the then branch, giving $x : \top + \mathbf{N}_x = x : \mathbf{N}$. This operation makes use of the *update* and *restrict* metafunctions, defined in figures 6.5 and 6.7. This is sufficient to make (*add1 x*) typecheck correctly, and of course the else expression typechecks under any environment. Therefore, the whole expression has type **N**, as desired, with filter set $\epsilon|\epsilon$ and object $\emptyset$.

6.4 Manipulating Filters

, The metafunctions *applyfilter*, *abstractfilter*, and *combfilter* play a central role in the functioning of the type system, and their definitions determine its expressiveness. The definitions of these metafunctions (and the others defined in this chapter/) are given in a ordered fashion, i.e. each clause in their definition is considered only if the preceding clauses do not apply.

- *abstractfilter*, the simplest, maps a variable *x* and a filter set to a latent filter set. In the case of a filter referring to *x*, it removes the variable, producing a latent filter. When it encounters a filter that refers to a variable distinct from *x*, that filter is discarded.

- *applyfilter* is an inverse to *abstractfilter*. Given a latent filter set (from a function operator) and a type and an object, it combines them to produce a filter set. In most cases, this simply replaces the variable removed by *abstractfilter* with the new variable. In cases where the combination of the type and latent filter are impossible, $\bot$ is produced. For example, the expression (*number?* 3) has filter set $\epsilon|\bot$ because *applyfilter*($\mathbf{N}\,|\overline{\mathbf{N}}, \mathbf{N}, \emptyset$) produces $\bot$ (since $\vdash$ $\mathbf{N}$ $<:$ $\mathbf{N}$). Note that the *no-overlap*(τ, σ) metafunction determines if there are any values that have both type τ *and* σ.

- *combfilter* allows for reasoning about the behavior of complex **if** expressions. The simplest definition for it is

  $$\textit{combfilter}(\phi_1, \phi_2, \phi_3) = \epsilon|\epsilon$$

  With this definition, nothing is known about the result of evaluating a conditional expression. However, more sophisticated definitions can significantly increase the expressiveness of the system. For example, (**if** #f e_2 e_3) is equivalent to e_3. The following rules capture this intuition:

  $$\textit{combfilter}(\epsilon|\bot, \phi_2, \phi_3) = \phi_2$$
  $$\textit{combfilter}(\bot|\epsilon, \phi_2, \phi_3) = \phi_3$$

  Since the compilation of many conditional and boolean operations in Scheme generate instances of **if**, *combfilter* benefits from additional extensions. For example, (**and** e_1 e_2) expands to (**if** e_1 e_2 #f). As seen previously, if an **and** expression produces a true value, then the filter sets of both subexpressions should be combined, but if #f is produced, then nothing is learned. This is captured as follows:

  $$\textit{combfilter}(\overrightarrow{\psi_{1_+}}|\overrightarrow{\psi_{1_-}}, \overrightarrow{\psi_{2_+}}|\overrightarrow{\psi_{2_-}}, \bot|\epsilon) = \overrightarrow{\psi_{1_+}}, \overrightarrow{\psi_{2_+}}|\epsilon$$

$$\begin{aligned}
\delta_\tau(\mathit{number?}) &= \top \xrightarrow[\emptyset]{\mathbf{N}|\overline{\mathbf{N}}} \mathbf{B} \\
\delta_\tau(\ \mathit{procedure?}) &= \top \xrightarrow[\emptyset]{\bot \xrightarrow[\emptyset]{\epsilon|\epsilon} \top | \overline{\bot \xrightarrow[\emptyset]{\epsilon|\epsilon} \top}} \mathbf{B} \\
\delta_\tau(\ \mathit{boolean?}) &= \top \xrightarrow[\emptyset]{\mathbf{B}|\overline{\mathbf{B}}} \mathbf{B} \\
\delta_\tau(\ \mathit{add1}) &= \mathbf{N} \xrightarrow[\emptyset]{\epsilon|\epsilon} \mathbf{N} \\
\delta_\tau(\ \mathit{not}) &= \top \xrightarrow[\emptyset]{\epsilon|\epsilon} \mathbf{B} \\
\delta_\tau(\ \mathit{zero?}) &= \mathbf{N} \xrightarrow[\emptyset]{\epsilon|\epsilon} \mathbf{B}
\end{aligned}$$

Figure 6.6: Constant Typing

As seen in example 13, uses of **or** can be combined into filters that describe union types. Here is an appropriate rule:

$$\mathit{combfilter}(\tau_x|\overline{\tau}_x, \epsilon|\bot, \sigma_x|\overline{\sigma}_x) = (\bigcup\ \tau\ \sigma)_x|\overline{(\bigcup\ \tau\ \sigma)_x}$$

Additional rules are possible to handle **or** in greater generality, as well as other patterns of **if** usage.

$$
\begin{array}{lll}
\mathit{restrict}(\tau, \sigma) & = \bot & \text{if } \mathit{no\text{-}overlap}(\tau, \sigma) \\
\mathit{restrict}((\bigcup\ \overrightarrow{\tau}), \sigma) & = (\bigcup\ \overrightarrow{\mathit{restrict}(\tau, \sigma)}) & \\
\mathit{restrict}(\tau, \sigma) & = \tau & \text{if } \vdash \tau <: \sigma \\
\mathit{restrict}(\tau, \sigma) & = \sigma & \text{otherwise} \\
 & & \\
\mathit{remove}(\tau, \sigma) & = \bot & \text{if } \vdash \tau <: \sigma \\
\mathit{remove}((\bigcup\ \overrightarrow{\tau}), \sigma) & = (\bigcup\ \overrightarrow{\mathit{remove}(\tau, \sigma)}) & \\
\mathit{remove}(\tau, \sigma) & = \tau & \text{otherwise} \\
 & & \\
\mathit{no\text{-}overlap}(\mathbf{N}, \mathbf{B}) & = \mathrm{true} & \\
\mathit{no\text{-}overlap}(\mathbf{N}, \tau \xrightarrow[O]{\phi} \sigma) & = \mathrm{true} & \\
\mathit{no\text{-}overlap}(\mathbf{B}, \tau \xrightarrow[O]{\phi} \sigma) & = \mathrm{true} & \\
\mathit{no\text{-}overlap}((\bigcup\ \overrightarrow{\tau}), \sigma) & = \bigwedge \overrightarrow{\mathit{no\text{-}overlap}(\tau, \sigma)} & \\
\mathit{no\text{-}overlap}(\tau, \sigma) & = \mathrm{true} & \text{if } \mathit{no\text{-}overlap}(\sigma, \tau) \\
\mathit{no\text{-}overlap}(\tau, \sigma) & = \mathrm{false} & \text{otherwise}
\end{array}
$$

Figure 6.7: Type Operations

Chapter 7

Extensions to Occurrence Typing

Having described the basics of occurrence typing and the λ_{TS} calculus, in this chapter we extend the calculus with paths into compound data structures as well as complex logical propositions.

7.1 Adding Paths

The base λ_{TS} calculus deals with predicates applied to variables. To accommodate compound data structures, however, the base system is insufficient. For example, (*number?* (*car x*)) has only a trivial filter set in our base system, because the object for (*car x*) cannot be a variable, so it must be $\emptyset$. In this section, we extend the system to overcome this limitation.

The key concept that allows us to typecheck this is the *path*.[1] A path is simply a sequence of data structure selectors. For example, the expression (*cdr* (*cdr* (*car x*))) follows the path $\mathbf{cdr}, \mathbf{cdr}, \mathbf{car}$ from *x*.

7.1.1 Extending the Language

To model paths and data structure access in λ_{TS} , we extend the language in several ways. Figure 7.1 specifies the extended grammar. The extensions to the expression grammar are minor: a (*cons e e*) form is added, and a

[1] The term is due to Cartwright et al. [Cartwright, Hood, and Matthews 1981] in distantly related work.

$$\begin{array}{lll}
e ::= \ldots \mid (\,\textit{cons}\ e\ e) & & \text{Expressions} \\
v ::= \ldots \mid (\,\textit{cons}\ v\ v) & & \text{Values} \\
c ::= \ldots \mid \ \textit{cons?} \mid \ \textit{car} \mid \ \textit{cdr} & & \text{Primitive Operations} \\
E ::= \ldots \mid (\,\textit{cons}\ E\ e) & & \text{Evaluation Contexts}
\end{array}$$

$$\begin{array}{lll}
\sigma, \tau ::= \ldots \mid \langle \tau, \tau \rangle & & \text{Types} \\
\psi \quad ::= \tau_{\pi(x)} \mid \overline{\tau}_{\pi(x)} \mid \bot & & \text{Filters} \\
\boldsymbol{\psi} \quad ::= \tau_{\pi} \mid \overline{\tau}_{\pi} \mid \perp\!\!\!\perp & & \text{Latent Filters} \\
o \quad ::= \pi(x) \mid \emptyset & & \text{Objects} \\
O \quad ::= \pi \mid \emptyset & & \text{Latent Objects} \\
\pi \quad ::= \overrightarrow{pe} & & \text{Paths} \\
pe \quad ::= \textbf{car} \mid \textbf{cdr} & & \text{Path Elements}
\end{array}$$

Figure 7.1: Grammar Extensions for Paths

pair of values is itself a value. The accessors are *car* and *cdr*, and there is a predicate, *cons?*, for pairs. Similarly, a pair type is added, written $\langle -, - \rangle$

The more significant changes concern filters and objects. A filter with a path $\tau_{\pi(x)}$ means that the portion of x selected by path π has type τ. Similarly, a latent filter includes a path, so that the function

(λ ([*x* **: (Pair Any Any)**]) (*number?* (*car x*)))

has the latent filter set $\mathbf{N}_{\mathbf{car}}\,|\overline{\mathbf{N}}_{\mathbf{car}}$.

Objects also include paths, which describe which portion of the environment is accessed. Latent objects are merely paths, with the singleton latent object $\bullet$ now representing the empty path.

7.1.2 Extending the Type Rules

Adding paths to the system requires a change to all those parts of the type system that manipulate filters and objects. In particular, the T-ABS and T-APP rules change to compute objects with paths. For revised rules, see figure 7.2. We also need new rules for typechecking the use of *cons*, *car* and *cdr*, which are presented in figure 7.3, along with the subtyping rule for pair types. [2]

[2] With a polymorphic type system, these could be simply treated as function applications by the type system.

$$\textsc{T-Abs}\quad \frac{\begin{array}{c}\Gamma, x : \sigma \vdash e : \tau \;;\; \phi \;;\; o \\ \varphi = \mathit{abstractfilter}(x, \phi) \\ O = \begin{cases} \pi & \text{if } o = \pi(x) \\ \emptyset & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash \lambda x : \sigma . e : \sigma \xrightarrow[O]{\varphi} \tau \;;\; \epsilon | \bot \;;\; \emptyset}$$

$$\textsc{T-App}\quad \frac{\begin{array}{c}\Gamma \vdash e_{op} : \tau_{op} \;;\; \phi_{op} \;;\; o_{op} \\ \Gamma \vdash e_a : \tau_a \;;\; \phi_a \;;\; o_a \\ \vdash \tau_a <: \tau_f \qquad \vdash \tau_{op} <: \tau_f \xrightarrow[O]{\varphi_f} \tau_r \\ \phi_r = \mathit{applyfilter}(\varphi_f, \tau_a, o_a) \\ o_r = \begin{cases} \pi'(\pi(x)) & \text{if } O = \pi' \text{ and } o_a = \pi(x) \\ \emptyset & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash (e_{op}\ e_a) : \tau_r \;;\; \phi_r \;;\; o_r}$$

$$\textsc{S-Pair}\quad \frac{\vdash \tau_1 <: \tau_2 \qquad \vdash \sigma_1 <: \sigma_2}{\vdash \langle \tau_1, \sigma_1 \rangle <: \langle \tau_2, \sigma_2 \rangle}$$

Figure 7.2: Modified Type Rules for Paths

$$\textsc{T-Cons}\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \;;\; \phi_1 \;;\; o_1 \\ \Gamma \vdash e_2 : \tau_2 \;;\; \phi_2 \;;\; o_2\end{array}}{\Gamma \vdash (\ \mathit{cons}\ e_1\ e_2) : \langle \tau_1, \tau_2 \rangle \;;\; \epsilon | \bot \;;\; \emptyset}$$

$$\textsc{T-Car}\quad \frac{\begin{array}{c}\Gamma \vdash e : \langle \tau_1, \tau_2 \rangle \;;\; \phi \;;\; o \\ \phi_r = \mathit{applyfilter}(\overline{\#\mathbf{f}}_{\mathbf{car}} | \#\mathbf{f}_{\mathbf{car}}, \langle \tau_1, \tau_2 \rangle, o) \\ o_r = \begin{cases} \mathbf{car}(\pi(x)) & \text{if } o = \pi(x) \\ \emptyset & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash (\ \mathit{car}\ e) : \tau_1 \;;\; \phi_r \;;\; o_r}$$

$$\textsc{T-Cdr}\quad \frac{\begin{array}{c}\Gamma \vdash e : \langle \tau_1, \tau_2 \rangle \;;\; \phi \;;\; o \\ \phi_r = \mathit{applyfilter}(\overline{\#\mathbf{f}}_{\mathbf{cdr}} | \#\mathbf{f}_{\mathbf{cdr}}, \langle \tau_1, \tau_2 \rangle, o) \\ o_r = \begin{cases} \mathbf{cdr}(\pi(x)) & \text{if } o = \pi(x) \\ \emptyset & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash (\ \mathit{cdr}\ e) : \tau_2 \;;\; \phi_r \;;\; o_r}$$

Figure 7.3: New Type Rules for Pairs

$$\mathit{abstractfilter}(x, \overrightarrow{\psi_+}|\overrightarrow{\psi_-}) = \overrightarrow{\mathit{abo}(x,\psi_+)}|\overrightarrow{\mathit{abo}(x,\psi_-)}$$

$$\begin{aligned}
\mathit{abo}(x, \perp) &= \perp\!\!\!\perp \\
\mathit{abo}(x, \tau_{\pi(x)}) &= \tau_{\pi()} \\
\mathit{abo}(x, \overline{\tau}_{\pi(x)}) &= \overline{\tau}_{\pi()} \\
\mathit{abo}(x, \tau_{\pi(y)}) &= \epsilon \text{ where } x \neq y \\
\mathit{abo}(x, \overline{\tau}_{\pi(y)}) &= \epsilon \text{ where } x \neq y
\end{aligned}$$

$$\mathit{applyfilter}(\overrightarrow{\psi_+}|\overrightarrow{\psi_-}, \sigma, o) = \overrightarrow{\mathit{apo}(\psi_+,\sigma,o)}|\overrightarrow{\mathit{apo}(\psi_-,\sigma,o)}$$

$$\begin{aligned}
\mathit{apo}(\perp\!\!\!\perp, \sigma, o) &= \perp \\
\mathit{apo}(\overline{\tau}_\epsilon, \sigma, o) &= \perp \text{ where } \vdash \sigma <: \tau \\
\mathit{apo}(\tau_\epsilon, \sigma, o) &= \perp \text{ where } \mathit{no\text{-}overlap}(\sigma, \tau) \\
\mathit{apo}(\psi, \sigma, \emptyset) &= \epsilon \\
\mathit{apo}(\tau_{\pi'}, \sigma, \pi(x)) &= \tau_{\pi'(\pi(x))} \\
\mathit{apo}(\overline{\tau}_{\pi'}, \sigma, \pi(x)) &= \overline{\tau}_{\pi'(\pi(x))}
\end{aligned}$$

$$\begin{aligned}
\Gamma + \tau_{\pi(x)}, \overrightarrow{\psi} &= (\Gamma, x : \mathit{update}(\Gamma(x), \tau_\pi)) + \overrightarrow{\psi} \\
\Gamma + \overline{\tau}_{\pi(x)}, \overrightarrow{\psi} &= (\Gamma, x : \mathit{update}(\Gamma(x), \overline{\tau}_\pi)) + \overrightarrow{\psi} \\
\Gamma + \perp, \overrightarrow{\psi} &= \Gamma' \qquad \text{where } \forall x \in \mathrm{dom}(\Gamma).\Gamma'(x) = \perp \\
\Gamma + \epsilon &= \Gamma
\end{aligned}$$

$$\begin{aligned}
\mathit{update}(\langle\tau,\sigma\rangle, \upsilon_{\pi::\mathbf{car}}) &= \langle \mathit{update}(\tau, \upsilon_\pi), \sigma\rangle \\
\mathit{update}(\langle\tau,\sigma\rangle, \overline{\upsilon}_{\pi::\mathbf{car}}) &= \langle \mathit{update}(\tau, \overline{\upsilon}_\pi), \sigma\rangle \\
\mathit{update}(\langle\tau,\sigma\rangle, \upsilon_{\pi::\mathbf{cdr}}) &= \langle \tau, \mathit{update}(\sigma, \upsilon_\pi)\rangle \\
\mathit{update}(\langle\tau,\sigma\rangle, \overline{\upsilon}_{\pi::\mathbf{cdr}}) &= \langle \tau, \mathit{update}(\sigma, \overline{\upsilon}_\pi)\rangle \\
\mathit{update}(\tau, \upsilon_\epsilon) &= \mathit{restrict}(\tau, \upsilon) \\
\mathit{update}(\tau, \overline{\upsilon}_\epsilon) &= \mathit{remove}(\tau, \upsilon)
\end{aligned}$$

Figure 7.4: New Metafunctions for Paths

The rule for *cons* is straightforward—it produces a pair type, and, like all other non-#f values, pairs are always true. T-Car and T-Cdr are similar to T-App. Note that the expression (*car x*) has the filter set $\overline{\#\mathbf{f}}_{\mathbf{car}(x)}|\#\mathbf{f}_{\mathbf{car}(x)}$, since if it evaluates to #f, the *car* field of *x* must have type **#f**.

Metafunctions In addition to the new typing rules, several of the metafunctions must also be revised:

- Paths in filters and latent filters are propagated in *abstractfilter* and the environment update operation ($\Gamma + \overrightarrow{\psi}$).
- *update* follows the path to update the relevant type.

- *applyfilter* composes the path of the object and of the filter.

The new definitions are in figure 7.4.

7.1.3 An Example

We can now return to example 18 from section 2 and see how the extended type system typechecks it correctly:

```
(if (number? (car p))
    (add1 (car p))
    7)
```

We begin with the type environment $\Gamma = p : \langle\top, \top\rangle$. Considering the test expression first, the expression *p* has object p. Thus (*car p*) has the object $\mathbf{car}(p)$ and type $\top$. When combined with the latent filter set of *number?*, this results in the filter set $\mathbf{N}_{\mathbf{car}(p)}\,|\overline{\mathbf{N}}_{\mathbf{car}(p)}$ for the test expression.

The type environment for the then branch is therefore $\Gamma + \mathbf{N}_{\mathbf{car}(p)}$, which is p : *update*$(\langle\top, \top\rangle, \mathbf{N}_{\mathbf{car}})$. This gives us the desired environment $p : \langle\mathbf{N}, \top\rangle$ to bless the then branch.

7.2 Using Logic

The second addition enables the type system to reason logically about predicates. Let us return to example 21. At the point where the second (*number? x*) test is performed, the type system must prove that if this expression evaluates to true, *y* cannot be a string. We represent this knowledge with a new kind of filter, $\mathbf{N}_x \supset \overline{\mathbf{S}}_y$. This filter should be read as an implication, that if *x* is a number, *y* is not a string. The extension to the grammar of filters and latent filters is shown in figure 7.5.

Such filters are not immediately applicable to modify the type environment, so they must be kept around until the implication can be discharged. To accomplish this, we extend the type judgment with a proposition environment Δ, which is a set of filters (ψ). The new judgment has the form

$$\begin{array}{llll} \psi & ::= \ldots & \mid \overrightarrow{\psi} \supset \psi & \text{Filters} \\ \psi & ::= \ldots & \mid \overrightarrow{\psi} \supset \psi & \text{Latent Filters} \end{array}$$

Figure 7.5: Grammar Extension for Logical Reasoning

$$\textsc{T-If}\quad \frac{\begin{array}{c} \Delta,\Gamma \vdash e_1 : \tau_1 \;;\; \overrightarrow{\psi_+}|\overrightarrow{\psi_-} \;;\; o_1 \\ \Delta \cup \overrightarrow{\psi_+} \vdash \overrightarrow{\psi'_+} \qquad \Delta \cup \overrightarrow{\psi_-} \vdash \overrightarrow{\psi'_-} \\ \Delta \cup \overrightarrow{\psi_+},\Gamma + \overrightarrow{\psi'_+} \vdash e_2 : \tau_2 \;;\; \phi_2 \;;\; o_2 \\ \Delta \cup \overrightarrow{\psi_-},\Gamma + \overrightarrow{\psi'_-} \vdash e_3 : \tau_3 \;;\; \phi_3 \;;\; o_3 \\ \vdash \tau_2 <: \tau \qquad \vdash \tau_3 <: \tau \\ \phi = \mathit{combfilter}(\overrightarrow{\psi_+}|\overrightarrow{\psi_-}, \phi_2, \phi_3) \end{array}}{\Delta,\Gamma \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \tau \;;\; \phi \;;\; \emptyset}$$

Figure 7.6: If Rule with Logical Environment

$$\textsc{L-Env}\quad \frac{\psi \in \Delta}{\Delta \vdash \psi} \qquad\qquad \textsc{L-MP}\quad \frac{\Delta \vdash \overrightarrow{\psi} \qquad \Delta \vdash \overrightarrow{\psi} \supset \overrightarrow{\psi'}}{\Delta \vdash \overrightarrow{\psi'}}$$

Figure 7.7: Filter Derivation

$$\Delta,\Gamma \vdash e : \tau \;;\; \phi \;;\; o$$

For almost all of the typing rules, this new environment is passed through unchanged. The only exception is the T-If rule, whose redefinition is provided in figure 7.6. There are two important changes. First, the two halves of the filter set of the test, $\overrightarrow{\psi_+}$ and $\overrightarrow{\psi_-}$, are added to the proposition environment for the then and else branches, respectively. Second, these new extended environments are used to derive new propositions, $\overrightarrow{\psi'_+}$ and $\overrightarrow{\psi'_-}$, according to the $\Delta \vdash \psi$ relation, defined in figure 7.7. These new sets of propositions are also used in the typechecking of the then and else branches, respectively.

Our proof system for deriving new propositions is quite simple. The first rule is environment lookup, and the second is just *modus ponens*. This

```
(if (if (number? input) (number? (car extra)) #f)
    (+ input (car extra))
    (if (number? (car extra))
        (+ (string→number input) (car extra))
        0))
```

Figure 7.8: Expansion of Example 22

straightforward system, however, is sufficient for handling all the Scheme idioms described in this thesis. Of course, extending both the grammar of filter propositions and the set of proof rules would add even greater expressivity, which we leave to future work in the event that we discover idioms that need it.

7.2.1 Generating Implications

We have seen how filters using implication are used, but not where they are generated. Since we want to use these filters to express the semantics of complex **if** expressions, the point in the system to extend is the *combfilter* metafunction:

$$\begin{aligned}\mathit{combfilter}((\overrightarrow{\psi_{1_+}}|\overrightarrow{\psi_{1_-}}),(\overrightarrow{\psi_{2_+}}|\overrightarrow{\psi_{2_-}}),\bot|\epsilon) = \\ \overrightarrow{\psi_{1_+}},\overrightarrow{\psi_{2_+}}|(\overrightarrow{\psi_{1_+}}\supset\overrightarrow{\psi_{2_-}}),(\overrightarrow{\psi_{2_+}}\supset\overrightarrow{\psi_{1_-}})\end{aligned}$$

This pattern corresponds to the encoding of **and** using conditionals. In particular, (**and A B**) is encoded as (**if A B** #f). Therefore, the *combfilter* rule considers cases where the third argument corresponds to false. Thus, instead of having no information if an **and** expression is false, we now have two implications: $\overrightarrow{\psi_{1_+}} \supset \overrightarrow{\psi_{2_-}}$ and $\overrightarrow{\psi_{2_+}} \supset \overrightarrow{\psi_{1_-}}$.

Of course, other minor changes are required in the other metafunctions to abstract and apply filters with implication. We omit the laborious details.

$\epsilon\ ;\Gamma_0 \vdash$ (**if** ...) : $\mathbf{N}\ ; \epsilon|\epsilon;\emptyset$

- $\epsilon\ ;\Gamma_0 \vdash$ (**if** (*number? input*) ...) : $\mathbf{B}\ ;\ \Delta_1|\Delta_2\ ;\ \emptyset$
  - $\epsilon\ ;\Gamma_0 \vdash$ (*number? input*) : $\mathbf{B}\ ;\ \mathbf{N}_{input}\ |\overline{\mathbf{N}}_{input}\ ;\emptyset$
    - ...
  - $\mathbf{N}_{input};\Gamma_0\vdash$ (*number?* (*car extra*)) : $\mathbf{B};\ \Delta_3|\overline{\mathbf{N}}_{\mathbf{car}(extra)};\emptyset$
    - ...
  - $\overline{\mathbf{N}}_{input};\Gamma_0\vdash$ #f : **#f**; $\bot|\epsilon;\emptyset$
    - ...
- Δ_1;*input* : **N**, *extra* : (**Pair N A**) $\vdash$ (+ *input* (*car extra*)) : $\mathbf{N}\ ;\epsilon|\epsilon;\emptyset$
  - ...
- $\Delta_2;\ \Gamma_0 \vdash$ (**if** (*number?* (*car extra*)) ...) : $\mathbf{N}\ ;\epsilon|\epsilon;\emptyset$
  - $\Delta_2;\Gamma_0\vdash$ (*number?* (*car extra*)) : $\mathbf{B};\ \Delta_3|\overline{\mathbf{N}}_{\mathbf{car}(extra)};\emptyset$
    - ...
  - $\Delta_2, \Delta_3, \overline{\mathbf{N}}_{input}; \Gamma_2 \vdash$ (+ (*string→number input*) (*car extra*)) : $\mathbf{N};\epsilon|\epsilon;\emptyset$
    - ...
  - $\Delta_2, \overline{\mathbf{N}}_{\mathbf{car}(extra)}; \Gamma_0 \vdash$ 0 : $\mathbf{N};\epsilon|\bot;\emptyset$

  where

$$\begin{array}{lcl}
\Gamma_0 & = & \textit{input} : (\bigcup\ \mathbf{S}\ \mathbf{N}),\ \textit{extra} : (\mathbf{Pair\ A\ A}) \\
\Gamma_1 & = & \textit{input} : \mathbf{S},\ \textit{extra} : (\mathbf{Pair\ N\ A}) \\
 & & \\
\Delta_1 & = & \mathbf{N}_{input}, \mathbf{N}_{\mathbf{car}(extra)} \\
\Delta_2 & = & \mathbf{N}_{\mathbf{car}(extra)} \supset \overline{\mathbf{N}}_{input}, \overline{\mathbf{N}}_{input} \supset \mathbf{N}_{\mathbf{car}(extra)} \\
\Delta_3 & = & \mathbf{N}_{\mathbf{car}(extra)}
\end{array}$$

Figure 7.9: Type Derivation for Expanded Example 22

7.2.2 A Logical Example

We now turn again to example 22. The expansion into the core forms of λ_{TS} is given in figure 7.8. The full type derivation is given in figure 7.9. Here the filter set for the first test expression, expanded into an **if** expression in figure 7.8, in the **cond** is

$$\mathbf{N}_{input}, \mathbf{N}_{\mathbf{car}(extra)} \,|(\mathbf{N}_{\mathbf{car}(extra)} \supset \overline{\mathbf{N}}_{input}, \overline{\mathbf{N}}_{input} \supset \mathbf{N}_{\mathbf{car}(extra)})$$

abbreviated as $\Delta_1|\Delta_2$. The first half is added to the environment for the first expression, and the second half to the environment for the remainder. Then the filter set for the second test expression is

$$\mathbf{N}_{\mathbf{car}(extra)} \,|\overline{\mathbf{N}}_{\mathbf{car}(extra)}$$

Therefore, following the T-IF and L-MP rules, the type system can derive $\overline{\mathbf{N}}_{input}$ from Δ_2 and Δ_3, and therefore *input*: **S**. The type system can thus prove that *input* must be a string in the right hand side of the second clause, meaning that the call to *string→number* is safe.

7.3 Proving Soundness

Next we turn our attention to the type soundness of λ_{TS} . Two issues deserve consideration.

7.3.1 Typing Intermediate Steps

The system as described does not obey the usual subject reduction property.[3] For example, the following program

```
((λ ([x : Any])
   (if (number? x)
       (add1 x)
       0))
 #f)
```

reduces in one step to

```
(if (number? #f)
    (add1 #f)
    0)
```

[3] This problem, with the same solution, is also described in the original paper on Typed Scheme [Tobin-Hochstadt and Felleisen 2008].

T-IfTrue

$$\dfrac{\begin{array}{c}\Delta,\Gamma \vdash e_1 : \tau_1 \;;\; \epsilon|\bot \;;\; o_1 \\ \Delta,\Gamma \vdash e_2 : \tau_2 \;;\; \phi_2 \;;\; o_2 \\ \vdash \tau_2 <: \tau \\ \phi = \textit{combfilter}(\epsilon|\bot, \phi_2, \epsilon|\epsilon)\end{array}}{\Delta,\Gamma \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \tau \;;\; \phi \;;\; \emptyset}$$

T-IfFalse

$$\dfrac{\begin{array}{c}\Delta,\Gamma \vdash e_1 : \tau_1 \;;\; \bot|\epsilon \;;\; o_1 \\ \Delta,\Gamma \vdash e_3 : \tau_3 \;;\; \phi_3 \;;\; o_3 \\ \vdash \tau_3 <: \tau \\ \phi = \textit{combfilter}(\bot|\epsilon, \epsilon|\epsilon, \phi_3)\end{array}}{\Delta,\Gamma \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \tau \;;\; \phi \;;\; \emptyset}$$

Figure 7.10: Auxiliary Type Rules for Subject Reduction Proof

which does not typecheck under the typing rules for λ_{TS} , since *add1* requires a numeric argument. Of course, this untypeable code is never executed, so the program is actually safe. To solve this problem, we add two new typing rules; see figure 7.10.

Since there is no overlap between **N** and **#f**, we have that

$$\Delta,\Gamma \vdash (\textit{number?}\ \#f) : \mathbf{B} \;;\; \bot|\epsilon \;;\; \emptyset$$

Therefore T-IfFalse applies to our problematic example, allowing the type system to ignore the troublesome then branch. With these additional rules, we can proceed to proving the usual subject reduction theorem.

Of course, the original system is still sound. We can also prove that these rules are unnecessary. Any program which typechecks without the new rules also typechecks with them. Therefore, the system without the new rules is sound, since by the soundness theorem for the extended system, no runtime errors are possible.

Further, the existence of the T-IfTrue and T-IfFalse rules demonstrate that Typed Scheme can, in many instances, detect dead code in the program. In many cases, this code is the result of macro expansion, but in the case where user-written code is provably unreachable, Typed Scheme issues a warning.

7.3.2 Relating Substitutions and Filters

As usual, proving subject reduction for λ_{TS} requires a lemma showing that substitution preserves types. However, this is non-trivial for λ_{TS} , since the

$$\frac{\forall x \in \mathrm{dom}(\Gamma) \vdash (\Gamma + \overrightarrow{\psi'})(x) <: (\Gamma + \overrightarrow{\psi})(x)}{\Gamma \vdash \overrightarrow{\psi} < \overrightarrow{\psi'}}$$

$$x : \tau \models \bot \qquad \frac{\vdash \tau'@\pi <: \tau}{x : \tau' \models \tau_{\pi(x)}} \qquad \frac{\vdash \tau'@\pi \not<: \tau}{x : \tau' \models \overline{\tau}_{\pi(x)}}$$

$$\frac{x \neq y}{x : \tau' \models \tau_{\pi(y)}} \qquad \frac{x \neq y}{x : \tau' \models \overline{\tau}_{\pi(y)}} \qquad \frac{x : \tau \models \psi \text{ or } x : \tau' \not\models \psi'}{x : \tau' \models \psi' \supset \psi}$$

Figure 7.11: Relations on Filters

filter set may change in significant ways due to a substitution. Consider the example from section 7.3.1. The filter set for (*number? x*) where *x* has type $\top$ is $\mathbf{N}_x|\overline{\mathbf{N}}_x$. If we apply the substitution $[0/x]$, the resulting expression has filter set $\epsilon|\bot$. If we instead apply the substitution $[\#f\,/x]$, we get the filter set $\bot|\epsilon$. These resulting filter sets do not have a simple relationship either to each other or to the original filter set. Therefore, we must have a way of incorporating the substitution into the relation between filters as well.

To describe this relation, first we define a “more specific than” relation on sequences of filters, $\Gamma \vdash \overrightarrow{\psi_1} < \overrightarrow{\psi_2}$ (with $\overrightarrow{\psi_1}$ more specific), ; see figure 7.11. This relation captures the intuition that a sequence of filters is more specific if it allows more terms to be typed in a given environment.

Second, we define a “models” relation that relates pairs of variables and types to filters. A variable with a type models a filter if it agrees with the types specified by the filter for the relevant variables. Thus $x : \mathbf{N} \models \mathbf{N}_x$, $x : \mathbf{\#f} \not\models \mathbf{N}_x$, and $x : \mathbf{\#f} \models \overline{\mathbf{N}}_x$. Using this relation, we can see that if we substitute a value v of type τ for a variable x in expression e, one half of the filter set of the result is modeled by $x : \tau$, and one is not. Furthermore, the filter set of the resulting term is more specific in the environment without x.

Equipped with this additional machinery, we can state the central lemma about substitution for the subject reduction proof.

Lemma 7.3.1. *If* $\Delta, \Gamma, x : \sigma \vdash e : \tau \; ; \; \overrightarrow{\psi_+}|\overrightarrow{\psi_-} \; ; \; o$

and $\vdash v : \sigma' \; ; \; \phi_0 \; ; \; o_0$

and $\vdash \sigma' <: \sigma$

Then $\Delta,\Gamma \vdash e[v/x] : \tau' \; ; \; \overrightarrow{\psi'_+}|\overrightarrow{\psi'_-} \; ; \; o'$

and $\vdash \tau' <: \tau$

and $x : \sigma' \models \overrightarrow{\psi_+} \quad \Rightarrow \quad \Gamma \vdash \overrightarrow{\psi'_+} < \overrightarrow{\psi_+}$

and $x : \sigma' \models \overrightarrow{\psi_-} \quad \Rightarrow \quad \Gamma \vdash \overrightarrow{\psi'_-} < \overrightarrow{\psi_-}$

and $x : \sigma' \not\models \overrightarrow{\psi_+} \quad \Rightarrow \quad \overrightarrow{\psi'_+} = \bot$

and $x : \sigma' \not\models \overrightarrow{\psi_-} \quad \Rightarrow \quad \overrightarrow{\psi'_-} = \bot$

and either $o = \pi(x) \wedge o' = \emptyset$ *or* $o = o'$

This lemma is proved by induction on the original type derivation.

Given this lemma, we can prove the usual subject reduction and progress theorems in the style of Wright and Felleisen [1994].

CHAPTER 8

Refinement Types

Refinement types are a powerful facility for expressing constraints on existing types, often beyond what the type checker can statically verify. In Typed Scheme, occurrence typing gives us a simple way to add refinement types without involving theorem proving or complex set constraints.

The key idea is that every predicate defines a set, which is the values for which that predicate returns #t. We then consider that set as a type—the refinement type corresponding to that predicate. This chapter formalizes this idea, restricted to just the predicates *even?* and *odd?*. The second portion of this chapter presents an extended example, using refinement types to check that form input is safe for use in SQL statements.

8.1 Formalizing Refinements

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. We restrict refinements to built-in functions so that any refinement type that can be given to an expression can also be given to the value the expression reduces to. We then add two new built-in functions, *even?*, with type

$$(\mathbf{N} \overset{(\mathbf{R}\ \overset{even?}{\rightarrow}\ \mathbf{N})}{} \mathbf{B})$$

and *odd?*, with type

$$(\mathbf{N}\ ^{(\mathbf{R}\ \overset{odd?}{\rightarrow}\ \mathbf{N})}\ \mathbf{B})$$

and the obvious semantics.

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

$$\Sigma \vdash_r \tau <: \tau$$

The new rule for refinement types is

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

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

The addition of the Σ environment to the subtyping judgement requires a similar addition to the typing judgement, which now has the form

$$\Sigma, \Delta, \Gamma \vdash_r e : \tau; \phi; o$$

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

Example 34

(λ ([*f* **:** ((**R** *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.

8.2 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 7.3 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 necessarily have any 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 proof of soundness has been done for an earlier formulation of occurrence typing, and is presented elsewhere [Tobin-Hochstadt and Felleisen 2009].

8.3 An Extended Example

To demonstrate the utility of refinement types as provided by Typed Scheme, we present an extended example, tackling the problem of 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 is 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 puts the function *sql-safe?* into the refinement environment Σ in the formalization of refinement types, with the addition that it changes the type of *sql-safe?* to be a predicate for (**Refinement** *sql-safe?* **String**).

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 "';")))
```

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 is up to the programmer.

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 our style 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.

CHAPTER 9

Variable-Arity Polymorphism[1]

In section 2.7.6, we saw the basics of typing variable-arity polymorphism. The key ingredients are

- Distinguishing uniform from non-uniform variable-arity functions.
- Dotted type variables and dotted pre-types.
- Special handling of *map* on terms with dotted pre-types.
- Handling of *apply*.

This chapter synthesizes those insights into a formal calculus whose type system is able to statically reject programs that misapply both uniform and non-uniform variable-arity functions.

The development of our formal model starts from the syntax of a multi-arity version of System F [Girard 1971], enriched with variable-arity functions. A technical report [Strickland et al. 2008] contains the full set of type rules as well as a semantics and soundness theorem for this model.

9.1 Syntax

We extend System F with multiple-arity functions at both the type and term level, lists, as well as function that accept rest arguments. The use of

[1] This is joint work with T. Stephen Strickland [Strickland et al. 2009].

$$
\begin{array}{lcl}
p & ::= & \texttt{=} \mid \texttt{plus} \mid \texttt{minus} \mid \texttt{mult} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{null?} \\
v & ::= & n \mid b \mid p \mid \texttt{null}_{\tau} \mid (\texttt{cons}_{\tau}\ v\ v) \mid (\lambda\ (\overrightarrow{[x : \tau]})\ e) \mid (\Lambda\ (\overrightarrow{\alpha})\ e) \\
 & \mid & (\Lambda\ (\overrightarrow{\alpha}\ \alpha\ \boldsymbol{\ldots})\ e) \mid (\lambda\ (\overrightarrow{[x : \tau]}\ \texttt{.}\ [x : \tau^{*}])\ e) \mid (\lambda\ (\overrightarrow{[x : \tau]}\ \texttt{.}\ [x : \tau\ \boldsymbol{\ldots}_{\alpha}])\ e) \\
e & ::= & v \mid x \mid (e\ \overrightarrow{e}) \mid (\texttt{if}\ e\ e\ e) \mid (\texttt{cons}_{\tau}\ e\ e) \mid \texttt{error}_{\mathrm{L}} \\
 & \mid & (\texttt{@}\ e\ \overrightarrow{\tau}) \mid (\texttt{@}\ e\ \overrightarrow{\tau}\ \tau\ \boldsymbol{\ldots}_{\alpha}) \mid (\texttt{apply}\ e\ \overrightarrow{e}\ e) \\
 & \mid & (\texttt{map}\ e\ e) \mid (\texttt{ormap}\ e\ e) \mid (\texttt{andmap}\ e\ e) \\
\tau & ::= & \mathbf{Integer} \mid \mathbf{Boolean} \mid \alpha \mid (\mathbf{Listof}\ \tau) \mid (\overrightarrow{\tau} \rightarrow \tau) \\
 & \mid & (\overrightarrow{\tau}\ \tau^{*} \rightarrow \tau) \mid (\overrightarrow{\tau}\ \tau\ \boldsymbol{\ldots}_{\alpha} \rightarrow \tau) \mid (\forall\ (\overrightarrow{\alpha})\ \tau) \mid (\forall\ (\overrightarrow{\alpha}\ \alpha\ \boldsymbol{\ldots})\ \tau)
\end{array}
$$

Figure 9.1: Syntax

multiple-arity functions establishes the proper problem context. Lists and rest-argument functions suffice to explain how both kinds of variable-arity functions interact.

The grammar in figure 9.1 specifies the abstract syntax. We use a syntax close to that of Typed Scheme, including the use of @ to denote type application. The use of the vector notation $\overrightarrow{e}$ denotes a (possibly empty) sequence of forms (in this case, expressions). In the form $\overrightarrow{e_k}^n$, n indicates the length of the sequence, and the term e_{k_i} is the ith element. The subforms of two sequences of the same length have the same subscript, so $\overrightarrow{e_k}^n$ and $\overrightarrow{\tau_k}^n$ are identically-sized sequences of expressions and types, respectively, whereas $\overrightarrow{e_j}^m$ is unrelated. If all vectors are the same size the sizes are dropped, but the subscripts remain. Otherwise the addition of starred pre-types, dotted type variables, dotted pre-types, and special forms is needed to operate on non-uniform rest arguments.

A *starred pre-type*, which has the form τ^* (by analogy to the Kleene star), is used in the types of uniform variable-arity functions whose rest parameter contains values of type τ. It only appears as the last element in the domain of a function type or as the type of a uniform rest argument.

A *dotted type variable*, which has the form $\alpha\ \boldsymbol{\ldots}$, serves as a placeholder in a type abstraction. Its presence signals that the type abstraction can be applied to an arbitrary number of types. A dotted type variable can only appear as the last element in the list of parameters to a type abstraction. We call type abstractions that include dotted type variables *dotted type abstractions*.

A *dotted pre-type*, which has the form $\tau\, \ldots_\alpha$, is a type that is parameterized over a dotted type variable. When a type instantiation associates the dotted type variable α ... with a sequence $\vec{\tau}^n$ of types, the dotted pre-type $\tau\, \ldots_\alpha$ is replaced by n copies of τ, where α in the ith copy of τ is replaced with τ_i. In the syntax, dotted pre-types can appear only in the rightmost position of a function type, as the type of a non-uniform rest argument, or as the last argument to `@`.

In this model the special forms `ormap`, `andmap`, and `map` are restricted to applications involving non-uniform rest arguments, and `apply` is restricted to applications involving rest arguments. In Typed Scheme, of course, the *map*, *ormap*, *andmap* and *apply* functions work on list types in their most general form.

9.2 Type System

The type system is an extension of the type system of System F to handle the new linguistic constructs. We start with the changes to the environments and judgments, plus the major changes to the type validity relation. Next we present relations used for dotted types and expressions that have dotted pre-types instead of types. Then we discuss the changes to the standard typing relation, and finally we discuss the metafunctions used to define the new typing judgments.

The environments and judgments used in our type system are similar to those used for System F except as follows:

- The type variable environment (Δ) includes both dotted and non-dotted type variables.

- There is a new class of environments (Σ), which map non-uniform rest

$$\text{TE-DVAR}\quad \frac{\alpha\ \ldots \in \Delta}{\Delta \vdash \alpha\ \ldots} \qquad \text{TE-DFUN}\quad \frac{\Delta \triangleright \tau_r\ \ldots_\alpha \quad \overrightarrow{\Delta \vdash \tau_j} \quad \Delta \vdash \tau}{\Delta \vdash (\overrightarrow{\tau_j}\ \tau_r\ \ldots_\alpha \rightarrow \tau)} \qquad \text{TE-DALL}\quad \frac{\Delta \cup \{\overrightarrow{\alpha_j}, \beta\ \ldots\} \vdash \tau}{\Delta \vdash (\forall\ (\overrightarrow{\alpha_j}\ \beta\ \ldots)\ \tau)}$$

$$\text{TDE-PRETYPE}\quad \frac{\Delta \vdash \alpha\ \ldots \quad \Delta \cup \{\alpha\} \vdash \tau}{\Delta \triangleright \tau\ \ldots_\alpha}$$

Figure 9.2: Type Validity rules

parameters to dotted pre-types.

- There is also an additional validity relation $\Delta \triangleright \tau\ \ldots_\alpha$ for dotted pre-types.
- The use of Σ makes typing relation $\Gamma, \Delta, \Sigma \vdash e : \tau$ a five-place relation.
- There is an additional typing relation $\Gamma, \Delta, \Sigma \vdash e \triangleright \tau\ \ldots_\alpha$ for assigning dotted pre-types to expressions.

The type validity relation checks the validity of two forms—types and dotted type variables. The additional rules for establishing type validity of non-uniform variable-arity types are provided in figure 9.2, along with an additional relation which checks the validity of dotted pre-types.

When validating a dotted pre-type $\tau\ \ldots_\alpha$, the bound α is checked to make sure that it is indeed a valid dotted type variable. Then τ is checked in an environment where the bound is allowed to appear free. It is possible for a dotted pre-type to be nested somewhere within a dotted pre-type over the same bound:

Example 35

```
(: f (∀ (α . . .)
       ((α . . .α → α) . . .α → (α . . .α → (Listof Integer)))))
(define ((count . fs) . args)
  (map (λ: ([x : Any]) (if x 1 0))
       (map (λ: ([f : (α . . .α → α)]) (apply f args))
            fs)))
```

$$\textsc{TD-Var}\quad \frac{\Sigma(x) = \tau \ldots_{\alpha}}{\Gamma, \Delta, \Sigma \vdash x \triangleright \tau \ldots_{\alpha}} \qquad \textsc{TD-Map}\quad \frac{\begin{array}{c}\Gamma, \Delta, \Sigma \vdash e_r \triangleright \tau_r \ldots_{\alpha} \\ \Gamma, \Delta \cup \{\alpha\}, \Sigma \vdash e_f : (\tau_r \to \tau)\end{array}}{\Gamma, \Delta, \Sigma \vdash (\texttt{map}\ e_f\ e_r) \triangleright \tau \ldots_{\alpha}}$$

Figure 9.3: Typing Rules for Pre-types

To illustrate how such a type might be used, we instantiate this sample type with the sequence of types **Integer Boolean**:

((*Integer* **Boolean** → *Integer*) (*Integer* **Boolean** → **Boolean**)
→ (*Integer* **Boolean** → (**Listof** *Integer*)))

There are two functions in the domain of the type, each of which corresponds to an element in our sequence. All functions have the same domain—the sequence of types; the ith function returns the ith type in the sequence.

The rules in figure 9.3 are the typing rules for the two forms of expressions that have dotted pre-types. The TD-VAR rule just checks for the variable in Σ. The TD-MAP rule assigns a type to a function position. Since the function needs to operate on each element of the sequence represented by e_r, not on the sequence as a whole, the domain of the function's type is the base τ_r instead of the dotted type $\tau_r \ldots_{\alpha}$. This type may include free references to the bound α, however. Therefore, we must check the function in an environment extended with α as a regular type variable.

As expected, most of the typing rules are simple additions of multiple-arity type and term abstractions and lists to System F. For uniform variable-arity functions, the introduction rule treats the rest parameter as a variable whose type is a list of the appropriate type. There is only one elimination rule, which deals with the special form `apply`; other eliminations such as direct application to arguments are handled via the coercion rules.

The type rules in figure 9.4 concern non-uniform variable-arity functions. These functions also have one introduction and one elimination rule. The rule T-ORMAP and its absent counterpart T-ANDMAP are similar to that of TD-MAP in that the dotted pre-type bound of the second argument is al-

T-DAbs

$$\frac{\overrightarrow{\Delta \vdash \tau_k} \qquad \Delta \rhd \tau_r \ldots_\alpha \qquad \Gamma[\overrightarrow{x_k \mapsto \tau_k}], \Delta, \Sigma[x_r \mapsto \tau_r \ldots_\alpha] \vdash e : \tau}{\Gamma, \Delta, \Sigma \vdash (\lambda\ (\overrightarrow{[x_k : \tau_k]}\ .\ [x_r : \tau_r \ldots_\alpha])\ e) : (\overrightarrow{\tau_k}\ \tau_r \ldots_\alpha \rightarrow \tau)}$$

T-DApply

$$\frac{\begin{array}{c}\Gamma, \Delta, \Sigma \vdash e_f : (\overrightarrow{\tau_k}\ \tau_r \ldots_\alpha \rightarrow \tau) \\ \overrightarrow{\Gamma, \Delta, \Sigma \vdash e_k : \tau_k} \qquad \Gamma, \Delta, \Sigma \vdash e_r \rhd \tau_r \ldots_\alpha\end{array}}{\Gamma, \Delta, \Sigma \vdash (\texttt{apply}\ e_f\ \overrightarrow{e_k}\ e_r) : \tau}$$

T-Ormap

$$\frac{\begin{array}{c}\Gamma, \Delta, \Sigma \vdash e_r \rhd \tau_r \ldots_\alpha \\ \Gamma, \Delta \cup \{\alpha\}, \Sigma \vdash e_f : (\tau_r \rightarrow \textbf{Boolean})\end{array}}{\Gamma, \Delta, \Sigma \vdash (\texttt{ormap}\ e_f\ e_r) : \textbf{Boolean}}$$

T-DTAbs

$$\frac{\Gamma, \Delta \cup \{\overrightarrow{\alpha_k}, \beta \ldots\}, \Sigma \vdash e : \tau}{\Gamma, \Delta, \Sigma \vdash (\Lambda\ (\overrightarrow{\alpha_k}\ \beta \ldots)\ e) : (\forall\ (\overrightarrow{\alpha_k}\ \beta \ldots)\ \tau)}$$

T-DTApp

$$\frac{\overrightarrow{\Delta \vdash \tau_j}^n \qquad \overrightarrow{\Delta \vdash \tau_k}^m \qquad \overrightarrow{\beta_k}^m \text{ fresh} \qquad \Gamma, \Delta, \Sigma \vdash e : (\forall\ (\overrightarrow{\alpha_j}^n\ \beta \ldots)\ \tau)}{\Gamma, \Delta, \Sigma \vdash (\texttt{@}\ e\ \overrightarrow{\tau_j}^n\ \overrightarrow{\tau_k}^m) : \textit{td}_\tau(\tau[\overrightarrow{\alpha_j \mapsto \tau_j}^n], \beta, \overrightarrow{\beta_k}^m)[\overrightarrow{\beta_k \mapsto \tau_k}^m]}$$

T-DTAppDots

$$\frac{\overrightarrow{\Delta \vdash \tau_k} \qquad \Delta \rhd \tau_r \ldots_\beta \qquad \Gamma, \Delta, \Sigma \vdash e : (\forall\ (\overrightarrow{\alpha_k}\ \alpha_r \ldots)\ \tau)}{\Gamma, \Delta, \Sigma \vdash (\texttt{@}\ e\ \overrightarrow{\tau_k}\ \tau_r \ldots_\beta) : \textit{sd}(\tau[\overrightarrow{\alpha_k \mapsto \tau_k}], \alpha_r, \tau_r, \beta)}$$

Figure 9.4: Selected Type Rules

lowed free in the type of the first argument. In contrast to uniform variable-arity functions, non-uniform variable-arity functions must be applied with the `apply` function in this calculus.

While T-DTAbs, the introduction rule for dotted type abstractions, follows straightforwardly from the rule for normal type abstractions, the elimination rules are different. There are two elimination rules: T-DTApp and T-DTAppDots. The former handles type application of a dotted type abstraction where the dotted type variable corresponds to a sequence of types, and the latter deals with the case when the dotted type variable corresponds to a dotted pre-type.

$$\begin{array}{lcl}
\boldsymbol{sd}(\alpha_r, \alpha_r, \tau_r, \beta) & = & \tau_r \\
\boldsymbol{sd}(\alpha, \alpha_r, \tau_r, \beta) & = & \alpha \qquad \text{where } \alpha \neq \alpha_r \\
\boldsymbol{sd}((\overrightarrow{\tau_j}\ \tau'_r \boldsymbol{\ldots}_{\alpha_r} \rightarrow \tau), \alpha_r, \tau_r, \beta) & = & \\
\quad (\overrightarrow{\boldsymbol{sd}(\tau_j, \alpha_r, \tau_r, \beta)}\ \boldsymbol{sd}(\tau'_r, \alpha_r, \tau_r, \beta) \boldsymbol{\ldots}_{\beta} \rightarrow \boldsymbol{sd}(\tau, \alpha_r, \tau_r, \beta)) & & \\
\boldsymbol{sd}((\overrightarrow{\tau_j}\ \tau'_r \boldsymbol{\ldots}_{\alpha} \rightarrow \tau), \alpha_r, \tau_r, \beta) & = & \\
\quad (\overrightarrow{\boldsymbol{sd}(\tau_j, \alpha_r, \tau_r, \beta)}\ \boldsymbol{sd}(\tau'_r, \alpha_r, \tau_r, \beta) \boldsymbol{\ldots}_{\alpha} \rightarrow \boldsymbol{sd}(\tau, \alpha_r, \tau_r, \beta)) \qquad \text{where } \alpha \neq \alpha_r & & \\
\boldsymbol{sd}((\forall\ (\overrightarrow{\alpha_j}\ \alpha\ \boldsymbol{\ldots})\ \tau), \alpha_r, \tau_r, \beta) & = & (\forall\ (\overrightarrow{\alpha_j}\ \alpha\ \boldsymbol{\ldots})\ \boldsymbol{sd}(\tau, \alpha_r, \tau_r, \beta))
\end{array}$$

$$\begin{array}{lcl}
\boldsymbol{td}_\tau((\overrightarrow{\tau_j}^{n}\ \tau_r \boldsymbol{\ldots}_{\beta} \rightarrow \tau), \beta, \overrightarrow{\beta_k}^{m}) & = & \\
\quad (\overrightarrow{\boldsymbol{td}_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^{m})}^{n}\ \overrightarrow{\boldsymbol{td}_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^{m})[\beta \mapsto \beta_k]}^{m} \rightarrow \boldsymbol{td}_\tau(\tau, \beta, \overrightarrow{\beta_k}^{m})) & & \\
\boldsymbol{td}_\tau((\overrightarrow{\tau_j}^{n}\ \tau_r \boldsymbol{\ldots}_{\alpha} \rightarrow \tau), \beta, \overrightarrow{\beta_k}^{m}) & = & \\
\quad (\overrightarrow{\boldsymbol{td}_\tau(\tau_j, \beta, \overrightarrow{\beta_k}^{m})}^{n}\ \boldsymbol{td}_\tau(\tau_r, \beta, \overrightarrow{\beta_k}^{m}) \boldsymbol{\ldots}_{\alpha} \rightarrow \boldsymbol{td}_\tau(\tau, \beta, \overrightarrow{\beta_k}^{m})) \qquad \text{where } \alpha \neq \beta & &
\end{array}$$

Figure 9.5: Subst-dots and trans-dots

The T-DTAPPDOTS rule is more straightforward, as it is just a substitution rule. Replacing a dotted type variable with a dotted pre-type is more involved than normal type substitution, however, because we need to replace the dotted type variable where it appears as a dotted pre-type bound. The metafunction *sd* performs this substitution. Selected cases of the definition of *sd* appear in figure 9.5; the remaining clauses perform structural traversals.

The T-DTAPP rule must first expand out dotted pre-types that use the dotted type variable before performing the appropriate substitutions. To do this, it uses the metafunction td_τ on a sequence of fresh type variables of the appropriate length to expand dotted pre-types that appear in the body of the abstraction's type into a sequence of copies of their base types. These copies are first expanded with td_τ and then in each copy the free occurrences of the bound are replaced with the corresponding fresh type variable. Normal substitution is performed on the result of td_τ, mapping each fresh type variable to its corresponding type argument. The interesting cases of the definition of td_τ also appear in figure 9.5.

CHAPTER 10

Implementation[1]

Most aspects of the implementation of Typed Scheme are standard fare for typed programming languages. However, there is one key novelty: the implementation is done entirely in terms of PLT Scheme macros. This implementation choice provides the following advantages:

- Running the program runs the typechecker—there is no separate type checker to run, as there is with many static checkers for Scheme.
- By expanding into untyped PLT Scheme code, integration with untyped code is seamless and requires no translation.
- The typechecker integrates relatively smoothly with the rest of the macro and module system.
- The typechecker can take advantage of existing PLT Scheme infrastructure.

Implementing a typechecker as a macro provides its own challenges. In particular, the system must deal with the existence of other macros, with cross-module interaction, and it must be able to communicate information about the source program to the typechecker, even though the expander is oblivious to the type system.

[1] This is joint work with Ryan Culpepper and Matthew Flatt [Culpepper et al. 2007].

The macro and module system in PLT Scheme is uniquely well-suited for the implementation of Typed Scheme. It contains a multitude of small and large features, whose development has been guided by the goal of supporting research on and development of new languages. Even though they may be of marginal use individually, together they form a comprehensive language implementation framework.

This chapter first introduces the relevant prerequisites of the PLT Scheme macro system (section 10.1), its integration with the module system (section 10.2), as well as how different macros can communicate with each other (section 10.3); Typed Scheme's implementation uses all of these tools. The implementation is first described relative to a single module (section 10.4), and then for the multi-module case (section 10.5).

10.1 Macros

PLT Scheme's macro system is based on the hygienic [Clinger and Rees 1991; Kohlbecker, Friedman, Felleisen, and Duba 1986] **syntax-case** system [Dybvig, Hieb, and Bruggeman 1993], named after the syntactic form it provides for destructuring the syntax of macro occurrences. A distinguishing aspect of this system is its use of a syntax object system, a rich datatype for representing program fragments.

The **syntax-case** system includes procedural macros, which have to major advantages over the more widely know pattern-rewriting macros:

- Procedural macros can perform computation at compile time.
- Procedural macros allow the programmer to detect and report syntax errors. Macro writers can enforce constraints on legal syntax (e.g., that a given list of identifiers must not contain duplicates); detect when those constraints are violated; and report errors in an appropriate, context-specific fashion.

```
(define-syntax (define-getter+setter stx)
  ;; symbol-append : symbol ... → symbol
  (define (symbol-append . syms)
    (string->symbol (apply string-append (map symbol->string syms))))
  (syntax-case stx ()
    [(define-getter+setter name init-value)
     ;; constraint checking:
     (unless (identifier? #'name)
       (raise-syntax-error 'define-get+set "expected identifier" #'name))
     ;; transformation:
     (with-syntax
         ([getter (datum->syntax
                   #'name
                   (symbol-append 'get- (syntax->datum #'name)))]
          [setter (datum->syntax
                   #'name
                   (symbol-append 'set- (syntax->datum #'name) '!))])
       #'(define-values (getter setter)
           (let ([name init-value])
             (values (λ () name)
                     (λ (new-value) (set! name new-value))))))]))
```

Figure 10.1: A **syntax-case** macro

The macro definition in Figure 10.1 demonstrates the major capabilities of **syntax-case** macros. Its purpose is to create procedures that access and update a shared, hidden variable. For example, a programmer can write (*define-getter+setter balance*) to create definitions for *get-balance* and *set-balance!*.

The macro defines a procedural abstraction (*symbol-append*) to help construct names. Within the **syntax-case** clause, the macro checks that the given name is an identifier (a syntax object containing a symbol); otherwise, it raises an error. Then it uses the macro system's *datum->syntax* procedure together with its own *symbol-append* abstraction to construct the names of the getter and setter procedures. This macro breaks hygiene, because the hygiene principle states that introduced names only capture references to the same name that are introduced by the same macro transformation.

10.2 Modules, or You Want it When, Again?

The PLT Scheme module system [Flatt 2002] allows programmers to group definitions, use imports and exports to control the scope of names, and specify the dependencies between modules. The presence of macros complicates the notion of dependence between modules.

In the presence of procedural macros, a compiler must execute parts of a program in order to deal with the remainder of the program. This blurs the line between compilation and execution. In particular, an interpreter may draw the line in a different place than the compiler, requiring programmers to debug their compiled program after they have already debugged their interpreted program. To eliminate this potential for inconsistency, the PLT Scheme module system require explicit module dependencies and, based on these, provides uniform behavior in both interactive and batch-compilation mode.

10.2.1 Split environments

Syntactically, a module declaration contains a module reference specifying the language that the module is written in, the module's name, and a sequence of definitions and expressions. In our examples, the module's name is left implicit, and provided in a comment. In the PLT Scheme implementation, the name is taken from the filename.

```
#lang initial-language ;; module-name
module-contents ···
```

Denotationally, a module consists of two code parts (plus a dependency specification): a compile-time component and a run-time component. The compile-time part consists of the syntax definitions. The run-time part consists of ordinary definitions and expressions.

The compiler keeps separate environments for the compile-time expressions and run-time expressions. If a module defines a procedure as a run-

```
#lang scheme ;; macro-util
(provide check-for-duplicate-identifier)
(define (check-for-duplicate-identifier ids) omitted )

#lang scheme ;; rec
(require (for-syntax macro-util))
(define-syntax (recur stx)
  (syntax-case stx ()
    [(recur name ([var init] ...) . body)
     (begin
       (check-for-duplicate-identifier #'(var ...))
       #'(letrec ([name (λ (var ...) . body)])
           (name init ...)))]))
(define (build-list n f)
  (recur loop ([i 0])
    (if (< i n)
        (cons (f i) (loop (+ i 1)))
        null)))
```

Figure 10.2: Four kinds of references

time value, a macro transformer in the same module cannot *use* that procedure; the binding is unavailable in the compile-time phase. The macro can, of course, expand into code that *refers* to the procedure. Likewise, a binding in the compile-time phase cannot be used in the run-time phase. This phase separation permits the compiler to compile a module without also executing its entire contents.[2]

The two environments yield two kinds of module dependencies and thus two distinct module import forms. The plain **require** form imports bindings into the environment for run-time expressions, and the **for-syntax** variant imports bindings into the environment for compile-time expressions.

Macros bridge the gap between the two phases. The implementation of a macro is a compile-time expression, but the macro definition extends the environment for run-time expressions. To understand this idea, it is

[2] The same name may have (possibly distinct) meanings in both phases simultaneously. For example, modules written in the *scheme* language automatically import all primitive bindings into both phases.

important to distinguish between the notions of macro versus value bindings from the notions of environments compile-time versus run-time expressions.

The modules in Figure 10.2 illustrate the four different possibilities. In the context of the *rec* module, *check-for-duplicate-identifier* is a value binding in the compile-time environment; thus, it is available for use in the body of the **recur** macro definition. Even though *check-duplicate-identifier* is a "compile-time procedure," it is not a macro. In fact, it cannot be used in run-time expressions at all. In contrast, **recur** is a macro binding in the run-time environment. It is bound to a compile-time value, but the binding is available to run-time expressions such as the definition of *build-list*. The occurrence of **syntax-case** refers to a macro binding in the compile-time environment. Finally, the definition of *build-list* creates a value binding in the run-time environment.

Compilation of a module involves executing its dependencies[3] and expanding uses of macros in the module's body. The dependencies include the compile-time part of the module's initial language module, the compile-time part of every module imported with **require**, and both compile-time and run-time parts of every module imported with **for-syntax** inside **require**.

The rules for compilation (and also for invoking a module's compile time part) are as follows:

- For every **require** import, including the initial language module, invoke that module's compile-time part in the same phase.
- For every **for-syntax** import, invoke that module's compile-time and run-time parts in the next higher phase.

If a module is imported twice, once with plain **require** and once with **for-syntax**, the two corresponding invocations of the module are separate. They do not share mutable state. The module system uses phase numbers to dis-

[3] If the module depends on modules that are not already compiled, they are automatically compiled when the dependency is detected.

tinguish the different instances. Finally, a module is only invoked once per phase, per compilation. Multiple modules that depend on a single module in the same phase share a single invocation of that module and its state.

10.2.2 Compilation independence

True separate compilation is impossible in a module system that supports the import and export of macros. Instead, the module system has a principle of compilation independence:

> Compiling a module depends only on the compiled forms of the modules that it (transitively) requires.

This principle has two consequences:

- The compilation of two modules, neither of which transitively requires the other, should produce the same two results no matter which is compiled first, or whether they are compiled in parallel.

- The compilation of a module does not depend on side effects that occurred during the compilation of modules that it transitively requires. This has important implications for the use of side-effects at compile time.

The compiler effectively creates a new store for each module that it compiles. Each compilation gets a new execution of all supporting module code. Since the result of the compilation process is nothing but a body of code, the states of mutable variables and objects created during the compilation process of any module are discarded at the end.

The following pair of modules illustrates the interaction between side-effects and compilation:

```
#lang scheme ;; storage
(define storage '())
(define (add! x) (set! storage (cons x storage)))
(provide storage add!)
```

```
#lang scheme ;; memory
(require (for-syntax storage))
(define-syntax (remember stx)
  (syntax-case stx ()
    [(remember sym)
     (begin (add! (syntax->datum #’sym))
            (with-syntax ([syms storage])
              #‘(begin (display (quote syms))
                       (newline))))]))
(remember a)
(remember b)
```

The first module defines two variables. The second module accesses the variables at compile time, so it imports the first module via **for-syntax**. It defines a **remember** macro that adds a symbol to the remembered list and generates code to print out the updated list of remembered symbols. Then it uses the macro twice. At the end of compiling the *memory* module, the *storage* variable has the value (b a). Executing the *memory* module prints out the lists (a) and (b a), as expected.

Consider the following addition to the program:

```
#lang scheme ;; inspect-storage
(require storage)
(require memory)
(display storage) (newline)
```

When this module is executed, the last line it prints out is (), not (b a), because the *run-time* instance of the *storage* module is distinct from the *compile-time* instance. That is, side-effects do not cross phases.

Now consider this further addition to the program:

```
#lang scheme ;; more
(require memory)
(remember c)
```

When this module is executed, the last line it prints out is (c), not (c b a). The reason that the (**remember** *c*) in *more* prints just (c) is that *more* was compiled with a fresh instance of *storage* (initially the empty list), and because executing the compile-time part of *memory* does not change that

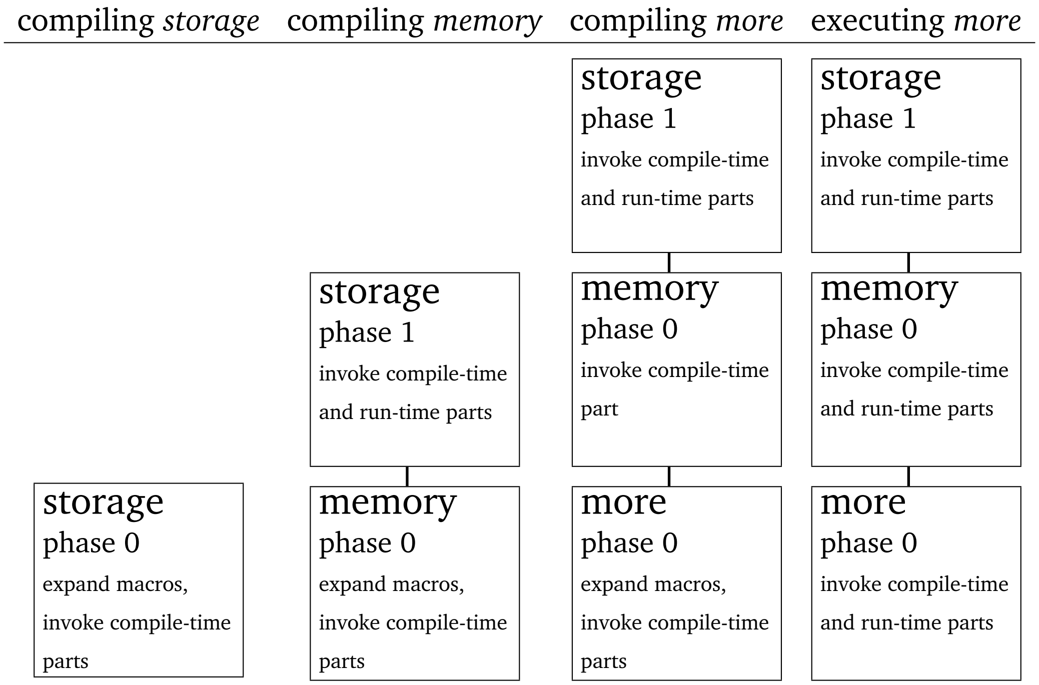
compiling storage
compiling memory
compiling more
executing more
storage
phase 1
invoke compile-time and run-time parts
storage
phase 1
invoke compile-time and run-time parts
storage
phase 1
invoke compile-time and run-time parts
memory
phase 0
invoke compile-time part
memory
phase 0
invoke compile-time and run-time parts
storage
phase 0
expand macros, invoke compile-time parts
memory
phase 0
expand macros, invoke compile-time parts
more
phase 0
expand macros, invoke compile-time parts
more
phase 0
invoke compile-time and run-time parts

Figure 10.3: Module invocations for the execution of *more*

value. The variable is updated during *macro expansion*; the side-effects are not present in the compiled form of *memory*:

```
(compiled-module memory
   (require scheme)
   (require (for-syntax storage))
   (define-syntax (remember stx) omitted )
   (begin (display '(a)) (newline))
   (begin (display '(b a)) (newline)))
```

Figure 10.3 shows all of the module invocations involved in compiling and executing the program *more*. Each box represents a module invocation, and the text at the bottom of each box indicates what parts of the module are executed. Each column represents a shared store; effects in one column are not visible in another column.

The furthest left column simply represents the compilation of *storage*—this module has no **for-syntax** dependencies, and so its compilation triggers no computation in other modules. The second column is the compilation of *memory*, which requires first running the compile-time portions of the *storage* module, since *memory* requires *storage* **for-syntax**, then expanding any macros in the *memory* module. The first two columns are performed since *storage* and *memory* are both dependencies of *more*. Third, the *more* module is compiled. This requires running the compile-time portion of *memory* (which is **require**d by *more*) and therefore the compile- and run-time portions of *storage* (which is **require**d **for-syntax** by *memory*). Finally, the fourth column is the final runtime, which invokes both the compile- and run-time portions of *more* and *memory*, as well as *storage*.

10.2.3 Persistent effects

The compilation rules of the module system require the development of a design pattern for expressing persistent effects. Since compile-time side effects are transient, only the code in the compiled module is permanent. Thus, the way to express a persistent effect is to make it part of the module:

```
#lang scheme ;; memory.v2
(require (for-syntax storage))
(define-syntax (storage-now stx)
  (syntax-case stx ()
    [(storage-here)
     (with-syntax ([syms storage])
       #'(quote syms))]))
(define-syntax (remember stx)
  (syntax-case stx ()
    [(remember sym)
     #'(begin (define-syntax _ (add! (quote sym)))
              (display (storage-now))
              (newline))]))
(remember a)
(remember b)
```

The effect of adding new symbols to the *storage* variable is not executed within the macro, but the macro expander executes the resulting **define-syntax** form when it continues expanding the module body, so the effect of the first addition to the list still occurs before the second **remember** is expanded. This version introduces a helper macro, **storage-now**, to retrieve the value of *storage* after the update.

Since the compile-time part of a compiled module includes all of the macro definitions, the side-effect is preserved:

```
(compiled-module memory.v2
  (require scheme)
  (require (for-syntax storage))
  (define-syntax (storage-now stx) omitted )
  (define-syntax (remember stx) omitted )
  (define-syntax _1 (add! 'a))
  (display '(a)) (newline)
  (define-syntax _2 (add! 'b))
  (display '(b a)) (newline))
```

The calls to *add!* are executed whenever *memory* is required for the compilation of another module. Thus they are executed when *more.v2* is compiled (refer back to Figure 10.3), so the storage is already set to (*b a*) when the use of **remember** in *more* is expanded. Thus, executing the new version of the program prints (*c b a*).

As a matter of readability, the **begin-for-syntax** form accomplishes the same effect as the awkward use of **define-syntax** with a throw-away name. Using **begin-for-syntax** also explicitly signals the programmer's intent to generate an expression that creates a persistent effect.

10.2.4 Local expansion

Some special forms must partially expand their bodies before processing them. For example, primitive forms such as λ handle internal definitions by partially expanding each form in the body to detect whether it is a definition or an expression. The prefix of definitions is collected and transformed into a **letrec** expression with the remainder of the original forms in the body.

Macros can perform the same kind of partial expansion via the *local-expand* procedure, which applies not just to expressions but to entire modules as well.

10.2.5 Compilation-unit hooks

There are two basic compilation scenarios in PLT Scheme. In interactive mode, the compiler receives expressions from the read-eval-print loop. In module mode, the compiler processes an entire module at once. For each mode, the compiler provides a hook so the macro system can be used to control compilation of that body of code.

Top-level transformers The top-level read-eval-print loop automatically wraps each interaction with the **#%top-interaction** macro. By defining a new version of the **#%top-interaction** macro, a programmer can customize the behavior of each interaction.

Module transformers The macro expander processes a module from top to bottom, partially expanding to uncover definitions, **require** and *require-for-syntax* forms, and **provide** forms. It executes syntax definitions and module import forms as it encounters them. Then it performs another pass, expanding the remaining run-time expressions. The module system provides a hook, called **#%module-begin**, that allows language implementations to override the normal expansion of modules.

The module transformer hook is typically used to constrain the contents of the module or to automatically import modules into the compile-time environment. For example, the *scheme* module transformer inserts (**require** (**for-syntax** *scheme*)) at the beginning of the module body, so they automatically get the *scheme* bindings in the compile-time phase.

The module hook technique has been used before in language experimentation. Specifically, Pettyjohn, Clements, Marshall, Krishnamurthi, and Felleisen [2005] prototyped a language for programming web servlets using continuations. This prototype was the first evidence that the module transformer is useful for general-purpose language experimentation.

10.3 Macro protocols

Some language extensions involve not just a single macro definition, but a collection of collaborating macros, or one macro whose multiple uses collaborate. Those collaborating macros need ways to share information at expansion time.

For example, any datatype created with **define-struct** can be recognized and destructured using **match**, as follows:

```
(define-struct posn (x y))

(define (dist-to-origin p)
  (match p
    [(struct posn (a b))
     (sqrt (+ (sqr a) (sqr b)))]))
```

The **define-struct** macro gives **match** access to the names of *posn*'s predicate and accessor functions, and **match** uses those names in the expansion of the pattern to test the value, extract its contents, and bind the results to the pattern variables *a* and *b*.

PLT Scheme provides three mechanisms for compile-time communication between macros: static bindings, side-effects, and syntax properties. Each mechanism fits a particular form of communication.

10.3.1 Static binding

PLT Scheme generalizes **define-syntax** to bind names to arbitrary compile-time data. The definition of the *posn* structure above produces something similar to the following:

```
(begin
  (define-values (make-posn posn? posn-x posn-y) omitted )
  (define-syntax posn
    (list #'make-posn
          #'posn?
          (list #'posn-x #'posn-y))))
```

Despite the use of **define-syntax**, the definition of **posn** is not a macro, as its value is not a transformer procedure. The static information it carries is accessible from other macros (such as **match**) via the *syntax-local-value* procedure.

With static binding, the availability of information is tied to the name it is bound to. Static binding also relies on the ability to define the name; it cannot attach information to a name that is already bound. Still, static binding is the most common mechanism for defining macro protocols in the PLT Scheme libraries, including protocols for structs and component signatures [Culpepper, Owens, and Flatt 2005].

10.3.2 Side-effects

Side-effects are commonly used to provide implicit channels of communication between collaborating run-time components. They are just as capable of providing such channels at compile time for macros, provided the programmer recognizes the difference between ephemeral and persistent effects and uses the appropriate technique.

10.3.3 Syntax properties

Dybvig et al. [1993] define a syntax datatype that extends S-expressions with hygienic binding information and source location tracking. PLT Scheme adds *syntax properties*, key-value pairs of arbitrary associated data, as a way of attaching information to particular terms. By default, syntax properties are simply preserved by macros and primitive syntactic forms, so protocols defined via syntax properties generally do not interfere if they choose distinct keys. Accessing information contained in syntax properties requires only access to the term that carries the property and the key to the property. Syntax properties are available even to observers that cannot access the expansion environment (necessary to access static bindings and compile-time variables).

For these reasons, syntax properties are well-suited to conveying information from macros to code analyzers that examine programs after they have been expanded to core Scheme. For example, DrScheme's Check Syntax tool examines expanded programs to graphically display the program's binding structure. This should work even when the reference is no longer apparent in the residual program, as with the expansion of **match**, which uses the information bound to structure name, although the structure name does not occur in the expansion. The **match** macro leaves a 'disappeared-use syntax property on its expansion telling the Check Syntax tool to color the occurrence of **posn** as a reference and connect it to the corresponding definition.

Macros can introduce and examine syntax properties in their arguments using the *syntax-property* procedure.

10.4 Typing Terms

The implementation of Typed Scheme illustrates like no other language design experiment the power of PLT Scheme's macro system. In this section, we explain the process for type-checking a single definition or expression, with a focus on type annotations and the use of type environments. The following section extends the implementation to handle modules.

Typed Scheme is designed to interoperate with PLT Scheme's existing macro and module systems. In particular, typed programs should be able to use existing macros (provided they produce typecheckable code) and define and use new macros. Since it is generally impossible to derive type rules for arbitrary macros, the type-checker must analyze the program after expansion has eliminated all occurrences of macros and reduced the program to core syntax. This is also the strategy adopted by the ACL2 theorem prover for Common Lisp [Kaufmann et al. 2000].

```
#lang scheme ;; typed-scheme
(provide (rename-out top-interaction #%top-interaction))
(define-syntax (top-interaction stx)
  (syntax-case stx ()
    [(top-interaction . term)
     (let ([expanded-term
            (local-expand #'term 'top-level null)])
       (type-check-top-level expanded-term)
       expanded-term)]))
omitted
```

Figure 10.4: *typed-scheme* module

The type-checker hooks into the compilation process as a macro using the **#%top-interaction** interface described in section 10.2.5. The type-checking macro receives the original unexpanded program, and it calls *local-expand* to fully expand the program for analysis. The type-checker then either approves the expanded program or raises an error, aborting compilation. Figure 10.4 shows the beginning of the *typed-scheme* language module.

The type-checker has rules for each primitive syntactic form. It knows how to assign types to Scheme constants. It also knows the types of the Scheme primitive operators. When it encounters a programmer-introduced variable, however, it needs to find the type of the variable, and although that information is present in the original program, type information is not part of fully expanded, core Scheme code. The rest of this section discusses the treatment of variable types and the communication between Typed Scheme's binding forms and its type-checker across macro expansion.

10.4.1 Variables

Typed Scheme requires type annotations on binding occurrences of many variables; type-checking depends on that information. Consequently, the typed binding forms and the type-checker employ a protocol regarding the communication of variable types.

Type annotations are local to the terms where they appear. They must be robust in the face of local expansion and re-expansion. Since the type-checker works on the fully-expanded program, it makes sense to put the type annotations into the program. At the same time, the result of expansion is a core Scheme term, and Scheme's primitive syntactic forms are unaware of types and do not accept Typed Scheme's typed binding syntax. Syntax properties provide an appropriate method for implementing the protocol by attaching type information to terms.[4]

> **The Variable Protocol:** Every typed binding form decorates its declared variables with a type attached to the 'type-label syntax property of the bound identifiers.

Typed Scheme implements the variable protocol by defining typed binding forms such as λ: as macros that convert the [*variable* **:** *type*] variable syntax into primitive binding forms with the types attached to the 'type-label syntax property of the variable names. Figure 10.5 shows the definition of typed binding macros. The λ: macro expands into the primitive λ form. For each formal parameter name, it creates a new syntax object with a 'type-label property holding the type. Likewise, the **define:** macro handles typed definitions. The first clause handles the simple case with just a name being bound to a value. The second clause handles the function definition syntax by desugaring it to a **define:** form with an explicit λ: form. It also synthesizes the function type from the argument types and the result type, adding it to the expanded definition. The **:** annotation form, used chapter 2, works similarly to **define:**. In this section, we focus on **define:** for simplicity.

The type-checker, at the other end of the protocol, consumes the syntax properties produced by the typed binding forms. When the type-checker encounters a binding form, it scans the bound variables and extracts their types with the *get-id-type* procedure:

[4] Clinger and Hansen [1994] present an alternative method for communicating this information through phases of a compiler.

```
(define-syntax (λ: stx)
  (syntax-case stx (:)
    [(λ: ([formal : formal-type] ...) . body)
     (with-syntax ([(typed-formal ...)
                    (map
                     (λ (id type)
                       (syntax-property id 'type-label type))
                     (syntax->list #'(formal ...))
                     (syntax->list #'(formal-type ...)))])
       #'(λ (typed-formal ...) . body))]))

(define-syntax (define: stx)
  (syntax-case stx (:)
    [(define: var : type expr)
     (identifier? #'var)
     (with-syntax ([tvar (syntax-property #'var 'type-label #'type)])
       #`(define #,tvar expr))]
    [(define: (f [formal formal-type] ...) : result-type . body)
     #'(define: f : (formal-type ... → result-type)
         (λ: ([formal formal-type] ...) . body))]))
```

Figure 10.5: Typed definition and binding forms

```
;; get-id-type : identifier → type
(define-for-syntax (get-id-type id)
  (let ([type (syntax-property id 'type-label)])
    (unless type (raise-missing-type-error id))
    type))
```

The type-checker maintains a two-part type-environment. One part holds the types of global variables, including variables defined via **define:** and all primitive variables. The other part holds the lexical variables, such as those bound by λ: and other local binding forms. Figure 10.6 shows the outline of the environment module. The *declare-type!* operation updates the global type environment; *extend-env* extends the local type environment; and *lookup-env* finds the type of an identifier, searching first the local bindings then the global bindings.

The type-checker consumes the information attached to bound variables. Figure 10.7 lists the code for the type-checker. When the type-checker en-

```
#lang scheme ;; env
(provide (all-defined-out))

;; An environment is a (list-of binding).

;; A binding is (make-binding identifier type).
(define-struct binding (id type))

;; the-type-env : environment
;; Associates global variables with their types.
;; Initially contains types for the scheme primitives.
(define the-type-env omitted)

;; declare-type : identifier type → void
;; Add a type association to the global type environment.
(define (declare-type! id type) omitted)

;; empty-env : environment
;; The empty lexical environment.
(define empty-env null)

;; extend-env : environment (list-of binding) → environment
(define (extend-env env bindings) omitted)

;; lookup-type : lexical-env identifier → type
;; Searches the lexical environment, then the global environment.
(define (lookup-type env var) omitted)
```

Figure 10.6: Type Environment

counters a definition, it extracts the type annotations from the bound identifiers and extends the type environment with the new type association. It finally checks that the declared type matches the type computed for the right-hand side expression. When the type-checker encounters an expression, it switches to expression mode.

The *type-check-expr* procedure computes the type of the expression. In the simplest case, variable reference, the type-checker just looks up the type in the type environment. If the variable is not present, the *lookup-env* procedure raises an error. When the type-checker sees a λ form, it gathers the types of the bound variables and extends the type environment before checking the body in the extended environment. It also uses the types of the formals, in addition to the computed type of the body, to create the type of the function. Finally, the application case involves finding the type of the operator, verifying that it is a function type of the right arity, and checking the expected parameter types against the actual parameter types. If the application is valid, the result is the function's result type.

10.5 Typing Modules

Type-checking a typed module is more complicated than type-checking an isolated definition or expression. Module bodies may refer to variables that are neither primitive nor locally-defined, but imported from other modules. Furthermore, module exports must be protected from misuse in other modules, both typed and untyped.

As with a single definition or expression, type-checking a module involves fully expanding the contents of the module and then analyzing the result. Typed Scheme uses the module transformer hook to type-check the contents of the module.

The variable protocol handles variables whose definitions or bindings occur within the body of the module, but typing imported variables requires additional communication between typed modules. The revised protocol affects the way a typed module's exports are compiled.

There are three kinds of module interactions that typed modules can participate in:

1. A typed module requires an untyped module.
2. A typed module requires another typed module.
3. An untyped module requires a typed module.

The first case simply requires a method of importing untrusted code in such a way that it cannot break the type system's invariants, which demands appropriate input from the programmer. The other two cases determine the behavior of a typed module's exports. Those two cases essentially demand different behaviors from a typed module depending on its use context.

This section explains how Typed Scheme interacts with the module system. We begin with the simplest case, a typed module importing untyped code. This case can be explained in terms of just the import statement. Then we consider the case of a typed module importing another typed module, and we develop the basic typed-module framework. Finally, we show how to extend the behavior of exports to support the case of importing a typed module into an untyped context.

10.5.1 Untyped to Typed

Typed modules cannot use untyped modules without additional protection.[5] Instead, typed modules use a special **require/typed** form to import names at specific types. The **require/typed** form wraps the untyped imports with contracts [Findler and Felleisen 2002] that enforce the supplied types via runtime checks. It also adds the name to the type environment with the specified type.

For example, the following use of **require/typed** imports the *find-files* procedure from a standard library module:

```
(require/typed scheme/file
  [find-files ((Path → Boolean) Path → (Listof Path))])
```

It is equivalent to the following code fragment:

```
(require (rename-in scheme/file unsafe-find-files find-files))TRUST
(define: find-files : ((Path → Boolean) Path → (Listof Path))
  (contract (type->contract
              ((Path → Boolean) Path → (Listof Path)))
            unsafe-find-files
            'find-files
            '<typed-scheme>)TRUST)
```

[5] However, typed modules can safely import untyped *macro* libraries (such as **match**) if the macros do not expand into untyped, non-primitive variables.

The TRUST annotation indicates a syntax property that directs the type-checker to accept the labeled expression as-is. The **contract** expression wraps the unsafe version of the *find-files* procedure with a contract derived from the given type. The last two arguments indicate the parties involved in the contract; if something goes wrong, one of the parties is blamed.

The *find-files* contract checks the procedure's arguments and result. If the untyped version of *find-files* returns a non-path result, the contract catches it and blames 'find-files before the faulty value can interfere with the typed program. The first argument contract is itself a higher-order contract, so the contract system wraps the function passed to *find-files* with a contract corresponding to the (**Path** → **Boolean**) type. This contract prevents the untyped *find-files* from calling the function with faulty arguments; if it does so, the contract system raises an error and blames 'find-files for the violation. The second argument contract is a first-order contract. It can only be violated if typed code supplies an argument of the wrong type, which cannot happen if the type system is sound. Finally, if *find-files* were to return something other than a list of paths, the contract system would stop the program and thus protect the typed code that expects to process the result.

10.5.2 Typed to Typed

Typed Scheme installs a **#%module-begin** macro that first performs the normal module expansion (using *local-expand*), analyzes the result, and produces a module body that follows a new *module variable protocol*, which provides the type-checker with the types of module variables:

```
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(module-begin form ...)
     (type-check-module-body
      (local-expand #'(#%plain-module-begin form ...)
                    'module-begin
                    null))]))
```

Unlike the type-checking procedure for top-level forms, *type-check-module-body* not only type-checks the module body; it also transforms the code to produce the module body.

When one typed module requires another typed module, type-checking the first module requires knowing the types associated with the all of the definitions of the second module. The type-checker needs the types for all of the definitions, even the unexported ones, because an imported macro can expand into references to the unexported variables of the module it was defined in. This requires a new protocol, the module variable protocol.

Let us consider the protocol mechanisms introduced in section 10.3. An imported identifier does not carry any syntax properties, so syntax properties alone are insufficient. Static binding provides a partial solution: instead of directly providing a variable, a typed module could instead provide a macro that expands into a use of the actual variable. The macro would place a type annotation on the reference as a syntax property. The problem with the static binding approach is that it annotates only the references that cross the public import/export boundary. Variable references introduced by imported macros, however, do not go through the static binding mechanism; they refer directly to the module variables. Since Typed Scheme aims to support macros, static binding is not a viable approach.

That leaves compile-time side effects. We extend the type environment table to include all known typed-module definitions instead of just primitives and local definitions. A typed module relies on the global type environment to contain types for all variables that appear within its body, and it guarantees that its client modules have access to its own type associations.

> **The Module Variable Protocol:** During the compilation of a typed module, the global type environment contains bindings for all definitions in all typed modules transitively required by the module being compiled.

Since a module's contributions to the global type environment need to be present during the compilation of every module that depends on it, we use the persistent effect pattern described in section 10.2.3. In addition to verifying the correctness of the module's contents, the *type-check-module-body* procedure also appends compile-time type declarations to the end of the module. We illustrate the effect of the module transformer on the following modules:

```
#lang typed-scheme ;; one
(provide one)
(: one Number)
(define one 1)

#lang typed-scheme ;; plus
(provide plus1)
(: plus1 (Number → Number))
(define (plus1 n)
  (+ n one))
```

The first module passes the type-checker, which also adds a type declaration for *one* to the end of the compiled module:

```
(compiled-module one
  (require typed-scheme)
  (provide one)
  (define one 1)
  (begin-for-syntax
    (declare-type! #'one (type Number))))
```

The reference to *declare-type!* was inserted by a macro from the *typed-scheme* module. Even though *one* does not import the *env* module directly, the procedure is available indirectly through *typed-scheme*. Since *typed-scheme* imports *env* via **for-syntax**, it is correct to use *declare-type!* within the compile-time part of *one*.

When the compiler encounters the *plus* module, the module system invokes the compile-time part of *typed-scheme*, initializing the global type environment with the primitive bindings only. Then, when the compiler encounters the import of *one* in the module body, it invokes the compile-time part of the *one* module, which loads its type declaration for *one* into the type environment.

The *plus* module includes just one new definition, and the module transformer adds the corresponding declaration to the module:

```
(compiled-module plus
  (require typed-scheme)
  (provide plus)
  (define plus (λ (n) (+ n 1)))
  (begin-for-syntax
    (declare-type! #’plus (type (Number → Number)))))
```

The two modules are able to communicate using *typed-scheme*’s type environment because the compile-time parts of the *one* module and the *plus* module share a single invocation of *typed-scheme* and thus a single invocation of the *env* module.

Figures 10.8 and 10.10 show the implementation of typed modules and the module variable protocol.

10.5.3 Typed to Untyped

When a typed module is imported into another typed module, it must provide its definitions and load the type declarations into the global type environment. The type-checker ensures that the exported values are used safely, so there is no need for run-time checking or wrapping.

In contrast, when a typed module is imported into an untyped module, it should protect its exports so that the untyped context cannot destroy the type invariants. As in the “untyped to typed” case, we use contracts to enforce the type constraints of the definitions. For any defined variable, it is a simple matter to generate a definition that wraps the variable in the protection of the appropriate contract. For example, the *plus* module above has a *plus1* procedure with type (**Number** → **Number**). Given that information, we can generate *defensive-plus1*:

```
(define/contract defensive-plus1
    (type->contract (Number → Number))
  plus1)
```

The **define/contract** form is like a definition that uses **contract** explicitly, except that it automatically computes the blame parties.

A typed module, then, needs to provide one set of definitions to typed contexts and another set of definitions to untyped contexts. Of course, no module can actually change the contents of its **provide** clauses once it is compiled. Instead, it can provide a set of *indirection* macros that choose whether to expand into the trusting or defensive versions of exported names, assuming the macros can determine whether the importing context is typed or untyped. PLT Scheme provides *rename transformers* as a convenient way of writing such identifier-to-identifier translations.

Continuing the *plus* module example, the module transformer rewrites

```
(provide plus1)
```

into the following indirection definition and renamed-provide clause:

```
(define-syntax export-plus1
  (if omitted ;; Will it be used in a typed context?
      (make-rename-transformer #'plus1)
      (make-rename-transformer #'defensive-plus1)))
(provide (rename export-plus1 plus1))
```

The indirection definitions depend on some way of determining whether the context they are imported into is typed or untyped. The context that matters is the main module currently being compiled. If the require chain includes intervening modules, they have already been compiled, and references within the compiled modules are already resolved to the right version of the exports. Thus, the problem boils down to determining whether the main module currently being compiled is a typed module.

The property that distinguishes a typed module is that it specifies *typed-scheme* as its language module, and thus its module body is under the control of the typed module transformer. Given that, it is critical to understand the exact order of events in the compilation process:

1. The compiler invokes the initial language module's compile-time part.[6]

2. Then, it executes the initial language module's module transformer on the body of the module being compiled.

3. As the compiler encounters **require**s in the module's body, it invokes the compile-time parts of the relevant modules.

In particular, the execution of the module transformer precedes the execution of any of the indirection definitions in compiled typed modules. The Typed Scheme module transformer can therefore set a flag indicating that the module being compiled is a typed module, and the indirection definitions can simply check the value of the flag. Figure 10.9 presents the modified *typed-scheme* module.

The *type-check* module also adds (**require** *context*) so that the indirection definitions it inserts can refer to *typed-context?*.

The following program illustrate how the flag works. We add an untyped *main* module to the *one* and *plus* modules from our earlier examples.

```
#lang typed-scheme ;; one
(provide one)
(: one Number)
(define one 1))

#lang typed-scheme ;; plus
(require one)
(provide plus1)
(: plus1 (Number → Number))
(define (plus1 x)
  (+ x one)))

#lang scheme ;; main
(require plus)
(display (plus1 41)) (newline)
```

[6] Although this invocation occurs prior to any compilation of a typed module, it cannot be used to determine whether compilation is occurring in a typed context, since the Typed Scheme module can be required from untyped as well as typed modules.

The compiler processes the typed *one* module first, creating the context-dependent indirection definition for the exported variable *one*. When the compiler encounters the typed *plus* module, it first invokes the compile-time part of *typed-scheme*. That, in turn, causes the invocation of the *context* module, including a new *typed-context?* box initialized to false. Executing the Typed Scheme **#%module-begin** macro sets the value in the *typed-context?* box to true. Subsequently, when the compiler encounters the (**require** *one*) form in the module body, it invokes *one*'s compile-time part. Since the *typed-context?* variable is set to true, the indirections are set to the typed variants, and the compiler resolves uses of the imported names to the unwrapped definitions.

The compilation of the *main* module proceeds differently. When the compiler encounters the (**require** *plus*) form, it invokes *plus*'s compile-time part, which invokes *typed-scheme*'s compile-time part and invokes *context*. This creates a fresh *typed-context?* box initialized to false, just as before. The box's value is never changed to true, however, because Typed Scheme's **#%module-begin** macro is not used in the expansion of the *main* module. Thus when *plus*'s indirection definitions are executed, they point to the contract-wrapped variants. Thus the occurrence of *plus1* in the *main* module is wrapped in code to verify the type of its argument.

```
;; type-check-top-level : syntax → void
(define-for-syntax (type-check-top-level form)
  (syntax-case form (define)
    [(define var expr)
     (let* ([var-type (get-id-type #'var)])
       (declare-type! #'var var-type)
       (let ([expr-type (type-check-expr #'expr empty-env)])
         (check-type var-type expr-type form)))]
    [expr
     (type-check-expr #'expr empty-env)]))

;; type-check-expr : syntax lexical-env → type
(define-for-syntax (type-check-expr expr env)
  (syntax-case expr (λ #%app omitted)
    [var
     (identifier? #'var)
     (lookup-type env #'var)]
    [(λ (formal ...) body)
     (let* ([formal-types
             (map get-id-type (syntax->list #'(formal ...)))]
            [formal-bindings
             (map make-binding
                  (syntax->list #'(formal ...))
                  formal-types)]
            [body-type
             (type-check-expr #'body
                              (extend-env env formal-bindings))])
       (make-function-type formal-types body-type))]
    [(#%app op arg ...)
     (let ([op-type (type-check-expr #'op env)]
           [arg-types
            (map (λ (arg) (type-check-expr arg env))
                 (syntax->list #'(arg ...)))])
       (check-function-type op-type #'op)
       (check-types (function-type-params op-type)
                    op-types
                    expr)
       (function-type-result op-type))]
    omitted))
```

These functions are defined using **define-for-syntax**, which creates a value binding in the compile-time environment, so the **top-interaction** macro can use the procedures.

Figure 10.7: The type-checker

```
#lang scheme ;; typed-scheme
(require (for-syntax type-check))
(provide (rename-out module-begin #%module-begin)
         (rename-out top-interaction #%top-interaction)
         (except-out (all-from-out scheme)
                     #%module-begin #%top-interaction)
         define:
         λ:)
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(module-begin form ...)
     (type-check-module-body
      (local-expand #'(#%plain-module-begin form ...)
                    'module-begin
                    null))]))
(define-syntax top-interaction omitted )
(define-syntax define: omitted )
(define-syntax λ: omitted )
```

Figure 10.8: The *typed-scheme* module

```
#lang scheme ;; context
(provide typed-context?)
;; typed-context? : (box-of boolean)
;; True when the module being compiled is a typed module.
(define typed-context? (box #f))

#lang scheme ;; typed-scheme
omitted
(require (for-syntax context))
(define-syntax (module-begin stx)
  (syntax-case stx ()
    [(module-begin form ...)
     (begin
       (set-box! typed-context #t)
       (type-check-module-body
        (local-expand #'(#%plain-module-begin form ...)
                      'module-begin
                      null)))]))
omitted
```

Figure 10.9: Modified *typed-scheme* module

```
#lang scheme ;; type-check
(require env)
(provide (all-defined-out))
;; type-check-top-level : syntax → void
(define (type-check-top-level form) omitted )
;; type-check-module-body : syntax → syntax
(define (type-check-module-body form)
  (syntax-case form ()
    [(module-begin top-level-form . . . )
     (let ([def-types
            (get-definition-types (syntax->list #'(top-level-form . . . )))])
       (for ([def def-types])
         (declare-type! (binding-id def) (binding-type def)))
       (for-each type-check-module-level-form
                 (syntax->list #'(top-level-form . . . )))
       ;; Generate declarations to reload types into the
       ;; global type environment
       (with-syntax ([(type-declaration . . . )
                      (map binding->type-declaration def-types)])
         #'(module-begin top-level-form . . . type-declaration . . . )))]))
;; type-check-module-level-form : syntax → void
(define (type-check-module-level-form form) omitted )
;; type-check-expression : syntax environment → type
(define (type-check-expression expr env) omitted )
;; get-definition-types : (list-of syntax) → (list-of binding)
(define (get-definition-types forms)
  (if (null? forms)
     null
     (syntax-case (car forms) (define)
       [(define name rhs)
        (cons (make-binding #'name (get-id-type #'name))
              (get-definition-types (cdr forms)))]
       [_ (get-definition-types (cdr forms))])))
;; get-id-type : identifier → type
(define (get-id-type id) omitted )
;; binding→type-declaration : binding → syntax
(define (binding->type-declaration b)
  (with-syntax ([id (binding-id b)]
                [type-expr (type->type-expression (binding-type b))])
     #'(begin-for-syntax (declare-type! #'id type-expr))))
;; type→type-expression : type → syntax
(define (type->type-expression type) omitted )
```

Figure 10.10: Type Checker

CHAPTER 11

Evaluation[1]

Assessing whether Typed Scheme truly provides an easy transition from untyped to typed code demands practical experience.[2] This chapter presents the result of porting several scripts, libraries, and complete applications from PLT Scheme to Typed Scheme. The chosen examples are representative of working code in that they are created by experienced programmers and they have been in use for a significant amount of time. Similarly, the process itself is representative: for all of the examples, all communication between the original creator and the "porter" was disallowed.

The first section discusses the selected examples and the particular problems that they pose. The second section describes the changes that had to be made for the selected programs to typecheck. Most changes simply added type annotations. Some required changes to the code to appease the typechecker, but other changes fixed bugs, or handled errors properly where they had been ignored. In particular, a description of all the changes made to the code of the largest example is given in section 11.2.4. The third section

[1] Many of the programs in this chapter were originally ported to Typed Scheme by Ivan Gazeau.

[2] For the development of Typed Scheme, several thousand lines of text book Scheme code were used as the initial data set for determining what idioms to consider. Since such code tends to be illustrative of idioms, it is well suited for this purpose but not for evaluation. Additionally, a system should never be evaluated for performance on its own training data.

provides a quantitative account of the complexity involved in the porting process, as well as a subjective account.

11.1 Selected Programs

The chosen programs run the gamut from widely distributed libraries to single-use scripts that rely heavily on PLT Scheme features. Together, they total over 5000 lines of code. In addition, over 1000 lines of code in the PLT Scheme standard library is written in Typed Scheme, including wrapper modules for many libraries that are used in the selected programs.

Squadron Scramble This application, created by Matthias Felleisen, is a version of the multi-player card game Squadron Scramble [Kaplan 2002], which resembles Rummy. The original untyped implementation consists of 10 PLT Scheme modules, totaling 2200 lines of implementation code, plus 500 lines of unit tests. The program maintains data about the set of cards in an external XML file, which it reads on startup.

Scheme Code Metrics This script is designed for applying syntactic metrics to collections of PLT Scheme modules and files. It traverses directory trees, looking for source files, and analyzes the files found to determine the frequency of various programming constructs. The program does not have a test suite; it is used by its author on a regular basis. It consists of a single module of just over 400 lines.

Money Management This program, also developed by Matthias Felleisen, helps with balancing checkbooks, and has been used for this purpose for the past decade. It is not released for any other use, nor does it have a test suite. During its extensive use, the program has seemed bug-free over the past several years. The program consists of one script and one library module, totaling just under 400 lines.

Spam Filtering This script is used to analyze the contents of an spam folder. It has only been used by its creator, and consists of one PLT Scheme module of 311 lines.

System Administration This library is designed for handling system administration tasks, with particular emphasis on those relating to Subversion repository administration. It consists of 15 modules and one data file, totaling more than 1200 lines.

Random Number Generation This is an old version of the PLT Scheme implementation of the widely-used SRFI 27 library for random number and bit generation. It has been in wide distribution along with PLT Scheme for several years. It consists of a single library module of almost 600 lines.

11.2 Program Changes

Porting this wide variety of programs poses a number of common challenges and brings similar benefits to the programs involved. Most changes are simply providing types, but some require changes to the code as executed.

11.2.1 Annotations

The vast majority of the changes require are simple type annotations. The most common is providing the type of a top-level function definition. Often, this simply involves translating an existing comment to a statically-checked type annotation. For example, the untyped code:

```
;; Player Hand → Void
(define (clean-up player hand)
  (set-player-hand! player hand))
```

becomes

```
(: clean-up (player Hand → Void))
(define (clean-up player hand)
  (set-player-hand! player hand))
```

Almost all top-level definitions are annotated in this manner, without requiring changes to the definition of the function.

The other common site of annotations is structure definitions. The fields of a structure must be annotated with types, so that Typed Scheme can determine the types of the structure functions. For example, the untyped code:

```
;; Administrator = (make-administrator [Listof IPlayer])
(define-struct administrator (iplayers))
```

becomes

```
(define-struct: administrator ([iplayers : (Listof iplayer)]))
```

Again, as the example shows, this is often a translation of a pre-existing comment.

Two other forms of required annotation are rare. One is the definition of new type aliases:

```
(define-type-alias Raw-Group (Pair Symbol (Listof Raw-Student)))
```

This specifies that the type name *Raw-Group* is equivalent to a pair of a symbol and list of *Raw-Students*. Type aliases are useful for simplifying and shortening type annotations.

Finally, when an untyped library is used, the type of imported values must be specified with the **require/typed** form. For example, this require of the untyped library *srfi/13*:

```
(require srfi/13)
```

becomes, in the typed version:

```
(require/typed srfi/13
  [string-pad-right (String Number Char → String)])
```

The **require/typed** form explicitly specifies the type of each imported identifier, allowing Typed Scheme to generate the appropriate contracts as described in chapter 5.

In many cases, the PLT Scheme standard library provides a typed version of existing libraries.[3] In that case, the new **require** statement would be (**require** *typed/srfi/13*), and no annotations would be required for the *string-pad-right* function.

All four of these annotation forms merely specify the programmers' understanding of the type structure of their program, and provide executable form for documentation that is often already in the source code.

Some other annotations are required. Typed Scheme's inference for polymorphic functions does not handle the case where polymorphic values are used as the argument to polymorphic functions. In this case, one of the functions must be explicitly instantiated at the appropriate type. In some other cases, local type inference cannot determine the type of a variable, and it must be explicitly annotated. Finally, the bound variables of λ s used as the argument to polymorphic functions must be explicitly annotated, since local inference is also unable to determine the desired type. Fortunately, all of these occur rarely in the programs ported—see section 11.3 for precise measurements.

11.2.2 Improvements

The first improvement concerns simple mistakes, such as passing the wrong number of arguments to a function. The chosen examples contain only five such errors, four of which are in example uses of data structures, rather than in critical code paths. This is not surprising because these programs had been in significant prior use. In the context of finding simple errors, type systems are mostly useful during the actual programming process. Such simple mistakes do turn up in comments, which are not exercised by testing, and Typed Scheme discovered several instances where the comments about a procedure disagreed with its implementation.

[3] These libraries were contributed in part by Yinso Chen, Felix Klock, Ivan Gazeau, and David van Horn.

The second category of problems concerns handling of erroneous input. This manifests itself in three forms:

- Many scripts, especially those originally intended to be used only once, are not designed for robustness. Therefore, the possibility of failures is ignored to simplify the programming task. In PLT Scheme, the simplest case is when a script uses a function that produces #f to indicate failure, but the script does not deal with such results. For example, the expression (*regexp-match* "([ˆ<,]*)@acm\\.org" *s*) produces a list containing the email address in the string *s* if it succeeds, and #f if it fails. Many scripts that use such library functions ignore the possibility of failure.

  Our solution is to provide an *assert* function, which translates #f into an exception. This allows programmers to compactly state assumptions in a type-correct way. For example, the expression

  ```
  (assert (regexp-match "([ˆ<, ]*)@acm\\.org" s))
  ```

  always produces a list—or, if the regular expression matching fails, raises an error. *assert* has the type

  $$(\forall\ (X)\ ((\bigcup X\ \#\mathsf{f}) \to X))$$

- A similar case occurs when a function *f* uses a function *g* with a restricted domain. Ideally, *f* should check the applicability of *g* and raise an error if the condition isn't met. Unfortunately, many scripts ignore such cases, and thus fail late and in unexpected ways. For example, the / function requires at least one argument. One function in the code base applied / to a list of arguments, even when that list might be empty. To address this, the Typed Scheme versions of our samples make types as precise as possibles for functions such as *g* and thus force errors to be discovered as early as possible, with informative error messages. In the example case, the function was changed to always take a minimum of one argument.

- The most serious problem is due to input obtained from an external source and assumed to be of a certain shape. For example, the Squadron Scramble game keeps information about the cards of the game in an external XML file. When the data is read in, it is assumed to be of the required format, rather than being parsed. The resulting data is used directly as input to remainder of the computation, although the required invariant are not checked. Indeed, the documentation for the code contains a schema for the XML format, but this is not integrated into the system.

  These assumptions about input can be pervasive in large portions of code, as in Squadron Scramble. If they are, it can be helpful to factor out all of the input-handling code into a module that remains untyped, exporting only functions that produce the relevant results in a type-correct faction. In simple cases, additional error checking suffices.

Our third category of improvements are due to the elimination of type-inconsistent assignment statements. For example, one program always creates a data structure using an *init* function that constructs the structure with potentially type-incorrect data and then mutates it to be correct. This pattern has two problems. On one hand, it assumes that all the instances of this data structure are created using the *init* function, but this is not guaranteed. On the other hand, it makes the invariant harder to discover and maintain by future programmers. In such cases, our Typed Scheme programs forgo mutation, and ensure that the data structure is always correct by construction. Another example of this is a list variable that is later initialized by mutation. In two examples, the initial value was #f, a value which is not acceptable to the other uses of the variable. The necessary change is simply to initialize the variable with ’() instead.

11.2.3 Limitations of Typed Scheme

Unfortunately, not all changes that Typed Scheme requires are clear-cut improvements to the program. Our experiments have pinpointed three weaknesses in our approach. This section discusses a few categories of such problems and how they affect the porting process.

First, some facts about a program that are obvious to the programmer can not be proved by typechecker. For example, this **cond** expression covers all possibilities, even though Typed Scheme cannot prove it:

```
(cond [(= x 0) . . . ]
      [(< x 0) . . . ]
      [(> x 0) . . . ])
```

Fortunately, it is easy to rewrite this expression to use **else**, at which point Typed Scheme is able to check it correctly.

Another example of this is that sometimes information is known about the relationship between the types of two variables: if x is #f, then y is not. While this can be expressed in some cases via the logical system describe in chapter 7, it is not always possible. In this case, additional dynamic checks must be added to the program.

Second, Typed Scheme cannot, in general, type check macros that generate hidden recursive code. In most cases, local type inference is sufficient to infer the types of the generated loop variables. When this is not the case, however, the programmer must rewrite the code with an explicit loop, which can then be given an explicit type annotation. This is a rare problem, but it causes significant difficulty to the porter when it occurs.

Third, some functions are used in special ways in PLT Scheme, such that the type system handles them specially. An example is the *list* function, which generates fixed-length lists when applied to a known set of arguments. When used to generate fixed-length lists in a higher-order context, Typed Scheme is unable to apply the special type rule. The workaround is to η-expand the *list* function for the particular number of arguments, which allows Typed Scheme to apply the type rule for *list* to the explicit application. Another example of such a function is the *values* function.

Fourth, in some cases, occurrence typing does not properly track the uses of type predicates. For example, the expression (*andmap number? xs*) should verify that *xs* is a list of numbers: However, this is not currently the case for Typed Scheme. The porting effort found a few of these examples, which I hope to address in future work.

11.2.4 Changes in Squadron Scramble

The Squadron Scramble game is the largest of our experiments. This section describes all of the non-annotation changes required for Squadron Scramble to typecheck. A total of twelve changes were required to the code of the game, some of which fixed bugs, and some of which worked around limitations of Typed Scheme.

Workarounds By far the most significant change was factoring the XML reading code into a separate, untyped file, and then using **require/typed** to import the relevant values into Typed Scheme with appropriate types. This involved creating a new file of 30 lines, all taken from one of the existing files. Two values from the new file were then **require/typed** back into the original file.

In two places, the same datatype was assumed to be covered by two predicates in the untyped code. However, Typed Scheme could not prove that these predicates were sufficient to cover the data type—in fact, the untyped program assumed that all data fit a format that was never checked. The Typed Scheme code added an additional error case in both places.

In one place, a function produced multiple values of correlated types.

Typed Scheme is not able to keep track of this correlation, necessitating an additional dynamic check.

As mentioned, Typed Scheme does not yet propagate type information from the *andmap* function, which necessitated an additional check in one place. Similarly, Typed Scheme does not allow the programmer to express that the user-written *set?* predicate verifies that the argument is a list when it is a set. Finally, the expansion of **or** uses a temporary variable, which does not fit with the rules for *combfilter* given in chapter 6. Manually expanding the (**or A B**) expression into (**if A** #t **B**) is sufficient to work around this limitation. Each of these occurred precisely once in Squadron Scramble (and in none of the other experiments).

Bug Fixes In one location, the *string→number* procedure was used, while neglecting that the result might be #f if the provided string cannot be parsed as a number. The *assert* procedure converts this possibility into an exception, as described above.

Four similar functions, all used as examples, call a constructor function with the wrong number of arguments. Fixing these is very simple.

11.3 Statistics and Results

Figure 11.1 presents the quantitative results of our measurement effort. Each column represents one of the programs described in section 11.1. The rows quantify the size of the programs, and the effort required to port them to Typed Scheme:

- LINES: This is the total line count for each program after porting, including comments and blank lines.

- INCREASE: This is the percentage increase in number of lines from the untyped version to the typed version. The **Metrics** program is an

	Squad	Metrics	Acct	Spam	Sys	Rand	Total
Lines	2369	511	407	315	1290	618	5510
Increase	7%	25%	7%	6%	1%	3%	7%
Useful Ann	96	51	25	16	53	26	267
λ: Ann	34	22	2	8	22	0	88
Other Ann	12	7	1	1	0	1	22
define-struct:	16	2	0	0	4	1	23
Type Alias	13	7	0	2	2	3	27
require/typed	3	1	0	0	5	0	9
Ann/100 Line	7.3	18	6.9	8.6	6.7	5.0	7.9
Fixes	5	3	4	5	8	0	25
Fixes/100 Line	0.21	0.59	0.98	1.6	0.62	0.0	0.45
Problems	7	4	3	1	0	1	16
Prob/100 Line	0.29	0.78	0.73	0.32	0.0	0.16	0.29
Difficulty	⋆⋆	⋆⋆⋆	⋆⋆	⋆	⋆	⋆	

Figure 11.1: Statistics

outlier here—many of its types are quite large and were formatted on multiple lines.

- Useful Ann: This the count of annotations of variables that recorded valuable design information about the program. It includes top-level functions, local functions, top-level constants in which the untyped program provided a comment specifying the type, and specifications of the types of mutable data, such as hash tables.

- λ: Ann This is the count of uses of the λ: form, which annotates the bound variables of a λ with types. It is almost exclusively necessary when an anonymous function is provided as the argument to a polymorphic function such as *map*.

- Other Ann: This is a count of all other variable annotations and instantiations in the program, which do not appear to record useful design information.

- **define-struct:** This is the number of uses of **define-struct:** in the typed version of the program. In all cases, this is the same as the number of

define-structs in the original program.

- TYPE ALIAS: This is the number of type aliases defined in the typed version of the program.

- **require/typed**: This is the number of identifiers imported with **require/typed**, each of which requires a type to be specified.

- ANN/100 LINE: This is the number of annotations, including structs, aliases, and **require**s, per 100 lines of the typed version.

- FIXES: This is the number of changes to the program that fixed actual or potential errors in the original code.

- FIXES/100 LINE: The number of such fixes per 100 lines of the typed program.

- PROBLEMS: This is the number of changes to the program that are merely to work around the typechecker. These varied widely in lines of code: moving XML parsing to a separate file affected close to 100 lines, whereas other changes involved the addition of a single function call.

- PROB/100 LINE: The number of such workarounds per 100 lines of the typed program.

- DIFFICULTY: Finally, the last row indicates the subjective difficulty, as experienced by the programmer(s) doing the port of the program. One star is easiest, and three represent the most difficult. Here, the **Metrics** system is the most difficult, largely due to its complex data structures. In particular, one type of structure is used for multiple different purposes in the program, which makes it difficult to express its intended type in Typed Scheme. Some of the complexity of these data structures is reflected in the significant increase in the size of the program.

> In contrast, the programs that were simplest to port have a simple type structure, mostly dealing with only one type or having no interesting data structures. These distinctions, and not the quantifiable labor effort, ultimately make the greatest difference in porting effort.
>
> The **Squadron Scramble** game implementation fell in between. It does not pose serious conceptual problems, and the code is meticulously commented and maintained. But it is a code artifact of significant size, and thus has its own idiosyncrasies, such as the use of an external XML file to store data. Therefore, it requires more than a trivial amount of effort to port. Ultimately, the statistical metrics indicate that **Squadron Scramble** was representative of the overall sample.

11.3.1 Interpretation

This quantitative measurement provides a guide to the effort required to port programs written in PLT Scheme to Typed Scheme. It indicates that the program increases in size by less than a tenth, that annotations are not overly frequent, and that content changes to the running code are rarely required.

These numbers can only give a rough guide to the effort of porting programs to Typed Scheme—the most difficult portion of any programming task is understanding the nature of the problem and the form of the correct solution. In some cases, the porting programmer must develop a sophisticated understanding of the workings of the system. No type system can offer guidance on the ultimate nature of this task. But the estimates offered here show that Typed Scheme requires a reasonably small effort, especially given the benefits for future program maintenance.

CHAPTER 12

Related Work

Typed Scheme is related to numerous other systems and research programs. In particular, many researchers have explored connections between typed and untyped languages, with much recent work under the heading of "gradual typing". Typed Scheme also uses numerous type system features pioneered in other languages. Finally, the particular innovative features of the type system, such as occurrence typing and variable-arity polymorphism, are related to other work on similar questions in other languages. This chapter provides a survey of this extensive literature.

12.1 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, Tomb, and Flanagan 2008; Wadler and Findler 2009; Ina and Igarashi 2009; Wrigstad, Nystrom, and Vitek 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 that prove that only the untyped portions of the program can go wrong. This is analogous to earlier work on Typed Scheme [Tobin-Hochstadt and Felleisen 2006], which provides such a soundness theorem, given in chapter 5, which I believe scales to full Typed Scheme and PLT Scheme.

Second, Furr, An, Foster, and Hicks [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.

12.2 Type System Features

Many of the type system features in 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. My use of visible filters and especially latent filters is inspired by prior work on effect systems [Gifford, Jouvelot, Lucassen, and Sheldon 1987].

12.3 Refinement Types

Refinement types are due to Freeman and Pfenning [1991]. Since then, refinement types have been used in a wide variety of systems [Rondon, Kawaguci, and Jhala 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. 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 [Rondon et al. 2008] and Hybrid Typing [Flanagan 2006] approaches.

The approach taken by Typed Scheme differs from both of these approaches. First, refinements are not specified using the language of data constructors 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 a theorem prover.

12.4 Types and Logic

Considering types as logical propositions has a long history, going back to Curry and Howard [Curry and Feys 1958; Howard 1980]. In a dependently typed language such a Coq [Bertot and Castéran 2004] or Agda [Norell 2007], the relationships we describe with filters and objects could be encoded in types, since types can contain arbitrary Coq terms, including ones that reference other variables or the expression itself.

In Typed Scheme, the rule for typechecking **if** expressions propagates information known when the test evaluates to true to the then branch, and information known when the test evaluates to false to the else branch. This could be expressed with an `if` combinator of the following Coq type:

```
forall Q1 Q2 Q P, (P = true -> Q1) -> (P = false -> Q2)
 -> (Q1 -> Q) -> (Q2 -> Q) -> Q
```

In a language with dependent types, this combinator could be used with decision procedures that supply witnesses, a style already used in Coq. Procedures such as *number?* would be reinterpreted as such decision procedures.

From this perspective, the Typed Scheme type system carves out a subset of the logical reasoning process available in programming languages with such rich type systems. This subset is tailored to the idioms and styles of existing untyped Scheme code bases, and has allowed Typed Scheme to type-check a wide variety of existing Scheme code.

12.5 Occurrence Typing

The term "occurrence typing" was coined by Komondoor et al. [2005] in their work on automatic understanding of legacy COBOL programs. Their system considers a limited form of occurrence typing, restricted to equality tests between variables and character constants, which are used as tag checks in existing COBOL programs. Their system treats such equalities as propositions, and negates them in the else branch. It does not allow abstractions over predicates, more general forms of predicates, or tests on non-variables.

Some features similar to those of occurrence typing 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, Weirich, and Morrisett [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 occurrence typing in Typed Scheme, 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. Intensional polymorphism without refinement of the types of variables appears in calculi by Harper and Morrisett [1995], among others.

12.6 Variable-Arity Polymorphism

Variable-arity functions are nearly ubiquitous in the world of programming languages, but no typed language supports them in a systematic and principled manner. Here we survey existing systems as well as several theoretical efforts.

ANSI C provides “varargs,” but the functions that implement this functionality serve as a thin wrapper around direct access to the stack frame. Java [Gosling, Joy, Steele Jr., and Bracha 2005] and C# are two statically typed languages that have only uniform variable-arity functions, since access occurs via a homogeneous array.

Dzeng and Haynes [1994] come close to our goal of providing a practical type system for variable-arity functions. As part of the Infer system for type-checking Scheme [Haynes 1995], they use an encoding of “infinitary tuples” as row types for an ML-like type inference system that handles optional arguments and uniform and non-uniform variable-arity functions.

In comparison to the Typed Scheme approach, Dzeng and Haynes’ system has several limitations. Most importantly, since their system does not support first-class polymorphic functions, they are unable to type many of the definitions of variable-arity functions, such as *map* or *fold*. Additionally, their system requires full type inference to avoid exposing users to the underlying details of row types, and it is also designed around a Hindley-Milner style algorithm. This renders it incompatible with the remainder of the design of Typed Scheme, which is based on a system with subtyping.

Gregor and Järvi [2007] propose an extension for variadic templates to

C++ for the upcoming C++0x standard. This proposal has been accepted by the C++ standardization committee. Variadic templates provide a basis for implementing non-uniform variable-arity functions in templates. Since the approach is grounded in templates, it is difficult to translate their approach to other languages without template systems. The template approach addresses a simpler problem because template expansion is a pre-processing step and types are only checked after template expansion. It also significantly complicates the language, since arbitrary computation can be performed during template expansion. Further, the template approach prevents checking of variadic functions at the definition site, meaning that errors in the definition are only caught when the function is used.

Tullsen [2000] attempts to bring non-uniform variable-arity functions to Haskell via the Zip Calculus, a type system with restricted dependent types and special kinds that serve as tuple dimensions. This work is theoretical and comes without practical evaluation. The presented limitations of the Zip Calculus imply that it cannot assign a variable-arity type to the definition of `zipWith` (Haskell's name for Scheme's *map*) without further extension, whereas Typed Scheme can do so.

Similarly, McBride [2002] and Moggi [2000] present restricted forms of dependent typing in which the number of arguments is passed as a parameter to variadic functions. Our system, while not allowing the expression of every dependently-typable program, is simpler than dependent typing, suffices for most examples we have encountered, and does not require an extra function parameter.

12.7 Types and Flow Analysis

In addition to the work on soft typing discussed in chapter4, other flow analysis research has attempted to solve problems similar to those that occurrence typing addresses.

In chapter 9 of his thesis [Shivers 1991], Shivers describes a type recovery analysis that includes refining the type of variables in type tests. However, this only covered the particular case of tests for integers, and did not include a general mechanism for handling arbitrary type tests, or abstraction over such tests.

The conditional types of Aiken et al. [1994] are closely related to occurrence typing. Conditional types rely on a 'case' expression with patterns. These patterns can include nested patterns, allowing inspection of portions of compound data structures, as in Typed Scheme. However, conditional types do not support abstraction over predicates, e.g., in Typed Scheme the expression (λ ([*x* **: Any**]) (**or** (*number? x*) (*string? x*))) has a non-trivial latent filter, whereas patterns cannot be abstracted over. Furthermore, conditional typing does not propagate information about possible values matched in one pattern to subsequent case branches. For example, in contrast to occurrence typing, conditional typing cannot determine for 'case x of cons(true,b) : true , cons(a,b) : false' that 'a' cannot be 'true' in the second branch.

12.7.1 Deriving Propositions

Some flow analysis systems derive formulae that describe the program that are not simply richer types. For example, Might's Logic Flow Analysis [Might 2007] derives propositions about the equality of numeric values during the run of a flow analysis. The overall design of Might's system is different, since it involves an external theorem prover in interaction with a flow analysis, and derives propositions that are radically different from mine. However, in a type checking setting, we expect that Might's system would be able to derive similar theorems to those of λ_{TS} .

Chapter 13

Conclusion

When scripts grow up, all too often they become unmaintainable. One important reason for this is that the design information the original programmer had is lost, and difficult to recover. For these scripts to become maintainable programs, recovering that design information is essential. I have proposed that modular porting of untyped code to a typed sister language makes the transition from scripts to programs not only possible, but easy, by facilitating this recovery process.

13.1 Contributions

To support this thesis, I have developed Typed Scheme, a typed sister language of PLT Scheme. Typed Scheme consists of two key pieces: a system for sound interaction between typed and untyped modules, and a type system that works with traditional Scheme idioms.

Typed Scheme allows typed and untyped modules to freely interoperate, with higher-order values able to flow in both directions. It also automatically generates runtime software contracts to enforce the types of the typed portions of the program and blame the appropriate offending party. This integration satisfies a soundness theorem that proves that the typed portions of the program are never blamed, generalizing Milner's slogan to "well-typed *modules* don't go wrong".

Typed Scheme includes a novel type system designed to accommodate typical Scheme programming idioms. This includes the idea of occurrence typing, which allows type information from tests to be used in the typing of branches. This formulation of occurrence typing also allows for a lightweight form of refinement types. The type system also includes a novel system for handling variable arity functions, even those with complex relationships among their argument types.

Finally, Typed Scheme is implemented and distributed as a part of PLT Scheme. The implementation of Typed Scheme uses the PLT Scheme module and macro system to implement a sound type system as a library for an existing untyped language. This implementation has been used to port thousands of lines of existing code, and it is used in the implementation of the PLT suite of tools. Experiments with the implementation indicate that porting existing untyped PLT Scheme code requires few changes to existing code.

13.2 Future Work

While Typed Scheme is useful today, much more work remains to be done.

13.2.1 Flow Analysis for Porting

Existing flow analyses [Shivers 1991; Flanagan and Felleisen 1999] generate results that resemble Typed Scheme types. While experience with soft typing suggests that flow analysis is not appropriate for a type system, it may well be useful for the process of porting untyped Scheme code to Typed Scheme. A tool using flow analysis could be run once, generating a preliminary typing for Typed Scheme. This could alleviate much of the manual burden of annotating variables, as well as allowing complex and computationally-expensive algorithms to be used, since they would not be a constant part of the development process.

13.2.2 Types for Complex Macros

While Typed Scheme's strategy of examining only post-expansion code performs well in most cases, it fails on the most complex macros. The PLT Scheme **class** [Flatt et al. 2006] and **unit** [Flatt and Felleisen 1998] systems are implemented entirely with macros—but the expansion of these macros is insufficient to recover the programming discipline that they enforce. Thus, typing programs written using these systems must rely on an understanding of these macros, as well as a type system that handles their unique features.

13.2.3 Beyond Scheme

Scheme is not the only language that can benefit from modular addition of types. Language such as JavaScript, Python, and Ruby are likely to benefit from the lessons of Typed Scheme. These languages will require their own type system, as well as affordances for their own peculiar idioms and features.

Bibliography

Alexander Aiken and Brian R. Murphy. Static type inference in a dynamically typed language. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 279–290. ACM Press, 1991.

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. ACM Press, 1994.

Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*, volume XXV of *EATCS Texts in Theoretical Computer Science*. Springer-Verlag, 2004.

R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, NY, USA, second edition, 1997.

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

Robert Cartwright. User-defined data types as an aid to verifying LISP programs. In *International Conference 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. ACM Press, 1991.

Robert Cartwright, Robert Hood, and Philip Matthews. Paths: an abstract alternative to pointers. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–27. ACM Press, 1981.

William Clinger and Jonathan Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162. ACM Press, 1991.

William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for scheme. In *LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 128–139. ACM Press, 1994.

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. ACM Press, 1998.

Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 373–388. Springer-Verlag, 2005.

Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. 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, pages 1–13, 2007.

Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

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. ACM Press, 1994.

ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), third edition, 1999.

ECMA. ECMAScript Edition 4 group wiki, 2007. `http://wiki.ecmascript.org/`.

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

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

David Fisher and Olin Shivers. Building language towers with Ziggurat. *Journal of Functional Programming*, 18(5 & 6):707–780, 2008.

Cormac Flanagan. Hybrid type checking. In *Conference Record of POPL '06: The 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256. ACM Press, 2006.

Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 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: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 23–32. ACM Press, 1996.

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

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

Matthew Flatt and PLT. Reference: PLT Scheme. Reference Manual PLT-TR2009-reference-v4.2.2, PLT Scheme Inc., 2009. `http://download.plt-scheme.org/doc/4.2.2/pdf/reference.pdf`. `http://plt-scheme.org/techreports/`.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems (APLAS) 2006*, volume 4279 of *Lecture Notes in Computer Science*, pages 270–289. Springer-Verlag, 2006.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 268–277. ACM Press, 1991.

Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for ruby. In *SAC '09: Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1859–1866. ACM Press, 2009a.

Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. 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], pages 14–16.

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, 1987.

J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java Language Specification*. Addison-Welsley, third edition, 2005.

Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20thannual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 231–245. ACM Press, 2005.

Douglas Gregor and Jaakko Järvi. Variadic templates for C++. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1101–1108. ACM Press, 2007.

Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 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: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 306–317. ACM Press, 1994.

Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

Fritz Henglein and Jakob Rehof. Safe polymorphic type inference foir 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. ACM Press, 1995.

David Herman and Cormac Flanagan. Status report: specifying JavaScript with ML. In *ML '07: Proceedings of the 2007 Workshop on ML*, pages 47–52. ACM Press, 2007.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007*, pages 1–18, 2008.

William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, NY, USA, 1980. Reprint of 1969 article.

Lintaro Ina and Atsushi Igarashi. Gradual typing for Featherweight Java. *Computer Software*, 26(2):18–40, 2009.

Stuart Kaplan. *Squadron Scramble*. US Game Systems, Stamford, CT, 2002.

Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 151–161. ACM Press, 1986.

Raghavan Komondoor, G. Ramalingam, Satish Chandra, and John Field. Dependent types for program understanding. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 157–173. Springer-Verlag, 2005.

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

Rasmus Lerdorf, Kevin Tatroe, and Peter MacIntyre. *Programming PHP*. O'Reilly Media, second edition, 2006.

John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–57. ACM Press, 1988.

Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *ICFP '97: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 136–149. ACM Press, 1997.

Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly Media, 2001.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Transactions on Programming Languages and Systems*, 31(3):1–44, 2009.

Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(5):375–392, 2002.

Drew McDermott. Revised NISP manual. Technical Report YALE/DCS/RR-642, Yale University, Department of Computer Science, 2004.

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

Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, 1992.

Matthew Might. Logic-flow analysis of higher-order programs. In *Conference Record of POPL '07: The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–198. ACM Press, 2007.

Eugenio Moggi. Arity polymorphism and dependent types. In *Proceedings of the International Workshop on Subtyping and Dependent Types in Programming*, 2000.

Randal Munroe. Exploits of a mom. `http://xkcd.com/327/`, 2007.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

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

Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 216–227. ACM Press, 2005.

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

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

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

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *PLDI '08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 159–169. ACM Press, 2008.

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

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

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

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *Journal of Functional Programming*, 19(Supplement S1):1–301, 2009.

Guy Lewis Steele Jr. *Common Lisp—The Language*. Digital Press, Woburn, MA, second edition, 1990.

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Variable-Arity Polymorphism. Technical Report NU-CCIS-08-03, Northeastern University, 2008. `http://www.ccs.neu.edu/scheme/pubs/NU-CCIS-08-03.pdf`.

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *ESOP '09: Proceedings of the Eighteenth European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 2009.

Gerald J. Sussman and Guy Lewis Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report AIM-349, MIT Artificial Intelligence Laboratory, 1975.

Audrey Tang. Perl 6: reconciling the irreconcilable. In *Conference Record of POPL '07: The 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–1. ACM Press, 2007. `http://pugscode.org`.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. Mansuscript under submission, 2009.

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

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *POPL '08: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 395–406. ACM Press, 2008.

Sam Tobin-Hochstadt and Robert Bruce Findler. Cycles without pollution: a gradual typing poem. In Wrigstad et al. [2009], pages 47–57.

Mark Tullsen. The zip calculus. In Roland Backhouse and Jose Nuno Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 28–44. Springer-Verlag, 2000.

Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual: (Python Documentation Manual Part 2)*. CreateSpace, 2009.

Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP '09: Proceedings of the Eighteenth European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2009.

Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, second edition, 1996.

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

Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information & 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 Transactions on Programming Languages and Systems*, 19 (1):87–152, 1997.

Tobias Wrigstad, Nate Nystrom, and Jan Vitek, editors. *STOP '09: Proceedings for the 1st workshop on Script to Program Evolution*, New York, NY, USA, 2009. ACM Press.