Logical Types for Untyped Languages *

Sam Tobin-Hochstadt Matthias Felleisen
Northeastern University
{samth,matthias}@ccs.neu.edu

Abstract

Programmers reason about their programs using a wide variety of formal and informal methods. Programmers in untyped languages such as Scheme or Erlang are able to use any such method to reason about the type behavior of their programs. Our type system for Scheme accommodates common reasoning methods by assigning variable occurrences a subtype of their declared type based on the predicates prior to the occurrence, a discipline dubbed *occurrence typing*. It thus enables programmers to enrich existing Scheme code with types, while requiring few changes to the code itself.

Three years of practical experience has revealed serious shortcomings of our type system. In particular, it relied on a system of ad-hoc rules to relate combinations of predicates, it could not reason about subcomponents of data structures, and it could not follow sophisticated reasoning about the relationship among predicate tests, all of which are used in existing code.

In this paper, we reformulate occurrence typing to eliminate these shortcomings. The new formulation derives propositional logic formulas that hold when an expression evaluates to true or false, respectively. A simple proof system is then used to determine types of variable occurrences from these propositions. Our implementation of this revised occurrence type system thus copes with many more untyped programming idioms than the original system.

1. Reasoning about Untyped Languages

Developing programs in a typed language helps programmers avoid mistakes. It also forces them to provide some documentation, and it establishes some protective abstraction barriers. As such, the type system imposes a discipline on the programming process.

Nevertheless, numerous programmers continue to choose untyped scripting languages for their work, including many who work in a functional style. When someone eventually decides that explicitly stated type information reduces maintenance cost, they face a dilemma. To address this situation, we need to develop typed sister languages for untyped languages. With those, programmers can enrich existing programs with type declarations as needed while maintaining smooth interoperability.

A type system for an existing untyped language must accommodate the existing programming idioms in order to keep the cost of type enrichment low. Otherwise, type enrichment requires changes to code, which may introduce new mistakes. Put positively, the ideal typed sister language requires nothing but the addition of type specifications to function headers, structure definitions, etc.

Our experience shows that programming idioms in untyped functional languages rest on a combination of traditional type-based reasoning with reasoning about control flow. In particular, conditionals and data-type predicates are used to establish the nature of variables' values, and based on this flow-sensitive reasoning, programmers use variables at more specific types than expected. Put differently, the programmer determines the type of each variable occurrence based on the predicates that flow-dominate it.

Multiple researchers over the decades have discovered this insight. In his paper on the static analysis of untyped programs, Reynolds [1968] notes that such reasoning is necessary, stating that in future systems, "some account should be taken of the premises in conditional expressions." In his work on TYPED LISP, Cartwright [1976, §5] describes having to abandon the policy of rejecting type-incorrect programs because the variables in conditionals had overly broad types. Similarly, in their paper on translating Scheme to ML, Henglein and Rehof [1995] state "type testing predicates aggravate the loss of static type information since they are typically used to steer the control flow in a program in such a fashion that execution depends on which type tag an object has at run-time."

We exploited this insight for the development of Typed Scheme, a typed sister language for Racket (formerly PLT Scheme) [Tobin-Hochstadt and Felleisen 2008]. Its type system combines several preexisting elements—"true" recursive union types, subtyping, polymorphism—with the novel idea of *occurrence typing*,[1] a type discipline for exploiting the use of data-type predicates in the test expression of conditionals. Thus, if a test uses (*number? x*), the type system uses the type **Number** for *x* in the `then` branch and the declared type of *x*, minus **Number**, otherwise.

Three years of extensive use have revealed several shortcomings in our original design. One significant problem concerns control flow governed by logical combinations (e.g., **and**, **or**, *not*) of predicate tests. Another is that the type system cannot track uses of predicates applied to structure selectors such as *car*.

This lack of expressiveness in the type system is due to fundamental limitations. First, our original system does not consider asymmetries between the `then` and `else` branches of conditionals. For example, when the expression (**and** (*number? x*) (> *x* 100)) is true, the type system should know that *x* is a number, but *x* might or might not be a number when the expression is false, since it might be `97` or `"Hello"`. Second, the type system does not appropriately distinguish between selector expressions such as (*car x*) and predicate expressions such as (*number? x*). Third, the treatment of combinations of tests relies on an ad-hoc collection of rules.

* This research was partially supported by grants from the US NSF and a donation from the Mozilla Foundation.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

[1] Komondoor et al. [2005] independently coined the term "occurrence typing" in the context of providing an advanced type system for COBOL.

In this paper, we present a new and simple framework for occurrence typing that eliminates all three problems via an increase in expressive power. The key innovation is to turn control flow predicates into formulas in propositional logic for reasoning about the types of variables. The atomic propositions include statements that a particular variable has a particular type, replacing the previous collection of special cases with textbook rules of logical inference.

This new design allows the type system to reason about combinations of predicates:

```
(cond
  [(and (number? (node-left x)) (string? (node-right y)))
  ;; known: (node-right y) is a string, (node-left x) is a number
  ...]
  [(number? (node-left x))
  ;; known: (node-right y) is not a string
  ...]
  [(string? (node-right y))
  ;; known: (node-left x) is not a number
  ...])
```

Now the programmer and the revised type system both determine that since (*number?* (*node-left x*)) is true in the second clause, (*string?* (*node-right y*)) must be false, and thus, (*node-right y*) is *not* a string. Using propositional logic to reason about predicates handles this and many other similar situations.

Beyond the new type system, this paper contributes:

- a full-fledged implementation of the calculus, now known as Typed Racket, addressing the full complexities of the functional core of Racket, such as mutable data and multiple arguments;
- an empirical study of the usefulness of our extensions; and
- a novel model-theoretic proof technique for type soundness.

The paper begins with a brief review of the essence of occurrence typing with an emphasis on programming idioms that our original system cannot typecheck. We then describe the new system in a semi-formal manner. In the three following sections, we describe the system formally: first the core system of occurrence typing, then several extensions demonstrating the expressiveness of the system, and third the proof of soundness. These sections are followed by a description of our implementation strategy and empirical measures of its usefulness on existing code. Finally, we discuss related work and conclude.

2. A Brief Introduction to Occurrence Typing

Here is the simplest example of occurrence typing:

```
... (if (number? x) (add1 x) 0) ...
```
Example 1

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 the type is not legitimate for *add1*. The key to typing this program 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.

2.1 Existing Capabilities

The following function *f* always produces a number:

```
(define: (f [x : (⋃ String Number)])
  (if (number? x) (add1 x) (string-length x)))
```
Example 2

If (*number? x*) produces #t, *x* is an appropriate input for *add1*. If it produces #f, *x* must be a **String** by process of elimination; it is therefore an acceptable input to *string-length*. To handle this program, the type system must take into account not only when predicates hold, but also when they fail to hold.

Our next fragment represents the essence of a common idiom:

```
... (let ([x (member v l)])
      (if x
          — compute with x —
          (error 'fail))) ...
```
Example 3

This idiom, seen here in *member*, uses arbitrary non-#f values as true and uses #f as a marker for missing results, analogous to ML's NONE. The type for *member* specifies this behavior with an appropriate type signature. It can thus infer that in the then branch, *x* has the type of the desired result and is not #f.

2.2 Challenges

Of course, programmers write tests beyond simple applications of predicates such as (*number? x*). For example, logical connectives can combine the results of predicates:[2]

```
... (if (or (number? x) (string? x)) (f x) 0) ...
```
Example 4

For this fragment to typecheck, the type system must recognize that (**or** (*number? x*) (*string? x*)) ensures that *x* has type (⋃ **String Number**) in the then branch, the domain of *f* from example 2.

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

```
... (if (and (number? x) (string? y))
         (+ x (string-length y))
         0) ...
```
Example 5

Example 5 is perfectly safe, regardless of the values of *x* and *y*. In contrast, the next example shows how little we know when a conjunction evaluates to false:

```
;; x is either a Number or a String
... (if (and (number? x) (string? y))
         (+ x (string-length y))
         (string-length x)) ...
```
Example 6

Here a programmer falsely assumes *x* to be a **String** when the test fails. But, the test may produce #f because *x* is actually a **String**, or because *y* is not a **String** while *x is* a **Number**. In the latter case, (*string-length x*) fails. In general, when a conjunction is false, we do not know which conjunct is false.

Finally, **and** is expressible using nested **if** expressions, a pattern that is often macro-generated:

```
... (if (if (number? x) (string? y) #f)
         (+ x (string-length y))
         0) ...
```
Example 7

One way for the type system to deal with this pattern is to reason that it is equivalent to the conjunction of the two predicates.

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

```
(define: (strnum? [x : ⊤]) ;; ⊤ is the top type
  (or (string? x) (number? x)))
```
Example 8

Take the previous example of a test for (⋃ **String Number**). A programmer can use the test to create the function *strnum?*, which behaves as a predicate for that type. This means the type system must represent the fact that *strnum?* is a predicate for this type, so that it can be exploited for conditionals.

[2] The original system could handle only an encoding of **or**, with different semantics than that provided by Racket.

In example 4, we saw the use of **or** to test for disjunctions. Like **and**, **or** is directly expressible using **if**:

Example 9

```
(if (let ([tmp (number? x)])
      (if tmp tmp (string? x)))
    (f x)
    0)
```

The expansion is analyzed as follows: if (*number? x*) is #t, then so is *tmp*, and thus the result of the inner **if** is also #t. Otherwise, the result of the inner **if** is (*string? x*). This code presents a new challenge for the type system, however. Since the expression tested in the inner **if** is the variable reference *tmp*, but the system must also learn about (*number? x*) from the test of *tmp*.

Selectors All of the tests thus far only involve variables. It is also useful to subject the result of arbitrary expressions to type tests:

Example 10

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

Even if *p* has the pair type $\langle \top, \top \rangle$, then example 10 should produce a number.[3] 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 the next example:

Example 11

```
(λ: ([p : ⟨⊤, ⊤⟩])
  (if (and (number? (car p)) (number? (cdr p)))
      (g p)
      'no))
```

the test expression refines the type of *p* from the declared $\langle \top, \top \rangle$ to the required $\langle \mathbf{Number}, \mathbf{Number} \rangle$. This is the expected result of the conjunction of tests on the *car* and *cdr* fields.

Example 12 shows how programmers can simultaneously abstract over the use of both predicates and selectors:

Example 12

```
(define carnum?
  (λ: ([x : ⟨⊤, ⊤⟩]) (number? (car x))))
```

The *carnum?* predicate tests if the *car* of its argument is a **Number**, and its type must capture this fact.

Reasoning Logically Of course, we *do* learn something when conjunctions such as those in examples 5 and 6 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. This reasoning principle is used in multi-way conditionals, which is a common idiom extensively illustrated in *How to Design Programs* [Felleisen et al. 2001]:

Example 13

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

This program represents a common idiom. 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 effectively typecheck such programs, the type system must be able to follow this reasoning.

[3] Racket pairs are immutable; this reasoning is unsound for mutable pairs.

2.3 Putting it all Together

Our type system correctly handles all of the preceding examples. Finally, we combine these features into an example that demonstrates all aspects of our system:

Example 14

```
(λ: ([input : (⋃ Number String)]
     [extra : ⟨⊤, ⊤⟩])
  (cond
    [(and (number? input) (number? (car extra)))
     (+ input (car extra))]
    [(number? (car extra))
     (+ (string-length input) (car extra))]
    [else 0]))
```

In section 5.3, we return to this example with a type system that checks it correctly.

3. How to Check the Examples

Next we use the preceding examples to explain the basic ideas of our new system for occurrence typing.

3.1 Propositions and Objects

Recall example 1:

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

In this example, the typechecker must propagate information from the test to the then branch. Therefore, the typechecker really proves the proposition that "if the test evaluates to #t, then *x* is a number", a proposition abbreviated as $\mathbf{N}_x$, with **N** short for **Number**. The typechecker then uses this proposition to check the then branch.

The proposition $\mathbf{N}_x$ is computed from (*number? x*) by combining information from two sources. On one hand, the type of *number?* includes the information that it is a predicate. On the other, testing *x* produces information about the variable *x*.

The addition of a proposition as part of the type of the *number?* function accomplishes the first goal. Specifically, the added proposition allows the type of a function to describe what propositions are derivable when the function produces a true value. Borrowing terminology from work on effect systems [Lucassen and Gifford 1988], we refer to these propositions as *latent*. Borrowing notation from dependent types, we name the argument in each function, allowing latent propositions to be well-scoped in function types. If the argument to *number?* is named *y*, the latent proposition is $\mathbf{N}_y$. This makes the type of *number?*:

$$y{:}\top \xrightarrow{\mathbf{N}_y} \mathbf{B}$$

To satisfy the second goal, we modify the type system so that it derives an *object* for each expression. The object describes which part of the environment an expression accesses. In our example, it is simply x.

Given these pieces of information, the typechecker obtains the desired proposition about a predicate application from the substitution of the actual object for the formal parameter in the latent proposition. For the first example, the result is $\mathbf{N}_x$.

In example 2, *x* initially has the type (⋃ **String Number**). To check the else branch, the typechecker needs the information that *x* is not a **Number**; i.e., that it is a **String**. It computes this information via *two* propositions, one for each of the then and else branches. For the then branch the proposition is $\mathbf{N}_x$, as above. For the else branch, the type checker must propagate the proposition "*x* is not a **Number**"—written $\overline{\mathbf{N}}_x$—from the test to the else branch. To this end, function types are actually equipped with *two* latent propositions: one for when the function produces a true value, and

one for when it produces a false value. Thus, the type of *number?* is now

$$y{:}\top \xrightarrow{\mathbf{N}_y \mid \overline{\mathbf{N}}_y} \mathbf{B}$$

with the two propositions separated by $|$. Substituting x for y in the latent propositions produces the desired results.

Contrary to appearances, pairs of propositions need not be complementary. Recall (**and** (*number? x*) (> *x* 100)) from section 1. In this case, the then proposition should be $\mathbf{N}_x$, because if the **and** expression produces #t, *x* must be a number. But the else proposition *cannot* be $\overline{\mathbf{N}}_x$, since *x* might have been 97, which would produce #f but is nonetheless a number.

3.2 Handling Complex Tests

For complex tests, such as those of example 4, the type system combines the propositions of different subexpressions. In the cited example, the propositions for (*number? x*) and (*string? x*) are $\mathbf{N}_x|\overline{\mathbf{N}}_x$ and $\mathbf{S}_x|\overline{\mathbf{S}}_x$, respectively. For the **or** expression, these should be combined to $\mathbf{N}_x \vee \mathbf{S}_x$ for the then branch and $\overline{\mathbf{N}}_x \wedge \overline{\mathbf{S}}_x$ for the else branch.

From these complex propositions, the typechecker derives propositions about the type of *x*. If *x* is a number or a string, *x* is in $(\bigcup \mathbf{N}\ \mathbf{S})$. By codifying this as a rule of inference, it is possible to derive $(\bigcup \mathbf{N}\ \mathbf{S})_x$ from $\mathbf{N}_x \vee \mathbf{S}_x$, just what is needed to check the then branch. From $\overline{\mathbf{N}}_x \wedge \overline{\mathbf{S}}_x$ it is similarly possible to derive $\overline{(\bigcup \mathbf{N}\ \mathbf{S})}_x$, as expected for the else branch. To propagate propositions, we use a proposition environment instead of a type environment; the type environment becomes a special case.

Examples 5 and 6 are dealt with in the same manner, but with conjunction instead of disjunction. In example 7, the test expression of the outer **if** is itself an **if** expression. The typechecker must derive propositions from it and propagate them to the then and else branches. Thus, it first computes the propositions derived for each of the three subexpressions, giving $\mathbf{N}_x|\overline{\mathbf{N}}_x$ for the test and $\mathbf{S}_y|\overline{\mathbf{S}}_y$ for the then branch. Since the else branch—a plain #f—never produces a true value, the relevant propositions are $\mathbb{ff}$ and $\mathbb{tt}$, the impossible and trivial propositions.

3.3 Abstracting over Predicates

The next challenge, due to example 8, is to include proposition information in function types for user-defined predicates:

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

As explained above, the typechecker assigns the body of the function type **B** and derives $(\bigcup \mathbf{N}\ \mathbf{S})_x|\overline{(\bigcup \mathbf{N}\ \mathbf{S})}_x$ as the then and else propositions. To add these to a function type, it merely moves these propositions into the arrow type:

$$x{:}\top \xrightarrow{(\bigcup \mathbf{N}\ \mathbf{S})_x \mid \overline{(\bigcup \mathbf{N}\ \mathbf{S})}_x} \mathbf{B}$$

The key to the simplicity of this rule is that the bound variable of the **λ** expression becomes the name of the argument in the function type, keeping the propositions well-scoped.

3.4 Variables as Tests

In examples 3 and 9, the test expression is just a variable. For such cases, the typechecker uses the proposition $\mathbf{\#f}_x$ in the else branch to indicate that variable *x* has the value #f. Conversely, in the then branch, the variable must be true, giving the proposition $\overline{\mathbf{\#f}}_x$.

Example 9 demands an additional step. In the then branch of the **if**, *tmp* must be true. But this implies that (*number? x*) must also be true; the propsition representing this implication, $\overline{\mathbf{\#f}}_{tmp} \supset \mathbf{N}_x$, is added to the environment used to check the body of the **let** expression.

3.5 Selectors

The essence of example 10 is the application of predicates to selector expressions, e.g., (*car p*). Our type system represents such expressions as complex *objects*. For example, (*number?* (*car p*)) involves a predicate with latent propositions $\mathbf{N}_x|\overline{\mathbf{N}}_x$ applied to an expression whose object indicates that it accesses the *car* field of *p*. We write $\mathbf{car}(p)$ for this object. Thus, the entire expression has proposition $\mathbf{N}_{\mathbf{car}(p)}$ for the then branch and $\overline{\mathbf{N}}_{\mathbf{car}(p)}$ for the else branch, obtained by substituting $\mathbf{car}(p)$ for x in the latent propositions. Combinations of such tests (example 11) and abstraction over them (example 12) work as before.

To specify the access behavior of selectors, each function type is equipped with a latent object, added below the arrow. For *car*, it is **car**. But, since selectors can be composed arbitrarily, the function

(**λ:** ([*x* **:** ⟨⊤, ⟨⊤, ⊤⟩⟩]) (*car* (*cdr x*)))

has the type

$$x{:}\langle \top, \langle \top, \top \rangle \rangle \xrightarrow[\mathbf{car}(\mathbf{cdr}(x))]{\overline{\mathbf{\#f}}_{\mathbf{car}(\mathbf{cdr}(x))} \mid \mathbf{\#f}_{\mathbf{car}(\mathbf{cdr}(x))}} \top$$

3.6 Reasoning Logically

Next we revisit conjunctions such as (**and** (*number? x*) (*string? y*)). If this expression evaluates to #f, the typechecker *can* infer some propositions about *x* and *y*. In particular, since the original expression derived the proposition $\overline{\mathbf{N}}_x \vee \overline{\mathbf{S}}_y$ for the else branch, the type system can combine this environmental information with the results of subsequent tests. In example 13, the type system derives the proposition $\mathbf{N}_x$ when the second **cond** clause produces true, which means $\mathbf{S}_y$ holds, too. In short, maintaining propositions in the environment allows the typechecker to simulate the reasoning of the programmer and to track the many facts available for deducing type correctness for an expression.

3.7 The Form of the Type System

The essence of our discussion can be distilled into five ideas:

- Propositions express relationships *between* variables and types.
- Instead of type environments, we use *proposition environments*.
- Typechecking an expression computes two *propositions*, which 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*.
- *Latent* propositions and objects are attached to function types in order to describe facts about the result of applying the function.

The next sections translate these ideas into a typed calculus, λ_{TR}.

4. The Base Calculus

We begin our presentation of λ_{TR} with the base system, a typed lambda calculus with booleans, numbers, and conditionals. In Section 5, we extend the system with pairs and local variable binding.

The fundamental judgment of the type system is

$$\Gamma \vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$$

It states that in environment Γ, the expression e has type τ, comes with *then proposition* ψ_+ and *else proposition* ψ_-, and references *object* o. That is, if e evaluates to a true value, then proposition ψ_+ holds; conversely, if e evaluates to a false value, ψ_- is true. Further, if e evaluates to a value, then looking up o in the runtime environment produces the same value.

$$
\begin{array}{lll}
d, e ::= x \mid (e\ e) \mid \lambda x^{\tau}.e \mid (\textbf{if}\ e\ e\ e) \mid c \mid \#\mathtt{t} \mid \#\mathtt{f} \mid n & & \text{Expressions} \\
c \quad ::= \mathit{add1} \mid \mathit{zero?} \mid \mathit{number?} \mid \mathit{boolean?} \mid \mathit{procedure?} & & \text{Primitive Operations} \\
\sigma, \tau ::= \top \mid \mathbf{N} \mid \textbf{\#t} \mid \textbf{\#f} \mid (\bigcup\ \overrightarrow{\tau}) \mid x{:}\tau \xrightarrow[o]{\psi \mid \psi} \tau & & \text{Types} \\
\psi \quad ::= \tau_x \mid \overline{\tau}_x \mid \psi \supset \psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathtt{fff} \mid \mathtt{ttt} & & \text{Propositions} \\
o \quad ::= x \mid \emptyset & & \text{Objects} \\
\Gamma \quad ::= \overrightarrow{\psi} & & \text{Environments}
\end{array}
$$

Figure 1. Syntax of Terms, Types, Propositions, and Objects

4.1 Syntax

The syntax of expressions, types, propositions, and objects is given in figure 1. The expression syntax is standard, with conditionals, numeric and boolean constants, and primitive operators, in addition to the basics of abstraction, application, and variable reference. Abstractions come with type annotations on the bound variable. The presentation of the standard operational semantics is deferred to section 6, in conjunction with the soundness proof.

As for **types**, $\top$ is the supertype of all types; **N** is the type of numeric values; **#t** and **#f** are the types of the true and false constants, respectively; and $(\bigcup\ \overrightarrow{\tau})$ is the untagged or "true" union of its components. We abbreviate $(\bigcup$ **#t #f**$)$ as **B** and $(\bigcup)$ as $\bot$. Function types name their arguments. This name is in scope in the latent propositions and objects of a function type—written above and below the arrow, respectively—and in the result type. The latent propositions are knowledge about the types of variables when the function produces a true or false value, respectively.

Most **propositions** come in familiar form, borrowed from propositional logic: implications, disjunctions, and conjunctions, plus always-true ($\mathtt{ttt}$) and always-false ($\mathtt{fff}$) propositions. Atomic propositions relate variables to their types: τ_x states that x has type τ; $\overline{\tau}_x$ states that x never assumes a value with type τ.

An **object** describes a portion of the runtime environment. In the base system, it is either a variable or the empty object. For example, the expression (*add1* 7) has object $\emptyset$ because it does not access any portion of the environment.

Environments are simply collections of arbitrary propositions.

Types and Propositions Unlike many systems that relate type systems and logic, λ_{TR} distinguishes types, propositions, and terms. Propositions state claims about the runtime environment and thus *relate* types and variables. This choice allows a simple and decidable logic to be used to derive types from propositions to achieve the desired expressiveness.

4.2 Typing Rules

Figures 2 and 3 specify the typing and subtyping rules.

Constants 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**, numeric constants are assigned the propositions $\mathtt{ttt}|\mathtt{fff}$, indicating that no new information is acquired when the number evaluates to a true value; if it evaluates to false, a contradiction is obtained. The rule for function constants, T-CONST, work in similar manner, though we use a δ_τ function to assign types to function constants. The boolean constants are given singleton types by T-TRUE and T-FALSE, along with propositions that reflect that $\#\mathtt{t}$ is always true, and $\#\mathtt{f}$ is always false. All of the constants have the object $\emptyset$, since none refer to any portion of the environment.

Variables The rule for typing variables, T-VAR, exploits the proof system. If the current environment proves that x has type τ, represented by the proposition τ_x, then the type system assigns x the type τ. The object for a variable is itself. Finally, the propositions for a variable indicates that if x evaluates to a true value, x cannot have type **#f**. Similarly, if x evaluates to $\#\mathtt{f}$, its type is **#f**.

Abstraction and Application The rule for checking an abstraction, T-ABS, takes the propositions and object from the body and makes them the latent propositions and object in the function type. By taking the bound variable from the abstraction and turns it into the name of the argument, references to the variable in the types, propositions, and object remain well-scoped.

Correspondingly, in T-APP, the latent propositions and object are used as the result propositions and object, just as with the result type. In all cases, the actual object o_a is substituted for the name of the formal parameter, x. Consider the abstraction:

$$\lambda y^{\top}.(\mathit{number?}\ y)$$

which typechecks as follows. In the body of the abstraction, $\Gamma = \top_y$. Thus, $\Gamma \vdash y : \top$; $\overline{\textbf{\#f}}_y|\textbf{\#f}_y$; y, and $\mathit{number?}$ has the above-mentioned type. To check the application, the typechecker substitutes y for x in the result type, latent propositions, and latent object of $\mathit{number?}$, which yields $\Gamma \vdash (\mathit{number?}\ y)$:**B** ; $\mathbf{N}_y|\overline{\mathbf{N}}_y$; $\emptyset$. Finally, the function is assigned the desired type via T-ABS:

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

In our prior system, this example required multiple special-purpose rules and the use of several metafunctions, whereas here it is a simple matter of scope and substitution.

Of course, substitution of an object for a variable must account for the case when the object is $\emptyset$. When this happens, any references to the variable are forgotten, and propositions or objects that refer to it become trivial. Figure 8 defines the full substitution function.

Conditionals As far as types and objects are concerned, the T-IF rule is straightforward. The test may have any type and object, and the then and else branches must have identical types and objects, which then become the type and the object of the entire expression. The key difference between T-IF and conventional rules for conditionals is due to the differential propagation of knowledge from the test to the branches. Specifically, the rule uses two distinct environments to check the then and else branches, because ψ_{1_+} holds in the then branch and ψ_{1_-} holds in the else branch.

The resulting propositions follow from a simple principle about the evaluation of **if**. If a true value is produced, either the then branch or the else branch must have evaluated to a true value, and similarly for a false value. Therefore, in the true case, either the then proposition of the then branch, ψ_{2_+}, or the then proposition of the else branch, ψ_{3_+}, must be true, which means $\psi_{2_+} \vee \psi_{3_+}$ is the then proposition of the entire expression and, correspondingly, $\psi_{2_-} \vee \psi_{3_-}$ is the else proposition.

Subsumption & Subtyping Finally, λ_{TR} comes with subtyping. Expressions of type τ can be viewed as having a larger type τ'. Objects can also be lifted to larger objects. The ordering on propositions is simply provability in the current environment.

$$\text{T-NUM}\quad \Gamma \vdash n : \mathbf{N} \; ; \; \mathtt{tt}|\mathtt{ff} \; ; \; \emptyset \qquad \text{T-CONST}\quad \Gamma \vdash c : \delta_\tau(c) \; ; \; \mathtt{tt}|\mathtt{ff} \; ; \; \emptyset$$

$$\text{T-TRUE}\quad \Gamma \vdash \#\mathsf{t} : \mathbf{\#t} \; ; \; \mathtt{tt}|\mathtt{ff} \; ; \; \emptyset \qquad \text{T-FALSE}\quad \Gamma \vdash \#\mathsf{f} : \mathbf{\#f} \; ; \; \mathtt{ff}|\mathtt{tt} \; ; \; \emptyset$$

$$\text{T-VAR}\quad \frac{\Gamma \vdash \tau_x}{\Gamma \vdash x : \tau \; ; \; \overline{\mathbf{\#f}}_x|\mathbf{\#f}_x \; ; \; x}$$

$$\text{T-ABS}\quad \frac{\Gamma, \sigma_x \vdash e : \tau \; ; \; \psi_+|\psi_- \; ; \; o}{\Gamma \vdash \lambda x^\sigma.e : x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau \; ; \; \mathtt{tt}|\mathtt{ff} \; ; \; \emptyset}$$

$$\text{T-APP}\quad \frac{\Gamma \vdash e : x{:}\sigma \xrightarrow[o_f]{\psi_{f_+}|\psi_{f_-}} \tau \; ; \; \psi_+|\psi_- \; ; \; o \qquad \Gamma \vdash e' : \sigma \; ; \; {\psi_+}'|{\psi_-}' \; ; \; o'}{\Gamma \vdash (e \; e') : \tau[o'/x] \; ; \; \psi_{f_+}|\psi_{f_-}[o'/x] \; ; \; o_f[o'/x]}$$

$$\text{T-IF}\quad \frac{\Gamma \vdash e_1 : \tau_1 \; ; \; \psi_{1_+}|\psi_{1_-} \; ; \; o_1 \qquad \Gamma, \psi_{1_+} \vdash e_2 : \tau \; ; \; \psi_{2_+}|\psi_{2_-} \; ; \; o \qquad \Gamma, \psi_{1_-} \vdash e_3 : \tau \; ; \; \psi_{3_+}|\psi_{3_-} \; ; \; o}{\Gamma \vdash (\textbf{if} \; e_1 \; e_2 \; e_3) : \tau \; ; \; \psi_{2_+} \vee \psi_{3_+}|\psi_{2_-} \vee \psi_{3_-} \; ; \; o}$$

$$\text{T-SUBSUME}\quad \frac{\Gamma \vdash e : \tau \; ; \; \psi_+|\psi_- \; ; \; o \qquad \Gamma, \psi_+ \vdash \psi'_+ \qquad \Gamma, \psi_- \vdash \psi'_- \qquad \vdash \tau <: \tau' \qquad \vdash o <: o'}{\Gamma \vdash e : \tau' \; ; \; \psi'_+|\psi'_- \; ; \; o'}$$

Figure 2. Typing Rules

$$\text{SO-REFL}\quad \vdash o <: o \qquad \text{SO-TOP}\quad \vdash o <: \emptyset \qquad \text{S-REFL}\quad \vdash \tau <: \tau \qquad \text{S-TOP}\quad \vdash \tau <: \top$$

$$\text{S-UNIONSUPER}\quad \frac{\exists i. \vdash \tau <: \sigma_i}{\vdash \tau <: (\bigcup \; \overrightarrow{\sigma}^i)} \qquad \text{S-UNIONSUB}\quad \frac{\overrightarrow{\vdash \tau_i <: \sigma}^i}{\vdash (\bigcup \; \overrightarrow{\tau}^i) <: \sigma}$$

$$\text{S-FUN}\quad \frac{\vdash \sigma' <: \sigma \qquad \vdash \tau <: \tau' \qquad \psi_+ \vdash {\psi_+}' \qquad \psi_- \vdash {\psi_-}' \qquad \vdash o <: o'}{\vdash x{:}\sigma \xrightarrow[o]{\psi_+|\psi_-} \tau <: x{:}\sigma' \xrightarrow[o']{{\psi_+}'|{\psi_-}'} \tau'}$$

Figure 3. Subtyping Rules

Given these definitions, the rules for subtyping are straightforward. All types are subtypes of $\top$ and of themselves. Subtypes of elements of a union are subtypes of the union, and any type that is a supertype of every element is a supertype of the union. Finally, function types are ordered in the usual fashion.

4.3 Proof System

Figure 4 specifies the proof rules for our logic. The first nine rules—L-ATOM through L-ORE—use the natural deduction style to express the standard rules of propositional logic.

The subsequent four rules relate the atomic propositions. In particular, L-SUB says that if x has type τ, then it has any larger type. Similarly, L-SUBNOT says that if x does not have type τ, then it does not have any smaller type. By L-BOT, if x has an empty type, it is possible to conclude anything since this is impossible.

The L-UPDATE rule refines the type of a variable via a combination of multiple propositions. Roughly speaking, this metafunction satisfies the equations

$$\mathsf{update}(\tau, \sigma) = \tau \cap \sigma \qquad \mathsf{update}(\tau, \overline{\sigma}) = \tau - \sigma$$

See figure 9 for the full definition.

4.4 A Worked Example

At this point, eight of our 14 examples typecheck. To illustrate the workings of the type system, let us work example 7:

(**if** (**if** (*number? x*) (*string? y*) $\#$f)
  ($+$ *x* (*string-length y*))
  0)

First, assume that the initial environment is $\Gamma = \top_x, \top_y$. Now consider the inner **if** expression. The test has then proposition $\mathbf{N}_x$ and else proposition $\overline{\mathbf{N}}_x$. The then branch has propositions $\mathbf{S}_y$ and $\overline{\mathbf{S}}_y$, or by subsumption $\mathbf{N}_x \wedge \mathbf{S}_y|\mathtt{tt}$, since T-IF adds the then proposition of the test to the environment for checking the then branch. The else branch has propositions $\mathtt{ff}|\mathtt{tt}$, and by subsumption $\mathbf{N}_x \wedge \mathbf{S}_y|\mathtt{tt}$ since $\mathtt{ff} \vdash \mathbf{N}_x \wedge \mathbf{S}_y$. Therefore, the entire inner **if** expression has then proposition

$$(\mathbf{N}_x \wedge \mathbf{S}_y) \vee (\mathbf{N}_x \wedge \mathbf{S}_y) = \mathbf{N}_x \wedge \mathbf{S}_y$$

and else proposition $\mathtt{tt}$.

Second, we typecheck the then branch of the main **if** expression in the environment $\Gamma_1 = \top_x, \top_y, \mathbf{N}_x \wedge \mathbf{S}_y$. Since $\Gamma_1 \vdash \mathbf{N}_x$ and $\Gamma_1 \vdash \mathbf{S}_y$, we can give x and y the appropriate types to check the expression ($+$ *x* (*string-length y*)).

5. Extensions

The base system of section 4 lacks several important features, including support for compound data structures and **let**. This section shows how to extend the base system with these features.

5.1 Pairs

The most significant extension concerns compound data, e.g., pairs. We extend the expression, type, and proposition grammars as shown in figure 5.[4] Most significantly, in all places where a variable appeared previously in propositions and objects, it is now legal to specify a path—a sequence of selectors—rooted at a variable, written $\pi(x)$. This allows the system to refer not just to variables in the environment, but to parts of their values.

Typing Rules Figure 6 shows the extensions to the typing and subtyping rules. Again, the subtyping rule S-PAIR and typing rule for *cons* are straightforward; all pair values are treated as true. The T-CAR and T-CDR rules are versions of the application rule specialized to the appropriate latent propositions and objects, which here involve non-trivial paths. Substitution of objects for variables is also appropriately extended; the full definition is in figure 8.

None of the existing typing rules require changes.

[4] In a polymorphic λ_{TR}, pair operations could be added as primitives.

$$\text{L-ATOM}\ \frac{\psi \in \Gamma}{\Gamma \vdash \psi} \qquad \text{L-TRUE}\ \Gamma \vdash \mathtt{tt} \qquad \text{L-FALSE}\ \frac{\Gamma \vdash \mathtt{ff}}{\Gamma \vdash \psi} \qquad \text{L-ANDI}\ \frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \wedge \psi_2} \qquad \text{L-ANDE}\ \frac{\Gamma, \psi_1 \vdash \psi \text{ or } \Gamma, \psi_2 \vdash \psi}{\Gamma, \psi_1 \wedge \psi_2 \vdash \psi}$$

$$\text{L-IMPI}\ \frac{\Gamma, \psi_1 \vdash \psi_2}{\Gamma \vdash \psi_1 \supset \psi_2} \qquad \text{L-IMPE}\ \frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_1 \supset \psi_2}{\Gamma \vdash \psi_2} \qquad \text{L-ORI}\ \frac{\Gamma \vdash \psi_1 \text{ or } \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \vee \psi_2} \qquad \text{L-ORE}\ \frac{\Gamma, \psi_1 \vdash \psi \quad \Gamma, \psi_2 \vdash \psi}{\Gamma, \psi_1 \vee \psi_2 \vdash \psi}$$

$$\text{L-SUB}\ \frac{\Gamma \vdash \tau_x \qquad \vdash \tau <: \sigma}{\Gamma \vdash \sigma_x} \qquad \text{L-SUBNOT}\ \frac{\Gamma \vdash \overline{\sigma}_x \qquad \vdash \tau <: \sigma}{\Gamma \vdash \overline{\tau}_x} \qquad \text{L-BOT}\ \frac{\Gamma \vdash \bot_x}{\Gamma \vdash \psi} \qquad \text{L-UPDATE}\ \frac{\Gamma \vdash \tau_x \qquad \Gamma \vdash \nu_x}{\Gamma \vdash \mathsf{update}(\tau, \nu)_x}$$

(The metavariable ν ranges over τ and $\overline{\tau}$ (without variables).)

Figure 4. Proof System

$e ::= \ldots \mid (cons\ e\ e)$	Expressions	$\psi ::= \ldots \mid \tau_{\pi(x)} \mid \overline{\tau}_{\pi(x)}$	Propositions
$c ::= \ldots \mid cons? \mid car \mid cdr$	Primitive Operations	$o ::= \pi(x) \mid \emptyset$	Objects
$\sigma, \tau ::= \ldots \mid \langle \tau, \tau \rangle$	Types	$\pi ::= \overrightarrow{pe}$	Paths
		$pe ::= \mathbf{car} \mid \mathbf{cdr}$	Path Elements

Figure 5. Syntax Extensions for Pairs

$$\text{S-PAIR}\ \frac{\vdash \tau_1 <: \tau_2 \quad \vdash \sigma_1 <: \sigma_2}{\vdash \langle \tau_1, \sigma_1 \rangle <: \langle \tau_2, \sigma_2 \rangle} \qquad \text{T-CONS}\ \frac{\Gamma \vdash e_1 : \tau_1\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1 \quad \Gamma \vdash e_2 : \tau_2\ ;\ \psi_{2_+}|\psi_{2_-}\ ;\ o_2}{\Gamma \vdash (cons\ e_1\ e_2) : \langle \tau_1, \tau_2 \rangle\ ;\ \mathtt{tt}|\mathtt{ff}\ ;\ \emptyset}$$

$$\text{T-CAR}\ \frac{\Gamma \vdash e : \langle \tau_1, \tau_2 \rangle\ ;\ \psi_{0_+}|\psi_{0_-}\ ;\ o \quad \psi_+|\psi_- = \overline{\mathbf{f}}_{\mathbf{car}(x)}|\mathbf{f}_{\mathbf{car}(x)}[o/x] \quad o_r = \mathbf{car}(x)[o/x]}{\Gamma \vdash (car\ e) : \tau_1\ ;\ \psi_+|\psi_-\ ;\ o_r}$$

$$\text{T-CDR}\ \frac{\Gamma \vdash e : \langle \tau_1, \tau_2 \rangle\ ;\ \psi_{0_+}|\psi_{0_-}\ ;\ o \quad \psi_+|\psi_- = \overline{\mathbf{f}}_{\mathbf{cdr}(x)}|\mathbf{f}_{\mathbf{cdr}(x)}[o/x] \quad o_r = \mathbf{cdr}(x)[o/x]}{\Gamma \vdash (cdr\ e) : \tau_2\ ;\ \psi_+|\psi_-\ ;\ o_r}$$

Figure 6. Type and Subtype Extensions

$$\text{L-SUB}\ \frac{\Gamma \vdash \tau_{\pi(x)} \qquad \vdash \tau <: \sigma}{\Gamma \vdash \sigma_{\pi(x)}} \qquad \text{L-SUBNOT}\ \frac{\Gamma \vdash \overline{\sigma}_{\pi(x)} \qquad \vdash \tau <: \sigma}{\Gamma \vdash \overline{\tau}_{\pi(x)}} \qquad \text{L-BOT}\ \frac{\Gamma \vdash \bot_{\pi(x)}}{\Gamma \vdash \psi} \qquad \text{L-UPDATE}\ \frac{\Gamma \vdash \tau_{\pi'(x)} \qquad \Gamma \vdash \nu_{\pi(\pi'(x))}}{\Gamma \vdash \mathsf{update}(\tau, \nu, \pi)_{\pi'(x)}}$$

Figure 7. Logic Extensions

Logic Rules Figure 7 specifies the changes to the logic for dealing with paths. For the first three rules, the only change needed is allowing paths in the appropriate syntactic locations. For the L-UPDATE rule, there is an additional change. When the environment proves both $\langle \top, \top \rangle_x$ and $\mathbf{N}_{\mathbf{car}(x)}$, it must be possible to derive $\langle \mathbf{N}, \top \rangle_x$. The new version of L-UPDATE allows this inference via a revised version of update. Its third argument specifies a path to follow *before* refining the type. See figure 9 for details.

Of course, none of the rules implementing the standard proof theory of propositional logic change with this extension.

With the addition of pairs, the type system can cope with 12 of the 14 examples from section 2.

5.2 Local Binding

To add a local binding construct, we again extend the grammar:

$$d, e ::= \ldots \mid (\mathbf{let}\ (x\ e)\ e)$$

Recall our motivating example 9. The crucial aspect is to relate the propositions about the initialization expression to the variable itself. Logical implication precisely expresses this connection, giving us the following rule:

$$\text{T-LET}\ \frac{\Gamma \vdash e_0 : \tau\ ;\ \psi_{0_+}|\psi_{0_-}\ ;\ o_0 \quad \Gamma, \tau_x, \overline{\mathbf{f}}_x \supset \psi_{0_+}, \mathbf{f}_x \supset \psi_{0_-} \vdash e_1 : \sigma\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1}{\Gamma \vdash (\mathbf{let}\ (x\ e_0)\ e_1) : \sigma[o_0/x]\ ;\ \psi_{1_+}|\psi_{1_-}[o_0/x]\ ;\ o_1[o_0/x]}$$

This rule has three components. The first antecedent checks the right-hand side. The second checks the body with an environment extended both with the type of the bound variable (τ_x) and with implications stating that if x is not false, e_0 must evaluate to true, and similarly if x is false, e_0 must evaluate to false. The consequence replaces all references to x with the object of e_0.

5.3 The Final Example

With this extension, we are now able to check all the examples from section 2. To demonstrate the complete system, consider example 14. We begin with $\Gamma_0 = (\bigcup \mathbf{N}\ \mathbf{S})_{input}, \langle \top, \top \rangle_{extra}$. The two tests, (*number? input*) and (*number?* (*car extra*)), yield the propositions $\mathbf{N}_{input}|\overline{\mathbf{N}}_{input}$ for the former and $\mathbf{N}_{\mathbf{car}(extra)}|\overline{\mathbf{N}}_{\mathbf{car}(extra)}$ for the latter. Using T-IF, T-SUBSUME, and the definition of **and**

$$
\begin{array}{lll}
\psi_+|\psi_-[o/x] & = \psi_+[o/x]|\psi_-[o/x] & \\
\nu_{\pi(x)}[\pi'(y)/x] & = (\nu[\pi'(y)/x])_{\pi(\pi'(y))} & \\
\nu_{\pi(x)}[\emptyset/x]_+ & = \mathtt{tt} & \\
\nu_{\pi(x)}[\emptyset/x]_- & = \mathtt{ff} & \\
\nu_{\pi(x)}[o/z] & = \nu_{\pi(x)} & x \neq z \text{ and } z \notin \mathsf{fv}(\nu) \\
\nu_{\pi(x)}[o/z]_+ & = \mathtt{tt} & x \neq z \text{ and } z \in \mathsf{fv}(\nu) \\
\nu_{\pi(x)}[o/z]_- & = \mathtt{ff} & x \neq z \text{ and } z \in \mathsf{fv}(\nu) \\
\mathtt{tt}[o/x] & = \mathtt{tt} & \\
\mathtt{ff}[o/x] & = \mathtt{ff} & \\
(\psi_1 \supset \psi_2)[o/x]_+ & = \psi_1[o/x]_- \supset \psi_2[o/x]_+ & \\
(\psi_1 \supset \psi_2)[o/x]_- & = \psi_1[o/x]_+ \supset \psi_2[o/x]_- & \\
(\psi_1 \vee \psi_2)[o/x] & = \psi_1[o/x] \vee \psi_2[o/x] & \\
(\psi_1 \wedge \psi_2)[o/x] & = \psi_1[o/x] \wedge \psi_2[o/x] & \\
\pi(x)[\pi'(y)/x] & = \pi(\pi'(y)) & \\
\pi(x)[\emptyset/x] & = \emptyset & \\
\pi(x)[o/z] & = \pi(x) & x \neq z \\
\emptyset[o/x] & = \emptyset &
\end{array}
$$

Substitution on types is capture-avoiding structural recursion.

Figure 8. Substitution

$$
\begin{array}{lll}
\mathsf{update}(\langle \tau, \sigma \rangle, \nu, \pi :: \mathbf{car}) & = \langle \mathsf{update}(\tau, \nu, \pi), \sigma \rangle & \\
\mathsf{update}(\langle \tau, \sigma \rangle, \nu, \pi :: \mathbf{cdr}) & = \langle \tau, \mathsf{update}(\sigma, \nu, \pi) \rangle & \\
\mathsf{update}(\tau, \sigma, \epsilon) & = \mathsf{restrict}(\tau, \sigma) & \\
\mathsf{update}(\tau, \overline{\sigma}, \epsilon) & = \mathsf{remove}(\tau, \sigma) & \\
 & & \\
\mathsf{restrict}(\tau, \sigma) & = \bot & \\
 & \text{if } \nexists v. \vdash v : \tau \ ; \ \psi_1 \ ; \ o_1 \text{ and } \vdash v : \sigma \ ; \ \psi_2 \ ; \ o_2 & \\
\mathsf{restrict}((\cup \overrightarrow{\tau}), \sigma) & = (\cup \overrightarrow{\mathsf{restrict}(\tau, \sigma)}) & \\
\mathsf{restrict}(\tau, \sigma) & = \tau & \text{if} \vdash \tau <: \sigma \\
\mathsf{restrict}(\tau, \sigma) & = \sigma & \text{otherwise} \\
 & & \\
\mathsf{remove}(\tau, \sigma) & = \bot & \text{if} \vdash \tau <: \sigma \\
\mathsf{remove}((\cup \overrightarrow{\tau}), \sigma) & = (\cup \overrightarrow{\mathsf{remove}(\tau, \sigma)}) & \\
\mathsf{remove}(\tau, \sigma) & = \tau & \text{otherwise}
\end{array}
$$

Figure 9. Type Update

(see example 7), we can therefore derive the then proposition $\mathbf{N}_{input} \wedge \mathbf{N}_{\mathbf{car}(extra)}$ and the else proposition $\overline{\mathbf{N}}_{input} \vee \overline{\mathbf{N}}_{\mathbf{car}(extra)}$. The then proposition, added to the environment for the right-hand side of the first **cond** clause, also proves $\langle \mathbf{N}, \top \rangle_{extra}$ by the L-UPDATE rule, which suffices for typechecking the expression (+ *input* (*car extra*)).

In the second **cond** clause, the test has the then proposition $\mathbf{N}_{\mathbf{car}(extra)}$, and the environment is $\Gamma_0, \overline{\mathbf{N}}_{input} \vee \overline{\mathbf{N}}_{\mathbf{car}(extra)}$ From this, we can derive $\overline{\mathbf{N}}_{input}$ since $\mathbf{N}_{\mathbf{car}(extra)}$ and $\overline{\mathbf{N}}_{\mathbf{car}(extra)}$ are contradictory. Then, using $(\bigcup \mathbf{N}\ \mathbf{S})_{input}$ from Γ_0 and $\overline{\mathbf{N}}_{input}$, we can derive $\mathbf{S}_{input}$, which is required to typecheck the application of *string-length*. This completes the second clause.

The third clause is a constant and thus obvious.

5.4 Metafunctions

Equipped with the full formal system, we can now provide a detailed description of the substitution and type update metafunctions; see figures 8 and 9.

Substitution replaces a variable with an object. When the object is of the form $\pi(x)$, this is in general a straightforward structural recursion. There are a few tricky cases to consider, however. First, if the object being substituted is $\emptyset$, then references to the variable must be erased. In most contexts, such propositions should be erased to $\mathtt{tt}$, the trivial proposition. But, just as with contravariance of function types, such propositions must be erased to $\mathtt{ff}$ when to the left of an odd number of implications. Second, if a proposition such as τ_x references a variable z in τ, then if z goes out of scope, the entire proposition must be erased.

In comparison, the update metafunction is simple. It follows a path into its first argument and then appropriately replaces the type there with a type that depends on the second argument. If the second argument is of the form τ, update computes the intersection of the two types; if the second argument is of the form $\overline{\tau}$, update computes the difference. To compute the intersection and difference, update uses the auxiliary metafunctions restrict and remove, respectively.

$$
\begin{array}{ll}
\delta_\tau(number?) & = x{:}\top \xrightarrow[\emptyset]{\mathbf{N}_x \,|\, \overline{\mathbf{N}}_x} \mathbf{B} \\
\delta_\tau(procedure?) & = x{:}\top \xrightarrow[\emptyset]{\bot \to \top \,|\, \overline{\bot \to \top}} \mathbf{B} \\
\delta_\tau(boolean?) & = x{:}\top \xrightarrow[\emptyset]{\mathbf{B}_x \,|\, \overline{\mathbf{B}}_x} \mathbf{B} \\
\delta_\tau(cons?) & = x{:}\top \xrightarrow[\emptyset]{\langle \top, \top \rangle_x \,|\, \overline{\langle \top, \top \rangle}_x} \mathbf{B} \\
\delta_\tau(add1) & = \mathbf{N} \to \mathbf{N} \\
\delta_\tau(zero?) & = \mathbf{N} \to \mathbf{B}
\end{array}
$$

Figure 10. Constant Typing

$$
\begin{array}{lll}
\delta(add1, n) & = n + 1 & \\
\delta(zero?, 0) & = \#\mathsf{t} & \\
\delta(zero?, n) & = \#\mathsf{f} & \text{otherwise} \\
\delta(number?, n) & = \#\mathsf{t} & \\
\delta(number?, v) & = \#\mathsf{f} & \text{otherwise} \\
\delta(boolean?, \#\mathsf{t}) & = \#\mathsf{t} & \\
\delta(boolean?, \#\mathsf{f}) & = \#\mathsf{t} & \\
\delta(boolean?, v) & = \#\mathsf{f} & \text{otherwise} \\
\delta(procedure?, \lambda x^{\tau}.e) & = \#\mathsf{t} & \\
\delta(procedure?, c) & = \#\mathsf{t} & \\
\delta(procedure?, v) & = \#\mathsf{f} & \text{otherwise} \\
\delta(cons?, (cons\ v_1\ v_2)) & = \#\mathsf{t} & \\
\delta(cons?, v) & = \#\mathsf{f} & \text{otherwise} \\
\delta(car, (cons\ v_1\ v_2)) & = v_1 & \\
\delta(cdr, (cons\ v_1\ v_2)) & = v_2 &
\end{array}
$$

Figure 13. Primitives

6. Semantics, Models, and Soundness

To prove type soundness, we introduce an environment-based operational semantics, use the environments to construct a model for the logic, prove the soundness of the logic with respect to this model, and conclude the type soundness argument from there.

6.1 Operational Semantics

Figure 11 defines a big-step, environment-based operational semantics of λ_{TR}. The metavariable ρ ranges over *value environments* (or just environments), which are finite maps from variables

$$\text{B-VAR}\ \frac{\rho(x) = v}{\rho \vdash x \Downarrow v} \qquad \text{B-DELTA}\ \frac{\rho \vdash e \Downarrow c \qquad \rho \vdash e' \Downarrow v \qquad \delta(c, v) = v'}{\rho \vdash (e\ e') \Downarrow v'} \qquad \text{B-LET}\ \frac{\rho \vdash e_a \Downarrow v_a \qquad \rho[x \mapsto v_a] \vdash e_b \Downarrow v}{\rho \vdash (\textbf{let}\ (x\ e_a)\ e_b) \Downarrow v}$$

$$\text{B-VAL}\ \rho \vdash v \Downarrow v \qquad \text{B-ABS}\ \rho \vdash \lambda x^{\tau}.e \Downarrow [\rho, \lambda x^{\tau}.e]$$

$$\text{B-BETA}\ \frac{\rho \vdash e_f \Downarrow [\rho_c, \lambda x^{\tau}.e_b] \qquad \rho \vdash e_a \Downarrow v_a \qquad \rho_c[x \mapsto v_a] \vdash e_b \Downarrow v}{\rho \vdash (e_f\ e_a) \Downarrow v} \qquad \text{B-CONS}\ \frac{\rho \vdash e_1 \Downarrow v_1 \qquad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash (cons\ e_1\ e_2) \Downarrow (cons\ v_1\ v_2)}$$

$$\text{B-IFTRUE}\ \frac{\rho \vdash e_1 \Downarrow v_1 \qquad v_1 \neq \#\mathsf{f} \qquad \rho \vdash e_2 \Downarrow v}{\rho \vdash (\textbf{if}\ e_1\ e_2\ e_3) \Downarrow v} \qquad \text{B-IFFALSE}\ \frac{\rho \vdash e_1 \Downarrow \#\mathsf{f} \qquad \rho \vdash e_3 \Downarrow v}{\rho \vdash (\textbf{if}\ e_1\ e_2\ e_3) \Downarrow v}$$

Figure 11. Operational Semantics

$$\text{M-OR}\ \frac{\rho \models \psi_1 \text{ or } \rho \models \psi_2}{\rho \models \psi_1 \vee \psi_2} \qquad \text{M-IMP}\ \frac{\rho \models \psi \text{ implies } \rho \models \psi'}{\rho \models \psi \supset \psi'} \qquad \text{M-AND}\ \frac{\rho \models \psi \qquad \rho \models \psi'}{\rho \models \psi \wedge \psi'} \qquad \text{M-TOP}\ \rho \models \mathbb{t}\mathbb{t}$$

$$\text{M-TYPE}\ \frac{\vdash \rho(\pi(x)) : \tau\ ;\ \psi_+|\psi_-\ ;\ o}{\rho \models \tau_{\pi(x)}} \qquad \text{M-NOTTYPE}\ \frac{\vdash \rho(\pi(x)) : \sigma\ ;\ \psi_+|\psi_-\ ;\ o \qquad \text{there is no } v \text{ such that } \vdash v : \tau\ ;\ \psi_{1_+}|\psi_{1_-}\ ;\ o_1 \text{ and } \vdash v : \sigma\ ;\ \psi_{2_+}|\psi_{2_-}\ ;\ o_2}{\rho \models \overline{\tau}_{\pi(x)}}$$

Figure 12. Satisfaction Relation

to closed values. We write $\rho(x)$ for the value of x in ρ, and $\rho(\pi(x))$ for the value at path π in $\rho(x)$. The central judgment is

$$\rho \vdash e \Downarrow v$$

which states that in environment ρ, the expression e evaluates to the value v. Values are given by the following grammar:

$$v ::= c \mid \#\mathsf{t} \mid \#\mathsf{f} \mid n \mid [\rho, \lambda x^{\tau}.e] \mid (cons\ v\ v)$$

For the interpretation of primitives, see figure 13.

6.2 Models

A model is any value environment, and an environment ρ *satisfies* a proposition ψ, $\rho \models \psi$, as defined in figure 12 mostly in the usual manner. The satisfaction relation is extended to proposition environments in a pointwise manner. To formulate the satisfaction relation, we need a typing rule for closures:

$$\text{T-CLOS}\ \frac{\exists \Gamma.\rho \models \Gamma \text{ and } \Gamma \vdash \lambda x^{\tau}.e : \sigma\ ;\ \psi_+|\psi_-\ ;\ o}{\vdash [\rho, \lambda x^{\tau}.e] : \sigma\ ;\ \psi_+|\psi_-\ ;\ o}$$

Two clauses in figure 12—M-TYPE and M-NOTTYPE—need some explanation. They state that if a value of x in the environment has the type τ, the model satisfies τ_x, and if x has a type that does not overlap with τ, the model satisfies $\overline{\tau}_x$.

We can see immediately that not all propositions are consistent, such as $\mathbb{ff}$, as expected, but also propositions such as $\mathbf{N}_x \wedge \mathbf{B}_x$.

Our first lemma says that the proof theory respects models.

Lemma 1. *If $\rho \models \Gamma$ and $\Gamma \vdash \psi$ then $\rho \models \psi$.*

Proof: Structural induction on $\Gamma \vdash \psi$. □

Conversely, we can use this lemma as the guideline that any logical rule that satisfies this lemma is appropriate.

6.3 Soundness for λ_{TR}

With the model theory and the operational semantics in place, we can state and prove the second major lemma.

Lemma 2. *If $\Gamma \vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$, $\rho \models \Gamma$, and $\rho \vdash e \Downarrow v$ then all of the following hold:*

1. *either $o = \emptyset$ or $\rho(o) = v$,*
2. *either $v \neq \#\mathsf{f}$ and $\rho \models \psi_+$ or $v = \#\mathsf{f}$ and $\rho \models \psi_-$, and*
3. *$\vdash v : \tau\ ;\ {\psi_+}'|{\psi_-}'\ ;\ o'$ for some ${\psi_+}', {\psi_-}', \text{ and } o'$.*

Proof: By induction on the derivation of the typing judgment.

For illustrative purposes, we examine the T-IF case with $e = (\textbf{if}\ e_1\ e_2\ e_3)$. Proving part 1 is trivial: either o is $\emptyset$, or both e_2 and e_3 have an identical object, and e must evaluate to the results of one of them. To prove part 2, we note that if $v = \#\mathsf{f}$, either e_2 or e_3 must evaluate to $\#\mathsf{f}$. If it is e_2, we have $\rho \models \psi_{2_-}$, and thus $\rho \models \psi_{2_-} \vee \psi_{3_-}$ by M-OR, giving the desired result. The cases for e_3 evaluating to false and the whole expression evaluating to true are dealt with in an analogous manner. Finally, part 3 is trivial, since both the then and else branches have the same type. □

Given this, we can state the desired soundness theorem.

Theorem 1 (Type Soundness for λ_{TR})**.** *If $\vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$ and $\vdash e \Downarrow v$ then $\vdash v : \tau\ ;\ {\psi_+}'|{\psi_-}'\ ;\ o'$ for some ${\psi_+}', {\psi_-}', \text{ and } o'$.*

Proof: Corollary of lemma 2. □

This theorem comes with the standard drawbacks of big-step soundness proofs. It says nothing about diverging or stuck terms.[5]

7. From λ_{TR} to Typed Racket

As a core calculus, λ_{TR} lacks many of the features of a programming language such as Racket, which consists of a rich functional core enriched with support for object-oriented and component-oriented programming. Creating an implementation from the calculus demands additional research and engineering.

This section reports on the most important implementation ideas. Our current implementation—dubbed Typed Racket—deals with the functional core of Racket, which also supports mutable

[5] To deal with errors, we would need the following additional steps:
1. Add an additional value, $wrong$, which has every type.
2. Add evaluation rules that propagate $wrong$.
3. Add evaluation rules to generate $wrong$ for each stuck state.
4. Add clauses to the δ function to generate $wrong$ for undefined clauses.
5. Prove that if $\vdash e : \tau\ ;\ \psi_+|\psi_-\ ;\ o$, then $\not\vdash e \Downarrow wrong$.

data structures, assignable variables and Scheme's multiple value returns. In addition, the section discusses user interface issues and the implementation of the logical reasoning system.

7.1 Paths and Mutability

The λ_{TR} calculus assumes immutable data structures. In Racket, some forms of data are mutable, however. Predicate tests on paths into mutable data cannot soundly refine the type environment. Consider an example using Racket's equivalent of ref cells:

```
(let*: ([b : (Box ⊤) (box 0)] [b* : (Box ⊤) b])
  (if (number? (unbox b))
      (begin (set-box! b* 'no) (unbox b))
      0))
```

A naive implementation might assign this fragment the type **Number**, but its evaluation produces a symbol, because b and $b*$ are aliased. To avoid this unsoundness, the *unbox* procedure is assigned trivial latent propositions and object. In general, Racket provides structures that are mutable on a per-field basis and that are accessed with per-field selectors. Typed Racket therefore does not assign the selectors of mutable fields any latent propositions or objects.

Assignable variables are a second concern in the same realm. If a variable is the target of an assignment, the L-UPDATE rule is unsound for this variable. Hence, Typed Racket uses a simple analysis to find all assignment statements and to disable the L-UPDATE rule for those. Since variables are assignable only in the module that declares them, the analysis is modular and straightforward.

7.2 Multiple Arguments, Multiple Values

All λ_{TR} functions consume one argument and produce one value. In contrast, Racket functions are multi-ary and may produce multiple values [Flatt and PLT 2010]. For example, the function

```
(define two-val?
  (λ ([x : ⊤] [y : ⊤]) (values (number? x) (string? y))))
```

determines *both* whether its first argument is a number, *and* whether its second argument is a string.

Expressing this form of reasoning in our type system demands a different representation of function types. On the domain side, no additional work is needed because propositions and objects refer directly to the names of the parameters. On the range side, each function produces a sequence of values, each of which comes with a type, two latent propositions, and a latent object. In our running example, the latent propositions for the first return value are $\mathbf{N}_x|\overline{\mathbf{N}}_x$.

Although test positions cannot cope with multiple values, the following idioms exploits multiple values for tests:

```
(let-values ([(x y) (two-val? e1 e2)]) (if x — —))
```

Our new function type representation allows the type system to prove e_1 is a number in the then branch of the **if** expression, and to use y in a later test expression for checking if e_2 is a string.

7.3 User Presentation

While the Typed Racket types capture much useful information about the program, they can also be hard for programmers to read and write. Fortunately, in most cases a simplified type presentation suffices, and complex types are reserved for special occasions.

First, few type errors involve types with non-trivial propositions directly. In our experience with Typed Scheme, almost all type errors are directly explicable with the underlying, non-occurrence typing portion of the system, meaning that users' primary experience with occurrence typing is that it just works.

Second, when users need to specify or read types with propositions or objects, these are primarily latent and symmetric between the two propositions. For example, to specify the type of *strnum?* in Typed Racket syntax, the user writes

```
(: strnum? (⊤ → Boolean : (⋃ String Number)))
```

The syntax states that the latent then proposition is $(\bigcup \mathbf{S}\ \mathbf{N})_x$, where x is the name of the argument, and the latent else proposition is conversely $\overline{(\bigcup \mathbf{S}\ \mathbf{N})}_x$, with a latent object of $\emptyset$. In practice, this syntax suffices for capturing the substantial majority of the types with latent propositions.

Third, Typed Racket uses local type inference and bi-directional typechecking [Pierce and Turner 2000]. Since all top-level definitions are annotated in the above fashion, the type system can propagate the latent propositions into non-latent propositions for the bodies of functions such as *strnum?*. In short, programmers almost never need to write down non-latent propositions.

7.4 Implementing the Logical System

The type system presented in section 4 is non-algorithmic. For an implementation, we must both eliminate the subsumption rule and implement the T-VAR rule. The former is accomplished via a distribution of the subtyping obligations among the rules. The latter demands the implementation of a notion of provability.

Since propositional satisfiability is decidable, the logical system is straightforward to implement in principle. We employ three strategies to avoid paying the full cost of deciding the logic in almost all cases. First, we split the type environment from the proposition environment. This separation avoids invoking the logic to typecheck each variable reference. Second, Typed Racket eagerly simplifies logical formulas, significantly decreasing their typical size. Third, it also refines the type environment each time a new formula is added to the proposition environment. These optimizations mean that most code does not pay the cost of the proof system.

These techniques are well-known from work on SAT solvers. Since deciding the logical inference rules of λ_{TR} can be cast as a satisfiability-modulo-theories problem, we plan to investigate applying existing off-the-shelf SMT solvers [Ganzinger et al. 2004].

8. Empirical Measurements

Numerous encounters with difficult-to-type idioms in Racket code triggered the development of our new system. In order to measure its actual effectiveness in comparison with the original system, we inspected the existing Racket code base and measured the frequency of certain idioms in practice.

Since precise measurement of programming idioms is impossible, this section begins with a detailed explanation of our empirical approach and its limitations. The following two subsections report on the measurements for the two most important idioms that motivate the Typed Racket work: those that involve predicates applied to selectors, as in example 10, and those that involve combinations of predicates, as in example 4. In both cases, our results suggest that our new approach to occurrence typing greatly improves our capability to enrich existing Racket code with types.

8.1 Methodology

Measuring the usefulness of Typed Racket for typing existing code presents both opportunities and challenges. The major opportunity is that the existing Racket code base provides 650,000 lines of code on which to test both our hypotheses about existing code and our type system. The challenge is assessing the use of type system features on code that does not typecheck.

Since the purpose of Typed Racket is to convert existing untyped Racket programs, it is vital to confirm its usefulness on existing code. Our primary strategy for assessing the usefulness of our type system has been the porting of existing code, which is the ultimate test of the ability of Typed Racket to follow the reasoning

of Racket programmers. However, Typed Racket does not operate on untyped code; it requires type annotations on all functions and user-defined structures. Therefore, it is not possible to simply apply our new implementation to existing untyped code.

Instead, we have applied less exact techniques to empirically validate the usefulness of our extensions to Typed Racket. Starting from the knowledge of particular type predicates, selectors, and patterns of logical combinations, we searched for occurrences of the relevant idioms in the existing code base. We then randomly sampled these results and analyzed the code fragments in detail; this allows us to discover whether the techniques of occurrence typing are useful for the fragment under consideration.

This approach has two major drawbacks. First, it only allows us to count a known set of predicates, selectors, idioms, and other forms. Whether a function is a selector or a type predicate could only be answered with a semantics-based search, which is currently not available. Second, our approach does not determine if a program would indeed typecheck under Typed Racket, merely that the features we have outlined are indeed necessary. Further limitations may be discovered in the future, requiring additional extensions. However, despite these drawbacks, we believe that empirical study of the features of Typed Racket is useful. It has already alerted us to uses of occurrence typing that we had not predicted.

8.2 Selectors

Our first measurement focuses on uses of three widely used, built-in selectors: *car*, *cdr*, and *syntax-e* (a selector that extracts expression representations from a AST node). A search for compositions of any predicate-like function with any of these selectors yields:

1. 254 compositions of built-in predicates to uses of *car* for which λ_{TR} would assign a non-trivial object;
2. 567 such applications for *cdr*; and
3. 285 such applications for *syntax-e*.

Counting only known predicate names means that (*number?* (*car x*)) is counted but neither (*unknown?* (*car y*)) or (*string?* (*car* (*f*))) are included because (*f*) is not known to have a non-trivial object. In sum, this measurement produces a total of at least 1106 useful instances for just three selectors composed with known predicates.

A manual inspection of 20 uses each per selector suggests that the extensions to occurrence typing presented in this paper are needed for just under half of all cases. Specifically, in the case of *car*, seven of 20 uses require occurrence typing; for *cdr*, nine of 20 benefit; and the same number applies to *syntax-e*. Additionally, in four cases the type correctness of the code would rely on flow-sensitivity based on predicate tests, but using exceptional control flow rather than conditionals.

In conclusion, our manual inspection suggests that some 40% to 45% of the 1106 cases found can benefit from extending occurrence typing to selector expressions, as described in section 5. This measurement together with the numerous user requests for this feature justifies the logical extensions for selector-predicate compositions.

8.3 Logical Combinations

Since our original system cannot cope with disjunctive combination of propositions, typically expressed using **or**, measuring **or** expressions in the code base is a natural way to confirm the usefulness of general propositional reasoning for Typed Racket. The source of Racket contains approximately 4860 uses of the **or** macro; also, **or** expressions are expanded more than 2000 times for the compilation of the minimal Racket library, demonstrating that this pattern occurs widely in the code base.

The survey of all **or** expressions in the code base reveals that **or** is used with 37 different primitive type predicates a total of 474 times, as well as with a wide variety of other functions that may be user-defined type predicates. Each of these uses requires the extension for local binding described in section 5.2, as well as the more general logical reasoning framework of this paper to generate the correct filters.

9. Related Work

Intensional Polymorphism Languages with intensional polymorphism [Crary et al. 1998] also offer introspective operations, e.g., `typecase`, allowing programs to dispatch on type of the data provided to functions. The λ_{TR} calculus provides significantly greater flexibility. In particular, it is able to use predicates applied to selectors, reason about combinations of tests, abstract over type tests, use both the positive *and* negative results of tests, and use logical formulas to enhance the expressiveness of the system. In terms of our examples, the system of Crary et al. could only handle the first.

Generalized Algebraic Data Types Generalized algebraic data types [Peyton Jones et al. 2006] are an extension to algebraic data types in which "pattern matching causes type refinement." This is sometimes presented as a system of type constraints, in addition to the standard type environment, as in the HMG(X) and LHM(X) systems [Simonet and Pottier 2007, Vytiniotis et al. 2010].

Such systems are similar to λ_{TR} in several ways—they typecheck distinct branches of `case` expressions with enriched static environments and support general constraint environments from which new constraints can be derived. The λ_{TR} calculus and constraint-based such as HMG(X) differ in two fundamental ways, however. First, HMG(X), like other GADT systems, relies on pattern matching for type refinement, whereas λ_{TR} combines conditional expressions and selector applications, allowing forms abstractions that patterns prohibit. Second, all of these systems work solely on type variables, whereas λ_{TR} refines arbitrary types.

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 as Agda [Norell 2007], Coq [Bertot and Castéran 2004], or Epigram [McBride and McKinna 2004], the relationships we describe with predicates and objects could be encoded in types, since types can contain arbitrary terms, including terms that reference other variables or the expression itself.

The innovation in λ_{TR} is to consider propositions that *relate* types and variables. This allows us to express the relationships needed to typecheck existing Racket code, while keeping the logic decidable and easy to understand.

Types and Flow Analysis for Untyped Languages Shivers [1991] describes a type recovery analysis—exploited by Wright [1997] and Flanagan [1999] for soft typing systems—that includes refining the type of variables in type tests. This analysis is only for particular predicates, however, and does not support abstraction over predicates or logical reasoning about combinations of predicates.

Similarly, Aiken et al. [1994] describe a type inference system using *conditional types*, which refine the types of variables based on patterns in a `case` expression. Since this system is built on the use of patterns, abstracting over tests or combining them, as in examples 12 or 5 is impossible. Further, the system does not account for preceding patterns when typing a right-hand side and thus cannot perform logical reasoning as in examples 13 and 14.

Types for Untyped Languages There has long been interest in static typechecking of untyped code. Thatte [1990] and Henglein [1994] both present systems integrating static and dynamic types, and Henglein and Rehof [1995] describe a system for automatic translation of untyped Scheme code into ML. These systems did not take into account the information provided by predicate tests, as described by Henglein and Rehof in the quote from section 1.

In the past few years, this work has been picked up and applied to existing untyped languages. In addition to Typed Scheme, proposals have been made for Ruby [Furr et al. 2009], Thorn [Wrigstad et al. 2010], JavaScript [ECMA 2007], and others, and theoretical studies have been conducted by Siek and Taha [2006] and Wadler and Findler [2009]. To our knowledge, none have yet incorporated occurrence typing or other means of handling predicate tests, although the authors of DRuby have stated that occurrence typing is their most significant missing feature [priv. comm.].

Semantic Subtyping Bierman et al. [2010] present Dminor, a system with a rule for conditionals similar to T-IF. Their system supports extremely expressive refinement types, with subtyping determined by an SMT solver. However, while λ_{TR} supports higher-order use of type tests, due to the limitations of the semantics subtyping framework, Dminor is restricted to first order programs.

10. Conclusion

This paper describes a new framework for occurrence typing. The two key ideas are to derive general propositions from expressions and to replace the type environment with a propositions environment. These ideas increase the type system's expressive power via reasoning tools from propositional logic.

Acknowledgements

Discussions with Aaron Turon greatly improved this paper. The development of Typed Racket has been supported by Stevie Strickland, Eli Barzilay, Hari Prashanth K R, Vincent St-Amour, Ryan Culpepper and many others. Jed Davis provided assistance with Coq.

References

A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st Symposium on Principles of Programming Languages*, pages 163–173. ACM Press, 1994.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*, volume XXV of *EATCS Texts in Theoretical Computer Science*. Springer-Verlag, 2004.

G. M. Bierman, A. D. Gordon, C. Hricu, and D. Langworthy. Semantic subtyping with an SMT solver. In *Proc. Fifteenth International Conference on Functional Programming*. ACM Press, 2010.

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

K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proc. Third International Conference on Functional Programming*, pages 301–312. ACM Press, 1998.

H. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

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

M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001. URL `http://www.htdp.org/`.

C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Progr. Lang. Sys.*, 21(2):370–416, 1999.

M. Flatt and PLT. Reference: Racket. Reference Manual PLT-TR2010-reference-v5.0, PLT Scheme Inc., January 2010. `http://plt-scheme.org/techreports/`.

M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for ruby. In *SAC '09: Proc. 2009 ACM Symposium on Applied Computing*, pages 1859–1866. ACM Press, 2009.

H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *16th International Conference on Computer Aided Verification, CAV'04*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer-Verlag, 2004.

F. Henglein. Dynamic typing: Syntax and proof theory. *Sci. Comput. Programming*, 22(3):197–230, 1994.

F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: translating Scheme to ML. In *Proc. Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 192–203. ACM Press, 1995.

W. A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

R. Komondoor, G. Ramalingam, S. Chandra, and J. 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.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. 15th Symposium on Principles of Programming Languages*, pages 47–57. ACM Press, 1988.

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. Eleventh International Conference on Functional Programming*, pages 50–61. ACM Press, 2006.

B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Progr. Lang. Sys.*, 22(1):1–44, 2000.

J. C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.

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

J. G. Siek and W. 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.

V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Progr. Lang. Sys.*, 29(1):1–54, 2007.

S. Thatte. Quasi-static typing. In *Proc. 17th Symposium on Principles of Programming Languages*, pages 367–381. ACM Press, 1990.

S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proc. 35th Symposium on Principles of Programming Languages*, pages 395–406. ACM Press, 2008.

D. Vytiniotis, S. Peyton Jones, and T. Schrijvers. Let should not be generalized. In *TLDI '10: Proc. 5th workshop on Types in language design and implementation*, pages 39–50. ACM Press, 2010.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *ESOP '09: Proc. Eighteenth European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2009.

A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Progr. Lang. Sys.*, 19(1):87–152, 1997.

T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proc. 37th Symposium on Principles of Programming Languages*, pages 377–388. ACM Press, 2010.