7-2007

A Framework for Implementing Type Systems

Brian Dorn
Iowa State University

Gary T. Leavens
Iowa State University

A Framework for Implementing Type Systems

Abstract
Type systems are ubiquitous in the study of programming languages. Although the basic mechanisms are well understood, a new type system can still be a challenge to implement. We present the design and implementation of a domain-specific language (i.e., a functional framework) for writing type system implementations. This domain-specific language has been embedded in both Haskell and Scheme. It allows users to write down the axioms and inference rules of the type system in a stylized notation that closely resembles the formal type rules, and automates the rest of the work of type checking or type inference.

Keywords
Type inference, type checking, Scheme language, Typedscm language

Disciplines
Programming Languages and Compilers | Software Engineering

A Framework for Implementing Type Systems

Brian Dorn and Gary T. Leavens

TR #07-12a
July 2007

Keywords: Type inference, type checking, Scheme language, Typedscm language.
2006 CR Categories:
D.2.13 [*Software Engineering*] Reusable Software — reusable libraries; D.3.2 [*Programming Languages*] Language Classifications — applicative (functional) languages, Scheme.
Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

A Framework for Implementing Type Systems

Brian Dorn
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
dorn@cc.gatech.edu

Gary T. Leavens
Department of Computer Science
Iowa State University
Ames, IA, USA
leavens@cs.iastate.edu

Abstract

Type systems are ubiquitous in the study of programming languages. Although the basic mechanisms are well understood, a new type system can still be a challenge to implement. We present the design and implementation of a domain-specific language (i.e., a functional framework) for writing type system implementations. This domain-specific language has been embedded in both Haskell and Scheme. It allows users to write down the axioms and inference rules of the type system in a stylized notation that closely resembles the formal type rules, and automates the rest of the work of type checking or type inference.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—applicative (functional) languages, Scheme; D.2.13 [*Software Engineering*]: Reusable Software—reusable libraries

General Terms Languages

Keywords type inference, Scheme language, Typedscm language

1. Introduction

A type system for a programming language is typically presented as a set of axioms and inference rules, possibly with various side conditions. We call such a presentation a *formal type system*. Typically the formal type system is syntax-directed in the sense that for each case of the language's abstract syntax, there is at least one axiom or rule whose conclusion matches such an abstract syntax tree. If there is exactly one rule for each kind of conclusion in such formal type system, then the formal type system is said to be *deterministic*, and the formal type system can be regarded as an algorithm for type checking or type inference. Programming language researchers often present algorithms for type checking and type inference in this way.

Ideally, such a deterministic formal type system would be directly executable. This would simplify and automate the process of implementing such a formal type system for people working with programming languages, implementors of tools, and researchers. It would also ensure that the implementation correctly matches the formal type system, since there would be no possibility of making mistakes in the translation. It would also ease maintenance, as changes could be made directly in the formal type system and would be directly reflected in its implementation.

Unfortunately, this ideal seems difficult to realize. Formal type systems are written in a variety of mathematical description notations, such as LaTeX, and there is no universally agreed-upon concrete syntax that we could use as input. The dream of a compiler from mathematics, even in the limited domain of formal type systems, seems beyond what is possible with present technology. Furthermore, we are not interested in the kind of standardization effort that would be needed to get all workers in the area of programming languages to agree on a single concrete syntax for presenting formal type systems.

The next best thing would be a domain-specific language, which would allow writing the type system's implementation in a notation that closely matches the formal type system. The advantages of such a domain-specific language would be nearly as great as the advantages of the ideal described above. That is, such a domain-specific language would simplify and automate the process of implementing a formal type system. It would also make it easy to check that the implementation matches the formal type system, minimizing the possibility of implementation mistakes. It would also ease maintenance, as changes to the formal type system could be easily reflected in its implementation.

To illustrate this point, consider the following two cases from the formal type system of the simply typed lambda calculus with constants. Our goal is to develop a notation that can be evaluated by some host language (Scheme in this case), but at the same time is a clear isomorph for the formal specification. It is often the case that type system designers have a need for specifying axioms. For example, the type of a constant in the lambda calculus is traditionally denoted with the axiom:

$$\Gamma \vdash c : B$$

Choosing as direct a translation as possible from the formal notation, we might end up with something that looks like:

```
(tc:axiom (:- gamma (: c B)))
```

Note that the alterations are only surface level ones, made to fit the syntax of the Scheme host language: transposition from infix operators to prefix ones and the use of `:-` as a stand-in for $\vdash$. We also use names like `tc:axiom`, instead of just `axiom`, to avoid the possibility of clashes in Scheme.

More often, type system designers specify inference rules with both hypotheses and conclusions. The function application rule from the lambda calculus is typical of such rules:

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : \tau \rightarrow \tau' \\ \Gamma \vdash e_1 : \tau\end{array}}{\Gamma \vdash e_0\ e_1 : \tau'}$$

Following a similar approach, we could imagine a resultant rule of the form:

```
(tc:rule (list
   (:- gamma (: e0 (function-type-expr tau tau*)))
   (:- gamma (: e1 tau)))
   ;;-------------------------------------------
   (:- gamma (: expr tau*)))))
```

Here we have a rule specified as a list of hypotheses and a conclusion, also in the prefix notation. The `function-type-expr` procedure represents the type constructor →, and a comment mimics the horizontal line used in inference rules.

Implementing the type system for the lambda calculus in this way would result in a clear—and obvious—relationship between the formal type rules and their implementation. However, evaluating an expression using type rules written in this fashion would require a framework capable of interpreting the various forms contained therein (e.g., `tc:rule`, `:-`, `tc:axiom`). To be powerful enough to be used in useful contexts, the framework would also require additional rule specification forms (e.g., rules with side conditions). This is the goal of the system we describe in this paper.

To meet this goal, we have designed and implemented a domain-specific language for implementing type systems. The language was embedded both in Scheme and in Haskell; we present details of the Scheme version here. An embedded implementation is advantageous for such a system, given that one often wishes to do other aspects of language prototyping in the same language. In our case, we originally developed the technique for Haskell as we were using Haskell in teaching Schmidt's *The Structure of Typed Programming Languages* (1994), and so an embedding in Haskell was helpful. This was revised, corrected, and greatly improved by the first author in his M.S. thesis (Dorn, 2005) for use with teaching an undergraduate language principles course using Friedman, Wand, and Haynes's book *Essentials of Programming Languages* (2001).[1] For that use, we wanted the program to be understandable by students working in Scheme, and hence a Scheme embedding was appropriate. We conjecture that such an embedding would be helpful for other language prototyping uses as well.

Most previous work on the implementation of type systems has either been very generic, such as techniques for implementing attribute grammars, or has simply offered material for teaching how to write such type systems. (Examples of teaching material are presented by Cardelli (1987) and Friedman et al. (2001).) However, the domain-specific language approach offers more advantages to the type system designer, as our experience testifies.

2. System Overview

Our experience comes from the evolution and maintenance of a type system for Scheme used in a course, "Principles of Programming Languages" taught at Iowa State University. The type checker used in this course, which we refer to as Typedscm, is built upon our framework. This framework provides a notation for typing rules, and is capable of interpreting these rules to do the type checking of student programs. This abstract framework was customized for the Scheme dialect used in the book *Essentials of Programming Languages* (Friedman et al., 2001), which was itself extended with additional syntax for the type system (Leavens, Clifton, & Dorn, 2006).

The framework implements a standard process for type checking or type inference, shown in Figure 1. This mirrors a traditional compiler/interpreter architecture of parsing, type checking, and code generation/execution.

The "Typing Engine" component depicted in Figure 1 is of primary interest in the discussion here. The engine analyzes the *abstract syntax tree* (AST) generated by the parser for typing anomalies based on the specified typing rules. As it traverses the tree, it builds an *annotated AST* that associates type annotations with each abstract syntax component from the original AST nodes. The resultant annotated AST is then examined for errors, and a decision is made as to whether the program should be evaluated by the underlying Scheme interpreter.

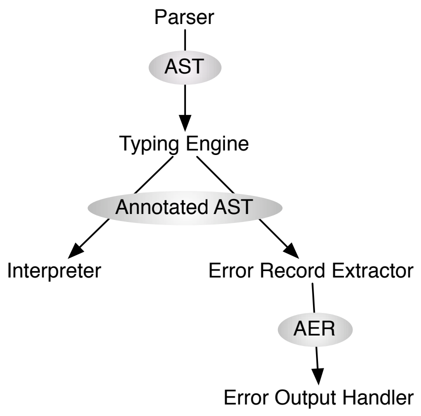

Figure 1. Process Overview

It is the implementation strategy used for the typing engine which makes our approach unique. Three sub-components make up the bulk of the engine: an abstract unification system, extensions which provide details about the specific type system, and a listing of the syntax-directed typing rules and axioms. These subcomponents are implemented by modules that we will refer to as the *type helpers*, the *method dictionary*, and the *type annotation rules*, respectively. Figure 2 illustrates the relationship of these modules abstractly.

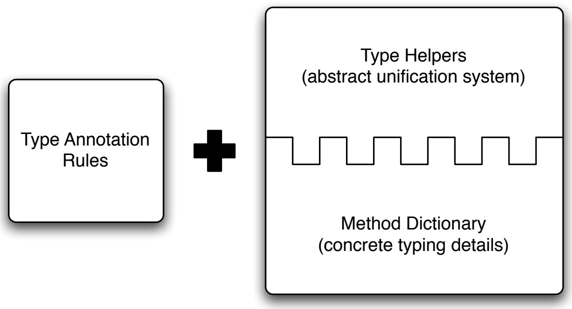

Figure 2. Typing Engine

The type helpers designate an interface through which a type system can be specified; that is, they provide the necessary framework for evaluating the type rules which make up the type system. The various Scheme procedures comprising the type helpers provide such functionality as the ability to declare axioms, to specify type rules, and to denote judgments within rules about subexpressions in a given piece of syntax. They also implement a modified Hindley-Milner unification algorithm (Milner, 1978; Cardelli, 1987) and are responsible for maintaining all information about the unification process during type checking. This core component of the typing engine is the reusable framework that can be applied to many other type system implementations.

[1] The same corrections also work in the Haskell version.

The type helpers are internally oblivious to the specific representation of syntax, types, and errors in a given type system implementation; they make use of abstract procedures that must be provided through a method dictionary in order to manipulate the specific representations. The *method dictionary* is a record which encapsulates several required procedures defined outside of the scope of the type helpers. Put simply, these procedures provide an interface to the specifics of the type expression grammar used and tailor the type helpers to meet the needs of the client type system. For example, the method dictionary contains procedures for extracting a type expression's subterms and for testing whether a given type represents the super-most type in a type hierarchy.

When the type helpers are instantiated with a valid method dictionary, they generate the concrete procedures used, in essence, to evaluate type annotation rules. To complete a type system implementation, a designer then uses these concrete procedures to specify axioms and rules for each case of the implemented language's abstract syntax. For a non-trivial language, building these annotation rules makes up the bulk of the designer's implementation effort. Fortunately, as we will see, these rules are built using a notation that can be directly mapped to the underlying formal type system.

The remainder of this paper elaborates on these components from an external perspective. We treat the type helper framework as a black box and illustrate the aspects of its use which a third-party designer would need in order to implement his or her own type system.[2] The required components of the method dictionary are outlined in Section 3. The type helpers that constitute our domain-specific language are detailed in Section 4. An example implementation of a type system for the lambda calculus is presented in Section 5. A brief overview of how type errors and their associated messages can be generated is given in Section 6. The final sections conclude with a commentary on related work and a discussion of our experience using this approach to implement a relatively complex type system.

3. The Method Dictionary

Type system designers wishing to use our framework need to specify a number of procedures that allow the abstract system to work with the concrete, user-defined type expressions that are declared outside of the framework. This collection of functionality is the method dictionary. The dictionary is a record containing 21 procedures, which are later extracted by the type helpers and used within the typing engine.

This section enumerates the elements of the dictionary and explains the purpose of each in turn. Throughout, we use sample invocations to describe the behavior of these procedures. Because these invocations are not tied to any particular form of type expression, we use the word *term* to refer abstractly to a value that is a type expression. One example of a term would include the result of `(function-type-expr tau tau*)`, shown above. The exact definition of such terms is specified concretely in an external module by defining a variant record where each production in the type expression grammar is represented with a corresponding variant.

3.1 Type Variables and Expressions

As with any implementation of a type system, our framework requires some basic functionality for dealing with terms and type variables. Typedscm provides the type system designer with a representation of *logical variables*, variables that may be bound to specific terms, other logical variables, or nothing within the unification environment. It also defines the notion of a *substitution* which serves as a function from logical variables to terms. The type helpers then are able to process typing rules by using logical variables as placeholders for types not yet known, binding them to their discovered values with substitutions in the unification environment, and then resolving the variables at a later point by applying the various substitutions.

The following seven method dictionary elements are those required for basic manipulation of variables and simple terms by the type helpers:

`(to-term` lv`)` converts the logical variable, lv, into a term that represents lv. This requires that the type system designer provides a representation of type variables in his or her type expression grammar. Typically, this procedure would just wrap lv up as data in the new term.

`(get-var` t`)` extracts a logical variable from a term, t. Its value is a *maybe*[3] that is `(make-something` lv`)` when t represents the logical variable lv, and `(make-nothing)` when it does not. This procedure provides the opposite functionality of `to-term`; it converts from terms to logical variables.

`nullSubst` is the empty substitution. When applied using the framework procedure `tc:subst-apply`, it behaves as an identity function in that, for all logical variables lv:

```
(tc:subst-apply nullSubst lv)
= (to-term lv).
```

Often the `nullSubst` is used as a seed value for the unification environment when no types are initially known.

`(bind` lv t`)` produces a substitution that maps a particular logical variable, lv, to a specific term, t. In all other cases, this substitution behaves like `nullSubst`. In other words, when the resulting substitution is applied to lv, the original term t is returned.

`(app` s`)` transforms the substitution s into a function that maps terms to terms. This is particularly useful within the type helpers as it allows for direct manipulation of terms, rather than logical variables.

`(subterms` t`)` returns a list of all subterms contained in t. Revisiting our previous example from the lambda calculus,

```
(subterms (function-type-expr tau tau*))
```

results in the list `(tau tau*)`, where `tau` and `tau*` are themselves terms.

`(same-kind` t s`)` is true if and only if t and s are the same "kind" of term, which is to say that they have the same operator. For example, if t and s were both `function-type-expr` terms, then `(same-kind` t s`)` would return true.

3.2 Subtyping

The current type helpers allow for a primitive notion of subtyping. Our restricted implementation allows for the types *top* (the supertype of all types) and *bottom* (the subtype of all types), with all other types having equal levels in the hierarchy between top and bottom. This choice was made in order to enable the implementation of the type systems described in (Jenkins & Leavens, 1996) and (Leavens et al., 2006), which we had used in previous offerings of the aforementioned undergraduate course. However, a more advanced subtype unification algorithm could be implemented in the

[2] Those interested in the internal functioning of the type helpers are referred to (Dorn, 2005) for full details.

[3] As in Haskell, an instance of a maybe type represents a value that is either an actual value or nothing at all.

type helpers to enable more advanced schema. Additionally, such a change could be made without impacting the makeup of the method dictionary procedures related to subtyping. Note that it is also possible to disable subtyping by providing stubs for these procedures. These eight required procedures are outlined below:

`(is-bottom? `*t*`)` returns true only when the term t is the bottom type within the type hierarchy.

`(is-top? `*t*`)` tests whether t is the top type.

`(contravar-subterms `*t*`)` is a list of all subterms in t that are to be considered contravariant in the unification of subtypes. Contravariant subterms are treated in the inverse direction from their containing types with respect to subtyping. For example, if `A <: B`, then `(contravar-subterms `*t*`)` returns a list of terms that should have types that are supertypes of `A` (such as terms that have type `B`).

`(covar-subterms `*t*`)` extracts the covariant subterms of t. Covariant subterms are checked in the same direction as their containing types for subtyping purposes.

`(invar-subterms `*t*`)` provides the subterms that are invariant in t. Invariant subterms are those terms which are not to be handled with the subtype unification algorithm. They are instead checked with the same algorithm used for terms in `(subterms `*t*`)`.

`(subtype-replace `*sub super*`)` returns a pair, like `(`*sub* `.` *super*`)`, except that any special replacements due to subtyping have been made. This is useful for the treatment of advanced type constructs that need to be de-sugared to more simple types during the tests for subtyping. For example, in the case of variable arity function types it is necessary to rewrite the term as a function type of a fixed length with the appropriate number of parameters.

`(find-intersection-subtyping `*i s*`)` attempts to subtype unify its arguments and returns the result substitution of the first type in i that unifies with s.

`(find-intersection-supertyping `*s i*`)` tries to find a substitution that makes s a subtype of each type contained in i. If no such substitution exists, it results in the empty list.

3.3 Special Types

The final six procedures needed allow the typing engine to test and manipulate three special terms: error types, intersection types, and declaration types. In addition to polymorphic variable types, type system designers are also required to treat these three types in their implementations. However, as with subtyping it is possible to provide stubs for these procedures in the event that the designer's type system has no need for them.

`(is-error? `*t*`)` is true only when t is an error type.

`(is-intersection-type? `*t*`)` tests whether t is an intersection of types. That is, t is thought to have more than one type.

`(is-dec-type-expr? `*t*`)` is true when t is a type expression that represents a declaration type in the system. Such types are useful for specifying rules for syntax elements that result in new variable bindings (e.g., formal parameter lists, definitions, etc).

`(dec-type-expr->env `*d*`)` extracts the environment of bindings contained within the declaration type d.

`(gen-error-mismatch `*s e i*`)` returns an error term that may contain an error record for the AST element s, generated when there is a mismatch between the expected type e and the inferred type i.

`(gen-error-rule `*s ss*`)` creates an error type expression, which may contain a custom error record, for when the rule corresponding to the AST element s fails, when ss is the list of its immediate subtrees. This is used for generating more specific error messages than can be achieved with the basic mismatch generator.

In actual use, many of these procedures are commonly implemented while specifying the other external aspects of a type system, like type expressions. Thus, the method dictionary is fairly simple to produce. We present example method dictionary entries for a type checker for the simply typed lambda calculus later in Section 5.

4. The Type Helpers

This section describes the type helpers, which users can treat as a domain specific language for writing executable versions of formal type systems. We discuss both the kinds of data used by the type helpers and then the helpers themselves.

4.1 Data Structures for Type Systems

Several kinds of data are used to interface with the type helpers.

Logical variables, of type `tc:logicalvar`, are the most basic data. A *logical variable* is a simply a name that is unique in some environment. Logical variables are the domain of the unification system's environment. Binding logical variables to values or other logical variables is how the unification process gathers constraints (Milner, 1978; Cardelli, 1987).

Attributed syntax pairs are pairs that contain an AST and an attribute, which is typically a type expression. The procedure named "`:`" is used as a constructor for such pairs. Thus attributed syntax pairs associate the given type information with a particular AST.

A *judgment* is used to specify a typing constraint to be checked by the system. The code in Figure 3 defines a variant record for judgments using the **`define-datatype`** syntax from Friedman, Wand, and Haynes' (2001) textbook. A **`define-datatype`** declaration gives the name of the variant record (in this example, `tc:judgment`), the name of a type predicate for the variant record type (`tc:judgment?`), and several record declarations. Each record declaration consists of a record name or *tag* (in Figure 3, the names are `:-d`, `:?-`, `<:`, and `<:>`), and a list of field-type pairs. Each field-type pair consists of the name of a field (such as `lvar` on line 4) and the name of a type predicate that tests for the type of value stored in that field (such as `tc:logicalvar?` on line 4). Each record represents a different kind of typing judgment, which can be used with the various type helpers.

```
(define-datatype tc:judgment tc:judgment?
  (:-d (env tc:environment?)
       (attr-pair (tc:attrib-pair-of datum? datum?)))
  (:?- (lvar tc:logicalvar?)
       (attr-pair (tc:attrib-pair-of datum? datum?)))
  (<:  (subtype datum?) (supertype datum?))
  (<:> (t1 datum?) (t2 datum?)
       (subtype-var tc:logicalvar?)
       (supertype-var tc:logicalvar?)))
```

Figure 3. The judgment datatype.

Each of the record names in the judgment datatype is also the name of a procedure that constructs records of that type. Thus, for example, `(<: t1 t2)` represents a judgment that the type represented by `t1` is a subtype of that represented by `t2`.

Typedscm also provides a convenience form, `(<:> t1 t2 sub super)`, which asserts that one of two types, `t1` and `t2`, must subtype the other. In subsequent judgments, the logical variables `sub` and `super` are available as aliases for the subtype and supertype as they are bound appropriately during the processing of the `<:>` form. While not often necessary, this syntactic sugar significantly simplifies creating rules for expressions that have an arbitrary number of subexpressions that must maintain a subtype relationship. For example, in Typedscm, the inference rule for the Scheme `cond` expression dynamically generates $n - 1$ of these judgments to ensure that the n clauses have compatible types within the hierarchy and that the result type is the super-most type of all clause types.

The most common form of judgment, named `:-`, is derived from `:-d`. The judgment `(:- gamma (: s t))` says that in the type environment `gamma`, the AST `s` has type `t`. In our Scheme implementation, the `:-` form is a macro, defined as follows.

```
(define-syntax :-
  (syntax-rules (:)
    ((:- gamma (: s t)) (:-d gamma (: s (delay t))))))
```

Thus `:-` delays evaluation of the type expression `t`, which often depends on the computation in a rule's hypotheses. The typing engine internally forces the evaluation of this computation when it is safe to do. In practice users of our framework only use the `:-` form and never use `:-d` directly.

The remaining kind of judgment form, `:?-`, allows for the environment field to be a logical variable, which can be defined elsewhere in a rule. This is useful writing rules for ASTs that contain internal declarations, such as **`let`**.

Since type expressions include logical variables, they may be used freely in judgments wherever a type expression is desired. For example, in a judgment of the form `(<: t1 t2)`, both `t1` and `t2` may be terms that represent logical variables.

4.2 tc:axiom

The first of the type helpers is `tc:axiom`. This helper allows one to write axioms—rules with no hypotheses. Thus the judgment provided in an axiom is always treated as true. This can be used to both assert that some AST has a type determined outside of the type system's formal rules, or to assert an error type for an AST. Both of these uses are shown in Figure 4.

```
   (define tc:scheme-exp-annotate
     (lambda (gamma e)
       (cases tc:scheme-exp e
         (tc:varref (variable)
5          ;; ...
           (if (tc:env-bound? variable
                             (tc:global-var-types))
               (let ((global-type
                       (tc:type-expr-instance
10                       (tc:env-value
                           variable
                           (tc:global-var-types)))))
                 (tc:axiom (:- gamma (: e global-type))))
               (tc:axiom
15               (:- gamma
                   (: e (tc:varref-error-maker e))))))
         ;; ... other expression cases
       )))
```

Figure 4. A `tc:axiom` example from the implementation of Typedscm, written in Typedscm

Figure 4 gives the declaration of a procedure for annotating Scheme code. The procedure takes a type environment, `gamma`, and an expression AST, `e`. It uses a **`cases`** expression (Friedman et al., 2001) to take action depending on the tag of the AST. (The **`cases`** expression takes the name of a type and an expression e, and then has several clauses that match the tag of e's value against the tag named in each of the clauses. It executes the body of the first matching clause, after positionally binding the fields of the record to the clause's formals.) The clause shown is for a variable reference expression. In Typedscm, there is a fairly complex way to determine the type of such a variable, one case of which is shown in the figure, that of global variables (lines 6–16). In this case, each branch of the if-expression that starts on line 6 ends in a use of `tc:axiom`.

The other examples in this section are all taken from the same procedure, as some of the additional cases alluded to by the ellipsis on line 17 of Figure 4.

4.3 tc:rule

The most commonly used rule helper is `tc:rule`, which represents the notion of a type inference rule with a list of hypotheses and a conclusion. The hypotheses and conclusion are all judgments. An example appears in lines 4–8 of Figure 5. This example says that if the subexpressions `e1` and `e2` both have type number (denoted by the constant `*tc:number*`), then the equals expression itself has type boolean.

```
   (tc:equals-exp
    (e1 e2)
    (tc:rule-or
     (tc:rule (list
5      (:- gamma (: e1 *tc:number*))
       (:- gamma (: e2 *tc:number*)))
       ;;------------------------
       (:- gamma (: e *tc:boolean*)))
     (tc:rule (list
10     (:- gamma (: e1 *tc:boolean*))
       (:- gamma (: e2 *tc:boolean*)))
       ;;--------------------------
       (:- gamma (:e *tc:boolean*)))))
```

Figure 5. Two `tc:rule` examples within a `tc:rule-or` example

An instance of `tc:rule` fails if one of the hypotheses fails due to a type error.

4.4 tc:rule-or

A convenience that allows a more fine-grained splitting of rules into cases is `tc:rule-or`. It combines two or more rules and checks that at least one of them holds. It is implemented as a macro that delays the rules passed to it, so that it can process its rule arguments in order, using the first one that does not fail.

An example rule for a hypothetical equals expression is shown in Figure 5. This rule asserts that a `tc:equals-exp` expression type checks with type boolean if its two subforms, `e1` and `e2`, either both have type number or if they both have type boolean.

4.4.1 tc:rule-if

Rules with simple side conditions can be specified using the `tc:rule-if` form. It requires a list of hypotheses and a conclusion (like `tc:rule`), but it also takes two procedures. The first is a function from a list of type attributes to a boolean. If the list of the hypotheses checked without failure, then this procedure is passed the list of the type attributes computed for the hypotheses. If the procedure returns false when passed this list, then the rule fails. Otherwise, the list of type attributes is passed to the second procedure, which returns a substitution defining any number of new variables in the unification environment. (When no definitions need to be made, the second procedure can return the null substitution.)

Figure 6 depicts a simple use of this helper that checks a **`lambda`** expression with multiple formal parameters. The single hypothesis in the list checks that the `body` has a type, which is bound to a logical variable (contained within `rt`). The side condition on lines 9–10 enforces the restriction that the formal parameters must be unique. The side definition procedure on lines 11–12 makes no new definitions.

```
   (tc:lambda-exp
    (formals body)
    (let ((rt (tc:new-variable-type-expr))
          (formal-ts (tc:listof-new-variable-type-expr
5                     (length formals))))
      (tc:rule-if (list
        (:- (tc:extend-mult-env gamma formals formal-ts)
            (: body rt)))
        (lambda (ts)                  ;; Side Condition
10        (tc:no-duplicates? formals))
        (lambda (ts)                  ;; Side Definition
          (tc:type-expr-null-subst))
        ;;-----------------------------------------
        (:- gamma
15          (: e (tc:function-type-expr formal-ts rt))))))
```

Figure 6. A `tc:rule-if` example

4.5 tc:rule-seq

The type helpers described above are adequate for encoding most formal typing rules. However, an additional type helper form is needed for a few cases, such as sequential declarations, in which the programmer needs more explicit control over the order of checking hypotheses, side conditions, and making side definitions. We provide a helper `tc:rule-seq` that allows this control at the price of a slightly more verbose, but still declarative, specification of the what happens when during processing of the rule. We omit additional discussion of this helper here given its infrequent use, though those interested can consult Dorn's thesis (2005) for full details.

5. An Example Implementation

Having examined the framework's components in some detail, it is now possible to use them to implement a complete type system. In this section we present a basic implementation of a type system for the lambda calculus using Hindley-Milner (Milner, 1978) inference rules. We delineate a typical process a type system designer would undertake when using our framework, and illustrate the close relationship between various formalisms and their corresponding Scheme encodings.

In what follows we first build the Scheme representations for syntactic elements. Next we devise an abstraction for types. We then develop the annotation procedure containing the inference rules and build the method dictionary.

5.1 Language Syntax

The first step is to formally define the language's abstract syntax trees (ASTs). Figure 7 provides a Scheme variant record definition on the left, and a corresponding grammar for lambda calculus terms on the right. The `tc:lambda-calc` datatype has four variants, one corresponding to each production in the formal grammar. The `tc:le-self-evaluating` variant represents abstract syntax trees self-evaluating terms (e.g., constants, literals); this record has one field, named datum, which can hold any type (since the type predicate `datum?` always returns true). In the `tc:le-varref` ASTs, Scheme symbols are used to represent identifiers. The other two productions, representing function applications and lambda expressions, have fields that hold subexpressions that are recursively `tc:lambda-calc` variants as indicated by the type predicate `tc:lambda-calc?`. Thus there is a one-to-one mapping between ASTs and the formal syntax.

5.2 Type Expressions

Now that we have a definition for the abstract syntax, the next step is to specify a representation for the types associated with values in the language. Figure 8 depicts a type expression grammar and its associated Scheme datatype, `tc:type-expr`. The function type (i.e., $\rightarrow$) is represented by `tc:function-type-expr`, which uses two type expression parameters for the argument and result types. In addition, we want to support basic types for self-evaluating constants. That is, a self-evaluating term B can take on a value from a set of allowable basic types like $\{symbol, number\}$. We use the tag `tc:basic-type-expr` for this type and its internal `symbol` is used to store a value from the set of allowable literal types. With these types the Scheme model again mirrors the mathematical definition.

However, at this step we make two additions beyond that directly suggested by our formal grammar. Recall from the discussion in Section 3 that the type helpers require a minimal set of types to function. We must provide, through the method dictionary, support for (or the explicit exclusion of) polymorphic variable, error, intersection, and declaration types. We will exclude intersections and declarations later when we define the method dictionary as they are not needed in this example, but we do need some means to represent both type variables and type errors. Our `tc:variable-type-expr` need only envelope a logical variable, a datatype provided by the framework, so that it is a valid type expression. The `tc:error-type-expr` form allows us to note the detection of a typing error, but for simplicity does not encapsulate any information about the nature of the error.

5.3 Annotation Rules

Leaving aside the creation of the method dictionary for the moment (see Section 5.4), we now describe how one can create the annotation procedure that implements the type system. The Scheme code for this procedure is given in Figure 9, along with the formal lambda calculus rules for comparison.

The procedure `tc:lambda-calc-annotate` is the top level procedure called to annotate a piece of abstract syntax with its type information. It initializes the unification environment to be empty and then calls `tc:lambda-calc-annotate-lower` (described below), which is responsible for doing the bulk of the annotation. In both procedures, `gamma` is the current type environment, and `e` is the expression being annotated.

In `tc:lambda-calc-annotate-lower` each variant in the abstract syntax is handled individually with a corresponding rule or axiom. Self evaluating expressions are annotated using an axiom after looking up the basic type of the underlying `datum` (lines 11–14).

Variable references are typed using the `tc:rule-if` form and a new logical variable `tau` (lines 15–22). This rule has no hypotheses *per se*, but it does require that the identifier be bound in the type environment. We specify this in a side condition that checks `gamma` for a binding of `variable`. The side definition to `tc:rule-if` allows us to make the located value available in the conclusion through `tau`.

Procedure calls are handled using a simple `tc:rule` with the help of two new type variables (lines 23–30). This rule corresponds exactly to its formal counterpart with a hypothesis for each of the operator and operand.

The lambda expression form also uses a `tc:rule` and employs a call to the framework procedure `tc:extend-env` to model the extension of type environment Γ in the first hypothesis (lines 31–38). This completes the simply typed lambda calculus rule implementation.

Clearly, there is a close mapping between the rule implementation in Figure 9 and the formal specification. The domain specific language provided by the type helpers coupled with carefully selected abstract syntax and type expressions allows for a highly readable implementation. Using our framework a type system de-

Scheme Representation **λ Calculus**

```
(define-datatype tc:lambda-calc tc:lambda-calc?
  (tc:le-self-evaluating (datum datum?))
  (tc:le-varref (variable symbol?))
  (tc:le-procedure-call (operator tc:lambda-calc?) (operand tc:lambda-calc?))
  (tc:le-lambda-exp (formal symbol?) (body tc:lambda-calc?)))
```

$$\begin{array}{rl} \langle e \rangle ::= & \\ & se \\ | & x \\ | & \langle e_0 \rangle \ \langle e_1 \rangle \\ | & \lambda x.\ \langle e \rangle \end{array}$$

Figure 7. Lambda Calculus Syntax

Scheme Representation **λ Calculus**

```
(define-datatype tc:type-expr tc:type-expr?
  (tc:basic-type-expr (symbol symbol?))
  (tc:function-type-expr (arg-type tc:type-expr?) (result-type tc:type-expr?))
  ;; For type helpers
  (tc:variable-type-expr (lvar tc:logicalvar?))
  (tc:error-type-expr))
```

$$\begin{array}{rl} \langle \tau \rangle ::= & \\ & B \\ | & \langle \tau_0 \rangle \rightarrow \langle \tau_1 \rangle \end{array}$$

Figure 8. Lambda Calculus Types

Scheme Representation **λ Calculus**

```
(define tc:lambda-calc-annotate
  (lambda (gamma e)
    ;; initialize the unification environment one the top level call
    (tc:set-current-subst-list! (list (tc:null-subst)))
    ;; use helper function to do the annotation
    (tc:lambda-calc-annotate-lower gamma e)))

(define tc:lambda-calc-annotate-lower
  (lambda (gamma e)
    (cases tc:lambda-calc e


      (tc:le-self-evaluating (datum)
        (let ((B (tc:infer-simple-datum-type datum)))
          ;;----------------------------------------
          (tc:axiom (:- gamma (: e B)))))
```

$$\frac{}{\Gamma \vdash se : B}$$

```
      (tc:le-varref (variable)
       (let ((tau (tc:new-logicalvar)))
         (tc:rule-if
          '()   ;; no hypotheses here
          (lambda (ts) (tc:env-bound? variable gamma))
          (lambda (ts) (tc:type-expr-bind tau (tc:env-value variable gamma)))
          ;;-----------------------------------------------------------------
          (:- gamma (: e (tc:variable-type-expr tau))))))
```

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

```
      (tc:le-procedure-call (operator operand)
       (let ((tau  (tc:new-variable-type-expr))
             (tau* (tc:new-variable-type-expr)))
         (tc:rule (list
           (:- gamma (: operator (tc:function-type-expr tau tau*)))
           (:- gamma (: operand tau)))
           ;;---------------------------------------------------
           (:- gamma (: e tau*)))))
```

$$\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'}$$

```
      (tc:le-lambda-exp
       (formal body)
       (let ((tau  (tc:new-variable-type-expr))
             (tau* (tc:new-variable-type-expr)))
         (tc:rule (list
           (:- (tc:extend-env gamma formal tau) (: body tau*)))
           ;;-----------------------------------------------
           (:- gamma (: e (tc:function-type-expr tau tau*))))))

       )))
```

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.\ e) : \tau \rightarrow \tau'}$$

Figure 9. Lambda Calculus Inference Rules

signer can fully specify a prototype with relatively little effort and no need to rebuild the underlying unification infrastructure with each new type system.

5.4 Defining the Method Dictionary

We now return to the remaining task, implementing the method dictionary. Recall that the method dictionary is needed to allow the type helpers to manipulate the representation of types. Figure 10 outlines a representative subset of nine dictionary elements. The procedures not shown do not vary significantly in length or complexity from those depicted (Dorn, 2005). The dictionary itself is implemented as an instance of `tc:unifiable-md-dict` and is comprised of a series of anonymous procedures. For clarity, we comment each procedure's name immediately preceding the corresponding **`lambda`**.

The implementation for many of the procedures is quite straightforward. As shown, `to-term`, `get-var`, and `subterms` perform basic operations on the `tc:type-expr` datatype in order to create variable terms, extract the logical variable from a term, and build a list of subterms, respectively. In most type systems, the implementation of these procedures will look similar, with the possible exception that additional forms would be handled in the procedure for subterm extraction.

Of all the method dictionary procedures, `app` is the most complicated. Recall that its purpose is to create a function that maps terms to terms based on a given substitution. It should be a curried procedure that applies the input substitution from the first argument recursively to the term given in the second argument. The resultant term should be identical except that any indicated substitutions of its subterms have been made. The implementation in Figure 10 illustrates the recursive substitution application in lines 25–33. Because the input substitution may indicate multiple—and possibly overlapping—variable replacements, it is useful to have `app` compute a fixed point prior to returning. Lines 34–38 show a simple iteration until there are no further changes to the term from the substitution.

We have noted previously that the method dictionary requires several procedures that may or may not apply to a designer's type system. For example, this lambda calculus example requires no notion of subtyping; yet, the dictionary procedures for subtyping must have some implementation to prevent runtime errors. In this case, stub procedures are given that do not impact the semantic behavior of the type system. In Figure 10 stubs are given for `contravar-subterms`, `subtype-replace`, and `is-bottom?`. These procedures return the empty list, the input arguments unchanged, and the constant false (respectively). Additionally, stubs would be given for procedures related to intersection and declaration types as they are not used in this type system.

The last two procedures given relate to errors. The first procedure, `is-error?`, determines if the argument term is an instance of `tc:error-type-expr`. The procedure used to generate type errors when mismatches occur between expected and inferred types (`gen-error-mismatch`) is artificially short in this example. Since we chose not to encode any information about the nature of errors, it simply builds an empty error expression. Typical implementations would save some of the arguments for later use.

As we have shown, most aspects of the method dictionary are trivial to implement. Nearly all of the functionality needed is present once a designer specifies the set of type expressions using **`define-datatype`**, and only a small amount of additional coding is required. The method dictionary provides functionality to handle advanced typing constructs, but also accommodates the needs of simpler type systems with convenient procedure stubs.

```
   (define tc:type-expr-as-unifiable
     (lambda ()
       (tc:unifiable-md-dict
        ;; to-term
 5      (lambda (lv) (tc:variable-type-expr lv))
        ;; get-var
        (lambda (t)
          (cases tc:type-expr t
            (tc:variable-type-expr (v)
10             (make-something v))
            (else (make-nothing))))
        ;; subterms
        (lambda (t)
          (cases tc:type-expr t
15          (tc:function-type-expr
               (arg-type result-type)
               (cons arg-type (list result-type)))
            (else '())))
        ;; app
20      (lambda (s)
          (lambda (attrib)
            (letrec
              ((app-once
                 (lambda (t)
25                 (cases tc:type-expr t
                     (tc:variable-type-expr (lvar)
                        (tc:subst-apply s lvar))
                     (tc:function-type-expr
                       (arg-type result-type)
30                     (tc:function-type-expr
                         (app-once arg-type)
                         (app-once result-type)))
                     (else t))))
              (app
35              (lambda (attrib applied)
                  (if (equal? attrib applied)
                    applied
                    (app applied (app-once applied))))))
            (app attrib (app-once attrib)))))
40      ;; contravar-subterms
        (lambda (t) '())
        ;; subtype-replace
        (lambda (sub super) (cons sub super))
        ;; is-bottom?
45      (lambda (t) #f)
        ;; is-error?
        (lambda (t)
          (cases tc:type-expr t
            (tc:error-type-expr () #t)
50          (else #f)))
        ;; gen-error-mismatch
        (lambda (s e i)
          (tc:error-type-expr))
        ;; ... additional method dictionary procedures
55      )))
```

Figure 10. Excerpt from the Lambda Calculus Method Dictionary

6. Error Handling

The example implementation in the previous section glossed over the issue of error generation. However, given that the ability to generate meaningful error messages is one of the most important roles of any type checker, a more thorough discussion is warranted. Our framework permits a great deal of flexibility in the way that errors are handled. Through the method dictionary, type system designers may specify highly advanced error message generation routines or ones that do very little. As we saw in Section 5, it is possible to define one generic error type generator that is used for all errors and nothing more, though in practice, this is not very helpful. The Typedscm implementation provides a good example of detailed error processing.

Our approach to error message creation is strongly related to the type annotation process. Type system designers using our framework can leverage the fact that, given any particular piece of syntax, a list of the annotation results associated with its subexpressions, and knowledge of the underlying inference rule, one can determine what situation caused the error. After deducing the cause, we encapsulate the relevant information in an error record that will be used to generate a message to the end-user at a later point. Error records are implemented as another variant type where each form corresponds to particular message to be displayed for the end-user. They store information for later use by an error output mechanism.

Consider the annotation rule for conditional expressions given in Figure 11. There are three possible error scenarios for such a rule. First, a non-boolean value could be specified for the test subexpression. Second, some general type error could be found in the test, consequent, and/or alternate arms. Third, the consequent and alternate types could be incompatible (neither being a subtype of the other). Any one of these cases will produce an error in the typing engine, and the type system designer's `gen-error-rule` method dictionary procedure will be invoked.

```
(tc:conditional-exp
 (position test consequent alternate)
 (let* ((t1 (tc:new-variable-type-expr))
        (t2 (tc:new-variable-type-expr))
        (supertype-var (tc:new-logicalvar))
        (subtype-var (tc:new-logicalvar))
        (super (tc:variable-type-expr supertype-var)))
   (tc:rule (list
          (:- gamma (: test *tc:boolean*))
          (:- gamma (: consequent t1))
          (:- gamma (: alternate  t2))
          (<:> t2 t1 subtype-var supertype-var))
          ;; ---------------------------------
          (:- gamma (: e super)))))
```

Figure 11. If Inference Rule

An excerpt of the `tc:scheme-exp-error-maker` procedure is shown in Figure 12. This procedure is supplied in the Typedscm method dictionary for the `gen-error-rule` field. It takes two arguments, *s* and *ss*, where the first is the syntax element being checked and the second is a list containing the results obtained from checking the immediate subexpressions. The specific element of the **cases** expression shown corresponds to the inference rule above. Line 5 in the figure determines that *s* is a `tc:conditional-exp`, and binds the fields of *s* to the variables listed on line 6.

The rest of Figure 12 relies on the logic outlined above regarding the possible causes for errors within the rule, treating each possible error cause in a sequential case analysis. The first case (lines 9–16) inspects the type of the test condition for errors and emits a new error type containing a `tc:badtest-error-record` to disambiguate the error's cause. Any remaining errors from the subexpressions will already contain detailed information (inserted at their point of generation) and the prior results can merely be concatenated (lines 17–18). At this point the only remaining source of failure in the rule is a bad subtyping relationship. Thus, a `tc:if-subtype-error-record` can be produced containing information about the types inferred for the consequent and alternate arms (lines 19–23).

It should be noted that our choice to make use of abstract error records implemented as variant records was entirely optional. The framework does not require any particular method for dealing with errors. Our design decision was made based on a desire to support multiple forms of error output like text-based messages and syntax highlighting. Representing error messages more abstractly allowed us to easily present them in numerous forms at a later point. One might also take a more straightforward approach and use `gen-error-rule` to directly build error message strings that could be printed following type annotation.

```
   (define tc:scheme-exp-error-maker
     (lambda (s ss)
       (let ((tree-tops (map tc:root trees)))
         (cases tc:scheme-exp s
 5         (tc:conditional-exp
            (position test consequent alternate)
            (let ((test-type (car ss)))
              (cond
               ((and (tc:error-type-expr? test-type)
10                    (tc:mismatch-error-record?
                       (tc:error-type-expr->error-record
                         test-type)))
                 (tc:error-type-expr
                   (tc:badtest-error-record
15                   (tc:error-type-expr->error-record
                       test-type))))
               ((tc:contains-errors? ss)
                (tc:composite-error-maker ss))
               (else
20              (tc:error-type-expr
                 (tc:if-subtype-error-record
                   consequent (cadr ss)
                   alternate (caddr ss)))))))
           ;; ... rest of error record generators
25       ))))
```

Figure 12. Error Generation for Conditional

In any case, the actual point of error output is outside of the type annotation process. Type system designers can freely interpret the results of their chosen approach. For our implementation using error records, we would note that the top-most resultant type is an error, extract the internal error record from the type expression, and provide this record to a message generator. The output mechanism that interprets the records is trivial since all that need be done is to print the necessary textual information along with the variable data already contained in the error record.

7. Related Work

Component-based frameworks have be used to promote reuse within type system development. For example, Vanilla (Dobson, Nixon, Wade, Terzis, & Fuller, 1999) can be used to modularize type checking (and behavior) of individual language elements. It also provides the necessary framework for interpreting these modules. However, the modules remain written in the host language (Java in Vanilla's case), which can still be prone to implementation errors in the translation from the formal typing rules.

A domain-specific language approach has been used in systems that aim to support user-defined type refinements, like CLARITY (Chin, Markstrum, & Millstein, 2005; Chin, Markstrum, Millstein, & Palsberg, 2006) and in frameworks for the creation of pluggable type systems, like JavaCOP (Andreae, Noble, Markstrum, & Millstein, 2006). These frameworks are designed to enable third-party type system extensions while ours is primarily targeted at users who are prototyping complete type systems. However, we share a common goal in providing the user of the framework a natural way to specify typing constraints. Andreae et al. elaborate on JavaCOP's use of a declarative rule-based language by noting: "While it would be possible to write the rules directly in Java with respect to an AST API, we believe rules in JavaCOP's language will be significantly easier for rule designers and programmers to understand and define correctly" (2006, p. 58).

That said, the language presented to users in JavaCOP differs substantially from ours. Figure 13 illustrates a JavaCOP rule named `checkNonNull` that enforces a user-defined annotation that some variables can not take on null values in an assignment. The rule's argument specifies the AST syntax elements to which it applies and the framework enables access to the syntax element's sub-terms.

The `where` and `require` clauses define the rule's semantics, and an `error` clause is used to provide feedback when the rule fails. While certainly more readable than a Java-only implementation, complex annotations may result in deeply nested clauses and interwoven error messages. Our work allows more separation between error message generation and the rules.

```
rule checkNonNull(Assign a){
  where(requiresNonNull(a.lhs)){
    require(definitelyNotNull(a.rhs)):
      error(a,"Possible null assignment"
            + " to @NonNull"); } }
```

Figure 13. Example JavaCOP Rule (Andreae et al., 2006, p. 57)

The ability to quickly prototype new type systems and easily maintain them in the future was among our primary concerns in developing Typedscm. Levin and Pierce created the TinkerType system with similar goals. TinkerType decomposes families of formal systems into collections of *clauses* (inference rules) and *features* that determine which clauses make up particular systems (Levin & Pierce, 2003). Once a repository of clauses and features is created, their TinkerType Assembler can be used to conveniently generate numerous ML type checkers for instances in the language family. The inference rules in TinkerType components are written in ML so that they can be assembled into an executable type checker by the system. However, the notation is still ML-based and also mixes error message generation with the rule.

8. Discussion, Conclusions, and Future Work

The Typedscm dialect of Scheme (Leavens et al., 2006) is both an example of what can be done with our framework and of its advantages. The system has about 80 rules, using all the different helpers provided by our framework. These rules handle declarations, statements, and expressions, covering all of Scheme (except macros). In comparison to a previous hand-written type checker used in the undergraduate curriculum, we have found (anecdotally) that the Typedscm implementation using our framework is much easier to understand and modify. For example, it was only a matter of 10 minutes to add a dynamic type test expression to the language.

The framework also made it possible to incorporate more sophisticated typing rules. We have already mentioned the use of a simple subtyping lattice. We were also able to handle data abstraction (Jenkins & Leavens, 1996), and variant record declarations with their associated **`cases`** expressions (Friedman et al., 2001). The framework also was sufficient to incorporate bi-directional type checking (Pierce & Turner, 1998). Thus Typedscm uses top-down type checking instead of bottom-up type inference for checking procedures, when those procedures have a declared type. This results in better error messages for students.

In summary, our framework provides a readable and easily maintainable way to write type checking and inference rules for real languages, with helpful error messages. Our domain-specific language approach results in type system implementations that have a clear relationship to the usual mathematical presentation of such rules. Furthermore, our framework allows a clean separation between the type system's rules and error message output.

One avenue for future work is to do a comparative study of our framework and systems like JavaCOP. We feel our choice of syntax has advantages over other domain-specific languages for typing constructs, but whether the differences matter to type system designers in practice remains an open question. Evaluating which rule styles are preferred by those experienced with formal type systems could be insightful.

There are also several possibilities for merging the approaches of Typedscm and TinkerType (Levin & Pierce, 2003). Internally, TinkerType treats the body of the clause as an arbitrary string value. Thus, we could easily create an ML embedding of our type helper framework that would enable users to write clauses in a more natural syntax. Further, it may be possible to employ their notion of features to build an assembler for Typedscm that not only emits AST annotation procedures but also appropriate method dictionaries for various classes of languages. This would be advantageous as resulting type checkers would have the implementation quickness of TinkerType and the code readability of Typedscm.

Our implementation of Typedscm, which includes the Scheme realization of our type system framework, is available from `http://www.cs.iastate.edu/~leavens/typedscm/`.

Acknowledgments

Thanks to Steven Jenkins and Curtis Clifton for their work on early versions of Typedscm. We also thank Hridesh Rajan for comments on an earlier draft. The work of Leavens was supported in part by NSF grants CCF-0428078 and CCF-0429567.

References

Andreae, C., Noble, J., Markstrum, S., & Millstein, T. (2006). A framework for implementing pluggable type systems. In *OOPSLA 2006: Proceedings of the 21st International Conference on Object-Oriented Programming Systems, Languages, and Applications* (p. 57-74).

Cardelli, L. (1987). Basic polymorphic typechecking. *Science of Computer Programming*, *8*(2), 147-172.

Chin, B., Markstrum, S., & Millstein, T. (2005). Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (p. 85-95).

Chin, B., Markstrum, S., Millstein, T., & Palsberg, J. (2006). Inference of user-defined type qualifiers and qualifier rules. In *European Symposium on Programming (ESOP 2006).*

Dobson, S. A., Nixon, P., Wade, V. P., Terzis, S., & Fuller, J. (1999). Vanilla: An open language framework. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering* (p. 91-104).

Dorn, B. J. (2005). *Design and implementation of a reusable type inference engine and its application to Scheme* (Tech. Rep. No. 05-16). 226 Atanasoff Hall, Ames, Iowa 50011: Department of Computer Science, Iowa State University.

Friedman, D. P., Wand, M., & Haynes, C. T. (2001). *Essentials of programming languages* (Second ed.). Cambridge, MA: MIT Press.

Jenkins, S., & Leavens, G. T. (1996). Polymorphic type-checking in Scheme. *Computer Lanugages*, *22*(4), 215-223.

Leavens, G. T., Clifton, C., & Dorn, B. (2006). *A type notation for Scheme* (Tech. Rep. No. 05-18a). Ames, Iowa, 50011: Department of Computer Science, Iowa State University.

Levin, M. Y., & Pierce, B. C. (2003). TinkerType: A language for playing with formal systems. *Journal of Functional Programming*, *13*(2).

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, *17*(3), 348-375.

Pierce, B. C., & Turner, D. N. (1998). Local type inference. In *POPL 98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (p. 252-265).

Schmidt, D. A. (1994). *The structure of typed programming languages*. Cambridge, MA: MIT Press.