A Practical Subtyping System For Erlang

Simon Marlow
simonm@dcs.gla.ac.uk
University of Glasgow

Philip Wadler
wadler@research.bell-labs.com
Bell Labs, Lucent Technologies

Abstract

We present a type system for the programming language Erlang. The type system supports subtyping and declaration-free recursive types, using subtyping constraints. Our system is similar to one explored by Aiken and Wimmers, though it sacrifices expressive power in favour of simplicity. We cover our techniques for type inference, type simplification, and checking when an inferred type conforms to a user-supplied type signature, and report on early experience with our prototype.

1 Introduction

We can stop waiting for functional languages to be used in practice—that day is here! Erlang is a strict, untyped functional language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly code reloading, and multiple platforms [AVW93]. Applications exist that consist of upwards of half a million lines of code.

This paper documents our experience in designing and building a type system for Erlang. Our type system provides type inference with subtyping, declaration-free recursive types, type signature checking, and data abstraction. So far we have successfully applied our prototype to about 5000 of the 13000 lines of code in the Erlang standard library, and anticipate no difficulties in applying it to the remainder.

We expect that adding a type system to Erlang will improve documentation, maintenance, and reliability. Our type system had two goals. First, it should type existing Erlang code with little or no modification. Second, it should be easy to comprehend. Whereas many type systems strive to maximise expressive power, our aim is to maximise simplicity, consistent with having sufficient power to describe Erlang as it is typically used in practice.

Our first goal rules out the popular type system devised by Hindley and Milner [Hin79, Mil78, DM82]. The difficulty is that with Hindley-Milner each type must involve a set of constructors distinct from those used in any other types, a convention not adhered to by Erlang programmers.

So we need a type system that allows one constructor to belong to several different types. One possibility is types based on row variables, as introduced by Wand [Wan87], and used as the basis of the soft type system for Scheme by Cartwright, Fagan, and Wright [CF91, WC94]. It turns out that the row variable system rejects some programs that seem quite natural to us, and the circumlocutions we had to go through to construct an equivalent program that was well typed struck us as hard to explain. This isn't a problem for soft typing systems, where the goal is to improve performance by removing run-time type checking, and therefore maximum information is of greater benefit than a natural notion of typing.

The alternative that we adopted is to build a type system based on subtyping. Type systems with subtyping have been studied by several researchers [Mit91, FM88, MR85, Rey85], and are based on solving systems of typing constraints of the form $U \subseteq V$, where U and V are types. Hindley-Milner systems, by contrast, are based around equality constraints of the form $U = V$, which can be solved by unification. Subtyping systems are strictly more general than Hindley-Milner systems: each program that can be typed by Hindley-Milner has a typing in a subtyping system, but not vice versa.

Our type system is based around the system developed by Aiken and Wimmers [AW93], except sacrifice expressive power in favour of simplicity. We chose the smallest type language consistent with describing typical Erlang programs: we support disjoint unions, a limited form of complement, and recursive types; but not general unions or intersections, or conditional types [AWL94]. Our expectation was that these additional features would not help programmers, but would make inferred types less readable. We have succeeded in that our simplified inferred types are usually readable. On the other hand, we have encountered at least one situation where conditional types would be useful, as discussed in Section 9.3).

The type system presented here does not include function types, since first-class functions are not a feature of the current version of Erlang. However, the soon-to-be-released Erlang 4.4 supports first-class functions and we have successfully extended our prototype implementation to include function types. While we conjecture that our type checking algorithm without function types is complete, the work of Trifonov and Smith [TS96] shows a similar system *with* function types is incomplete, although they claim this is not a serious problem in practice. We discuss this in Section 8.1.

We demonstrate our system with a small program to manipulate sorted binary trees of keys and values. Figure 1 shows a type declaration and three Erlang functions with

Appears in the International Conference on Functional Programming, Amsterdam, June 1997

```
-deftype tree(A,B) =
    T when T = empty | branch{A,B,T,T}.

-type new() -> tree(0,0).
new() -> empty.

-type insert(A,B,tree(A,B)) -> tree(A,B).
insert(K0,V0,empty) ->
    {branch,K0,V0,empty,empty};
insert(K0,V0,{branch,K,V,L,R}) ->
    if  K0 <  K ->
           {branch,K,V,insert(K0,V0,L),R};
        K0 == K ->
           {branch,K0,V0,L,R};
        true    ->
           {branch,K,V,L,insert(K0,V0,R)}
    end.

-type lookup(A,tree(A,B)) -> B | error
       when B \ error.
lookup(K0,empty) -> error;
lookup(K0,{branch,K,V,L,R}) ->
    if  K0 <  K -> lookup(K0,L);
        K0 == K -> V;
        true    -> lookup(K0,R)
    end.
```

Figure 1: Binary Tree Example

type signatures. All type information is treated as annotations, which in Erlang are prefaced with a dash (`-`).

The basic data structures in Erlang are integers, floats, atoms (such as `empty`) and tuples (such as `{branch,K,V,L,R}`, where `branch` is an atom and `K`, `V`, `L`, `R` are structures). In our system, the types of these are respectively written `integer()`, `float()`, `empty`, and `branch{A,B,T,T}` (where `K` has type `A`, `V` has type `B`, and `L` and `R` have type `T`). (Why we write `branch{A,B,T,T}` instead of `{branch,A,B,T,T}` will be explained in Section 2.) The empty type, containing no values, is written `0` and the universal type, containing all values, is written `1`.

In Erlang, atoms and functions begin with a small letter while variables begin with a capital letter; functions may be distinguished from atoms because they are followed by parentheses. The same thing works at the type level, where `empty` is the type of an atom, `integer()` is a built-in type returning the type of all integers, and `A` is a type variable.

The `deftype` annotation defines the type `tree(A,B)`, while the three `type` annotations specify type signatures for the functions `new`, `insert`, and `lookup`. These annotations allow the user to document the program, and the type tool checks that the program conforms to that documentation. When multiple modules are processed, one may specify that the name of a type is exported without exporting its definition, thereby supporting type abstraction. For instance, if this module exports the type `tree` without exporting its definition, then the type system will ensure that only the three functions defined in it have access to the representation of trees.

The function `empty` takes no arguments and returns a tree; the function `insert` takes a key, a value, and a tree

```
new() -> A when empty <= A

insert(B, C, D) -> A
when
branch{E,F,G,A} <= A; branch{B,C,G,H} <= A;
branch{E,F,A,H} <= A;
branch{B,C,empty,empty} <= A;
D <= empty | branch{E,F,G,H};
G <= empty | branch{E,F,G,H};
H <= empty | branch{E,F,G,H};
H <= D; G <= D.

lookup(B, C) -> A
when
error <= A
C <= empty | branch{D,E,F,G};
F <= empty | branch{D,E,F,G};
G <= empty | branch{D,E,F,G};
E <= A; F <= C; G <= C.
```

Figure 2: Inferred Types

```
new() -> empty.

insert(D, E, F) -> A
when
empty | branch{D,E,A,A} <= A;
F <= empty | branch{D,E,F,F}.

lookup(1, B) -> error | A
when
B <= empty | branch{1, error | A, B, B};
A \ error.
```

Figure 3: Simplified Types

and returns a new tree with the key and value inserted; and the function `lookup` takes a tree and a key, and returns the value corresponding to that key or the atom `error` if the key is not found. To avoid any possibility of confusing a success and failure, the type specified for `lookup` adds the constraint that the value type `B` cannot include the atom `error`, written `B \ error`. In general, the form of a type signature is U **when** C, where U is a type and C is a set of constraints.

One problem with using a type system built around subtyping is that the set of constraints can be arbitrarily large, and inferred types can be difficult to read if they are not simplified. This is one reason for favouring simplicity over expressiveness: the more expressive the type language, the more difficult it is to simplify types. Experiments with our prototype have been promising, as for most functions we can derive a natural and readable version of its inferred type.

Figure 2 shows the types derived by our inference algorithm for the three tree operations, and Figure 3 shows the same types after simplification. The user-provided type signatures differ considerably from the inferred and simplified types. First, user-provided types may refer to type definitions, which do not appear in inferred or simplified types.

For instance, signature for `new` declares the function to have type `tree(0,0)`, where `0` is the empty type; whereas the simplified inferred type is simply `empty`, the empty tree. Second, the user-provided types may be more specific than the inferred type. For instance, the signature for `lookup` restricts the type `A` of the lookup key to be the same as the key fields in the tree, and the type `B` of the value fields in the tree not to contain `error`; whereas in the simplified inferred type `B` is the type of the tree, and `A` is the type of non-error value fields, but value fields may specifically include error.

Inferred types are principal, in that the inferred type of a function has all other possible types of that function as instances. However, as simplification demonstrates, the same type may be written in many forms. Simplified types are equivalent to the original, in that they represent the same set of values in the semantic domain.

To check user-supplied type signatures, we need to determine when a one type is an instance of another type. This problem turns out to be surprisingly tricky. The problem has been analysed by Trifonov and Smith [TS96] for a type language containing function types but no type union (conversely, ours contains type union but no function types), and they provide an algorithm which is sound but not complete, and leave the question of decidability open. In this paper, we give an algorithm which we believe is both sound and complete for our type language, but so far we have been unable to find a proof.

The paper is organised as follows. Section 2 introduces the syntax of expressions and types. Section 3 describes the typing rules for expressions. Section 4 presents a type reconstruction algorithm. Section 5 shows how to solve systems of typing constraints. Section 6 explains how to simplify types. Section 7 describes our algorithm for type signature matching. Section 8 sketches extensions to the type system for processes and higher-order functions. Section 9 relates our experience with the prototype implementation. Section 10 concludes.

2 Expressions and Types

The syntax of expressions we will be using is given in Figure 4. The language is a small subset of Erlang, containing variables, constructor applications (called tagged tuples in Erlang), function calls and simple case expressions.

We use the overbar to indicate a sequence of objects, for example $\overline{E} = E_1, \ldots, E_n$. The length of the sequence is normally discernable from the context. Each constructor has a fixed arity, so in $c\{\overline{E}\}$ the length of the sequence of expressions $\overline{E}$ is equal to the arity of the constructor c. When there is no length-fixing context, the length of the sequence is arbitrary.

Standard Erlang doesn't have constructor applications. It has atoms, which we represent as nullary constructor applications, and it has arbitrary tuples, written $\{E_1, \ldots, E_n\}$. In our type system we use the convention that if the first element of a tuple in an Erlang program is an atom, then we convert the tuple to a tagged tuple, using the atom name as the constructor. If the first element is some other expression, then the tuple is an anonymous tuple, and we assign the constructor $tuple_n$, where n is its arity.

There are several other differences between Erlang and our small subset:

- All pattern matching is compiled into simple **case** expressions. Algorithms to do this are well known [Aug85, Wad87].
- Case expressions always have a default alternative, of the form $X \to E$. A special form of this is used to indicate that the alternative should never be taken: $X \to \mathit{empty}(X)$, where *empty* is a built-in function which always fails.
- Interprocess communication is omitted for now. We discuss an extension to handle this in Section 8.2.

Programs consist of a set of top-level function declarations, which may be recursive. Our type system will assign polymorphic types to these function declarations.

The semantics of our language is *strict* i.e. function arguments are always evaluated before the function is called, and constructor arguments are evaluated before the structure is built.

The syntax of types is given in Figure 5.

Prime types are written $c\{U_1, \ldots, U_n\}$, and represent the type of a tagged tuple with tag c, arity n, and field types $U_1 \ldots U_n$.

The general form of a union type, U, is

f, g			function names
c, d			constructors
X, Y, Z			variables
E	::=	X	expression
		$\mid\ c\{E_1, \ldots, E_n\}$	
		$\mid\ f(E_1, \ldots, E_n)$	
		$\mid\ \mathbf{case}\ E_0\ \mathbf{of}$	
		$\quad c_1\{\overline{X_1}\} \to E_1;$	
		$\quad \vdots$	
		$\quad c_n\{\overline{X_n}\} \to E_n;$	
		$\quad X \to E_{n+1}$	
		$\mathbf{end}$	
prog	::=	$f_1(\overline{X_1}) \to E_1;$	program
		$\vdots$	
		$f_n(\overline{X_n}) \to E_n$	

Figure 4: Expressions

c, d			constructors
α, β			type variables
U, V	::=	$P \mid U$	union type
		$\mid\ R$	
P, Q	::=	$c\{\overline{U}\}$	prime type
R	::=	α^{cs}	remainder
		$\mid\ 1^{cs}$	
		$\mid\ 0$	

Figure 5: Types

$$c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid R$$

where R represents a *remainder*. A remainder is either a type variable α^{cs}, the universal type 1^{cs}, or the empty type 0.

The syntax α^{cs} for a type variable means that α ranges over all types not containing elements with the tags $c_1, \ldots, c_n$ $(= cs)$. The tags cs are called the *excluded tags* on the type variable α. Similarly, the syntax 1^{cs} represents the universal type excluding types with tags $c_1, \ldots, c_n$. If the list cs is empty, it is normally omitted. When the remainder is 0 or 1^{cs}, the type is called a monotype.

The operator '|' is a disjoint union. This means that in the type expression $P|U$, the types P and U cannot overlap under any legal substitution for the free variables of either type. This implies two further restrictions on the form of the general union type $c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid R$: the tags cs must be distinct, and if R is of the form α^{ds} or 1^{ds} then $cs \subseteq ds$, where cs is $c_1, \ldots, c_n$.

This syntax of types provides exactly the level of generality we require for typing Erlang. The main differences between this system and that of Aiken and Wimmers [AW93] are:

- the lack of a general intersection, instead we only allow excluded tags (which would be represented with intersection in the Aiken-Wimmers system).
- the lack of function types. As mentioned in the introduction, we leave out function types because the current version of Erlang doesn't support them.

Our union operator is equivalent to that used in the Aiken-Wimmers system except that we do not allow general (non-disjoint) union on the left of a constraint. However, general union on the left of a constraint can always be replaced by several constraints without union operators.

The presence of the universal type is interesting: in fact, we never infer types containing the universal type, although some types are simplified by replacing type variables in negative positions[1] by the universal type. Experimentation with the prototype persuaded us that the universal type in a positive position is useful for expressing the type of certain Erlang built-in functions for which our type system cannot provide precise types. For example, the function `element` selects the n^{th} element of an arbitrary tuple. The best type for this function in our system is $(int(), tuple()) \to 1$, where $int()$ and $tuple()$ are built-in types representing integers and tuples respectively. Without the universal type, it would not be possible to give a sound type to this function.

2.1 Subtyping Constraints

Subtyping constraints are written $U \subseteq V$. A constraint is valid if all values of the type U are also values of the type V. We use the identifiers C and D to refer to sets of subtyping constraints, where a constraint set is valid if and only if all the individual constraints are valid.

If the types in a constraint set contain type variables, the validity of the set depends on a substitution σ from type variables to *type values*. A type value is a certain portion of the semantic domain, for example: 'integer', 'list of characters', and 'tree of integer pairs' are all type values (a set of values is another way of thinking of it). An ideal model similar to that in [MPS86] or recursive type equations as used by Trifonov and Smith [TS96] would provide a suitable framework for defining type values.

A substitution σ is a *solution* of a constraint set iff its application renders the constraint set valid.

We use some shorthand forms for sequences of similar subtyping constraints:

$$\begin{array}{ll} \overline{U} \subseteq \overline{V} & \Rightarrow U_1 \subseteq V_1, \ldots, U_n \subseteq V_n \\ U \subseteq \overline{V} & \Rightarrow U \subseteq V_1, \ldots, U \subseteq V_n \\ \overline{U} \subseteq V & \Rightarrow U_1 \subseteq V, \ldots, U_n \subseteq V \end{array}$$

2.2 Entailment

We also introduce an entailment operator over constraint sets, written as follows:

$$C \Vdash D$$

which is true if all solutions of C are also solutions of D. Entailment is reflexive and transitive.

We will also use another form of entailment:

$$\forall\overline{\alpha}.\exists\overline{\beta}.C \Vdash D$$

which is true if every substitution mapping $\overline{\alpha}$ to type values that solves C can be extended to a solution of D by adding mappings for $\overline{\beta}$. It follows from this definition that $\overline{\alpha}$ must contain at least the free type variables of C. If the free type variables of D are a subset of those of C, then the two operators above are equivalent.

The latter operator is used to define type instance, in Section 7.

2.3 Function Types

Although our type language does not have a general function space operator $(\to)$, we assign type schemes to top-level functions in the source program. Top-level function type schemes are polymorphic and constrained. They take the following form:

$$\forall\overline{\alpha}.(\overline{U}) \to V \textbf{ when } C$$

Function type schemes cannot have any free type variables (that is, $\mathrm{FTV}((\overline{U}) \to V \textbf{ when } C) \subseteq \overline{\alpha}$).

3 Typing Rules

We give the typing rules for expressions in two forms. This section describes a traditional set of typing rules for subtyping, and the following two sections describe our algorithm to determine the most general typing of an expression.

The typing rules in traditional format are given in Figure 6. The form of a judgement is

$$F;\ A;\ C \vdash E : U$$

Elements of F have the form $f : \forall\overline{\alpha}.(\overline{U}) \to V \textbf{ when } C$. Elements of A have the form $x : U$. The judgement asserts that under function assumption F and variable assumption

[1] A type is in a negative position if it appears in an argument position of a function type or on the right hand side of a constraint. Result types and the left hand sides of constraints are positive positions.

$$\text{Var}\ \frac{}{F;\ A, X : U;\ C \vdash X : U} \qquad \text{Sub}\ \frac{F;\ A;\ C \vdash E : U \qquad C \Vdash U \subseteq V}{F;\ A;\ C \vdash E : V}$$

$$\text{Fun}\ \frac{}{F, f : \forall \overline{\alpha}.(\overline{U}) \to V\ \textbf{when}\ D;\ A;\ C, D[\overline{V}/\overline{\alpha}] \vdash f : ((\overline{U}) \to V)[\overline{V}/\overline{\alpha}]}$$

$$\text{Con}\ \frac{F;\ A;\ C \vdash \overline{E} : \overline{U}}{F;\ A;\ C \vdash c\{\overline{E}\} : c\{\overline{U}\}} \qquad \text{Call}\ \frac{F;\ A;\ C \vdash f : (\overline{U}) \to V \qquad F;\ A;\ C \vdash \overline{E} : \overline{U}}{F;\ A;\ C \vdash f(\overline{E}) : V}$$

$$\text{Case}\ \frac{\begin{array}{c} F;\ A;\ C \vdash E_0 : c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid U \\ F;\ A, \overline{X_1} : \overline{U_1};\ C \vdash E_1 : V \quad \ldots \quad F;\ A, \overline{X_n} : \overline{U_n};\ C \vdash E_n : V \\ F;\ A, X : U;\ C \vdash E_{n+1} : V \end{array}}{F;\ A;\ C \vdash (\textbf{case}\ E_0\ \textbf{of}\ c_1\{\overline{X_1}\} \to E_1; \ldots c_n\{\overline{X_n}\} \to E_n; X \to E_{n+1}\ \textbf{end})\ : V}$$

$$\text{Multi}\ \frac{F;\ A;\ C \vdash E_1 : U_1 \quad \ldots \quad F;\ A;\ C \vdash E_n : U_n}{F;\ A;\ C \vdash \overline{E} : \overline{U}}$$

$$\text{Def}\ \frac{F, f : ((\overline{U}) \to V\ \textbf{when}\ C);\ \overline{X} : \overline{U};\ C \vdash E : V \quad \text{FTV}((\overline{U}) \to V\ \textbf{when}\ C) = \overline{\alpha}}{F;\ \emptyset;\ C \vdash f(\overline{X}) \to E : (\forall \overline{\alpha}.(\overline{U}) \to V\ \textbf{when}\ C)}$$

Figure 6: Typing Rules

A, expression E has type U whenever the constraint set C is satisfied.

The typing rule for function names (*Fun*) instantiates the quantified variables with arbitrary types. It also copies the constraints D from the function type into the current constraint set C, ensuring that the constraints on the function type are satisfied each time the function is called.

As mentioned in Section 2 we assume one built-in function *empty*, with type

$$\mathit{empty} : (0) \to 0$$

The empty function is the only means by which a function may fail: if empty is ever called at runtime, the program exits.

The rules allow any expression to be assigned a typing, but only typings in which C has at least one solution are useful (these are called valid typings). If any valid typing exists, then it is guaranteed that the program will never call *empty* at runtime.

Subtyping is introduced with the subsumption rule (*Sub*), which allows the type U of an expression to be replaced with any larger type V provided the entailment relation $C \Vdash U \subseteq V$ holds (the entailment operator was described in Section 2.1). The simplest way for this relation to hold is if C contains $U \subseteq V$.

For **case** expressions, the type of the selector E_0 is required to be smaller than the union of the types of the pattern alternatives and the type of the variable X bound in the default alternative ($c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid U$, where each type $c_i\{\overline{U_i}\}$ is the type of a pattern, and U is the type of X). By the rules of the disjoint union operator, this implies that the type U cannot contain any elements with the tags $c_1, \ldots, c_n$. Compare this with traditional Hindley-Milner type checking, where there is no way of representing or exploiting the fact that the type of the default variable may exclude types handled by earlier case branches.

In order to type an arbitrary expression we need to know two things: whether a valid typing exists, and the most general form of that typing.

Note that in general there is no unique most-general type for a given function in this system. Indeed, the type simplification process that we will describe in Section 6 attempts to replace a type with an equivalent simpler type. Two types are equivalent if they are instances of each other; we will discuss subtype instance in Section 7.

In our implementation, typings are derived in two stages:

- Firstly, type reconstruction derives a type and a set of typing constraints for the expression. This is the most general typing of the expression if the constraints are satisfiable.
- The second stage, constraint set reduction, determines the solvability of the constraint set. If the set is solvable, then we have a type for the expression.

These two stages are described in the next two sections.

4 Type reconstruction

The typing rules can be used in a syntax-directed way to generate a constrained type as follows:

- Assign a fresh type variable to each new bound variable. Thus the assumption A binds variables to unique type variables.
- Assign a fresh type variable for the type of each case expression.
- Use subsumption in the following places: to promote the type of each function argument in a function call, to promote the type of each branch in a **case** expression to a common supertype (which is a fresh type variable), and to promote the type of the selector in a **case** expression to the union of the pattern types. For each use of subsumption place the required constraint in C, so that the entailment relation is trivially satisfied.

$P \mid U \subseteq V$	$\Rightarrow$	$P \subseteq V, U \subseteq V$	
$0 \subseteq U$	$\Rightarrow$	none	
$1^{cs} \subseteq 0$	$\Rightarrow$	fail	
$1^{cs} \subseteq c\{\overline{U}\} \mid U$	$\Rightarrow$	$1 \subseteq \overline{U},\ 1^{cs} \subseteq U$	if $c \notin cs$
		$1^{cs} \subseteq U$	otherwise
$1^{cs} \subseteq 1^{ds}$	$\Rightarrow$	none	if $ds \subseteq cs$
		fail	otherwise
$1^{cs} \subseteq \alpha^{ds}$	$\Rightarrow$	$1^{cs} \subseteq \alpha^{ds}$	if $ds \subseteq cs$
		fail	otherwise
$c\{\overline{U}\} \subseteq 0$	$\Rightarrow$	fail	
$c\{\overline{U}\} \subseteq c'\{\overline{U}'\} \mid U$	$\Rightarrow$	$\overline{U} \subseteq \overline{U}'$	if $c = c'$
		$c\{\overline{U}\} \subseteq U$	otherwise
$c\{\overline{U}\} \subseteq 1^{cs}$	$\Rightarrow$	none	if $c \notin cs$
		fail	otherwise
$c\{\overline{U}\} \subseteq \alpha^{cs}$	$\Rightarrow$	$c\{\overline{U}\} \subseteq \alpha^{cs}$	if $c \notin cs$
		fail	otherwise
$U \subseteq \alpha^{cs},\ \alpha^{cs} \subseteq V$	$\Rightarrow$	$U \subseteq V, U \subseteq \alpha^{cs}, \alpha^{cs} \subseteq V$	

Figure 7: Reduction Rules

Proposition 1 (principal type property) Every type derivable for a function using the typing rules is an instance of the type derived by the type reconstruction algorithm.

The proof of this proposition is similar to Mitchell's proof of principal types for his subtyping system [Mit91].

5 Constraint Set Reduction

Constraint Reduction is the process of determining whether a system of constraints is solvable. If a constraint system generated by the type reconstruction algorithm is not solvable, it indicates that the program has a type error.

We do not have to discover an actual solution to the set, merely prove that one or more solutions exist. This is done by repeatedly transforming the constraint system while maintaining transitive closure and checking for type errors. The transformation system is given in Figure 7. Each rule applies to one or more constraints from the current set, and yields one of the following results:

- 'fail', indicating that the constraint system has no solutions, and the original function therefore contains a type error,
- 'none', indicating that the constraint should be removed from the system,
- one or more transformed constraints. The original constraints are to be removed from the set.

A constraint set is only fully reduced when both of the following conditions apply:

1. Each constraint in the set is of the form $\alpha^{cs} \subseteq U$ or $U \subseteq \alpha^{cs}$, and
2. Any rule that can be applied would take the constraint set into a state that has occurred before.

If the reduction process terminates without failure, the resulting constraint set is said to be *consistent*.

For reasons of efficiency, we ignore the strictness of constructors for type inference purposes. In other words, the type of $c\{\bot\}$ in our system is $c\{0\}$, not 0. A similar situation is found in [AW93] where some solutions to the constraint set are discarded for efficiency.

Proposition 2 When applied to an arbitrary constraint set, the reduction process either fails or terminates yielding a consistent constraint set.

Proof sketch. Observe that all rules in the transformation algorithm, except the last (transitivity), either fail or split a constraint into zero or more constraints on subterms of the originals, until all the constraints have a variable on one side or the other. All the cases are covered; so any constraint set can be reduced to a state where all constraints are of the form $\alpha^{cs} \subseteq U$ or $U \subseteq \alpha^{cs}$.

The transitivity rule forms a new constraint from existing types. This rule cannot be applied indefinitely without reaching a fixed point because there are only a finite number of possible constraints to add. We must therefore reach a state where (1) all constraints are on variables, and (2) the only rule which can be applied to change the set is transitivity, and all possible constraints have already been added to the set. □

Proposition 3 A consistent constraint set is solvable.

Proof sketch. We prove this property by relation with the inductive constraint system of Aiken and Wimmers [AW93]. A consistent constraint set in our system can be transformed into an inductive constraint set, and the process cannot fail. Inductive constraint sets were shown to be solvable by Aiken and Wimmers.

The subject of inductive form and the algorithm for converting a consistent constraint set into an inductive constraint set are discussed in Section 7.

6 Type Simplification

The type assigned to a function by the type inference algorithm can be large and unwieldy, making it difficult for a user to interpret, and expensive for an implementation to deal with. Therefore we apply a number of simplifying transformations to the type, in an attempt to derive an equivalent type that contains fewer typing constraints.

Our simplification transformations are similar to those of Fähndrich and Aiken [FA96], who use type simplification amongst other techniques to show that set-constraint-based analyses are scalable to large examples.

We have not found a suitable 'normal form' for a constrained type, nor have other researchers in this area. We cannot therefore hope for a simplification procedure that is complete. There are sometimes typing constraints which cannot be eliminated; one example is when the constraints are being used to represent a recursive type. However, the transformations given in this section are based on heuristics that we have found to be effective. In many cases, the derived type can be simplified to the type that one would normally assign to the function.

The constraint set generated by the type inference algorithm satisfies three important properties that we can make use of during simplification.

- Each constraint in the system is of the form $\alpha^{cs} \subseteq U$ or $U \subseteq \alpha^{cs}$, termed upper and lower bounds on α respectively. For an implementation, this means that we can represent the constraint set as a mapping from variables to sets of upper and lower bounds. This property will hold throughout simplification.
- Separate occurrences of the same variable will have identical excluded tag lists. This property will hold throughout simplification.
- The constraint set is transitively closed. That is, for each constraint pair $\alpha^{cs} \subseteq \beta^{ds}$ and $\beta^{ds} \subseteq U$, the set contains the constraint $\alpha^{cs} \subseteq U$, and for each pair $U \subseteq \beta^{ds}$ and $\beta^{ds} \subseteq \alpha^{cs}$, the set contains $U \subseteq \alpha^{cs}$. We will not retain this property during simplification, although the transitive closure can always be recovered by adding the necessary constraints to the system.

6.1 General Simplifications

The following transformations are applied whenever the opportunity arises during the simplification process.

$$\begin{array}{lll} \alpha^{cs} \subseteq \alpha^{cs} & \Rightarrow \text{none} & \\ 0 \subseteq \alpha^{cs} & \Rightarrow \text{none} & \\ \alpha^{cs} \subseteq 1^{ds} & \Rightarrow \text{none} & \text{if } ds \subseteq cs \end{array}$$

6.2 Eliminating Cycles

Our implementation eliminates cycles in the constraint set as a first step, since it has a dramatic effect on the efficiency of the rest of the simplification process and is relatively cheap to perform.

The idea is to first identify all cycles between variables. Since in any solution of this constraint set the values of these variables must be identical, we can replace all occurrences of the variables with a new variable. The transformation is given in Figure 8a.

If the constraint set is treated as a graph with the variables as nodes, then cycles can be found in linear time using standard algorithms, and removed in linear time using the above substitution. The result is a directed acyclic graph. A constraint set in this form is called *contractive* [TS96].

6.3 Combining Upper and Lower Bounds

In general, a variable may have several upper and lower bounds. The purpose of this simplification stage is to reduce the number of upper and lower bounds on each variable by combining them where possible.

6.3.1 Combining Lower Bounds

It is always possible to combine the lower bounds on a variable such that we achieve a normal form:

$$\begin{array}{rl} \beta_1 & \subseteq \alpha^{ds} \\ \vdots & \\ \beta_n & \subseteq \alpha^{ds} \\ c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid 0 & \subseteq \alpha^{ds} \end{array}$$

After combination, the lower bounds on a variable will consist of zero or more variable-only lower bounds, and at most one constructed lower bound (a union type where the remainder is 0). This is achieved by applying the transformation rules of Figure 8b, and collecting the (now distinct) prime type lower bounds on each variable into a singe union type.

For example, if we have the following constraint set:

$$c\{U\} \subseteq \alpha, \quad c\{V\} \subseteq \alpha$$

then we can replace this with

$$c\{\beta\} \subseteq \alpha, \quad U \subseteq \beta, \quad V \subseteq \beta$$

where there is now only a single lower bound on the variable α. There are now two lower bounds on the variable β, which can be combined in the same way.

6.3.2 Combining Upper Bounds

Unlike lower bounds, we have found no useful normal form for upper bounds because we cannot compute the intersection of several union types and represent the result in our type syntax. Instead, we use some heuristics to combine upper bounds where possible.

An example of one of the transformations used is given in Figure 8c, where two upper bounds are combined if they are monotypes. This is an important transformation for our type checking algorithm, in Section 7.

6.4 Transitive Kernel

During the simplification process, we work with the *transitive kernel* of the constraint set. The transitive kernel of a transitively closed constraint set C is defined as the smallest constraint set D such that the transitive closure of D is C. If C is contractive (Section 6.2), then there is a single unique D, computed by applying the transformation rule in Figure 8d as many times as possible to the constraint set.

The advantages of working with the transitive kernel are:

- The set is smaller, but contains the same information. The original constraint set can be recovered by forming the transitive closure.
- Our simplifying transformations are equally valid when applied to the transitive kernel. In fact, removing the transitive constraints can enable some transformations that were not previously possible.

As an example of the second point, the variable elimination transformation is only applicable when a variable has a single upper or lower bound; if there are other constraints on the variable that are present due to transitivity then the transformation cannot be applied. We could take into account these transitive constraints during the transformation, but it is simpler to compute the transitive kernel once and maintain it throughout simplification.

6.5 Eliminating Variables

The transformation described in this section is simple, and yet remarkably effective in simplifying types. The basic idea is to find a variable with a single upper bound or a single

$$\alpha_1^{cs_1} \subseteq \alpha_2^{cs_2}, \ldots, \alpha_n^{cs_n} \subseteq \alpha_1^{cs_1}, C \Rightarrow \quad C[\beta^{ds}/\alpha_1^{cs_n}, \ldots, \beta^{ds}/\alpha_n^{cs_n}] \quad \text{where } \beta \text{ fresh}, \quad ds = cs_1 \cup \ldots \cup cs_n$$

a. Eliminating Cycles

$$c\{\overline{U}\} \mid U \subseteq \alpha^{cs} \quad \Rightarrow c\{\overline{U}\} \subseteq \alpha^{cs},\ U \subseteq \alpha^{cs}$$
$$c\{\overline{U}\} \subseteq \alpha^{cs}, c\{\overline{U}'\} \subseteq \alpha^{cs} \quad \Rightarrow c\{\overline{\alpha}\} \subseteq \alpha^{cs},\ \overline{U} \subseteq \overline{\alpha},\ \overline{U}' \subseteq \overline{\alpha} \quad \text{where } \overline{\alpha} \text{ fresh}$$

b. Combining Lower Bounds

$$\begin{array}{l}\alpha^{cs} \subseteq c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid a_1\{\overline{V_1}\} \mid \ldots \mid a_m\{\overline{V_m}\} \mid R_1 \\ \alpha^{cs} \subseteq c_1\{\overline{U_1'}\} \mid \ldots \mid c_n\{\overline{U_n'}\} \mid b_1\{\overline{V_1'}\} \mid \ldots \mid b_l\{\overline{V_l'}\} \mid R_2 \\ \quad \Rightarrow \ \alpha^{cs} \subseteq c_1\{\overline{\alpha_1}\} \mid \ldots \mid c_n\{\overline{\alpha_n}\} \mid \{a_i\{\overline{V_i}\} \mid \mathrm{in}(a_i, R_2)\} \mid \{b_i\{\overline{V_i'}\} \mid \mathrm{in}(b_i, R_1)\} \\ \qquad \overline{U_i} \subseteq \overline{\alpha_i} \quad 1 \le i \le n \\ \qquad \overline{U_i'} \subseteq \overline{\alpha_i} \quad 1 \le i \le n \\ \qquad \text{where } \overline{\alpha_i} \text{ fresh} \quad 1 \le i \le n \\ \qquad\qquad \mathrm{in}(c, 1^{cs}) \quad = c \notin cs \\ \qquad\qquad \mathrm{in}(c, 0) \quad = \mathit{false}\end{array}$$

c. Combining Upper Bounds

$$U_1 \subseteq U_2, \ldots, U_{n-1} \subseteq U_n, U_1 \subseteq U_n \Rightarrow U_1 \subseteq U_2, \ldots, U_{n-1} \subseteq U_n$$

d. Transitive Kernel

Figure 8: Simplifying Transformations

lower bound and replace it with this bound when it is legal to do so. For example, the type α **when** $c\{\} \subseteq \alpha$ is equivalent to simply $c\{\}$, and the first type can be simplified to the second by replacing the variable α with its single lower bound $c\{\}$.

The first transformation applies to variables with a single upper bound:

$$U \textbf{ when } \alpha^{cs} \subseteq V, C \ \Rightarrow \ (U \textbf{ when } C)[V/(\alpha^{cs})]$$

There are some restrictions on this transformation:

- There are no cycles in the constraint set involving α^{cs},
- α^{cs} appears only negatively in U and C,
- α^{cs} must have variable-only lower bounds, unless V is a variable. This is to retain the invariant that all constraints are on variables.
- The substitution $(U \textbf{ when } C)[V/\alpha^{cs}]$ must be legal with respect to the disjoint union operator.

The first restriction is to prevent the transformation from being applied indefinitely, as would be the case if the variable α^{cs} were part of the definition of a recursive type in the constraint set.

The dual of this transformation applies to single lower bounds:

$$U \textbf{ when } V \subseteq \alpha^{cs}, C \ \Rightarrow \ (U \textbf{ when } C)[V/(\alpha^{cs})]$$

The restrictions are similar to the upper-bound case:

- There are no cycles in the constraint set involving α^{cs},
- α^{cs} appears only positively in U and C,
- α^{cs} must have variable-only upper bounds, unless V is a variable.

There is no need for a restriction equivalent to the fourth restriction for upper bounds, since it would always be satisfied.

There are two subsidiary transformations, which apply to variables with no upper bounds or no lower bounds:

If α^{cs} has no upper bounds and appears only negatively in U and C:

$$U \textbf{ when } C \ \Rightarrow \ (U \textbf{ when } C)[(1^{cs})/(\alpha^{cs})]$$

And the dual case, when α^{cs} has no lower bounds and appears only positively in U and C:

$$U \textbf{ when } C \ \Rightarrow \ (U \textbf{ when } C)[0/(\alpha^{cs})]$$

6.6 Eliminating lower bounds

The following transformation has less restrictions than the variable elimination transformation, but it is less beneficial in general since it doesn't eliminate any type variables, only constraints. We generally use this transformation as the last stage of simplification.

$$U \textbf{ when } c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid 0 \subseteq \alpha^{ds}, C \Rightarrow (U \textbf{ when } C)[(c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid \alpha^{(ds \cup cs)})/\alpha^{ds}]$$

- α^{cs} must have no constructed upper bounds (this is to retain the property that constraints are on variables).
- There must be no cycles in the constraint set involving α^{cs}.

7 Type Checking

Type inference systems normally provide a way for the user to supply a type for a function and have that type checked against the inferred one. This serves two purposes:

- The user-supplied types serve as documentation for the function, and the documentation is always guaranteed to be correct because it is checked by the type system.
- The user-supplied type may be more restrictive than the inferred type. This is useful in cases where the user wishes to place additional restrictions on the use of a function over those provided by the inferred type, or to use a more general definition of a function when this would be more efficient.

A user-supplied type is valid if it is an instance of the inferred type. In Hindley-Milner type systems, an instance of a type is formed by replacing one or more of its universally-quantified type variables by more specific types, and it is straightforward to check whether one type is an instance of another.

When subtyping constraints are involved, however, the problem is somewhat more difficult. Determining when one constrained type is an instance of another has so far received little attention in the literature [TS96, FF96]. In this section, we outline an algorithm for determining this relation. We do not have proofs of soundness or completeness, but we also have not found any counter examples to either property. There doesn't seem to be a straightforward extension of our algorithm to handle function types, since the obvious extension suffers from incompleteness (Section 8.1).

In a subtyping system, the instance relation is really a subtype relation: we are determining whether one type represents a smaller portion of the semantic domain than another. The term *instance* makes sense in Hindley-Milner style systems where the problem reduces to an instance relation, but in a subtyping system we must be more general.

The subtyping relation over quantified constrained types can be defined using the entailment operator. For two types $(\forall\overline{\alpha}.U$ **when** $C)$ and $(\forall\overline{\beta}.V$ **when** $D)$ where the quantified variables $\overline{\alpha}$ and $\overline{\beta}$ are distinct,

$$(\forall\overline{\alpha}.U \textbf{ when } C) \subseteq (\forall\overline{\beta}.V \textbf{ when } D) \text{ iff } \forall\overline{\beta}.\exists\overline{\alpha}.D \Vdash U \subseteq V, C$$

In the context of type checking, the term on the left of the subtype relation is the inferred type, and the type on the right is the user-supplied type.

In brief, the algorithm works as follows. The constraint sets on either side of the entailment relation are converted to inductive form [AW93], and canonical lower and upper bounds are computed for each type variable in the set D. The algorithm then proceeds in a similar way to that proposed by Trifonov/Smith [TS96], the main difference being that canonical upper bounds are more complicated to compute since we cannot form the intersection of several types in general.

7.1 Inductive Form

Our entailment algorithm makes use of an inductive form for constraint sets [AW93]. An inductive constraint set can be formed from a consistent constraint set (i.e. one that has been reduced, Section 5), by first choosing an ordering on variable names and then applying the transformations in Figure 9.

The transformation makes use of a function $TLV(U)$, which returns the top-level variable (the variable remainder) of the type U if it exists. Thus $\alpha > TLV(U)$ iff $TLV(U)$ exists and is smaller than α in the chosen variable ordering.

Two other operations are used in the transformation. The first, $\oplus$, forms the union of a type and a set of constructor applications whose elements are all 1:

$$U \oplus cs \Rightarrow c_1\{\overline{1}\} \mid \ldots \mid c_n\{\overline{1}\} \mid (U \backslash cs)$$

The second operator is $\backslash$, which excludes certain tags from a type:

$$\begin{array}{lcll} (c\{\overline{U}\}|U)\backslash cs & \Rightarrow & U\backslash cs & \text{if } c \in cs \\ & & c\{\overline{U}\}|(U\backslash cs) & \text{otherwise} \\ (\alpha^{ds})\backslash cs & \Rightarrow & \alpha^{(ds \cup cs)} & \\ (1^{ds})\backslash cs & \Rightarrow & 1^{(ds \cup cs)} & \\ 0^{cs} & \Rightarrow & 0 & \end{array}$$

Once the transformations have been fully applied, each constraint in the set will be of the form $\alpha \subseteq U$ or $U \subseteq \alpha$, where $\alpha > TLV(U)$. In other words, each constraint is expressed as a bound on a variable α that only refers to variables lower than α at the top level. In the Aiken-Wimmers system this allows the constraint set to be solved, whereas we use this form to calculate upper bounds for our entailment algorithm.

One problem with using constraints in inductive form is that we no longer have the invariant that separate occurrences of the same variable have identical sets of excluded tags. Our entailment algorithm needs to reduce constraints which do not satisfy this property, so we must alter the reduction algorithm accordingly. The new reduction algorithm is identical to Figure 7 except that we remove the transitivity rule (the last rule in the figure) and add the following two rules:

$$\begin{array}{ll} \alpha^{cs} \subseteq \alpha^{ds} & \Rightarrow \alpha^{cs} \subseteq 1^{ds} \\ U \subseteq \alpha^{cs},\ \alpha^{ds} \subseteq V & \Rightarrow U \subseteq V \oplus ds, U \subseteq \alpha^{cs},\ \alpha^{ds} \subseteq V \end{array}$$

The second of the two new rules is a new transitivity rule that takes into account the differing excluded tags on the variable α.

In the following discussion of the entailment algorithm, we will use the functions reduce(C) and induct(C) to refer to the new reduction and inductive transformations respectively.

7.2 Computing Canonical Upper and Lower Bounds

We have already presented a canonicalisation of lower bounds, as part of type simplification in Section 6.3.1. We use that transformation again here. If C is an inductive constraint set that has had the lower-bound transformation applied, then the function lowers(C, α) returns a pair consisting of a set of variables (all less than α in the inductive ordering) and a union type with a remainder of zero.

Canonicalising upper bounds is somewhat more difficult, as we cannot form the intersection of several union types

$$
\begin{array}{lll}
c\{\overline{U}\} \subseteq \alpha^{cs} & \Rightarrow c\{\overline{U}\} \subseteq \alpha & \\
1^{ds} \subseteq \alpha^{cs} & \Rightarrow 1^{cs} \subseteq \alpha & \\
\alpha^{cs} \subseteq U & \Rightarrow \alpha \subseteq U \oplus cs & \alpha > TLV(U) \\
\alpha^{as} \subseteq c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid \beta^{ds} & \Rightarrow \alpha^{as} \subseteq c_1\{\overline{U_1}\} \mid \ldots \mid c_n\{\overline{U_n}\} \mid 1^{ds}, \quad \alpha^{(as \cup cs)} \subseteq \beta & \alpha < \beta \\
\alpha^{cs} \subseteq \beta,\ \beta \subseteq U & \Rightarrow \alpha^{cs} \subseteq U &
\end{array}
$$

Figure 9: Inductive Form

in our type language. However, the following two sections describe a method that allows us to combine several upper bounds into a single type for the purposes of our entailment algorithm. The problem is to determine entailments of the following form:

$$\alpha \subseteq V_1, \ldots, \alpha \subseteq V_n, C \Vdash \alpha \subseteq U$$

which is true if and only if

$$\alpha \subseteq V_1, \ldots, \alpha \subseteq V_n, C \Vdash V_1 \cap \ldots \cap V_n \subseteq U$$

The problem is that we cannot compute the value of the intersection in general. However, for the purposes of entailment, there are two methods that can be used to prove this entailment relation.

7.2.1 U is a monotype

When U is a monotype, we can calculate the largest type that each top-level variable in the intersection can take, reducing the intersection to an intersection between monotypes which we can reduce to a single type.

Definition 1 If $V_1 \ldots V_n$ are the upper bounds of α in the inductive constraint set C, then the *absolute upper bound* of the type variable α is calculated as follows: for each top level variable β^{cs} in $V_1 \ldots V_n$, replace β^{cs} with $V \backslash cs$ where V is the absolute upper bound of β. Then form the intersection of the remaining monotypes using the transformations from Section 6.3.2.

This definition is recursive, but it is guaranteed to terminate because we are working with inductive constraints. The lowest variable in the ordering cannot refer to any top-level variables in its upper bounds, so its absolute upper bound can be calculated, the second variable in the ordering can only refer to the first, and so on.

Using the absolute upper bound we can form the correct constraint if U is a monotype: for the entailment to hold,

$$\text{upper}(C, \alpha) \subseteq U$$

where $\text{upper}(C, \alpha)$ is the absolute upper bound of the type variable α in the inductive constraint set C. Absolute upper bounds can be pre-calculated for any given constraint set.

7.2.2 U is not a monotype

When the type U is not a monotype, using the absolute upper bound is not good enough: it doesn't allow us to prove the following entailment:

$$\gamma \subseteq c\{\} \mid \alpha^{\{c,d\}}, \gamma \subseteq d\{\} \mid \beta^{\{c,d\}} \Vdash \gamma \subseteq \alpha^{\{c,d\}}$$

The absolute upper bound for γ is $1^{\{c,d\}}$, and it is not true that $1^{\{c,d\}} \subseteq \alpha^{\{c,d\}}$ for all α. We need to calculate the absolute upper bound with respect to a particular variable, α in this case. In general, the problem is how to prove the entailment relation:

$$C \Vdash V_1 \cap \ldots \cap V_n \subseteq P_1 \mid \ldots \mid P_n \mid \beta^{cs}$$

Firstly, we can split the inequation on the right as follows:

$$
\begin{array}{ll}
V_1 \cap \ldots \cap V_n & \subseteq P_1 \mid \ldots \mid P_n \mid 1^{cs} \\
(V_1 \cap \ldots \cap V_n) \backslash ds & \subseteq \beta
\end{array}
$$

where ds are the tags of $P_1 \ldots P_n$. The first inequation has a monotype on the right, so we can solve it using the method above. For the second inequation, we can form the absolute upper bound with respect to β for the type on the left:

Definition 2 If $V_1 \ldots V_n$ are the upper bounds of α in the inductive constraint set C, then the absolute upper bound of α with respect to β is calculated as follows: for each top-level variable γ^{cs} in $V_1 \ldots V_n$ where $\gamma \neq \beta$, replace γ^{cs} with $V \backslash cs$ where V is the absolute upper bound of γ with respect to β. Express the result as an intersection of unions by distributing '|' where necessary. Combine all intersections of monotypes into a single monotype using the transformations of Section 6.3.2.

The absolute upper bound of α with respect to β looks like this:

$$(U_1 \mid \beta^{cs_1}) \cap \ldots \cap (U_n \mid \beta^{cs_n}) \cap M$$

where M is a monotype. If we multiply out the intersection, we get

$$((U_1 \mid 0) \cap \ldots \cap (U_n \mid 0) \cap M) \cup \ldots$$

where the final '...' represents the rest of the terms, all of which are intersections involving β. Now, back to our original problem, we have

$$(((U_1 \mid 0) \cap \ldots \cap (U_n \mid 0) \cap M) \cup \ldots) \backslash ds \subseteq \beta$$

We can discount all the terms represented by '...' since they are all smaller than β by virtue of being intersections involving β. The intersection on the left can be normalised to a single union type, and the problem is solved.

To recap, the entailment

$$C \Vdash \alpha \subseteq P_1 \mid \ldots \mid P_n \mid \beta^{cs}$$

holds, if and only if

$$C \Vdash \text{upper}(\alpha, C) \subseteq P_1 \mid \ldots \mid P_n \mid 1^{cs}$$

and

$$C \Vdash \text{upper}(\alpha, \beta, C) \backslash ds \subseteq \beta$$

where $\text{upper}(\alpha, \beta, C)$ is the absolute upper bound of α with respect to β in the constraint set C, and ds are the tags of $P_1 \ldots P_n$.

7.3 Algorithm to determine entailment

The following algorithm determines whether the entailment relation

$$\forall\overline{\beta}.\exists\overline{\alpha}.D \Vdash U \subseteq V, C$$

holds. We express the algorithm in an imperative manner, using global variables C', D' and Q, where C' is an evolving constraint set initialised from the inductive reduced form of C, D' is an evolving constraint set initialised from the inductive reduced form of D, and Q is a queue of inequations left to prove.

The general strategy used is to attempt to prove that each constraint on the right of the relation is implied by D. The proof process generates new constraints which must also be shown to be implied by D, along with any constraints which are required for the transitive closure of the constraint set on the right.

1. Let $D' = \text{induct}(\text{reduce}(D))$. If this fails, then the entailment is trivially satisfied since D has no solutions.
2. Compute lower and upper bounds for D'.
3. Initialise $C' = \text{induct}(\text{reduce}(\{U \subseteq V\} \cup C))$. If the reduction fails, then fail.
4. Initialise queue $Q = C'$.
5. Remove a constraint from Q, and analyse it using the following table, where 'continue' means repeat step 5 until Q is empty:

$$\begin{array}{ll}
\alpha \subseteq U & \text{continue} \\
U \subseteq \alpha & \text{continue} \\
U \subseteq \beta & \text{let}(\overline{\beta}, V) = \text{lowers}(\beta, D') \\
 & \quad \text{if } U \in (V \cup \{\overline{\beta}\}),\ \text{continue} \\
 & \quad \text{else new}(V \subseteq U),\ \text{continue} \\
\beta \subseteq U & \text{analyse } U : \\
 & \quad P_1 \mid \ldots \mid P_n \mid \beta'^{cs} \\
 & \qquad V = \text{upper}(\beta, D') \\
 & \qquad new(V \subseteq P_1 \mid \ldots \mid P_n \mid 1^{cs}) \\
 & \qquad V = \text{upper}(\beta, \beta', D') \\
 & \qquad \text{new}(V \subseteq \beta),\ \text{continue} \\
 & \quad \text{otherwise} \\
 & \qquad V = \text{upper}(\beta, D') \\
 & \qquad \text{new}(V \subseteq U),\ \text{continue}
\end{array}$$

where

$$\begin{array}{lll}
\text{new}(U \subseteq V) = & F & = \text{induct}(\text{reduce}(U \subseteq V)) \\
 & F' & = F - F \cap C' \\
 & T & = \text{induct}(\text{reduce}(\text{trans}(F', C'))) \\
 & T' & = T - T \cap C' \\
 & C' & := C' \cup F' \cup T' \\
 & Q & := Q +\!\!+ F' +\!\!+ T'
\end{array}$$

where $\text{trans}(C, D)$ represents the set of constraints required to retain transitive closure of the set D when the constraints from set C are added. The operator $+\!\!+$ stands for list append.

If the algorithm completes, then the specified entailment relation holds. If any of the reduce operations fail, the the entailment relation is false.

8 Extensions

8.1 First-class Functions

Plans are afoot to extend Erlang to include first-class functions, by including lambda expressions and application in the syntax. Our type inference system extends in a straightforward way to include function types, as follows. Add a new prime type $(U_1, \ldots, U_n) \to V$ and a new tag $\to$ to the type syntax, and add lambda expressions $\lambda(U_1, \ldots, U_n) \to V$, application $E(E_1, \ldots, E_n)$, and references to top-level functions f/n (where n is the arity of f) to the expression syntax. Add the following rules to the type system

$$\text{Lam}\ \frac{F;\ A, \overline{X} : \overline{U};\ C \vdash E : V}{F;\ A;\ C \vdash (\lambda(\overline{X}) \to E) : (\overline{U}) \to V}$$

$$\text{App}\ \frac{F;\ A;\ C \vdash E : (\overline{U}) \to V \qquad F;\ A;\ C \vdash \overline{E} : \overline{U}}{F;\ A;\ C \vdash E(\overline{E}) : V}$$

The reduction algorithm can be extended to function types with the addition of two rules:

$$\begin{array}{lll}
(\overline{U}) \to V \subseteq ((\overline{U'}) \to V') \mid U & \Rightarrow & \overline{U'} \subseteq \overline{U}, V \subseteq V' \\
(\overline{U}) \to V \subseteq c\{\overline{U'}\} \mid U & \Rightarrow & (\overline{U}) \to V
\end{array}$$

Our algorithm for determining entailment can be extended to support function types, but the resulting algorithm is known to be incomplete [TS96]. Decidability of entailment in the presence of function types is not known.

8.2 Interprocess Communication

One of the most important features of Erlang is its support for concurrency and transparent distribution. With some straightforward extensions, we can extend our type system to check the types of messages passing between processes. Erlang provides primitives for sending and receiving messages, and spawning new processes.

To type check message passing, it is necessary to keep track of the types of messages received by a given expression. Therefore, we propose extending the typing rules to provide two types for an expression: the type of values it returns, and the type of messages it accepts. Typing judgements now have the form $F; X; C \vdash E : U$ **receives** R, where R is the type of messages received by the expression. Similarly,

function types should be written $(\overline{U}) \xrightarrow{R} V$, where R is the type of messages received by the function body.

In addition, we need to provide a new primitive data type, of the form $\mathbf{pid}(U)$ which is the type of a process that accepts messages of type U. We can then support type checking of message sending with a primitive

$$send : (\mathbf{pid}(U), V) \rightarrow 0 \textbf{ when } V \subseteq U$$

and process spawning with a primitive

$$spawn : ((\overline{U}) \xrightarrow{R} V, \overline{U}) \rightarrow \mathbf{pid}(R)$$

This extension is a special case of effect type systems [TJ94]. It would allow the construction of polymorphic server applications, but it doesn't support checking of protocols or detection of possible deadlocks.

9 Practical Experience

The original goal of designing a type system for Erlang was for the resulting system to be usable in a production environment by Erlang programmers.

We have so far constructed a proof-of-concept prototype implementation in Haskell. While lacking in performance in certain areas, the prototype has provided valuable insight into how our type system will co-exist with the Erlang environment and what changes are required to typecheck existing Erlang code.

The prototype supports the following features:

- Type inference for the whole Erlang 4.4 language.
- Type simplification for derived types.
- Type signature checking.
- Type abbreviations for use in type signatures (for example, see the `tree` datatype in Figure 1).
- Separate compilation through the use of interface files which record the types of exported functions in a module.
- Type abstraction. By default, type abbreviations are exported abstractly, so the definition of a type is hidden from external modules. The programmer may optionally request that certain type definitions be exported in full. The type checker detects abstraction violations and reports these to the programmer.
- Partial type checking. On a function-by-function basis, the programmer may provide types that the type system will assume without type checking the function definition. This is implemented with an `unchecked` directive in the code.

This prototype has been used to typecheck a large portion of the Erlang standard library, and we are currently using it to construct a production version of the type checker in Erlang. The production version consists of 3500 lines of typed Erlang.

9.1 Performance

We have found performance of the type inference engine to be adequate in most cases, although due to the quadratic complexity of constraint reduction it can blow up on large constraint sets. Programs that cause most problems are those involving large groups of mutually recursive functions, which must be typechecked together. It is not unusual to find types with upwards of ten thousand constraints.

Performance of the type simplification process is poor for large examples. This is due to two factors: simplification usually has to be performed several times before a fixed-point is reached, and our prototype implementation uses algorithms with worse complexity than is achievable for some of the simplification transformations. We expect performance of the simplifier to improve with the production version of the type checker.

There is a tradeoff involving type simplification and type checking: if the user supplies a type signature, should the inferred type be simplified before checking against the signature, or should the original inferred type be used? If the type is simplified, type checking should be faster, but we would pay for simplification time instead. By experimentation we have found that type checking is a relatively cheap operation compared to simplification, so we opt not to simplify types if the user has supplied a signature.

In practice, we have found it convenient to use type signatures almost everywhere, for two reasons:

- supplying a type signature avoids the poor performance of the type simplifier, and
- type signatures may use abbreviations making them more readable than inferred types.

To provide some concrete numbers, here are some of the timings for typechecking some modules from the production version of the type checker:

Module	Lines	Time(s)
`tc_types.terl`	526	2.7
`tc_typeutils.terl`	274	5.1
`tc_reduce.terl`	288	101.2
`tc_syn.terl`	250	60.0

These times were produced by the prototype type checker (written in Haskell) on a 200Mhz Pentium Pro. The module `tc_types.terl` contains a large number of small unrelated functions and type definitions, whereas the module `tc_reduce.terl` contains a small number of complicated mutually recursive functions, hence the large typechecking time.

9.2 Diagnostics

The usability of a type system is directly affected by the quality of error messages generated for untypable expressions. It is possible to generate type errors which include function and line number information in our type system using the following technique: instead of generating the whole constrained type for a function before reducing it, we can reduce constraints as they are generated and maintain a fully reduced constraint set.

Using this method it is possible to identify the program construct that generated the constraint that led to an inconsistent typing. This technique is not perfect: for example,

it is not possible to identify whether a type error is caused by an error in a function definition or in an application of that function. In our system the error is always reported at the application site.

Errors in type signatures are another matter: when a signature is found not to be an instance of the inferred type, we simply report the inferred type and the constraint that failed during the entailment algorithm. In most cases, this information is insufficient to identify the cause of the error, and matters are worse when the inferred type is large and complicated. We intend to explore this problem as future work.

9.3 Pattern Matching

In order to typecheck the full Erlang language, we must compile the pattern matching into simple case expressions. Standard algorithms exist to perform this transformation [Aug85, Wad87]. In most cases, the use of pattern matching compilation in our type checker is transparent to the programmer, but there are cases where unexpected types are derived.

For example, one possible way to write the boolean `and` function is as follows:

```
and(true,true) -> true;
and(false,X)   -> false;
and(X,false)   -> false.
```

Given the type declaration `bool() = true | false`, we would expect to be able to assign the type

```
-type and(bool(),bool()) -> bool().
```

as an instance of the inferred type. However, in our system this is not the case. Pattern matching compilation transforms the function as follows:

```
and(X,Y) ->
   let Z = (case Y of false -> false end) in
   case X of
      true  ->
         case Y of
            true -> true;
            X -> Z
         end;
      false -> false;
      X -> Z
  end.
```

which yields the type `(1, false) -> false | true`. The second argument is restricted to being `false`, due to the first case expression on `Y` in the transformed code. The type is unexpected, since it does not have the type `(bool(),bool()) -> bool()` as an instance.

This example displays a difference between our system and Hindley-Milner, since in that system the declaration of `bool()` would enable the type system to derive the expected type. Our system does not require a type declaration and assumes the worst, namely that in order to ensure that the function can never fail it is necessary to restrict the second argument to being `false` only.

A generalisation of our type system, such as conditional types [AWL94], would be necessary to derive a more accurate type which would have the type `(bool(),bool()) -> bool()` as an instance. However, such an extension would also increase the complexity of type checking, and decrease the readability of types.

9.4 Changes to existing code

We were pleasantly surprised that very little existing code needed to be changed to get through the typechecker. The changes we have had to make fall into the following categories:

- Actual programming errors, or (more commonly) cases where the original programmer had deliberately left out a failure case. Our type system guarantees that typechecked code will never fail except where the programmer has used an explicit `exit` call. This means that all failure cases must be explicitly checked for in typed code.
- Clashes between tagged tuples and anonymous tuples. In our type system, it is not the case that (for instance) `{a,1} <= {1,1}`, although some Erlang code assumes this.
- Pattern matching oddities, as described in Section 9.3).

10 Conclusion

While much has been achieved in our Typed Erlang project, several areas remain to be explored. Our principle achievements to date are the type system itself, several heuristics for simplifying inferred types, the entailment algorithm for determining type instance, and our experience with a prototype implementation which supports many of the features one would expect from a typed programming language.

Several gaps remain in the theory; most notably is a proof of completeness for our entailment algorithm (without function types). Once this proof is complete we intend to use the entailment algorithm to prove the correctness of our type simplifications.

The production version of our type checker is currently under construction, and will be distributed along with a future version of the Erlang system. We expect the final version to improve on the prototype in areas of performance, robustness and the quality of diagnostics.

References

[Aug85] L. Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computing Science, pages 368 – 381, Nancy, September 1985. Springer-Verlag.

[AVW93] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming with Erlang*. Prentice Hall, 1993.

[AW93] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[AWL94] A. Aiken, E.L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Symposium on Principles of Programming Languages*, 1982.

[FA96] M. Fähndrich and A. Aiken. Making set-constraint-based program analyses scale. Technical Report CSD-96-917, UC Berkeley, 1996. Also *Workshop on Set Constraints*, Cambridge, MA, August 1996.

[FF96] C. Flanagan and M. Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR96-266, Rice University, 1996.

[FM88] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In *European Symposium on Programming*, 1988.

[Hin79] R. Hindley. The principal type scheme of am object in combinatory logic. *Transactions of the American Mathematics Society*, 146:26–60, 1979.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Mit91] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[MPS86] D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Information and Control*, volume 71, pages 95–130, 1986.

[MR85] P. Mishra and U. Reddy. Declaration-free type checking. In *Symposium on Principles of Programming Languages*, pages 7–21, 1985.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Rey85] J. C. Reynolds. Three approaches to type structure. In *Proc. TAPSOFT Advanced Seminar on the Role of Semantics in Software Development*, Lecture Notes in Computing Science. Springer-Verlag, 1985.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[TS96] V. Trifonov and S. Smith. Subtyping constrained types. In *Third International Static Analysis Symposium*, September 1996. To Appear.

[Wad87] P. Wadler. Efficient compilation of pattern-matching. In [Pey87], 1987.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987.

[WC94] A. K. Wright and R. Cartwright. A practical soft type system for scheme. In *ACM Symposium on Lisp and Functional Programming*, 1994.