Type Inclusion Constraints and Type Inference

Alexander Aiken
IBM Almaden Research Center
650 Harry Rd., San Jose, CA 95120
aiken@almaden.ibm.com

Edward L. Wimmers
IBM Almaden Research Center
650 Harry Rd., San Jose, CA 95120
wimmers@almaden.ibm.com

November 16, 1993

Abstract

We present a general algorithm for solving systems of inclusion constraints over type expressions. The constraint language includes function types, constructor types, and liberal intersection and union types. We illustrate the application of our constraint solving algorithm with a type inference system for the lambda calculus with constants. In this system, every pure lambda term has a (computable) type and every term typable in the Hindley/Milner system has all of its Hindley/Milner types. Thus, the inference system is an extension of the Hindley/Milner system that can type a very large set of lambda terms.

1 Introduction

Type inference systems for functional languages are based on solving systems of type constraints. The best known and most widely used type inference algorithm was first discovered by Hindley and later independently by Milner [Hin69, Mil78]. In its simplest form, the algorithm generates type equations from the program text and then solves the equations. If the equations have a solution, the program is well-typed—it cannot "go wrong" and produce a type error when executed. If the constraints do not have a solution, the program is considered to be ill-typed—it might produce a run-time type error.

Many generalizations of the Hindley/Milner algorithm have been proposed; see [CW85, Rey85] for surveys of the major research directions. One approach that has received considerable attention is relaxing the form of the type constraints from equations $X = Y$ to inclusions $X \subseteq Y$ [Mit84, Rey85, FM88, Tha88, KPS92]. In inclusion-based type systems, deciding whether a program has a type is reduced to the question of whether a system of inclusion constraints has a solution. So far as we know, however, there have been no general results on the problem of solving systems of type inclusion constraints. For this reason, proposed algorithms for type inference based on solving inclusion constraints are quite restrictive (see Section 8).

The main contribution of this paper is a general algorithm for solving systems of type inclusion constraints. The type language we consider includes a least type 0, a universal type 1, intersection and union types, function types, constructor types such as pairs, and recursive types. Our algorithm cannot solve arbitrary systems of inclusion constraints; we find it necessary to restrict the use of intersection and union types to obtain an effective algorithm.

The work we present is based on earlier work in solving systems of *set constraints*, which are inclusion constraints over sets of terms of a free algebra [AW92a]. The essential difference between set constraints and the constraints we discuss here is the addition of function types. With function types come all the difficulties inherent in reasoning about sets of (possibly partially defined) functions, so the constraint theory is substantially different and more difficult than that in [AW92a], although the same basic techniques apply.

To help motivate and illustrate the potential of type inference systems based on our algorithm for solving type inclusion constraints, in Section 3 we introduce a simple inclusion-based type inference system for the lambda calculus with constants. This inference system has two interesting properties: first, every pure lambda term has a (computable) type, and second, every term has its Hindley/Milner type (if it exists). We also present several examples taken from our implementation of the type inference system. These examples show that inclusion constraints can be used to infer very accurate types for programs that usually are considered untypable.

Our algorithm for solving type inclusion constraints works by incrementally transforming a system of constraints until the system is discovered to be inconsistent (i.e., has no solutions) or until the system is *inductive* (see Definition 5.2). The algorithm itself consists of two relatively simple steps. First, all constraints $X \subseteq Y$ are simplified to constraints on variables of the form $\alpha \subseteq Z$ or $Z \subseteq \alpha$. This step is essentially a large case analysis on the form of X and Y. Second, the system is closed under transitive constraints: if $X \subseteq \alpha$ and $\alpha \subseteq Y$, then $X \subseteq Y$ is added to the system. These two steps are repeated until no new constraints on variables can be added to the system.

There are two difficulties that must be overcome. The first is to prove that if the algorithm does not detect an inconsistency, then the constraints in fact have a solution. Since the algorithm terminates whenever the system is transformed into an inductive system or when an inconsistency is detected, it suffices to show that inductive systems always have solutions. An inductive system (defined precisely in Definition 5.2) is a system of constraints with one lower and upper bound per variable $L_i \subseteq \alpha_i \subseteq U_i$. Using techniques developed in [AW92a], we show that inductive systems can be transformed to a set of equations $\alpha_i = L_i \cup (\beta_i \cap U_i)$ where the β_i are fresh variables. Intuitively, β_i is a parameter that allows α_i to be anything "in between" its lower and upper bounds. The advantage of converting from containments to equations is that type equations are well understood; we apply a known result to prove that for every choice for the β_i the equations have a unique solution [MPS84].

The second difficulty is determining the rules for decomposing arbitrary constraints into constraints on variables. In many cases the rules are obvious; for example, the constraint $X \cup Y \subseteq Z$ holds iff $X \subseteq Z$ and $Y \subseteq Z$. Most other cases of $X \subseteq Y$ also simplify into constraints on subexpressions of X and Y. However, constraints of the form $X \subseteq Y \cup Z$ as well as of the form $X \cap Y \subseteq Z$ are problematic. For example, the constraint $X \subseteq Y \cup Z$ is difficult to decompose because the constraint can be satisfied even if X is not a subset of either Y or Z. In simple set theory, $X \subseteq Y \cup Z$ iff $X \cap \neg Y \subseteq Z$, where $\neg Y$ is the set complement of Y. Unfortunately, as we discuss in Section 6, the simple set-theoretic definition of $\neg Y$ is not a type. We introduce a weaker definition of $\neg Y$ that is a type and show how it can be used to simplify constraints $X \subseteq Y \cup Z$ provided $Y \cap Z = 0$ (i.e., where Y and Z are disjoint). Constraints of the form $X \cap Y \subseteq Z$ can be simplified if Y is restricted to be *upward closed* and a *monotype* (see Section 6).

Using the results of Sections 5 and 6, Section 7 introduces the class of *proper* constraint systems and presents an algorithm for solving proper systems. Proper systems have very liberal (but not unrestricted) union and intersection types. Section 8 compares our work with some additional related work; Section 9 concludes with a discussion of current and future work.

2 Lambda Calculus and Types

The definitions in this section are either standard or minor variations on standard definitions; the reader familiar with semantic models of types [MPS84] may skip to Section 3 and use this section only for reference.

Our programming language is the strict lambda calculus with a finite set of strict constructors C. Each $c \in C$ has a fixed arity; c may be a nullary constructor (a constant). The expressions of this language are

$$e ::= x \mid \lambda x.e_1 \mid e_1\ e_2 \mid c(e_1, \ldots, e_n)$$

which are respectively variables, function abstractions, function applications, and data constructions.

We essentially adopt the *ideal* model of types, in which types are certain subsets of the semantic domain [MPS84]. For the untyped, strict lambda calculus with constructors the semantic domain is given by the equation

$$D = \{\bot\} \cup (D \rightarrow D) \cup \bigcup_{c \in C} c(D - \{\bot\}, \ldots, D - \{\bot\}) \cup \mathit{wrong}$$

We use the standard call-by-value semantic function μ that assigns elements of D to every expression e. The values $\bot$ and $\lambda x.\ \bot$ are not equal; the first denotes a divergent computation, while the second denotes a function that diverges when applied. The value *wrong* denotes a run-time error, which in this small language results whenever a data structure $c(\ldots)$ is applied as a function.

The domain D is constructed from the limit $\mathbf{D}_\omega$ of a series of finite sets of *finite elements* $\mathbf{D}_0 \subseteq \mathbf{D}_1 \subseteq \ldots$ where $\mathbf{D}_0 = \{\bot\}$ and

$$\begin{array}{rcl} \mathbf{D}_{i+1} & = & \mathbf{D}_i \cup (\mathbf{D}_i \rightarrow_{\mathrm{M}} \mathbf{D}_i) \cup \\ & & \bigcup_{c \in C} c(\mathbf{D}_i - \{\bot\}, \ldots, \mathbf{D}_i - \{\bot\}) \cup \{\mathit{wrong}\} \end{array}$$

where $\mathbf{D}_i \rightarrow_{\mathrm{M}} \mathbf{D}_i$ is the set of finite, strict, monotonic functions from $\mathbf{D}_i$ to $\mathbf{D}_i$.[1] We use the standard partial order $\leq$ on the elements of D.[2] A set $D' \subseteq D$ is *downward-closed* iff $x \in D$ and $y \leq x$ implies that $y \in D$. Types are certain downward-closed subsets of D. By convention, *wrong* and values containing *wrong* should not have a type; this makes it easy to prove that typable terms cannot make runtime errors. As an aid to formally defining types, we introduce a subset T of D that does not contain *wrong*:

$$T = \{\bot\} \cup (T \rightarrow T) \cup \bigcup_{c \in C} c(T - \{\bot\}, \ldots, T - \{\bot\})$$

Definition 2.1 A *type* is a non-empty, downward-closed set of finite elements that is a subset of T.

[1]Normally continuous functions are used; in this case monotonic functions are sufficient because each D_i is a finite set and for finite functions monotonicity implies continuity.

[2]$\bot \leq x$ for all $x \in D$;
for $f, g \in D \rightarrow D, f \leq g$ iff $\forall x\ f(x) \leq g(x)$;
and $c(x_1, \ldots, x_n) \leq c(y_1, \ldots, y_n)$ iff $x_i \leq y_i$ for $i = 1, \ldots, n$

This definition of type is essentially equivalent to the usual definition based on ideals [MPS84] because every ideal is isomorphic with a downward-closed set of finite elements. We choose to work with the finite elements both because it is simpler and because we make direct use of induction on the finite elements in proofs. As an aside, Definition 2.1 can be simplified by letting a type be any downward-closed set of finite elements; here we have followed standard practice and excluded *wrong* from types.

An expression has a given type iff the set of finite approximations to the meaning of the expression is a subset of the type. More formally,

Definition 2.2 Let e be an expression. Then e has type τ, written $e : \tau$, iff

$$\{v \in \mathbf{D}_\omega \mid v \leq \mu(e)\} \subseteq \tau$$

Types are ordered by set containment. The least type is the set $\{\bot\}$ which is denoted by 0.

The complete language for type expressions is

$$\tau ::= \tau_1 \cup \tau_2 \mid \tau_1 \cap \tau_2 \mid \tau_1 \to \tau_2 \mid c(\tau_1, \ldots, \tau_n) \mid \alpha \mid 0$$

where α denotes a type variable. It is possible to express a universal type 1 (which has every finite element of T; see above) and those instances of type complement $\neg X$ that we need for solving constraints in terms of more primitive operations; thus, we do not include 1 and $\neg X$ among the primitive operations on types. To give semantics to type expressions, any substitution σ mapping variables to types is extended to type expressions as follows:

$$\begin{array}{rcl}
\sigma(0) & = & \{\bot\} \\
\sigma(X \cap Y) & = & \sigma(X) \cap \sigma(Y) \\
\sigma(X \cup Y) & = & \sigma(X) \cup \sigma(Y) \\
\sigma(c(X_1, \ldots, X_n)) & = & \\
 & & \{c(t_1, \ldots, t_n) | t_i \in \sigma(X_i) - \{\bot\}\} \cup \{\bot\} \\
\sigma(X \to Y) & = & \\
\multicolumn{3}{c}{\bigcup_i \{f \in \mathbf{D}_i \to_{\mathrm{M}} \mathbf{D}_i \mid f(\sigma(X) \cap \mathbf{D}_i) \subseteq \sigma(Y)\} \cup \{\bot\}}
\end{array}$$

It is easy to check that $\sigma(X)$ is a downward-closed set.

A system of constraints has the form $\{X_i \subseteq Y_i\}$ where the X_i and Y_i are type expressions. A *solution* of the constraints is a substitution σ such that $\sigma(X_i) \subseteq \sigma(Y_i)$. The set of all solutions of a system S of constraints is written $\mathcal{S}(S)$.

We conclude this section with two examples. Let $X = Y$ stand for the pair of constraints $X \subseteq Y$ and $Y \subseteq X$. Consider a binary constructor *cons* and a nullary constructor *nil*. The equation $\alpha = \mathit{cons}(\beta, \alpha) \cup \mathit{nil}$ defines α to be any list with elements of type β. This example shows that an explicit fixed point operator is not needed in the type language, because a fixed point can be defined by constraints. The next example shows that the universal type 1 can be defined as the unique solution of the equation:

$$\alpha = (0 \to \alpha) \cup \bigcup_{c \in C} c(\alpha, \ldots, \alpha)$$

The first disjunct contains all functions (in a strict language) and the second disjunct closes the set under all constructions. Thus, this equation has a unique solution where α is the set of all finite elements not involving *wrong*.

$$\frac{}{A \cup \{x : \alpha\}, S \vdash x : \alpha} \qquad \frac{A, S \vdash e_i : \tau_i \quad 1 \leq i \leq n}{A, S \cup \{c(\tau_1, \ldots, \tau_n) = \alpha\} \vdash c(e_1, \ldots, e_n) : \alpha}$$

$$\frac{A \cup \{x : \tau_1\}, S \vdash e : \tau_2}{A, S \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{A, S \vdash e_1 : \tau_1, \; e_2 : \tau_2}{A, S \cup \{\tau_2 = \alpha, \; \tau_1 = \alpha \to \beta\} \vdash e_1 \, e_2 : \beta}$$

Figure 1: Hindley/Milner type inference using equality constraints.

$$\frac{}{A \cup \{x : \alpha\}, S \vdash x : \alpha} \qquad \frac{A, S \vdash e_i : \tau_i \quad 1 \leq i \leq n}{A, S \cup \{c(\tau_1, \ldots, \tau_n) \subseteq \alpha\} \vdash c(e_1, \ldots, e_n) : \alpha}$$

$$\frac{A \cup \{x : \tau_1\}, S \vdash e : \tau_2}{A, S \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{A, S \vdash e_1 : \tau_1, \; e_2 : \tau_2}{A, S \cup \{\tau_2 \subseteq \alpha, \; \tau_1 \subseteq \alpha \to \beta\} \vdash e_1 \, e_2 : \beta}$$

Figure 2: Type inference using inclusion constraints.

3 Type Inference

In this section we present a simple type inference system based on inclusion constraints and compare it with the Hindley/Milner system. The purpose of this is to illustrate the potential of type inference systems based on solving systems of inclusion constraints; our algorithm for solving inclusion constraints is presented in Sections 5-7.

A set of rules for Hindley/Milner type inference is given in Figure 1. These rules are presented in a non-standard form; following Wand [Wan87], a set of type constraints S is associated with each inference rule. A conclusion $A, S \vdash e : \tau$ holds for all solutions of S. This proof system is deterministic—there is only one type derivation for any lambda term, up to renaming of type variables. It is well-known that the solutions of the constraints S can be computed using unification [Rob65].

Figure 2 gives inference rules for an inference system based on type inclusion. The only changes are to replace equality constraints by containments. In the construction rule $c(\tau_1, \ldots, \tau_n) = \alpha$ is replaced by $c(\tau_1, \ldots, \tau_n) \subseteq \alpha$, and in the application rule $\tau_2 = \alpha$ is replaced by $\tau_2 \subseteq \alpha$ and $\tau_1 = \alpha \to \beta$ is replaced by $\tau_1 \subseteq \alpha \to \beta$. It is easy to see that the rules in Figure 2 are sound (i.e., they generate only valid typings). More interestingly, well-typed terms cannot "go wrong".

Proposition 3.1 If $\emptyset, S \vdash e : \tau$ and constraints S have a solution, then $\mu(e) \neq$ *wrong.*

Proof: Follows from soundness of the rules and the fact that *wrong* is not a member of any type. $\Box$

The inclusion system has two additional properties. First, every solution of the Hindley/Milner constraints for a term e is also a solution of the inclusion constraints for e. This follows from the fact that equality solutions are also solutions of the inclusions. The second property is that every pure term (i.e., a term with no constructors) is typable in the inclusion system, because in any type derivation for a pure term the constraints always have a solution. To see this, set all variables in the constraints to

α_0 where α_0 is the unique type such that $\alpha_0 = \alpha_0 \rightarrow \alpha_0$; since a pure term has no data constructors every term has type α_0 and every constraint is satisfied. Thus, the inclusion system is an extension of the Hindley/Milner system that can type a very large class of lambda terms. Of course, to determine whether a term is typable or not it is necessary to decide whether the constraints have a solution. In subsequent sections we give an effective algorithm for solving a general class of type constraints such as the ones found in Figure 2.

3.1 Let-Polymorphism

One important component of the Hindley/Milner type system that we have not yet discussed is *let-polymorphism* [Mil78]. A full discussion of polymorphism is beyond the scope of this paper; in this short section we very briefly show how let-polymorphism is incorporated into our system. In the end, except for the non-standard presentation using constraints, the inference rules are the same as in the Hindley/Milner system.

We add a new construct to the language "let $x = e$ in e'" with the usual semantics. The point of let-polymorphism is that distinct occurrences of x in e' can be typed independently. This cannot be done using the inference rules presented so far, because there can be only one assumption about the type of x and every instance of x is assigned this same type. To overcome this problem we introduce a type scheme $\forall\alpha.\tau$ *where* S, which is the universal quantification over a type expression τ and its associated constraints S. Given a substitution σ, the semantics of a quantified type is an intersection over all solutions of the constraints:

$$\sigma(\forall\alpha.\tau \textit{ where } S) = \bigcap_{\pi \in X} \pi(\tau)$$

where $X = \mathcal{S}(S) \cap \{\sigma' | \sigma'(\beta) = \sigma(\beta) \text{ if } \beta \neq \alpha\}$.

The next step is to add inference rules for quantifier introduction and elimination. These rules are just the normal Hindley/Milner generalization and instantiation rules recast using constraints. To introduce a universal quantifier, the constraints must have a solution (i.e., the term must have a type without quantification) and there must be no assumptions about the quantified variable:

$$\frac{A, S \vdash e : \tau \text{ and } \mathcal{S}(S) \neq \emptyset \text{ and } \alpha \text{ not free in } A}{A, \emptyset \vdash e : \forall\alpha.\tau \textit{ where } S}$$

To eliminate a universal quantifier, we simply drop the quantifier and substitute a type expression for the quantified variable:

$$\frac{A, S \vdash e : \forall\alpha.\tau \textit{ where } S'}{A, S \cup S'[\tau'/\alpha] \vdash e : \tau[\tau'/\alpha]}$$

Finally, the inference rule for "let" is the Hindley/Milner rule extended with constraints:

$$\frac{A, S \vdash e : \tau \quad A \cup \{x : \tau\}, S \vdash e' : \tau'}{A, S \vdash \text{ let } x = e \text{ in } e' : \tau'}$$

With these additional inference rules, each occurrence of x in e' can be typed using a different instantiation of a quantified type for x.

3.2 Examples

We conclude this section with some examples taken from our implementation of the inclusion constraint system. Our system infers quantified types for terms. In general, if a quantified type $\forall\alpha.\tau$ *where* S is monotonic (resp. anti-monotonic) in α, then α can be eliminated without changing the meaning of the type by setting α to the lower bound (resp. upper bound) implied by the constraints S. Our implementation performs these optimizations (as well as others) to make quantified types more readable. The first example is the identity function:

```
I = \X.X : forall (a). a -> a
```

Our system prints quantified types as "`forall`" with an explicit list of quantified variables. The type in this case is the same as the Hindley/Milner type. The type for the functional `K` that forms constant functions is also equivalent to the Hindley/Milner type, but differs in appearance:

```
K = \X.\Y.X : forall (a).a -> 1 -> a
```

The Hindley/Milner type is $\alpha \rightarrow \beta \rightarrow \alpha$; since the type is anti-monotonic in β, it is instantiated to its upper bound 1 in our system. For the next function, the inclusion system infers a more accurate type than the Hindley/Milner system:

```
twice = \F.\X.(F (F X)) :
forall (a,b,c).((b -> a) & (a -> c)) -> b -> c
```

The symbol "`&`" stands for type intersection. The function `twice` has type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ in the Hindley/Milner system which loses the distinction between a, b, and c.

The fixed point combinator is not typable in the Hindley/Milner system but (like every pure term) is typable in the inclusion constraint system:

```
Y = \U.(\X.(U (X X)) \X.(U (X X))) :
forall (a,b).((a -> b) & (a -> a)) -> b
```

The type for `Y` looks a little curious, but by letting $a = b = c$ it is easy to see that one instance of it is $(c \rightarrow c) \rightarrow c$, so it is as accurate as this more conventional signature. The inference system is also able to do a good job with applications of `Y`:

```
Y (twice I) : 0
```

Our system proves that `Y twice I` is a non-terminating expression. An even fancier example is `Y K`:

```
Y K : 1 -> b where b = 1 -> b
```

In this example some constraints remain after variables are eliminated. The type b is the unique solution of the recursive equation $b = 1 \rightarrow b$. It is noteworthy that all the rules employed in the actual derivation of the above examples are valid for lazy functions as well as strict functions, so all the above types are valid for lazy systems as well. In fact, the type inferred above is the best possible type for `Y K` in a lazy system. Intuitively, this is because `Y K` has the property that `Y K x = Y K` for all `x` and this functionality is exactly captured by the type b.

Finally, expressions that apply constants as functions are ill-typed because the constraints have no solutions:

```
true true
*type error*
```

4 Sample Run of the Algorithm

Before giving a formal specification of the algorithm, we present a sample run. The algorithm is presented in the following sections and the rules are given in Figures 3, 4, and 5 in Appendix A. Consider the function *SWAP* that changes the unary constructor a (assuming it occurs at the outermost level) into the unary constructor b and vice versa. *SWAP* highlights the role that unions and intersections play in the algorithm. Since
$SWAP : \forall(\alpha,\beta).a(\alpha) \cup b(\beta) \rightarrow b(\alpha) \cup a(\beta)$,
the rules given in Figure 2 imply that
$\lambda x.SWAP(a(x)) : \forall(\alpha_1,\alpha_2,\alpha_3,\alpha_4,\alpha_5).\quad ($
$\alpha_1 \rightarrow \alpha_5$ where
$a(\alpha_2) \cup b(\alpha_3) \rightarrow b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_4 \rightarrow \alpha_5$
$a(\alpha_1) \subseteq \alpha_4 \quad)$

In order to obtain an inductive system (see Definition 5.2), we apply the algorithm to the system S1 of constraints:

S1: $a(\alpha_2) \cup b(\alpha_3) \rightarrow b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_4 \rightarrow \alpha_5$
$a(\alpha_1) \subseteq \alpha_4$

System S2 is obtained by using the fact that function types are anti-monotonic in their first argument and monotonic in their second argument. (See Rule 4 of Figure 5.)

S2: $\alpha_4 \subseteq a(\alpha_2) \cup b(\alpha_3)$
$b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_5$
$a(\alpha_1) \subseteq \alpha_4$

By combining constraints on α_4 we obtain:

S3: $b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_5$
$a(\alpha_1) \subseteq \alpha_4 \subseteq a(\alpha_2) \cup b(\alpha_3)$

System S4 is obtained by applying transitivity to the constraints on α_4.

S4: $b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_5$
$a(\alpha_1) \subseteq \alpha_4 \subseteq a(\alpha_2) \cup b(\alpha_3)$
$a(\alpha_1) \subseteq a(\alpha_2) \cup b(\alpha_3)$

Next we eliminate the union on the right-hand side in the last constraint by moving appropriate monotypes to the left-hand side. This step is explained in Section 6, formalized in Rule 8 of Figure 5, and justified by Lemma 6.4.

S5: $b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_5$
$a(\alpha_1) \subseteq \alpha_4 \subseteq a(\alpha_2) \cup b(\alpha_3)$
$a(\alpha_1) \cap \neg a(1) \subseteq b(\alpha_3)$
$a(\alpha_1) \cap \neg b(1) \subseteq a(\alpha_2)$

The left-hand sides of the last two constraints can be simplified using the rules in Figures 3 and 4 and the rule $\alpha_1 \cap 1 = \alpha_1$.

S6: $b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_5$
$a(\alpha_1) \subseteq \alpha_4 \subseteq a(\alpha_2) \cup b(\alpha_3)$
$0 \subseteq b(\alpha_3)$
$a(\alpha_1) \subseteq a(\alpha_2)$

The third constraint is always true and can be dropped; the fourth constraint is simplified by dropping the constructor a. These steps are formalized in Rules 1 and 2 of Figure 5.

S7: $b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_5$
$a(\alpha_1) \subseteq \alpha_4 \subseteq a(\alpha_2) \cup b(\alpha_3)$
$\alpha_1 \subseteq \alpha_2$

At this point, the algorithm terminates because the system is inductive. Thus, we have shown that the type is $\lambda x.SWAP(a(x))$ is $\forall(\alpha_1, ..., \alpha_5).\quad (\alpha_1 \rightarrow \alpha_5$
where $b(\alpha_2) \cup a(\alpha_3) \subseteq \alpha_5$
$a(\alpha_1) \subseteq \alpha_4 \subseteq a(\alpha_2) \cup b(\alpha_3)$
$\alpha_1 \subseteq \alpha_2 \qquad)$

After optimizing the representation of the type, the algorithm reports that the type is $\forall\alpha.\alpha \rightarrow b(\alpha)$. It is easy to check this by hand by letting $\alpha_2 = \alpha_1 = \alpha$, $\alpha_3 = 0$, $\alpha_4 = a(\alpha)$, and $\alpha_5 = b(\alpha)$.

5 Inductive Systems

In the remainder of the paper we present our algorithm for solving systems of type inclusion constraints. As the first step, we show that every *inductive* system of constraints has a solution. We make use of two previous results in the proof. The first is a technique for transforming inclusion constraints to an equivalent system of equations [AW92a]. The second is the fact that systems of *contractive* equations have unique solutions [MPS84]. The constraint-solving algorithm presented in Section 7 reduces an initial system of constraints to a set of systems of inductive constraints or reports that the initial system is inconsistent.

To help motivate the technical definitions that follow, consider the following natural inductive strategy for showing that an arbitrary system of inclusion constraints over variables $\alpha_1, \ldots, \alpha_n$ has a solution. Initially, let $\alpha_j = 0$ for $1 \leq j \leq n$. Recall that the semantic domain is constructed from an increasing sequence of sets of finite elements $\mathbf{D_0}, \mathbf{D_1}, \ldots$ (Section 2). At step i of the induction, assign some finite elements of $\mathbf{D_i}$ to α_1, then to α_2, and so on, up to α_n. At each step (i, j) of this double induction over the finite elements $\mathbf{D_i}$ and variables α_j, we must ensure that the constraints are satisfied for all finite elements in $\mathbf{D_i}$. If this can be done for all pairs (i, j) then the system has a solution.

In such an inductive proof, we must distinguish between variables inside of constructors $c(\alpha)$, which contribute finite elements from $\mathbf{D_{i-1}}$, and variables outside of constructors $\alpha \cap c(\ldots)$, which contribute finite elements from $\mathbf{D_i}$.

Definition 5.1 The *top-level variables* of X (denoted $TLV(X)$) are the variables in X that appear outside of a type constructor. Formally,

$$\begin{array}{rclrcl} TLV(\alpha_i) & = & \{\alpha_i\} & TLV(0) & = & \emptyset \\ TLV(c(\ldots)) & = & \emptyset & TLV(X \rightarrow Y) & = & \emptyset \end{array}$$
$$TLV(X \cup Y) = TLV(X) \cup TLV(Y)$$
$$TLV(X \cap Y) = TLV(X) \cup TLV(Y)$$

Top-level variables are also called the *non-expansive* variables [MPS84]. There is one problem in transferring the intuition given above to the actual proof. Because function types are anti-monotonic in the domain argument, the sets $\mathbf{D_i}$ are not necessarily downward-closed and so are not types. In addition, values containing *wrong* are in the $\mathbf{D_i}$; the $\mathbf{D_i}$ are not types for this reason as well (see Definition 2.1). To ensure that we work with types, we define D_i to be the downward closure of $\mathbf{D_i} \cap \mathbf{T}$; D_i is the smallest type containing $\mathbf{D_i} \cap \mathbf{T}$. The next definition formalizes the idea that a substitution satisfies the constraints "up to level" D_i and variable α_j.

Definition 5.2 A system S of constraints is *inductive* if the following three conditions hold:

1. $S = \{L_i \subseteq \alpha_i \subseteq U_i \mid i = 1, \ldots, n\}$
2. $\mathit{TLV}(L_i) \cup \mathit{TLV}(U_i) \subseteq \{\alpha_1, \ldots, \alpha_{i-1}\}$ for $1 \leq i \leq n$
3. For all $i_0 = 1, \ldots, n$ and integers j, the following holds in all substitutions:

   $(\forall i = 1, \ldots, i_0 - 1\ (L_i \cap D_j \subseteq \alpha_i \cap D_j \subseteq U_i \cap D_j)$ and
   $\forall i = i_0, \ldots, n\ (L_i \cap D_{j-1} \subseteq \alpha_i \cap D_{j-1} \subseteq U_i \cap D_{j-1}))$
   $\Rightarrow L_{i_0} \cap D_j \subseteq U_{i_0} \cap D_j$

Definition 5.2 makes it possible to build solutions inductively at level D_j by assigning values in order to $\alpha_1, \ldots, \alpha_n$ since part 2 ensures that variables are constrained only by lower-numbered variables at the top level and part 3 ensures that α_{i_0} can be given a value between L_{i_0} and U_{i_0}. Part 3 guarantees that the constraints are closed under transitive constraints; systems that are not closed under transitivity (e.g., $1 \subseteq \alpha_1 \subseteq 0$) need not have solutions.

We show that inductive systems have solutions in two steps: first, we show that an inductive system is equivalent to a system of equations; we then show that the equations always have solutions.

Definition 5.3 A set of equations $\{\alpha_1 = E_1, \ldots, \alpha_n = E_n\}$ is *cascading* if $\mathit{TLV}(E_i) \cap \{\alpha_i, \ldots, \alpha_n\} = \emptyset$.

Theorem 5.4 Let $S = \{L_i \subseteq \alpha_i \subseteq U_i\}$ be an inductive system of constraints. Then S is equivalent to the cascading equations $\alpha_i = L_i \cup (\beta_i \cap U_i)$ where the β_i are fresh variables.

Proof: The proof adapts a similar proof for solving systems of set constraints over the Herbrand Universe [AW92a]. Assume that $L_i \subseteq \alpha_i \subseteq U_i$ and let $\beta_i = \alpha_i$. Then

$$\begin{array}{rcll} \alpha_i & = & L_i \cup (\alpha_i \cap U_i) & \text{since } L_i \subseteq \alpha_i \subseteq U_i \\ & = & L_i \cup (\beta_i \cap U_i) & \text{since } \alpha_i = \beta_i \end{array}$$

Thus, every solution of the constraints induces a solution of the equations. For the other direction, assume that $\alpha_i = L_i \cup (\beta_i \cap U_i)$ for some β_i. Clearly, $L_i \subseteq \alpha_i$. To show $\alpha_i \subseteq U_i$, we first show that for all i and j, $\alpha_i \cap D_j \subseteq U_i \cap D_j$. For the sake of obtaining a contradiction, assume $\alpha_i \cap D_j \not\subseteq U_i \cap D_j$ for some i and j. Pick the smallest such pair (j, i) ordered lexicographically. Note $L_k \cap D_l \subseteq \alpha_k \cap D_l \subseteq U_k \cap D_l$

holds if $(k,l) < (j,i)$ by assumption and because $L_k \subseteq a_k$. Since the system is inductive, it follows that $L_i \cap D_j \subseteq U_i \cap D_j$. Therefore

$$\begin{array}{rl} & \alpha_i \cap D_j \\ = & (L_i \cup (\beta_i \cap U_i)) \cap D_j \\ = & (L_i \cap D_j) \cup (\beta_i \cap U_i \cap D_j) \\ \subseteq & U_i \cap D_j \end{array}$$

which contradicts the assumption. Thus for all i,

$$\begin{array}{rll} & \alpha_i \cap D_j \subseteq U_i \cap D_j & \text{for all } j \\ \Rightarrow & \alpha_i \cap D_j \subseteq U_i & \text{for all } j \\ \Rightarrow & \alpha_i \subseteq U_i & \text{since } \bigcup_j D_j = D_\omega \end{array}$$

□

Theorem 5.5 shows that every choice for the β_i induces a unique solution to the cascading equations.

Theorem 5.5 Let $E = \{\alpha_1 = E_1, \ldots, \alpha_n = E_n\}$ be a set of cascading equations and let σ be any substitution for the variables other than the $\{\alpha_1, \ldots, \alpha_n\}$. There is a unique extension σ' of σ that is a solution of the equations.

Proof: [sketch] Variable α_i can be eliminated from the top-level variables of every equation by substituting E_i for α_i in E_{i+1} through E_n. Then the only top-level variables are variables other than the α_i. For any fixed substitution σ for these top-level free variables, the equations become *contractive* (have no top-level variables). Contractive equations have unique solutions [MPS84]. □

6 Type Complement, Union, and Intersection

At the highest level, our strategy for solving systems of type inclusion constraints is to transform an arbitrary system of constraints into an inductive system. This requires decomposing constraints into simple constraints on variables and adding transitive constraints. In this section we focus on how a system of constraints can be reduced to constraints only on variables.

Most constraints $X \subseteq Y$ decompose easily into constraints on subexpressions of X and Y. For example, $X \subseteq Y \cap Z$ iff $X \subseteq Y$ and $X \subseteq Z$, and $X \cup Y \subseteq Z$ iff $X \subseteq Z$ and $Y \subseteq Z$. There are only two difficult cases: an intersection on the left $X \cap Y \subseteq Z$ and a union on the right $X \subseteq Y \cup Z$.

Consider a constraint of the form $X \subseteq Y \cup Z$. How can this be transformed into a "simpler" constraint? One possibility is to use set complement to move sets from one side to the other:

$$X \subseteq Y \cup Z \Leftrightarrow X \cap (1 - Y) \subseteq Z$$

There is, however, a serious problem with this idea. The set $1 - Y$ is not downward-closed and, therefore, is not a type. For example, $1 - (1 \rightarrow 0)$ contains every function except the least function $\lambda x.\ \bot$. The problem is to find a definition of $\neg X$ that is a type. This motivates the following:

Definition 6.1 $\neg X$ is the largest type such that $X \cap \neg X = 0$. More formally, $\neg X$ is the unique type such that for all types Y, $Y \cap X = 0$ iff $Y \subseteq \neg X$.

As an example, for any X and Y the type $\neg(X \to Y)$ is the set of all non-functions $\bigcup_{c \in C} c(1, \ldots, 1)$. (Recall from Section 2 that $\lambda x.\bot \neq \bot$.) Our definition of $\neg X$ is not quite the set complement of X; this forces restrictions on the constraints $X \cap Y \subseteq Z$ and $X \subseteq Y \cup Z$ that can be solved. Consider again constraints of the form $X \subseteq Y \cup Z$. One might hope that $X \subseteq Y \cup Z \Leftrightarrow X \cap \neg Y \subseteq Z$, but unfortunately this is false. (The constraint $0 \to 1 \subseteq (1 \to 0) \cup \mathit{int}$ has no solutions, but $(0 \to 1) \cap \neg(1 \to 0) = 0 \subseteq \mathit{int}$ holds in all substitutions.) The statement is true, however, if Y is *upward-closed*.

Definition 6.2 A type X is *upward-closed* if $X = \mathit{Up}(X)$ where $\mathit{Up}(X) = \{\bot\} \cup \bigcup_{x \in X - \{\bot\}} \{y | y \geq x\}$

Lemma 6.3 Let Y be upward-closed. Then

$$X \subseteq Y \cup Z \Leftrightarrow X \cap \neg Y \subseteq Z$$

To apply Lemma 6.3 to more general constraints, we need a way to transform any type expression X into an upward-closed type. Define $\overline{X}$ to be the smallest upward-closed *monotype* (a type expression with no variables) such that $\sigma(X) \subseteq \overline{X}$ for all substitutions σ. For example, $\overline{\alpha} = 1$ for any variable α, and $\overline{X \to Y} = 0 \to 1$, the set of all functions. Using $\overline{X}$ and $\neg\overline{X}$, we give a method for decomposing constraints of the form $X \subseteq Y \cup Z$ when Y and Z are disjoint.

Lemma 6.4 Let $X \subseteq Y \cup Z$ be a constraint where $\sigma(Y \cap Z) = 0$ for all σ. Then

$$X \subseteq Y \cup Z \;\Leftrightarrow\; X \cap \neg\overline{Y} \subseteq Z \wedge X \cap \neg\overline{Z} \subseteq Y$$

Proof: It is easy to show that $\forall\sigma\ \sigma(Y \cap Z) = 0$ iff $\overline{Y \cap Z} = 0$. Now we reason as follows:

$$\begin{array}{lll} & X \subseteq Y \cup Z & \\ \Rightarrow & X \subseteq \overline{Y} \cup Z & \text{since } Y \subseteq \overline{Y} \\ \Rightarrow & X \cap \neg\overline{Y} \subseteq Z & \text{Lemma 6.3} \end{array}$$

For the other direction we have

$$\begin{array}{lll} & X \cap \neg\overline{Y} \subseteq Z \wedge X \cap \neg\overline{Z} \subseteq Y & \\ \Rightarrow & X \subseteq Z \cup \overline{Y} \wedge X \subseteq \overline{Z} \cup Y & \text{Lemma 6.3} \\ \Rightarrow & X \subseteq (Z \cup \overline{Y}) \cap (\overline{Z} \cup Y) & \\ \Rightarrow & X \subseteq Z \cap \overline{Z} \cup Z \cap Y \cup \overline{Y} \cap \overline{Z} \cup \overline{Y} \cap Y & \\ \Rightarrow & X \subseteq Z \cup Y & \end{array}$$

where the last line follows because $Z \cap Y = \overline{Z} \cap \overline{Y} = 0$, and $Z \subseteq \overline{Z}$, $Y \subseteq \overline{Y}$. □

We restrict unions on the right of constraints to be disjoint; using Lemma 6.4 we can decompose such constraints. For the other problem constraints $X \cap Y \subseteq Z$, we restrict intersection on the left of a constraint to be of the form $X \cap \overline{Y} \subseteq Z$, which is equivalent to $X \subseteq Z \cap \overline{Y} \cup \neg\overline{Y}$. We form $Z \cap \overline{Y}$ on the right of the transformed constraint to guarantee that the union is disjoint.

To finish this section, Figure 3 gives an algorithm for eliminating $\overline{X}$ and $\neg\overline{X}$ from types. Complement is used only in expressions of the form $\neg\overline{X}$, so it is convenient to define the elimination of $\overline{X}$ and $\neg\overline{X}$ simultaneously. The equivalences in Figure 3 guarantee that all unions are from disjoint sets and all intersections are with upward-closed monotypes.

7 Solving Systems of Constraints

This section defines a class of inclusion constraints, gives rules for reducing those constraints to simple constraints on variables, and finally gives an algorithm that uses the rules to solve the constraints. We first define two classes of type expressions, L (for "Left") and R (for "Right"):

$$\begin{aligned} L &::= 0 \mid \alpha \mid c(L_1, \ldots, L_n) \mid R \to L \mid L_1 \cap \overline{L_2} \mid L_1 \cup L_2 \\ R &::= 0 \mid \alpha \mid c(R_1, \ldots, R_n) \mid L \to R \mid R_1 \cap R_2 \mid \\ &\qquad R_1 \cup R_2 \text{ where } \overline{R_1 \cap R_2} = 0 \end{aligned}$$

A system of *proper* constraints has the form $\{L_i \subseteq R_i\}$. As discussed in Section 6, the restrictions on intersection in L types and unions in R types arise from the asymmetry of $\subseteq$ and limitations on defining a complement operation on types. Note that arbitrary unions are permitted in L types and arbitrary intersections in R types. This turns out to be useful, and, to our knowledge, this possibility for intersection and union types has not been explored before. The type 1 does not appear in the grammar because it can be defined using proper constraints (see Section 2); we continue, however, to use 1 in expressions for convenience.

The inclusion-based inference system in Figure 2 generates proper constraints since the initial constraints have no intersections or unions. The Hindley/Milner system in Figure 1 also generates proper constraints, since $X = Y$ is equivalent to $X \subseteq Y$ and $Y \subseteq X$ and there are no intersections or unions.

Referring again to the inclusion constraints in Figure 2, it is easy to see that functions (including primitive functions) must be assigned L types for the type inference rule for application to work. Before continuing, we give an example of the expressive type signatures that can be given for primitive functions. Consider a higher-order conditional *if* $e_1\ e_2\ e_3\ x$. In our type language, *if* has type

$$\begin{aligned} (\alpha_1 \to \mathit{bool}) \to (\alpha_2 \to \beta_2) \to (\alpha_3 \to \beta_3) \to \\ \alpha_1 \cap \alpha_2 \cap \alpha_3 \to \beta_2 \cup \beta_3 \end{aligned}$$

The whole expression is an L type; in the last function $\alpha_1 \cap \alpha_2 \cap \alpha_3$ is an R type (as required) and $\beta_1 \cup \beta_2$ is an L type (as required). Thus, we are able to use arbitrary intersection and union exactly where the natural signature dictates. We have found that the division into L and R types lends itself naturally to writing function signatures.

In giving the rules for simplifying constraints, it is useful to assume types are in disjunctive normal form. From here on, we assume that all types are normalized using the equivalences in Figure 4. These rules drive intersections "in" and eliminate redundant conjuncts and disjuncts. Note that after normalization every intersection in an L type is of the form $\alpha \cap \overline{L}$ for some variable α.

Before giving the rules for simplifying constraints we need one more definition. A constraint $c(\alpha, \beta) \subseteq 0$ can be satisfied in one of two ways: either $\alpha \subseteq 0$ or $\beta \subseteq 0$.[3] More formally, we can say that the solutions of $c(\alpha, \beta) \subseteq 0$ are the union of the solutions of $\alpha \subseteq 0$ and the solutions of $\beta \subseteq 0$. This motivates the following:

Definition 7.1 Let Γ, Γ' be sets of systems of constraints. We say $\Gamma \equiv \Gamma'$ iff $\bigcup_{S \in \Gamma} \mathcal{S}(S) = \bigcup_{S' \in \Gamma'} \mathcal{S}(S')$.

[3]This would not be the case in a lazy language.

Figure 5 gives rules for simplifying constraints. Each rule maps a set of systems to an equivalent set of systems. There are several things to check: that each rule is correct, that all cases are covered, and that generated constraints are of the form $L \subseteq R$. Except for a brief discussion of the correctness of Rule 4, we leave these to the reader. Rule 4 says that either function types are related by the usual ordering (contravariant in the first component, covariant in the second) or the type on the right-hand side is the set of all functions. Using the semantics of function types in Section 2, it can be shown that in a strict language $X \rightarrow Y$ is the set of all functions iff $X = 0$ or if $Y = 1$. (In a lazy language, only types of the form $X \rightarrow 1$ denote the set of all functions.)

For efficiency, our implementation of the constraint solver discards some solutions. Since universally quantified types are intersections over all instantiations of the variables that solve the constraints, discarding solutions amounts to removing elements from the intersection, resulting in a potentially larger type. Inferring a larger type is always permissible but may result in a loss of accuracy.

Theorem 7.2 Every proper system $\{L_i \subseteq R_i\}$ is equivalent to a finite set of inductive systems.

Proof: [sketch] The following algorithm transforms a proper system S to a finite set of inductive systems Γ. Initially, let $\Gamma = \{S\}$. A single constraint $\alpha_j \subseteq X$ or $X \subseteq \alpha_j$ is *inductive* iff $TLV(X) \subseteq \{\alpha_1, \ldots, \alpha_{j-1}\}$ (see Definition 5.1). Iterate the following steps until all constraints are inductive, no additional inductive constraints can be added, and there are no inconsistent systems:

1. For any constraint that is not inductive, apply the lowest numbered applicable rule in Figure 5.

2. For any pair of inductive constraints $L \subseteq \alpha_j$ and $\alpha_j \subseteq R$ in a system S, add the transitive constraint $L \subseteq R$ to S.

3. Delete any system from Γ with a constraint $1 \subseteq 0$ or $b \subseteq 0$ for nullary constructor b; such systems have no solutions.

Finally, for each $S \in \Gamma$, combine lower bounds $L_1 \subseteq \alpha$, $L_2 \subseteq \alpha$ into $L_1 \cup L_2 \subseteq \alpha$ and upper bounds $\alpha \subseteq R_1$, $\alpha \subseteq R_2$ into $\alpha \subseteq R_1 \cap R_2$. The result is a set of inductive systems.

This algorithm can be proven correct in three steps. First, show that rules 1-11 transform any constraint $L \subseteq R$ into inductive constraints. Second, show that the transitive closure terminates. Third, show that a system of inductive constraints closed under transitivity is an inductive system. We show only the first two parts; the third part can be proven by adapting a similar proof in [AW92a].

For the first part, rules 1-10 either make the right-hand side smaller or make the left-hand side smaller and do not change the right-hand side. Whenever rule 11 is applied, the result is an inductive constraint. For the second part, no rule increases the nesting depth of constructors, and there are only finitely many disjunctive-normal form expressions with no duplicate conjuncts or disjuncts and a fixed depth of constructors. Since no inductive constraints are deleted, the set of inductive constraints must eventually reach a fixed point. □

The algorithm given in the proof of Theorem 7.2 produces a finite set of inductive systems Γ equivalent to the original proper system S. Since inductive systems always have solutions (Theorems 5.4 and 5.5), it follows that $\Gamma = \emptyset$ iff $\mathcal{S}(S) = \emptyset$. Thus, the algorithm computes a representation of all solutions of S and S is inconsistent iff $\Gamma = \emptyset$.

8 Related Work

In this section we briefly survey other work on solving systems of type inclusion constraints. Because most of this work has taken place in the context of type inference and not solely for the study of inclusion constraints, we also compare other type inference systems with the one presented in Section 3. Unfortunately, the literature on type inference is enormous and we must pass over some interesting systems for lack of space.

We begin by reconsidering the Hindley/Milner system. We have already argued informally that our system captures all Hindley/Milner typings. The following lemma makes this precise.

Lemma 8.1 Let e be a lambda term and let S^{HM} be the Hindley/Milner constraints of a type derivation, and let S^{IC} be the corresponding inclusion constraints. Then $\mathcal{S}(S^{HM}) \subseteq \mathcal{S}(S^{IC})$.

Proof: By inspection of the constraints. □

Partial types are a generalization of the Hindley/Milner system. Partial types were introduced in [Tha88]; the problem of solving inclusion constraints over partial types has received considerable attention recently [OW92, KPS92]. The partial types are

$$\tau ::= c(\tau_1, \ldots, \tau_n) \mid \tau_1 \rightarrow \tau_2 \mid \alpha \mid 1$$

In works on partial types, 1 is written Ω. Only the inference rule for application is modified from the Hindley/Milner system. The translation of the rule in [Tha88] into our notation is

$$\frac{A, S \vdash e_1 : \tau_1, e_2 : \tau_2}{A, S \cup \{\alpha = \tau_2, \tau_1 \subseteq \alpha \rightarrow \beta\} \vdash e_1\ e_2 : \beta}$$

Lemma 8.2 Let e be a lambda term and let S^{P} be the partial type constraints of a type derivation, and let S^{IC} be the corresponding inclusion constraints. Then $\mathcal{S}(S^{P}) \subseteq \mathcal{S}(S^{IC})$.

Proof: By inspection of the constraints. □

The work of Mishra and Reddy on declaration-free type checking was one of the original inspirations for this work [MR85]. Their algorithm for solving inclusion constraints is more restrictive than ours in several ways. It is restricted to a first-order language, and all unions are required to be *discriminative*, which means that disjuncts must have different outermost constructors (i.e., $c(\ldots) \cup d(\ldots)$). Intersection is also restricted so that it cannot appear on the left-hand side of constraints.

The inference system that is closest in spirit to ours is *soft typing*, proposed by Cartwright and Fagan [CF91]. Their typing algorithm also generates type constraints that must be solved. The constraints are not solved directly; they are first encoded in a special representation in which circular unification can be used to obtain representations of solutions, which are then decoded back to types. While their system handles higher-order functions, unions are required to be discriminative and there are no intersection types.

9 Current and Future Work

The implementation described in Section 3 is being extended to handle analysis of programs written in FL, a dynamically typed functional language [BWW+89]. In this variation, the solutions of the constraints are used to determine where run-time type checks are required [AW92b]. Based on our previous experience with implementing set constraints, we believe that the algorithms presented here can be implemented efficiently in practice [AM91].

There are interesting types that the system described here cannot handle. For example, the type of an overloaded function such as

$$+ : (\text{int} \to \text{int} \to \text{int}) \cap (\text{real} \to \text{real} \to \text{real})$$

is not an L type and thus cannot be used as the signature for $+$ in the inference algorithm. Similarly, strictness properties such as

$$+ : (0 \to 1 \to 0) \cap (1 \to 0 \to 0)$$

are also not L types. We plan to investigate whether the restrictions on unions in R types and intersections in L types can be relaxed further. While some restrictions seem necessary, the results of Section 6 are not necessarily the best possible.

10 Acknowledgements

We would like to thank Lennart Augustsson, T.K. Lakshman, John Mitchell, Satish Thatte, and John Williams for discussions and comments on earlier drafts of this paper. We are also grateful to T.K. Lakshman for implementing the constraint solving algorithm.

References

[AM91] A. Aiken and B. Murphy. Implementing regular tree expressions. In *Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture*, pages 427–447, August 1991.

[AW92a] A. Aiken and E. Wimmers. Solving systems of set constraints. In *Symposium on Logic in Computer Science*, pages 329–340, June 1992.

[AW92b] A. Aiken and E. Wimmers. Type inference with set constraints. Research Report 8956, IBM, 1992.

[BWW+89] J. Backus, J. H. Williams, E. L. Wimmers, P. Lucas, and A. Aiken. The FL language manual parts 1 and 2. Technical Report RJ 7100 (67163), IBM, 1989.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, June 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surverys*, 17(4):471–522, December 1985.

[FM88] Y. Fuh and P. Mishra. Type inference with subtypes. In *Proceedings of the 1988 European Symposium on Programming*, pages 94–114, 1988.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[KPS92] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. In *Foundations of Computer Science*, pages 363–371, October 1992.

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

[Mit84] J. Mitchell. Coercion and type inference (summary). In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 175–185, January 1984.

[MPS84] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymophic types. In *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 165–174, January 1984.

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

[OW92] P. N. O'Keefe and M. Wand. Type inference for partial types is decidable. In *Proceedings of the 1992 European Symposium on Programming*, 1992.

[Rey85] J. C. Reynolds. Three approaches to type structure. In *Proc. TAPSOFT Advanced Seminar on the ROle of Semantics in Software Development*, Berlin, March 1985. Springer Lecture Notes in Computer Science.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Tha88] S. Thatte. Type inference with partial types. In *Automata, Languages and Programming: 15th International Colloquium*, pages 615–629. Springer-Verlag Lecture Notes in Computer Science, vol. 317, July 1988.

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, X:115–122, 1987.

A Transformations Performed by the Algorithm

$$\begin{array}{rcl@{\qquad}rcl}
\overline{X \cap Y} & = & \overline{X} \cap \overline{Y} & \neg\overline{X \cap Y} & = & \neg\overline{X} \cap \overline{Y} \cup \neg\overline{X} \cap \neg\overline{Y} \cup \overline{X} \cap \neg\overline{Y} \\
\overline{X \cup Y} & = & \overline{X} \cap \neg\overline{Y} \cup \overline{X} \cap \overline{Y} \cup \neg\overline{X} \cap \overline{Y} & \neg\overline{X \cup Y} & = & \neg\overline{X} \cap \neg\overline{Y} \\
\overline{X \to Y} & = & 0 \to 1 & \neg\overline{X \to Y} & = & \bigcup_{c \in C} c(1, \ldots, 1) \\
\overline{\alpha} & = & 1 & \neg\overline{\alpha} & = & 0 \\
\overline{0} & = & 0 & \neg\overline{0} & = & 1 \\
\overline{c(X_1, \ldots, X_n)} & = & c(\overline{X_1}, \ldots, \overline{X_n}) & & & \\
\neg\overline{c(X_1, \ldots, X_n)} & = & \multicolumn{4}{l}{\bigcup\{d(1, \ldots, 1) | d \in C - \{c\}\} \cup 0 \to 1 \cup} \\
 & & \multicolumn{4}{l}{c(\neg\overline{X_1}, 1, \ldots, 1) \cup c(\overline{X_1}, \neg\overline{X_2}, 1, \ldots, 1) \cup \ldots \cup c(\overline{X_1}, \ldots, \overline{X_{n-1}}, \neg\overline{X_n})}
\end{array}$$

Figure 3: Simplifying $\overline{X}$ and $\neg\overline{X}$.

$$\begin{array}{rcl@{\qquad}rcl}
X \cup X & = & X & X \cap X & = & X \\
(X \cup Y) \cap Z & = & (X \cap Z) \cup (Y \cap Z) & (\alpha \cap X) \cap Y & = & \alpha \cap (X \cap Y) \\
c(X_1, \ldots, X_n) \cap c(Y_1, \ldots, Y_n) & = & c(X_1 \cap Y_1, \ldots, X_n \cap Y_n) & c(\ldots) \cap d(\ldots) & = & 0 \text{ if } c \neq d \\
X \to Y \cap 0 \to 1 & = & X \to Y & c(\ldots) \cap X \to Y & = & 0 \\
X \cap 0 & = & 0 & & &
\end{array}$$

Figure 4: Putting types in disjunctive normal form.

$$\begin{array}{rcll}
\Gamma, S \cup \{0 \subseteq R\} & \equiv & \Gamma, S & (1) \\
\Gamma, S \cup \{c(L_1, \ldots, L_n) \subseteq c(R_1, \ldots, R_n)\} & \equiv & \Gamma, S \cup \{L_i \subseteq R_i | 1 \leq i \leq n\}, (S \cup \{c(L_1, \ldots, L_n) \subseteq 0\})^* & (2) \\
\Gamma, S \cup \{c(L_1, \ldots, L_n) \subseteq F\} & \equiv & \Gamma, (S \cup \{L_1 \subseteq 0\})^*, \ldots, (S \cup \{L_n \subseteq 0\})^* & \\
 & & \text{if } F \text{ is } 0, X \to Y, \text{ or } d(\ldots) \text{ where } d \neq c & (3) \\
\Gamma, S \cup \{R_1 \to L_1 \subseteq L_2 \to R_2\} & \equiv & \Gamma, S \cup \{L_2 \subseteq R_1, L_1 \subseteq R_2\},\ (S \cup \{L_2 \subseteq 0\})^*,\ (S \cup \{1 \subseteq R_2\})^* & (4) \\
\Gamma, S \cup \{R \to L \subseteq F\} & \equiv & \Gamma \quad \text{if } F \text{ is } 0 \text{ or } c(\ldots) & (5) \\
\Gamma, S \cup \{L_1 \cup L_2 \subseteq R\} & \equiv & \Gamma, S \cup \{L_1 \subseteq R, L_2 \subseteq R\} & (6) \\
\Gamma, S \cup \{L \subseteq R_1 \cap R_2\} & \equiv & \Gamma, S \cup \{L \subseteq R_1, L \subseteq R_2\} & (7) \\
\Gamma, S \cup \{L \subseteq R_1 \cup R_2\} & \equiv & \Gamma, S \cup \{L \cap \neg\overline{R_1} \subseteq R_2, L \cap \neg\overline{R_2} \subseteq R_1\} & (8) \\
\Gamma, S \cup \{\alpha \subseteq \alpha\} & \equiv & \Gamma, S & (9) \\
\Gamma, S \cup \{\alpha \cap \overline{L} \subseteq \alpha\} & \equiv & \Gamma, S & (10) \\
\Gamma, S \cup \{\alpha \cap \overline{L} \subseteq R\} & \equiv & \Gamma, S \cup \{\alpha \subseteq (R \cap \overline{L}) \cup \neg\overline{L}\} & (11)
\end{array}$$

The systems marked with * are usually discarded for efficiency.

Figure 5: Rules for simplifying constraints.