Type Inference with Inequalities

Michael I. Schwartzbach
mis@daimi.aau.dk

Computer Science Department
Aarhus University
Ny Munkegade
DK-8000 Århus C, Denmark

Abstract

Type inference can be phrased as constraint-solving over types. We consider an implicitly typed language equipped with recursive types, multiple inheritance, 1st order parametric polymorphism, and assignments. Type correctness is expressed as satisfiability of a possibly infinite collection of (monotonic) inequalities on the types of variables and expressions. A general result about systems of inequalities over semilattices yields a solvable form. We distinguish between deciding *typability* (the existence of solutions) and *type inference* (the computation of a minimal solution). In our case, both can be solved by means of nondeterministic finite automata; unusually, the two problems have different complexities: polynomial vs. exponential time.

1 Introduction

In programming languages one distinguishes between *explicit* and *implicit* typings. For the λ-calculus, the explicit case looks like

$$\lambda x : \tau . e$$

where τ is the type of the parameter. The implicit case looks like

$$\lambda x . e$$

where the type of the parameter must be *inferred* from e. The philosophical and proof theoretical differences between the two approaches have been studied

in great depth [1,7]. Implicit typings form the basis for languages like ML and MIRANDA, whereas more complicated λ-systems, and all PASCAL-like languages, use explicit typings.

The legality of calls are determined by the formal and actual parameter types. In the implicit case, the formal parameter types are inferred from the body. Thus, a change in the body may invalidate existing calls. In the explicit case, the formal parameters are independent of the body, which ensures a certain modularity in the program. However, there are also advantages to implicit typings. Programs become more succinct, and no calls will needlessly be deemed illegal when the explicit formal parameter types are unnecessarily specific.

The traditional view of type inference is to consider a typed language and a function ERASE that removes type annotation in a program. One then seeks a function INFER that reconstructs the type annotation, if possible; when several typings are available, INFER should produce an optimal one.

In this paper we consider the untyped language first. Given a program P we define a collection of constraints $\textsc{correct}(P)$ on the (unknown) types of variables and expressions occurring syntactically in P. We then consider the program to be *type correct* if these constraints are solvable over the types. When we have a solution, we may then choose to annotate various parts of the program.

The definition of $\textsc{correct}(P)$ must be *sound* with respect to the dynamic semantics of the language. This means that for type correct programs certain invariants must be guaranteed to hold during their execution. These invariants are crucial for reasoning about program correctness, and are also very useful for developing efficient implementations and performing compile-time optimizations. Typical invariants are: operations are only performed on arguments of the proper types, e.g. the successor function in only allowed on integers, the length function is only allowed for lists, and a product is never confronted with requests for components whose names are not mentioned in its type.

Type *checking* is now the problem of, given a typing $\mathcal{T}$, to decide if

$$\mathcal{T} \models \textsc{correct}(P)$$

whereas type *inference* is the problem of deciding

$$\exists \mathcal{T} : \mathcal{T} \models \textsc{correct}(P)$$

In fact, we shall distinguish between *typability*, which is the above decision problem, and *type inference*, which is the *computation* of an optimal solution. Since the typing $\mathcal{T}$ may be partially known, we have a continuous spectrum between type checking and type inference.

Note that also the usual ML-style type inference may be viewed in this manner. The constraints are *equalities* of type variables ranging over *type schemes*; they are solved using *unification* to yield a *principal* typing.

In this paper we consider a language with recursive types, multiple inheritance, 1st order parametric polymorphism, and assignments. In [10] this is analyzed with *explicit* typings, and a polymorphic mechanism is proved sound and optimal. This time we analyze *implicit* typings. The constraints are *inequalities* over *types*; they are solved using *finite state automata* to yield a *minimal* typing. This is a novel kind of language to subject to type inference, and the applied technique may be interesting in its own right.

2 The Types

Types are defined by means of a set of *type equations*

$$\textbf{Type } N_i = \tau_i$$

where the N_i's are *type variables* and the τ_i's are *type expressions*, which are defined as follows

τ ::=	Int | Bool |	simple types
	N_i |	type variables
	$*\tau$ |	lists
	$(n_1 : \tau_1, \ldots, n_k : \tau_k)$	partial products, $k \geq 0$, $n_i \neq n_j$

Here the n_i's are names. Note that type definitions may involve arbitrary recursion.

The $*$-operator corresponds to ordinary finite lists. The *partial product* is a generalization of sums and products; its values are *partial* functions from the tag names to values of the corresponding types, in much the same way that values of ordinary products may be regarded as *total* functions.

The values of types may be taken to be the $\subseteq$-least solutions to the corresponding induced equations on sets. Other interpretations of types are investigated in [9].

Several type expressions may be taken to denote the same *type*. These can be identified by an congruence relation $\approx$, which is defined as the identity of *normal forms*, using the techniques of [3]. To each type expression we associate a unique normal form, which is a labeled tree. Informally, the tree is obtained by repeatedly *unfolding* the type expression. Formally, we use the fact that the set of labeled trees form a *complete partial order* under the partial ordering where $t_1 \sqsubseteq t_2$, *iff* t_1 can be obtained from t_2 by replacing any number of subtrees with the singleton tree Ω. In this setting, normal forms can be defined as limits of chains of approximations. The singleton tree Ω is smaller than all other trees and corresponds to the normal form of the type defined by

Type $N = N$

We shall write Ω to denote any such type expression.

The type ordering is a refinement of $\sqsubseteq$. We want to include relations between partial product types, such that

$$(n_i : T_i) \preceq_0 (m_j : S_j) \;\; \textit{iff} \;\; \{n_i\} \subseteq \{m_j\} \;\wedge\; (\forall i, j :\; n_i = m_j \Rightarrow T_i \preceq_0 S_j)$$

This possibility must extend to infinite types as well; if $\preceq_0$ is the above inductive refinement of $\sqsubseteq$, then the full ordering is defined as

$$S \preceq T \;\Leftrightarrow\; \forall S' \sqsubseteq S,\; |S'| < \infty :\; S' \preceq_0 T$$

Thus, products with fewer components are smaller than products with more components. As noted in [8], trees under this ordering no longer form a cpo. However, all the chains definable by type equations still have limits.

Proposition 2.1:

1) Ω is the smallest type.
2) The type constructors are monotonic and continuous.
3) If $T = F(T)$ is a type equation, then $T = \bigsqcup F^i(\Omega)$.
4) If $T_1 \preceq T_2$, then all values of type T_1 are also values of type T_2.
5) All non-empty subsets of types have greatest lower bounds.
6) Only some subsets of types have least upper bounds.
7) All of $\approx$, $\preceq$, $\sqcap$, and $\sqcup$ are computable.

Proof: 1) holds since every tree in its entirety can be replaced by Ω. 2) follows since the type constructors are syntactic combinations of trees. 3) follows directly from the interpretation of type equations. 4) is proved in [8]. 6) is true since e.g. Int and Bool does not have an upper bound. 7) is the subject of [11]. To prove 5), we start out with a set of types $\{T_i\}$. We shall describe a chain of types whose limit is $\sqcap\{T_i\}$. First of all, the 0'th approximant equals Ω. To define the j'th approximant, we look at the roots of all T_i's. We have a few cases

- all T_i's equal Int; then the j'th approximant is Int.
- all T_i's equal Bool; then the j'th approximant is Bool.
- all T_i's are of the form $*S_i$; then the j'th approximant is $*A$, where A is the $(j-1)$'th approximant of $\sqcap\{S_i\}$.

- all T_i's are partial products; then the j'th approximant is the partial product whose component names are the intersection of the component names of the T_i's (a finite set), in which a component with name n has type the $(j-1)$'th approximant of $\sqcap\{T_i.n\}$.
- in all other cases, the j'th approximant is Ω.

These approximants form a chain whose limit is the greatest lower bound of $\{T_i\}$. It is a lower bound since every approximant is $\preceq_0$ each T_i; it is the greatest such since any finite type $\sqsubseteq$ all T_i's is $\preceq_0$ some sufficiently large approximant. Note that we have not assumed $\{T_i\}$ to be countable. $\Box$

The inclusion ordering on partial products provides the multiple inheritance, whereas parametric polymorphism is obtained through the existence of the minimal type Ω.

3 The Language

We use the example language from [10]. It is a standard imperative language that employs the above type system. However, to obtain *implicit* typings we remove all type annotations from the program text.

3.1 Syntax

The syntactic categories are: statements (S), (program) variables (σ), expressions (ϕ), declarations (D), and programs (P). In the following grammar the symbols P, n_i, x range over arbitrary names, and k is any non-negative number.

S ::= σ:=ϕ |
σ:-n_i |
σ:+(n_i:ϕ) |
P(ϕ_1,...,ϕ_k) |
if ϕ **then** S **end** |
while ϕ **do** S **end** |
S_1 ; S_2

σ ::= x | σ.n_i | σ[ϕ]

P ::= D_1 D_2 ... D_k S

ϕ ::= 0 | ϕ+1 | ϕ-1 |
σ |
ϕ_1 = ϕ_2 |
[ϕ_1,...,ϕ_k] |
|ϕ| |
(n_1:ϕ_1,...,n_k:ϕ_k) |
has(ϕ,n_i)

D ::= **Proc** P(ρ x_1,...,ρ x_k)
S
end P |
Var x

ρ ::= **var** | **val**

3.2 Informal Semantics

Most of the language is quite standard: simple expressions, variables, assignments, comparisons, control structures and procedures with variable- or value parameters. There are static scope rules, but global variables may not be accessed from within procedures.

The partial product acts as a partial function where $\sigma\texttt{:-}n_i$ removes n_i from the domain of σ, $\sigma\texttt{:+(}n_i\texttt{:}\phi\texttt{)}$ updates σ with the value ϕ on n_i, and $\textbf{has}\texttt{(}\phi\texttt{,}n_i\texttt{)}$ decides whether n_i is in the domain of ϕ. Arbitrary partial product constants can be denoted by $\texttt{(}n_1\texttt{:}\phi_1\texttt{,}\ldots\texttt{,}n_k\texttt{:}\phi_k\texttt{)}$; notice that only the defined components are written. A subvariable of a partial product may be selected by $\sigma\texttt{.}n_i$ (provided it is in the domain of σ). A list constant is denoted by $\texttt{[}\phi_1\texttt{,}\ldots\texttt{,}\phi_k\texttt{]}$, and the subvariable with index ϕ is selected by $\sigma\texttt{[}\phi\texttt{]}$ (provided σ has length greater than ϕ). The expression $\texttt{|}\phi\texttt{|}$ denotes the length of the list ϕ.

4 Defining Correctness

In this section we define a collection of inequalities $\textsc{correct}(P)$ for each program P. The definition will be in two parts. First we define the *local* constraints for statement sequences, and then we define the *global* constraints for a complete program.

4.1 Local Constraints

Ignoring for the moment procedure calls, we can state correctness as satisfiability of constraints on the types of all syntactic occurrences of variables and expressions. For every such occurrence ϕ we introduce a distinct type variable $[\![\phi]\!]$. We shall, however, identify the type variables corresponding to different occurrences of the same program variable, since a consistent typing must exist. In contrast, two occurrences of e.g. the empty list can clearly have different types.

For a statement S the *local* constraints are generated as follows from all the phrases in the derivation of S. The right-hand sides of constraints are *type expressions*.

	Phrase:	Constraint:
1)	x	$[\![x]\!] = [\![x]\!]$
2)	$\sigma.n_i$	$[\![\sigma]\!] \succeq (n_i : [\![\sigma.n_i]\!])$
3)	$\sigma\,\texttt{[}\phi\texttt{]}$	$[\![\sigma]\!] \succeq *[\![\sigma\,\texttt{[}\phi\texttt{]}]\!] \wedge [\![\phi]\!] = \mathrm{Int}$
4)	$\sigma\,\texttt{:=}\,\phi$	$[\![\sigma]\!] \succeq [\![\phi]\!]$
5)	$\sigma\,\texttt{:-}n_i$	$[\![\sigma]\!] \succeq (n_i : \Omega)$
6)	$\sigma\,\texttt{:+(}n_i\texttt{:}\phi\texttt{)}$	$[\![\sigma]\!] \succeq (n_i : [\![\phi]\!])$
7)	$0, \phi\texttt{+1}, \phi\texttt{-1}$	$[\![0]\!] = [\![\phi\texttt{+1}]\!] = [\![\phi\texttt{-1}]\!] = [\![\phi]\!] = \mathrm{Int}$
8)	$\phi_1\ \texttt{=}\ \phi_2$	$[\![\phi_1]\!] = [\![\phi_2]\!] \wedge [\![\phi_1\ \texttt{=}\ \phi_2]\!] = \mathrm{Bool}$
9)	$\texttt{[]}$	$[\![\texttt{[]}]\!] \succeq *\Omega$
10)	$\texttt{[}\phi_1\texttt{,}\ldots\texttt{,}\phi_k\texttt{]}$	$\forall i : [\![\texttt{[}\phi_1\texttt{,}\ldots\texttt{,}\phi_k\texttt{]}]\!] \succeq *[\![\phi_i]\!]$
11)	$\texttt{|}\phi\texttt{|}$	$[\![\texttt{|}\phi\texttt{|}]\!] = \mathrm{Int} \wedge [\![\phi]\!] \succeq *\Omega$
12)	$\texttt{(}n_1\texttt{:}\phi_1\texttt{,}\ldots\texttt{,}n_k\texttt{:}\phi_k\texttt{)}$	$[\![\texttt{(}n_1\texttt{:}\phi_1\texttt{,}\ldots\texttt{,}n_k\texttt{:}\phi_k\texttt{)}]\!] \succeq \texttt{(}n_1\texttt{:}[\![\phi_1]\!]\texttt{,}\ldots\texttt{,}n_k\texttt{:}[\![\phi_k]\!]\texttt{)}$
13)	$\textbf{has}\texttt{(}\phi\texttt{,}n_i\texttt{)}$	$[\![\textbf{has}\texttt{(}\phi\texttt{,}n_i\texttt{)}]\!] = \mathrm{Bool} \wedge [\![\phi]\!] \succeq (n_i : \Omega)$
14)	**if** ϕ **then** S **end**	$[\![\phi]\!] = \mathrm{Bool}$
15)	**while** ϕ **do** S **end**	$[\![\phi]\!] = \mathrm{Bool}$

The above definition is very easy to motivate, since in each case the constraint precisely guarantees the correctness of the corresponding phrase. For example, a constraint

$$[\![\phi]\!] \succeq *\Omega$$

simply states that ϕ must be a list. The case **1)** is only introduced to ensure that every variable name occurs in at least one constraint.

As an example, consider the following statement

```
x.a:=[];
y[0]:=x
```

The imposed constraints are

$$\begin{aligned}
[\![\mathrm{x}]\!] &= [\![\mathrm{x}]\!] \\
[\![\mathrm{y}]\!] &= [\![\mathrm{y}]\!] \\
[\![\mathrm{x}]\!] &\succeq (a : [\![\mathrm{x.a}]\!]) \\
[\![\mathrm{x.a}]\!] &\succeq [\![\texttt{[]}]\!] \\
[\![\texttt{[]}]\!] &\succeq *\Omega \\
[\![\mathrm{y}\texttt{[0]}]\!] &\succeq [\![\mathrm{x}]\!] \\
[\![0]\!] &= \mathrm{Int} \\
[\![\mathrm{y}]\!] &\succeq *[\![\mathrm{y}\texttt{[0]}]\!]
\end{aligned}$$

or, reducing by hand, equivalently

$$\begin{aligned}
[\![\mathrm{x}]\!] &\succeq (a : *\Omega) \\
[\![\mathrm{y}]\!] &\succeq *[\![\mathrm{x}]\!]
\end{aligned}$$

which clearly are satisfiable. In contrast, the statement

```
z := [];
z.a := []
```

imposes the constraints

$$\begin{aligned} [\![\mathrm{z}]\!] &\succeq [\![\texttt{[]}]\!]_1 \\ [\![\mathrm{z}]\!] &\succeq (a : [\![\texttt{[]}]\!]_2) \\ [\![\texttt{[]}]\!]_1 &\succeq *\Omega \\ [\![\texttt{[]}]\!]_2 &\succeq *\Omega \end{aligned}$$

which cannot be satisfied, as lists and products do not have common upper bounds. Notice that the two occurrences of `[]` yield distinct type variables.

We can give a more uniform presentation of these constraints. They can all be expressed as inequalities of the form

$$\alpha \succeq H$$

where α is a type variable and H is a type expressions involving other type variables. Only two kinds of constraints are not already in this form. The cases **1)** and **8)** express an equality between two type variables; this we can write as two symmetric inequalities. Several cases involve equalities between a type variable and a simple type; this we can write as the corresponding inequality, since simple types are maximal in the type ordering.

4.2 Global Constraints

The local constraints determine the correctness of simple code in procedure bodies. To obtain the *global* constraints for the entire program, we must combine local constraints as indicated by procedure calls.

Intuitively, we expect the actual parameters to simply inherit the constraints of the formal parameters. Consider the program

```
Proc P(var x, var y)
   x.a := [];
   y[0] := x
end P

P(r,s)
```

One idea is to express correctness by equating the type variables for the formal and actual parameters, e.g. the constraints would essentially be

$$\begin{aligned} [\![\mathrm{x}]\!] &\succeq (a : *\Omega) \\ [\![\mathrm{y}]\!] &\succeq *[\![\mathrm{x}]\!] \\ [\![\mathrm{r}]\!] &= [\![\mathrm{x}]\!] \\ [\![\mathrm{s}]\!] &= [\![\mathrm{y}]\!] \end{aligned}$$

While this definition is sound, it is, however, too restrictive. If we consider another call P(t,u) and the analogous constraints

$$[\![\mathrm{t}]\!] = [\![\mathrm{x}]\!]$$
$$[\![\mathrm{u}]\!] = [\![\mathrm{y}]\!]$$

then it follows that $[\![\mathrm{r}]\!] = [\![\mathrm{t}]\!]$ and $[\![\mathrm{s}]\!] = [\![\mathrm{u}]\!]$. Thus, the procedure P is *monomorphic*, i.e. only one set of actual parameter types is allowed.

Another idea is to simply require that the actual types be larger the the formal types. This definition allows polymorphism but is unsound, as illustrated by the following example. The program

```
Proc Q(var x, var y)
   x:=y
end Q

var a,b
a:=7;
b:=true;
Q(a,b);
a:=a+1
```

will clearly cause a run-time error, since the addition involves a Bool-value. With the above idea, however, the global constraints would essentially be

$$[\![\mathrm{x}]\!] \succeq [\![\mathrm{y}]\!]$$
$$[\![\mathrm{a}]\!] = \mathrm{Int}$$
$$[\![\mathrm{b}]\!] = \mathrm{Bool}$$
$$[\![\mathrm{a}]\!] \succeq [\![\mathrm{x}]\!]$$
$$[\![\mathrm{b}]\!] \succeq [\![\mathrm{y}]\!]$$

which are satisfiable, since we can choose $[\![\mathrm{x}]\!] = [\![\mathrm{y}]\!] = \Omega$.

To obtain sound, polymorphic procedures we must *substitute* actual for formal type variables in the constraints of the procedure body. Thus, the constraints for the above two calls of the procedure P should essentially be

$$[\![\mathrm{r}]\!] \succeq (a : *\Omega) \qquad [\![\mathrm{t}]\!] \succeq (a : *\Omega)$$
$$[\![\mathrm{s}]\!] \succeq *[\![\mathrm{r}]\!] \qquad [\![\mathrm{u}]\!] \succeq *[\![\mathrm{t}]\!]$$

which allow different sets of actual parameter types.

We can give a fairly simple definition that implements these substitutions by means of *renamings*. For a program P we first construct a tree $\textsc{call}(P)$, which contains the pattern of procedure calls. The root is labeled with the set of local

constraints from the main statement in P, and for each procedure call we have a subtree which is the CALL-tree rooted by the corresponding procedure body. The edge to each subtree is labeled with the type variables for the actual parameters. Because of recursion $\text{CALL}(P)$ may be infinite; however, since programs are finite it will only have finitely many *different* subtrees, i.e. $\text{CALL}(P)$ is *regular* [4].

The set $\text{CORRECT}(P)$ of global correctness constraints is obtained from $\text{CALL}(P)$ as follows. Firstly, for every node we rename its associated type variables by indexing them with its unique tree address. This ensures that no two nodes contain a common type variable. Secondly, for each procedure call we equate formal and actual type variables. Finally, we obtain $\text{CORRECT}(P)$ as the union of all constraints in the tree.

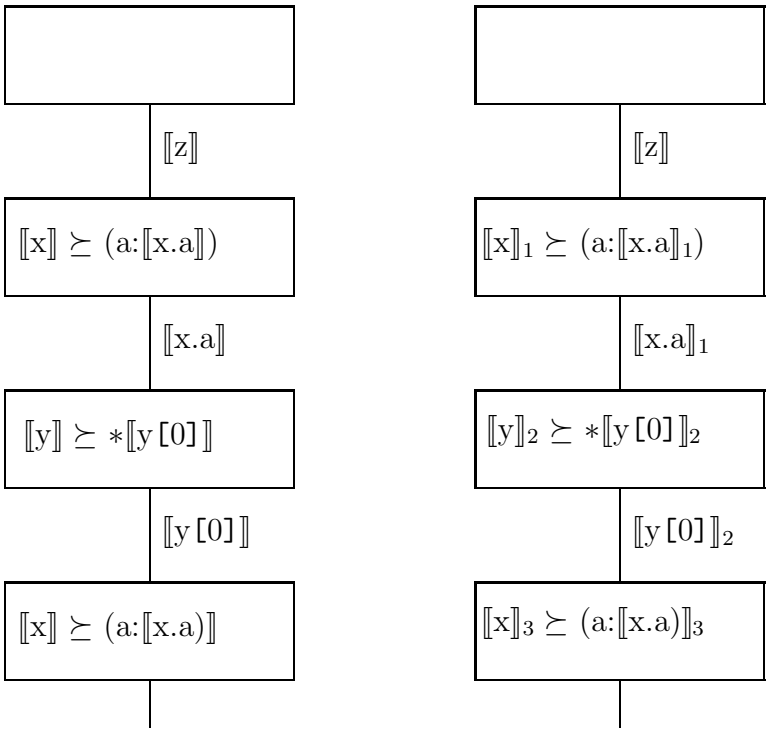
Through recursive procedure calls, $\text{CORRECT}(P)$ may contain infinitely many constraints. For example, given the program

```
Proc R(var x)
  S(x.a)
end R

Proc S(var y)
  R(y[0])
end S

var z
R(z)
```

we find that the CALL-tree and its renamed version are the infinite trees



Now, the global constraints become

$$\begin{aligned} \llbracket \mathrm{z} \rrbracket &= \llbracket \mathrm{x} \rrbracket_1 \\ \llbracket \mathrm{x} \rrbracket_1 &\succeq (a : \llbracket \mathrm{a.x} \rrbracket_1) \\ \llbracket \mathrm{a.x} \rrbracket_1 &= \llbracket \mathrm{y} \rrbracket_2 \\ \llbracket \mathrm{y} \rrbracket_2 &\succeq *\llbracket \mathrm{y\,[0]} \rrbracket_2 \\ \llbracket \mathrm{y\,[0]} \rrbracket_2 &= \llbracket \mathrm{x} \rrbracket_3 \\ &\vdots \end{aligned}$$

We cannot hope to obtain a truly optimal definition of type correctness, since the occurrence of a run-time error is undecidable. However, the present definition is clearly sound and very liberal.

We are left with developing a type inference algorithm.

5 Solving Inequalities

The required analysis of inequalities can take place in a more general setting.

Definition 5.1: Let $(D, \preceq)$ be a poset where every non-empty subset has a

greatest lower bound. An *inequality system* on $(D, \preceq)$ consists of a possibly infinite set of inequalities of the form

$$\alpha_0 \succeq f(\alpha_1, \alpha_2, \ldots, \alpha_k)$$

where the α_i's are variables and $f : D^k \to D$ is a monotonic function. A *solution* L assigns to each variable α some value $L(\alpha) \in D$ such that all the inequalities hold. The system is *satisfiable* when a solution exists. □

Lemma 5.2: If an inequality system has solutions, then it has a unique smallest solution.
Proof: Let $\{L_i\}$ be all solutions. Then $L(\alpha) = \sqcap_i L_i(\alpha)$ is also a solution, since

$$\begin{array}{ll} & L(\alpha_0) \\ = & \sqcap_i L_i(\alpha_0) & \text{by definition} \\ \succeq & \sqcap_i f(L_i(\alpha_1), \ldots, L_i(\alpha_k)) & \text{since } L_i \text{ is a solution} \\ \succeq & f(\sqcap_i L_i(\alpha_1), \ldots, \sqcap_i L_i(\alpha_k)) & \text{since } f \text{ is monotonic} \\ = & f(L(\alpha_1), \ldots, L(\alpha_k)) & \text{by definition} \end{array}$$

Clearly now, L is the unique smallest solution. □

In general, we cannot hope to solve infinite systems. However, if their structure is sufficiently regular, then we can transform infinite systems into finite ones.

Definition 5.3: The equivalence relation $\mathcal{S}_1 \equiv \mathcal{S}_2$ states that the inequality systems $\mathcal{S}_1$ and $\mathcal{S}_2$ are identical up to renaming of variables. □

Definition 5.4: If $\mathcal{S}$ is an inequality system in which α is a variable, then $\mathcal{S} \downarrow \alpha$ is the smallest subset of inequalities such that

- if $q \in \mathcal{S}$ is an inequality that has α on the left-hand side, then $q \in \mathcal{S} \downarrow \alpha$.
- if $q \in \mathcal{S}$ has a variable on the left-hand side that occurs on the right-hand side of $p \in \mathcal{S} \downarrow \alpha$, then $q \in \mathcal{S} \downarrow \alpha$.

We call $\mathcal{S} \downarrow \alpha$ the *closure* of α in $\mathcal{S}$. Intuitively, it contains the relevant constraints on α in $\mathcal{S}$. □

For example, in the system $\mathcal{X}$:

$$\begin{array}{c}
\alpha_1 \succeq f(\alpha_2) \\
\alpha_1 \succeq g(\alpha_3) \\
\alpha_2 \succeq h(\alpha_3) \\
\alpha_3 \succeq f(\alpha_4) \\
\alpha_3 \succeq g(\alpha_5) \\
\alpha_4 \succeq h(\alpha_5) \\
\alpha_5 \succeq f(\alpha_6) \\
\alpha_5 \succeq g(\alpha_7) \\
\alpha_6 \succeq h(\alpha_7) \\
\vdots
\end{array}$$

all closures are co-finite segments like the following

$$\begin{array}{rr}
\alpha_i \succeq f(\alpha_{i+1}) & \alpha_j \succeq h(\alpha_{j+1}) \\
\alpha_i \succeq g(\alpha_{i+2}) & \alpha_{j+1} \succeq f(\alpha_{j+2}) \\
\alpha_{i+1} \succeq h(\alpha_{i+2}) & \alpha_{j+1} \succeq g(\alpha_{j+3}) \\
\vdots & \vdots
\end{array}$$

for odd i and even j.

Definition 5.5: If $\mathcal{S}$ is an inequality system, then $\mathcal{S}/\equiv$ is a new inequality system with a variable $[\mathcal{S}\downarrow\alpha]_\equiv$ for every equivalence class of closures of variables in S. For each $\mathcal{S}$-inequality of the form

$$\alpha_0 \succeq f(\alpha_1, \alpha_2, \ldots, \alpha_k)$$

there is an $\mathcal{S}/\equiv$-inequality of the form

$$[\mathcal{S}\downarrow\alpha_0]_\equiv \succeq f([\mathcal{S}\downarrow\alpha_1]_\equiv, [\mathcal{S}\downarrow\alpha_2]_\equiv, \ldots, [\mathcal{S}\downarrow\alpha_k]_\equiv)$$

i.e. the same inequality on equivalence classes. □

For illustration, we observe that $\mathcal{X}/\equiv$ equals

$$\begin{array}{c}
\textsc{odd} \succeq f(\textsc{even}) \\
\textsc{odd} \succeq g(\textsc{odd}) \\
\textsc{even} \succeq h(\textsc{odd})
\end{array}$$

where ODD and EVEN are the two equivalence classes corresponding to variables with odd and even indices.

Definition 5.6: Let $\mathcal{S}$ be an inequality system. A solution L is *simple* when for

all variables α, β we have

$$\mathcal{S}\downarrow\alpha \equiv \mathcal{S}\downarrow\beta \Rightarrow L(\alpha)=L(\beta)$$

i.e. variables with equivalent closures are assigned the same values. □

Theorem 5.7: If $\mathcal{S}$ has a minimal solution, then it is simple and can be obtained from the minimal solution of $\mathcal{S}/\equiv$.
Proof: Suppose that $\mathcal{S}$ has a minimal solution M. Clearly, the appropriate restriction of M is a solution for any $\mathcal{S}\downarrow\alpha$. If $\mathcal{S}\downarrow\alpha$ had a smaller solution, then M could be modified to yield a smaller solution, since the variables in $\mathcal{S}\downarrow\alpha$ only appear on monotonic right-hand sides outside of $\mathcal{S}\downarrow\alpha$. Thus, M restricted to $\mathcal{S}\downarrow\alpha$ gives its minimal solution. Since this by lemma 5.2 is unique, we know that M will give the same result on all equivalent closures; hence, M is simple. Any simple solution L of $\mathcal{S}$ gives a solution L' of $\mathcal{S}/\equiv$ by

$$L'([\mathcal{S}\downarrow\alpha]_{\equiv}) = L(\alpha)$$

Conversely, any solution Q of $\mathcal{S}/\equiv$ gives a simple solution Q' of $\mathcal{S}$ by

$$Q'(\alpha) = Q([\mathcal{S}\downarrow\alpha]_{\equiv})$$

Since both translations are monotonic, the minimal solution of $\mathcal{S}/\equiv$ yields the minimal simple solution of $\mathcal{S}$, which is just the minimal solution of $\mathcal{S}$. □

The solutions of $\mathcal{S}$ that are not simple take advantage of the freedom to choose different solutions of equal subsystems. By lemma 5.2, this freedom is not allowed for the minimal solution.

Definition 5.8: An inequality system $\mathcal{S}$ is *regular* when $\mathcal{S}/\equiv$ is finite. □

We finally show that a *finite* set of inequalities can be transformed into a finite set of equalities.

Definition 5.9: Let $\mathcal{S}$ be a finite inequality system. We define $\text{LUB}(\mathcal{S})$ to be the set of equalities, where for each $\mathcal{S}$-variable α we include

$$\alpha = H_1 \sqcup H_2 \sqcup \ldots \sqcup H_n$$

when the H_i's are all the right-hands sides of inequalities in $\mathcal{S}$ with α on the left-hand side. $\square$

Continuing the example, we have that $\text{LUB}(\mathcal{X}/\equiv)$ equals

$$\begin{aligned} \text{ODD} &= f(\text{EVEN}) \sqcup g(\text{ODD}) \\ \text{EVEN} &= h(\text{ODD}) \end{aligned}$$

which is a system of recursive $\sqcup$-equations.

Lemma 5.10: If two elements in $(D, \preceq)$ have an upper bound, then they have a least upper bound.
Proof: Let $\{u_i\}$ be all the upper bounds of $d_1, d_2 \in D$. Define $u = \sqcap u_i$. Clearly, $u \preceq u_i$ for all i. Since u is the *greatest* lower bound, it must be larger than both d_1 and d_2, each of which is a lower bound. Hence, $u = d_1 \sqcup d_2$. $\square$

Theorem 5.11: Let $\mathcal{S}$ be a finite inequality system. Then $\mathcal{S}$ has a minimal solution *iff* $\text{LUB}(\mathcal{S})$ has a minimal solution; furthermore, the two solutions are equal.
Proof: Any solution of $\text{LUB}(\mathcal{S})$ is clearly a solution of $\mathcal{S}$. When $\mathcal{S}$ has a solution, then for any α the corresponding H_i's have an upper bound; thus, from lemma 5.10 they have a least upper bound. Suppose for some solution L of $\mathcal{S}$ that $L(\alpha) \succ \sqcup_i L(H_i)$. Then

$$L'(\beta) = \begin{cases} L(\beta) & \text{if } \alpha \neq \beta \\ \sqcup_i L(H_i) & \text{if } \alpha = \beta \end{cases}$$

is a smaller solution, since $L'(\alpha) = \sqcup_i L(H_i) \succeq \sqcup_i L'(H_i)$, and the right-hand sides are monotonic. Thus, for the unique minimal solution M we must have $M(\alpha) = \sqcup_i M(H_i)$, from which the result follows. $\square$

6 Algorithms on Types

The results in the previous section allow us to develop the necessary algorithms on types.

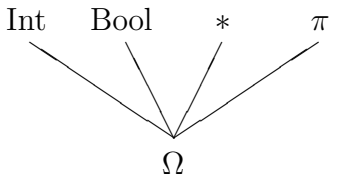
Proposition 6.1: For any P, $\text{CORRECT}(P)$ is a regular inequality system on types.
Proof: From proposition 2.1 we know that non-empty subsets of types have

greatest lower bounds, and that (composite) type constructors are monotonic. Hence, we have an inequality system. Regularity follows from programs being finite. The closure of a variable is completely determined by the subtree of CALL(P) in whose root the variable is introduced. Since we have only finitely many different subtrees, and CORRECT(P) is obtained from CALL(P) by renamings, it follows that we only have finitely many different equivalence classes of closures. □

Proposition 6.2: Given P we can compute LUB(CORRECT(P)/ $\equiv$) in polynomial time.
Proof: The key observation is that CORRECT(P)/$\equiv$ can be constructed directly from CALL(P), since the closures correspond to the finitely many different subtrees. This leaves only the simple task of collecting the related right-hand sides. □

Theorem 6.3: There is an exponential time algorithm that finds the minimal solution to a finite set of $\sqcup$-equalities on types, or decides that no solution exists.
Proof: This is proved in [11]. □

The main idea behind this algorithm is easily explained. There is an isomorphism between type equations and a special kind of finite automata; they must be deterministic, partial automata in which also states are labeled by elements of the following partial order of *coarse types*



Here, $*$ indicates any list type and π any partial product type. The alphabet symbols are the component names of partial products and the special symbol [] that indicates components of lists.

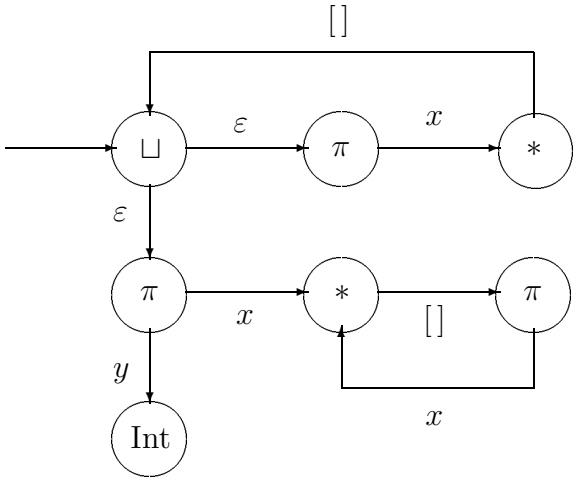
Type equations with $\sqcup$'s will now correspond to *nondeterministic* automata. The algorithm first constructs the deterministic version of the underlying automaton; this may take exponential time. In a second stage the algorithm computes the new labels of the deterministic states. Each such state corresponds to a finite set of nondeterministic states. The new label is the least upper bound of their individual labels, which may or may not exist. If no such label exists, then the

equations have no solution; otherwise, the minimal solution is represented by the deterministic automaton.

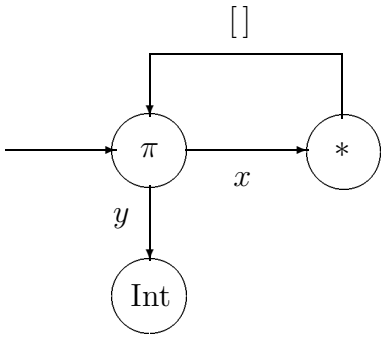
As an example, consider the type A defined by the $\sqcup$-equations

Type $A = (x : B) \sqcup (x : C, y : \text{Int})$
Type $B = *A$
Type $C = *(x : C)$

They give rise to the following automaton



Its deterministic version is



from which we obtain the solution

Type $A = (x : *A, y : \text{Int})$

Since any automaton may be represented as a set of type equations (using only partial products), it follows that we have a tight exponential bound on the worst-case time of this algorithm for constructing the minimal types.

Type equations with both $\sqcup$'s and $\sqcap$'s are similarly solved using *alternating* finite automata [11].

If we only want to decide the *existence* of such types, then we can obtain a more efficient algorithm.

Theorem 6.4: There is a polynomial time algorithm that decides the existence of a solution to a finite set of $\sqcup$-equalities on types.
Proof: Since the partial order of coarse types is a tree, the solution does not exist *iff* some new state contains a *pair* of old states, whose labels are incomparable. Now, we can compute the set of all pairs of old states that end up in the same new state, essentially by a transitive closure. As we have only quadratically many such pairs to check, we obtain a polynomial time decision algorithm. $\Box$

There is no need to be discouraged by the exponential time complexity of the inference algorithm. Typability and type inference in ordinary λ-systems has recently been shown to have similar complexities [5], but in practice the running times are quite acceptable.

7 Conclusion

We have demonstrated that type inference can be viewed as constraint-solving. This allowed us to do type inference for an interesting imperative language. The use of inequality systems and finite automata produced the required algorithms-

The following table succinctly compares ML-style type inference with this present approach.

	ML	This
Unknowns are	var. types	var. and exp. types
Constraints are	equalities	inequalities
Resolution is	unification	determinization of NFSA
Typing is	principal	minimal
Deciding typability is	exponential time	polynomial time
Type inference is	exponential time	exponential time

In some respects the present type system is simpler than the ML-system, since it does not have function types. In some respects it is more complicated, since it has assignments and subtypes; also, in ML the types of variables completely determine the types of all expressions. It is not obvious how to extend the present system with function types, since their contravariance would prevent the inequalities from being purely monotonic. But, on the other hand, it is not obvious how to extend ML with assignments, reference parameters, and subtypes [2].

As a further point of comparison, we consider the possibilities for including partial type information in programs. In ML-style languages an exact typing of selected parameters may be specified at will; this can make programs more legible and resolve detrimental ambiguities. In our approach, the inequalities we employ only allow for specifications of lower bounds of typings.

Future Work

It would, of course, be desirable to extend the present approach to more general constraints and more general type systems. One could possibly achieve inequalities with type expressions on both sides, rather than merely variables on the left-hand side.

Recent work shows that an extension with *conditional* inequalities is appropriate for type inference in object-oriented languages with *late bindings*.

It would be worth pursuing the observation that ML-style type inference also can be viewed as inequality solving over *type schemes* ordered by the existence of *substitutions*; function types are, of course, monotonic under this ordering. A general framework encompassing several such situations may be lurking in the background.

References

[1] **Barendregt, H.** "Types in Lambda Calculi and Programming Languages" in *Proceedings of ESOP'90*, *LNCS Vol 432*, Springer-Verlag, 1990.

[2] **Cardelli, L. & Mitchell, J.** "Operations on Records" in *Proceedings of MFPS'90*, *LNCS Vol 442*, Springer-Verlag, 1990.

[3] **Courcelle, B.** "Infinite Trees in Normal Form and Recursive Equations Having a Unique Solution" in *Mathematical Systems Theory 13, 131-180.* Springer-Verlag 1979.

[4] **Courcelle, B.** "Fundamental Properties of Infinite Trees" in *Theoretical Computer Science Vol 25 No 1*, North-Holland 1983.

[5] **Mairson, H.G.** "Decidability of ML Typing is Complete for Deterministic Exponential Time" in *Proceedings of POPL'90*, ACM 1990.

[6] **Milner, R.** "A Theory of Type Polymorphism in Programming Languages" in *Journal of Computer and Systems Sciences 17*, 1978.

[7] **Reynolds, J.C.** "Three approaches to type structure" in *Mathematical Foundations of Software Development*, *LNCS Vol 185*, Springer-Verlag, 1985.

[8] **Schmidt, E.M. & Schwartzbach, M.I.** "An Imperative Type Hierarchy with Partial Products" in *Proceedings of MFCS'89*, *LNCS Vol 379*, Springer-Verlag, 1989.

[9] **Schwartzbach, M.I.** "Infinite Values in Hierarchical Imperative Types" in *Proceedings of CAAP'90*, *LNCS Vol 431*, Springer-Verlag, 1990.

[10] **Schwartzbach, M.I.** "Static Correctness of Hierarchical Procedures" in *Proc. of ICALP'90*, *LNCS Vol 443*, Springer-Verlag, 1990.

[11] **Schwartzbach, M.I. & Schmidt, E.M.** "Types and Automata". *PB-316*, Department of Computer Science, Aarhus University, 1990.