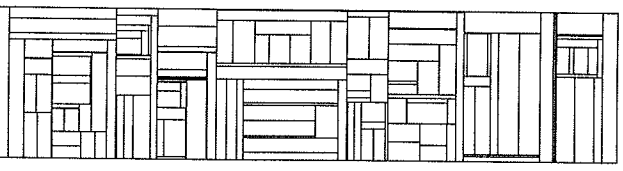
ISSN 0105-8517

Types and Automata

Michael I. Schwartzbach
Erik M. Schmidt

DAIMI PB – 316
July 1990

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark



ISSN 0105-8517

PB – 316 Schwartzbach & Schmidt: Types and Automata

Types and Automata

Michael I. Schwartzbach[1] **Erik M. Schmidt**[2]

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

Abstract

A hierarchical type system for imperative programming languages gives rise to various computational problems, such as type equivalence, type ordering, etc. We present a particular class of finite automata which are shown to be isomorphic to type equations. All the relevant type concepts turn out to have well-known automata analogues, such as language equality, language inclusion, etc. This provides optimal or best known algorithms for the type system, by a process of translating type equations to automata, solving the analogous problem, and translating the result back to type equations. Apart from suggesting an implementation, this connection lends a certain naturality to our type system. We also introduce a very general form of extended (recursive) type equations which are explained in terms of (monotone) alternating automata. Since types are simply equationally defined trees, these results may have wider applications.

1 Introduction

We concern ourselves with a particular type system in which types are possibly infinite, labeled trees. In order to employ this system in a programming language we need to demonstrate decidability of certain properties and computability of certain constructions. Furthermore, we want to obtain reasonably efficient algorithms. Among others, we need to provide solutions for equality, ordering, least upper bounds, and greatest lower bounds of types.

[1]E-mail address: mis@daimi.dk
[2]E-mail address: emschmidt@daimi.dk

The main contribution of this paper is that type equations can be translated into a special kind of finite automata, such that the computational problems for types become well-known automata problems. For example, type equivalence is language equality, type ordering is language inclusion, and least upper bounds and greatest lower bounds correspond to the constructions of union and intersection languages.

This connection serves the dual purpose of providing the desired algorithms and justifying that our type system is, in some sense, very *natural.*

Inspired by the prefixing notation of class hierarchies we introduce a very general form of extended type equations by allowing least upper bounds and greatest lower bounds to appear as type operators in (recursive) type equations. This mechanism is demonstrably very useful in simpler cases, but we resolve the completely general situation by exploiting further the strong connections to automata theory. We present an algorithm to transform extended equations into equivalent basic equations, or to decide that no such transformation is possible. The central part of this algorithm finds a deterministic version of an alternating finite automaton with monotone connectives *and* ε-transitions.

The extended type equations can be very succinct, as the transformation of a collection of n extended equations may require $2^{2^{O(n)}}$ basic equations.

2 The Types

In this section we give a brief presentation of the type system in question. For the purposes of this paper it is sufficient to view types as just *labeled trees.* The semantic aspects of the present type system are discussed in detail in [8,9,10].

2.1 Type Equations

The types are specified through a set of basic type equations

$$\begin{aligned}
&\textbf{Type } T_1 = E_1\\
&\textbf{Type } T_2 = E_2\\
&\quad\vdots\\
&\textbf{Type } T_n = E_n
\end{aligned}$$

where the T_i's are type *variables* and the E_i's are type *expressions* of the following form

$$\begin{array}{lll} E ::= & \alpha_i & \text{simple types} \\ | & T_i & \text{type variables} \\ | & *E & \text{lists} \\ | & (n_1 : E_1, \ldots, n_k : E_k) & \text{partial products, } k \geq 0,\ n_i \neq n_j \end{array}$$

Here the n_i's are component *names*. Note that type equations may involve arbitrary recursion. The simple types could be e.g. Int or Bool.

The *types* themselves are (possibly infinite) labeled trees generated by unfolding the type equations. Informally, the trees are obtained by a (never-ending) process of replacing type variables with the right-hand sides of their definitions. A simple type yields a singleton tree labeled with the type name. The expression $*E$ yields a node labeled $*$ with a single subtree corresponding to E. The expression $(n_1 : E_1, \ldots, n_k : E_k)$ yields a node labeled $(n_1, \ldots, n_k)$ with k subtrees corresponding to the E_i's.

Formally, these trees can be defined as limits of approximations in the complete partial order of labeled trees ordered by $\sqsubseteq$, where $T_1 \sqsubseteq T_2$ if T_1 can be obtained from T_2 by replacing some subtrees with the special symbol Ω. This technique is developed in [5]; details of our application may be found in [8].

Notice that the singleton tree Ω is the bottom element. This tree is also a type, as it can be defined by the equation

$$\textbf{Type } A = A$$

We introduce the *constant* Ω to denote this type directly.

In the following it will prove convenient to work exclusively with *normalized* type equations, which are all of the form

$$\begin{array}{l} \textbf{Type } A = \alpha_i \\ \textbf{Type } A = *B \\ \textbf{Type } A = (n_1 : B_1, \ldots, n_k : B_k) \\ \textbf{Type } A = \Omega \end{array}$$

where the B's are type variables. Thus, normalized type equations have exactly one type constructor on the right-hand side. Clearly, all type

equations can be normalized by the introduction of extra type variables.

2.2 Equivalence and Ordering

A type will be presented as $(T, \mathcal{E})$ where T is a type variable defined by the (normalized) equations $\mathcal{E}$.

We shall use the notation $(T_1, \mathcal{E}_1) \approx (T_2, \mathcal{E}_2)$ to indicate that the two denoted types are equal. Obviously, this is an important question.

Another vital concept is a *partial order* $\preceq$ on types, which is obtained as a refinement of $\sqsubseteq$. The idea is that products with fewer components are smaller than products with more components. If $\preceq_0$ is the smallest refinement of $\sqsubseteq$ where

$$(n_i : A_i) \preceq_0 (m_j : B_j) \quad \textit{iff} \quad \{n_i\} \subseteq \{m_j\} \wedge n_i{=}m_j \Rightarrow A_i \preceq_0 B_j$$

then the full ordering is

$$A \preceq B \quad \textit{iff} \quad \forall A_0 \sqsubseteq A,\ |A_0| < \infty :\ A_0 \preceq_0 B$$

In [8,10] this ordering is used as the basis for a type system which allows 1st order polymorphism, multiple inheritance, and general specialization in a language with assignments. This turns out to provide optimal code reuse.

We shall use the notation $(T_1, \mathcal{E}_1) \preceq (T_2, \mathcal{E}_2)$ to indicate that the two denoted types are related. This is also an important property to decide.

2.3 Inheritance and Specialization

Multiple inheritance and general specialization are canonical concepts in this type system. They are realized as respectively *least upper bounds* and *greatest lower bounds* in the partial order of types.

Greatest lower bounds, denoted $\sqcap$, always exist. Given $(T_i, \mathcal{E}_i)$, the problem is to find $(T, \mathcal{E}) = (T_1, \mathcal{E}_1) \sqcap (T_2, \mathcal{E}_2)$. For example

$$*(a : A, b : \mathrm{Int}) \sqcap *(b : \mathrm{Bool}, c : C) = *(b : \Omega)$$

In contrast, least upper bounds, denoted $\sqcup$, may or may not exist. Given $(T_i, \mathcal{E}_i)$, the problem is to find $(T, \mathcal{E}) = (T_1, \mathcal{E}_1) \sqcup (T_2, \mathcal{E}_2)$, or to say *no*

when no upper bound exists. For example

$$(a : A, b : B) \sqcup (b : B, c : C) = (a : A, b : B, c : C)$$

but $\mathrm{Int} \sqcup *A$ does not exist.

2.4 Hierarchical Consistency

The type ordering works by allowing *hierarchical* procedure calls, where the types of actual parameters are larger than those of the formal parameters. In [10] is presented an optimal *consistency* requirement, which determines the legality of actual parameter types.

The formulation of this requirement involves the notion of *type addresses*. A type address denotes a subtree of a type by indicating the path from the root to the subtree. These are sequences of component names and the special symbol []. The names indicate edges from a partial product, and [] indicates the unique edge from a list. We let $add(A)$ denote the set of type addresses which lead to existing subtrees in the type A. If $\gamma \in add(A)$ then $A{\downarrow}\gamma$ denotes the corresponding type.

Two types A and B are consistent *iff*

$$A \preceq B \ \wedge \ \forall \gamma, \gamma' \in add(A) : \ (A{\downarrow}\gamma = A{\downarrow}\gamma' \ \Rightarrow \ B{\downarrow}\gamma = B{\downarrow}\gamma')$$

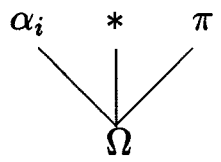
The problem is, of course, given $(T_1, \mathcal{E}_1)$ and $(T_2, \mathcal{E}_2)$ to decide if the denoted types are consistent.

3 The Automata

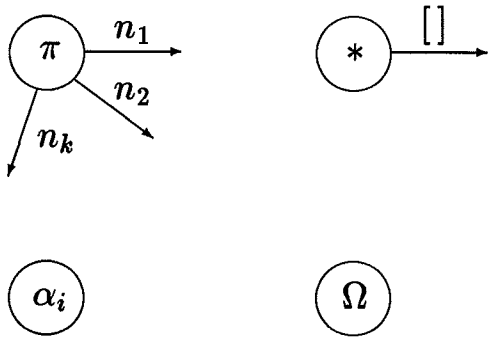
In this section we introduce a slightly modified notion of finite automata, which we shall call *typical* automata. A typical automaton is a partial, deterministic, finite automaton in which all states are labeled, and all states accept. The labels are the so-called *coarse types*, which are the symbols in $\{\Omega, \alpha_i, *, \pi\}$. If A is a type then

$$coarse(A) = \begin{cases} \Omega & \text{if } A = \Omega \\ \alpha_i & \text{if } A = \alpha_i \\ * & \text{if } A \text{ is a list} \\ \pi & \text{if } A \text{ is a partial product} \end{cases}$$

The coarse types inherit the ordering of real types



i.e. Ω is smaller than the others. A typical automaton has exclusively states of these four kinds



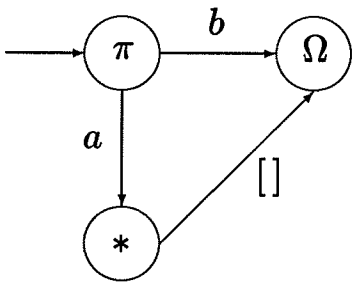
We associate two languages with a typical automaton, $\mathcal{A}$. The first, $L(\mathcal{A})$, is the usual one

$$L(\mathcal{A}) = \{w \mid \mathcal{A} \text{ accepts } w\}$$

whereas the second, $TL(\mathcal{A})$, is slightly unusual

$$TL(\mathcal{A}) = \{wl \mid \mathcal{A} \text{ accepts } w \text{ in an } l\text{-labeled state }\} \cup \{w\Omega \mid \mathcal{A} \text{ accepts } w\}$$

For example, if $\mathcal{A}$ is the automaton



then $L(\mathcal{A}) = \{\varepsilon, a, a[\,], b\}$ and $TL(\mathcal{A}) = \{\Omega, \pi, a\Omega, a*, a[\,]\Omega, b\Omega\}$.

We now provide the correspondence between (normalized) type equations

and typical automata.

Definition 3.1: Let T be a type defined by the equations $\mathcal{E}$. Then AUTO$(T, \mathcal{E})$ is the typical automata where

- the alphabet is the union of $\{\Omega, *, \pi, \alpha_i, [\,]\}$ with the set of component names of partial products mentioned in $\mathcal{E}$.
- the states are exactly the variables in $\mathcal{E}$.
- states are labeled with the coarse type of the right-hand side of the corresponding type variable.
- the initial state is T.
- all states accept.
- equations in $\mathcal{E}$ give rise to transitions in the following manner
  1) **Type** $A = *B$ yields the transition $(A, [\,]) \rightarrow B$.
  2) **Type** $A = (n_i : B_i)$ yields the transitions $(A, n_i) \rightarrow B_i$.
  3) **Type** $A = \alpha_i$ yields no transitions.
  4) **Type** $A = \Omega$ yields no transitions.

Definition 3.2: Let $\mathcal{A}$ be a typical automata. Then TYPE$(\mathcal{A})$ is the pair $(T, \mathcal{E})$ where

- there is a variable for each state in $\mathcal{A}$.
- the initial state corresponds to T.
- equations are generated in the following manner
  1) if A has a $*$-label and $(A, [\,]) \rightarrow B$, then we get the equation **Type** $A = *B$.
  2) if A has a π-label and the A-transitions are $(A, n_i) \rightarrow B_i$, then we get the equation **Type** $A = (n_i : B_i)$.
  3) if A has an α_i-label, then we get the equation **Type** $A = \alpha_i$.
  4) if A has an Ω-label, then we get the equation **Type** $A = \Omega$.

The following two lemmas and theorem show that questions concerning the ordering of types can be replaced by questions concerning the languages associated with the corresponding typical automata.

Lemma 3.3:

$A \preceq B$ *iff* $add(A) \subseteq add(B) \wedge \forall \gamma \in add(A) : coarse(A \downarrow \gamma) \preceq coarse(B \downarrow \gamma)$

Proof:

Only if:
We first show that if $A \preceq B$ then $add(A) \subseteq add(B)$. Let $\gamma \in add(A)$. We proceed by induction in the length of γ. If this length is zero, then the result is immediate. If $\gamma = n_i\gamma'$ then both A and B are partial products with an n_i-component of type respectively A_i and B_i, where $A_i \preceq B_i$. Now, $\gamma' \in add(A_i)$ and by induction $\gamma' \in add(B_i)$, so $\gamma \in add(B)$. Similarly, if $\gamma = []\gamma'$.

We complete the result by induction in the length of $\gamma \in add(A)$. If this is zero, then $A \downarrow \gamma = A$ and $B \downarrow \gamma = B$. We must then show $coarse(A) \preceq coarse(B)$, which follows from $A \preceq B$. If $\gamma = []\gamma'$ then $A = *A'$ and $B = *B'$ where $A' \preceq B'$. By induction, $coarse(A' \downarrow \gamma') \preceq coarse(B' \downarrow \gamma')$ which is the same as $coarse(A \downarrow \gamma) \preceq coarse(B \downarrow \gamma)$. Similarly if $\gamma = n_i\gamma'$.

If:
Look at any finite $A_0 \sqsubseteq A$. By transitivity we have that $add(A_0) \subseteq add(B)$ and that $coarse(A_0 \downarrow \gamma) \preceq coarse(B \downarrow \gamma)$. We now show by induction in A_0 that $A_0 \preceq_0 B$. If $A_0 = \Omega$, then we are done. If $A_0 = \alpha_i$ then $B = \alpha_i$, since their coarse types must match. For the same reason, if $A_0 = *A'$ then $B = *B'$; also, we see that $add(A') \subseteq add(B')$ and $\forall \gamma \in add(A') : coarse(A' \downarrow \gamma) \preceq coarse(B' \downarrow \gamma)$. By induction, we have that $A' \preceq_0 B'$, so $A_0 \preceq_0 B$. Similarly, if A_0 is a partial product. We conclude that $A_0 \preceq_0 B$. Since this is true for all finite $A_0 \sqsubseteq A$, it follows that $A \preceq B$. □

Lemma 3.4: If $\mathcal{A}$ is a typical automaton then

$$TL(\mathcal{A}) = \bigcup_{\gamma \in add(\text{TYPE}(\mathcal{A}))} \{\gamma\Omega, \gamma\, coarse(\text{TYPE}(\mathcal{A}) \downarrow \gamma)\}$$

Proof: Both inclusions follow from easy inductions in the length of γ. □

Theorem 3.5: AUTO and TYPE are order-preserving inverses, i.e.

1) $\text{AUTO} \circ \text{TYPE}(\mathcal{A}) = \mathcal{A}$
2) $\text{TYPE} \circ \text{AUTO}(T, \mathcal{E}) \approx (T, \mathcal{E})$
3) $\text{TYPE}(\mathcal{A}) \preceq \text{TYPE}(\mathcal{B})$ *iff* $TL(\mathcal{A}) \subseteq TL(\mathcal{B})$

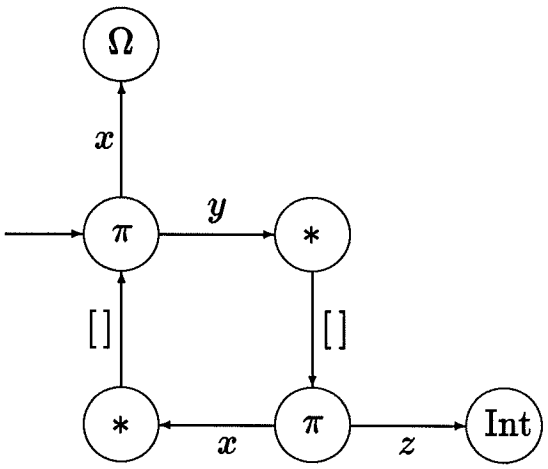
Proof: 1) and 2) follows from inspection of definitions 3.1 and 3.2. To prove 3) we employ lemma 3.4. Clearly, if $TL(\mathcal{A}) \subseteq TL(\mathcal{B})$, then $add(\text{TYPE}(\mathcal{A})) \subseteq add(\text{TYPE}(\mathcal{B}))$. Also, for any $\gamma \in add(\text{TYPE}(\mathcal{A}))$, we have $coarse(\text{TYPE}(\mathcal{A}) \downarrow \gamma) \in \{\Omega, coarse(\text{TYPE}(\mathcal{B}) \downarrow \gamma)\}$. It follows that $coarse(\text{TYPE}(\mathcal{A}) \downarrow \gamma) \preceq coarse(\text{TYPE}(\mathcal{B}) \downarrow \gamma)$. It is equally clear that these properties are sufficient to ensure $TL(\mathcal{A}) \subseteq TL(\mathcal{B})$. Now, the result follows from lemma 3.3. □

In fact, with an appropriate representation both AUTO and TYPE can be implemented as the identity function. Hence, the point of these definitions is to emphasize the faithfulness of this conceptual recasting.

As an example, let $\mathcal{E}$ be the following type equations

Type $A = (x : B, y : C)$
Type $B = \Omega$
Type $C = *D$
Type $D = (x : E, z : F)$
Type $E = *A$
Type $F = \text{Int}$

Then $\text{AUTO}(A, \mathcal{E})$ is the automaton



4 The Algorithms

We can now present the translation of computational problems for type equations into automata analogues. We shall make several references to standard algorithms for finite state automata, all of which are described in e.g. [1]. Note that for every typical automaton $\mathcal{A}$ there is an ordinary automaton $\mathcal{A}'$, with at most three times the size, such that $TL(\mathcal{A}) = L(\mathcal{A}')$.

4.1 Equivalence is Equality

This first result is quite well-known for type systems [2,3] and for infinite trees in general [6]. However, type equations are usually related to *graphs* rather than to automata.

Corollary 4.1: Type equivalence translates to language equality, i.e.

$$(T_1, \mathcal{E}_1) \approx (T_2, \mathcal{E}_2) \;\; \textit{iff} \;\; TL(\textsc{auto}(T_1, \mathcal{E}_1)) = TL(\textsc{auto}(T_2, \mathcal{E}_2))$$

Proof: Follows from theorem 3.5 and anti-symmetry of $\preceq$. □

The best known algorithm for language equality runs in time $O(n\alpha(n))$ for two automata of size n.[3]

[3]Here $\alpha(n)$ is the inverse of *Ackermann's function* [1].

4.2 Ordering is Inclusion

We note that this next result does not hold for type orderings that rely on coercions of values [3,7]. In these situations the inclusions of component names go in opposite directions for type *sums* and *products*, which disables this automata technique. Hence, partial products seem to enable a more natural type ordering.

Corollary 4.2: Type ordering translates to language inclusion, i.e.

$$(T_1, \mathcal{E}_1) \preceq (T_2, \mathcal{E}_2) \;\; \textit{iff} \;\; TL(\textsc{auto}(T_1, \mathcal{E}_1)) \subseteq TL(\textsc{auto}(T_2, \mathcal{E}_2))$$

Proof: Follows directly from theorem 3.5. □

The best known algorithm for language inclusion runs in time $O(n^2)$ for two automata of size n.

4.3 Specialization is Intersection

This result is interesting since it demonstrates in what sense specialization can be thought of as an *intersection*. It is, of course, *not* the case that specialization yields the intersection of *values*.

Corollary 4.3: Type specialization translates to language intersection, i.e.

$$TL(\textsc{auto}((T_1, \mathcal{E}_1) \sqcap (T_2, \mathcal{E}_2))) = TL(\textsc{auto}(T_1, \mathcal{E}_1)) \cap TL(\textsc{auto}(T_2, \mathcal{E}_2))$$

Proof: Follows from theorem 3.5. □

The algorithm can be obtained as a special case of the general algorithm in section 5. The running time can in this case be observed to be $O(n^2)$ for two automata of size n. This is optimal in the case where we want to construct the specialized type since it may have size $\Omega(n^2)$.

4.4 Inheritance is Union

In analogy with specialization, this result shows how inheritance can be interpreted as a *union*. Again, inheritance does *not* yield the union of *values*.

Corollary 4.4: Type inheritance translates to language union, i.e. when it exists, then

$$TL(\text{AUTO}((T_1, \mathcal{E}_1) \sqcup (T_2, \mathcal{E}_2))) = TL(\text{AUTO}(T_1, \mathcal{E}_1)) \cup TL(\text{AUTO}(T_2, \mathcal{E}_2))$$

Proof: Follows from theorem 3.5. □

The algorithm can be obtained as a special case of the general algorithm in section 5. The running time can in this case be observed to be $O(n^2)$ for two automata of size n. This is optimal in the same sense as above.

4.5 Hierarchical Consistency is Restriction

This final result is less intuitive but all the more needed to secure an implementation.

Definition 4.5: Let $\mathcal{A}$ be an automaton. If $x \in L(\mathcal{A})$, then $\mathcal{A}(x)$ is identical to $\mathcal{A}$ except that the initial state is now the state that accepts x; this is well-defined since $\mathcal{A}$ is deterministic. We now define an equivalence relation $\sim$ on $L(\mathcal{A})$, where

$$x \sim_{\mathcal{A}} y \;\; \textit{iff} \;\; TL(\mathcal{A}(x)) = TL(\mathcal{A}(y))$$

Definition 4.6: Let $\mathcal{A}$ and $\mathcal{B}$ be automata; $\mathcal{A}$ *restricts* $\mathcal{B}$ *iff*

$$TL(\mathcal{A}) \subseteq TL(\mathcal{B}) \;\wedge\; \forall x \in L(\mathcal{A}) : \; [x]_{\mathcal{A}} \subseteq [x]_{\mathcal{B}}$$

where $[x]_{\mathcal{C}}$ is the equivalence class of x under $\sim_{\mathcal{C}}$.

Theorem 4.7: Hierarchical consistency translates to restriction, i.e. $(T_1, \mathcal{E}_1)$ and $(T_2, \mathcal{E}_2)$ are consistent *iff* $\text{AUTO}(T_1, \mathcal{E}_1)$ restricts $\text{AUTO}(T_2, \mathcal{E}_2)$.

Proof: Let $\mathcal{A}_i$ equal AUTO$(T_i, \mathcal{E}_i)$. By definition we have

$$\begin{aligned}
& \forall x, y \in add(T_1, \mathcal{E}_1) : (T_1, \mathcal{E}_1)\downarrow x = (T_1, \mathcal{E}_1)\downarrow y \Rightarrow (T_2, \mathcal{E}_2)\downarrow x = (T_2, \mathcal{E}_2)\downarrow y \\
\Leftrightarrow\ & \forall x, y \in L(\mathcal{A}_1) : x \sim_{\mathcal{A}_1} y \Rightarrow x \sim_{\mathcal{A}_2} y \\
\Leftrightarrow\ & \forall x \in L(\mathcal{A}_1) : [x]_{\mathcal{A}_1} \subseteq [x]_{\mathcal{A}_2}
\end{aligned}$$

which, together with corollary 4.2, yields the desired result. □

We must now check restriction of two automata $\mathcal{A}$ and $\mathcal{B}$ of size n. We assume without loss of generality that they are both minimal; otherwise, we expend time $O(n \log n)$ to achieve this. In minimal automata each equivalence class is represented by a single state.

Together with the language inclusion, we must thus require that any two words that reach the same state in $\mathcal{A}$ will also reach the same state in $\mathcal{B}$. Consider the relation

$$F = \{(q_{\mathcal{A}}, q_{\mathcal{B}}) \mid \exists x \in L(\mathcal{A}) : x \text{ is accepted in state } q_{\mathcal{A}} \ (q_{\mathcal{B}}) \text{ in } \mathcal{A} \ (\mathcal{B})\}$$

The requirement that any two words that reach the same state in $\mathcal{A}$ must also reach the same state in $\mathcal{B}$, is equivalent to the requirement that F is a *function* with domain the state set of $\mathcal{A}$. Since the (reflexive and transitive closure of) the transition function of a finite automaton is a homomorphism w.r.t. concatenation, we need only check functionality of F for short representatives for the equivalence classes $[x]_{\mathcal{A}}$. This can be done in linear time by a simple traversal of the graph of $\mathcal{A}$. Furthermore, it follows from corollary 4.2 and lemma 3.3 that language inclusion can be checked simultaneously by verifying that the labels $l_{\mathcal{A}}, l_{\mathcal{B}}$ (the coarse types) associated with a pair of states in F must satisfy $l_{\mathcal{A}} \preceq l_{\mathcal{B}}$.

5 Extended Type Equations

It is very convenient to allow ⊔- and ⊓-operators to appear on the right-hand sides of type equations. For example, a simple prefixing can be expressed as

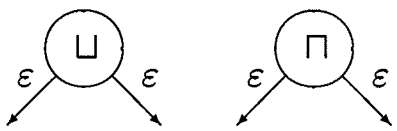
$$\textbf{Type}\ \text{New} = (x : T) \sqcup \text{Old}$$

where x is the extra field of type T. The idea behind such extended equations is that the compiler computes the indicated types, if they exist. In general, the computation of arbitrary $\sqcup$'s and $\sqcap$'s of already defined types is an acknowledged, useful tool for structuring programs, and with these limitations the situation is very easily resolved. However, it is a natural choice to allow the extended equations to be recursive, too. The denoted types are then the smallest ones, if any, that satisfy the equations.

It turns out that we can always decide whether such extended equations denote types and, if so, transform them into equivalent basic equations.

5.1 Alternating Automata

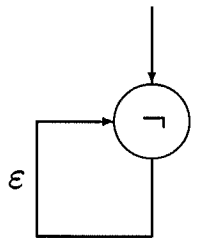
The alternating finite automata were introduced in [4]. A modification of these are needed for our purposes. We extend the notion of a typical automata to include two special kinds of nodes



The language of a $\sqcup$-node is the union of those of its descendants; the language of a $\sqcap$-node is the intersection of those of its descendants. The *basic* states of an extended automaton are the ones that are neither $\sqcup$- nor $\sqcap$-nodes.

It is clear that we now have a perfect relationship between normalized, extended type equations and extended typical automata. The problem of resolving extended type equations is reduced to removing $\sqcup$- and $\sqcap$-nodes from the corresponding automaton.

In [4] it is shown that an alternating automaton with n states can always be transformed into an equivalent deterministic automaton with at most 2^{2^n} states. We follow closely the ideas presented there; however, we give a concrete algorithm. Furthermore, we solve a slightly different problem, since we allow ε-transitions but disallow non-monotone connectives; in [4] the opposite choice has been made. The problem with allowing both at the same time is illustrated by the following self-contradictory automaton



for which it is awkward to define an acceptance criterion and, hence, a language. In fact, it is not very intuitive what the language should be even if the transition consumed an alphabet symbol.

5.2 The Algorithm

Our set-up has the nice property that it permits an algorithm that can be viewed as a very direct generalization of the usual algorithm for non-deterministic automata [1].

The states of the deterministic automaton will correspond to monotone *propositions* with the basic states of the extended automaton as variables. Such propositions can be viewed as generalizations of finite sets; in particular, a finite set $\{s_1, s_2, \ldots, s_n\}$ corresponds to the proposition $s_1 \sqcup s_2 \sqcup \ldots \sqcup s_n$.

In analogy with the non-deterministic case, we next define a function *reach* that maps states in the old automaton to states in the new automaton. In the non-deterministic case, $reach(S)$ is the set of states that can be reached from S via ε-transitions. The natural generalization of this is to view each $reach(S)$ as a variable in a collection of mutually recursive equations, where we define

- $reach(S) = S$, if S is a basic node.
- $reach(S) = reach(S_1) \sqcup reach(S_2)$, if S is a $\sqcup$-node with descendants S_1 and S_2.
- $reach(S) = reach(S_1) \sqcap reach(S_2)$, if S is a $\sqcap$-node with descendants S_1 and S_2.

The monotone propositions can be partially ordered as follows

$$P \sqsubseteq Q \;\; \textit{iff} \;\; \forall \sigma : \sigma \models P \;\Rightarrow \sigma \models Q$$

In the degenerate case of sets, this ordering is just inclusion. We have a complete partial order, since *false* is the bottom element and all chains are finite. Hence, recursive equations, such as the above, have unique least solutions obtained by iterating from the bottom element in the usual manner. This defines the *reach*-function.

The initial state of the new automaton is obtained as $reach(I)$, where I is the initial state in the old automaton.

We now have a minor special case, which is the price we must pay for the convenience of working with partial automata. If the new initial state is *false*, then the new automaton is an Ω-labeled singleton. This is the correct choice since Ω is evidently the least solution to the corresponding type equations. Henceforth, *false* will not appear as a state in the new automaton. The choice of *least* solutions to the *reach*-equations allows us to avoid *true* as a state, too.

Lemma 5.1: The proposition *true* is never the least solution to any recursive equation.
Proof: The solutions are obtained as a finite number of iterations of monotone functions starting from *false*, which is not identical to *true*. Monotone functions preserve this property. If $\sigma_1 \not\models P_1$ and $\sigma_2 \not\models P_2$, then $\sigma_1 \not\models P_1 \sqcap P_2$ and $\sigma_1 \sqcap \sigma_2 \not\models P_1 \sqcup P_2$. □

Thus, the propositions corresponding to states can all be represented as $\sqcap$,$\sqcup$-expressions over basic states. This is used for structural induction in later proofs.

The new initial state is placed in a list of *unfinished* new states. The algorithm proceeds by selecting an unfinished state, determining its transitions, and (perhaps) adding more unfinished states to the list. All the states that are constructed in this manner will be accept states in the resulting partial automaton.

Each step is performed as follows: The selected unfinished state is S. For each alphabet symbol a we determine $trans(S, a)$ as follows

- $trans(S, a) = reach(S')$, if S is a basic state and $(S, a) \rightarrow S'$ is a transition in the old automaton.
- $trans(S, a) = false$, if S is a basic state without a-transitions in the

old automaton.

- $trans(S_1 \sqcup S_2, a) = trans(S_1, a) \sqcup trans(S_2, a)$.
- $trans(S_1 \sqcap S_2, a) = trans(S_1, a) \sqcap trans(S_2, a)$.

Since we want a partial automaton, we add no transitions if $trans(S, a) = \mathit{false}$; otherwise, we include the transition $(S, a) \rightarrow trans(S, a)$ and add the state $trans(S, a)$ to the list, if we have not seen it before. After this process, the state S is finished and can be removed from the list. Note that we identify equivalent propositions.

This process will eventually terminate since we can only generate finitely many new states. There are 2^{2^n} n-variable propositions and slightly fewer ($2^{2^{O(n)}}$) monotone ones. In [4] it is shown that for some alternating automata all of these states are necessary, too.

To get a typical automaton we must further determine labels of the new states. This can be done by placing the proposition in *disjunctive* normal form and computing $label(S)$ as follows

- $label(S)$ = the old label of S, if it is a basic state.
- $label(S_1 \sqcup S_2) = label(S_1) \sqcup label(S_2)$.
- $label(S_1 \sqcap S_2) = label(S_1) \sqcap label(S_2)$.

Here we interpret $\sqcup$ as least upper bound and $\sqcap$ as greatest lower bound in the partial order of coarse types. Since we do not have a *lattice* of coarse types such computations may sometimes fail. This is exactly why a collection of extended type equations need not always denote types.

Lemma 5.2: If all new labels exist, then the new automaton is typical.
Proof: We proceed by induction in the structure of a proposition S.

- If S is a singleton, then it has the same label and outgoing transition arcs as in the old automaton.
- If $S = S_1 \sqcup S_2$, then we have two cases: 1) S_1 and S_2 have the same label. In this case, S gets that label, too, and by induction S_1 and S_2 are both legal. The transitions from S is the union of those from S_1 and S_2, and for all choices of labels this will preserve legality of the state. 2) S_1 and S_2 have different labels. In this case, at least

one of these labels is Ω, so S inherits legal label and transitions from the other.

- If $S = S_1 \sqcap S_2$, then we have two cases: 1) S_1 and S_2 have the same label. In this case, S gets that label too, and by induction S_1 and S_2 are both legal. The transitions from S is the intersection of those from S_1 and S_2, and for all choices of labels this will preserve legality of the state. 2) S_1 and S_2 have different labels. In this case, S will have label Ω and no transitions, since the intersection of transitions from S_1 and S_2 must be empty.

In all cases, we have shown that the new state is legal. □

Lemma 5.3: If all labels exist, then the new automaton has the same *TL*-language as the original.
Proof: Let w be a word and l be a label. We show by induction that: wl is in the language of state S in the old automaton *iff* wl is in the language of state $reach(S)$ in the new automaton. The induction is performed in lexicographically ordered pairs of the form (n, k), where n is the length of w and k is the height of the proposition $reach(S)$.

- *The base case* $(0, k)$*:* In both automata $w = \varepsilon$ is accepted since all states accept. If $l = \Omega$, then the result is trivial since $w\Omega$ is in the language of any typical automaton that accepts w. If $l \neq \Omega$ and all new labels exist, then it follows from lemma 5.2 that all old labels reachable from S are identical and will equal the label of $reach(S)$.

- *The case* $(n+1, 0)$ *becomes* (n, k)*:* If $w = aw'$ and S is a basic node, then the old automaton will have exactly one transition $(S, a) \rightarrow S'$ where $w'l$ is in the language of S'. Since $reach(S) = S$, the new automaton has a similar unique transition $(S, a) \rightarrow reach(S')$. By induction hypothesis it follows that $w'l$ is in the language of S' in the old automaton *iff* $w'l$ is in the language of $reach(S')$ in the new automaton.

- *The case* $(n, k+1)$ *becomes* (n, k)*:* We have that wl is in the language of $S = S_1 \sqcup S_2$ *iff* it is in the language of S_1 or in the language of S_2. By induction hypothesis, this is true *iff* it is in the languages of $reach(S_1)$ or $reach(S_2)$. But this is by definition the language of $reach(S_1 \sqcup S_2)$. Similarly if $S = S_1 \sqcap S_2$.

We have covered sufficiently many cases for the induction to work. By considering the initial state the result follows. □

Lemma 5.4: If some label does not exist, then no typical automaton has the same language as the original.
Proof: Assume that state S is in disjunctive normal form and has no label. Since conjunctions of labels always exist, we have that $S = S_1 \sqcup S_2$, where $l_1 = label(S_1) \neq \Omega$, $l_2 = label(S_2) \neq \Omega$, and $l_1 \neq l_2$. Let w be any word accepted by S. Then wl_1 and wl_2 are both in the language of the old automaton. By definition, no language defined by a typical automaton can have two words with this property. □

Theorem 5.5: The algorithm correctly resolves extended type equations.
Proof: If the new labels exist, then from lemma 5.2 we know that the new automaton is typical, and from lemma 5.3 we know that it has the same language as the old automaton. If some new labels do not exist, then the new automaton will not work, but lemma 5.4 tells us that in this case no typical automaton exists. □

Some special cases are quite noteworthy.

1) If the original has only $\sqcup$-nodes (or $\sqcap$-nodes), then the construction will have at most 2^n states, since the propositions all look like $p \sqcup q \sqcup \ldots \sqcup r$ (or $p \sqcap q \sqcap \ldots \sqcap r$).

2) If the original has only a single $\sqcup$-node (or $\sqcap$-node) combining two independent subautomata, then the construction will only have n^2 states, since the propositions will all look like $p \sqcup q$ (or $p \sqcap q$).

3) If the original has only $\sqcap$-nodes, then the new labels will always exist.

As promised, case 2) covers the situations in sections 4.3 and 4.4.

6 Conclusion

We have demonstrated the very strong connections between our type system and finite automata. This provided efficient algorithms, as well as a deeper intuition about the types.

The generality of the extended type equations may lead to two slightly incongruous views.

- The $2^{2^{O(n)}}$-succinctness shows that this is a very powerful mechanism for specifying types.
- The $2^{2^{O(n)}}$-explosion shows that one has little chance of knowing which types one has specified.

However, they have many uses apart from specifications by the programmer. For example, type inference of an implicitly typed version of the underlying language can be transformed into satisfiability of system of recursive $\sqcup$-equations that is automatically generated from the program text [11].

References

[1] **Aho, A.V. & Ullman, J.D.** "The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing.", Prentice-Hall 1972.

[2] **Cardelli, L.** "Typeful Programming" *DEC Research Report 45*, 1989.

[3] **Cardelli, L. & Wegner, P.** "On Understanding Types, Data Abstraction, and Polymorphism" in *Computing Surveys, Vol 17 No 4*, ACM 1985.

[4] **Chandra, A.K., Kozen, D.C. & Stockmeyer, L.J.** "Alternation", in *JACM Vol 28 No 1*, ACM 1981.

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

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

[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*, Springer-Verlag, 1990.

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

[11] **Schwartzbach, M.I.** "Type Inference with Inequalities". In preparation. Aarhus University, 1990.