A Framework for Ad-Hoc Type Inference
(Extended Abstract)

Hosein Askari, Ole I. Hougaard[*], Michael I. Schwartzbach
{hosask,hougaard,mis}@daimi.aau.dk
BRICS[†]
Department of Computer Science
University of Aarhus
8000 Aarhus C, Denmark

Abstract

Languages based on variations of the lambda calculus are designed to permit the slick, unification-based technique for type inference, which is by now a well-established discipline.

Other widely used languages have been created less by design and more by coincidence and compromise. It seems therefore that the question of type inference for such languages could be infeasible or should at least permit only ad-hoc solutions.

In this paper we argue the existence of a uniform framework for developing type inference algorithms, even for seemingly ad-hoc languages. This framework provides useful generalizations of well-known concepts related to type inference. It is, however, by no means a universal algorithm. In fact, the essential algorithmic problem must generally be solved from scratch.

We demonstrate the viability of our approach by developing a type inference algorithm for a full version of the Turbo Pascal language.

1 Introduction

The concept of *type inference* has been firmly established as a strong quality in a programming language. It has also traditionally been associated with *functional* programming languages, such as ML, which are based on variations of the lambda calculus and hence allow the slick, unification-based technique for polymorphic type inference.

However, type inference may certainly be just as relevant for ad-hoc languages, which have been created less by design and more by coincidence and

[*]Direct communications to this author
[†]Basic Research in Computer Science, Centre of the Danish Research Foundation

compromise. Clearly, if the type constraints are rather obscure, then automatic support will be correspondingly more useful.

It can naturally be argued that these ad-hoc languages should be eliminated and replaced by simpler, theoretically well-founded alternatives. For the purposes of this paper, we will merely observe that this process is sufficiently slow to justify an interest in building tools for these existing languages. Besides, the algorithmic challenge should certainly be met.

Note that if a language has a notion of type checking, then the problem of type inference is perfectly well-defined: Given a program (partially) without type annotations, does there exist a full annotation that allows the program to be type checked? This is simply an algorithmic problem, which one may attempt to solve.

It could be expected that the type inference problem for ad-hoc languages will turn out to be infeasible or at best permit only ad-hoc solutions. Certainly, it does not seem possible to generalize the unification-based approach.

In this paper we argue the existence of a uniform framework for developing type inference algorithms that to a large extent allows the ad-hoc nature of a programming language to be factored out. This framework should be viewed as a set of formalized rules-of-thumb that may guide the development of the algorithm; by no means do we present a universal algorithm.

This paper has two main parts, where we first provide useful generalizations of well-known concepts related to type inference and then show how to adapt them to ad-hoc situations.

In Section 2 we review the well-known unification-based technique for type inference in ML-style languages. For each fundamental concept do we provide suitable generalizations that enable the inclusion of ad-hoc features. The type domain is generalized from simple regular terms to term automata that capture the essence of name equivalence. The type constraints are generalized from equalities to more arbitrary relations respecting equivalence preserving homomorphisms. Based on these generalizations we propose a technique for representing solutions to type constraints, which may suggest the structure of a type inference algorithm. The essential algorithmic problem must however be solved from scratch. Finally, we show how polymorphism may be introduced through code expansion even in the likely absence of principal type schemes.

In Section 3 we demonstrate the viability of our framework by developing a polymorphic type inference algorithm for a full version of the Turbo Pascal language. It is safe to say that this language contains no special structure that will give us an unfair advantage when applying our framework. As predicted, we must solve an algorithmic problem specific to this application. However, we do this in a more general context that should cover several variations of the problem. The Turbo Pascal type inference algorithm has been implemented as a working preprocessor and suggests some interesting possibilities.

2 Generalized Concepts

We now recapitulate the well-known technique for performing type inference in ML-style languages. While presenting the familiar concepts, we will also introduce the generalizations necessary to handle more ad-hoc languages.

2.1 Type Domain

The types of core-ML are the expressions generated by the grammar below:

$$\begin{aligned} T_{\mathrm{ML}} ::= \quad & \alpha \mid \beta \mid \gamma \mid \ldots \\ & \mid T_{\mathrm{ML}} \to T_{\mathrm{ML}} \end{aligned}$$

where $\alpha, \beta, \gamma, \ldots$ are type variables. Such type expressions can be described as finite *terms* as defined by Courcelle [2]. Here we adopt the notation of Kozen [5].

Definition 2.1 *Let ω denote the set of natural numbers and let Σ be a ranked alphabet. A (Σ-)*term *is a partial function $t : \omega^* \to \Sigma$ whose domain is nonempty, prefix-closed, and respects arities in the sense that if $t(\alpha)$ is defined then*

$$\{i \mid t(\alpha i) \textit{ is defined}\} = \{1, 2, \ldots, \mathit{arity}(t(\alpha))\}$$

A term is regular *if it has only finitely many subterms up to isomorphism.*

In the case of ML we have $\Sigma_{\mathrm{ML}} = \{\to, \alpha, \beta, \ldots\}$, $\mathit{arity}(\to) = 2$, and $\mathit{arity}(\alpha) = \mathit{arity}(\beta) = \cdots = 0$.

From [2] we adopt the notion of a *subtree* of t rooted at a node α, t/α, defined as $t/\alpha(\beta) = t(\alpha\beta)$. We will not distinguish between the singleton tree $\epsilon \mapsto l$ and the label l.

In our framework we represent types as finite *term automata* taken from Kozen [5].

Definition 2.2 *A* pure (Σ-)term automaton *is a tuple*

$$M = (Q, \Sigma, l, \delta)$$

where:

- *Q is a finite set of* states.
- *Σ is a ranked alphabet.*
- *$l : Q \to \Sigma$ is a labeling.*
- *$\delta : Q \times \omega \to Q$ is a partial function such that for all $q \in Q$,*

$$\{i \mid \delta(q, i) \textit{ is defined}\} = \{1, 2, \ldots, \mathit{arity}(l(q))\}$$

Note that the automata used here differs from the ones in [5] in that they are finite. We will make another addition to the definition.

Definition 2.3 *A* $(\Sigma\text{-})$*term automaton with* initial state *is a tuple*

$$M = (Q, q_0, \Sigma, l, \delta)$$

where (Q, Σ, l, δ) *is a pure* $(\Sigma\text{-})$*term automaton, and* $q_0 \in Q$ *is an initial state. We write* $A(\Sigma)$ *for the set of* Σ*-term automata with initial state.*

In both kinds of term automata the function δ extends into a partial function $\hat{\delta} : Q \times \omega^* \to Q$ according to the inductive definition:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q \\ \hat{\delta}(q, \alpha i) &= \delta(\hat{\delta}(q, \alpha), i) \end{aligned}$$

with the understanding that the last expression is undefined if $\hat{\delta}(q, \alpha)$ is undefined.

Let $M = (Q, q_0, \Sigma, \delta, l)$ or $M = (Q, \Sigma, \delta, l)$ and $q \in Q$. The *subautomaton* of M rooted at q is

$$M[q] = (Q, q, \Sigma, \delta, l)$$

For $M = (Q, q_0, \Sigma, \delta, l)$ we also define M/α defined as

$$M/\alpha = M[\hat{\delta}(q_0, \alpha)]$$

For any automaton $M = (Q, q_0, \Sigma, \delta, l)$ the partial function defined by:

$$t^M(\alpha) = l(\hat{\delta}(q_0, \alpha))$$

is a regular Σ-term. For $q \in Q$ we write $t_q^M = t^{M[q]}$. Similarly for pure term automata. It is easy to see that $t^{M/\alpha} = t^M/\alpha$.

Conversely, every regular term corresponds to t^M for some term automaton, M, with initial state [7, lemma 8]. So we have the following.

Proposition 2.1 *The set of regular* Σ*-terms is isomorphic to the set of* Σ*-term automata with initial state modulo term equality.*

We can thus consider the ML types to be Σ_{ML}-term automata with initial state.

Let us now consider the set $\{t^M \mid M \in A(\Sigma)\}$. Clearly, it forms a free Σ-algebra. But there are type domains with restrictions on the types to which an operator can be applied. In order to deal with such situations we apply restrictions to the terms of the automata in the type domain.

Definition 2.4 *Let* Σ *be a ranked alphabet, and let* L *be a regular tree language. The class of legal automata with respect to* Σ *and* L *is*

$$\mathcal{A}(\Sigma, L) = \{M \in A(\Sigma) \mid t^M \in L\}$$

In ML the type domain is then $\mathcal{A}(\Sigma_{\mathrm{ML}}, \{t^M \mid M \in A(\Sigma_{\mathrm{ML}})\}) = A(\Sigma_{ML})$

2.2 Type Equivalence

The notion of type equivalence a central concept in a language definition, reflecting the essential properties of types. Most languages incorporate either *structural equivalence* or *name equivalence*. In the following we will generalize structural equivalence in a way that provides a simple framework for dealing with both equivalences.

It follows from the previous subsection that structural equivalence between term automata is simply equality between their terms. It is clear that for any two terms, t and t', we have that

$$t = t' \Leftrightarrow t(\epsilon) = t'(\epsilon) \wedge \forall i \in \{1, 2, \ldots, arity(t(\epsilon))\} : t/i = t'/i$$

In the case of finite trees, this is actually an unambiguous definition of the relation $=$. In the more general case of regular trees, there is a number of relations having the same property as $=$. In fact, $=$ is the most general of the equivalence relations with the above property. We would expect that the equivalence relation $t^M = t^{M'}$ would have a similar property, and it does as stated in the lemma below.

Lemma 2.2 *The equivalence relation $t^M = t^{M'}$ is the most general of all equivalence relations, $\approx$, with the property that for all M and M':*

$$M \approx M' \Leftrightarrow t^M(\epsilon) = t^{M'}(\epsilon) \wedge \forall i \in \{1, \ldots, arity(t^M(\epsilon))\} : M/i \approx M'/i$$

That is for any $\approx$ with the above property $M \approx M' \Rightarrow t^M = t^{M'}$.

The property in the above lemma implies that every label in Σ is equivalence preserving. Now, let us consider a situation where only *some* of the labels have this property. Choose a subset $S \subseteq \Sigma$. We want the above to apply only if the label is in S. If we write

$$M \approx M' \Leftrightarrow t^M(\epsilon) = t^{M'}(\epsilon) \in S \wedge \forall i \in \{1, \ldots, arity(t^M(\epsilon))\} : M/i \approx M'/i$$

then we wind up with a relation that is not necessarily reflexive. Hence we need to take the reflexive closure. The definition is as follows.

Definition 2.5 *Let $S \subseteq \Sigma$. The equivalence relation $\approx_S$ is the most general among the equivalence relations, $\approx$, with the property that for all M and M':*

$$\begin{array}{l} M \approx M' \\ \Updownarrow \\ (t^M(\epsilon) = t^{M'}(\epsilon) \in S \wedge \forall i \in \{1, \ldots, arity(t^M(\epsilon))\} : M/i \approx M'/i) \vee (M = M') \end{array}$$

Definition 2.5 is unpleasantly indirect. The proposition below gives a more direct definition of $\approx_S$.

Proposition 2.3 *$\approx_S$ is the relation:*

$$M \approx_S M' \Leftrightarrow \forall \alpha \in \omega^* : t^M(\alpha) = t^{M'}(\alpha) \in S \vee M/\alpha = M'/\alpha$$

with the understanding that undefined equals undefined.

2.3 Type Variables

A given term in core-ML gives rise to a finite collection of *type variables*, in principle one for each subterm or parse tree node. However, the type variables corresponding to different occurrences of the same identifier are identified. We shall use the notation $[\![E]\!]$ for the type variable corresponding to the subterm E. We denote by $\mathcal{V}$ the set of type variables.

2.4 Type Constraints

In general, a type constraint is a relation on types. We can, however, be more specific than simply to allow arbitrary relations. Even seemingly ad-hoc constraints share some common structure.

Given a term in core-ML, we generate the required constraints inductively in the syntax as follows. For a subterm of the form $\lambda x.E$ we have the constraint

$$[\![\lambda x.E]\!] = [\![x]\!] \rightarrow [\![E]\!],$$

and for a subterm of the form $E_1 E_2$ the constraint

$$[\![E_1]\!] = [\![E_2]\!] \rightarrow [\![E_1 E_2]\!].$$

The type constraint $[\![E]\!] = [\![E']\!] \rightarrow [\![E'']\!]$ can be described as follows. If M, M' and M'' are automata for the values of the type variables in $[\![E]\!] = [\![E']\!] \rightarrow [\![E'']\!]$, then we require

$$M = M' \rightarrow M'' \Leftrightarrow (t^M(\epsilon) = \rightarrow) \wedge (M/1 \approx_\Sigma M') \wedge (M/2 \approx_\Sigma M'').$$

Since types are automata, it is possible to define very general constraints. In particular, it is possible to define constraints which are dependent on the actual initial state (or other states) of the automaton, e.g. by demanding that the initial state should be an integer in the range from 1 to 7. We consider constraints of this sort to be an abuse of this framework. In order to avoid it, we demand that all the constraints should be resistant to replacement of the set of states as long as this replacement does not violate the type equivalence. We need a further concept to formalize this.

Definition 2.6 *A* homomorphism *is a function:*

$$f : (Q, q_0, \Sigma, \delta, l) \rightarrow (Q', q_0', \Sigma, \delta', l')$$

where

$$f : Q \rightarrow Q'$$
$$f(q_0) = q_0'$$
$$\delta'(f(q), i) = f(\delta(q), i)$$
$$l'(f(q)) = f(l(q))$$

It is clear that any homomorphism is term preserving. That is, if $f : M \to M'$ is a homomorphism, then $M \approx_{\Sigma} M'$. This means that any relation over terms is resistant to any homomorphism (i.e. $R(M) \Leftrightarrow R(M')$, where M and M' are as above). For general S, though, we need a stronger restriction on the homomorphisms.

Definition 2.7 *An indexed family of homomorphisms:*

$$(f_i : M_i \to M_i')_{i \in I}$$

is $\approx_S$-preserving *iff:*

$$\forall i, j \in I, q \in Q_i, q' \in Q_j : M_i[q] \approx M_j[q'] \Leftrightarrow M_i'[f_i(q)] \approx M_j'[f_j(q')]$$

where $M_i = (Q_i, q_{i0}, \Sigma, \delta_i, l_i)$ *and* $M_i' = (Q_i', q_{i0}', \Sigma, \delta_i', l_i')$.

For an indexed family of homomorphisms, $\Phi = (f_i : M_i \to M_i')_{i \in I}$, we introduce the following notation:

$$\begin{aligned} \Phi(M_i) &= M_i' \\ dom(\Phi) &= \{M_i \mid i \in I\} \\ codom(\Phi) &= \{M_i' \mid i \in I\} \end{aligned}$$

Definition 2.8 *A* constraint alphabet *is a ranked alphabet,* Γ*, of relations, where each* $R \in \Gamma$ *satisfies the following: For all automata* $M_1, \ldots, M_n$*, and for any* $\approx_S$*-preserving family of homomorphisms,* Φ*, such that* $\{M_1, \ldots, M_n\} \subseteq dom(\Phi)$*, we have:*

$$R(M_1, ..., M_n) \Leftrightarrow R(\Phi(M_1), ..., \Phi(M_n))$$

where $n = arity(R)$*. A* constraint *from* Γ *is of the form* $R(v_1, \ldots, v_n)$*, where* $R \in \Gamma$*,* $v_i \in \mathcal{V}$*, and* $n = arity(R)$*.*

In general, the type constraints generated for a given term can be described using simple rules following the syntax. For each production in the context-free grammar, such as $X ::= aYbZc$, a finite number of constraints of the form $R([\![X]\!], [\![Y]\!], [\![Z]\!])$ are generated.

2.5 Solutions

The hardest problem in a type inference arises when trying to find a solution to the set of constraints generated from a specific program. This is often done by at any point to represent the set of solutions to the constraints read so far. We present here a uniform representation of a single solution that may sometimes suggest how to solve the constraints.

A solution to a set of constraints is an assignment of a type to each type variable that occurs in the constraints, such that the constraints are satisfied when the type variables are replaced with the assigned types. We can thus represent a solution by a partial function: $\mathcal{S} : \mathcal{V} \to \mathcal{A}(\Sigma, L)$. We will look at an alternative representation of solutions that are more closely connected to automata, the *base automaton*.

Definition 2.9 *A* base automaton *over a set $\mathcal{V}$ of type variables and an alphabet, Σ, is a pure Σ-term automaton, $M = (Q, \Sigma, \delta, l)$, where Q is a set of non-empty, finite, pairwise disjoint subsets of $\mathcal{V}$. We write $\langle v \rangle_M$ for the element of Q containing v if it exists.*

Definition 2.10 *The solution corresponding to a base automaton $M = (Q, \Sigma, \delta, l)$ is the partial function $\mathcal{S}_M$ defined by $\mathcal{S}_M(v) = (Q, \langle v \rangle_M, \Sigma, \delta, l)$.*

In the following we will show that it is sufficient to look at base automata. Actually, S-minimal base automata can always be found. An automata is S-minimal, if it cannot be smaller and still have the same $\approx_S$ properties as in the definition below.

Definition 2.11 *A base automaton, M, is S-minimal, iff for each automaton M', $\mathcal{S}_{M'}(v) \approx_S \mathcal{S}_{M'}(v') \Leftrightarrow \mathcal{S}_M(v) \approx_S \mathcal{S}_M(v')$ implies that M has fewer states than M'.*

The advantage with S-minimal automata is that the relation $\approx_S$ becomes equality between initial states.

Proposition 2.4 *A base automaton M is minimal, iff $\mathcal{S}_M(v) \approx_S \mathcal{S}_M(v') \Rightarrow \langle v \rangle_M = \langle v' \rangle_M$.*

Proposition 2.5 *Let $\mathcal{S} : \mathcal{V} \to \mathcal{A}(\Sigma, L)$. There exists an S-minimal base automaton, M, and a $\approx_S$-preserving family of homomorphisms, Φ, such that for all $v \in dom(\mathcal{S})$: $\mathcal{S}_M(v) = \Phi(\mathcal{S}(v))$.*

Corollary 2.6 *Let C be a set of constraints. If C has a solution, then there is a minimal base automaton, M, such that $\mathcal{S}_M$ is a solution to C.*

This means that we can restrain ourselves to looking at types of the form $M[q]$, where M is an S-minimal base automaton. For such automata equivalence actually reduces to simple equality.

2.6 The Algorithm

In the end, type inference reduces to an algorithmic problem that is clearly outlined by the above definitions. Since the relations involved in the type constraints can be quite expressive, it is not possible to provide a completely general algorithm. For the simple case of core-ML, it is well-known that the constraints can be solved by means of the unification algorithm in time $O(n)$, where n is the size of the program [10].

It is sometimes thought that for more general languages one must seek to develop a more general notion of unification. It is our experience that this is not so. In spite of the uniform framework we present, it seems that ad-hoc languages give rise to ad-hoc algorithms. We have seen occurrences of algorithms as varied as the following: determinization of non-deterministic finite state automata [9], dynamic transitive closure [6], and in this paper a mixture between a name equivalent version of unification and a special case of general satisfiability.

2.7 Polymorphism

An integral part of the ML language is its approach to polymorphism. It is traditionally linked to the notion of *principal type scheme*, but we shall argue that the proper generalization is that of *code expansion*.

Functions defined by the polymorphic let-construct in ML are assigned a principal type scheme, which is instantiated with fresh type variables at all call sites. Imagine now that we syntactically expand these let-constructs. This will yield a potentially exponentially larger monomorphic term that is typable if and only if the original term is typable. This is not surprising, since type variables are derived from parse tree nodes. Mairson [8] and Kfoury et al [4] have shown that type inference in the polymorphic case is complete for deterministic exponential time, so we have not even compromised efficiency in the asymptotic sense.

For more ad-hoc type systems it is less likely that principal type schemes will exist. Thus, our generalized approach to polymorphism will be that of code expansion, which is clearly always a meaningful transformation. For ML-style languages we obtain in principle the traditional result, even if we are then left to discover the possibilities of compact representations of types and efficient polymorphic code generation.

3 An Application to Turbo Pascal

We now have generalized the concepts underlying the technique for ML-style type inference. We next endeavor to demonstrate that these generalizations are sensible and practical, rather than merely a formal exercise. To that purpose we develop an algorithm for type inference in the Turbo Pascal language—truly a worthy representative of the ad-hoc school of languages.

There is of course not a formalized description of the Turbo Pascal type system. We have therefore had to rely on the informal reference manual, from which we have been able to deduce a formal model that has been verified through experiments with the implementation.

3.1 Type Domain

We define the type domain for Turbo Pascal by giving the alphabet and the regular tree language. This is an example where the regular tree language is different from the free algebra generated by Σ.

Turbo Pascal provides a finite set $\mathcal{C}_{TP}$ of type constants and a ranked alphabet:

$$\begin{array}{rcl} \Sigma_{TP} & = & \mathcal{C}_{TP} \\ & & \cup \{(a_1, \ldots, a_n) \mid n > 0 \wedge i \neq j \Rightarrow a_i \neq a_j\} \\ & & \cup \{\text{record}(a_1, \ldots, a_n) \mid n \geq 0 \wedge i \neq j \Rightarrow a_i \neq a_j\} \\ & & \cup \{\text{set}, \text{array}, \hat{}\,, \text{file}\} \end{array}$$

where $arity(c) = 0$ for all $c \in \mathcal{C}$, $arity(a_1, \ldots, a_n) = 0$, $arity(\text{set}) = arity(\text{file}) = arity(\hat{}) = 1$, $arity(\text{record}(a_1, \ldots, a_n)) = n)$, and $arity(\text{array}) = 2$.

In Turbo Pascal there is a subset $\mathcal{O}_0 \subseteq \mathcal{C}_{TP}$ of *ordinal types*. This set is extended to $\mathcal{O} = \mathcal{O}_0 \cup \{(a_1, \ldots, a_n) \mid n > 0 \wedge i \neq j \Rightarrow a_i \neq a_j\}$. Furthermore we have the set $\overline{F}$ of Σ_{TP}-terms *not* containing the label *file*. The regular tree language L_{TP} is the set of Σ_{TP}-terms such that the subterm of a node with label *set* will be in $\mathcal{O}$, the left subterm of a node with label *array* will be in $\mathcal{O}$, and the subterm of a node with label *file* will be in $\overline{F}$. That is

$$L_{TP} = \{t \mid \forall \alpha : t(\alpha) \in \{\text{set}, \text{array}\} \Rightarrow t/\alpha 1 \in \mathcal{O} \wedge t(\alpha) = \text{file} \Rightarrow t/\alpha 1 \in \overline{F}\}$$

To see that L_{TP} is a regular tree language simply construct the obvious grammar with nonterminals *ord* generating $\mathcal{O}$ and *notfile* generating $\overline{F}$.

All in all, the type domain in Turbo Pascal is $\mathcal{A}(\Sigma_{TP}, L_{TP})$.

3.2 Type Equivalence

The Turbo Pascal type system is distinguished from many others by using the notion of *name equivalence*, whereas ML incorporates *structural equivalence*. That is, two types are considered equal if and only if they have the same tree structure. As we have seen, this corresponds to term equality between automata. In name equivalence, types are equal if and only if they have the same name or they are explicitly defined to be equal in an equational system.

In general, we consider a ranked alphabet, Σ, and a countably infinite set, $\mathcal{V}$, of variables. The set of *right-hand sides* over $\mathcal{V}$ and Σ is $\mathcal{R}(\mathcal{V}, \Sigma) = \mathcal{V} \cup \{l(v_1, \ldots, v_n) \mid l \in \Sigma, arity(l) = n, v_i \in \mathcal{V}\}$. An *equational system*, $\mathcal{E}$, over $\mathcal{V}$ and Σ, is a finite subset $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{R}(\mathcal{V}, \Sigma)$. We will say that $\mathcal{E}$ is *well-defined*, if the left-hand sides are all different, *closed* if any variable on a right-hand side occurs on a left-hand side, and *normalized* if every right-hand side is of the form $l(v_1, \ldots, v_n)$.

If $\mathcal{E}$ is well-defined, we will view it as a partial function $\mathcal{E} : \mathcal{V} \rightharpoonup \mathcal{R}(\mathcal{V}, \Sigma)$. So for $(v, r) \in \mathcal{E}$, we will write $\mathcal{E}(v) = r$. Similarly, we take $dom(\mathcal{E})$ and $codom(\mathcal{E})$ to be $\{v \mid (v, r) \in \mathcal{E}\}$ and $\{r \mid (v, r) \in \mathcal{E}\}$, respectively.

Definition 3.1 *Let $\mathcal{E}$ be a well-defined, closed type equation system. The name equivalence relation with respect to $\mathcal{E}$ is the reflexive, symmetric, and transitive closure of the relation $\mathcal{E} \cap \mathcal{V} \times \mathcal{V}$. We write this relation as $\sim_{\mathcal{E}}$, and the equivalence classes as $\langle v \rangle_{\mathcal{E}} = \{v' \mid v \sim_{\mathcal{E}} v'\}$.*

Definition 3.2 *Let $\mathcal{E}$ be a well-defined, closed equational system over $\mathcal{V}$ and Σ. The* quotient *of $\mathcal{E}$ is an equational system over $2^{\mathcal{V}}$ and $\Sigma \cup \{\Omega\}$:*

$$\begin{aligned} \mathcal{E}/\sim_{\mathcal{E}} \;=\; & \{(\langle v \rangle_{\mathcal{E}}, l(\langle v_1 \rangle_{\mathcal{E}}, \ldots, \langle v_n \rangle_{\mathcal{E}})) \mid (v, l(v_1, \ldots, v_n)) \in \mathcal{E}\} \\ & \cup \{(\langle v \rangle_{\mathcal{E}}, \Omega) \mid \forall v' \in \langle v \rangle_{\mathcal{E}} : \mathcal{E}(v) \in \mathcal{V}\} \end{aligned}$$

Clearly, $\mathcal{E}/\sim_{\mathcal{E}}$ is well-defined, closed, and normalized.

Definition 3.3 *Let $\mathcal{E}$ and $\mathcal{E}'$ be well-defined, closed equational systems. We say that $\mathcal{E}$ and $\mathcal{E}'$ are* congruent, *iff $\mathcal{E}/\sim_{\mathcal{E}} = \mathcal{E}'/\sim_{\mathcal{E}'}$.*

Definition 3.4 *Let $\mathcal{E}$ be a well-defined, closed, and normalized equational system over $\mathcal{V}$ and Σ. We define* $\textsc{Auto}(\mathcal{E}) = (Q, \Sigma, \delta, l)$*, where* $Q = dom(\mathcal{E})$*, and for each* $(v, \lambda(v_1, \ldots, v_n)) \in \mathcal{E}$*, we have* $\delta(v, i) = v_i$ *and* $l(v) = \lambda$*.*

For a well-defined, closed equational system, $\mathcal{E}$, we can construct the automaton $\textsc{Auto}(\mathcal{E}/\sim_{\mathcal{E}})$. With this construction we can prove the proposition below.

Proposition 3.1 *The set of base automata over $\mathcal{V}$ and $\Sigma \cup \{\Omega\}$ is isomorphic to the set of well-defined, closed equational systems over $\mathcal{V}$ and Σ modulo congruence.*

In Turbo Pascal the programmer defines an equational system, and then uses the type variables to annotate the program.

Definition 3.5 *A type in Turbo Pascal is a pair* $(\mathcal{E}, v)$*, where $\mathcal{E}$ is a well-defined, closed equation system over $\mathcal{V}$ and Σ_{TP}, and* $v \in dom(\mathcal{E})$*.*

Definition 3.6 *Two types,* $(\mathcal{E}, v)$ *and* $(\mathcal{E}', v')$ *are* name equivalent*, iff $\mathcal{E}$ and $\mathcal{E}'$ are congruent and* $v \sim_{\mathcal{E}} v'$*. We write this as* $(\mathcal{E}, v) \approx (\mathcal{E}', v')$*.*

Lemma 3.2 *Let* $M = \textsc{Auto}(\mathcal{E}/\sim_{\mathcal{E}})$ *and* $M' = \textsc{Auto}(\mathcal{E}'/\sim_{\mathcal{E}'})$*. We have that*

$$(\mathcal{E}, v) \approx (\mathcal{E}', v') \Leftrightarrow \mathcal{S}_M(v) \approx_{\emptyset} \mathcal{S}_{M'}(v')$$

It is impractical to use the relation $\approx_{\emptyset}$. Consider for example the following program:

```
Var x: Integer;
Var y: Integer;
x := y
```

Any programmer would expect this to work properly. So in Turbo Pascal the name equivalence relation is loosened to $\approx_{\mathcal{C}_{TP}}$ (which we shall write $\approx_{\mathcal{C}}$).

3.3 Type Variables

As for ML we generate a finite collection of type variables, but in this case only for parse tree nodes corresponding to expressions or variable references, since these are the only ones that can have types. We identify type variables corresponding to different occurrences of the same variable reference. We use the same notation as for ML.

3.4 Type Constraints

The Turbo Pascal constraints are quite complex. They reflect the ad-hoc nature of the language.

In Turbo Pascal we have a finite set $\mathcal{W}$ of finite sets of terms. The intention with $\mathcal{W}$ is to represent restrictions to overloading of operators. We have the constraint alphabet:

$$\begin{aligned}\Gamma_{TP} \quad = \quad & \{\in w \mid w \in \mathcal{W}\} \\ & \cup \{=l \mid l \in \Sigma_{TP}\} \\ & \cup \{Rec_\alpha \mid \alpha \text{ is a field name}\} \\ & \cup \{=, Tc, :=, Io, Op\}\end{aligned}$$

where $\in w$ has arity 1, $=l$ has arity $arity(l)+1$, Rec_α, $=$, Tc, $:=$, and Io has arity 2, and Op has arity 3. Let us look more closely at the constraints from this alphabet.

- $v \in w$ is used to restrict the other constraints, when they are too general. If M is the type corresponding to v, we must have $t^M \in w$.
- $v = l(v_1, \ldots, v_n)$ appears in connection with referencing subvariables. E.g. for the variable reference $\hat{}x$ we have the constraint $[\![x]\!] = \hat{}([\![\hat{}x]\!])$. If $M, M_1, \ldots, M_n$ are the types of $v, v_1, \ldots, v_n$ then we require $t^M(\epsilon) = l \wedge \forall i \in \{1, \ldots, n\} : M/i \approx_{\mathcal{C}} M_i$.
- $Rec_\alpha(v, v')$ is similar to the above. The variable reference $x.\alpha$ gives rise to the constraint $Rec_\alpha([\![x]\!], [\![x.\alpha]\!])$. If v and v' have types M and M' there must be a sequence $\alpha_1, \ldots, \alpha_n$ and an i between 1 and n, such that $t^M(\epsilon) = record(\alpha_1, ..., \alpha_n) \wedge \alpha = \alpha_i \wedge M' \approx_{\mathcal{C}} M/i$. This definition differs from the above by allowing records to be extended.
- $v = v'$ arises in a number of places. For example in procedure calls, where the actual type must equal the formal type of a variable parameter. M and M' is a solution to $v = v'$, iff $M \approx_{\mathcal{C}} M'$.
- $v\ Tc\ v'$, $v := v'$, $v\ Io\ v'$, and $Op(v, v', v'')$. These constraints are concerned with type coercion, overloading and the like. They are the most complex of the Turbo Pascal constraints. As examples of where they arise, we can take $x := e$, where the constraint is $[\![x]\!] := [\![e]\!]$ or $e + e'$, where we generate the constraints $Op([\![e]\!], [\![e']\!], [\![e+e']\!])$, $[\![e]\!] \in w$, $[\![e']\!] \in w$, and $[\![e+e']\!] \in w$, where

  $$w = \{\text{Integer}, \text{Real}, \text{String}, \text{Char}, \text{array}(\text{Integer}, \text{Char})\} \cup \{t \mid t(\epsilon) = \text{set}\}.$$

  The definitions of these relations show how complex they are. For example Tc is defined as the least symmetric relation such that $M\ Tc\ M'$, if one of the following holds:

  - $M \approx_{\mathcal{C}} M'$
  - $t^M, t^{M'} \in \{\text{Integer}, \text{Real}\}$
  - $t^M(\epsilon) = t^{M'}(\epsilon) = \text{set} \wedge M/1\ \ Tc\ M'/1$
  - $t^M = t^{M'} = \text{array}(\text{Integer}, \text{Char})$
  - $t^M = \text{String} \wedge t^{M'} \in \{\text{String}, \text{Char}, \text{array}(\text{Integer}, \text{Char})\}$

- $t^M = \text{Pointer} \wedge t^M(\epsilon) = ^\wedge$

To see that the relations in Γ_{TP} are resistant to $\approx_{\mathcal{C}}$-preserving families of homomorphisms, it is enough to note that these are defined only by the terms of the automata and the relation $\approx_{\mathcal{C}}$.

3.5 Solutions

In Turbo Pascal it becomes very natural to represent the solution as a base automaton, since this is isomorphic to type equations. As we have hoped for the concept of a base automaton suggests an algorithm that can solve some of the constraints. For the rest of the constraints we have to be more inventive. The construction of the algorithm is outlined in the next subsection.

3.6 The Algorithm

The Turbo Pascal type constraints may seem quite arbitrary and without any redeeming structure. For example, the relation Tc is reflexive and symmetric, but not transitive. Even worse, the relation := is neither reflexive, symmetric, transitive, nor antisymmetric. Nonetheless, these relations have *some* structure that can be exploited to obtain an algorithm.

The basic idea of the algorithm is to represent a set of solutions (base automata) by a base automaton labeled by sets of labels. The idea is to label each state with the set of labels that it might take according to the constraints read so far. For each label we have the possible transitions out of the state. If we have yet to read a constraint for any of the variables in the state, we say that those variables are *fresh*. We mark this by a special label, $*$.

By using base automata over label sets we can represent the set of solutions to the relations $\in w$, $= l$, Rec_α, and $=$. Furthermore, by using name equivalent unification we can maintain a representation of the set of solutions to a set of constraints over these relations. What remains is to deal with the relations Tc, :=, Io, and Op. As one might expect, these relations are too complex to fit into the model of base automata over sets of labels.

In general, we consider relations $R(M_1, \ldots, M_n)$, which can be described as finite disjunctions of clauses, where each clause is a finite conjunction of literals, and each literal is either of the form $l(q) = \lambda$ or $M_i[q] \approx_{\mathcal{C}} M_j[q']$. Finding solutions to such relations is in general NP-hard, so we add extra requirements to the relations R. If these requirements are satisfied we can find a solution to a set of $O(n)$ constraints over $O(n)$ variables in time $O(n^2 \log n)$.

Turbo Pascal constraints violates these requirements in a few places, but in each case we can argue that the algorithm succeeds here as well. It seems that in general the requirements on the strong labels and the partition of the labels can be weakened somewhat without affecting the correctness of the algorithm, but it is not clear at the moment how to describe the full generality of the algorithm.

3.7 Polymorphism

As predicted in the previous section, Turbo Pascal does not allow principal types schemes. This can be demonstrated by the following simple example.

Procedure P(**var** x; y);
begin
x := 2*y;
end;

In the above procedure, both the type of $[\![x]\!]$ and the type of $[\![y]\!]$ can be any of the types Integer and Real but not independently. If the type of $[\![x]\!]$ is Integer then the type of $[\![y]\!]$ must be Integer as well. If the type of $[\![x]\!]$ is Real then the type of $[\![y]\!]$ can be any of the types Integer and Real. Clearly, it is impossible to express this interdependence through type schemas, where the interdependence of two types is expressed through common type variables. It might be possible to express the interface of a procedure in a more compact form than just the set of constraints. At this moment, however, it is not clear to us how this should be done. The only remaining possibility is to resort to code expansion. In any case, we would have to generate expanded code since Turbo Pascal has no native polymorphism.

We proceed by taking a procedure, which is called from two places within the program. We make a copy of the procedure with a new name, and replace one of the two calls by a similar call to the new procedure. We repeat this until no procedure is called from two different places. In the presence of recursive procedures this method will lead to infinite copying. This means, that just as in ML we must disallow recursive polymorphism.

We can view the problem as one concerning graphs in the following way. Any procedure is a node in the graph. In addition there is a main node for the main body of the program. For each procedure call there is an edge from the calling procedure to the called procedure. Assuming that all nodes in the graph are reachable from the main node, we can proceed by a breadth-first search from the main node, coloring the nodes as we meet them and copying a node when we meet it for the second time. In order to deal with cycles in the graph (corresponding to recursive calls) we construct the *component graph* before beginning the search. The component graph has as nodes the maximal, strongly connected components and as edges those going between different components.

By proceeding down the block structure of a program in a depth-first, post-order fashion and using the above method on each scope we have the full code expansion algorithm. In the worst case the size of the expanded graph is exponential, so the asymptotic time of code expansion will be exponential in the size of the program. For a program of size n we get that the total time for type inference with code expansion is $2^{O(d \log n)}$, where d is the longest path through the component graph. This is the same time bound that Kanellakis and Mitchell [3] found for type inference of ML.

4 Conclusion

The framework that we have presented is sufficiently general to cover the spectrum from ML to Turbo Pascal. Even so, it is sufficiently specific to provide substantial assistance in the development of algorithms.

The tool that we have constructed for Turbo Pascal suggests how the benefits of type inference may also come to influence the programming methodology for standard imperative languages. The algorithm we have presented can easily be adapted to, say, Modula-2 or C.

It is safe to assume that ad-hoc languages will continue to have a presence in our culture. We hope by this work to demonstrate that modern tools can provide substantial help in using them.

References

[1] Bruno Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.

[2] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(1):95–169, 1983.

[3] Paris C. Kanellakis and John C. Mitchell. Polymorphic unification and ML typing. In *16th ACM conf. on Principles Of Programming Languages*, pages 105–116, 1989.

[4] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41:368–398, 1994.

[5] Dexter Kozen. Logical aspects of set constraints. TR 94-1421, Department of Computer Science, Cornell University, Ithaca, New York, May 1994.

[6] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. In *Proc. 33rd Symposium on Foundations of Computer Science.*, pages 363–371. IEEE, October 1992.

[7] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proc. 20th Symp. Princip. Programming Lang.*, pages 419–428. ACM, January 1992.

[8] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *Seventeenth Symposium on Principles of Programming Languages*, pages 382–401. ACM Press, January 1990.

[9] Michael I. Schwartzbach. Type inference with inequalities. In *Proceedings of TAPSOFT'91*. Springer-Verlag (*LNCS* 493), 1991.

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