ML TYPABILITY IS DEXPTIME-COMPLETE *

(Extended Summary)

A.J. Kfoury
Dept of Computer Science
Boston University

J. Tiuryn
Institute of Mathematics
University of Warsaw

P. Urzyczyn
Institute of Mathematics
University of Warsaw

Abstract

We carry out an analysis of typability of terms in ML. Our main result is that this problem is DEXPTIME-hard, where by DEXPTIME we mean DTIME($2^{n^{O(1)}}$). This, together with the known exponential-time algorithm that solves the problem, yields the DEXPTIME-completeness result. This settles an open problem of P. Kanellakis and J.C. Mitchell.

Part of our analysis is an algebraic characterization of ML typability in terms of a restricted form of semi-unification, which we identify as *acyclic semi-unification*. We prove that ML typability and acyclic semi-unification are *log*-space equivalent problems. We believe this result is of independent interest.

1 Introduction

Despite the great success of ML as a programming language, in part due to the type-checking facilities it offers, no completely satisfactory analysis for ML typability has ever been given.

Consider a term M of the λ-calculus augmented with the **let-in** constructor. To test whether M is typable, the conventional ML type-checker transforms M into an instance $\mathcal{I}_M$ of first-order unification. Although there are efficient linear-time algorithms for first-order unification (see [12]), the presence of polymorphically typed let-bound variables can make the size $|\mathcal{I}_M|$ an exponential in the size $|M|$ (see [6]), so that the procedure will also execute in time exponential in $|M|$. On the other hand, if there are no let-bound variables in M, $|\mathcal{I}_M|$ is linear in $|M|$ and the time required to set up $\mathcal{I}_M$ is polynomial in $|M|$, so that the procedure will also run in time polynomial in $|M|$. In fact, the problem of typability in ML of a pure λ-term is known to be PTIME-complete ([3, 6]).[1]

A very nice result due to P. Kanellakis and J.C. Mitchell [6] establishes a lower bound on the complexity of the ML typability problem, showing that it is PSPACE-hard. The main contribution of the present paper is an improvement of this result. We show that the ML typability problem is DEXPTIME-hard, where by DEXPTIME we mean DTIME($2^{n^{O(1)}}$). This, together with the known exponential time algorithm that solves the problem, yields the DEXPTIME completeness result.[2]

*This work is partly supported by NSF grant CCR–8901647 and by a grant of the Polish Ministry of National Education, No. RP.I.09 .

[1] This result was independently obtained in [13].

[2] A few days after submitting this report to CAAP, we found out that Harry Mairson independently obtained the same result by an altogether different method.

One benefit of our approach is the formulation of a unification problem, more general than first-order, which is *log*-space equivalent to the problem of **ML** typability. This is a certain fragment of *semi-unification* that we refer to as *acyclic semi-unification*. The importance of the semi-unification problem for problems of typability in programming languages has been known for some time (cf. [5, 8]). This problem also arises in other areas of Computer Science, e.g., in term rewriting (cf. [7]). We believe that the result establishing the *log*-space equivalence mentioned above is interesting in itself. The general case of semi-unification has been proven to be undecidable in [9].

The paper is organized as follows. Sections 2 and 3 are background material. In Section 2 we recall the typing system of **ML** (cf. [4, 10, 11]) together with a syntax-oriented equivalent of it (cf. [1]). In Section 3 we introduce acyclic semi-unification and develop some of its meta-theory. In Section 4 we prove that acyclic semi-unification and **ML** typability are *log*-space equivalent. In Section 5 we introduce an auxiliary kind of automata, *stratified two-push-down store automata*, and show that the problem of *boundedness*[3] for these automata is DEXPTIME-hard. We conclude the paper with Section 6 where we show a polynomial-time reduction of this boundedness problem to acyclic semi-unification.

The present report is only an extended summary of the final journal paper. All the proofs and several intermediary lemmas are omitted in this report.

2 An Alternative Type-Inference System For ML

We consider a λ-calculus augmented with the **let-in** constructor. The resulting *augmented λ-terms* are defined by the grammar:

$$M ::= x \mid (M\ N) \mid (\lambda x\ M) \mid (\textbf{let}\ x = N\ \textbf{in}\ M)$$

Whenever we refer to a "term" we mean an "augmented λ-term" as defined above. The *open types* are defined by the grammar:

$$\tau ::= \alpha \mid (\tau \to \tau')\ ,$$

where α ranges over type variables.[4] The *universal types* are expressions of the form $\forall\bar{\alpha}.\tau$, where $\bar{\alpha}$ is a finite set of type variables and τ is an open type. We reserve the letter σ to denote universal types, and τ and other Greek letters (different from α, β, γ and δ) as well as t and u to denote open types.

Types are assigned to terms according to the system shown in Figure 1. This is not the conventional type-inference system for **ML**, with which we assume the reader familiar (see, among others, [4, 10, 11]). The system of Figure 1 is temporarily called **ML*** to distinguish it from the conventional system **ML**.

In Figure 1, we follow standard notation and terminology. An *environment* A is a finite set of *type assumptions* $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ associating at most one type σ with each object variable x. Viewing A as a partial function from object variables to types, we may write $A(x) = \sigma$ to mean that the assumption $x : \sigma$ is in A. An *assertion* is an expression of the form $A \vdash M : \tau$ where A is an environment, M a term and τ a type. In such an assertion, the σ's (mentioned in A) are called the *environment types*, and τ the *assigned* or *derived* type.

Let τ and τ' be open types, and $\bar{\alpha}$ the set of type variables $\{\alpha_1, \ldots, \alpha_n\}$ for some $n \geq 0$. We write $\forall\bar{\alpha}.\tau \preceq \tau'$ to mean that τ' is an *instantiation* of $\forall\bar{\alpha}.\tau$, more precisely:

[3] This is the problem of determining whether for a given automaton there is a constant k such that the number of different instantaneous descriptions reachable from any one is at most k.

[4] We simplify the analysis by omitting constants from type expressions. We can omit them because our core object language is stripped of all constants and constructors (such as **if-then-else**) requiring constants (such as BOOL) in type expressions.

$\forall \vec{\alpha}.\tau \preceq \tau'$ iff $\tau' = \tau[\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n]$ for some open types $\tau_1, \ldots, \tau_n$.

$\mathrm{FV}(\sigma)$ is the set of type variables that are free in σ; if σ is open then $\mathrm{FV}(\sigma)$ is the set of all variables in σ.

VAR	$A \vdash x : \tau$	$A(x) = \sigma, \quad \sigma \preceq \tau, \quad \tau$ open
APP	$\dfrac{A \vdash M : \tau \to \tau', \qquad A \vdash N : \tau}{A \vdash (M\ N) : \tau'}$	τ, τ' open
ABS	$\dfrac{A[x : \tau] \vdash M : \tau'}{A \vdash (\lambda x\ M) : \tau \to \tau'}$	τ, τ' open
LET	$\dfrac{A \vdash N : \tau, \qquad A[x : \forall \vec{\alpha}.\tau] \vdash M : \tau'}{A \vdash (\mathbf{let}\ x = N\ \mathbf{in}\ M) : \tau'}$	$\vec{\alpha} = \mathrm{FV}(\tau) - \mathrm{FV}(A), \quad \tau, \tau'$ open

Figure 1. System **ML***: environment types are universal, derived types are open.

The following result is established in [1].

Proposition 1 *For all term M, M is typable in* **ML** *iff M is typable in* **ML***.

This proposition allows us to identify **ML** and **ML***, and whenever we mention **ML** we mean **ML***.

3 Semi-Unification

Let Σ be a first-order signature consisting of exactly one binary function symbol $\to$, used in infix notation, with the usual convention that $\alpha_1 \to \alpha_2 \to \alpha_3$ stands for $(\alpha_1 \to (\alpha_2 \to \alpha_3))$.[5] Let $V = \{\alpha_0, \alpha_1, \ldots\}$ be a countably infinite set of variables. $\mathcal{T}$ denotes the set of all algebraic terms over Σ and V. A *substitution* is a function $S : V \to \mathcal{T}$ such that $\{\alpha \in V | S(\alpha) \neq \alpha\}$ is finite. Every substitution extends in a natural way to a Σ-homomorphism $S : \mathcal{T} \to \mathcal{T}$.

An instance Γ of the Semi-Unification Problem is a finite set of pairs in $\mathcal{T} \times \mathcal{T}$. Each such pair is called an *inequality* and written as $t \leq u$ where $t, u \in \mathcal{T}$. A substitution S is a *solution* of instance $\Gamma = \{t_1 \leq u_1, \ldots, t_n \leq u_n\}$ iff there are substitutions $R_1, \ldots, R_n$ such that:

$$R_1(S(t_1)) = S(u_1)\ ,\ \ldots\ ,\ R_n(S(t_n)) = S(u_n)$$

The *Semi-Unification Problem* (henceforth abbreviated SUP) is the problem of deciding, for any instance Γ, whether Γ has a solution. We prove elsewhere that SUP is an undecidable problem [9].

[5] A more general form of semi-unification is based on a signature $\Sigma = \{\to, a_1, \ldots, a_n\}$ containing constant symbols in addition to $\to$. Since we do not include constants in type expressions, we can carry out the analysis using semi-unification based on $\Sigma = \{\to\}$. The results of this paper are easily adapted to the case when constants are introduced in λ-terms and in type expressions.

3.1 Acyclic Semi-Unification

An instance Γ of semi-unification is *acyclic* if it can be organized as follows. There are $n+1$ disjoint sets of variables, $V_0, \ldots, V_n$, for some $n \geq 1$, such that the inequalities of Γ can be placed in n columns:

$$\begin{array}{cccccc} t^{1,1} \leq u^{1,1} & t^{2,1} \leq u^{2,1} & \ldots & \ldots & t^{n,1} \leq u^{n,1} \\ t^{1,2} \leq u^{1,2} & t^{2,2} \leq u^{2,2} & \ldots & \ldots & t^{n,2} \leq u^{n,2} \\ \vdots & \vdots & & & \vdots \\ t^{1,r_1} \leq u^{1,r_1} & t^{2,r_2} \leq u^{2,r_2} & \ldots & \ldots & t^{n,r_n} \leq u^{n,r_n} \end{array}$$

where:

$$\begin{aligned} V_0 &= \mathrm{FV}(t^{1,1}) \cup \cdots \cup \mathrm{FV}(t^{1,r_1}) \\ V_1 &= \mathrm{FV}(u^{1,1}) \cup \cdots \cup \mathrm{FV}(u^{1,r_1}) \cup \mathrm{FV}(t^{2,1}) \cup \cdots \cup \mathrm{FV}(t^{2,r_2}) \\ &\vdots \\ V_{n-1} &= \mathrm{FV}(u^{n-1,1}) \cup \cdots \cup \mathrm{FV}(u^{n-1,r_{n-1}}) \cup \mathrm{FV}(t^{n,1}) \cup \cdots \cup \mathrm{FV}(t^{n,r_n}) \\ V_n &= \mathrm{FV}(u^{n,1}) \cup \cdots \cup \mathrm{FV}(u^{n,r_n}) \end{aligned}$$

For later reference, we call V_i the variables of *zone* i, for $i = 0, 1, \ldots, n$. In an acyclic instance Γ, there are n columns and $n+1$ zones. Because $V_0, \ldots, V_n$ are disjoint, the i-th column does not communicate with the j-th column, for $1 \leq i, j \leq n$, unless either $j = i-1$ (via variables in zone $i-1$) or $j = i+1$ (via variables in zone i).

The *Acyclic Semi-Unification Problem* (henceforth abbreviated ASUP) is the problem of deciding, for any acyclic instance Γ, whether Γ has a solution.[6]

3.2 Path Equations

For use in later sections, we introduce some additional theory concerning acyclic semi-unification. If Γ is an instance of ASUP, we assume for simplicity that the number of inequalities in each column is equal to r. (This assumption is not essential, as we can always add inequalities of the form $\alpha \leq \beta$, where α and β are new variables.) We thus decompose Γ as $\Gamma_1 \cup \cdots \cup \Gamma_n$, where Γ_i is the set of the r inequalities in the i-th column, identified by numbers in $\{1, \ldots, r\}$.

We introduce *auxiliary variables* of the form α_w, where $\alpha \in V$ and $w \in \{1, \ldots, r\}^*$. We identify α_ε with α. The intended meaning of the auxiliary variables is as follows: Suppose $\alpha \in V_i$ and let $w = j_1 j_2 \cdots j_p$. Then α_w is a new unknown, representing the instance of α obtained by a series of substitutions corresponding to the inequality numbered j_1 in the $i+1$-st column, j_2 in the $i+2$-nd column, etc., and finally j_p in the $i+p$-th column. Of course this makes sense only for $p \leq n-i$, and we will only use such auxiliary variables. Let $\overline{V}_i = V_i \cup \{\alpha_w | \alpha \in V_k \text{ and } |w| = i-k, \text{for some } k < i\}$. We now refer to $\overline{V}_i$ (rather than to V_i) as the set of zone i variables. We also use the notation $\overline{V} = \overline{V}_1 \cup \cdots \cup \overline{V}_n$. The notational convention below is that the initial lower case Greek letters are reserved for variables in V, while ξ, η, μ may denote members of $\overline{V}$.

[6] If we violate the acyclicity condition by allowing exactly one variable to appear in both the first and last zones, i.e., in both V_0 and V_n, we can show that the resulting semi-unification problem is equivalent to SUP without restriction and, therefore, undecidable.

We extend the lower index notation for arbitrary expressions by the rules: $(t \to u)_w = t_w \to u_w$ and $(t_w)_v = t_{wv}$.

We shall present our system of inequalities in the form of equations involving auxiliary variables. First, we associate one equation with each inequality obtaining a system of *base equations*, $E = E_1 \cup \cdots \cup E_n$, where:

$$E_i = \{\text{“}t^{i,j}_j = u^{i,j}\text{”} \mid j = 1, \ldots, r\}.$$

Then we add equations obtained by appropriate indexing of the equations in E, i.e., we define $\overline{E} = \overline{E}_1 \cup \cdots \cup \overline{E}_n$ where $\overline{E}_1 = E_1$ and

$$\overline{E}_i = \{\text{“}t_w = u_w\text{”} \mid \text{“}t = u\text{”} \in E_k \text{ and } |w| = i - k, \text{ for some } k < i\}.$$

Note that the equations in $\overline{E}_i$ involve only variables of zone i.

Let us recall here a well-known standard technique for solving systems of equations. Consider the following operations on systems of equations.

(a) Replace an equation of the form “$t \to t' = u \to u'$” by the equations “$t = u$” and “$t' = u'$”.

(b) Replace “$t = t'$” by “$t' = t$” .

(c) Replace “$\xi = t$” and “$\xi = t'$” by “$\xi = t$” and “$t = t'$”.

(d) Replace “$\xi = t$” and “$\eta = t'$” by “$\xi = t$” and “$\eta = t'[t/\xi]$”.

Clearly, each of the above operations transforms a system of equations into an equivalent one. If $\overline{E}$ has a solution, then, after a finite number of operations (a) – (d), it can be transformed into a system $\overline{E}^1$ with the following properties:

– all equations have the form “$\xi = t$”;

– variables occurring at left-hand sides do not occur at right-hand sides;

– a variable may occur at a left-hand side of at most one equation.

In this case, a most general solution $\overline{S}$ is obtained in an obvious way from the system $\overline{E}^1$. Namely, we first define $\overline{S}(\xi) = \xi$, for ξ not occurring at the left-hand sides, and then we extend $\overline{S}$ by $\overline{S}(\eta) = \overline{S}(t)$, provided “$\eta = t$” is in $\overline{E}^1$. If $\overline{E}$ does not have a solution, then a finite number of steps (a) – (d) yields an equation of the form “$\xi = t$” with ξ occurring in t as a proper leaf.

Lemma 2 *Γ has a solution iff $\overline{E}$ has a solution.*

A consequence of the above is that Γ has no solution iff one can derive from $\overline{E}$ an equation of the form “$\xi_w = t$” with ξ occurring as a proper subterm of t.

For the later simulation of a two push-down store automaton, we introduce the concept of a *path equation*, which is an expression of the form:

$$\text{“}\Pi\alpha_w = \Delta\beta_v\text{”}$$

where $\Pi, \Delta \in \{L, R\}^*$, and $\alpha_w, \beta_v \in \overline{V}_i$, for some i. The intended meaning of such an equation is that the subtrees of the unknowns, determined by the paths Π and Δ, are equal.

If ξ is an arbitrary variable, and $\Pi \in \{L, R\}^*$, then we call the expression ξ^Π a *shadow variable*. We identify ξ^ε with ξ. For any term t and $\Pi \in \{L, R\}^*$, we define the *subterm of t rooted at Π* by the rules: $\Pi(\xi) = \xi^\Pi$, $\varepsilon(t) = t$, $\Pi L(t_1 \to t_2) = \Pi(t_1)$, $\Pi R(t_1 \to t_2) = \Pi(t_2)$. Thus, $\Pi(t)$ is either a term, or a shadow variable (if Π is too long).

Let "$t = u$" be an equation between terms, and suppose that for some $\Pi, \Delta \in \{L, R\}^*$, and some $\xi, \eta \in V$ we have $\xi = \Pi(t)$ and $\eta = \Delta\Pi(u)$ (or, conversely, $\xi = \Pi(u)$ and $\eta = \Delta\Pi(t)$). Then the path equation:

$$\eta = \Delta\xi$$

is said to be *induced* by "$t = u$". We say that a path equation is induced by an inequality "$t^{i,j} \leq u^{i,j}$" of Γ iff it is induced by the equation "$t^{i,j}_j = u^{i,j}$" of E.

A proof system for path equations:

Axioms: All path equations induced by equations in E and all identity equations of the form $\alpha_w = \alpha_w$.

Rules:

(transitivity) $$\frac{\Pi\alpha_w = \Delta\beta_v, \quad \Delta\beta_v = \Sigma\gamma_u}{\Pi\alpha_w = \Sigma\gamma_u}$$

(symmetry) $$\frac{\Pi\alpha_w = \Delta\beta_v}{\Delta\beta_v = \Pi\alpha_w}$$

(instance) $$\frac{\Pi\alpha_w = \Delta\beta_v}{\Pi\alpha_{wu} = \Delta\beta_{vu}} \qquad \text{(provided } \alpha_{wu}, \beta_{vu} \in \overline{V}\text{)}$$

(subterm) $$\frac{\Pi\alpha_w = \Delta\beta_v}{\Sigma\Pi\alpha_w = \Sigma\Delta\beta_v}$$

Semantics:

We say that a path equation "$\Pi\alpha_w = \Delta\beta_v$" is satisfied by a substitution S iff $\Pi(S(\alpha_w)) = \Delta(S(\beta_v))$. The lemma establishes a soundness and (a weak sort of) completeness result for our proof system.

Lemma 3

(1) If a path equation is derivable then it is satisfied by every solution of $\overline{E}$. In particular, if "$\alpha_w = \Pi\beta_v$" is derivable, and S is a solution then $S(\alpha_w)$ is of depth at least equal to the length of Π.

(2) If $\overline{E}$ has no solution then there is a derivable path equation "$\alpha_w = \Pi\alpha_w$", for some $\alpha_w \in \overline{V}$ and some $\Pi \in \{L, R\}^*$, such that $|\Pi| > 0$.

4 An Algebraic Characterization of ML Typability

In this section we prove that ASUP and **ML** typability are *log*-space equivalent. One direction of this equivalence is not needed to prove that **ML** typability is DEXPTIME-complete, and we state the result without the intermediary lemmas.

Theorem 4 **ML** *typability is* log*-space reducible to ASUP.*

The other direction of the equivalence is one of the reduction steps we need to establish the DEXPTIME-completeness result. We outline our approach for this reduction.

Let z be an object variable and τ an open type. Type variables are named $\alpha_0, \alpha_1, \alpha_2, \ldots$, corresponding to which we introduce object variables $v_0, v_1, v_2, \ldots$. We define a term, denoted $\langle z : \tau \rangle$, by induction on open types:

1. if $\tau = \alpha_i$ for $i \in \omega$ then
   $\langle z : \tau \rangle = \lambda u_1.\lambda u_2.\ u_1(u_2 z)(u_2 v_i)$
2. if $\tau = \tau_1 \rightarrow \tau_2$ then
   $\langle z : \tau \rangle = \lambda z_0.\lambda z_1.\lambda z_2.\lambda u.\ z_0\langle z_1 : \tau_1\rangle\langle z_2 : \tau_2\rangle(u(zz_1))(uz_2)$

It is clear from the induction above that: $\mathrm{FV}(\langle z : \tau \rangle) = \{z\} \cup \{v_i | \alpha_i \in FV(\tau)\}$.

Lemma 5 *Let $\tau, \tau', \rho_1, \ldots, \rho_\ell$ be arbitrary open types such that $\mathrm{FV}(\tau) \subseteq \{\alpha_1, \ldots, \alpha_\ell\}$. Then the term $\langle z : \tau \rangle$ is* **ML** *typable in the environment $A = \{z : \tau', v_1 : \rho_1, \ldots, v_\ell : \rho_\ell\}$, i.e., $A \vdash \langle z : \tau \rangle : \tau''$ for some open type τ'', iff $\tau' = \tau[\alpha_1 := \rho_1, \ldots, \alpha_\ell := \rho_\ell]$.*

Consider an instance Γ of ASUP in the form specified in Subsection 3.1. We think of the t's and u's as open type expressions. There is no loss of generality in assuming that all the columns of Γ have an equal number r of inequalities, i.e.,

$$r = r_1 = \cdots = r_n$$

because we can add inequalities of the form $\beta \leq \gamma$, where β and γ are fresh variables, wherever needed to equalize column sizes. Let type variables in zone i, for $i = 0, 1, \ldots, n$, be: $\alpha_{i,1}, \alpha_{i,2}, \ldots, \alpha_{i,\ell_i}$, for some $\ell_i \geq 1$, corresponding to which we introduce object variables: $v_{i,1}, v_{i,2}, \ldots, v_{i,\ell_i}$. The notation $\langle z : \tau \rangle$ introduced earlier relative to singly subscripted variables, α_i and v_i, is now extended to doubly subscripted variables, $\alpha_{i,j}$ and $v_{i,j}$. We can assume that all the zones have an equal number ℓ of variables, i.e.,

$$\ell = \ell_0 = \ell_1 = \cdots = \ell_n$$

With these assumptions about Γ, we define a term M_Γ as follows:

$$\begin{aligned} M_\Gamma \equiv\ & \textbf{let } x_1 = N_1 \textbf{ in} \\ & \textbf{let } x_2 = N_2 \textbf{ in} \\ & \quad \vdots \\ & \textbf{let } x_{n-1} = N_{n-1} \textbf{ in} \\ & \textbf{let } x_n = N_n \textbf{ in } N_{n+1} \end{aligned}$$

where:

$$\begin{aligned}
N_1 &\equiv \lambda v_{0,1}\cdots v_{0,\ell}.\lambda z_0.\lambda z_1\cdots z_r.\ z_0\langle z_1 : t^{1,1}\rangle\cdots\langle z_r : t^{1,r}\rangle\\
N_2 &\equiv \lambda v_{1,1}\cdots v_{1,\ell}.\lambda z_0.\lambda z_1\cdots z_r.\ z_0 P_{1,1}\cdots P_{1,r}\langle z_1 : t^{2,1}\rangle\cdots\langle z_r : t^{2,r}\rangle\\
&\vdots\\
N_n &\equiv \lambda v_{n-1,1}\cdots v_{n-1,\ell}.\lambda z_0.\lambda z_1\cdots z_r.\ z_0 P_{n-1,1}\cdots P_{n-1,r}\langle z_1 : t^{n,1}\rangle\cdots\langle z_r : t^{n,r}\rangle\\
N_{n+1} &\equiv \lambda v_{n,1}\cdots v_{n,\ell}.\lambda z_0.\ z_0 P_{n,1}\cdots P_{n,r}
\end{aligned}$$

where:

$$P_{i,j} \equiv \lambda y'.\lambda w_1\cdots w_\ell.\lambda y_0.\lambda y_1\cdots y_r.\ y'(x_i w_1\cdots w_\ell y_0 y_1\cdots y_r)\langle y_j : u^{i,j}\rangle$$

for all $i = 1,\ldots,n$ and $j = 1,\ldots,r$. Observe that N_1 is a closed term and, for $i = 2,\ldots,n+1$, there is exactly one free variable in N_i, namely x_{i-1}, occurring exactly r times. The free variables of $P_{i,j}$ are x_i (occurring exactly once) and $v_{i,1},\ldots,v_{i,\ell}$. Based on Lemma 5 we can prove the following.

Lemma 6 *Term M_Γ is* **ML** *typable iff instance Γ has a solution.*

Lemma 7 *Consider the instance Γ of ASUP and the term M_Γ derived from Γ. The work space required to write M_Γ is at most $\mathcal{O}(\log|\Gamma|)$.*

An immediate consequence of Lemmas 6 and 7 is the following.

Theorem 8 *ASUP is* log*-space reducible to* **ML** *typability.*

5 Stratified Two-Push-Down Store Automata

A *stratified two-push-down store automaton*, S2PDS for short, is an ordered quadruple $M = \langle Q, A, B, R\rangle$, where Q is a finite set of *states*, A and B are finite *alphabets* of the first and the second push-down store of M, respectively, and $R \subseteq A^* \times Q \times B^* \times A^* \times Q \times B^*$ is a finite *transition relation* of M. The above items are subject to the following additional constraints.

- The set of states Q is partitioned into *levels* $Q_0,\ldots,Q_n$, for some $n \geq 0$,
- For every $(\Pi,\alpha,w,\Pi',\alpha',w') \in R$ there exists $0 \leq i < n$ such that one of the following three conditions holds
  1. $\alpha \in Q_i,\ \alpha' \in Q_{i+1},\ w = \varepsilon,\ |w'| = 1$
  2. $\alpha \in Q_{i+1},\ \alpha' \in Q_i,\ |w| = 1,\ w' = \varepsilon$
  3. $\alpha,\alpha' \in Q_i$, and $w = w' = \varepsilon$, in that case we also allow $i = n$
- For every $(\Pi,\alpha,w,\Pi',\alpha',w') \in R$ either $\Pi = \varepsilon$, or $\Pi' = \varepsilon$

Troughout this section we will use Π,Σ,Δ to range over the contents of the first push-down store (pds_1), α,β,γ to range over the states of a S2PDS, and v,w,y to range over the contents of the second push-down store (pds_2). The interpretation of a transition $(\Pi,\alpha,w,\Pi',\alpha',w') \in R$ is as follows: if M is in the state α,

reads Π on its first push-down store and w on its second push-down store, then M changes its state to α', replaces Π on its first push-down store by Π' and replaces w on its second push-down store by w'. A formal definition is given below.

An *instantaneous description*, ID for short, is any expression of the form $\Pi\alpha w$, where $\Pi \in A^*$, $\alpha \in Q$, and $w \in B^*$. It describes the situation when M is in the state α, and the contents of its first and second push-down stores are Π and w, respectively. For technical reasons we assume that the rightmost symbol in Π represents the topmost symbol of pds_1, and the leftmost symbol in w the topmost symbol of pds_2.

The next-ID relation, $\vdash_M$, is defined in a natural way.

$$\Delta\Pi\alpha wv \vdash_M \Delta\Pi'\alpha' w'v,$$

holds for every $(\Pi, \alpha, w, \Pi', \alpha', w') \in R$.

A S2PDS automaton M is said to be *bounded* iff there is a positive integer k such that for every ID, say $\Pi\alpha w$, the number of different ID's reachable by M from $\Pi\alpha w$ is at most k.

A S2PDS automaton M is *deterministic* if $\vdash_M$ is a partial function from the set of all ID's into itself. It is easy to check that it is decidable whether an arbitrary S2PDS is deterministic (it is easy to write down all the syntactic conditions on the transition relation that are equivalent to our definition of being deterministic).

Let $M = \langle Q, A, B, R\rangle$ be a S2PDS automaton. The *symmetric closure* of M, is a S2PDS automaton $M_S = \langle Q, A, B, R_S\rangle$, where

$$R_S = \{(\Pi, \alpha, w, \Pi', \alpha', w') \mid \text{either } (\Pi, \alpha, w, \Pi', \alpha', w') \in R \text{ or } (\Pi', \alpha', w', \Pi, \alpha, w) \in R\}$$

Our main goal in this section is to show DEXPTIME-hardness of the problem of boundedness for deterministic S2PDS automata. We start with a result that will be used later in the paper.

Lemma 9 *For every deterministic S2PDS automaton M, if M is bounded, then so is its symmetrical closure M_S.*

Lemma 10 *For every S2PDS automaton M there is a S2PDS automaton M' such that the alphabet of the first push-down store of M' has only two elements, and M is bounded iff M' is bounded. Moreover the construction can be performed in time polynomial in $|M|$.*

We assume that the reader is familiar with the notion of an *alternating* Turing machine and the fact that alternating polynomial space is equivalent to DEXPTIME. The willing reader may wish to consult [2] for details. Let $p(n)$ be a polynomial, and let T be a $p(n)$-space bounded alternating Turing machine. T is said to have a *binary control* if the following conditions hold:

- T has one semi-infinite tape (i.e., bounded on the left), and for every input x, every computation of T that starts with the input x always halts and it never leaves the area consisting of the first $p(|x|)$ cells.
- For every tape symbol a and for every universal or existential state p, the machine T has only two possible moves: $move_0(p, a) = (p, a, q, b, D)$ and $move_1(p, a) = (p, a, q', b', D')$, where b, b' are tape symbols, q, q' are states of T, and $D, D' \in \{L, R\}$ indicate the moves of the head. In order to cover the case of a deterministic machine, $move_0(p, a) = move_1(p, a)$ in the above definition is also allowed.

- T halts upon reaching any of its accepting or rejecting states.
- The initial state of T is either universal, or existential, i.e., T makes at least one step on every input.

Lemma 11 *Let T be a $p(n)$-space bounded alternating Turing machine. There is an alternating Turing machine $\hat{T}$ such that T and $\hat{T}$ accept the same languages and $\hat{T}$ has a binary control. Moreover the construction of $\hat{T}$ can be performed in time polynomial in $|T|$.*

Let T be an alternating Turing machine and let m be a positive integer. T is *m-mortal* iff for every instantaneous description c of T such that $|c| \leq m$, every computation of T that starts at c, halts.

Lemma 12 *Let T be a $p(n)$-space bounded alternating Turing machine with a binary control, and let m be a positive integer. There exists an alternating Turing machine $\hat{T}$ such that.*

1. *T and $\hat{T}$ are equivalent on all words of length m.*
2. *$\hat{T}$ has a binary control.*
3. *$\hat{T}$ is $p(m)$-mortal.*
4. *The construction of $\hat{T}$ can be performed in time polynomial in $|T|$, and m.*

5.1 Construction of $M_{(T,x)}$

Given a positive integer m, a $p(m)$-mortal alternating Turing machine T, with a binary control, and an input x of length m. We are going to construct in polynomial time a S2PDS automaton $M_{(T,x)}$, such that $M_{(T,x)}$ is bounded iff T accepts x . Let us assume that T has the set of states S, the input alphabet $\{0,1\}$, the tape alphabet Θ, and the transition relation R. The elements of R we will call the *moves* of T.

Before proceeding further let us fix some notation. Let C be the set of all ID's of T. We define a function $type : S \longrightarrow \{\forall, \exists, \mathcal{A}, \mathcal{R}\}$ which assigns the type $type(q)$ of any state q, i.e., whether this state is: universal, existential, accepting, or rejecting. We extend *type*, in a natural way, to all ID's of T, $type : C \longrightarrow \{\forall, \exists, \mathcal{A}, \mathcal{R}\}$, now $type(c)$ is the type of the state present in c. We also want to extend *type* to arbitrary moves, if $\xi = (p, a, q, b, D) \in R$ is a move of T, then $type(\xi) = type(p)$.

For every universal or existential instantaneous description c and for $i = 0, 1$, $move_i(c) = move_i(p, a)$, where p and a are the current state and tape symbol of c, respectively. Also, $Succ_i(c) = c'$, where c' is obtained from c by choosing the move $move_i(c)$.

Finally, we need one more function. Let $\xi \in R$ be any move of T. $Pred_\xi : C \longrightarrow C$ is defined as follows, $Pred_\xi(c) = c'$ iff c is obtained from c' by choosing the move ξ. Let us observe that for every c there is at most one c' with this property.

We first construct a S2PDS automaton M from which later $M_{(T,x)}$ is to be constructed. Despite of our notation, M clearly depends on T and x. Let us first describe it informally. M is going to use its pds_2 to store an ID of T of length $p(m)$. It will have $p(m)$ states to control how deep it looks into pds_2. The other push-down store, pds_1, is going to store the "history" of the computation in the computation tree of T, rooted at the initial ID $c(x)$. It will be also used sometimes as an auxiliary storage. The "history" consists of a sequence of pairs, the first element being either 0 , or 1, and the second element being a move in R. The sequence of 0's and 1's represents a path in the computation tree from the root to the present ID, stored at

the same time on pds_2. The sequence of moves is used to back up along a path in the tree. M will traverse the computation tree rooted at $c(x)$, in a depth-first-search mode, to determine whether T accepts x.

Now we start the actual construction of M. M consists of two parts. The first part starts in the state α_1 and loads pds_2 with the initial ID, $c(x)$, represented as a word of length $p(m)$. For this we use $p(m)$ different states $\alpha_1, \ldots, \alpha_{p(m)}$. The computation of M in this part consists in pushing on pds_2 the i-th symbol of $c(x)$ and changing the state from α_i to α_{i+1}, for $i = 1, 2, \ldots, p(m) - 1$. Finally we require that M when pushing the last symbol of $c(x)$ changes its state from $\alpha_{p(m)}$ to $\alpha_{(new,X,false)}$, where X is $\forall$ or $\exists$, depending on whether the initial state of T is universal or existential, repectively. The states $\alpha_1, \ldots, \alpha_{p(m)}, \alpha_{(new,X,false)}$ belong to different levels of M, $Q_0, \ldots, Q_{p(m)}$.

After reaching $\alpha_{(new,X,false)}$ the automaton enters the second part of the construction. In this part there are states of the form $\alpha_{(\mathtt{node},\mathtt{state},\mathtt{eval})}$, all of the same level $p(m)$, where node ranges over $\{new, 0, 1\}$, state ranges over $\{\forall, \exists, \mathcal{A}, \mathcal{R}\}$, and eval ranges over $\{true, false\}$. To understand the meaning of these flags let us assume that during the simulation we have arrived at an ID c. Then the flag node is new, if we have arrived at c for the first time. It is 0, if we have arrived at c via its left child; and it is 1, if we have arrived at c via its right child. The flag state is the type of c, and the flag eval is $false$ iff we cannot conclude at this stage that T accepts c. Thus $\alpha_{(new,X,false)}$ represents the situation that we have arrived at the current ID for the first time, the type of the ID is X, and (clearly) we don't know yet as to whether T will accept that ID or not. Some of the combinations of flags are impossible to arrive at, e.g., $\alpha_{(0,\mathcal{A},true)}$ or $\alpha_{(new,\exists,true)}$. This means that we will not need these states for M.

The description of (the second part of) M is given in the form of a flow-diagram presented in Figure 2. It explains how the states are being changed. This flow-diagram has to be repeated until β_{true} or β_{false} are reached, or until any kind of inconsistency has been discovered. It uses three subroutines UP, $DOWN_0$, and $DOWN_1$, which are presented in Figures 3 and 4.

These subroutines operate on contents of pds_2 viewed as ID's of T. This has to be understood in the following way. We consider only the topmost $p(m)$ cells of pds_2 as representing an ID, and disregard the items stored on pds_2 at depth greater than $p(m)$. Hence in order to look at the current ID we can use pds_1 as an auxiliary storage together with $p(m)$ new states which are used to measure how deep we went through pds_2 during the search. It is now a routine task to implement the actual subroutines that perform operations like

$$pds_2 := Succ_1(pds_2)$$

We assume that during the simulation if M discovers any kind of inconsistency, e.g., when looking through the contents of pds_2 it discovers that it does not represent an ID of T, or that pds_2 stores an ID such that in order to make the next move of T it would require to leave the $p(m)$ space reserved on pds_2, or $Pred_\xi$ is not defined for the current argument, then M aborts the simulation and halts in a special state. Also, copies of the subroutines $DOWN_0$, $DOWN_1$, or UP when used in different places of the chart given in Figure 2 have to use disjoint states.

```
if node = new    then if eval = false then if state ∈ {∀,∃}  then DOWN_0
                                           else if state = A    then eval := true; UP
                                           else if state = R    then UP
else if node = 0 then if eval = false then if state = ∀      then UP
                                           else if state = ∃    then DOWN_1
                      else if eval = true  then if state = ∀    then DOWN_1
                                           else if state = ∃    then UP
else if node = 1 then if state ∈ {∀,∃}  then UP
```

Figure 2. The main part of the machine M

```
push (i, move_i(pds_2)) on pds_1;
pds_2 := Succ_i(pds_2);
node := new;
state := type(pds_2)
```

Figure 3. Subroutine $DOWN_i$, for $i = 0, 1$.

```
if pds_2 = c(x)  then change state to β_eval and halt
                 else
                 pop pds_1; let the top of pds_1 be (i, ξ);
                 node := i;
                 state := type(ξ);
                 pds_2 := Pred_ξ(pds_2)
```

Figure 4. Subroutine UP. The states β_{true} and β_{false} are brand new.

5.2 The properties of M and $M_{(T,x)}$

It should be obvious that the constructed M is a S2PDS automaton, that it is deterministic, and that it is constructed in time that polynomially depends on M and $p(m)$.

Lemma 13 *Let M be the S2PDS automaton constructed above. Then M is bounded and the following are equivalent:*

1. *T accepts [rejects] x.*
2. *There is a computation of M that starts at α_1 (and certain contents of pds_1 and pds_2) and leads to β_{true} [β_{false}, respectively].*
3. *Every computation of M that starts at α_1 (and any contents of pds_1 and pds_2) leads to β_{true} [β_{false}, respectively].*

$M_{(T,x)}$ is obtained from M by adding a third part. This part starts in state β_{false} and, using $p(m)$ brand new states, transfers the topmost $p(m)$ items of pds_2 on pds_1. When transforming the last item, it changes its state to α_1. Let us observe that the resulting automaton is still a deterministic S2PDS. The next is the main result of this section, which is proved using the lemmas collected earlier in this section.

Theorem 14 *The boundedness problem for S2PDS automata is DEXPTIME-hard.*

6 From S2PDS to ASUP

From now on, let $M = \langle Q, A, B, R\rangle$ be a deterministic S2PDS. By Lemma 10, we can assume that the alphabet A of the first push-down store is just $\{L, R\}$. Let the alphabet of the second push-down store be $B = \{1, \ldots, r\}$. By M_S we denote the symmetric closure of M. Lemma 9 allows us to work now with M_S rather than M. We may assume that the sixtuples of R_S are in one of the following forms (up to symmetry):

- $\langle \Pi, \alpha, j, \varepsilon, \beta, \varepsilon\rangle$, with $\alpha \in Q_{i+1}, \beta \in Q_i$, for some i;
- $\langle \varepsilon, \alpha, j, \Pi, \beta, \varepsilon\rangle$, with $\alpha \in Q_{i+1}, \beta \in Q_i$, for some i;
- $\langle \varepsilon, \alpha, \varepsilon, \Pi, \beta, \varepsilon\rangle$, with $\alpha, \beta \in Q_i$, for some i;

Our goal is to construct an instance Γ of ASUP such that M_S is bounded iff Γ has a solution. The variables of Γ will include the states of M_S. The stratification of states into levels will determine the stratification of inequalities into columns. There will be r inequalities in each level, and the numbers stored on pds_2 will correspond to the inequalities. The intuition of the first push-down store contents is that it is understood as a path in a tree. The construction of Γ will be such that the path equations induced by inequalities in Γ correspond to possible moves of M_S.

Let us first introduce some conventions. If α is a variable in Q (a state of M_S), and $\Pi \in \{L, R\}^*$ then we define the notion of a Π-*term* for α by induction as follows:

– t is an ε-term for α iff $t = \alpha$;

– t is a ΔL-term for α iff $t = t' \rightarrow \xi$, where t' is a Δ-term for α, and ξ is not in Q and does not occur in t';

– t is a ΔR-term for α iff $t = \xi \to t'$, where t' is a Δ-term for α, and ξ is not in Q and does not occur in t'.

Informally, a Π-term for α consists of a single path Π to α, with all the necessary additional leaves different. For instance, if $\xi^1, \ldots, \xi^4$ are new different variables, then $(\xi^1 \to (\alpha \to \xi^2) \to \xi^3) \to \xi^4$ is an $LLRL$-term for α. In order to construct the j-th inequality in the i-th column of Γ (here, $i = 1, \ldots, n$ and $j = 1, \ldots, r$), we consider all sixtuples of R_S of the form

$$\langle \Pi, \alpha, j, \Pi', \beta, w' \rangle$$

where α is a state of level $n - i$. Let $S_1, \ldots, S_p$ be all such sixtuples. For each $\ell \in \{1, \ldots, p\}$ we construct a pair of terms (t^ℓ, u^ℓ). Then define $t^{i,j} = t^1 \to t^2 \to \cdots \to t^p$ and $u^{i,j} = u^1 \to u^2 \to \cdots \to u^p$. The j-th inequality in the i-th column is then $t^{i,j} \leq u^{i,j}$.

For the construction of (t^ℓ, u^ℓ) we consider two cases. If $S_\ell = \langle \Pi, \alpha, j, \varepsilon, \beta, \varepsilon \rangle$ then t^ℓ is α and u^ℓ is a Π-term for β. For example, if $S_\ell = \langle LLRL, \alpha, j, \varepsilon, \beta, \varepsilon \rangle$ then our pair of terms is $(\alpha, (\xi^1 \to (\beta \to \xi^2) \to \xi^3) \to \xi^4)$. Thus, S_ℓ is represented by the path equation "$\beta = LLRL\alpha_j$".

If $S_\ell = \langle \varepsilon, \alpha, j, \Pi, \beta, \varepsilon \rangle$ then u^ℓ is β and t^ℓ is a Π-term for α. The pairs (t^ℓ, u^ℓ) will thus contain new variables, not in Q. We assume that new variables introduced for each i, j and for each possible pair (t^ℓ, u^ℓ) are different. The inequalities constructed so far do not yet complete our construction of Γ. For each sixtuple S of the form

$$\langle \varepsilon, \alpha, \varepsilon, \Pi, \beta, \varepsilon \rangle,$$

where $\alpha, \beta \in Q_{n-i}$ we add to the i-th column an inequality $\xi \to \xi \leq \alpha \to t$, where ξ is a new variable and t is a Π-term for β, such that the variables introduced by t are new. (Note that this construction may require introducing a column numbered 0, in case when $\alpha \in Q_n$.)

Consider now the (undirected) graph G whose nodes are all expressions of the form "$\Pi\alpha_w$", where $\Pi \in \{L, R\}^*$, $\alpha \in V$ and $w \in \{1, \ldots, r\}^*$, and such that there is an edge between "$\Pi\alpha_w$" and "$\Sigma\beta_v$" iff the path equation "$\Pi\alpha_w = \Sigma\beta_v$" can be obtained from an axiom, by a number of applications of symmetry, instance and subterm rules. Clearly, a path equation is provable iff there is a path in our graph between the appropriate expressions. Note that, if $\gamma \in V - Q$, then there are two possibilities, depending on the way γ was introduced. One possibility is that all expressions involving γ are hanging nodes in our graph, i.e., have degree 1. The other case is when all expressions involving γ are nodes of degree 2 (this happens when γ was introduced because of a sixtuple of the form $\langle \varepsilon, \alpha, \varepsilon, \Pi, \beta, \varepsilon \rangle$).

Lemma 15 establishes a connection between reachability relation on the ID's of M_S, and the path equations for Γ. Lemma 16 is based on Lemmas 3 and 15.

Lemma 15 *Two arbitrary ID's, $\Pi\alpha w$ and $\Sigma\beta v$ of M_S are reachable from each other iff the path equation "$\Pi\alpha_w = \Sigma\beta_v$" is provable from Γ.*

Lemma 16 *M_S is bounded iff Γ has a solution.*

An immediate consequence of Theorem 8, Theorem 14, and Lemma 16, is the following theorem.

Theorem 17 **ML** *typability is DEXPTIME-hard.*

References

[1] Clément, D., Despeyroux, J., Despeyroux, T., Kahn, G., "A simple applicative language: Mini-ML", *Proc. ACM Conference on Lisp and Functional Programming*, pp 13-27, 1986.

[2] Chandra, A.K., Kozen, D. and Stockmeyer, L. "Alternation," *J. Assoc. Comput. Mach.*28:1, pp 114-133, 1981.

[3] Dwork, C., Kanelakis, P., Mitchell, J.C., "On the sequential nature of unification", *J. of Logic Programming*, **1**, pp 35-50, 1984.

[4] Damas, L. and Milner, R., "Principal type schemes for functional programs," *Proc. 9-th ACM Symp. Principles of Prog. Lang.*, pp 207-212, 1982.

[5] Henglein, F., "Type inference and semi-unification", *Proc. ACM Symp. LISP and Functional Programming*, July 1988.

[6] Kanellakis, P., Mitchell, J.C., "Polymorphic unification and ML typing", *Proc. 16-th Symp. on Principles of Prog. Lang.*, pp 105-115, Jan 1989.

[7] Kapur, D., Musser, D., Narendran, P., and Stillman, J., "Semi-unification", *Proc. of 8-th Conference on Foundations of Software Technology and Theoretical Computer Science*, Pune, India, Dec. 1988.

[8] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P., "Computational consequences and partial solutions of a generalized unification problem", *Proc. of IEEE 4-th LICS* 1989.

[9] Kfoury, A.J., Tiuryn, J. and Urzyczyn, P., "Undecidability of the semi-unification problem" Manuscript, October 1989.

[10] Milner, R., "A theory of type polymorphism in programming", *J. of Computer and System Sciences*, Vol. 17, pp 348-375, 1978.

[11] Milner, R., "The standard ML core language", *Polymorphism*, II (2), October 1985.

[12] Paterson, M.S., Wegman, M.N., "Linear unification", *JCSS*, **16**, pp 158-167, 1978.

[13] Tyszkiewicz, J. "Typability in the finitely typed lambda calculus is PTIME-complete," MS thesis, University of Warsaw, 1987.