# Haskell Overloading is DEXPTIME–complete

Helmut Seidl
Fachbereich Informatik
Universität des Saarlandes
Postfach 151150
D–66041 Saarbrücken
Germany
seidl@cs.uni-sb.de

Febr. 2, 1994

**Keywords:** Haskell type classes; emptiness; complexity.

## 1 Introduction

A Haskell *class* $C$ corresponds to a sequence $\underline{f}$ of (names of) functions. It denotes the set of all types for which implementations of the functions in $\underline{f}$ have been provided [2]. The class concept was introduced by Wadler and Blott in [9] to provide a clean framework for the integration of overloading into typed functional languages. Since the original type inference system is undecidable [7], some additional (syntactical) restrictions had to be imposed to obtain a type system where most general typings can effectively be computed. The (probably) most transparent derivation of such an algorithm in contained in [5]. Nipkow and Prehofer consider a simplified language where a class consists of all types supporting just one specific function. In order to deal with types that support several functions they introduce the notion of *sorts*. A sort $S$ is a set of classes $\{C_1, \ldots, C_k\}$. The meaning of $S$ is then the *intersection* of the classes $C_i \in S$.

For type inference, Nipkow and Prehofer suggest a three level approach: expressions, types and sorts. Thus (in some suitable context), expressions may have types whereas types may have sorts. Accordingly, they introduce two kinds of judgements. Let $\Gamma = \{\alpha_1 : S_1, \ldots, \alpha_k : S_k\}$ denote a *sort environment*, i.e., a finite mapping from type variables $\alpha_i$ to sorts $S_i$, and $E = \{x_1 : \sigma_1, \ldots, x_m : \sigma_m\}$ a *type environment*, i.e., a finite mapping from (expression) variables $x_j$ to type schemes $\sigma_j$.

The *sort judgement*:

$$\Gamma \vdash \tau : S$$

states that under assumption $\Gamma$, type $\tau$ has sort $S$, i.e., is contained in every $C \in S$.

The *type judgement* or *typing*:

$$\Gamma, E \vdash e : \tau$$

states that under the assumptions $\Gamma$ and $E$, $e$ has type $\tau$. Nipkow and Prehofer present inference rules for both kinds of judgements. For the case of "coregular" signatures, they derive an algorithm for computing a most general typing for expressions $e$ which is only slightly more complicated than the corresponding algorithm for ML. They leave it open however, to decide whether the sort assumptions $\alpha : S$ contained in $\Gamma$ can be met or not, i.e., whether or not the intersection of the classes $C \in S$ contains some (variable free) type.

The same question but stated within a different formalism has recently also been addressed in [8] where Volpano proves this problem to be NP–hard. Instead, we show:

**Theorem 1**
*Deciding emptiness of Haskell classes is DEXPTIME–complete.*

The rest of this note is organized as follows. Section 2 recalls Haskell's sort inference system from [5]. Section 3 considers the connection between deterministic topdown tree automata (DTTAs) and sort inference whereas Section 4 shows how linear–space bounded alternating Turing machines can be coded into the intersection of DTTAs. Section 5 concludes.

## 2 Sort Inference

A (Mini) Haskell program consists of a sequence of class and instance declarations followed by the expression to be evaluated. A class declaration **class** $C$ **where** $x : \forall\alpha : C.\ \sigma$ introduces the class name $C$ together with name and type scheme of the supported function $x$. An instance declaration **inst** $t : (S_1, \ldots, S_k)C$ **where** $x = e$ for a $k$–ary type constructor $t$ and class $C$ states necessary sort conditions $S_i$ on possible argument types $\tau_i$ for $t(\tau_1, \ldots, \tau_k)$ to be in class $C$, and provides a definition $e$ for $x$. Let us assume that for every pair $(t, C)$ there exists at most one instance declaration. Furthermore, let $\Delta$ denote the set of all pairs $t : (S_1, \ldots, S_k)C$, occurring in instance declarations. $\Delta$ is also called *signature*. Valid sort judgements (relative to $\Delta$) are those that can be derived by the following set of axioms and rules.

Axiom:

$$\Gamma \vdash \tau : \emptyset$$

Rules:

$$\frac{(\alpha : S) \in \Gamma}{\Gamma \vdash \alpha : S} \ (1)$$

$$\frac{\Gamma \vdash \tau : S \quad \Gamma \vdash \tau : C}{\Gamma \vdash \tau : S \cup \{C\}} \ (2)$$

$$\frac{\Gamma \vdash \tau : S \quad C \in S}{\Gamma \vdash \tau : C} \ (3)$$

Provided that $t : (S_1, \ldots, S_k)C \in \Delta$, we have:

$$\frac{\Gamma \vdash \tau_1 : S_1 \ldots \Gamma \vdash \tau_k : S_k}{\Gamma \vdash t(\tau_1, \ldots, \tau_k) : C} \ (4)$$

The given sort inference system has the deficiency that we may repeatedly apply rules (2) and (3) without deriving new information. Instead, we therefore consider the following system which only has rules (1') and (4'):

$$\frac{(\alpha : S) \in \Gamma \quad C \in S}{\Gamma \vdash' \alpha : C} \ (1')$$

Provided that $t : (S_1, \ldots, S_k)C \in \Delta$, we have:

$$\frac{\begin{array}{c}\forall C_1 \in S_1 : \Gamma \vdash' \tau_1 : C_1 \\ \vdots \\ \forall C_k \in S_k : \Gamma \vdash' \tau_k : C_k\end{array}}{\Gamma \vdash' t(\tau_1, \ldots, \tau_k) : C} \ (4')$$

By induction on the depth of proofs of sort judgements we find:

**Proposition 1** *For every sort environment $\Gamma$, type $\tau$ and class $C$,*

$$\Gamma \vdash \tau : C \quad \text{iff} \quad \Gamma \vdash' \tau : C$$

□

# 3 Topdown Automata

A *deterministic topdown tree automaton* (DTTA for short), is a quadruple $A = (Q, \Sigma, \delta, q_I)$ where $Q$ is a finite set of states, $\Sigma$ is the ranked input alphabet, $\delta : Q \times \Sigma \to Q^*$ (where the length of $\delta(q,t)$ equals the rank of $t$) is the partial transition function and $q_I$ is the initial state. A $q$–run of $A$ on some input term $\tau$ is a mapping of the nodes of $\tau$ to $Q$ where the root of $\tau$ is mapped to $q$ and if some node with label $t$ is mapped to state $p$ then the sequence of images of its successor nodes are given by $\delta(p,t)$. A $q_I$–run is called *accepting*. The language $\mathcal{L}(A)$ of terms accepted by $A$ consists of all terms with accepting runs.

**Theorem 2** *(1) For every DTTA $A$ a signature $\Delta$ can be constructed in polynomial time (even logarithmic space) such that for some class $C$, $\emptyset \vdash \tau : C$ (w.r.t. $\Delta$) iff $\tau \in \mathcal{L}(A)$.*

*(2) For every sequence of DTTAs $A_0, \ldots, A_m$ a signature $\Delta$ can be constructed in polynomial time (even logarithmic space) such that for some class $C$, $\emptyset \vdash \tau : C$ (w.r.t. $\Delta$) iff $\tau \in \bigcap_{j=0}^{m} \mathcal{L}(A_j)$.*

*(3) For every signature $\Delta$ and class $C$ an DTTA $A$ can be constructed in exponential time such that $\emptyset \vdash \tau : C$ (w.r.t. $\Delta$) iff $\tau \in \mathcal{L}(A)$.*

Since for languages defined by DTTAs, emptiness is decidable in polynomial time, Part (3) of Theorem 2 proves the upper bound in Theorem 1; Part (2) will be used to derive the lower bound.

**Proof:** For a proof of (1) assume $A = (Q, \Sigma, \delta, q_I)$. We view $\Sigma$ as a set of type constructors and $Q$ as a set of classes. Then $\Delta$ consists of all pairs $t : (\{q_1\}, \ldots, \{q_k\})q$ where $\delta(q,t) = q_1 \ldots q_k$. Using Prop. 1, it is easy to verify that $\emptyset \vdash \tau : q_I$ (w.r.t. $\Delta$) iff $\tau \in \mathcal{L}(A)$.

Now assume we are given a sequence $A_0, \ldots, A_m$ of DTTAs $A_j = (Q_j, \Sigma, \delta_j, q_{I,j})$. W.l.o.g. we may assume that the state sets of the $A_j$ are pairwise disjoint, and that the initial states do not occur in the images of the transition functions. We again view $\Sigma$ as a set of type constructors; as the set of classes we now use $\{C\} \cup Q_0 \cup \ldots \cup Q_m$ for one new class $C$. $\Delta$ consists of the following elements:

- $t : (S_1, \ldots, S_k)C \in \Delta$ provided $\delta_i(q_{I,i}, t)$ is defined for all $i$, and $S_j = \{p_{j,i} \mid 0 \le i \le m\}$ where $\delta_i(q_{I,i}, t) = p_{1,i} \ldots p_{k,i}$;
- $t : (\{q_1\}, \ldots, \{q_k\})q \in \Delta$ whenever $\delta_i(q,t) = q_1 \ldots q_k$ for some $i$.

The rules of the first item enforce that every $A_j$ must have a accepting run on a type $\tau$ in order to be in class $C$. Again, we may use Prop. 1 to prove the correctness of the construction.

It remains to consider Part (3). Assume that $\mathcal{C}$ and $\mathcal{T}$ are the sets of classes and type constructors, respectively, occurring in $\Delta$. The intuition behind the construction is as follows. During an accepting run of $A$ on $\tau$, the state at every node is supposed to record the minimal sort requirement for a subterm at this node necessary for $\tau$ to be a member of class $C$. Hence, we define $A = (Q, \mathcal{T}, \delta, q_I)$ where $Q = 2^{\mathcal{C}}$, and $q_I = \{C\}$. Moreover, $\delta(S,t)$ is defined only provided $\Delta$ contains some $t : (S_1, \ldots, S_k)C$ for every $C \in S$; in this case it is given by $\delta(S,t) = S'_1 \ldots S'_k$ where

$$S'_j = \bigcup \{S_j \mid C \in S, t : (S_1, \ldots, S_k)C \in \Delta\}$$

□

# 4 Alternating Turing Machines

The following theorem completes the proof of the lower bound in Theorem 1.

**Theorem 3** *The emptiness problem for the intersection of a sequence of DTTAs is DEXPTIME–hard.*

A corresponding remark (without detailed proof) on the emptiness problem for the intersection of arbitrary regular tree languages is contained in [1]. The only additional observation we need is that languages defined by DTTAs suffice.

**Proof:** To prove our theorem, we show: Given an alternating linear–space bounded Turing Machine $M$, and an input $w$ for $M$ of length $n$, we can construct in polynomial time (even in logarithmic space) a sequence $A_0, \ldots, A_{n+1}$ of DTTAs such that $\bigcap_{j=0}^{n+1} \mathcal{L}(A_j)$ equals the set of accepting computations of $M$ on $w$. It follows that $\bigcap_{j=0}^{n+1} \mathcal{L}(A_j) \neq \emptyset$ iff $M$ accepts $w$.

W.l.o.g. we may assume that every storage configuration of $M$ in an accepting computation is of length $n$. Also, we assume that in configurations with universal states, $M$ has exactly two successor configurations, and no transitions are possible in final states. Let $D = \{d_1, \ldots, d_k\}$ denote the tape alphabet of $M$, $Q = Q_0 \cup Q_1 \cup Q_2$ the set of states where $Q_0$ is the set of accepting states, $Q_1$ is the set of existential states, and $Q_2$ is the set of universal states; $q_0 \in Q$ is the initial state, and $\delta$ the transition relation.

Let $\Sigma = D \cup Q \cup \{\#\}$ for some new symbol $\#$ where $D$ and $Q$ are supposed to be mutually disjoint. The only symbols of rank 0 are the accepting states; the only symbols of rank 2 are the universal states; all remaining symbols have rank 1.

The position immediately to the right of $\#$ is meant to be the tape position of the read/write head of $M$. Thus, a configuration is coded as a tree $v_1\#v_2q$, $v_1\#v_2qx_1$ resp. $v_1\#v_2q(x_1,x_2)$ depending on whether $q$ is accepting, existential or universal. Let *Conf* denote the set of all configurations, and $\mathcal{R}$ the set of all (variable free) trees obtained from *Conf* by repeated substitution.

Then we define DTTAs $A_j$ as follows:

- $A_0$ is supposed to accept the set of all trees in $\mathcal{R}$ which start with the initial configuration $\#wq_0(\ldots)$;
- For $1 \leq j \leq n$, $A_j$ checks whether tape position $j$ is copied correctly;
- $A_{n+1}$ checks whether the head moves and state changes happen correctly.

It easily can be seen that all these automata exist and can be constructed in polynomial time (even using only logarithmic space). Thus, our result follows. □

# 5 Conclusion

Since coreML can be viewed as a subset of Haskell, the lower bound proven in [4, 3] also applies to Haskell in stating that deciding whether or not some Haskell program is type–correct is at least DEXPTIME–hard. Analyzing the type inference algorithm of Nipkow and Prehofer one observes that DEXPTIME is also enough to derive most general typings. The exponent in the estimation of the runtime thereby depends on the depth of nesting of **let**–constructs.

Our results complete the complexity analysis of the Haskell type system. We considered the question of deciding for a given type judgement whether or not the assumptions contained in the sort enviroment can be met or not. We found that this problem is DEXPTIME–complete as well – now however, the exponent in the runtime estimation depends on the maximal number of classes a type may be member of.

# 6 References

[1] T. Frühwirth, E. Shapiro, M. Vardi, E. Yardeni: Logic Programs as Types of Logic Programs. Proc. 6th Symp. on Logics in Computer Science, 300–309, 1991

[2] P. Hudak, S.P. Jones, P. Wadler: Report on the Programming Language Haskell: A Non–strict, Purely Functional Language. ACM SIGPLAN Notices, 27(5), 1992, Version 1.2.

[3] A.J. Kfoury, J. Tiuryn, P. Urzyczyn: An analysis of ML typability. Proc. 15–th Colloquium on Trees in Algebra and Programming, CAAP'90, Arnold (ed.), LNCS vol. 431, Springer, 206–220, 1990

[4] H.G. Mairson: Deciding ML typability is complete for deterministic exponential time. ACM Int. Conf. on *Principles of Programming Languages*, 382–401, 1990

[5] T. Nipkow, C. Prehofer: Type Checking Type Classes. ACM Int. Conf. on *Principles of Programming Languages*, 409–418, 1993

[6] H. Seidl: Deciding equivalence of finite tree automata. Proc. STACS'89, LNCS 349, pp. 480–492, 1989

[7] D. M. Volpano, G. S. Smith: On the complexity of ML typability with overloading. 5th Conf. Funct. Prog. Lang. and Comp. Arch. LNCS 523, 15–28, 1991

[8] D.M. Volpano: Haskell–style Overloading is NP–hard. IEEE Int. Conf. on *Computer Languages*, Université Paul Sabatier, Toulouse, France, May 1994

[9] P. Wadler, S. Blott: How to make *ad-hoc* polymorphism less *ad-hoc*. ACM Int. Conf. on *Principles of Programming Languages*, 60–76, 1989