A simple proof of a theorem of Statman

*Harry G. Mairson**
Cambridge Research Laboratory
Digital Equipment Corporation
Cambridge, Massachusetts 02139

Abstract

In this note, we reprove a theorem of Statman that deciding the $\beta\eta$-equality of two first-order typable λ-terms is not elementary recursive [Sta79]. The basic idea of our proof, like that of Statman's, is the Henkin quantifier elimination procedure [Hen63]. However, our coding is much simpler, and easy to understand.

1 Introduction

A well known theorem of Richard Statman states that if we have two λ-terms that are first-order typable, deciding whether the terms reduce to the same normal form is not Kalmar elementary: namely, it cannot be decided in $f_k(n)$ steps for any fixed integer $k \geq 0$, where n is the length of the two terms, and $f_0(n) = n$, $f_{t+1}(n) = 2^{f_t(n)}$. The theorem is often cited, but in contrast, its proof is not well understood. In this note, we give a simple proof of the theorem. The key idea that vastly simplifies the technical details of the proof is to use *list iteration* as a quantifier elimination procedure.

2 Preliminaries

2.1 Deciding truth of formulas in higher-order type theory

Let $\mathcal{D}_0 = \{\mathbf{true}, \mathbf{false}\}$, and define $\mathcal{D}_{k+1} = \mathit{powerset}(\mathcal{D}_k)$. Simple logical formulas usually quantify over elements of $\mathcal{D}_0$, but we consider the truth of formulas allowing higher-order quantification, that is, over the elements of $\mathcal{D}_k$, for all $k \geq 0$. Let x^k, y^k, z^k be variables allowed to range over $\mathcal{D}_k$; we define the *prime formulas* as x^0, $\mathbf{true} \in y^1$, $\mathbf{false} \in y^1$, and $x^k \in y^{k+1}$. Now consider a formula Φ built up out of prime formulas, the usual logical connectives $\vee$, $\wedge$, $\rightarrow$, $\neg$, and the quantifiers $\forall$ and $\exists$: is Φ true under the usual interpretation?

*On leave from Brandeis University, Waltham, Massachusetts. Supported in part by NSF Grant CCR-9017125, and grants from Texas Instruments and from the Tyson Foundation. This paper appeared in **Theoretical Computer Science** 103 (1992), pp. 387–394.

As shown by Meyer, this decision problem requires nonelementary time [Mey74]. Statman's theorem is a reduction to this problem: it shows how to use typed lambda calculus to simulate the logical connectives as well as a *quantifier elimination procedure* to *reduce* Φ, in the logical and λ-calculus sense, to either **true** or **false**. We indicate how *list iteration* is a straightforward way to code quantifier elimination. In addition, we give a generic reduction; that is, how to simulate an arbitrary Turing machine for nonelementary time, by reduction to our λ-calculus problem. This makes the presentation self contained.

2.2 List iteration

Let $\{x_1, x_2, \ldots, x_k\}$ be a set of λ-terms, each of first-order type α; then

$$L \equiv \lambda c : \alpha \to \tau \to \tau . \lambda n : \tau . c\ x_1\,(c\ x_2\ \ldots\ (c\ x_k\, n)\ \ldots)$$

is a λ-term of type $(\alpha \to \tau \to \tau) \to \tau \to \tau$, for *any* type τ. We abbreviate this list construction as $[x_1, x_2, \ldots, x_k]$; observe that the variables c and n abstract over the list constructors **cons** and **nil**. In the simply typed λ-calculus, list iteration can be used to implement primitive recursion. For example, given λ-terms **succ** and **0** for zero and successor on Church numerals, the length of a list of terms of type α can be computed by

$$\mathit{length} \equiv \lambda L : (\alpha \to \mathsf{Int} \to \mathsf{Int}) \to \mathsf{Int} \to \mathsf{Int}. L\ (\lambda x : \alpha.\mathbf{succ})\ \mathbf{0},$$

where $\mathsf{Int} \equiv\ (\nu \to \nu) \to \nu \to \nu$, and τ is set to Int in the above definition of L.

List iteration is ideal for realizing quantifier elimination: imagine that we code $\mathcal{D}_k$ as a λ-term $\mathbf{D}_k$ which lists all elements of $\mathcal{D}_k$, each coded appropriately as a λ-term of type Δ_k, and we have coded a Boolean function Φ as a λ-term $\hat{\Phi}$ of type $\Delta_k \to \mathsf{Bool}$. Then the truth of $\forall x^k.\Phi(x^k)$ can be coded as the λ-term $\mathbf{D}_k\ (\lambda x^k : \Delta_k.\textsc{and}\ (\hat{\Phi}\, x^k))\ \mathit{true}$, and the truth of $\exists x^k.\Phi(x^k)$ can be coded as the λ-term $\mathbf{D}_k\ (\lambda x^k : \Delta_k.\textsc{or}\ (\hat{\Phi}\, x^k))\ \mathit{false}$, where $\textsc{and}$, $\textsc{or}$, *true* and *false* are λ-terms coding up Boolean logic. Observe, for example, that the latter reduces to $\textsc{or}\ (\hat{\Phi}\, e_1)\,(\textsc{or}\ (\hat{\Phi}\, e_2)\ \ldots\ (\textsc{or}\ (\hat{\Phi}\, e_t)\, \mathit{false})\ \ldots)$, where e_j is a λ-term coding the jth element of $\mathcal{D}_k$, $1 \le j \le t = |\mathcal{D}_k|$. As we will see, the prime formulas can also be simulated using list iteration.

3 The proof

3.1 Booleans

Let σ be any first-order type, and define $\mathsf{Bool} \equiv \sigma \to \sigma \to \sigma$. The Boolean values and logical connectives are interpreted by their usual Church codings:

$$\begin{aligned}
\mathit{true} &\equiv \lambda x : \sigma.\lambda y : \sigma.x : \mathsf{Bool} \\
\mathit{false} &\equiv \lambda x : \sigma.\lambda y : \sigma.y : \mathsf{Bool} \\
\textsc{and} &\equiv \lambda p : \mathsf{Bool}.\lambda q : \mathsf{Bool}.\lambda x : \sigma.\lambda y : \sigma.p\,(qxy)\,y : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
\textsc{or} &\equiv \lambda p : \mathsf{Bool}.\lambda q : \mathsf{Bool}.\lambda x : \sigma.\lambda y : \sigma.p\,x\,(qxy) : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool} \\
\textsc{not} &\equiv \lambda p : \mathsf{Bool}.\lambda x : \sigma.\lambda y : \sigma.p\,y\,x : \mathsf{Bool} \to \mathsf{Bool} \\
\textsc{if} &\equiv \lambda p : \mathsf{Bool}.\lambda q : \mathsf{Bool}.\textsc{or}\ (\textsc{not}\ p)\,q : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}
\end{aligned}$$

3.2 Coding elements of the type hierarchy

The set $\mathcal{D}_0$ is represented as the list $\mathbf{D}_0$ containing *true* and *false*:

$$\mathbf{D}_0 \equiv \lambda c : \mathsf{Bool} \to \tau \to \tau . \lambda n : \tau . c\ \mathit{true} (c\ \mathit{false}\, n) : \ (\mathsf{Bool} \to \tau \to \tau) \to \tau \to \tau$$

We abbreviate the type of $\mathbf{D}_0$ as Δ_1; in general, let $\Delta_{k+1} \equiv \Delta_k^*$, where for any type α, we define $\alpha^* \equiv (\alpha \to \tau \to \tau) \to \tau \to \tau$, and $\Delta_0 \equiv \mathsf{Bool}$.

Next, for each integer $k > 0$, we define a λ-term $\mathbf{D}_k$ of length $\Theta(k)$ representing $\mathcal{D}_k$ as a *list* of (recursively defined codings of) all subsets of elements of $\mathcal{D}_{k-1}$ in the type hierarchy. To do so, we must introduce an explicit powerset construction, so as to build succinct terms coding these lists. First, we define a term *double* where, given an element $x : \alpha$ and a list $\ell : \alpha^{**}$ of lists of elements of type α, *double* appends ℓ to a list derived from adding x to each list in ℓ. For example, when $\alpha \equiv \mathsf{Bool}$, *double false* $[[\,], [\mathit{true}]]$ reduces to $[[\mathit{false}], [\mathit{false}, \mathit{true}], [\,], [\mathit{true}]]$.

$$\begin{aligned} \mathit{double} &\equiv \lambda x : \alpha . \lambda \ell : (\alpha^* \to \tau \to \tau) \to \tau \to \tau . \\ &\qquad \lambda c : \alpha^* \to \tau \to \tau . \lambda n : \tau . \\ &\qquad\quad \ell\ (\lambda e : \alpha^* . c\ (\lambda c' : \alpha \to \tau \to \tau . \lambda n' : \tau . c'\, x\, (e\, c'\, n')))\ (\ell\, c\, n) \\ \mathit{double} &: \alpha \to \alpha^{**} \to \alpha^{**} \end{aligned}$$

Notice that if a λ-term A^* coding a list of λ-terms of type α has type $\alpha^* \equiv (\alpha \to \tau \to \tau) \to \tau \to \tau$ for *any* τ, then A^* also has type $\alpha^* \equiv (\alpha \to \alpha^{**} \to \alpha^{**}) \to \alpha^{**} \to \alpha^{**}$. We may then define

$$\begin{aligned} \mathit{powerset} &\equiv \lambda A^* : (\alpha \to \alpha^{**} \to \alpha^{**}) \to \alpha^{**} \to \alpha^{**} . \\ &\qquad A^*\ \mathit{double}\ (\lambda c : \alpha^* \to \tau \to \tau . \lambda n : \tau . c\ (\lambda c' : \alpha \to \tau \to \tau . \lambda n' : \tau . n')\ n) \\ \mathit{powerset} &: ((\alpha \to \alpha^{**} \to \alpha^{**}) \to \alpha^{**} \to \alpha^{**}) \to \alpha^{**}. \end{aligned}$$

The function of *powerset* on lists is like that of *exponentiation* realized via iterated doubling on Church numerals, since Church numerals are just lists having *length* but containing no *data.*

Now we can succinctly define terms coding levels of the type hierarchy:

$$\begin{aligned} \mathbf{D}_1 &\equiv \mathit{powerset}\ \mathbf{D}_0 : \Delta_2 \equiv (\Delta_1 \to \tau \to \tau) \to \tau \to \tau \\ \mathbf{D}_2 &\equiv \mathit{powerset}\ \mathbf{D}_1 : \Delta_3 \equiv (\Delta_2 \to \tau \to \tau) \to \tau \to \tau \\ &\quad \ldots \\ \mathbf{D}_{n+1} &\equiv \mathit{powerset}\ \mathbf{D}_n : \Delta_{n+2} \equiv (\Delta_{n+1} \to \tau \to \tau) \to \tau \to \tau \end{aligned}$$

In the definition of $\mathbf{D}_{k+1}$, the leftmost occurance of *powerset* is given type $((\Delta_k \to \Delta_{k+2} \to \Delta_{k+2}) \to \Delta_{k+2} \to \Delta_{k+2}) \to \Delta_{k+2}$. Note that the length of each λ-term $\mathbf{D}_k$, with type information erased, grows as $\Theta(k)$. However, the length of its *normal form* grows as $\Omega(g(k))$, where $g(0) = 1$, $g(t+1) = 2^{g(t)}$.

3.3 Coding set theory in the $\mathbf{D}_k$

There is a natural idea of *equality* between elements of $\mathcal{D}_k$; when these elements are themselves sets, we can also define the idea of *subset* and of *element* of a set. We realize the prime formulas of type theory using these concepts, together with list iteration. For each integer $n > 0$, we define terms eq_n, $subset_n$, and $member_n$. When $n = 0$, we define only

$$eq_0 \equiv \lambda x^0 : \mathsf{Bool}.\lambda y^0 : \mathsf{Bool}.\textsc{or}\ (\textsc{and}\ x^0 y^0)\ (\textsc{and}\ (\textsc{not}\ x^0)(\textsc{not}\ y^0))$$

as a basis; eq_0 is just the Boolean 'iff.' For $n = k+1$, we set $\tau \equiv \mathsf{Bool}$ in the above definitions of Δ_j, so that λ-terms of type Δ_j can be used to iterate a Boolean function, and define

$$\begin{aligned}
member_{k+1} &\equiv \lambda x^k : \Delta_k.\lambda y^{k+1} : \Delta_{k+1}.\\
&\qquad y^{k+1}\ (\lambda y^k : \Delta_k.\textsc{or}\ (eq_k\ x^k\ y^k))\ false\\
&\qquad\quad : \Delta_k \to \Delta_{k+1} \to \mathsf{Bool}\\
subset_{k+1} &\equiv \lambda x^{k+1} : \Delta_{k+1}.\lambda y^{k+1} : \Delta_{k+1}.\\
&\qquad x^{k+1}\ (\lambda x^k : \Delta_k.\textsc{and}\ (member_{k+1}\ x^k\ y^{k+1}))\ true\\
&\qquad\quad : \Delta_{k+1} \to \Delta_{k+1} \to \mathsf{Bool}\\
eq_{k+1} &\equiv \lambda x^{k+1} : \Delta_{k+1}.\lambda y^{k+1} : \Delta_{k+1}.\\
&\qquad (\lambda op : \Delta_{k+1} \to \Delta_{k+1} \to \mathsf{Bool}.\textsc{and}\ (op\ x^{k+1}y^{k+1})(op\ y^{k+1}x^{k+1}))\ subset_{k+1}\\
&\qquad\quad : \Delta_{k+1} \to \Delta_{k+1} \to \mathsf{Bool}
\end{aligned}$$

The λ-terms defining $member_k$, $subset_k$, and eq_k, with type information erased, all have length $\Theta(k)$. Note how the trick in the definition of eq_{k+1} is essential: writing $subset_{k+1}$ twice causes exponential blowup in the term size.

The above definitions give a typed λ-calculus interpretation to all the logical formulas in type theory, in the spirit of their standard logical meaning. In particular, **true** and **false** are interpreted as *true* and *false*, and the prime formula $x^k \in y^{k+1}$ is interpreted as $member_{k+1}\ x^k\ y^{k+1}$, of type $\Delta_0 \equiv \mathsf{Bool}$, reducing to either *true* or *false* when x^k and y^{k+1} are closed λ-terms. The logical connectives, interpreted by their Church codings, take arguments of type Bool, producing terms of type Bool. Quantifier elimination, as described earlier, interprets $\forall x^k.\Phi(x^k)$ as the iterated conjunction $\mathbf{D}_k\ (\lambda x^k : \Delta_k.\textsc{and}\ (\hat{\Phi}\ x^k))\ true$, where $\hat{\Phi}$ is the interpretation of Φ; the complementary interpretation of $\exists x^k.\Phi(x^k)$ is the iterated disjunction $\mathbf{D}_k\ (\lambda x^k : \Delta_k.\textsc{or}\ (\hat{\Phi}\ x^k))\ false$.

As a consequence, a formula Φ in type theory is true if and only if its typed λ-calculus interpretation $\hat{\Phi} : \mathsf{Bool}$ is $\beta\eta$-equivalent to $true \equiv \lambda x : \sigma.\lambda y : \sigma.x : \mathsf{Bool}$.

3.4 Remarks on separation theorems in λ-calculus

It is instructive to realize how the notion of *nonrecursive* in the context of *untyped* λ-calculus functions precisely in the same manner as the notion of *non Kalmar-elementary* functions in the first-order typed λ-calculus [Sta82]. Scott's undecidability theorem (see, e.g., [HS86]) states that no two nonempty, disjoint sets of λ-terms are recursively separable: given such sets Λ and $\overline{\Lambda}$ of λ-terms closed under $\beta\eta$-equality, no algorithm can decide, given an arbitrary x chosen from $\Lambda \cup \overline{\Lambda}$, whether $x \in \Lambda$, or $x \in \overline{\Lambda}$. Statman's Theorem easily yields a similar corollary, where Λ and $\overline{\Lambda}$ contain terms of some fixed type τ, if we replace "recursive" with "Kalmar elementary."

The proof is simple. Let a and b be arbitrary elements of Λ and $\overline{\Lambda}$, respectively, and let $\mathbf{E} : \tau \to \tau \to \tau$ be a λ-term coding an expression E in higher-order type theory. Then $\mathbf{E}\,a\,b$ is $\beta\eta$-equivalent to a if E is true, and to b if E is false. Hence deciding membership of $\mathbf{E}\,a\,b$ in Λ or $\overline{\Lambda}$ is as hard as deciding the truth of E, which cannot be computed in elementary time.

Statman gives as well another corollary [Sta82]. Let Λ be a set of λ-terms of fixed type τ, closed under $\beta\eta$-equality, where membership in Λ is decidable in elementary time. Then Λ contains all or none of the terms of type τ. The proof is trivial: suppose by contradiction that Λ is nonempty, yet there exists a term b of type τ not in Λ. Let $\overline{\Lambda}$ be the λ-terms of type τ which are $\beta\eta$-equivalent to b, and repeat the argument of the previous corollary.

4 A generic reduction

To complete the exposition, we describe how type theory — equivalently, first-order typed λ-calculus — can be used to simulate an arbitrary Turing machine for nonelementary time.

4.1 Basic arithmetic in type theory

Since $|\mathcal{D}_{k+1}| = 2^{|\mathcal{D}_k|}$, it is easy to show that each element $x^{k+1} \in \mathcal{D}_{k+1}$ can be thought of as an integer, where the elements of x^{k+1} are just the *bit positions* set to 1 in its binary encoding. We can then define $<_{k+1}$ and $\mathbf{succ}_{k+1}$ over these elements. As a consequence, simulating a Turing machine is easy: successor is used to move the tape head.

To define a total order $<_k$, we take

$$\begin{aligned}
x^0 <_0 y^0 \;&\equiv\; \neg x^0 \wedge y^0 \\
x^{k+1} <_{k+1} y^{k+1} \;&\equiv\; \exists z^k . z^k \in y^{k+1} \wedge z^k \notin x^{k+1} \\
&\qquad \wedge \forall w^k . z^k <_k w^k \to (w^k \in x^{k+1} \leftrightarrow w^k \in y^{k+1})
\end{aligned}$$

(Translation: $x < y$ if the zth bit in y is 1, but in x is 0, and the bits of higher order than z are identical in x and y.) Successor is then defined as:

$$\begin{aligned}
\mathbf{succ}_{k+1}(x^{k+1}, y^{k+1}) \;&\equiv\; \exists z^k . z^k \in y^{k+1} \wedge z^k \notin x^{k+1} \\
&\qquad \wedge \forall w^k . w^k <_k z^k \to (w^k \in x^{k+1} \wedge w^k \notin y^{k+1}) \\
&\qquad \wedge \forall w^k . z^k <_k w^k \to (w^k \in x^{k+1} \leftrightarrow w^k \in y^{k+1})
\end{aligned}$$

(Translation: $y = x + 1$ if the zth bit of y is 1, and in x is 0; for the bits w of lower order than z, the wth bit of x is 1 and of y is 0, and the bits of higher order than z are identical in x and y. The zth bit is where the 'carry' propagates.)

4.2 Simulating a Turing machine

An element $x^{n+1} \in \mathcal{D}_{n+1}$ can code the tape contents, where the tape is of length $|\mathcal{D}_n|$, and tape cells hold a 0 (**true**) or 1 (**false**). x^{n+1} can also code the head position, as long as $|x^{n+1}| = 1$. An ordered pair $\langle x^{n+1}, y^{n+1}\rangle$ coding tape contents and head position can be represented in the standard set-theoretic way as $\{\{\{\}, x^{n+1}\}, \{y^{n+1}\}\} \in \mathcal{D}_{n+3}$; if we code the (finite) machine state into x^{n+1}, then a Turing machine ID can be represented as an element of $\mathcal{D}_{n+3}$. Using the logic of type theory, we can now code a binary *relation* $\tilde{\delta} \subseteq \mathcal{D}_{n+3} \times \mathcal{D}_{n+3}$, where $\tilde{\delta}(ID, ID')$ means ID' is reachable from ID in one machine transition. The logical specification of $\tilde{\delta}$ is straightforward, more or less on the level of the detailed coding in Cook's Theorem [Coo71, GJ79]. Let $\hat{\delta} : \Delta_{n+3} \rightarrow \Delta_{n+3} \rightarrow \mathsf{Bool}$ be the λ-calculus interpretation of $\tilde{\delta}$; instantiating $\mathsf{Bool} \equiv \sigma \rightarrow \sigma \rightarrow \sigma$ in this type as $\Delta_{n+3} \rightarrow \Delta_{n+3} \rightarrow \Delta_{n+3}$, we can define the transition *function* $\delta : \Delta_{n+3} \rightarrow \Delta_{n+3}$ as:

$$\delta \quad\equiv\quad \lambda ID : \Delta_{n+3}.\mathbf{D}_{n+3}\ (\lambda ID' : \Delta_{n+3}.\lambda ID'' : \Delta_{n+3}.(\hat{\delta}\ ID\, ID')\ ID'\ ID'')\ \mathit{emptyset}_{n+3}$$

(Using the list $\mathbf{D}_{n+3}$ of putative IDs, return the *leftmost* element ID' of the list where $\hat{\delta}\ ID\, ID'$ reduces to *true*. The term $\mathit{emptyset}_{n+3}$ is an arbitrary element of type Δ_{n+3}—the empty set of that type will do as well as any other term.)

Now that we have a term δ realizing the transition function, the rest is easy, and here the full power of the simply typed λ-calculus comes center stage: we use the Church numerals to iterate δ. Writing $\overline{2} \equiv \lambda s.\lambda z.s(sz)$, we have the typing

$$\overline{2}\overline{2}\cdots\overline{2} : (\Delta_{n+3} \rightarrow \Delta_{n+3}) \rightarrow \Delta_{n+3} \rightarrow \Delta_{n+3},$$

where

$$C \equiv \overline{2}\overline{2}\cdots\overline{2}\ \delta \quad \rhd_\beta \quad \lambda ID.\delta(\delta(\cdots(\delta\ ID)\cdots)) : \Delta_{n+3} \rightarrow \Delta_{n+3}.$$

The rightmost $\overline{2}$ has type $(\Delta_{n+3} \rightarrow \Delta_{n+3}) \rightarrow \Delta_{n+3} \rightarrow \Delta_{n+3}$; if the jth rightmost $\overline{2}$ has type κ, then the $(j+1)$st rightmost $\overline{2}$ has type $\kappa \rightarrow \kappa$. If there are n occurances of $\overline{2}$, there are $g(n)$ applications of δ, where $g(0) = 1$, $g(t+1) = 2^{g(t)}$. Apply C to another λ-term coding an initial ID, extract the final state, and check if it is an accepting state: the answer is of type Bool.

5 Final comments

Just as philosophy is said to be a long footnote to Plato, complexity results of this genre are a footnote to Cook's Theorem. The basic insight of Cook was that logical formulas could be succinct representatives of machine computations, and from this came a characterization of NP-completeness: a propositional formula existentially quantified over Booleans. When the quantifiers were allowed to alternate, more expressive power was gained, with completeness results for the polynomial-time hierarchy, and ultimately PSPACE-completeness. By successively increasing the range of quantification to sets of Booleans, sets of sets of Booleans, etc., it was possible to quantify over Boolean *functionals*, capturing more and more powerful complexity classes.

Statman's Theorem just uses the typed λ-calculus to realize the quantifier elimination procedure of Henkin [Hen63] on these succinct formulas. The recent results on complexity of type inference for higher-order typed lambda calculi [KMM91, HM91] follow a similar development, except that logical characterizations of computation are replaced by characterizations based on first-order unification. Although details of the type inference arguments are technically more complicated and quite different from Statman's result and its various analogues, the structural similarity — with virtually identical complexity-theoretic consequences — is that higher-order quantification is used to succinctly compose functions, and to generate long reduction sequences.

References

[Coo71] S. A. Cook. *The complexity of theorem-proving procedures.* **3rd Annual ACM Symposium on the Theory of Computing,** pp. 151–158.

[GJ79] M. R. Garey and D. S. Johnson. **Computers and Intractibility: A Guide to the Theory of NP-Completeness.** W. H. Freeman, 1979.

[HM91] F. Henglein and H. G. Mairson. *The complexity of type inference for higher-order typed lambda calculi.* **18th ACM Symposium on the Principles of Programming Languages**, January 1991, pp. 119–130.

[Hen63] L. Henkin. *A theory of propositional types.* **Fundamenta Mathematicae** 52, 1963, pp. 323–344.

[HS86] J. R. Hindley and J. P. Seldin. **Introduction to Combinators and λ-Calculus.** Cambridge University Press, 1986.

[KMM91] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. *Unification and ML type reconstruction.* In **Computational Logic: Essays in Honor of Alan Robinson,** ed. J.-L. Lassez and G. Plotkin, MIT Press, 1991, pp. 444–478.

[Mey74] A. R. Meyer. *The inherent computational complexity of theories of ordered sets.* **Proceedings of the International Congress of Mathematicians**, 1974.

[Sta79] R. Statman. *The typed λ-calculus is not elementary recursive.* **Theoretical Computer Science** 9, 1979, pp. 73–81.

[Sta82] R. Statman. *Completeness, invariance, and λ-definability.* **Journal of Symbolic Logic** 47:1, 1982, pp. 17–26.