J. Functional Programming **4** (4): 435–477, October 1994

The complexity of type inference for higher-order typed lambda calculi†

FRITZ HENGLEIN‡

DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen, Denmark

HARRY G. MAIRSON§

Computer Science Department, Brandeis University, Waltham, MA 02254, USA

Abstract

We analyse the computational complexity of type inference for untyped λ-terms in the second-order polymorphic typed λ-calculus (F_2) invented by Girard and Reynolds, as well as higher-order extensions $F_3, F_4, \ldots, F_\omega$ proposed by Girard. We prove that recognising the F_2-typable terms requires exponential time, and for F_ω the problem is non-elementary. We show as well a sequence of lower bounds on recognising the F_k-typable terms, where the bound for F_{k+1} is exponentially larger than that for F_k.

The lower bounds are based on generic simulation of Turing Machines, where computation is simulated at the expression and type level simultaneously. Non-accepting computations are mapped to non-normalising reduction sequences, and hence non-typable terms. The accepting computations are mapped to typable terms, where higher-order types encode reduction sequences, and first-order types encode the entire computation as a circuit, based on a unification simulation of Boolean logic. A primary technical tool in this reduction is the composition of polymorphic functions having different domains and ranges.

These results are the first nontrivial lower bounds on type inference for the Girard/Reynolds system as well as its higher-order extensions. We hope that the analysis provides important combinatorial insights which will prove useful in the ultimate resolution of the complexity of the type inference problem.

Capsule review

The polymorphic λ-calculi $F_2, F_3, \ldots, F_\omega$ form a useful foundation for the study of modern programming languages. Since the utility of a language's type system depends heavily on being able to ensure type correctness at compile time, the study of the complexity of type inference for F_2, $F_3, \ldots, F_\omega$ is well motivated. This paper takes a significant step forward by establishing interesting lower bounds for type inference for this class of languages. Although the

† A preliminary version of this work appeared in the *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, 1991, pp. 119–130.

‡ Research performed at Vaakgroep Informatica, University of Utrecht, and Computer Science Department, New York University, supported in part by Office of Naval Research grant N00014-90-J-1110.

§ Supported by National Science Foundation Grants CCR-9017125 and CCR-9216185, Office of Naval Research Grant N00014-93-1-1015, and the Tyson Foundation. Part of this work was done while the author was on leave at the Cambridge Research Laboratory of Digital Equipment Corporation.

decidability of type inference is left open, the lower bounds are non-trivial, and grow as language expressiveness grows. In addition, the proof methods used to establish the results are themselves interesting. In particular, the technique of encoding a Turing Machine within a language's type system used previously to establish complexity results for type inference for ML is used again here, although with some subtle differences. The reader will find this proof method both fascinating and mind-boggling, but more importantly, entirely convincing.

1 Introduction

One of the outstanding open problems in programming language theory and type theory is the decidability of type inference for the *second order polymorphic typed λ-calculus* invented by Jean-Yves Girard (1972) and John Reynolds (1974). More precisely, does there exist an effective procedure which, given an untyped λ-term, can decide whether the term is typable in the Girard/Reynolds system? If so, and the term is typable, can the algorithm produce the required type information?

While this decision problem remains tantalisingly open, we present techniques which can be used to prove significant *lower bounds* on the complexity of type inference for the Girard/Reynolds system, also called F_2, as well as higher-order extensions $F_3, F_4, \ldots, F_\omega$ proposed by Girard. In particular, we show that recognising the F_2-typable terms requires exponential time, and for F_ω the problem is non-elementary. We show as well a sequence of lower bounds on recognising the F_k-typable terms, k integer, where the bound for F_{k+1} is exponentially larger than that for F_k.

These results are the first non-trivial lower bounds on type inference for the Girard/Reynolds system as well as its higher-order extensions. We hope that the analysis provides important combinatorial insights which will prove useful in the ultimate resolution of the complexity of the type inference problem.

The problem of type inference is one of both theoretical and practical interest. Following the insights of Landin (1966), Strachey (1973), Penrose[†], and others, the untyped λ-calculus has long been recognised as not merely Turing-complete, but a syntactically natural foundation for the design of programming languages. The process of β-reduction is a simulation of computation and function call, while normal forms correspond to final returned answers.

Types augment programming languages with additional guarantees about resultant computational behaviour. For instance, *static typing* as in Pascal requires explicit typing by the programmer, but allows all type checking to occur at compile time, with the guarantee that no compiled program will 'go wrong' at run-time due to type mismatches. The price paid for this guarantee is a loss of *parametric polymorphism*

[†] In his 1977 Turing Award lecture, as well as his Foreword to Joseph Stoy's book on denotational semantics, Dana Scott mentions that it was physicist Roger Penrose who pointed Strachey in the direction of the λ-calculus as a useful device for describing programming language semantics (Scott, 1977; Stoy, 1977). Scott quotes Strachey as having written, 'The λ-calculus has been widely used as an aid in examining the semantics of programming languages precisely because it brings out very clearly the connections between a name and the entity it denotes, even in cases where the same name is used for more than one purpose. The earliest suggestion that λ-calculus might be useful in this way that has come to our notice was in a verbal communication from Roger Penrose to [me] in about 1958. At the time this suggestion fell on stony ground and had virtually no influence.' (Stoy, 1977, p. xxiii).

('code reuse'), so that programs designed for abstract data types must be recorded for each type on which they are used. As an example, the computation of the identity function $I(x) = x$ is certainly data-independent, yet its realisation in Pascal demands identical code with different type declarations for the identity function on integers, booleans, arrays of length 10 of characters, etc. – all this redundancy merely to please the compiler.

A powerful extension to this programming language methodology was proposed by Robin Milner, namely a theory of *type polymorphism* for achieving code reuse, while retaining the benefits of static typing. He gave an algorithm which, presented with an untyped program, could construct the most general type information, known as the *principal type* (Hindley, 1969; Damas and Milner, 1982) for the program (Milner 1978). These insights are implemented in the ML programming language (Harper *et al.*, 1990) as well as a variety of the other functional languages (Hudak and Wadler, 1988; Turner, 1985). The principal type of an ML program provides an important functional specification of the program, describing how it can be used by other programs; as such, types are useful as specifications, and to facilitate incremental compilation. The ML module system is an elegant realisation of these basic intuitions.

We view the type system of the ML language as not merely an example of successful software engineering. Because it provides a simplified, yet powerful subset of the polymorphic features inherent in more sophisticated type systems, it has served as an ideal initial subject in the investigation of the computational combinatorics of typed lambda calculi. The 'Core ML' language comprising simply typed λ-calculus with polymorphism (as embodied in `let`) enjoys the *strong normalisation property*: typable programs are guaranteed to terminate under all reduction strategies.† Reconstructing the type of an (untyped) ML expression is thus in essence the synthesis of a termination proof.

Of special interest here is the fact that typable ML expressions are, modulo syntactic sugar, a non-trivial subset of the λ-terms typable in $F_2, F_3, \ldots, F_\omega$. Furthermore, all of these type systems enjoy strong normalisation. Since λ-terms typable in F_k are also typable in F_{k+1}, we may regard the higher-order type systems as more and more powerful expression languages in which to encode termination proofs. It is natural to expect that greater expressiveness may facilitate the extraction of stronger lower bounds; proving lower bounds on type inference for F_ω should at least be *easier* than for F_2.

We note, however, that F_ω is not simply an esoteric variation on F_2, since it has been proposed as the mathematical foundation for a new generation of typed functional programming languages, for example Cardelli's (1989) language Quest, and the language LEAP of Pfenning and Lee (1989). The practical use of such languages, however, is considerably hampered by the absence of any type inference algorithm, forcing the programmer into detail and debugging of types as well as of the program. Some arguments have been made that the problem to be solved is *partial* type inference, where the programmer supplies constraints in the form of type information for certain fragments of the program. In view of the undecidability results of Pfenning (1988), from a theoretical perspective, it is clear that partial type inference is not easier

† As a consequence, ML is in practice augmented with a set of typed fixpoint operators.

than pure type inference; in fact, pure type inference might be decidable. Because no progress on (pure) type inference for F_2 and similarly sophisticated typed lambda calculi has seemed possible, there has no doubt been a redirection of research attention elsewhere, where the promise of success has been more encouraging. We intend to refocus attention on type inference by an incremental analysis of its combinatorics, suggesting that there is indeed hope for a better understanding of the problem.

The lower bounds we present on type inference are all proved via *generic reductions*, where an arbitrary Turing Machine (henceforth, TM) M with input x of length n is simulated by a λ-term $\Psi_{M,x}$, such that M accepts x in time $f(n)$ iff $\Psi_{M,x}$ is typable. In constructing strong lower bounds, the challenge is to encode as rapidly increasing an $f(n)$ as possible, while constraining the transducer reducing M and x to $\Psi_{M,x}$ to run in logarithmic space. By the time hierarchy theorem (Hartmanis and Stearns, 1965; Hopcroft and Ullman, 1979), these complexity-class relativised hardness bounds translate (via diagonalisation arguments) to non-relativised bounds. For instance, the DTIME[2^n]-hardness bound for typability in F_2 implies a $\Omega(c^n)$ lower bound for some constant $c > 1$.

The structure of $\Psi_{M,x}$ is basically a consequence of the following proposition (Kanellakis *et al.*, 1991):

Proposition 1.1
Given any strongly normalising λ-term E, the problem of determining whether the normal form of E is first-order typable is DTIME[$f(n)$]*-hard, for any total recursive function $f(n)$.*

Proof
(Sketch) Given a TM M halting in $f(n)$ steps on input x of length n, construct a λ-term δ encoding the transition function of M, so that if y codes a machine ID, (δy) β-reduces to a λ-term encoding the ID *after* a state transition. Let $\bar{f}$ be the Church-numeral encoding of f, $\bar{n}$ be the Church numeral for n, and ID_0 be the encoding of the initial ID. Consider the typing of the normal form of $E' \equiv \bar{f}\bar{n}\delta\, ID_0 \rhd \overline{f(n)}\,\delta\, ID_0 \rhd \delta^{f(n)}\, ID_0$, where $\rhd$ denotes a sequence of β-reductions. The normal form of E' codes a machine ID after $f(n)$ transitions; construct E (using E' as a subterm) to force a mistyping in the case of non-acceptance. □

The fundamental contribution of this paper is to detail what is absent from this proof sketch, strengthening the statement of the proposition to concern the typability of E (instead of its normal form) in the various systems F_k, while weakening the proposition by restricting the possible asymptotic growth of $f(n)$.† The key insight of the lower bound constructions is the understanding of how a sophisticated type system can be used to type terms having long reduction sequences to normal form.

The remainder of the paper details our elaboration on Proposition 1.1, mixed with some short tutorials on the type systems under study, where we have attempted to provide useful and informal intuition along with the usual parcels of formal inference

† Observe that these type systems preserve typings under β-reduction, the so-called *subject reduction lemma* (see, for example, Hindley and Seldin, 1986).

rules. In section 2, we briefly outline F_2, the second order polymorphic typed λ-calculus, and in section 3 we present an exposition of the DTIME[2^{n^k}]-hardness bound for typability in F_2. As corollaries, we present simple proofs of the DTIME[2^{n^k}]-completeness of recognising typable ML programs, as well as an utterly transparent proof that first-order unification is PTIME-complete. The latter is especially perspicuous in that it replaces the ingenious gadgets of Dwork *et al.* (1984) with classical combinators from the λ-calculus, and shows how the theorem might have been proved by merely writing a simple, `let`-free ML program.

In section 4, we provide a description of the systems $F_3, F_4, \ldots, F_\omega$ generalising F_2, with emphasis on the significance of *kinds* in these systems. In section 5 we prove the non-elementary lower bound on typability for F_ω, and show the connections between this bound and related lower bounds for the F_k. Our tutorial material was in many ways inspired by the presentation of Pierce *et al.* (1989), which we enthusiastically recommend to anyone desiring a readable introduction to programming in higher-order typed λ-calculi.

2 The second order polymorphic typed λ-calculus (F_2)

F_2 is best introduced by a canonical example: the identity function. In F_2, we write the typed polymorphic identity function† as $Id \equiv \Lambda\alpha{:}*.\lambda\mathbf{x}{:}\alpha.\mathbf{x}$. The $\lambda\mathbf{x}.\mathbf{x}$ should be familiar; the $\Lambda\alpha{:}*$ denotes abstraction over *types*‡. For instance, given a type Int encoding integers, we can represent the identity function for integers as:

$$Id[\mathsf{Int}] \equiv (\Lambda\alpha{:}*.\lambda\mathbf{x}{:}\alpha.\mathbf{x})[\mathsf{Int}] \rhd_\beta \lambda\mathbf{x}{:}\mathsf{Int}.\mathbf{x}$$

The $\rhd_\beta$ indicates β-reduction at the *type* level, where Int is substituted for free occurrences of α in the body $\lambda\mathbf{x}{:}\alpha.\mathbf{x}$. (We will henceforth use $\rhd$ to mean the reflexive, transitive closure of relation defined by $\rhd_\beta$.) Given a type Bool encoding Boolean values, we may similarly write $Id[\mathsf{Bool}]$ to get the identity function for booleans. In short, Id is a *polymorphic* function which may be parameterised with a given type to derive an identity function for that type. We write the type of Id as $Id \in \Delta\alpha{:}*.\alpha \to \alpha$, where Δ (sometimes written as $\forall$) represents universal quantification over types. Church numerals may be typed in a similar fashion, for example:

$$\bar{0} \equiv \Lambda\alpha{:}*.\lambda\mathbf{f}{:}\alpha\to\alpha.\lambda\mathbf{x}{:}\alpha.\mathbf{x}$$
$$\bar{1} \equiv \Lambda\alpha{:}*.\lambda\mathbf{f}{:}\alpha\to\alpha.\lambda\mathbf{x}{:}\alpha.\mathbf{fx}$$
$$\bar{2} \equiv \Lambda\alpha{:}*.\lambda\mathbf{f}{:}\alpha\to\alpha.\lambda\mathbf{x}{:}\alpha.\mathbf{f(fx)}$$

where the type of all Church numerals is:

$$\mathsf{Int} \equiv \Delta\alpha{:}*.(\alpha\to\alpha)\to\alpha\to\alpha$$

Here we see how Int need not be a built-in 'constant' type, but can actually be expressed as the type of Church's coding of integers. In the untyped λ-calculus, we

† For clarity, we show expression variables in **boldface** and type variables in *italic* when both occur in the same expression.

‡ For the moment, it seems that the $*$ is redundant and unnecessary, though in this case, it means that the type variables α ranges over types. In generalisations of F_2 this notation will become more meaningful, where variables will be able to range over *functions* of types.

realise the exponent n^m by reducing the expression $(\lambda f.\lambda x.f^m x)(\lambda f.\lambda x.f^n x)$ to normal form. The type language of F_2 is sufficiently expressive to type this reduction sequence, given an initial assumption that the two terms are typed as Church numerals of type Int:

$$
\begin{aligned}
&\Lambda\tau{:}*.(\bar{\mathbf{m}}[\tau\to\tau])(\bar{\mathbf{n}}[\tau])\\
&\quad\equiv \Lambda\tau{:}*.((\Lambda\alpha{:}*.\lambda\mathbf{f}{:}\alpha\to\alpha.{:}\mathbf{x}{:}\alpha.\mathbf{f}^m\mathbf{x})[\tau\to\tau])\\
&\qquad\qquad((\Lambda\alpha{:}*.\lambda\mathbf{g}{:}\alpha\to\alpha.\lambda\mathbf{y}{:}\alpha.\mathbf{g}^n\mathbf{y})[\tau])\\
&\quad\rhd_\beta \Lambda\tau{:}*.(\lambda\mathbf{f}{:}(\tau\to\tau)\to\tau\to\tau.\lambda\mathbf{x}{:}\tau\to\tau.\mathbf{f}^m\mathbf{x})(\lambda\mathbf{g}{:}\tau\to\tau.\lambda\mathbf{y}{:}\tau.\mathbf{g}^n\mathbf{y})\\
&\quad\rhd_\beta \Lambda\tau{:}*.\lambda\mathbf{x}{:}\tau\to\tau.(\lambda\mathbf{g}{:}\tau\to\tau.\lambda\mathbf{y}{:}\tau.\mathbf{g}^n\mathbf{y})^m\mathbf{x}\\
&\quad\rhd_\beta \Lambda\tau{:}*.\lambda\mathbf{x}{:}\tau\to\tau.(\lambda\mathbf{g}{:}\tau\to\tau.\lambda\mathbf{y}{:}\tau.\mathbf{g}^n\mathbf{y})^{m-1}(\lambda\mathbf{y}{:}\tau.\mathbf{x}^n\mathbf{y})\\
&\quad\rhd_\beta \Lambda\tau{:}*.\lambda\mathbf{x}{:}\tau\to\tau.(\lambda\mathbf{g}{:}\tau\to\tau.\lambda\mathbf{y}{:}\tau.\mathbf{g}^n\mathbf{y})^{m-2}(\lambda\mathbf{y}{:}\tau.\mathbf{x}^{n^2}\mathbf{y})\\
&\qquad\ldots\\
&\quad\rhd_\beta \Lambda\tau{:}*.\lambda\mathbf{x}{:}\tau\to\tau.\lambda\mathbf{y}{:}\tau.\mathbf{x}^{n^m}\mathbf{y}
\end{aligned}
$$

Observe that the normal form is also of type Int.† Hence we might define:

$$expt \equiv \lambda\bar{\mathbf{m}}{:}\mathsf{Int}.\lambda\bar{\mathbf{n}}{:}\mathsf{Int}.\Lambda\tau{:}*.(\bar{\mathbf{m}}[\tau\to\tau])(\bar{\mathbf{n}}[\tau])$$

Church numerals are merely polymorphic iteration functions which compose other functions having the same domain and range, while exponentiation is just a higher order mechanism for constructing such function composers. What happens when we want to compose a function having a *different* domain and range? We will show that the answer to this question is crucial to the development of lower bounds.

2.1 Syntax and inference rules of F_2

The syntax of F_2 terms is given by the following grammar:

$$\mathscr{T} ::= \alpha \mid \mathscr{T}\to\mathscr{T} \mid \Delta\alpha{:}*.\mathscr{T}$$

$$\mathscr{E} ::= x \mid \lambda x{:}\mathscr{T}.\mathscr{E} \mid \mathscr{E}\mathscr{E} \mid \Lambda\alpha{:}*.\mathscr{E} \mid \mathscr{E}[\mathscr{T}]$$

The non-terminals α and x range over a set of *type variables* and *expression variables*, respectively, while the non-terminals $\mathscr{T}$ and $\mathscr{E}$ define the set of *types* and the set of *expressions*.

Observe that types are either type variables, function types, or quantified types, where we are able to quantify only over type variables. Expressions are either expression variables, λ-abstractions over variables *of type* $\mathscr{T}$, function applications of an expression to another expression, Λ-abstractions over *type variables* appearing in an expression (of *kind* $*$; for more details, see section 4), or applications of an expression to a type (i.e. a *parameterisation*, as in the example above of the identity function).

The terms generated from this grammar are sometimes called *raw terms*, and contain a particular subset called the *typed terms*; we think of the latter as the terms

† Here, we allow α-renaming of Λ-bound variables at the type level.

that 'make sense'. We distinguish this sense of a term by a *type judgement* $\Gamma \vdash e \in \tau$, read 'in context Γ, term e has type τ'. Type judgements are derived via a set of *inference rules* characteristic to F_2; we adopt the basic presentation of Pierce *et al.* (1989).

The first inference rules define a well-formed *context*. A context is a function from a finite domain of expression variables to types. When Γ is a context, we write $\Gamma[x:\alpha]$ to mean the function identical to Γ, except that its value on x is α:

$$(\textit{Env-}\langle\rangle) \qquad \frac{}{\mathsf{wf}(\langle\rangle)}$$

$$(\textit{Env-term}) \qquad \frac{\Gamma \vdash \tau \in *}{\mathsf{wf}(\Gamma[x:\tau])}$$

$$(\textit{Env-type}) \qquad \frac{\mathsf{wf}(\Gamma)}{\mathsf{wf}(\Gamma[\alpha:*])} \qquad \alpha \notin \mathsf{FV}(\Gamma)$$

The next three rules define the well-formedness of types:

$$(\textit{Type-var}) \qquad \frac{\mathsf{wf}(\Gamma)}{\Gamma \vdash \alpha \in *} \qquad \Gamma(\alpha) = *$$

$$(\textit{Wff-}\rightarrow) \qquad \frac{\Gamma \vdash \tau \in * \quad \Gamma \vdash \tau' \in *}{\Gamma \vdash \tau \rightarrow \tau' \in *}$$

$$(\textit{Wff-}\Delta) \qquad \frac{\Gamma[\alpha:*] \vdash \tau \in *}{\Gamma \vdash \Delta\alpha:*.\tau \in *}$$

The last rules define the well-typedness of expressions, in a syntax-directed fashion:

$$(\textit{Var}) \qquad \frac{\Gamma \vdash \tau \in *}{\Gamma \vdash x \in \tau} \qquad \Gamma(x) = \tau$$

$$(\rightarrow\textit{-int}) \qquad \frac{\Gamma \vdash \tau \in * \quad \Gamma[x:\tau] \vdash e \in \tau'}{\Gamma \vdash \lambda x:\tau.e \in \tau \rightarrow \tau'}$$

$$(\rightarrow\textit{-elim}) \qquad \frac{\Gamma \vdash e \in \tau \rightarrow \tau' \quad \Gamma \vdash e' \in \tau}{\Gamma \vdash ee' \in \tau'}$$

$$(\Delta\textit{-int}) \qquad \frac{\Gamma[\alpha:*] \vdash e \in \tau}{\Gamma \vdash \Lambda\alpha:*.e \in \Delta\alpha:*.\tau} \qquad \alpha \notin \mathsf{FV}(\Gamma)$$

$$(\Delta\textit{-elim}) \qquad \frac{\Gamma \vdash e \in \Delta\alpha:*.\tau' \quad \Gamma \vdash \tau \in *}{\Gamma \vdash e[\tau] \in \tau'[\alpha/\tau]}$$

When giving a type judgement in an empty context, we write $e \in \tau$ rather than $\langle\rangle \vdash e \in \tau$. In addition, we adopt the following non-standard convention: when writing F_2 expressions, expression variables will appear in **boldface,** while we omit boldface when discussing the *erasure* of types in the expression. For example, in this slight abuse of language, we will write $\lambda x.x \in \Delta\alpha:*.\alpha \rightarrow \alpha$ as well as $\Lambda\alpha:*.\lambda\mathbf{x}:\alpha.\mathbf{x} \in \Delta\alpha:*.\alpha \rightarrow \alpha$.

3 An exponential lower bound on F_2 *type inference*

3.1 Paradise lost: lessons learned from ML

It has been known for some time that type inference for the simply-typed (first-order) λ-calculus can be solved in polynomial time. A simple and elegant exposition of this fact can be found in Wand (1987), where a syntax-directed algorithm is given that transforms an untyped λ-term into a linear sized set of *type equations* of the form $X = Y$ and $X = Y \to Z$, such that the solution of the equations via *unification* (Robinson, 1965; Paterson and Wegman 1978) determines the principal type of the term.

In progressing from this language to ML, it is necessary to understand the effect of quantification over type variables on the complexity of type inference. Naturally, this insight is also crucial in the case of F_2. The progress in understanding ML quantification and type inference is primarily due to two straightforward observations. The first, given by Mitchell (1990),† is that the following inference rule for `let` preserves exactly the type judgements for closed terms usually derived using the quantification rules:

$$(\texttt{let}) \qquad \frac{\Gamma \vdash M \in \tau_0 \quad \Gamma \vdash [M/x] N \in \tau_1}{\Gamma \vdash \texttt{let}\; x = M \;\texttt{in}\; N \in \tau_1}.$$

Because τ_0 and τ_1 are first-order types, this alternate inference rule is a classic instance of quantifier elimination. In the spirit of the Curry–Howard propositions-as-types analogy, it also acts as a sort of cut elimination, preserving propositional theorems at the expense of greatly enlarging the size of the proofs. A proof of this cut elimination theorem appears in the appendix of Kanellakis *et al.* (1991); a different and much cleaner proof inspired by the Tait (1967) strong normalisation theorem is found in Mairson (1992a). The added *combinatorial* insight comes from that fact that type inference can be completely reduced to first-order unification.

The second observation, due to Paris Kanellakis and John Mitchell, is that `let` can be used to compose functions an exponential number of times with a polynomial-length expression (Kanellakis and Mitchell, 1989):

Example 3.1

$$\begin{aligned}
\Psi \equiv \texttt{let}\; & x_0 = \delta \;\texttt{in} \\
& \texttt{let}\; x_1 = \lambda y . x_0 (x_0 y) \;\texttt{in} \\
& \quad \texttt{let}\; x_2 = \lambda y . x_1 (x_1 y) \;\texttt{in} \\
& \qquad \dots \\
& \qquad \quad \texttt{let}\; x_t = \lambda y . x_{t-1} (x_{t-1} y) \;\texttt{in}\; x_t
\end{aligned}$$

The above expression `let`-reduces‡ to $\lambda y . \delta^{2^t} y$, where the occurrences of δ are *polymorphic* – each occurrence has a different type.

† In this survey, the rule is attributed to Albert Meyer. However, it appears as well in the thesis of Luis Damas (1985), and in fact a question about it can be found in the 1985 postgraduate examination in computing at Edinburgh University (Sannella, 1988).

‡ Following the (*let*) rule given above, we say that *let* $x = M$ *in* N *let*-reduces in one step to $[M/x]N$.

The significance of this polymorphism is exploited in the lower bound of Mairson (1990) and Kanellakis *et al.* (1991), where it is shown that recognising typable ML expressions of length n can be solved in DTIME[2^n], and is DTIME[2^{n^k}]-hard for every integer $k \geqslant 1$ under logspace reduction.† The lower bound is a generic reduction: given a TM M accepting or rejecting its input x of length n in at most 2^{n^k} steps, we construct an ML term $\Phi_{M,x}$ using a logspace transducer, where M accepts x iff $\Phi_{M,x}$ is typable. We expand further on this proof technique in this section, extending its application to more powerful type systems.

The coding of the TM computation sketched above is embedded in the putative *typing* of $\Phi_{M,x}$, rather than in its *value*. The simulation of the TM is based on the observation that the transition function of M is merely a Boolean circuit computing state, head movement, and what to write on the tape, combined with rudimentary list processing to manipulate the left- and right-hand sides of the tape. The details of the proof show that these basic operations can be simulated by first-order unification problems which may be induced via the typing of `let`-free λ-terms. If δ is indeed the ML term simulating the transition function, and ID_0 codes the initial ID of the TM, then the type of $\Psi\, ID_0$ codes the ID of the TM after 2^t state transitions. Taking $t = n^k$, we can then construct an ML expression E containing $\Psi\, ID_0$ as a subterm, where the type of E necessarily codes whether M rejected x.

Using the methods of Dwork *et al.* (1984) for coding Boolean logic, the simulation codes the Boolean values *true* and *false* as:‡

$$true \equiv \lambda x.\lambda y.\lambda z.K z(Eq\, xy) \in \Delta a{:}*.\Delta b{:}*.a \to a \to b \to b$$

$$false \equiv \lambda x.\lambda y.\lambda z.z \in \Delta a{:}*.\Delta b{:}*.\Delta c{:}*.a \to b \to c \to c$$

where:

$$Eq \equiv \lambda x.\lambda y.K x(\lambda z.K(zx)(zy)) \in \Delta a{:}*.a \to a \to a$$

As a consequence, if the types of ML expressions P and Q cannot be unified, we know *true PQ* cannot be typed, while *false PQ* can be typed: *true* is a function insisting that its two arguments have the same type, while *false* makes no such restriction.

In its nascent state, the ML lower bound is useless to bound the complexity of F_2 type inference. The proof of 'machine accepts iff ML expression types' is made by a straightforward appeal to the simple logic of first-order unification: the proof construction computes a Boolean value coding the answer to 'Did M reject its input?' and uses the value, as described above, to force a first-order mistyping when the answer is *true*. To further claim that there is no F_2 typing is far from clear, since unlike ML, F_2 does not admit naive quantifier elimination, where an ML expression involving `let` is typable iff a similar, `let`-free expression is typable. Because of this equivalence, we see that arguments about typability based on first-order unification are simply too weak. Proving that strongly normalising terms are not F_2-typable is very difficult: as evidence, we point merely to the tremendous effort of Giannini and

† An alternate proof, based on the analysis of a problem called *acyclic seminunification*, is found in Kfoury *et al.* (1990).

‡ We use the syntax of F_2 types to give typings of ML terms, observing that ML types are merely outermostquantified *first-order* (i.e. quantifier-free) types. Such types are a proper subset of the F_2 types, where *any* subterm of a type may contain quantifiers.

Ronchi della Rocca (1988) in their identifying a single, simple, strongly normalising term which is not F_2-typable. The Giannini–Ronchi results are an indication that F_2 type inference might in fact be decidable, since they achieved a separation between F_2-typable terms and the r.e.-complete class of strongly normalisable terms. However, the techniques they employ require such overwhelming computational detail, and provide so little large-scale insight, that they seem virtually useless for showing whether or not the term we have constructed is F_2-typable.

3.2 Paradise regained: an F_2 lower bound

The force of the ML argument can be regained, however, by changing the simulation of Boolean logic from that found in Dwork *et al.* (1984) to the classic simulation in the λ-calculus. For example, we type the Boolean values as:

$$\mathit{true} \equiv \Lambda\alpha{:}*.\Lambda\beta{:}*.\lambda \mathbf{x}{:}\alpha.\lambda \mathbf{y}{:}\beta.\mathbf{x} \in \Delta\alpha{:}*.\Delta\beta{:}*.\alpha\to\beta\to\alpha$$
$$\mathit{false} \equiv \Lambda\alpha{:}*.\Lambda\beta{:}*.\lambda \mathbf{x}{:}\alpha.\lambda \mathbf{y}{:}\beta.\mathbf{y} \in \Delta\alpha{:}*.\Delta\beta{:}*.\alpha\to\beta\to\beta$$

We remark that this encoding is *not* Girard's inductive-type definition of Boolean values; observe simply that *true* and *false* have different types, while in Girard's construction, the Boolean values are both terms of type $\Delta\alpha{:}*.\alpha\to\alpha\to\alpha$.

By using this well-known coding of classical logic (see, for example, Hindley and Seldin, 1986), we discover a new class of directed acyclic graphs realising Boolean operations via first-order unification, in the style of Dwork *et al.* (1984). Moreover, the analysis of these graphs allows us to view types as explicit codings of certain reduction sequences in the λ-calculus. As a consequence, we derive a new and simplified proof that first-order unification is PTIME-complete, which is particularly striking since it shows how that theorem could have been proved by writing a very simple `let`-free ML program using the classic coding of Boolean logic. By generalising the realisation of Boolean logic to the realisation of *any* functions on finite domains, we derive a new and simpler proof of the DTIME[2^{n^k}]-hardness of recognising typable ML expressions.

Finally, by a slight augmentation of the ML proof, we derive a DTIME[2^{n^k}]-hardness bound on the recognition of F_2-typable terms. We make essential and powerful use of the strong normalisation theorem for F_2 (Girard, 1972; Girard *et al.*, 1989; Gallier, 1990), using the coded Boolean answer A to the question 'Did machine M reject its input of length n in 2^{n^k} steps?' to *choose* between a trivial terminating computation and a clearly nonterminating one:

$$\Psi \equiv (\lambda x.xx)(A(\lambda x.x)(\lambda y.yy))$$

Observe that if $A \rhd \mathit{false}$, then $\Psi \rhd (\lambda x.xx)(\lambda y.yy)$; since Ψ is not normalisable, it is not typable. The difficult technical question then becomes to show that if the TM accepts, then Ψ can be typed. In this case, we will have to look more carefully at the structure of the term A.

3.3 Encoding Boolean logic by terms and types

Using the definitions of *true* and *false* from the previous section, we can type the usual codings of Boolean functions as:

$$\begin{aligned}
\mathit{and} \equiv\ & \Lambda\alpha{:}*.\Lambda\beta{:}*.\Lambda\gamma{:}*.\Lambda\delta{:}*.\\
& \qquad \lambda \mathbf{p}{:}\alpha\to(\beta\to\gamma\to\gamma)\to\delta.\lambda \mathbf{q}{:}\alpha.\mathbf{p}\,\mathbf{q}(\mathit{false}[\beta][\gamma])\\
& \qquad\quad \in \Delta\alpha{:}*.\Delta\beta{:}*.\Delta\gamma{:}*.\Delta\delta{:}*.\\
& \qquad\qquad (\alpha\to(\beta\to\gamma\to\gamma)\to\delta)\to\alpha\to\delta\\
\mathit{or} \equiv\ & \Lambda\alpha{:}*.\Lambda\beta{:}*.\Lambda\gamma{:}*.\Lambda\delta{:}*.\\
& \qquad \lambda \mathbf{p}{:}(\alpha\to\beta\to\alpha)\to\gamma\to\delta.\lambda \mathbf{q}{:}\gamma.\mathbf{p}(\mathit{true}[\alpha][\beta])\,\mathbf{q}\\
& \qquad\quad \in \Delta\alpha{:}*.\Delta\beta{:}*.\Delta\gamma{:}*.\Delta\delta{:}*.\\
& \qquad\qquad ((\alpha\to\beta\to\alpha)\to\gamma\to\delta)\to\gamma\to\delta\\
\mathit{not} \equiv\ & \Lambda\alpha{:}*.\Lambda\beta{:}*.\Lambda\gamma{:}*.\Lambda\delta{:}*.\Lambda\varepsilon{:}*.\\
& \qquad \lambda \mathbf{p}{:}(\alpha\to\beta\to\beta)\to(\gamma\to\delta\to\gamma)\to\varepsilon.\\
& \qquad\quad \mathbf{p}(\mathit{false}[\alpha][\beta])(\mathit{true}[\gamma][\delta])\\
& \qquad\qquad \in \Delta\alpha{:}*.\Delta\beta{:}*.\Delta\gamma{:}*.\Delta\delta{:}*.\Delta\varepsilon{:}*.\\
& \qquad\qquad\quad ((\alpha\to\beta\to\beta)\to(\gamma\to\delta\to\gamma)\to\varepsilon)\to\varepsilon
\end{aligned}$$

Observe that all of these typings can be derived by the ML type inference algorithm. As in ML, all the types are outermost quantified, although we have written the typings in the notational style of F_2. The subterms *true* and *false* are explicitly parameterised to ensure that the terms are well-typed; in this manner, the typing rules of F_2 are used to simulate the unification mechanism of the ML type inference algorithm.

The computational significance of these typings is not particularly lucid as written. We can however clarify this significance by picturing the types as directed acyclic graphs (dags), having nodes labelled with appropriate subterms. For example, Fig. 1 shows the typing of *and* drawn as a graph, together with the dags for *true* and *false*. At the level of pure λ-terms, we know that the defined terms simulate Boolean logic, but Fig. 1 shows how the simulation is effected as well at the level of first-order unification and types. For instance, we know that *and true q* should reduce to q, and that *and false q* should reduce to *false*. To simulate the former reduction, we unify the dag rooted at p (the 'first input') with the dag representing the type of *true*, causing the node labelled q to be unified with the 'output node' labelled *pqfalse*, so that the *type* of input q is indeed the type of *and true q*. To simulate the reduction of *and false q*, observe that unification of 'input' p with the dag representing the type of *false* causes the substructure of the graph for *and* rooted at the node labelled *false* to be unified with the output node. Then the value and the type of the 'second input' q become irrelevant to the final output.

Similar arguments can be made that the definitions of *or* and *not* function properly at the level of reductions in the untyped λ-calculus, and that the types of these definitions simulate logic faithfully at the level of first-order unification. Figure 2 shows the relevant dags coding the types of *or* and *not*.

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

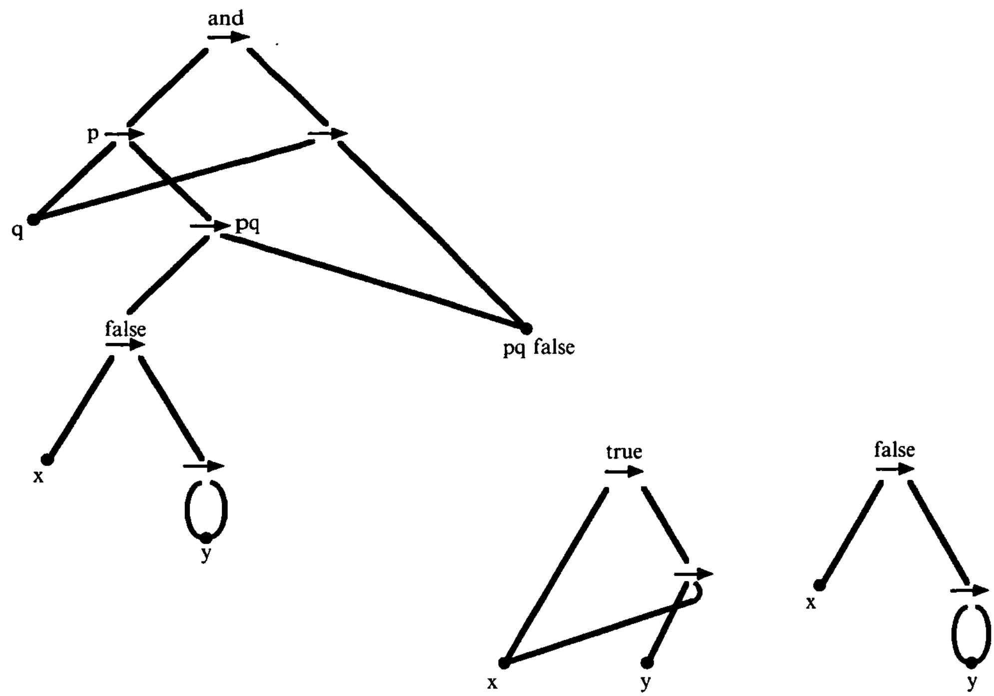

Fig. 1. Graph representations of *and*, *true*, *false*.

3.4 Unification is PTIME-complete

The graphs depicted in Figs. 1 and 2 have the same computational significance as the gadgets invented by Dwork, Kanellakis and Mitchell (1984) in their well-known proof that unification is complete for deterministic polynomial time. In this section, we show how their theorem could have been proved by writing an ML program using the classic λ-calculus encodings of the logical operations. The insight provided by this simpler problem is important in understanding the more detailed and sophisticated arguments we will see later on.

The proof of Dwork *et al.* (1984) was, essentially, that the *circuit value problem* (given a Boolean circuit with inputs, what is its output?) could be reduced to unification. The circuit value problem is logspace complete for polynomial time (Ladner, 1975) since any polynomial time TM computation can be described by a polynomial sized circuit, where the input to the circuit is the initial tape contents, and polynomial 'layers' of circuitry implement each state transition.

To carry out the simulation of a Boolean circuit in ML, we define (as above);†

```
- fun True x y= x;
val True = fn : 'a -> 'b -> 'a
- fun False x y= y;
val False=fn : 'a -> 'b -> 'b
```

† To avoid conflict with ML reserved words, examples using ML capitalise the names of declared functions.

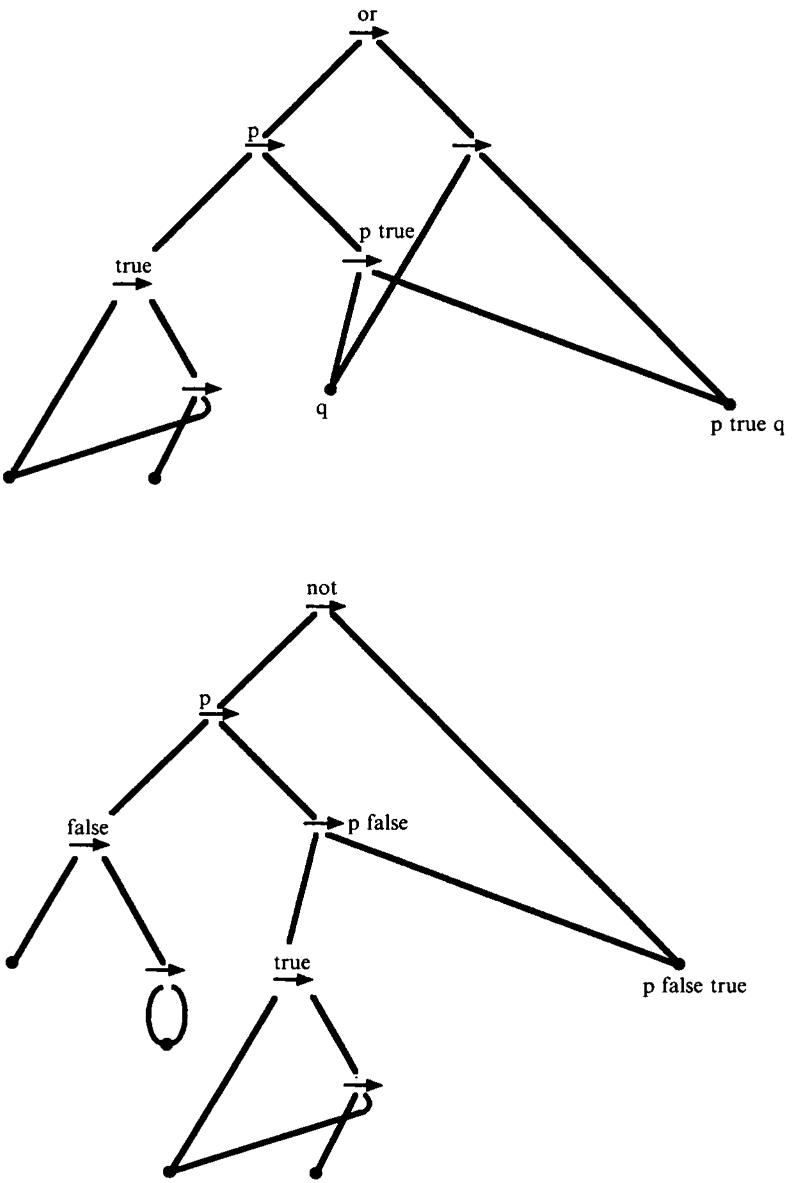

Fig. 2. Graph representations of *or*, *not*.

```
- fun And p q= p q False;
val And=fn : ('a -> ('b -> 'c -> 'c) -> 'd) -> 'a
   -> 'd
- fun Or p q=p True q;
val Or=fn : (('a -> 'b -> 'a) -> 'c -> 'd) -> 'c
   -> 'd
- fun Not p=p False True;
val Not=fn : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c)
   -> 'e -> 'e
```

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

When these Boolean functions are used, notice that they output functions as values; moreover, the principal types of these functions identify them uniquely as `True` or `False`:

```
- Or False True;
val True=fn : 'a -> 'b -> 'a
- And True False;
val False=fn : 'a -> 'b -> 'b
- Not True;
val False=fn : 'a -> 'b -> 'b
```

As a consequence, while the compiler does not *explicitly* reduce the above expressions to normal form, hence computing an 'answer', its type inference mechanism implicitly carries out that reduction to normal form, expressed in the language of first-order unification.

We now introduce pairing and fanout:

```
- fun Pair x y=fn z=> z x y;
val Pair=fn : 'a -> 'b -> ('a -> 'b -> 'c) -> 'c
- fun Fanout p=p (Pair True True) (Pair False False);
val Fanout=fn : (((('a -> 'b -> 'a) -> ('c -> 'd
   -> 'c) -> 'e) -> 'e)
 -> ((('f -> 'g -> 'g) -> ('h -> 'i -> 'i) -> 'j)
   -> 'j) -> 'k) -> 'k
```

The importance of `Fanout`, as in Mairson (1990) and Kanellakis *et al.* (1991), is that it produces two copies of a logic value which do not share type variables:

```
- Fanout True;
val it=fn : (('a -> 'b -> 'a) -> ('c -> 'd -> 'c)
   -> 'e) -> 'e
- Fanout False;
val it=fn : (('a -> 'b -> 'b) -> ('c -> 'd -> 'd)
   -> 'e) -> 'e
```

We code circuits so that every Boolean value is used *exactly once*. An intuitive correspondence to *linear logic* should be immediately apparent, in that the described simulation of logic breaks down if a Boolean value is used as an input to two different computations. For example, if we define `fun Break p=Or p (Not p)`, we find, rather peculiarly, that `Break True` has type `'a -> 'a -> 'a`, a type that is the most general unifier (least upper bound, in the lattice of unification) of the types of `True` and `False`. The function `break` uses input `p` in two different contexts, and each context imposes constraints on the (monomorphic) type. As a consequence, the output no longer uniquely codes a Boolean value.

To facilitate the understanding of our coding, we introduce some syntactic sugar for pattern matching, introducing the use of semicolon (;) to simulate a notion of sequentiality. We write $\langle v_1, \ldots, v_k \rangle = \psi;\ \phi$ for $\psi(\lambda v_1. \cdots \lambda v_k.\phi)$. For example, $\langle p, q \rangle = \mathit{fanout}\, r;\ \phi$ should be read as 'fanout Boolean value r, making two copies

p and q, and in that context return the value of ϕ'. We write $r = op\,p\,q;\ \phi$ for $(\lambda r.\phi)(op\,p\,q)$ and $r = op\,p;\ \phi$ for $(\lambda r.\phi)(op\,p)$ for binary and unary operators, respectively. The ; is meant to be right associative, so that $\phi_1; \phi_2; \phi_3$ means $\phi_1; (\phi_2; \phi_3)$.

A Boolean circuit can now be coded as a λ-term by labelling its (wire) edges and traversing them bottom-up, inserting logic gates and fanout gates appropriately. We consider the circuit example from (Dwork *et al.*, 1984, p. 43), pictured in Fig. 3; the circuit is realised by the code:

$$
\begin{aligned}
&\lambda e_1.\lambda e_2.\lambda e_3.\lambda e_4.\lambda e_5.\lambda e_6.\\
&\qquad e_7 = and\, e_2\, e_3;\\
&\qquad e_8 = and\, e_4\, e_5;\\
&\qquad \langle e_9, e_{10}\rangle = fanout(and\, e_7\, e_8);\\
&\qquad e_{11} = or\ e_1\, e_9;\\
&\qquad e_{12} = or\, e_{10}\, e_6;\\
&\qquad or\, e_{11}\, e_{12}
\end{aligned}
$$

Removing the syntactic sugar, this straight-line code 'compiles' to the slightly less comprehensible

```
- fun circuit e1 e2 e3 e4 e5 e6=
  (cp2 And) e2 e3 (fn e7=>
  (cp2 And) e4 e5 (fn e8=>
  (Fanout f (fn e9=>
  (cp2 Or) e1 e9 (fn e11=>
  Or e11 e12))))));
val circuit=fn : (('a -> 'b -> 'a) -> 'c -> ('d ->
 'e -> 'd) -> 'f -> 'g) -> ('h -> ('i -> 'j -> 'j)
 -> 'k -> ('l -> 'm -> 'm) -> ((('n -> 'o -> 'n)
 -> ('p -> 'q -> 'p) -> 'r) -> 'r) -> ((('s -> 't
 -> 't) -> ('u -> 'v -> 'v) -> 'w) -> 'w) -> ('c
 -> (('x -> 'y -> 'x) -> 'z -> 'f) -> 'g) -> 'ba)
 -> 'h -> ('bb -> ('bc -> 'bd -> 'bd) -> 'k)
 -> 'bb -> 'z -> 'ba
```

The type of `circuit` is the equivalent of the construction in Dwork *et al.* (1984) of the circuit as a unification structure. We can compute circuit values by instantiating the inputs appropriately, for instance:

```
- circuit False True True True True False;
val it = fn : 'a -> 'b -> 'a
```

Observe that this evaluation produces both the correct type, and the correct value: there is of course only one closed λ-term in normal form with the given type, namely $true \equiv \lambda x.\lambda y.x$. The computation of the value is performed by the interpreter, while that of the type is performed by the compiler, yet both are essentially the same. In essence, we have forced the compiler – more specifically, the type inference

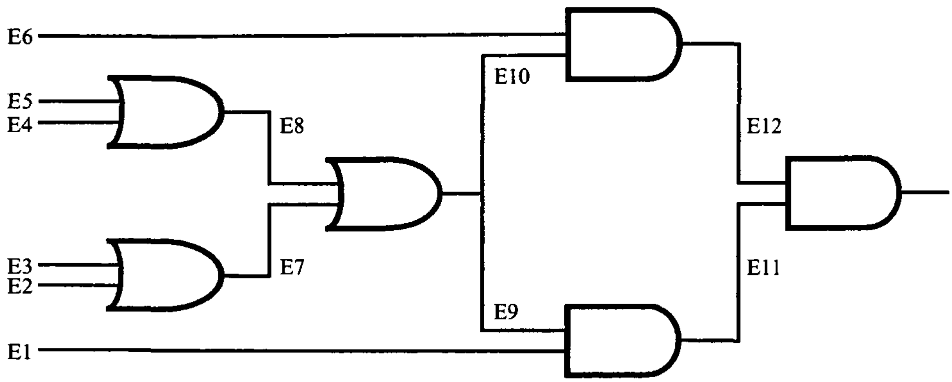

Fig. 3. Labelling of a Boolean circuit

mechanism – into doing computation typically carried out by the interpreter. It should be clear that any Boolean circuit can be transformed into such a λ-term. In mundane programming language terminology, the complexity theoretic *reduction* is merely a *compiler*. The size of the λ-term is clearly linear in the size of the circuit, and the transformation described can be effected in polynomial time. Observe that polynomial space is required by this translation scheme, since output wire names (for example, `e7`) are output by the transducer while their *values* (for example, `And e2 e3`) are pushed on a stack for subsequent output. The size of the stack can clearly be linear in the size of the ML program output by the transducer.

We can in fact carry out this sort of reduction in logarithmic space, curiously, by coding computation in a continuation-passing style. A hint towards carrying out such a reduction is given by the use of the `Fanout` gate in the above example, where `Fanout` takes input `And e7 e8`, and produces two outputs packaged together as a pair. The pair is then applied to the continuation `(fn e9=> fn e10=>...)`, so that the two (duplicate) truth values in the pair are bound to `e9` and `e10`. It is a simple matter to treat the single-output cases similarly, by coding a continuation-passing version of unary and binary logical functions:

```
- fun cp1 fnc p k=k (fnc p);
val cp1=fn : ('a -> 'b) -> 'a -> ('b -> 'c) -> 'c
- cp1 Not;
val it=fn :
(('a -> 'b -> 'b) -> ('c -> 'd -> 'c) -> 'e)
  -> ('e -> 'f) -> 'f
- fun cp2 fnc p q k=k (fnc p q);
val cp2=fn : ('a -> 'b -> 'c) -> 'a -> 'b ('c -> 'd)
  -> 'd
```

Now we use the continuation-passing version of the logic gates, in a style very much like straight-line code or machine language:

```
- fun circuit e1 e2 e3 e4 e5 e6=
  (cp2 And) e2 e3 (fn e7=>
```

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

```
    (cp2 And) e4 e5 (fn e8=>
    (cp2 And) e7 e8 (fn f=>
    Fanout f (fn e9=> fn e10=>
    (cp2 Or) e1 e9 (fn e11=>
    (cp2 Or) e10 e6 (fn e12=>
    Or e11 e12))))));
  val circuit=fn : (('a -> 'b -> 'a) -> 'c -> ('d
    -> 'e -> 'd) -> 'f
  -> 'g) -> ('h -> ('i -> 'j -> 'j) -> 'k -> ('l
    -> 'm -> 'm) -> ((('n
  -> 'o -> 'n) -> ('p -> 'q -> 'p) -> 'r) -> 'r)
    -> ((('s -> 't -> 't)
  -> ('u -> 'v -> 'v) -> 'w) -> 'w) -> ('c -> (('x
    -> 'y -> 'x) -> 'z ->
  'f) -> 'g) -> 'ba) -> 'h -> ('bb -> ('bc -> 'bd
    -> 'bd) -> 'k) -> 'bb
  -> 'z -> 'ba

  - circuit False True True True True False;
  val it=fn : 'a -> 'b -> 'a
```

The style of this coding is very similar to that used by Mitchell Wand (1992) in a framework for verifying compilers, where assembly code is generated in a version of λ-calculus; the idea also appears in Appel and Jim (1989) and Kelsey and Hudak (1989). A trivial analysis shows this translation scheme to be a logarithmic space reduction, since the transducer need only *count* right parentheses to be output at the end of the expression, instead of storing expressions on a stack. In this analysis, we have also assumed that the wire names have length logarithmic in the size of the circuit.

3.5 Coding functions with finite domains

The coding techniques used in the previous section make no particular use of the fact that the functions simulated are logical ones. On the contrary, the essential feature of the coding is that the Boolean functions are over finite domains and ranges. As a consequence, we may generalise the construction to any function with this characteristic.

Given a finite set $E^k = \{e_1^k, \ldots, e_k^k\}$, we code the ith element e_i^k by the λ-term:

$$d_i^k \equiv \lambda x_1 . \lambda x_2 . \cdots \lambda x_k . x_i$$

A *t-tuple* $e \equiv \langle e_1, \ldots, e_t \rangle$ is coded by the λ-term $\lambda z . z e_1 \cdots e_t$; note $e d_i^t \rhd e_i$, recalling $\rhd$ to be the reflexive, transitive closure of the basic reduction step denoted by $\rhd_\beta$. A function $m : E^k \to F$ with finite domain can then be coded as the tuple:

$$m = \langle m(e_1), \ldots, m(e_k) \rangle$$

so that $m d_i^k \rhd m(e_i)$. When the finite domain of a function is the product of several finite sets, we realise the function in its curried form.

In this manner, we can code the Boolean functions in tabular form: recalling $true \equiv \lambda x.\lambda y.x$ and $false \equiv \lambda x.\lambda y.y$, we have:

$$\begin{aligned} not &\equiv \langle false, true\rangle \\ and &\equiv \langle\langle true, false\rangle, \langle false, false\rangle\rangle \\ or &\equiv \langle\langle true, true\rangle, \langle true, false\rangle\rangle \end{aligned}$$

Again, the codings work simultaneously at the value level and at the type level; they are, moreover, ML-typable. Observe that the 'currying' trick of nested tuples is used for the binary operations.

3.6 Encoding Turing Machines by lambda terms

Given a deterministic TM M, we show how to encode the transition function of M as a λ-term δ such that if ID is a λ-term encoding a configuration of M, and ID' is the next configuration of M coded as a λ-term, then $\delta\, ID \rhd ID'$. We call this a simulation *at the value level*. In section 3.7, we will see that the type of δ also encodes a simulation at the type level. The encoding, which uses the methods of the previous section, also yields a very compact and simple proof of the DTIME[2^{n^k}]-hardness bound for recognising ML-typable terms. To reduce notational clutter, we blur the name distinctions between parts of the TM and their respective codings in the λ-calculus; for example, we write q_i for a TM state as well as its coding as a λ-term.

Since the TM manipulates a tape (represented as two lists, to the left and right of the read head), we need to code lists and relevant operations on them, while preserving the symmetry of values and types. Therefore, a *list* $[x_1, \ldots, x_k]$ denotes the tuple $\langle x_1, \langle x_2, \ldots, \langle x_k, \mathsf{nil}\rangle \cdots \rangle\rangle$, where $\mathsf{nil} \equiv \lambda z.z$.

Let M have finite states $Q = \{q_1, \ldots, q_k\}$ with initial state q_1 and final states $F \subset Q$; tape alphabet $C = \{c_1, \ldots, c_\ell\}$ with blank symbol $\not{b} \equiv c_1$; tape head movements $D = \{d_1, d_2, d_3\}$ (left, no movement, right); and transition table $\partial: Q \to C \to Q \times C \times D$.† A *configuration* (ID) of M is a triple $\langle q, L, R\rangle \in Q \times C^* \times C^*$ giving the state and contents of the left- and right-hand sides of the tape; we thus define the transition function of M by the usual extension of ∂. We assume that the TM never writes a blank, that it does not move its tape head iff it reads a blank, and that it never runs off the left end of the tape.

Using techniques of the previous section, we code each state q_i as the projection function $\lambda x_1.\lambda x_2.\cdots\lambda x_k.x_i$, each tape symbol c_i as $\lambda x_1.\lambda x_2.\cdots\lambda x_\ell.x_i$, and head movement d_i as $\lambda x_1.\lambda x_2.\lambda x_3.x_i$. If $\partial q_i c_j = t_{i,j}$ for some $t_{i,j} = \langle q, c, d\rangle \in Q \times C \times D$, then the map can be coded by the λ-term:

$$\partial \equiv \langle\langle t_{1,1}, t_{1,2}, \ldots, t_{1,\ell}\rangle, \langle t_{2,1}, t_{2,2}, \ldots, t_{2,\ell}\rangle, \cdots, \langle t_{k,1}, t_{k,2}, \ldots, t_{k,\ell}\rangle\rangle$$

Observe that $\partial q_i c_j \rhd t_{i,j}$. The term ∂ is just a table; q_i projects out the ith row, and then c_j projects out the jth column of that row.

We represent a TM ID by a tuple $\langle q, \mathbf{L}, \mathbf{R}\rangle$, where $\mathbf{L} \equiv [\ell_1, \ldots, \ell_m]$ and $\mathbf{R} \equiv [r_1, \ldots, r_n]$ are lists coding the left and right contents of the tape; we assume the tape head is reading r_1, and ℓ_1 is the cell contents to the immediate left.

† Note that we choose to represent ∂ in its curried form, rather than of type $\partial: Q \times C \to Q \times C \times D$.

Recalling the pattern-matching notation we have introduced in section 3.4, the transition function of M has a very simple encoding:

$$\begin{aligned}\delta \equiv \lambda ID.\, & \langle q, \mathbf{L}, \mathbf{R}\rangle = ID;\\ & \langle \ell, L\rangle = \mathbf{L};\\ & \langle r, R\rangle = \mathbf{R};\\ & \langle q', c', d'\rangle = \partial qr;\\ & d'\langle q', L, \langle \ell, \langle c', R\rangle\rangle\rangle\\ & \quad \langle q', \langle \ell, L\rangle, \langle c', \langle \not{b}, \mathsf{nil}\rangle\rangle\rangle\\ & \quad \langle q', \langle c', \langle \ell, L\rangle\rangle, R\rangle\end{aligned}$$

Notice that when the read head does not move, the ID $\langle q', \langle \ell, L\rangle, \langle c', \langle \not{b}, \mathsf{nil}\rangle\rangle\rangle$ is chosen; in this case, we know by the definition of the TM that a blank has been read, and the read head has therefore reached its *rightmost* position, where there should be an infinite sequence of blank cells to the right. This infinite sequence is not coded explicitly, but rather simulated implicitly by constructing a new right-hand side of the tape, namely $\langle c', \langle \not{b}, \mathsf{nil}\rangle\rangle$; we have 'tacked on' another blank cell. Should the TM move again to the right, the process will be repeated. We note that even though the TM can write several different values in a tape cell over time, the simulation of this behaviour manufactures a *new* coding of the cell each time the tap position is traversed, and uses list processing to place the cell in the correct position on the tape. In this way, no representation of a cell is every used more than once.

We also observe that $\langle q', c', d'\rangle$ codes the state, symbol written, and head direction for the next machine configuration, as computed by ∂; the term d' is then used *as a projection function* to choose the λ-term coding the next configuration. Because no value is 'used' more than once, no side-effecting of type variables occurs, and the fanout gates mentioned earlier, used in the proofs in Mairson (1990) and Kanellakis *et al.* (1991), are not necessary.

3.7 Encoding Turing Machines by types

The simple encoding δ of the transition function is typable in ML; moreover, it has the following property:

Proposition 3.2

Let ID and ID′ be λ-terms coding successive configurations of M, and let σ and σ′ be their respective first-order principal types. Then $\delta\, ID \rhd ID'$, *and* $\delta\, ID \in \sigma'$. *Furthermore, if* ID_k *is a λ-term with principal type* σ^k *coding the state of M after k transitions from ID, then* $(\bar{k}\delta)\, ID \rhd ID_k$, *and* $(\bar{k}\delta)\, ID \in \sigma^k$.

Proof

Before proceeding with details of the proof, the statement of this important Proposition deserves further explanation. It claims that the computation of TM M is simulated not only at the *value* level, but also at the *type* level. The first part of the Proposition, dealing with successive machine configurations, asserts this dual representation purely in the type system of ML, and by extension, in F_2.

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

The commutative diagram shown in Fig. 4 summarises this duality.

In typing $\delta\, ID$, first-order unification is performed on the principal types of δ and ID, producing a most general typing of $\delta\, ID$. The unification is merely a complicated variant of the calculation seen in sections 3.3 and 3.4, where unification simulated Boolean calculations; in the case of the Proposition, unification simulates calculations of finite functions specified by the transition map of the TM.

The second part of the Proposition, concerning the coding of multiple transitions of the TM, makes a similar assertion, but *not* purely in the type system of ML – essential use is made of the added expressiblity of F_2 types. Given that $\delta\, ID$ can indeed be typed, how can the typing technology be extended to type, say, $\delta(\delta(\delta(\delta\, ID)))$?

The answer is simple: each instance of δ is typed *differently*. Each such typing is indeed an instance of the principal type of δ, but changes according to the type of its argument. The key idea implicit in this construction is the composition of functions having different domains and ranges.

We are certainly familiar with composing functions of type Int –> Int, for example, while a function of type Int –> Bool cannot be so composed with itself. However, in the case of polymorphic functions where the range can be *parameterised* to have the same structure as the domain, we may in fact carry out such function composition. Rather than examining the fairly complicated type of δ, we consider as a motivating example the polymorphic composition of the Boolean function *not*. We begin by carrying out the composition in ML:

```
- val Not=Pair False True;
val Not=fn : (('a -> 'b -> 'b) -> ('c -> 'd -> 'c)
   -> 'e) -> 'e
- Not True;
val it=fn : 'a -> 'b -> 'b
- Not False;
val it=fn : 'a -> 'b -> 'a
- fun Notnot p=Not (Not p);
val Notnot=fn : (('a -> 'b -> 'b) -> ('c -> 'd
   -> 'c) ->
('e -> 'f -> 'f) -> ('g -> 'h -> 'g) -> 'i) -> 'i

- Notnot True;
val it=fn : 'a -> 'b -> 'a
- Notnot False;
val it=fn : 'a -> 'b -> 'b
```

In F_2, `Notnot` can be defined as:

$$\begin{aligned}
\mathit{notnot} = {} & \Lambda\alpha{:}*.\Lambda\beta{:}*.\Lambda\gamma{:}*.\Lambda\delta{:}*.\Lambda\varepsilon{:}*.\Lambda\phi{:}*.\Lambda\xi{:}*.\Lambda\eta{:}*.\Lambda\iota{:}*. \\
& \qquad \lambda \mathbf{p}{:}((\alpha\to\beta\to\beta)\to(\gamma\to\delta\to\gamma)\to(\varepsilon\to\phi\to\phi)\to(\xi\to\eta\to\xi))\to\iota \\
& \qquad\quad \mathit{not}[\varepsilon][\phi][\xi][\eta][\iota] \\
& \qquad\qquad (\mathit{not}[\alpha][\beta][\gamma][\delta][(\varepsilon\to\phi\to\phi)\to(\xi\to\eta\to\xi)\to\iota]\,\mathbf{p})
\end{aligned}$$

$$\begin{tikzcd} \mathrm{ID} \arrow[r, "\delta"] \arrow[d, dashed] & \mathrm{ID}' \arrow[d, dashed] \\ \sigma \arrow[r, "\text{unify } \delta"] & \sigma' \end{tikzcd}$$

Fig. 4. Duality of ID representation via types and terms.

In Fig. 2, the type of *not* is depicted as a dag. Even though we think of *not* as a function on the 'type' of Booleans, we have already noted that our codings of *true* and *false* are not of Boolean type in the standard inductive sense. Furthermore, the right-hand side of the dag, a single-node, clearly has less structure than the left-hand side. In fact, we can instantiate the right-hand side to look like the left-hand side, as shown in Fig. 5, to construct a type for *notnot*. The F_2 code for *notnot* contains type information that syntactically reproduces the information in the graph of Fig. 5. Notice that the outermost *not* in the F_2 code corresponds to the *deeper* graph for *not* in the figure.

Observe what happens when the dag rooted at p is unified with the dag for *true*: the leftmost *false* in Fig. 5 is forced to unify with the dag rooted at p' – in other words, *false* is 'input' into the rightmost *not* gate. Continuing the unification chain reaction, the rightmost dag for *true* is then forced to unify with the 'output' node p''. Hence application of *notnot* to *true* yields an answer of the type of *true*.

When we replace *not* with δ, and compose the coding δ of the transition function of a TM with itself, the details of the unification become more complicated, but the high-level structure of this argument remains unchanged. As in the case of `Notnot`, the left-hand side of the dag coding the type of δ has considerable structure, while the right-hand side is simply an external node (see Fig. 6). Assume that ϕ is the type of δ; we unify the dag $\mathscr{C}_1$ (the so-called 'TM circuitry') with a dag coding a TM ID. A dag coding the next ID of the TM is then forced to unify with node ID', and subsequent unification with the next copy $\mathscr{C}_2$ of TM circuitry simulates another transition, unifying ID'' with the following TM instantaneous description.

The essential property allowing polymorphic functions to be so composed is that the right-hand side of the dag encoding the type can be parameterised (i.e. instantiated by grafting of appropriate dags to the external nodes) to be identical to the left-hand side. When the right-hand side is simply a node (as in the case of the type of unary Boolean functions, and the type of δ as well), this property is obvious. Observe that the type ϕ^2 of $\delta \circ \delta$ then has the same property, and so it too can be composed with itself. In section 5, we will formalise this general property in the type language of F_ω to derive a nonelementary bound on type inference.

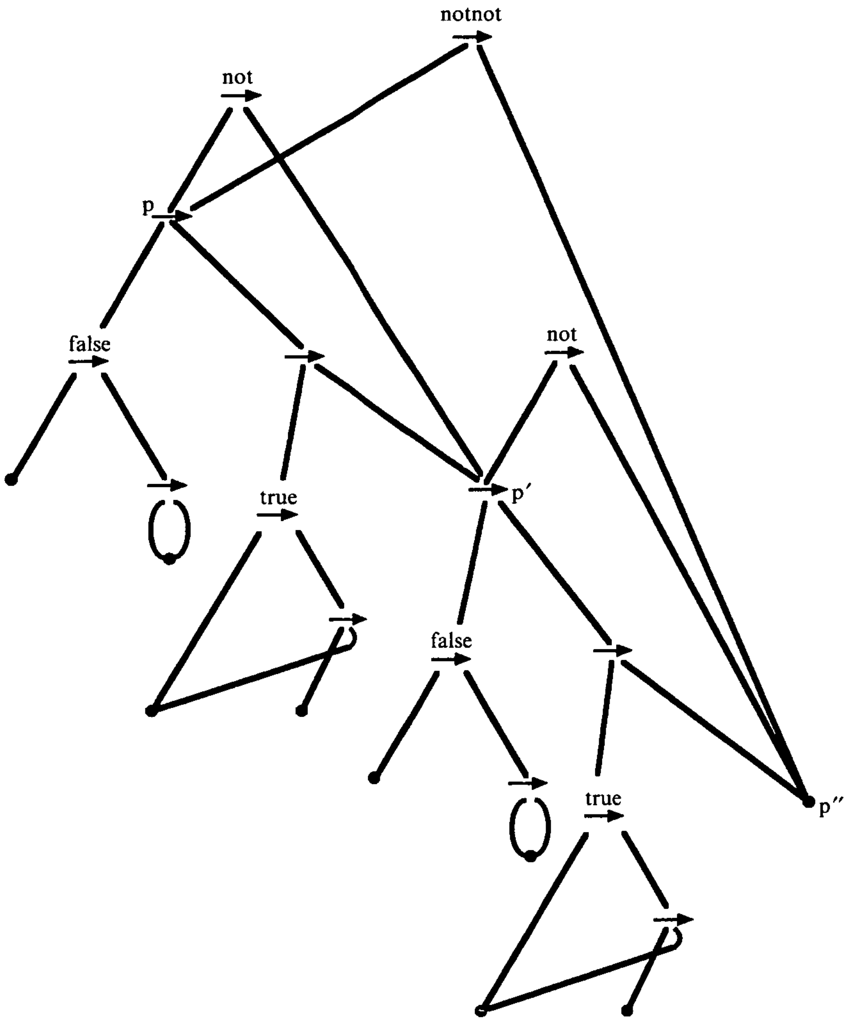

Fig. 5. Graph representation of the type of Notnot.

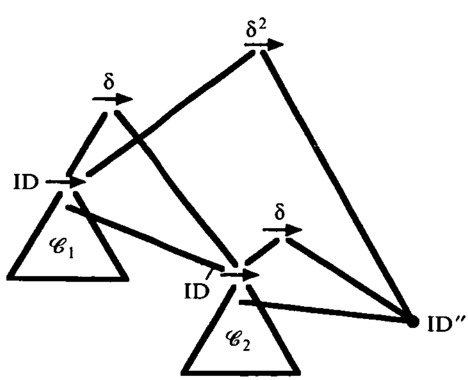

Fig. 6. Graph representation of the composition of δ.

Given these intuitions about composing functions, the proof of the first part of the Proposition is straightforward: it follows from principles of first-order unification. We can, however, elaborate further on the second part.

Define λ-term $\bar{k} \equiv \lambda s.\lambda z.s^k z$ and type $\delta \in \Phi \equiv \Delta v_1{:}*.\Delta v_2{:}*, \ldots .\Delta v_r{:}*. \mathscr{L}(v_1, v_2, \ldots, v_r) \rightarrow \mathscr{R}(v_1, v_2, \ldots, v_r)$, where $\mathscr{L}(v_1, v_2, \ldots, v_r)$ and $\mathscr{R}(v_1, v_2, \ldots, v_r)$ are *metanotation* representing quantifier-free (i.e. first-order) types over type variables $v_1, v_2, \ldots, v_r$. Given these definitions, we type $(\bar{k}\,\delta)$ *ID* so that $\bar{k}$ has type

$$\lambda \mathbf{s}{:}\Phi.\lambda \mathbf{z}{:}\sigma.\mathbf{s}[\pi_{1,1}] \cdots [\pi_{1,r}](\mathbf{s}[\pi_{2,1}] \cdots [\pi_{2,r}] \cdots (\mathbf{s}[\pi_{k,1}] \cdots [\pi_{k,r}]\mathbf{z}) \cdots)$$

where the $\pi_{i,j}$ are quantifier-free parameterisations of Φ such that if $\mathbf{s}[\pi_{i,1}] \cdots [\pi_{i,r}]$ has type $\alpha \rightarrow \beta$, then α unifies with σ^{k-i}, and β is equal to σ^{k-i+1} (up to renaming of type variables). We can then assign to z a type *unifying with* $\sigma \equiv \sigma^0$, and assign term $s^k z$ the type σ^k. □

The typing of $\bar{k}\,\delta$ in the above Proposition uses an essential feature of F_2: observe that the type of $\bar{k}$ is not outermost-quantified, since the type of $\mathbf{s}$ is the (polymorphic) type of δ, containing quantifiers.

Corollary 3.3
Let $\bar{k}$ denote the Church numeral for $\lambda s.\lambda z.s^k z$, and let:

$$E \equiv (\lambda f.\bar{2}(\bar{2} \cdots (\bar{2}f) \cdots))\,\delta\, ID_0$$

where there are m occurrences of $\bar{2}$, and ID_0 codes an initial ID of M. Then E has the same normal form and rank 2 type as $(\overline{2^m}\,\delta)\,ID_0$.

Proof
We write $\Lambda\vec{v}_t{:}*$ (resp. $\Delta\vec{v}_t{:}*$) to denote abstraction over a sequence $v_1, \ldots, v_t$ of type variables, and $[\vec{\pi}]$ to denote a sequence of parameterisations. We then write the type pictured in Fig. 6 as $\Delta\vec{v}_t v_{\mathrm{out}}{:}*.\mathscr{L}(\vec{v}_t v_{\mathrm{out}}) \rightarrow v_{\mathrm{out}}$, where $\mathscr{L}$ is a *type functional* mapping $t+1$ types to a type.† Here, as in Proposition 3.2, $\mathscr{L}$ is a *metanotation*, since it clearly is not part of the syntax of the type language of F_2; we see in section 5 how constructs similar to $\mathscr{L}$ can be formalised in F_ω. Using this notation, the rightmost $\bar{2}$ in E can be given the type:

$$\tau_1 \equiv (\Delta\vec{v}_t v_{\mathrm{out}}{:}*.\mathscr{L}(\vec{v}_t v_{\mathrm{out}}) \rightarrow v_{\mathrm{out}}) \rightarrow$$
$$\Delta\vec{v}_t \vec{v}'_t v'_{\mathrm{out}}{:}*.\mathscr{L}(\vec{v}_t \mathscr{L}(\vec{v}'_t v'_{\mathrm{out}})) \rightarrow v'_{\mathrm{out}},$$

that is:

$$\begin{aligned}&\lambda g{:}\Delta\vec{v}_t v_{\mathrm{out}}{:}*.\mathscr{L}(\vec{v}_t v_{\mathrm{out}}) \rightarrow v_{\mathrm{out}}.\\ &\quad\Lambda\vec{v}_t{:}*.\Lambda\vec{v}'_t{:}*.\Lambda v'_{\mathrm{out}}{:}*.\\ &\qquad\lambda \mathbf{y}{:}\mathscr{L}(\vec{v}_t \mathscr{L}(\vec{v}'_t v'_{\mathrm{out}}))\\ &\qquad\quad \mathbf{g}[\vec{v}'_t]\,[v'_{\mathrm{out}}]\\ &\qquad\qquad(\mathbf{g}[\vec{v}_t]\,[\mathscr{L}(\vec{v}'_t v'_{\mathrm{out}})]\,\mathbf{y})\end{aligned}$$

† Observe the informal use of concatenation of type variables, e.g. $\Delta\vec{v}_t v_{\mathrm{out}}{:}*$ denotes the type $\Delta v_1{:}*.\Delta v_2{:}*. \cdots \Delta v_t{:}*.\Delta v_{\mathrm{out}}{:}*$.

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

Iterating this construction, we can type the second rightmost $\bar{2}$ as:

$$\begin{aligned}
&\lambda \mathbf{g}:\Delta\vec{v}_t\,\vec{v}'_t\,v'_{\mathrm{out}}:*.\,\mathscr{L}(\vec{v}_t\,\mathscr{L}(\vec{v}'_t\,v'_{\mathrm{out}}))\rightarrow v'_{\mathrm{out}}.\\
&\quad\Lambda\vec{v}_t:*.\,\Lambda\vec{v}'_t:*.\,\Lambda\vec{v}''_t:*.\,\Lambda\vec{v}'''_t:*.\,\Lambda v'''_{\mathrm{out}}:*.\\
&\qquad\lambda \mathbf{y}:\mathscr{L}(\vec{v}_t\,\mathscr{L}(\vec{v}'_t\,\mathscr{L}(\vec{v}''_t\,\mathscr{L}(\vec{v}'''_t\,v'''_{\mathrm{out}}))))\\
&\qquad\quad \mathbf{g}[\vec{v}''_t][\vec{v}'''_t][v'''_{\mathrm{out}}]\\
&\qquad\qquad(\mathbf{g}[\vec{v}_t][\vec{v}'_t][\mathscr{L}(\vec{v}''_t\,\mathscr{L}(\vec{v}'''_t\,v'''_{\mathrm{out}}))]\,\mathbf{y})
\end{aligned}$$

with type:

$$\begin{aligned}
\tau_2 \equiv\ &(\Delta\vec{v}_t\,\vec{v}'_t\,v'_{\mathrm{out}}:*.\,\mathscr{L}(\vec{v}_t\,\mathscr{L}(\vec{v}'_t\,v'_{\mathrm{out}}))\rightarrow v'_{\mathrm{out}})\rightarrow\\
&\quad\Delta\vec{v}_t\,\vec{v}'_t\,\vec{v}''_t\,\vec{v}'''_t\,v'''_{\mathrm{out}}:*.\\
&\qquad\mathscr{L}(\vec{v}_t\,\mathscr{L}(\vec{v}'_t\,\mathscr{L}(\vec{v}''_t\,\mathscr{L}(\vec{v}'''_t\,v'''_{\mathrm{out}}))))\rightarrow v'''_{\mathrm{out}}
\end{aligned}$$

We continue the iteration to type each occurrence of $\bar{2}$, deriving a typing τ of $\bar{2}(\bar{2}(\cdots(\bar{2}f)\cdots))$, and hence a typing of $(\lambda f.\bar{2}(\bar{2}(\cdots(\bar{2}f))))\,\delta$. We then type E by first-order unifying the type of ID_0 with the left-hand side of τ. This unification is indeed possible, using an induction on m; we know the type of ID_0 unifies with $\phi \equiv \Delta\vec{v}_t\,v_{\mathrm{out}}:*.\,\mathscr{L}(\vec{v}_t\,v_{\mathrm{out}})\rightarrow v_{\mathrm{out}}$ as a basis, and the left-hand side of τ extends ϕ at the output variable. We then use the inductive step to follow through the 'chain reaction' of subsequent unifications with separate copies of TM circuitry. Since E reduces to $(\overline{2^m}\,\delta)\,ID_0$, the result follows from the so-called *subject reduction theorem* (see, for example, Hindley and Seldin, 1965), the λ-calculus interpretation of cut elimination: namely, if a term has a particular type, then any reduct of that term has the same type. □

We note that *rank 2 typing* refers to the fact that rank 2 is necessary for the type derivation, although the actual type of E is rank 1.

Theorem 3.4

Recognising the typable Core ML terms typable in F_2 is $DTIME[2^{n^t}]$-hard for any integer $t \geqslant 1$ under logspace reduction.

Proof

For a description of Core ML – essentially, the first-order typed λ-calculus with a polymorphic `let` such that `let` $x = E$ `in` B is syntactic sugar for $[E/x]\,B$ – see Harper *et al.* (1990), Kanellakis *et al.* (1991) and Mairson (1992a). Assume that $F = \{q_{p+1}\dots, q_k\} \subset Q$ are the accepting states of M. We define a combinator:

$$Eq \equiv \lambda x.\lambda y.K\,x(\lambda z.K(zx)(zy)) \in \Delta a:*.\,a\rightarrow a\rightarrow a$$

and consider the ML expression:

$$\begin{aligned}
\Psi \equiv\ &\mathtt{let}\;\delta_0 = \delta\;\mathtt{in}\\
&\quad\mathtt{let}\;\delta_1 = \lambda y.\delta_0(\delta_0\,y)\;\mathtt{in}\\
&\qquad\mathtt{let}\;\delta_2 = \lambda y.\delta_1(\delta_1\,y)\;\mathtt{in}\\
&\qquad\quad\dots\\
&\qquad\qquad\mathtt{let}\;\delta_{n^t} = \lambda y.\delta_{n^t-1}(\delta_{n^t-1}\,y)\;\mathtt{in}\\
&\qquad\qquad\quad\delta_{n^t}\,ID_0\\
&\qquad\qquad\qquad(\lambda \mathit{state}.\lambda\ell.\lambda r.Eq\,I((\mathit{state\,false}\dots\mathit{false\,true}\dots\mathit{true})\,I\,K))
\end{aligned}$$

where the first p arguments of *state* are *false*, and the remaining $k-p$ arguments are *true*. If TM M accepts its input, then $R \equiv$ *state false...false true...true* reduces to *true*, so that $RIK \rhd I$, and $Eq\,I\,I$ can be typed. If TM M rejects its input, then $RIK \rhd K$, and $Eq\,I\,K$ cannot be typed, since the types of I and K are not first-order unifiable, and the type constraints of Eq force their type equality. □

The construction in the above Theorem depends upon two basic components: a short coding of a long reduction sequence, and a gadget (based on first-order unification) to force a mistyping in the case of a rejecting computation. To derive a similar lower bound for F_2, we replace the coding of the reduction sequence by the construction of Corollary 3.3, and the mistyping gadget by one based on the strong normalisation theorem for F_2.

Theorem 3.5
Recognising the lambda terms typable in F_2 is $DTIME[2^{n^t}]$-hard for any integer $t \geqslant 1$ under logspace reduction.

Proof
Again, assume that $F = \{q_{p+1}, \ldots, q_k\} \subset Q$ are the accepting states of M. Consider:

$$A \equiv \langle q, L, R\rangle = (\lambda f.\bar{2}(\bar{2}\cdots(\bar{2}f)\cdots))\,\delta\, ID_0;$$
$$q\,\mathit{false}\ldots\mathit{false}\;\mathit{true}\ldots\mathit{true}$$

where $\bar{2}$ occurs n^t times, the first p arguments of q are *false* and the remaining $k-p$ arguments are *true*. If M accepts input x after exactly 2^{n^t} steps, then A β-reduces to *true*. By Lemma 3.2 and Corollary 3.3, the λ-expression A can be given type $\Delta\alpha{:}*.\Delta\beta{:}*.\alpha\to\beta\to\alpha$, so that $\Psi_{M,x} \equiv (\lambda x.xx)(A(\lambda x.x)(\lambda y.yy)$ is typable in rank 3:

$$(\lambda \mathbf{x}{:}\Delta\tau{:}*.\tau\to\tau.\mathbf{x}[\Delta\tau{:}*.\tau\to\tau]\mathbf{x})$$
$$(A[\Delta\tau{:}*.\tau\to\tau][(\Delta\tau{:}*.\tau \;{-}{>}\; \tau)\to(\Delta\tau{:}*.\tau\to\tau)](\Lambda\tau{:}*.\lambda\mathbf{x}{:}\tau.\mathbf{x})$$
$$(\lambda\mathbf{y}{:}\Delta\tau{:}*.\tau\to\tau.\mathbf{y}[\Delta\tau{:}*.\tau\to\tau]\mathbf{y}))$$

If M rejects x, then A β-reduces to $\lambda x.\lambda y.y$, and consequently $\Psi_{M,x}$ reduces to $(\lambda x.xx)(\lambda y.yy)$. By Girard's strong normalisation theorem (Girard, 1972; Girard *et al.*, 1989), $\Psi_{M,x}$ is not F_2-typable. It is easily seen that $\Psi_{M,x}$ can be constructed in logarithmic space from M and x, since the transducer need only count how many copies of the term $\bar{2}$ to output in the construction of A. □

Corollary 3.5 (Fixed type inference)
Let τ be an arbitrary but fixed F_2 type that is inhabited, *so that some lambda term exists with type τ. Then the problem of recognising the lambda terms which can be given the F_2 type τ is also $DTIME[2^{n^t}]$-hard for any integer $t \geqslant 1$ under logspace reduction.*

4 An overview of $F_3, F_4, \ldots, F_\omega$

The F_2 lower bound given above has two parts: (1) a simulation of the transition function of an arbitrary TM by a closed λ-term; and (2) a method for composing the transition function an exponential number of times. The analogous ML bound stops at exponential because of MLs limited ability to (polymorphically) compose arbitrary functions. No such limit is apparent in F_2 or its higher-order extensions, so a natural place to strengthen the F_2-bound is to improve the function composition realised in (2) and thus 'turn the "crank" (of the transition function) faster'. Note that the 'crank' of Example 3.1 is (without syntactic sugar) merely the λ-term:

$$(\lambda x_0 . (\lambda x_1 . \cdots (\lambda x_{t-1} . (\lambda x_t . x_t) (\lambda y . x_{t-1}(x_{t-1} y))) \cdots (\lambda y . x_1(x_1 y))) (\lambda y . x_0(x_0 y))) \delta$$

which has the same power as the term E in corollary 3.3. Might there be more powerful typable reduction sequences in the systems F_k?

We show that program can be carried out in F_ω to derive a nonelementary lower bound. Related superexponential bounds can be proven for the F_k so that recognising typable λ-terms of length n requires $f_k(n)$ time, where $f_k(n)$ is an 'exponential' stack of 2s growing linearly in k, with n on top of the stack. Before describing these lower bounds in more detail, we provide a brief overview of the type systems $F_3, F_4, \ldots, F_\omega$.

4.1 Kinds and abstraction over functions on types

In the first-order typed λ-calculus, the type language is made up of type variables, and a binary function $\rightarrow$ mapping a pair of types to a type. In F_2, we add universal quantification, but only over type variables. The higher-order systems $F_3, F_4, \ldots, F_\omega$ are designed to allow abstraction and quantification as well over *functions* on types, with varying degrees of freedom.

We introduce the notion of *kinds* to categorize types, similar to our use of types to categorise terms. For instance, we use $*$ (sometimes pronounced 'prop', as in *logical* prop*osition*) to denote the types found in F_2. (Not coincidentally, these types all have the essential syntax of logical propositions.) We describe the functionality of the (curried) function-space constructor $\rightarrow$ as $\rightarrow \in * \Rightarrow * \Rightarrow *$, where $\Rightarrow$ is a version of $\rightarrow$ at the *kind* level; the significance of this description is that, given two types τ_1 and τ_2 of kind $*$, the expression $\rightarrow \tau_1 \tau_2$ (usually written as the infix $\tau_1 \rightarrow \tau_2$) is also of kind $*$. Interpreted logically, this merely asserts that if τ_1 and τ_2 are logical propositions, so is $\tau_1 \rightarrow \tau_2$. Though $\rightarrow$ is not an F_2 *type*, we see clearly that it is a *type constructor*.

Following these intuitions, if we introduce λ-abstraction at the *type* level, imitating its existence at the expression level, we can describe other functions on types, and abstract over such functions. For example, given an arbitrary type A, the type $\Delta P{:}*.(A \rightarrow P \rightarrow P) \rightarrow P \rightarrow P$ can be used to code the type of *lists* of elements of type A, where we code the list $[x_1, x_2, \ldots, x_k]$ as the term:

$$\Lambda P{:}*.\lambda \mathbf{c}{:}A \rightarrow P \rightarrow P.\lambda \mathbf{n}{:}P.\mathbf{c}x_1(\mathbf{c}\,x_2(\cdots(\mathbf{c}\,x_k\,\mathbf{n})))$$

Note that with type information removed, this term is simply $\lambda c.\lambda n.cx_1 (cx_2(\cdots(cx_k n)))$, virtually identical to the familiar $\mathbf{cons}\,x_1(\mathbf{cons}\,x_2(\cdots(\mathbf{cons}\,x_k\,\mathbf{nil})))$, except that we have abstracted over the constructors **cons** and **nil**.

By abstracting over the arbitrary type A, we might define:

$$\mathsf{List} \equiv \lambda A{:}*.\Delta P{:}*.(A \rightarrow P \rightarrow P) \rightarrow P \rightarrow P$$

so that, for instance, we could write $Id[\mathsf{List\ Int}]$ to parameterise the identity function with the type of lists of integers.

In this case, List becomes a higher-order type of *kind* $* \Rightarrow *$ that, for example, maps Int (of kind $*$) to List Int $\equiv \Delta P{:}*.(\mathsf{Int} \rightarrow P \rightarrow P) \rightarrow P \rightarrow P$ (also of kind $*$). To use such definitions, equivalents of α-renaming and β-reduction must be introduced at the *type* level to effect substitution. Other examples of higher-order types and complex kinds occur in the encoding of intuitionistic logical connectives using minimal second order logic:

$$not \equiv \lambda A{:}*.A \rightarrow \Delta P{:}*.P$$

$$and \equiv \lambda A{:}*.\lambda B{:}*.\Delta P{:}*.(A \rightarrow B \rightarrow P) \rightarrow P$$

$$or \equiv \lambda A{:}*.\lambda B{:}*.\Delta P{:}*.(A \rightarrow P) \rightarrow (B \rightarrow P) \rightarrow P$$

In these examples, *not* has kind $* \Rightarrow *$, while *and* and *or* have kind $* \Rightarrow * \Rightarrow *$. As we noted earlier, observe that the idea of higher-order types takes what we might have used as *metanotation* in F_2 (giving names to complicated types we tired of writing over and over with minor changes, for instance the construction $\mathscr{L}$ in section 3.7), and embeds the notation formally in the typed λ-calculus under consideration, along with requisite substitution mechanisms at the type level.

The type systems F_k differ in the degree to which they allow this higher-order type abstraction. In F_2, no such λ-abstraction is allowed, and all types have kind $*$. In F_3, λ-abstraction is allowed only over types of kind $*$, and in F_{k+1} abstraction is allowed only over types of kinds found in F_k. In F_ω, there are no such restrictions. We can describe the kinds $\mathscr{K}_k$ allowed in F_k by a grammar:

$$\mathscr{K}_2 ::= *$$

$$\mathscr{K}_{\ell+1} ::= \mathscr{K}_\ell \mid \mathscr{K}_\ell \Rightarrow \mathscr{K}_{\ell+1}$$

$$\mathscr{K}_\omega ::= * \mid \mathscr{K}_\omega \Rightarrow \mathscr{K}_\omega$$

4.2 Syntax and inference rules for the systems $F_3, F_4, \ldots, F_\omega$

The syntax and inference rules of $F_3, F_4, \ldots, F_\omega$ are a generalisation of those found for F_2 in section 2.1. The systems differ only in their definition of kinds. Let $\mathscr{K}$ denote a grammar of kinds as described above; we then define the syntax of types and expressions as:

$$\mathscr{T} ::= \alpha \mid \mathscr{T} \rightarrow \mathscr{T} \mid \Delta\alpha{:}\mathscr{K}.\mathscr{T} \mid \lambda\alpha{:}\mathscr{K}.\mathscr{T} \mid \mathscr{T}\mathscr{T}$$

$$\mathscr{E} ::= x \mid \lambda x{:}\mathscr{T}.\mathscr{E} \mid \mathscr{E}\mathscr{E} \mid \Lambda\alpha{:}\mathscr{K}.\mathscr{E} \mid \mathscr{E}[\mathscr{T}]$$

We have the inference rules defining well formed contexts:

$$(\textit{Env-}\langle\rangle) \quad \frac{}{\mathsf{wf}(\langle\rangle)}$$

$$(\textit{Env-term}) \quad \frac{\Gamma \vdash \tau \in *}{\mathsf{wf}(\Gamma[x{:}\tau])}$$

$$(\textit{Env-type}) \quad \frac{\mathsf{wf}(\Gamma)}{\mathsf{wf}(\Gamma[\alpha{:}K])} \qquad \alpha \notin \mathsf{FV}(\Gamma)$$

Next, we define the well-formedness of types:

$$(\mathit{Type\text{-}var}) \quad \frac{\mathsf{wf}(\Gamma)}{\Gamma \vdash \alpha \in K} \qquad \Gamma(a) = K$$

$$(\mathit{Wff\text{-}}\!\rightarrow) \quad \frac{\Gamma \vdash \tau \in * \quad \Gamma \vdash \tau' \in *}{\Gamma \vdash \tau \rightarrow \tau' \in *}$$

$$(\mathit{Wff\text{-}}\Delta) \quad \frac{\Gamma[\alpha : K] \vdash \tau \in *}{\Gamma \vdash \Delta\alpha : K . \tau \in *}$$

$$(\Rightarrow\text{-}\mathit{int}) \quad \frac{\Gamma[\alpha : K] \vdash \tau \in K'}{\Gamma \vdash \lambda\alpha : K . \tau \in K \Rightarrow K'}$$

$$(\Rightarrow\text{-}\mathit{elim}) \quad \frac{\Gamma \vdash \tau \in K \Rightarrow K' \quad \Gamma \vdash \tau' \in K}{\Gamma \vdash \tau\tau' \in K'}$$

Notice also the introduction of rules defining the meaning of λ at the *type* level.

The last rules define the well-typedness of expressions, in a syntax-directed fashion. Observe that the (constant) function-type constructor $\rightarrow$ can only be applied to two terms of kind $*$.

$$(\mathit{Var}) \quad \frac{\Gamma \vdash \tau \in *}{\Gamma \vdash zx \in \tau} \qquad \Gamma(x) = \tau$$

$$(\rightarrow\text{-}\mathit{int}) \quad \frac{\Gamma \vdash \tau \in * \quad \Gamma[x : \tau] \vdash e \in \tau'}{\Gamma \vdash \lambda x : \tau . e \in \tau \rightarrow \tau'}$$

$$(\rightarrow\text{-}\mathit{elim}) \quad \frac{\Gamma \vdash e \in \tau \rightarrow \tau' \quad \Gamma \vdash e' \in \tau}{\Gamma \vdash ee' \in \tau'}$$

$$(\Delta\text{-}\mathit{int}) \quad \frac{\Gamma[\alpha : K] \vdash e \in \tau}{\Gamma \vdash \Lambda\alpha : K . e \in \Delta\alpha : K . \tau} \qquad \alpha \notin \mathsf{FV}(\Gamma)$$

$$(\Delta\text{-}\mathit{elim}) \quad \frac{\Gamma \vdash e \in \Delta\alpha : K . \tau' \quad \Gamma \vdash \tau \in K}{\Gamma \vdash e[\tau] \in \tau'[\alpha/\tau]}$$

$$(\approx) \quad \frac{\Gamma \vdash e \in \tau' \quad \tau \approx \tau' \quad \Gamma \vdash \tau \in *}{\Gamma \vdash e \in \tau}$$

In the last rule, $\tau \approx \tau'$ means that the types are $\beta\eta$-convertible.

5 Type inference for F_ω is nonelementary

To derive a nonelementary bound, we show how to type the λ-term $C\delta ID_0$, where:

$$C \equiv (\lambda f . \lambda x . f^2 x)(\lambda g_n . \lambda y_n . g_n^2 y_n)(\lambda g_{n-1} . \lambda y_{n-1} . g_{n-1}^2 y_{n-1}) \cdots (\lambda g_0 . \lambda y_0 . g_0^2 y_0)$$

and δ and ID_0 code the transition function and initial ID of a TM, as in section 3. The method we describe for typing C makes very broad assumptions about the reductions caused by δ, and thus provided a general technique for composing functions. Observe that:

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

$$C\,\delta\, ID_0 \rhd (\lambda y_1 . \lambda y_0 . y_1^{\Phi(n+2)}\, y_0)\, \delta\, ID_0 \rhd \delta^{\Phi(n+2)}\, ID_0,$$

where the function Φ is defined as $\Phi(0) = 1$, $\Phi(a+1) = 2^{\Phi(a)}$.† The technical challenge is to type C so that y_1 gets the type of δ, and y_0 the type of ID_0. The term C codes repeated exponentiation as in Example 3.1, *except* that the function δ being composed does not have the same domain and range. To understand how to compose functions with different domains and ranges, we have to examine the type of δ more closely; we abstract its structure as:

$$\delta \in \Delta v_1 : * . \Delta v_2 : * . \cdots \Delta v_r : * . \\ \bar{\mathscr{L}}(v_1, v_2, \ldots, v_r) \to \bar{\mathscr{R}}(v_1, v_2, \ldots, v_r)$$

Proposition 5.1
$\bar{\mathscr{L}}(v_1, v_2, \ldots, v_r)$ is a substitution instance of $\bar{\mathscr{R}}(w_1, w_2, \ldots, w_r)$.

Proof
They both encode TM IDs, and so are unifiable. The 'circuitry' of the unification logic exists on the '$\bar{\mathscr{L}}$' side, which induces structure on the '$\bar{\mathscr{R}}$' side. □

The construction that follows may in fact be used to type the iteration of *any* function δ, provided that the above Proposition is satisfied.

We now represent the type of δ by using higher-order type constructors. Divide the type variables $V = \{v_1, \ldots, v_r\}$ into disjoint sets $V_I = \{v_1, \ldots, v_p\}$ and $V_O = \{v_{p+1}, \ldots, v_{p+q=r}\}$, where the *output variables* V_O appear in $\bar{\mathscr{R}}(v_1, v_2, \ldots, v_r)$, and the *intermediate variables* V_I form the complement. For example, in Fig. 2, the type of *not* has a single output variable (the rightmost node labelled *p false true*), while the other external nodes of the graph comprise the intermediate variables. We can then define an *ID-constructor make-ID* as a function on types:

$$\textit{make-ID} \equiv \lambda x_1 : * . \lambda x_2 : * . \cdots \lambda x_q : * . \\ \mathscr{R}(x_1, x_2, \ldots, x_q) \in *^{q+1}$$

where we use the abbreviation $*^1 \equiv *$, $*^{a+1} \equiv * \Rightarrow *^a$, and $\mathscr{R}$ is $\bar{\mathscr{R}}$ restricted to the output variables.

Lemma 5.2
There exist type functions $\Gamma_i \in *^{r+1}$, $1 \leqslant i \leqslant q$, such that the type of δ can be represented as:

$$\delta \in \Delta v_1 : * . \Delta v_2 : * . \cdots \Delta v_r : * . \\ (\textit{make-ID}\,(\Gamma_1\, v_1\, v_2 \cdots v_r)\,(\Gamma_2\, v_1\, v_2 \cdots v_r) \cdots (\Gamma_q\, v_1\, v_2 \cdots v_r)) \\ \to \textit{make-ID}\; v_{p+1}\, v_{p+2} \cdots v_{p+q}$$

Proof
By first-order unification and Proposition 5.1. We note that the functions Γ_i encode what we have called 'TM circuitry'. In fact, for the coding of the TM we have given, the right-hand side of the type of δ consists of a single output variable, so $q = 1$,

† Recall from section 2 that the term $(\lambda s . \lambda z . s^m z)\,(\lambda s . \lambda z . s^n z)$ reduces to the normal form $\lambda s . \lambda z . s^{n^m} z$. Therefore, a λ-term consisting of a (left associating) sequence of k Church numerals for 2 will normalise to the Church numeral denoting a stack of n 2s.

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

there is only one functional Γ, and $make\text{-}ID \equiv \lambda x{:}*.x$. We maintain this more general notation to treat composition of polymorphic functions with more complex types. □

How is δ composed *polymorphically*, namely the equivalent of ML's `let` $\delta^2 = \lambda ID.\delta(\delta\, ID)$? In ML, the type of δ^2 is realised by first-order unification; we simulate this using the functions Γ_i.

Proposition 5.3
The λ-term δ^2 can be given the F_ω-type:

$$\begin{aligned}
&\Delta v_1{:}*.\Delta v_2{:}*.\cdots\Delta v_p{:}*.\Delta v_1'{:}*.\Delta v_2'{:}*.\cdots\Delta v_r'{:}*.\\
&\quad(make\text{-}ID\\
&\qquad(\Gamma_1 v_1 v_2\cdots v_p(\Gamma_1 v_1'\cdots v_r')\cdots(\Gamma_q v_1'\cdots v_r'))\\
&\qquad(\Gamma_2 v_1 v_2\cdots v_p(\Gamma_1 v_1'\cdots v_r')\cdots(\Gamma_q v_1'\cdots v_r'))\\
&\qquad\cdots\\
&\qquad(\Gamma_q v_1 v_2\cdots v_p(\Gamma_1 v_1'\cdots v_r')\cdots(\Gamma_q v_1'\cdots v_r')))\\
&\quad\to make\text{-}ID\, v_{p+1}' v_{p+2}'\cdots v_{p+q}'
\end{aligned}$$

Observe that the output variables $v_{p+1},\ldots,v_{p+q}$ in the type of δ have been instantiated so that $v_{p+i} = \Gamma_i v_1'\cdots v_r'$. The primed variables form a second *floor* of circuitry, while *make-ID puts a roof on* the type structures generated by the variables and the Γ_i. Repeated composition yields a giant directed acyclic graph, where the depth of the dag (i.e. the number of floors) is linearly proportional to the degree of composition.

The type constructors Γ_i and *make-ID* can therefore be used to define the types of λ-terms $\lambda ID.\delta(\delta\cdots(\delta\, ID)\cdots)$ in normal form, where δ is iterated some fixed number of times. However, in typing the λ-term C defined at the beginning of this section, we observe that C is not in normal form. In the next section, were use higher-order functionals to iterate the type constructors, so that the reduction of C to normal form at the value level proceeds in a well defined synchrony with reduction of the type constructors (and base types) to normal form at the type level.

5.1 Higher-order type data structures

We now show how λ-abstraction and application *at the type level* can be used to manufacture huge dags representing the t-fold composition of δ. The existence of λ at the type level allows the construction of such 'abstract' data structures.

The graphs in Figs. 5 and 6, as well as the introduction to section 5, show how the type of a function defined by (polymorphic) composition can be understood in terms of linking 'output nodes' to 'input nodes', building a larger graph with greater depth. The construction is like the construction of a large building, where each floor has the same design, and we can use the language of higher-order types to build these big buildings. For example, by λ-abstraction over the external nodes of a graph, we can 'plug in' another floor by function application at the type level, where substituting a type (i.e. graph) for a bound type variable in a higher-order type causes the graph to be grafted into the position of an external node. Since a large graph may need (type) variables at the leaves, we can use the standard λ-calculus hacks for maintaining a tuple of type variables to store a list of type variables. At every level (i.e. floor) of the construction where external nodes of the graph occur, we can use projection to get new type variables from the tuple, and plug them into the right position. The type language thus gets used as an ordinary, if somewhat arcane, programming language for building big graphs.

The basic idea is the following: we construct a certain λ-term $\mathscr{T}$ at the type level which in a precise sense *represents* the type of the j-fold composition of δ, where the kind κ of $\mathscr{T}$ does not depend on j. We will then define a function map: $\kappa \Rightarrow \kappa$ such that *map* $\mathscr{T}$ represents the type of the $(j+1)$-fold composition of δ. *Because the domain and range* (*both* kinds) *of map are identical, we can* at the type level *engage in 'conventional' function composition tricks that would not work at the expression level: the kind of map is identical to the kind of map* ∘ *map*. On the other hand, considerable technology was developed earlier for composing particular functions at the expression level (e.g. δ) that do not have identical domain and range. Notice that when we compose a function with identical domain and range we may define, for instance:

$$\begin{aligned}\bar{2}_0 &\equiv \lambda\sigma{:}\kappa \Rightarrow \kappa.\lambda\tau{:}\kappa.\sigma^2\tau\\ &\in (\kappa \Rightarrow \kappa) \Rightarrow \kappa \Rightarrow \kappa\\ \bar{2}_1 &\equiv \lambda\sigma{:}(\kappa \Rightarrow \kappa) \Rightarrow \kappa \Rightarrow \kappa.\lambda\tau{:}\kappa \Rightarrow \kappa.\sigma^2\tau\\ &\in ((\kappa \Rightarrow \kappa) \Rightarrow \kappa \Rightarrow \kappa) \Rightarrow\\ &\qquad (\kappa \Rightarrow \kappa) \Rightarrow \kappa \Rightarrow \kappa\end{aligned}$$

and write:

$$\bar{2}_1\,\bar{2}_0\,map \rhd \lambda\mathscr{T}{:}\kappa.\,map(map(map(map\,\mathscr{T})))$$

The coding of the type *map* is not pretty, but its use is quite elegant. The fundamental data structure manipulated by *map* is called a *pair*. A pair has two parts: a *prototype*, and a *variable list*.

A *prototype* is a λ-term of the form:

$$\lambda x_1{:}*.\lambda x_2{:}*.\cdots\lambda x_q{:}*.\lambda\Phi'{:}*^{q+pt+1}.\,\Phi'\phi_1\phi_2\cdots\phi_q d_1\cdots d_{pt}$$

The $d \equiv d_i$ are just 'dummy' type variables to 'pad' the kind, and the ϕ_i are types involving some set $v_1, ., v_{pt}$ of type variables, $x_1, \ldots, x_q$, and $\rightarrow$, so each ϕ_i is of kind $*$. We imagine the ϕ_i to be the dag 'under construction', so that *make-ID* $\phi_1 \cdots \phi_q$ would form a suitable $\mathscr{L}$, given type variables for the x_i.

A *variable list* is a λ-term of the form:

$$\lambda x_1{:}*.\lambda x_2{:}*.\cdots\lambda x_q{:}*.\lambda\Phi'{:}*^{q+pt+1}.\,\Phi' f_1 f_2\cdots f_q v_1\cdots v_{pt}$$

where the f_i are the output variables (i.e. external nodes) of the dag ultimately to be constructed, and the v_i are a list of type variables to be used during the construction. The λx_i-bindings are padding, since they make the kind of a prototype identical to the kind of a variable list.

A *pair* is a λ-term of the form:

$$\begin{aligned}&\lambda\Pi{:}\kappa' \Rightarrow \kappa' \Rightarrow \kappa'.\,\Pi P V\\ &\qquad \in (\kappa' \Rightarrow \kappa' \Rightarrow \kappa') \Rightarrow \kappa'\end{aligned}$$

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

where P is a prototype and V is a variable list (both of kind κ'). Because of the kind identity, fst and snd are definable on pairs:

$$\mathsf{fst} \equiv \lambda pair{:}(\kappa' \Rightarrow \kappa' \Rightarrow \kappa') \Rightarrow \kappa'.pair\,(\lambda P{:}\kappa'.\lambda V{:}\kappa'.P)$$
$$\mathsf{snd} \equiv \lambda pair{:}(\kappa' \Rightarrow \kappa' \Rightarrow \kappa') \Rightarrow \kappa'.pair\,(\lambda P{:}\kappa'.\lambda V{:}\kappa'.V)$$

Given a prototype or variable list applied to types $\tau_1, \ldots, \tau_q$ of kind $*$, the result is a tuple of kind $*^{q+pt+1} \Rightarrow *$, and we may then code terms $\mathsf{project}_{q+pt,j}$, similar to the definitions of fst and snd, which project the jth type from the tuple. For a variable list V, we write V_j for $\mathsf{project}_{q+pt,j}(V d_1 d_2 \cdots d_q)$.

The λ-term *map* maps pairs to pairs, where the new pair is one 'composition step' closer to the ultimate t-fold composition, as represented by the prototype. The definition of *map* involves straightforward list processing on pairs, where the type variables in the variable list, used as *intermediate variables* in the sense defined earlier, are repeatedly *shifted cyclically* and retrieved as the 'floors' are built; each floor represents the type of another iteration of δ.

To facilitate the description of type functionals, we use an ML-like let syntax, where let $x = E$ in B is syntactic sugar for $(\lambda x.B)E$. No ML-style 'kind polymorphism' exists in this programming style: the type language is entirely monomorphic:

$$
\begin{aligned}
map = {} & \lambda pair{:}(\kappa' \Rightarrow \kappa' \Rightarrow \kappa') \Rightarrow \kappa'.\\
& \quad \mathsf{let}\, P = \mathsf{fst}\, pair\, \mathsf{in}\\
& \quad\quad \mathsf{let}\, V = \mathsf{snd}\, pair\, \mathsf{in}\\
& \quad\quad\quad \mathsf{let}\, P' = \lambda x_1{:}*.\cdots\lambda x_q{:}*.\\
& \quad\quad\quad\quad\quad\quad \mathsf{let\, graft} = \lambda\Gamma{:}*^{r+1}.\Gamma x_1 x_2 \cdots x_q V_{q+1} V_{q+2} \cdots V_{q+p}\, \mathsf{in}\\
& \quad\quad\quad\quad\quad\quad\quad P(\mathsf{graft}\,\Gamma_1)(\mathsf{graft}\,\Gamma_2)\cdots(\mathsf{graft}\,\Gamma_q)\\
& \quad\quad\quad\quad \mathsf{in}\\
& \quad\quad\quad\quad \mathsf{let}\, V' = \lambda x_1{:}*.\cdots\lambda x_q{:}*.\lambda\Phi'{:}*^{q+pt+1} \rightarrow *.\\
& \quad\quad\quad\quad\quad\quad\quad \Phi' V_1 \cdots V_q V_{q+p+1} V_{q+p+2} \cdots V_{q+pt} V_{q+1} V_{q+2} \cdots V_{q+p}\, \mathsf{in}\\
& \quad\quad\quad\quad \lambda\Pi'{:}\kappa' \Rightarrow \kappa' \Rightarrow \kappa'.\Pi' P' V'
\end{aligned}
$$

The higher-order type functional *map* decomposes the pair into its prototype P and variable list V. It builds dags $\Gamma_i x_1 x_2 \cdots x_q V_{q+1} V_{q+2} \cdots V_{q+p}$ out of λ-abstracted type variables x_j, which mark external nodes in the dag where subsequent grafting will take place, and type variables V_{q+j} extracted from the variable list. Applying prototype P to the constructed dags grafts the dags onto the external nodes of the dags ϕ_j stored in the prototype, building the new 'floor' of constructed circuitry. The type variables v_j in variable list V are cyclically shifted in assembly-line fashion, so that applying *map* again to the just-constructed pair will select a different set of type variables to build the next 'floor' of circuitry. The variables f_j in the variable list are *not* rotated: they remain fixed, to be substituted as the final output variables.

To convert a prototype into a first-order type of the iterated polymorphic composition of δ, we substitute the 'final' type variables f_j from the variable list (which were never cyclically shifted) into the prototype, and *make-ID* is applied to the dags ϕ_j that have tediously been constructed:

$$\mathscr{I} \equiv \lambda pair\!:\!(\kappa' \Rightarrow \kappa' \Rightarrow \kappa') \Rightarrow \kappa'.$$
$$\quad \mathsf{let}\ P = \mathsf{fst}\ pair\ \mathsf{in}$$
$$\quad\quad \mathsf{let}\ V = \mathsf{snd}\ pair\ \mathsf{in}$$
$$\quad\quad\quad \mathsf{let}\ T = P V_1 V_2 \cdots V_q\ \mathsf{in}$$
$$\quad\quad\quad\quad make\text{-}ID\,(\mathsf{project}_{q+pt,1}\,T)\,(\mathsf{project}_{q+pt,2}\,T)\,(\mathsf{project}_{q+pt,q}\,T)$$

The relationship between these functionals and the type of iterative compositions of δ is given by the following lemma:

Lemma 5.4
The λ-term $\lambda ID.\delta(\delta \cdots (\delta\, ID))$, where there are t instances of δ, can be given type:

$$\Lambda d\!:\!*.\Lambda v_1\!:\!*.\Lambda v_2\!:\!*.\ \cdots \Lambda v_{q+pt}\!:\!*.$$
$$\quad \mathsf{let}\ P = \lambda x_1\!:\!*.\ \cdots \lambda x_q\!:\!*.\lambda\Phi'\!:\!*^{q+pt+1} \to *.$$
$$\quad\quad \Phi' x_1 \cdots x_q\, d \cdots d\ \mathsf{in}$$
$$\quad\quad \mathsf{let}\ V = \lambda x_1\!:\!*.\ \cdots \lambda x_q\!:\!*.\lambda\Phi'\!:\!*^{q+pt+1} \to *.$$
$$\quad\quad\quad \Phi' v_1 \cdots v_q\, v_{q+1} \cdots v_{q+pt}\ \mathsf{in}$$
$$\quad\quad\quad \mathsf{let}\ pair_0 = \lambda\Pi\!:\!\kappa' \Rightarrow \kappa' \Rightarrow \kappa'.\Pi P V\ \mathsf{in}$$
$$\quad\quad\quad\quad \mathscr{I}(map(map \cdots (map\, pair_0) \cdots)) \to \mathscr{I}(pair_0)$$

where there are t instances of map.

As an example of the use of this Lemma, when $t = 2$, we get (without quantifiers) the type described in Proposition 5.3.

5.2 Composing *map*

Now comes the elegant and truly fun part: we use the 'crank':

$$C \equiv (\lambda f.\lambda x.f^2 x)(\lambda g_n.\lambda y_n.g_n^2 y_n)(\lambda g_{n-1}.\lambda y_{n-1}.g_{n-1}^2 y_{n-1}) \cdots (\lambda g_0.\lambda y_0.g_0^2 y_0)$$

(at the *expression* level) to compose *map* $\Phi(n+2)$ times, where $\Phi(0) = 1$, $\Phi(a+1) = 2^{\Phi(a)}$. The dag gets constructed at a 'speed' controlled by the reduction sequence of C to normal form. Let $\kappa \equiv \kappa_0 \equiv (\kappa' \Rightarrow \kappa' \Rightarrow \kappa') \Rightarrow \kappa'$ be the kind of a pair, so that $map \in \kappa_0 \Rightarrow \kappa_0$, and $\mathscr{I} \in \kappa_0 \Rightarrow *$; we define $\kappa_{j+1} \equiv \kappa_j \Rightarrow \kappa_j$, and let α_j be a type variable of kind κ_{j+2}. Suppressing kinds temporarily to increase readability, we recursively define a set of types used to type C:

$$\mathscr{G}_0\{\alpha_0\} \equiv \Delta map.(\Delta\tau.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau) \to$$
$$\quad (\Delta\tau.\mathscr{I}(\alpha_0\, map\,\tau) \to \mathscr{I}\tau)$$
$$\mathscr{G}_1\{\alpha_1\} \equiv \Delta\alpha_0.\mathscr{G}_0\{\alpha_0\} \to \mathscr{G}_0\{\alpha_1\,\alpha_0\}$$
$$\equiv \Delta\alpha_0.\mathscr{G}_0\{\alpha_0\} \to$$
$$\quad \Delta map.$$
$$\quad\quad (\Delta\tau.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau) \to$$
$$\quad\quad\quad (\Delta\tau.\mathscr{I}(\alpha_1\,\alpha_0\, map\,\tau) \to \mathscr{I}\tau)$$
$$\mathscr{G}_{k+1}\{\alpha_{k+1}\} \equiv \Delta\alpha_k.\mathscr{G}_k\{\alpha_k\} \to \mathscr{G}_k\{\alpha_{k+1}\,\alpha_k\}$$

We may intuitively think of the $\alpha_j, j \geqslant 0$, as the types of higher-order Church numerals, and $\mathscr{G}_0\{\alpha_0\}$ as the type of an 'α_0-composer' which, given the transition function δ of type $\Delta\tau.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau$ as input, returns the α_0-fold composition of δ with itself. Accordingly, $\mathscr{G}_1\{\alpha_1\}$ is the type of an 'α_1-composer composer' which, given an α_0-composer, returns an $\alpha_1\alpha_0$-composer. Recall that the normal form of $\alpha_1\alpha_0$ corresponds to *exponentiation* of the respective Church numerals, in the style of the introductory example of section 2. In general, $\mathscr{G}_{k+1}\{\alpha_{k+1}\}$ is the type of a function taking a higher-order composition function of type $\mathscr{G}_k\{\alpha_k\}$, and returning a more powerful iterator of type $\mathscr{G}_k\{\alpha_{k+1}\alpha_k\}$.

Lemma 5.5
For each $0 \leqslant i \leqslant n$, $G_i \equiv \lambda g_i.\lambda y_i.g_i^2 y_i$ *can be typed as* $\mathscr{G}_i\{\bar{2}_i\}$, *where*:

$$\bar{2}_i \equiv \lambda\sigma{:}\kappa_i \Rightarrow \kappa_i.\lambda\tau{:}\kappa_i.\sigma(\sigma\tau)$$

is a type having the same kind as α_i, *namely* $\kappa_{i+2} \equiv (\kappa_i \Rightarrow \kappa_i) \Rightarrow \kappa_i \Rightarrow \kappa_i$.

Proof
For $G_0 \equiv \lambda g_0.\lambda y_0.g_0^2 y_0$, we have the construction:

$$\begin{aligned}
\mathbf{G}_0 \equiv \Lambda\, & map{:}\kappa_0 \Rightarrow \kappa_0. \\
& \lambda \mathbf{g}_0{:}\Delta\tau{:}\kappa_0.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau. \\
& \quad \Lambda\tau{:}\kappa_0. \\
& \quad\quad \lambda \mathbf{y}_0{:}\mathscr{I}(map(map\,\tau)) \equiv \mathscr{I}(\bar{2}_0\, map\,\tau). \\
& \quad\quad\quad \mathbf{g}_0[\tau](\mathbf{g}_0[map\,\tau]\,\mathbf{y}_0)
\end{aligned}$$

and for $G_{i+1} \equiv \lambda g_{i+1}.\lambda y_{i+1}.g_{i+1}^2 y_{i+1}, i \geqslant 0$, we have the construction:

$$\begin{aligned}
\mathbf{G}_{i+1} \equiv \Lambda & \alpha_i{:}\kappa_{i+2}. \\
& \lambda \mathbf{g}_{i+1}{:}\mathscr{G}_i\{\alpha_i\} \equiv \Delta\alpha_{i-1}{:}\kappa_{i+1}.\mathscr{G}_{i-1}\{\alpha_{i-1}\} \to \mathscr{G}_{i-1}\{\alpha_i\alpha_{i-1}\} \\
& \quad \Lambda\alpha_{i-1}{:}\kappa_{i+1}. \\
& \quad\quad \lambda \mathbf{y}_{i+1}{:}\mathscr{G}_{i-1}\{\alpha_{i-1}\}. \\
& \quad\quad\quad \mathbf{g}_{i+1}[\alpha_i\alpha_{i-1}](\mathbf{g}_{i+1}[\alpha_{i-1}]\,\mathbf{y}_{i+1})
\end{aligned}$$

In this term, notice that:

$$\begin{aligned}
\mathbf{g}_{i+1}[\alpha_{i-1}] &\in \mathscr{G}_{i-1}\{\alpha_{i-1}\} \to \mathscr{G}_{i-1}\{\alpha_i\alpha_{i-1}\} \\
\mathbf{g}_{i+1}[\alpha_i\alpha_{i-1}] &\in \mathscr{G}_{i-1}\{\alpha_i\alpha_{i-1}\} \to \mathscr{G}_{i-1}\{\alpha_i(\alpha_i\alpha_{i-1})\}
\end{aligned}$$

□

Lemma 5.6
In the term C:

$$F \equiv \lambda f.\lambda x.f^2 x \in \mathscr{G}_n\{\bar{2}_n\} \to \mathscr{G}_{n-1}\{\bar{2}_{n-1}\} \to \mathscr{G}_{n-1}\{\bar{2}_{n+1}\bar{2}_n\bar{2}_{n-1}\}$$

Proof
We have the construction:

$$\begin{aligned}
\mathbf{F} \equiv \lambda & \mathbf{f}{:}\mathscr{G}_n\{\bar{2}_n\}. \\
& \lambda \mathbf{x}{:}\mathscr{G}_{n-1}\{\bar{2}_{n-1}\}. \\
& \quad \mathbf{f}[\bar{2}_n\bar{2}_{n-1}](\mathbf{f}[\bar{2}_{n-1}]\,\mathbf{x})
\end{aligned}$$

Recall $\mathscr{G}_k\{\bar{2}_k\} \equiv \Delta\alpha_{k-1}:\kappa_{k+1}.\mathscr{G}_{k-1}\{\alpha_{k-1}\} \to \mathscr{G}_{k-1}\{\bar{2}_k\,\alpha_{k-1}\}$; with the given parameterisations, the subterms are typed as:

$$\mathbf{f}[\bar{2}_{n-1}] \in \mathscr{G}_{n-1}\{\bar{2}_{n-1}\} \to \mathscr{G}_{n-1}\{\bar{2}_n\,\bar{2}_{n-1}\}$$
$$\mathbf{f}[\bar{2}_n\,\bar{2}_{n-1}] \in \mathscr{G}_{n-1}\{\bar{2}_n\,\bar{2}_{n-1}\} \to \mathscr{G}_{n-1}\{\bar{2}_n(\bar{2}_n\,\bar{2}_{n-1})\}$$

However, observe that by β-reduction at the type level, $\bar{2}_n(\bar{2}_n\,\bar{2}_{n-1})$ and $\bar{2}_{n+1}\,\bar{2}_n\,\bar{2}_{n-1}$ are equivalent. □

Theorem 5.7

Recall the definitions of **F** and $\mathbf{G}_j$ *from Lemmas* 5.5 *and* 5.6. *Then the term C* (*the 'crank'*) *has typing*:

$$\mathbf{C} \equiv \mathbf{F}\mathbf{G}_n\,\mathbf{G}_{n-1}[\bar{2}_{n-2}]\,\mathbf{G}_{n-2}[\bar{2}_{n-3}]\cdots\mathbf{G}_1[\bar{2}_0]\,\mathbf{G}_0$$

so that:

$$\begin{aligned}\mathbf{C} &\in \mathscr{G}_0\{\bar{2}_{n+1}\,\bar{2}_n\cdots\bar{2}_0\}\\ &\equiv \Delta map:\kappa_0 \Rightarrow \kappa_0.(\Delta\tau:\kappa_0.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau) \to\\ &\qquad(\Delta\tau:\kappa_0.\mathscr{I}((\bar{2}_{n+1}\,\bar{2}_n\cdots\bar{2}_0)\,map\,\tau) \to \mathscr{I}\tau)\\ &\rhd \Delta map:\kappa_0 \Rightarrow \kappa_0.(\Delta\tau:\kappa_0.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau) \to\\ &\qquad(\Delta\tau:\kappa_0.\mathscr{I}(map^{\Phi(n+2)}\tau) \to \mathscr{I}\tau)\end{aligned}$$

Proof

From Lemmas 5.5 and 5.6, we know that:

$$\mathbf{F}\mathbf{G}_n\,\mathbf{G}_{n-1} \in \mathscr{G}_{n-1}\{\bar{2}_{n+1}\,\bar{2}_n\,\bar{2}_{n-1}\} \equiv \Delta\alpha_{n-2}:\kappa_n.\mathscr{G}_{n-2}\{\alpha_{n-2}\} \to \mathscr{G}_{n-2}\{\bar{2}_{n+1}\,\bar{2}_n\,\bar{2}_{n-1}\,\alpha_{n-2}\}$$

By parameterising this term with type $\bar{2}_{n-2}$, we derive a term of type:

$$\mathscr{G}_{n-2}\{\bar{2}_{n-2}\} \to \mathscr{G}_{n-2}\{\bar{2}_{n+1}\,\bar{2}_n\,\bar{2}_{n-1}\,\bar{2}_{n-2}\}$$

However, by Lemma 5.5, we know $\mathbf{G}_{n-2} \in \mathscr{G}_{n-2}\{\bar{2}_{n-2}\}$, so that:

$$\begin{aligned}\mathbf{F}\mathbf{G}_n\,\mathbf{G}_{n-1}[\bar{2}_{n-2}]\,\mathbf{G}_{n-2} &\in \mathscr{G}_{n-2}\{\bar{2}_{n+1}\,\bar{2}_n\,\bar{2}_{n-1}\,\bar{2}_{n-2}\}\\ &\equiv \Delta\alpha_{n-3}:\kappa_{n-1}.\mathscr{G}_{n-3}\{\alpha_{n-3}\} \to \mathscr{G}_{n-3}\{\bar{2}_{n+1}\,\bar{2}_n\,\bar{2}_{n-1}\,\bar{2}_{n-2}\,\alpha_{n-3}\}\end{aligned}$$

Formally, the proof consists of an induction on n; informally, we continue to 'unwind' the above construction. Each type parameterisation $\bar{2}_j$, followed by application to $\mathbf{G}_j$, augments the exponential stack contained in the type with another $\bar{2}$. The reduction of the stack of $\bar{2}$s to normal form (at the type level) results in a nonelementary Church numeral applying *map* to a (Δ-bound) pair τ. □

Theorem 5.8

Recognising the lambda terms of length n typable in F_ω *is* $DTIME[\Phi(n^t)]$*-hard for any integer* $t \geq 1$ *under logspace reduction, where*:

$$\Phi(0) = 1,$$
$$\Phi(a+1) = 2^{\Phi(a)}$$

https://doi.org/10.1017/S0956796800001143 Published online by Cambridge University Press

Proof

Again, we simulate the computation of a TM M accepting or rejecting its input x in $\Phi(|x|^k)$ steps by a λ-term $\Psi_{M,x}$, where M accepts x if and only if $\Psi_{M,x}$ is typable.

Using the typing of C in Theorem 5.7, we type $\hat{M} \equiv C\,\delta\,ID_0$ as:

$$\hat{\mathbf{M}} \equiv (\Lambda d{:}*.\Lambda v_1{:}*.\cdots\Lambda v_{q+pt}{:}*.\mathbf{C}[map]\,\delta[pair_0])$$
$$[\mu_d][\mu_1]\cdots[\mu_{q+pt}](ID_0[\pi])$$

Observe that we take the type of the 'crank' and parameterise it with the definition of *map*; since $\delta \in (\Delta\tau.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau)$, we know $\mathbf{C}[map]\,\delta \in \Delta\tau{:}\kappa_0.\mathscr{I}((\bar{2}_{n+1}\bar{2}_n\cdots\bar{2}_0)\,map\,\tau) \to \mathscr{I}\tau$. Next, parameterise this term over the *initial pair* $pair_0$ defined in Lemma 5.4, and abstract over all the variables v_j appearing in the pair. The term thus constructed is the $\Phi(t+2)$-fold unwinding of the transition function, and its outermost-quantified first order type codes the reductions on an arbitrary ID as a directed acyclic graph.

To apply this term to an initial ID, we need to instantiate the quantified variables v_j so that the left-hand side of the type is identical to a typing of the initial ID. The parameterisation of each v_j by a type μ_j, and the complementary parameterisation π of the initial ID, simulates the unification process of the ML type inference algorithm. The instantiations are huge: the substitutions for intermediate variables on the 'top floor' of the type will have non-elementary size. These instantiations must code the computations of the TM on the input.

The proof of the theorem now concludes exactly in the style of Theorem 3.5. By choosing the parameterisations of the output variables carefully, the term:

$$A \equiv \langle q, L, R\rangle = \hat{M}; \quad q\,\mathit{false}\ldots\mathit{false}\;\mathit{true}\ldots\mathit{true}$$

where the instances of *true* and *false* are appropriate substitutions of Boolean terms for the accepting and rejecting states, will be given type $\Delta\alpha{:}*.\Delta\beta{:}*.\alpha\to\beta\to\alpha$ if the TM accepts its input, and $\Delta\alpha{:}*.\Delta\beta{:}*.\alpha\to\beta\to\beta$ if the TM rejects its input. Then $\Psi_{M,x} \equiv (\lambda x.xx)(A(\lambda x.x)(\lambda y.yy))$ is typable if and only if the TM accepts its input. In the case of an accepting computation, the typing is straightforward; in the case of a rejecting computation, we appeal to the fact that all typable terms in F_ω are strongly normalising, and the term we have constructed is divergent. □

Corollary 5.9

Recognising the lambda terms of length n typable in F_k is DTIME$[f_{k-4}(n)]$-hard under logspace reduction, where:

$$f_0(n) = n$$
$$f_{k+1}(n) = 2^{f_k(n)}$$

Proof

The proof of the corollary is identical to that of the previous theorem, except that the restrictions on kinds imposed by F_k do not allow as powerful a 'crank' as given in Theorem 5.7, because the type language does not have functionals of high enough order.

In particular, the 'kinding' of prototypes and variable lists require the degree of higher-order abstraction found in system F_3; this assertion can be verified by examining their respective kinds and the grammar of kinds found at the end of section 4.1. As described in section 5.1, pairs are representable in F_4, and *map* and $\mathscr{I}$ are representable in F_5. The iterator $\bar{2}_j$ is then representable in F_{j+6}. As a consequence, the typing of the term **C** in Theorem 5.7 is realised in F_{n+6}. Note that $\mathscr{G}_j\{\bar{2}_j\}$ can then be represented in F_{j+5} by explicitly iterating the $\bar{2}_j$, i.e.:

$$\mathscr{G}_j\{\bar{2}_j\} \equiv \Delta\alpha_{j-1}:\kappa_{j+1}.\mathscr{G}_{j-1}\{\alpha_{j-1}\} \to \mathscr{G}_{j-1}\{\lambda\tau:\kappa_{j-2}.\alpha_{j-1}(\alpha_{j-1}\tau)\}$$

To achieve the simulation of an arbitrary TM for $f_{k-4}(n)$ steps, we repeat the argument of Theorem 5.7, except that we replace the term F by $N \equiv \lambda f.\lambda x.f^n x$, which can be given type:

$$\mathbf{N} \in \mathscr{G}_{k-5}\{\bar{2}_{k-5}\} \to \mathscr{G}_{k-6}\{\bar{2}_{k-6}\} \to \mathscr{G}_{k-6}\{\bar{n}_{k-4}\,\bar{2}_{k-5}\,\bar{2}_{k-6}\}$$

where $\bar{n}$ is the analogous Church numeral for n at the type level. By once again carefully manipulating the typing, we can represent $\bar{n}_{k-4}\,\bar{2}_{k-5}\,\bar{2}_{k-6}$ instead by a reduced form of the iteration, that is, $\bar{2}_{k-5}(\bar{2}_{k-5}(\cdots(\bar{2}_{k-5}\,\bar{2}_{k-6})\cdots))$, where $\bar{2}_{k-5}$ is iterated n times. The typed 'crank' that iterates the computation is then:

$$\mathbf{C}_k \equiv \mathbf{N}\mathbf{G}_{k-5}\,\mathbf{G}_{k-6}\,[\bar{2}_{k-7}]\,\mathbf{G}_{k-7}\,[\bar{2}_{k-8}]\cdots\mathbf{G}_1\,[\bar{2}_0]\,\mathbf{G}_0$$

so that:

$$\begin{aligned}
\mathbf{C}_k \in\ & \mathscr{G}_0\{\bar{n}_{k-4}\,\bar{2}_{k-5}\cdots\bar{2}_0\} \\
\equiv\ & \Delta map:\kappa_0 \Rightarrow \kappa_0.(\Delta\tau:\kappa_0.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau) \to \\
& \quad (\Delta\tau:\kappa_0.\mathscr{I}((\bar{n}_{k-4}\,\bar{2}_{k-5}\cdots\bar{2}_0)\,map\,\tau) \to \mathscr{I}\tau) \\
\rhd\ & \Delta map:\kappa_0 \Rightarrow \kappa_0.(\Delta\tau:\kappa_0.\mathscr{I}(map\,\tau) \to \mathscr{I}\tau) \to \\
& \quad (\Delta\tau:\kappa_0.\mathscr{I}(map^{f_{k-4}(n)}\,\tau) \to \mathscr{I}\tau)
\end{aligned}$$

□

The constant '-4' reflects the *kind overhead* of building pairs: the data structures in the construction, as well as the functionals acting on them, require a certain level of kind abstraction. We make no claim as to the optimality of this overhead; the goal of the analysis has rather been to show, for sufficiently large k, a bound on type inference for F_{k+1} that is at least exponentially harder than that for F_k. The cost of improving the constant would almost certainly be a pedagogically unnecessary complication of the proof.

6 Discussion; open problems

We have provided the first lower bounds on type inference for the Girard/Reynolds system F_2 and the extensions $F_3, F_4, \ldots F_\omega$. The lower bounds involve generic simulation of Turing Machines, where computation is simulated at the expression and type level simultaneously. Non-accepting computations are mapped to non-normalising reduction sequences, and hence non-typable terms. The accepting computations are mapped to typable terms, where higher-order types encode the reduction sequences, and first-order types encode the entire computation as a circuit, based on a unification simulation of Boolean logic. Our lower bounds employ combinatorial techniques which we hope will be useful in the ultimate resolution of the F_2 type inference problem, particularly the idea of composing polymorphic functions with different domains and ranges.

Even if our bounds are weak (if the F_2 problem is undecidable, they certainly are!), the analysis puts forward a certain *program*; it remains to be seen how far that program can be pushed. While the higher-order systems are of genuine interest, it is F_2 which occupies centre stage: in particular, we would like to know if the technique of the higher-order lower bounds can be 'lowered' to F_2, somehow using the F_2 *ranks* to simulate the expressiveness we have obtained from the *kinds* in $F_3, F_4, \ldots, F_\omega$. The computational power of the kinds includes not merely higher-order quantification, but more importantly β-reduction at the type level.

Generic simulation is a natural setting for lower bounds, particularly when the complexity classes are superexponential, and there are few difficult combinatorial problems on which to base reductions. It seems equally natural that the *type information* added to an (untyped) term is of a length proportional to the time complexity of the TM being simulated. Furthermore, the program of generic simulation generalises nicely, as expressed in the slogan 'how fast can the crank (of the transition function) be turned?': better lower bounds can be proven by analysing different 'cranks'. We observe in particular that the typing outlined in section 5 was discovered by studying the reduction sequence of the *untyped* term C to normal form, and constructing the type as an *encoding* of that sequence. This analysis suggests an examination of F_2 types, particularly in the light of the strong normalisation theorem, as *encodings of reduction sequences*. Of course, these encodings are in general ambiguous since, for example, different Church numerals are not interconvertible under β- and η-reduction, and as such they cause different reductions to take place when used as iterators, yet they have the same type. Note, however, that the programming style used to derive our lower bounds avoids exactly this kind of ambiguity: this is the essence of the duality of terms and types.

The non-elementary lower bound for F_ω type inference should immediately call to mind a well-known theorem of Statman: the theorem states that if we have two λ-terms typable in the *first order* typed lambda calculus, deciding whether the terms have the same normal form requires nonelementary time (Statman, 1979; Mairson 1992b). The proof of Statman's theorem is a reduction from deciding the truth of expressions in higher-order logic, where quantification is allowed not only over Boolean values, but over higher-order functions over Booleans (Meyer, 1974). Every formula in higher-order logic is transformed, using the reduction, into a λ-term that β-reduces to the standard term $\lambda t{:}\sigma.\lambda f{:}\sigma.t$ coding 'true' if and only if the formula is true, and otherwise to the term $\lambda t{:}\sigma.\lambda f{:}\sigma.f$ coding 'false'. We wish to emphasise both the abstract structural similarities between these results and the lower bounds described in this paper, as well as the necessary and profound structural differences at the level of detailed coding.

Introducing higher-order quantification and abstraction mechanisms in a calculus allows greater expressiveness and succinctness, and as a consequence, decision problems relating to expressions in the calculus invariably require greater computational resources. For example, deciding whether a propositional formula is true under a particular substitution of **true** and **false** for the variables is complete for polynomial time; this is the well- known *circuit value* problem (Ladner, 1975). When we existentially quantify over the propositional variables, asking instead whether there exists a substitution for which the formula is true, we get the *satisfiability* problem, complete for nondeterministic polynomial time (Cook, 1971; Garey and Johnson, 1979). By alternating existential and universal quantifiers, we derive the polynomial time hierarchy, and in the limit, the problem of *quantified boolean formulas*, complete for polynomial space (Stockmeyer and Meyer, 1973; Garey and Johnson, 1979). Finally, if we allow quantification over functions of Booleans, functions of functions of Booleans, etc., we get a problem complete for nonelementary time (Meyer, 1974; Statman, 1979).

The theorems described in this paper follow much the same pattern. First-order unification is complete for polynomial time (Dwork *et al.*, 1974) corresponding to the complexity of first-order type inference. The progressively stronger lower bounds in this paper are derived by similarly allowing greater and greater functional abstraction on types. A further similarity between the Statman theorem and the F_ω-bound is the particular use of the Church numeral for 2 as an iteration mechanism.

However, it is the problem of type inference, and not type equivalence, that is addressed in this paper. The structural similarity we have outlined above is between higher-order logic and the *type language* of F_ω, yet the problem of type inference is not really about the type language, but rather a decision problem about untyped λ-terms. As a consequence, while in the problems of Meyer and Statman, where the logic (equivalently, λ-terms) can be manipulated *directly*, there is a certain inescapable *indirection* in our construction, where the types can only be 'manipulated' by terms at the value level. To render this manipulation unambiguous, we have used the idea of making reductions at the value level correspond exactly to constructs at the type level. There is, of course, no such correspondence when the term at the value level does not strongly normalise, a situation that has no proper analogue. To summarise, while there are certainly high-level similarities – and indeed, this is what gives a certain classical flavour to our analysis from the perspective of complexity theory – the details are quite dissimilar, and for important structural reasons.

Finally, we should observe as well the pitfalls of the methods introduced and used in this paper, or at least the hurdles which wait to be surmounted. The 'cranks' described are all strongly normalising in a manner such that, while one might aspire to better lower bounds (say, at the level of non-primitive recursion) we will never get an undecidability result. As long as we pursue bounds for F_2 based on expressiveness of the type language, we are constrained by the strong normalisation theorem, and the representation theorem (that the representable integer functions are those provably total in second order Peano Arithmetic) (Girard, 1972, Girard *et al.*, 1989). We have some idea how to get around the first hurdle, but are a bit puzzled by the second. Does it seem possible that the representation theorem would allow reduction sequences of functionally unbounded length on typable terms?

This latter suggestion deserves further development; we sketch here how such an undecidability proof might look. It is clear that any theorem of the form 'λ-term E is typable iff TM M halts' cannot be proved if E contains the fixpoint combinator

$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, since $\mathbf{Y}$ is not strongly normalising, and hence not typable in any of the type systems we have considered. However, consider the following variant of $\mathbf{Y}$:

$$\bar{\mathbf{Y}} \equiv \lambda f.(\lambda x.f(\lambda y.yxx))(\lambda x.f(\lambda y.yxx))$$

Notice that $\bar{\mathbf{Y}}$ is, in contrast, strongly normalising: writing $\bar{\mathbf{Y}} \equiv \lambda f.H_f H_f$, its normal form is $\lambda f.f(\lambda y.yH_f H_f)$. Could we use such a combinator to code an unbounded computation?

Consider a λ-term F of the form:

$$F \equiv \lambda choose.\lambda ID.choose((halt?\,ID)(\lambda p.\lambda q.I)I)(\delta\, ID)$$

where we code a halting predicate *halt*? on IDs, using conventional techniques. Let ID_0 be an initial ID; then in the case that $halt?\ ID_0 \rhd true \equiv \lambda x.\lambda y.x$, so that a halting configuration has been reached, we have:

$$\begin{aligned}
\bar{\mathbf{Y}}F\,ID_0 &\rhd F(\lambda y.yH_F H_F)\,ID_0\\
&\rhd (\lambda y.yH_F H_F)((halt?\,ID_0)(\lambda p.\lambda q.I)I)(\delta\, ID_0)\\
&\rhd (\lambda y.yH_F H_F)(\lambda p.\lambda q.I)(\delta\, ID_0)\\
&\rhd (\lambda p.\lambda q.I)H_F H_F(\delta\, ID_0)\\
&\rhd \delta\, ID_0\\
&\rhd ID_0
\end{aligned}$$

On the other hand, if $halt?\ ID_0 \rhd false \equiv \lambda x.\lambda y.y$, we derive:

$$\begin{aligned}
\bar{\mathbf{Y}}F\,ID_0 &\rhd F(\lambda y.yH_F H_F)\,ID_0\\
&\rhd (\lambda y.yH_F H_F)((halt?\,ID_0)(\lambda p.\lambda q.I)I)(\delta\, ID_0)\\
&\rhd (\lambda y.yH_F H_F)I(\delta\, ID_0)\\
&\rhd H_F H_F(\delta\, ID_0)\\
&\rhd F(\lambda y.yH_F H_F)(\delta\, ID_0)
\end{aligned}$$

where the latter term is also a reduct of $\bar{\mathbf{Y}}F(\delta\, ID_0)$. Instead of an uncontrolled unwinding $F(F(F(\cdots)))$ via the $\mathbf{Y}$-combinator, we get a controlled unwinding guided at every step by the state of the TM computation.

This construction shows that the set of strongly normalising terms is not recursive, by constructing a term that strongly normalises iff a particular TM computation halts.† In the case of a divergent TM computation, the λ-term is clearly not normalising, and hence not typable. In the case of a convergent computation, can the λ-term be typed? We mentioned above our puzzlement with the representation theorem, yet the solution to the puzzle may be that unbounded computation is indeed allowed; however, the type of the (strongly normalising) term explicitly codes the reduction sequence.

We conclude with a final *caveat lector*. The lower bound we have proven for F_ω is unlikely to be improved further by naively trying a better 'crank', unless the

† This is, of course, a simple corollary of the Scott–Curry undecidability theorem (see, for example, Hindley and Seldin, 1986).

foundation of the simulation is changed substantially. The explanation of this limitation is that the *type* language of F_ω is fundamentally, as we have described earlier, the first-order typed λ-calculus with a single type constant (∗). The 'duality' approach forces reductions at the expression level to match those at the type level, and a result of Schwichtenberg (1982) indicates that our construction is using the type language at its maximum capacity. Encouraged and excited as we are to have made progress on these open questions in programming language theory, the hard work may have only just begun.

Acknowledgements

The results of section 3 were reported earlier by Henglein (1990). For their encouragement, suggestions and criticisms, we thank Paris Kanellakis, Georg Kreisel, Daniel Leivant, Angus Macintyre, Albert Meyer, Jon Riecke and Rick Statman. Many thanks to David Klein for his willing help with the figures, and to the editors for anglicising our spelling. The second author wishes to acknowledge the generosity of the Computer Science Department at UC Santa Barbara, the Music Academy of the West, and the Cate School of Carpenteria, for their hospitality during his visit to Santa Barbara in the summer of 1990. He would also like to thank the Cambridge Research Laboratory of Digital Equipment Corporation, where final work on the paper was completed.

References

Appel, A.W. and Jim, T. (1989) Continuation-passing, closure-passing style. In: *Proc. 16th ACM Symposium on the Principles of Programming Languages*, pp. 293–302, January.

Cardelli, L. (1989) Typeful programming. Lecture Notes for the *IFIP Advanced Seminar on Formal Methods in Programming Language Semantics*, Rio de Janeiro, Brazil (see also SRC Report 45, Digital Equipment Corporation).

Cook, S.A. (1971) The complexity of theorem-proving procedures. In: *Proc. 3rd Annual ACM Symposium on the Theory of Computing*, pp. 151–158.

Damas, L. (1985) *Type assignment in programming languages.* PhD dissertation, CST-33-85, Computer Science Department, Edinburgh University.

Damas, L. and Milner, R. (1982) Principal type schemes for functional programs. In: *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212, January.

Dwork, C., Kanellakis, P.C. and Mitchell, J.C. (1984) On the sequential nature of unification. *J. Logic Programming* **1**:35–50.

Gallier, J. (1990) On Girard's 'Candidats de Reducibilité'. In: *Logic and Computer Science* (P. Odifreddi, ed.), pp. 123–203. Academic Press.

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

Giannini, P. and Ronchi Della Rocca, S. (1988) Characterization of typings in polymorphic type discipline. In: *Proc. 3rd IEEE Symposium on Logic in Computer Science*, pp. 61–70, July.

Girard, J.-Y. (1972) *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieur.* Thèse de Doctorat d'Etat, Université de Paris VII.

Girard, J.-Y., Lafont, Y. and Taylor, P. (1989) *Proofs and Types.* Cambridge University Press.

Harper, R., Milner, R. and Tofte, M. (1990) *The Definition of Standard ML.* MIT Press.

Hartmanis, J. and Stearns, R.E. (1965) On the computational complexity of algorithms. *Trans. American Math. Soc.* **117**: 285–306.

Henglein, F. (1990) *A lower bound for full polymorphic type inference: Girard/Reynolds typability is DEXPTIME-hard.* University of Utrecht, Technical Report RUU-CS-90-14, April.

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

Hindley, R. (1969) The principal type scheme of an object in combinatory logic. *Trans. American Math. Soc.* **146**:29–60.

Hindley, J.R. and Seldin, J.P. (1986) *Introduction to Combinators and Lambda Calculus.* Cambridge University Press.

Hopcroft, J.E. and Ullman, J.D. (1979) *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

Hudak, P. and Wadler, P.L. (eds.) (1988) *Report on the functional programming language Haskell.* Yale University Technical Report YALEU/DCS/RR656.

Kanellakis, P.C. and Mitchell, J.C. (1989) *Polymorphic unification and ML typing.* Brown University Technical Report CS-89-40, August 1989. (Also in *Proc. 16th ACM Symposium on the Principles of Programming Languages*, pp. 105–115, January.)

Kanellakis, P.C., Mairson, H.G. and Mitchell, J.C. (1991) Unification and ML type reconstruction. In: *Computational Logic: Essays in Honor of Alan Robinson.* (J.-L. Lassez and G. Plotkin; eds.). MIT Press.

Kelsey, R. and Hudak, P. (1989) Realistic Compilation by Program Transformation. In: *Proc. 16th ACM Symposium on the Principles of Programming Languages*, pp. 281–292, January.

Kfoury, A.J., Tiuryn, J. and Urzyczyn, P. (1990) ML typability is DEXPTIME-complete. In: *Proc. 15th Colloquium on Trees in Algebra and Programming*, May. (See also Boston University Technical Report, October 1989.)

Kfoury, A.J. and Tiuryn, J. (1990) *Type reconstruction in finite rank fragments of the second-order lambda calculus.* Technical Report BUCS 89-11, Boston University, October. (Also in *Proc. 5th IEEE Symposium on Logic in Computer Science*, pp. 2–11, June.)

Ladner, R.E. (1975) The circuit value problem is log space complete for *P. SIGACT News* **7**(1): 18–20.

Landin, P. (1966) The next 700 programming languages. *Commun. ACM* **9**(3): 157–166.

Mairson, H.G. (1990) Deciding ML typability is complete for deterministic exponential time. In: *Proc. 17th ACM Symposium on the Principles of Programming Languages*, pp. 382–401, January.

Mairson, H.G. (1992a) Quantifier elimination and parametric polymorphism in programming languages. *J. Functional Programming* **2**(2): 213–226, April.

Mairson, H.G. (1992b) A simple proof of a theory of Statman. *Theoretical Computer Science* **103**: 387–394.

Meyer, A.R. (1974) The inherent computational complexity of theories of ordered sets. In: *Proc. Int. Congress of Mathematicians*, pp. 477–482.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Computer and System Sciences* **17**: 348–375.

Mitchell, J.C. (1990) Type systems for programming languages. In: *Handbook of Theoretical Computer Science, vol. B*, pp. 365–468 (J. van Leeuwen *et al.*, eds). North-Holland.

Paterson, M.S. and Wegman, M.N. (1978) Linear unification. *J. Computer and System Sciences* **16**: 158–167.

Pfenning, F. and Lee, P. (1989) LEAP: a language with eval and polymorphism. In: *TAPSOFT 1989: Proc. Int. Joint Conference on Theory and Practice in Software Development*, Barcelona, Spain. (See also CMU Ergo Report 88–065.)

Pfenning, F. (1988) Partial polymorphic type inference and higher-order unification. In: *Proc. ACM Conference on Lisp and Functional Programming*, pp. 153–163.

Pierce, B., Dietzen, S. and Michaylov, S. (1989) *Programming in higher-order typed lambda calculi.* Technical Report CMU-CS-89-111, Carnegie Mellon University, March.

Reynolds, J.C. (1974) Towards a theory of type structure. In *Proc. Paris Colloquium on Programming: Lecture Notes in Computer Science 19*, pp. 408–425. Springer-Verlag.

Robinson, J.A. (1965) A machine oriented logic based on the resolution principle. *J. ACM* **12** (1): 23–41.

Sannella, D.T. (ed.) (1988) *Postgraduate Examination Questions in Computation Theory, 1978–1988*. Laboratory for Foundations of Computer Science, Report ECS-LFCS-88-64.

Schwichtenberg, H. (1982) Complexity of normalisation in the pure typed lambda calculus. In: *The L. E. J. Brouwer Centenary Symposium* (A.S. Troelstra and D. van Dalen, eds.), pp. 453–457. North-Holland.

Scott, D.S. (1977) Logic and programming languages. *Commun. ACM* **20** (9): 634–641.

Statman, R. (1979) The typed λ-calculus is not elementary recursive. *Theoretical Computer Science* **9**: 73–81.

Stockmeyer, L.J. and Meyer, A.R. (1973) Word problems requiring exponential time. In: *Proc. 5th Annual ACM Symposium on Theory of Computing*, pp. 1–9.

Stoy, J. (1977) *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press.

Strachey, C. (1973) *The varieties of programming language*. Technical Monograph PRG-10, Programming Research Group, Oxford University.

Tait, W.W. (1967) Intensional interpretation of functionals of finite type I. *J. Symbolic Logic* **32**: 198–212.

Turner, D.A. (1985) Miranda: A non-strict functional language with polymorphic types. In: *IFIP Int. Conference on Functional Programming and Computer Architecture: Lecture Notes in Computer Science 201*, pp. 1–16. Springer-Verlag.

Wand, M. (1989) A simple algorithm and proof for type inference. *Fundamenta Informaticae* **10**.

Wand, M. (1992) Correctness of Procedure Representations in Higher-Order Assembly Language. In: *Mathematical Foundations of Programming Language Semantics 1991: Lecture Notes in Computer Science 598*, (S. Brookes, ed.), pp. 294–311. Springer-Verlag.