The Complexity of Type Inference for Higher-Order Typed Lambda Calculi

*Fritz Henglein**
Courant Institute of Mathematical Sciences
New York University
New York, N.Y. 10012
and
Computer Science Department
Utrecht University
3508 TB Utrecht
The Netherlands

Harry G. Mairson†
Computer Science Department
Brandeis University
Waltham, Massachusetts 02254

Abstract

We analyze 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 recognizing the F_2-typable terms requires exponential time, and for F_ω the problem is nonelementary. We show as well a sequence of lower bounds on recognizing 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-normalizing 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.

*Supported in part by Office of Naval Research grant N0014-90-J-1110

†Supported by grants from Texas Instruments and from the Tyson Foundation.

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 [Gir72] and John Reynolds [Rey74]. 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 tantalizingly 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 recognizing the F_2-typable terms requires exponential time, and for F_ω the problem is nonelementary. We show as well a sequence of lower bounds on recognizing 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 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.

The problem of type inference is one of both theoretical and practical interest. Following the insights of Landin [Lan66], Strachey [Str73], Penrose[1], and others, the untyped λ-calculus has long been recognized 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 simulate "final" returned answers.

Types augment programming languages with additional guarantees about resultant computational behavior. 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* ("code reuse"), so that programs designed for abstract data types must be recoded 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 realization 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 methodology was proposed by Robin Milner, namely a theory of *type polymorphism* for achieving code reuse, while retaining the benefits of strong typing. He gave an algorithm which, presented with an untyped program, could construct the most general type information (known as the *principal type* [Hin69, DM82]) for the program [Mil78]. These insights are implemented in the ML programming language [HMT90] as well as a variety of other functional languages [HW88, Tur85]. 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 realization of these basic intuitions.

The ML language is not merely an example of successful software engineering. It is also an important departure point and testbed in our theoretical examination of type inference: studying type inference for ML has provided important insights. The "Core ML" language comprising typed λ-calculus with polymorphism (as embodied in `let`) enjoys the *strong normalization property:* typable programs are guaranteed to terminate[2]. 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 nontrivial subset of the λ-terms typable in F_2, F_3,..., F_ω. Furthermore, all of these type systems enjoy strong normalization. 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 "just" an esoteric variation on F_2, since it has been proposed as the mathematical foundation for a new generation of typed functional programming languages, in particular Cardelli's language Quest [Car89] and the LEAP project at CMU [PL89].

The lower bounds presented here are all *generic reductions*, where an arbitrary 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 [HS65, HU79], these complexity-class relativized hardness bounds translate (via diagonalization arguments) to nonrelativized 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 [KMM90]:

Proposition 1.1 *Given a strongly normalizing 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, $(\delta\ y)$ β-reduces to a λ-term encoding the ID *after* a state transition. Let $\overline{f}$ be the Church-numeral encoding of f, $\overline{n}$ be the Church numeral for n, and ID_0 be the encoding of the initial ID. Consider the typing of the normal form of $E' \equiv \overline{f}\ \overline{n}\ \delta\ ID_0 \rhd \overline{f(n)}\ \delta\ ID_0 \rhd \delta^{f(n)}\ ID_0$. The normal form of

[1] In his 1977 Turing Award lecture, 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 [Sco77].

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

E' codes a machine ID after $f(n)$ transitions; construct E to force a mistyping in the case of nonacceptance. ∎

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 $f(n)$.[3]

The remainder of the paper gives these details, mixed with some short tutorials on the type systems under study, where we have forgone the formality of inference rules in preference for intuition. 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. In Section 4, we provide a description of the systems $F_3, F_4, \ldots, F_\omega$ generalizing F_2, with emphasis on the significance of *kinds* in these systems. In Section 5 we outline the nonelementary 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 follows the presentation of [PDM89], 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 canonical example: the identity function. In F_2, we write the typed polymorphic identity function[4] 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*[5]. For instance, given a type Int encoding integers, we can represent the identity function *for integers* as:

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

The $\rhd$ indicates β-reduction at the *type* level, where Int is substituted for free occurences of α. Given a type Bool encoding Boolean values, we may similarly write Id [Bool] to get the identity function for booleans. In short, Id is a *polymorphic* function which may be *parameterized* 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:

$$\begin{aligned}
\overline{0} &\equiv \Lambda\alpha{:}*.\lambda\mathbf{f}{:}\alpha \to \alpha.\lambda\mathbf{x}{:}\alpha.\mathbf{x} \\
\overline{1} &\equiv \Lambda\alpha{:}*.\lambda\mathbf{f}{:}\alpha \to \alpha.\lambda\mathbf{x}{:}\alpha.\mathbf{fx} \\
\overline{2} &\equiv \Lambda\alpha{:}*.\lambda\mathbf{f}{:}\alpha \to \alpha.\lambda\mathbf{x}{:}\alpha.\mathbf{f(fx)} \\
&\ldots
\end{aligned}$$

The type of all Church numerals is

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

In the untyped λ-calculus, we realize 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. This reduction can be typed in F_2:

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

Observe that the normal form is also of type Int.[6] Church numerals are merely polymorphic 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.

[3] Observe that these type systems preserve typings under β-reduction.

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

[5] For the moment, we consider the $*$ to be syntactic sugar, though in generalizations of F_2 this will not be so.

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

3 An exponential lower bound on F_2 type inference

3.1 Paradise lost: lessons learned from ML

It has been know 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 [Wan87], 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 $X = Y \rightarrow Z$, such that the solution of the equations (via unification [Rob65, PW78]) 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 in [Mit90][7], is that the following inference rule for `let` preserves exactly the type judgements for closed terms usually derived using the quantification rules:

$$(\texttt{let}) \qquad \frac{\Gamma \rhd M{:}\tau_0 \qquad \Gamma \rhd [M/x]N{:}\tau_1}{\Gamma \rhd \texttt{let } x = M \texttt{ in } N{:}\tau_1}$$

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. 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 formula [KM89]:

Example 3.1

$$\begin{aligned}\Psi \equiv \texttt{let } x_0 = f \texttt{ in}\\ \quad \texttt{let } x_1 = \lambda y.x_0(x_0 y) \texttt{ in}\\ \quad\quad \texttt{let } x_2 = \lambda y.x_1(x_1 y) \texttt{ in}\\ \quad\quad \cdots\\ \quad\quad\quad \texttt{let } x_n = \lambda y.x_{n-1}(x_{n-1} y) \texttt{ in } x_n\end{aligned}$$

The above expression `let`-reduces to $\lambda y.f^{2^n} y$, where the occurence of f is *polymorphic*.

The significance of this polymorphism is exploited in the lower bound of [Mai90, KMM90], where it is shown that recognizing typable ML expressions can be solved in DTIME[2^n], and is DTIME[2^{n^k}]-hard for every integer $k \geq 1$ under logspace reduction.[8] The lower bound is a generic reduction: it proceeds by using types to encode TM IDs, and constructing a λ-term simulating the transition function at the *type* level. The type of $(\Psi\ ID_0)$, where ID_0 is a λ-term whose type encodes the initial ID of the TM, and the transition function is substituted for f, then encodes the state of the machine after 2^{n^k} state transitions. A mistyping is *forced* in the case of a final rejecting state. The circuitry of the TM is effected through taking the simulation of Boolean logic by first-order unification found in [DKM84], and realizing the Boolean gadgets found there by the types of λ-terms.

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 formula types" is made by a straightforward appeal to the simple logic of first-order unification, where in the case of a rejecting computation, it is obvious how to force a mistyping. To further claim that there is no F_2 typing is far from clear, since F_2 types do not admit naive quantifier elimination, hence first-order arguments are too weak. Proving that strongly normalizing terms are not F_2-typable is very difficult: as evidence, we point merely to the tremendous effort of Giannini and Ronchi della Rocca [GR88] in their identifying a single, simple, strongly normalizing term which is not 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 [DKM84] to the classic simulation in the λ-calculus. For example, we type the Boolean values

[7] In this survey, the rule is attributed to Albert Meyer. However, it appears as well in the thesis of Luis Damas [Dam85], and in fact a question about it can be found in the 1985 postgraduate examination in computing at Edinburgh University [Edi88].

[8] An alternate proof is found in [KTU90].

as:

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

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 the "classic" logic, the ML proof can be simplified, with the added bonus that the simulation of TM computation is carried out (as before) on the *type* level, but *mirrored exactly* at the *value* level. For instance, the ML typings of the classical simulations of conjunction and disjunction produce the correct terms as outputs, and also the correct Boolean types as described above. *Values* of expressions are immaterial to the ML type checker, which computes only their *types*; the proof of [Mai90] goes to great lengths to exploit this point.

Restoring the duality between values and types is one of the key ideas in the F_2 bound. A Boolean value A is computed (via β-reduction) which answers the question "Did the TM accept its input?"; this term is used in the expression

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

Observe that if $A \rhd \mathit{false}$, then $\Psi \rhd (\lambda x.xx)(\lambda y.yy)$. By the simple appeal to Girard's strong normalization theorem for F_2 [Gir72, GLT89], we then know that if the TM rejects its input, Ψ is not typable. What remains is to show that if the TM accepts, then Ψ can be typed. We must in this case look more carefully at the structure of the term A.

3.3 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 $\delta\ ID \rhd_\beta ID'$. We call this a simulation *at the value level.* The encoding also yields a very compact and simple proof of the DTIME[2^{n^k}]-hardness bound for recognizing ML-typable terms.

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 map $\partial{:}\, Q \times C \to Q \times C \times D$. 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.

We represent the finite sets Q, C, D, and $\mathsf{Bool} = \{\mathit{true}, \mathit{false}\}$ by projection functions. Given a finite set $E^k = \{e_1^k, \ldots, e_k^k\}$, we code e_i^k by the λ-term

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

A *k-tuple* $\overline{e} \equiv \langle e_1, \ldots, e_k\rangle$ is coded by the λ-term $\lambda z.ze_1\cdots e_k$; note $\overline{e}d_i^k \rhd_\beta e_i$. 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 $\mathsf{nil} \equiv \lambda z.z$. A function $m{:}\, E^k \to F$ with finite domain can then be coded as the tuple

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

so that $\overline{m}d_i^k \rhd_\beta m(e_i)$. When the finite domain of a function is the product of several finite sets (as in ∂), we realize the function in its curried form.

The coding of the λ-term δ is simplified by using a notation for *pattern matching* on tuples. If t is a k-tuple, we write $\langle x_1, \ldots, x_k\rangle = t;\ e$ for $t(\lambda x_1.\cdots\lambda x_k.e)$. For example, **fst** can be defined as $\lambda t.\langle x_1, x_2\rangle = t;\ x_1$. We allow *nesting* of patterns, e.g., $\langle\langle x_1, x_2\rangle, y, \langle z_1, z_2\rangle\rangle = e;\ e'$ means $\langle x, y, z\rangle = e;\ \langle x_1, x_2\rangle = x;\ \langle z_1, z_2\rangle = z;\ e'$.

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. Given all these technicalities, the transition function of M has a very simple encoding:

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

Note that $\langle q', c', d'\rangle$ codes the state, symbol written, and head direction for the next machine configuration, as computed by ∂; 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 equivalent of the "fanout gates" of [Mai90, KMM90] is not necessary.

3.4 Encoding Turing Machines by types

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

Lemma 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 $\delta\, ID \rhd_\beta ID'$, 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 $(\overline{k}\,\delta)\, ID \rhd_\beta ID_k$, and $(\overline{k}\,\delta)\, ID \in \sigma^k$.*

The Lemma states that the computation of M is simulated not only at the *value* level, but also at the *type* level. Observe that the typing of $\overline{k}\ \delta$ is rank 2.

Corollary 3.3 *Let*

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

where there are m occurances of $\overline{2}$, 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. (sketch) Let τ^k be the rank 2 principal type of $\overline{k}\,\delta$. Type the m occurrences of $\overline{2}$, from left to right, as $\tau^{2^{m-1}} \to \tau^{2^m}, \tau^{2^{m-2}} \to \tau^{2^{m-1}}, \ldots, \tau^2 \to \tau^{2^2}, \tau^1 \to \tau^2$. ∎

Theorem 3.4 *Recognizing the lambda terms typable in F_2 is DTIME[2^{n^k}]-hard for any integer $k \geq 1$ under logspace reduction.*

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

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

where $\overline{2}$ occurs n^k 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^k} steps, then A β-reduces to *true*. By Lemma 3.2 and Corollary 3.3, the λ-expression A has principal 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. 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 normalization theorem [Gir72, GLT89], $\Psi_{M,x}$ is not F_2-typable. It is easily seen that $\Psi_{M,x}$ can be constructed in logarithmic space from M and x. ∎

The λ-expression E we use for simulating M on x has a rank 2 typing in F_2 whenever it has an F_2-typing. Since rank 2 Girard/Reynolds typability is equivalent to ML typability [KT89], which is DEXPTIME-complete, this result is the best we can achieve without resorting to higher-rank typings. Note that a slight variation of the representation $\Psi_{M,x}$ gives a compact and simple proof of DEXPTIME-hardness for recognizing ML-typable terms.

Corollary 3.5 *(Fixed type inference) Let τ be an arbitrary but fixed F_2 type. Then the problem of recognizing the lambda terms which can be given the F_2 type τ is also* DTIME[2^{n^k}]*-hard for any integer $k \geq 1$ 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 ML's 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 realized 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

$$(\lambda x_0.(\lambda x_1.\ \cdots\ (\lambda x_{n-1}.(\lambda x_n.x_n)(\lambda y.x_{n-1}(x_{n-1}y)))$$
$$\cdots(\lambda y.x_1(x_1y)))(\lambda y.x_0(x_0y)))\ f,$$

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 this program can be carried out in F_ω to derive a nonelementary lower bound. Related super-exponential bounds can be proven for the F_k so that recognizing typable λ-terms of length n requires $f_k(n)$ time, where $f_k(n)$ is an "exponential" stack of 2s growing linearly with k, and 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 $\to$ 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.

Let $*$ denote the *kind* of types that may be generated in F_2: we could describe the functionality of (a curried) $\rightarrow$ as $\rightarrow \in * \Rightarrow * \Rightarrow *$, where $\Rightarrow$ is a higher-order version of $\rightarrow$ describing functions from types to types. If we now introduce λ-abstraction at the *type* level, imitating its existence at the expression level, we can describe other functions on types, for example:

$$\begin{aligned} \mathsf{I} &\equiv \lambda\alpha{:}*.\alpha \rightarrow \alpha \in * \Rightarrow * \\ \mathsf{K} &\equiv \lambda\alpha{:}*.\lambda\beta{:}*.\alpha \in * \Rightarrow * \Rightarrow * \end{aligned}$$

This expressiveness adds considerable power to the type language. Its practical need is apparent in trying to give a meaning to List when parameterizing the identity function as *Id* [List Int]—we want List to be a higher-order type (i.e., one of kind $* \Rightarrow *$) which maps a type (Int) to a type (List-Int). A logical example of the need for higher-order types is found in the intuitionistic "program" for the disjunction of propositions (types) p and q: $p \vee q \equiv \Delta r{:}*.(p \rightarrow r) \rightarrow (q \rightarrow r) \rightarrow r$. (We consider $p \vee q$ to be of kind $*$—it is also a proposition.) By abstracting over p and q, in a higher-order system we may write: $\vee \equiv \lambda p{:}*.\lambda q{:}*.\Delta r{:}*.(p \rightarrow r) \rightarrow (q \rightarrow r) \rightarrow r$. To use such definitions, α-renaming and β-reduction are introduced at the type level.

The type systems F_k differ in the degree to which they allow 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 $\mathcal{K}_k$ allowed in F_k by a grammar:

$$\begin{aligned} \mathcal{K}_2 &::= * \\ \mathcal{K}_{\ell+1} &::= \mathcal{K}_\ell \mid \mathcal{K}_\ell \Rightarrow \mathcal{K}_{\ell+1} \\ \mathcal{K}_\omega &::= * \mid \mathcal{K}_\omega \Rightarrow \mathcal{K}_\omega \end{aligned}$$

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

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

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 provides a general technique for composing functions. Observe that

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

where the function Φ is defined as $\Phi(0) = 1$, $\Phi(t+1) = 2^{\Phi(t)}$. 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:

$$\begin{aligned} \delta \in\ & \Delta v_1{:}*.\Delta v_2{:}*.\cdots\Delta v_r{:}*. \\ & \overline{\mathcal{L}}(v_1, v_2, \ldots, v_r) \rightarrow \overline{\mathcal{R}}(v_1, v_2, \ldots, v_r). \end{aligned}$$

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

Proof. They both encode TM IDs, and so are unifiable. The "circuitry" of the unification logic exists on the "$\overline{\mathcal{L}}$" side, which induces structure on the "$\overline{\mathcal{R}}$" side. ∎

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 $\overline{\mathcal{R}}(v_1, v_2, \ldots, v_r)$, and the *intermediate variables* V_I form the complement. We can then define an *ID-constructor* *make-ID* as a function on types:

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

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

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

$$\begin{aligned} \delta \in\ & \Delta v_1{:}*.\Delta v_2{:}*.\cdots\Delta v_r{:}*. \\ & (\textit{make-ID}\ (\Gamma_1\, v_1 v_2 \cdots v_r) \\ & \qquad (\Gamma_2\, v_1 v_2 \cdots v_r) \\ & \qquad \cdots \\ & \qquad (\Gamma_q\, v_1 v_2 \cdots v_r)) \\ & \rightarrow \textit{make-ID}\ v_{p+1} v_{p+2} \cdots v_{p+q}. \end{aligned}$$

Proof. By first-order unification and Proposition 5.1. We remark that the functions Γ_i encode what we have called "TM circuitry." ∎

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 realized 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*

$$\begin{array}{l}\Delta v_1{:}*.\Delta v_2{:}*.\cdots\Delta v_p{:}*.\Delta v'_1{:}*.\Delta v'_2{:}*.\cdots\Delta v'_r{:}*.\\ \quad (\textit{make-ID}\\ \qquad (\Gamma_1\, v_1 v_2\cdots v_p\,(\Gamma_1\, v'_1\cdots v'_r)\cdots(\Gamma_q\, v'_1\cdots v'_r))\\ \qquad (\Gamma_2\, v_1 v_2\cdots v_p\,(\Gamma_1\, v'_1\cdots v'_r)\cdots(\Gamma_q\, v'_1\cdots v'_r))\\ \qquad \cdots\\ \qquad (\Gamma_q\, v_1 v_2\cdots v_p\,(\Gamma_1\, v'_1\cdots v'_r)\cdots(\Gamma_q\, v'_1\cdots v'_r)))\\ \quad \rightarrow\ \textit{make-ID}\ v'_{p+1}v'_{p+2}\cdots v'_{p+q}.\end{array}$$

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.

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 basic idea is the following: we construct a certain λ-term $\mathcal{T}$ at the type level which *represents* the type of the j-fold composition of δ, where the kind κ of $\mathcal{T}$ does not depend on j. We then define a function $map{:}\,\kappa\Rightarrow\kappa$ such that $map\,\mathcal{T}$ 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.* For instance, we can define

$$\begin{array}{rcl}\overline{2_1} & \equiv & \lambda\sigma{:}\,\kappa\Rightarrow\kappa.\lambda\tau{:}\,\kappa.\sigma^2\tau\\ & & \quad\in\ (\kappa\Rightarrow\kappa)\Rightarrow\kappa\Rightarrow\kappa\\ \overline{2_2} & \equiv & \lambda\sigma{:}\,(\kappa\Rightarrow\kappa)\Rightarrow\kappa\Rightarrow\kappa.\lambda\tau{:}\,\kappa\Rightarrow\kappa.\sigma^2\tau\\ & & \quad\in\ ((\kappa\Rightarrow\kappa)\Rightarrow\kappa\Rightarrow\kappa)\Rightarrow\\ & & \qquad\qquad (\kappa\Rightarrow\kappa)\Rightarrow\kappa\Rightarrow\kappa\end{array}$$

and write

$$\overline{2_2 2_1}map \rhd \lambda\mathcal{T}{:}\,\kappa.map(map(map(map\ \mathcal{T}))).$$

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

$$\begin{array}{l}\lambda x_1{:}*.\lambda x_2{:}*.\cdots\lambda x_q{:}*.\lambda\Phi{:}*^{q+pt+1}.\\ \qquad\qquad \Phi\ \phi_1\phi_2\cdots\phi_q d_1\cdots d_{pt}.\end{array}$$

The $d\equiv d_i$ are just "dummy" type variables to "pad" the kind, and the ϕ_i are types involving some set $v_1,\ldots,v_{pt}$ of type variables, $x_1,\ldots,x_r$, 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 $\mathcal{L}$, given type variables for the x_i.

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

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

where the f_i are the output variables of the dag ultimately to be constructed, and the v_i are a list of variables to be used during the construction. The λx_i-bindings are padding.

A *pair* is a λ-term of the form

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

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, as is projection of types of kind $*$ in the pair.

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* is tedious and is postponed to the final version of the paper; it involves straightforward list processing on pairs, where the type variables in the variable list are repeatedly *shifted cyclically* and retrieved as the "floors" are built. We also define a λ-term $\mathcal{I}$ which takes a "final" pair and produces a first-order type of the t-fold composition of δ.

5.2 Composing *map*

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

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

(at the *expression* level) to compose *map* $\Phi(n{+}2)$ times. The dag gets constructed at a "speed" controlled by the reduction sequence of C to normal form. Suppressing kinds for readability, we recursively define a set of types

used to type C:

$$\begin{aligned}
\mathcal{G}_0\{\alpha_0\} &\equiv \Delta map.(\Delta\tau.\mathcal{I}(map\ \tau) \to \mathcal{I}\tau) \to \\
&\quad (\Delta\tau.\mathcal{I}(\alpha_0 map\ \tau) \to \mathcal{I}\tau) \\
\mathcal{G}_1\{\alpha_1\} &\equiv \Delta\alpha_0.\mathcal{G}_0\{\alpha_0\} \to \mathcal{G}_0\{\alpha_1\alpha_0\} \\
&\equiv \Delta\alpha_0.\mathcal{G}_0\{\alpha_0\} \to \\
&\quad \Delta map. \\
&\quad (\Delta\tau.\mathcal{I}(map\ \tau) \to \mathcal{I}\tau) \to \\
&\quad (\Delta\tau.\mathcal{I}(\alpha_1\alpha_0 map\ \tau) \to \mathcal{I}\tau) \\
\mathcal{G}_{k+1}\{\alpha_{k+1}\} &\equiv \Delta\alpha_k.\mathcal{G}_k\{\alpha_k\} \to \mathcal{G}_k\{\alpha_{k+1}\alpha_k\}
\end{aligned}$$

Lemma 5.4 *For each $0 \le i \le n$, $\lambda g_i.\lambda y_i.g_i^2 y_i$ can be typed as $\mathcal{G}_i\{\overline{2}\}$, where $\overline{2} \equiv \lambda\sigma.\lambda\tau.\sigma^2\tau$ is a type having the same kind as α_i.*

Proof. For $\lambda g_0.\lambda y_0.g_0^2 y_0$, we have the typing

$\Lambda map.$
 $\lambda \mathbf{g}{:}\Delta\tau.\mathcal{I}(map\ \tau) \to \mathcal{I}\tau.$
  $\Lambda\tau.$
   $\lambda \mathbf{y}{:}\mathcal{I}(map(map\ \tau)) \equiv \mathcal{I}(\overline{2}\, map\, \tau).$
    $\mathbf{g}\ [\tau]\ (\mathbf{g}\ [map\ \tau]\ \mathbf{y})$

and for $\lambda g_{i+1}.\lambda y_{i+1}.g_{i+1}^2 y_{i+1}$, $i \ge 0$, we have the typing

$\Lambda\alpha_i.$
 $\lambda \mathbf{g}{:}\mathcal{G}_i\{\alpha_i\} \equiv \Delta\alpha_{i-1}.\mathcal{G}_{i-1}\{\alpha_{i-1}\} \to \mathcal{G}_{i-1}\{\alpha_i\alpha_{i-1}\}.$
  $\Lambda\alpha_{i-1}.$
   $\lambda \mathbf{y}{:}\mathcal{G}_{i-1}\{\alpha_{i-1}\}.$
    $\mathbf{g}\ [\alpha_i\alpha_{i-1}]\ (\mathbf{g}\ [\alpha_{i-1}]\ \mathbf{y})$

■

Lemma 5.5 *In the term C,*

$$\lambda f.\lambda x.f^2 x \ \in\ \mathcal{G}_n\{\overline{2}\} \to \mathcal{G}_{n-1}\{\overline{2}\} \to \mathcal{G}_{n-1}\{\overline{222}\}.$$

Theorem 5.6 *The term C (the "crank") has typing*

$$\begin{aligned}
C \equiv\ & (\lambda \mathbf{f}.\lambda \mathbf{x}.\mathbf{f^2 x})(\lambda \mathbf{g_n}.\lambda \mathbf{y_n}.\mathbf{g_n^2 y_n}) \\
& (\lambda \mathbf{g_{n-1}}.\lambda \mathbf{y_{n-1}}.\mathbf{g_{n-1}^2 y_{n-1}})[\overline{2}] \\
& (\lambda \mathbf{g_{n-2}}.\lambda \mathbf{y_{n-2}}.\mathbf{g_{n-2}^2 y_{n-2}})[\overline{2}] \\
& (\lambda \mathbf{g_{n-3}}.\lambda \mathbf{y_{n-3}}.\mathbf{g_{n-3}^2 y_{n-3}})[\overline{2}] \\
& \cdots \\
& (\lambda \mathbf{g_0}.\lambda \mathbf{y_0}.\mathbf{g_0^2 y_0}),
\end{aligned}$$

so that

$$\begin{aligned}
C \ &\in\ \mathcal{G}_0\{\overline{22}\cdots\overline{2}\} \qquad (n+2\ 2\text{s}) \\
&\equiv\ \Delta map.(\Delta\tau.\mathcal{I}(map\ \tau) \to \mathcal{I}\tau) \to \\
&\quad (\Delta\tau.\mathcal{I}((\overline{22}\cdots\overline{2})map\ \tau) \to \mathcal{I}\tau) \\
&\rhd\ \Delta map.(\Delta\tau.\mathcal{I}(map\ \tau) \to \mathcal{I}\tau) \to \\
&\quad (\Delta\tau.\mathcal{I}(map^{\Phi(n+2)}\ \tau) \to \mathcal{I}\tau)
\end{aligned}$$

We now briefly sketch how this typing for C can be used to type $C\ \delta\ ID_0$. We take the type and parameterize it with the definition of *map*, and then show that $\delta \in (\Delta\tau.\mathcal{I}(map\,\tau) \to \mathcal{I}\tau)$. Next, parameterize this term over an *initial pair* τ_0, abstracting over all the variables v_j appearing in the pair. By then parameterizing the v_j with (huge) first-order types μ_j, we simulate the unification process of ML, "matching" ID_0 with a suitable parameterization π. We then have:

$$\begin{aligned}
\mathcal{M} \ \equiv\ & (\Lambda d{:}*.\Lambda v_1{:}*.\cdots\Lambda v_{q+pt}{:}*.\, C[map]\delta[\tau_0]) \\
& [\mu_d][\mu_1]\cdots[\mu_{q+pt}]\ (ID_0[\pi])
\end{aligned}$$

The type of $\mathcal{M}$ codes the state of the TM after $\Phi(n+2)$ state transitions; we extract the accepting state A (known by assumption to be coded *true*), and type the term $(\lambda x.xx)(A(\lambda x.x)(\lambda y.yy))$ (details to be given in the full paper). We then have our nonelementary bound:

Theorem 5.7 *Recognizing the lambda terms of length n typable in F_ω is* DTIME$[\Phi(n^k)]$*-hard for any integer $k \ge 1$ under logspace reduction, where*

$$\begin{aligned}
\Phi(0) &= 1, \\
\Phi(t+1) &= 2^{\Phi(t)}.
\end{aligned}$$

Corollary 5.8 *Recognizing the lambda terms of length n typable in F_k is* DTIME$[f_{k-4}(n)]$*-hard under logspace reduction, where*

$$\begin{aligned}
f_0(n) &= n, \\
f_{k+1}(n) &= 2^{f_k(n)}.
\end{aligned}$$

We remark only that the "-4" reflects the kind overhead of building pairs.

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-normalizing 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 center stage: in particular, we would like to know if the techniques 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 no "known" difficult 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 generalizes nicely, as expressed in the slogan, "how fast can the crank (of the transition function) be turned?": better lower bounds can be proven by analyzing 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 normalization theorem, as *encodings of reduction sequences.*

We should observe as well the pitfalls of the method, or at least the hurdles which wait to be surmounted. The "cranks" described are all strongly normalizing in a manner such that 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 normalization theorem, and the representation theorem (that the representable integer functions are those provably total in second order Peano Arithmetic) [Gir72, GLT89]. We have some idea how to get around the first hurdle, but are done in by the second. Does it seem possible that the representation theorem would allow reduction sequences of functionally unbounded length on typable terms?

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 foundation of the simulation is changed substantially. The explanation of this limitation is that the *type* language of F_ω is fundamentally 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 Schwictenberg [Sch82] 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 in [Hen90]. For their encouragement, suggestions and criticisms, we thank Paris Kanellakis, Georg Kreisel, Daniel Leivant, Angus Macintyre, Jon Riecke, and Rick Statman. 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.

References

[Car89] L. Cardelli. *Typeful programming.* Lecture Notes for the IFIP Advanced Seminar on Formal Methods in Programming Language Semantics, Rio de Janeiro, Brazil, 1989. See also SRC Report 45, Digital Equipment Corporation.

[Dam85] L. Damas. *Type assignment in programming languages.* Ph. D. dissertation, CST-33-85, Computer Science Department, Edinburgh University, 1985.

[DM82] L. Damas and R. Milner. *Principal type schemes for functional programs.* In **9-th ACM Symposium on Principles of Programming Languages**, pp. 207–212, January 1982.

[DKM84] C. Dwork, P. Kanellakis, and J. C. Mitchell. *On the sequential nature of unification.* **Journal of Logic Programming** 1:35–50, 1984.

[Edi88] Edinburgh University. Postgraduate Examination Questions in Computation Theory, 1978–1988. Laboratory for Foundations of Computer Science, Report ECS-LFCS-88-64.

[GR88] P. Giannini and S. Ronchi Della Rocca. *Characterization of typings in polymorphic type discipline.* In **Proceedings of the 3-rd IEEE Symposium on Logic in Computer Science**, pp. 61–70, July 1988.

[Gir72] J.-Y. Girard. *Interprétation Fonctionelle et Elimination des Coupures de l'Arithmetique d'Ordre Superieur.* Thèse de Doctorat d'Etat, Université de Paris VII, 1972.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. **Proofs and Types.** Cambridge University Press, 1989.

[HMT90] R. Harper, R. Milner, M. Tofte. **The Definition of Standard ML.** MIT Press, 1990.

[HS65] J. Hartmanis and R. E. Stearns. *On the computational complexity of algorithms.* **Transactions of the American Mathematical Society** 117, pp. 285–306.

[Hen90] F. Henglein. *A lower bound for full polymorphic type inference: Girard/ Reynolds typability is DEXPTIME-hard.* University of Utrecht, Technical Report RUU-CS-90-14, April 1990.

[Hin69] R. Hindley. *The principal type scheme of an object in combinatory logic.* **Transactions of the American Mathematical Society** 146:29–60, 1969.

[HU79] J. E. Hopcroft and J. D. Ullman. **Introduction to Automata Theory, Languages, and Computation.** Addison Wesley, 1979.

[HW88] P. Hudak and P. L. Wadler, editors. *Report on the functional programming language Haskell.* Yale University Technical Report YALEU/DCS/RR656, 1988.

[KM89] P. C. Kanellakis and J. C. Mitchell. *Polymorphic unification and ML typing.* Brown University Technical Report CS-89-40, August 1989. Also in **Proceedings of the 16-th ACM Symposium on the Principles of Programming Languages**, pp. 105–115, January 1989.

[KMM90] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. *Unification and ML type reconstruction.* In **Computational Logic: Essays in Honor of Alan Robinson**, ed. J.-L. Lassez and G. Plotkin. MIT Press, 1990.

[KTU90] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. *ML typability is DEXPTIME-complete.* **Proceedings of the 15-th Colloquium on Trees in Algebra and Programming**, May 1990. (See also Boston University Technical Report, October 1989).

[KT89] A. J. Kfoury and J. Tiuryn. *Type reconstruction in finite rank fragments of the second-order lambda calculus.* Technical Report BUCS 89-011, Boston University, October 1989. Also in **Proceedings of the 5-th IEEE Symposium on Logic in Computer Science**, pp. 2–11, June 1990.

[Lan66] P. Landin. *The next 700 programming languages.* **Communications of the ACM** 9(3): 157–166.

[Mai90] H. G. Mairson. *Deciding ML typability is complete for deterministic exponential time.* In **Proceedings of the 17-th ACM Symposium on the Principles of Programming Languages**, pp. 382–401, January 1990.

[Mil78] R. Milner. *A theory of type polymorphism in programming.* **Journal of Computer and System Sciences** 17, pp. 348–375, 1978.

[Mit90] J. C. Mitchell. *Type systems for programming languages.* To appear as a chapter in the **Handbook of Theoretical Computer Science**, van Leeuwen et al., eds. North-Holland, 1990.

[PW78] M. S. Paterson and M. N. Wegman. *Linear unification.* **Journal of Computer and System Sciences** 16, pp. 158–167, 1978.

[PDM89] B. Pierce, S. Dietzen, and S. Michaylov. *Programming in higher-order typed lambda calculi.* Technical Report CMU-CS-89-111, Carnegie Mellon University, March 1989.

[PL89] F. Pfenning and P. Lee. *LEAP: a language with eval and polymorphism.* **TAPSOFT 1989: Proceedings of the International Joint Conference on Theory and Practice in Software Development**, Barcelona, Spain. See also CMU Ergo Report 88-065.

[Rey74] J. C. Reynolds. *Towards a theory of type structure.* In **Proceedings of the Paris Colloquium on Programming**, Lecture Notes in Computer Science 19, Springer Verlag, pp. 408–425, 1974.

[Rob65] J. A. Robinson. *A machine oriented logic based on the resolution principle.* **Journal of the ACM** 12(1):23–41, 1965.

[Sch82] H. Schwictenberg. *Complexity of normalization in the pure typed lambda calculus.* **The L. E. J. Brouwer Centenary Symposium**, A. S. Troelstra and D. van Daalen (eds.), pp. 453–457. North-Holland, 1982.

[Sco77] D. Scott. *Logic and programming languages.* **Communications of the ACM** 20(9):634–641, 1977.

[Str73] C. Strachey. *The varieties of programming language.* Technical Monograph PRG-10, Programming Research Group, Oxford University, 1973.

[Tur85] D. A. Turner. *Miranda: A non-strict functional language with polymorphic types.* In **IFIP International Conference on Functional Programming and Computer Architecture**, Nancy, Lecture Notes in Computer Science 201, pp. 1–16, Springer-Verlag, 1985.

[Wan87] M. Wand. *A simple algorithm and proof for type inference.* **Fundamenta Informaticae** 10 (1987).