A Lower Bound for Full Polymorphic Type Inference: Girard-Reynolds Typability is DEXPTIME-hard

Fritz Henglein

RUU-CS-90-14
April 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

A Lower Bound for Full Polymorphic Type Inference: Girard-Reynolds Typability is DEXPTIME-hard

Fritz Henglein

RUU-CS-90-14
April 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

1 Introduction

Kanellakis and Mitchell have shown that ML typing is decidable in deterministic exponential time [KM89]. They also gave a reduction of the Quantified Boolean Value problem to ML typing thus giving a lower bound of PSPACE-hardness. Mairson recently improved this result by presenting a generic simulation of deterministic Turing Machines for up to an exponential number of steps, thus proving DEXPTIME-hardness of ML typing. An alternative proof based on characterization by "acyclic" semi-unification was given by Kfoury, Tiuryn and Urzyczyn [KTU89].

Mairson's lower bound can be seen as a generic simulation of deterministic Turing Machines (for up to an exponential number of steps) "in the types" of λ-expressions. By this we mean that the Turing Machine is simulated by the type inference algorithm as it type-checks a λ-expression representing the Turing Machine and its input. Critical use is made of

- a universal "type equality", represented by a λ-expression (see below) that forces, in accordance with the typing rules for ML, the types of some variables to be equal;
- an encoding of the Boolean values **true** and **false** by equality or (possible) inequality of certain types;
- a mechanism ("fan-out gates") for producing several terms with the same Boolean input type from a single Boolean input type to "program around" the side effects of type equality; and
- an encoding of an exponential number of moves by a Turing Machine by a nested **let**-expression of polynomial size, exploiting the polymorphic typing rule of ML only at this point.

The use of universal type equality is at the basis of the simulation as it is used in the definition of the Boolean values, of the fan-out gates, and indirectly also in the remaining steps. We shall, in some sense, simplify Mairson's proof and give a simulation in which another, more "semantic" and more conventional representation of Boolean values is used without resorting to a universal type equality relation. The type equality relation is a consequence of ML's typing rule for λ-bound variables since all occurrences of a λ-bound variable are required to have the same monotype. Since the implicitly typed Girard-Reynolds Calculus makes no such provision, Mairson's technique does not directly extend to it.

The typability problem for the Girard-Reynolds Calculus has been very elusive since its inception while attracting a considerable amount of attention. As far as we know no nontrivial lower or upper bounds have been proved. In this paper we present the first nontrivial lower bound for Girard-Reynolds typability. We show that it is DEXPTIME-hard by adapting a new proof of DEXPTIME-hardness for ML typability, which builds on Mairson [Mai90]. The amazing simplicity of our lower-bound technique for GR-typability derives from the fact that with little change we can adapt a general purpose simulation of Turing Machines in the (untyped) λ-calculus to the typing disciplines imposed by ML and the Girard-Reynolds Calculus.

The outline of the rest of this paper is as follows. In section 2 we present ML with tuples and its typing rules. In section 3 elementary encodings of finite domains are given. Section 4 shows how Mairson's fan-out gates are encoded in our representations. In section 5 we give our version of how exponential-time deterministic Turing Machine computations can be encoded as an ML typability problem. Section 6 introduces the implicitly typed Girard-Reynolds Calculus, and section 7 presents the DEXPTIME-hardness result for Girard-Reynolds typability. Finally, section 8 contains a short summary and some concluding remarks.

2 ML with Tuples

We quickly review ML typing. For technical reasons we add tuples and a constant ω that has any type into our language. The ML typing rules (Milner Calculus) are shown in Table 1. For a review of ML typing the reader is referred to, e.g., [DM82].

3 Elementary Encodings

3.1 Finite Domains

We show that a conventional representation of finite domains in the λ-calculus can be "lifted" to the universe of types. This representation does not rely on the equality of certain type expressions, yet permits an encoding of equality for specific domains, instead of a "universal" encoding.

For the representation of Turing Machines it is necessary to represent and compute with finite domains of values; e.g., states, tape symbols, tape head directions. For this reason we present a general representation of finite domains with a *select* (or *case*) expression. As a special case we get a representation for the Booleans.

Let $D^k = \{d_1, \ldots, d_k\}$ be a given finite domain (set). We represent every element d_i by the i-th k-ary projection function, p_i^k.

$$\begin{aligned} d_1 &= \lambda x_1 \ldots x_k . x_1 \\ &\ldots \\ d_i &= \lambda x_1 \ldots x_k . x_i \\ &\ldots \\ d_k &= \lambda x_1 \ldots x_k . x_k \end{aligned}$$

Using this representation it is simple to give an encoding of a case expression.

$$\begin{aligned} \textbf{case } d \textbf{ of } &= d e_1 \ldots e_k \\ d_1 &: e_1 \\ &\ldots \\ d_k &: e_k \end{aligned}$$

Let A range over type environments; x over variables; e, e' over λ-expressions; α over type variables; $\tau, \tau', \tau_1, \ldots, \tau_k$ over monotypes; σ, σ' over polytypes. The following are the type inference axiom and rule schemes for ML typing with tuples.

Name	Axiom/rule
(CONS)	$A \supset \omega : \sigma$
(TAUT)	$A\{x : \sigma\} \supset x : \sigma$
(GEN)	$\dfrac{A \supset e : \sigma \qquad (\alpha \text{ not free in } A)}{A \supset e : \forall\alpha.\sigma}$
(INST)	$\dfrac{A \supset e : \forall\alpha.\sigma}{A \supset e : \sigma[\tau/\alpha]}$
(ABS)	$\dfrac{A\{x : \tau'\} \supset e : \tau}{A \supset \lambda x.e : \tau' \rightarrow \tau}$
(APPL)	$\dfrac{A \supset e : \tau' \rightarrow \tau \quad A \supset e' : \tau'}{A \supset (ee') : \tau}$
(LET)	$\dfrac{A \supset e : \sigma \quad A\{x : \sigma\} \supset e' : \sigma'}{A \supset \textbf{let } x = e \textbf{ in } e' : \sigma'}$
(TUPLE)	$\dfrac{A \supset e_1 : \tau_1 \quad \ldots \quad A \supset e_k : \tau_k}{A \supset (e_1, \ldots, e_k) : (\tau_1, \ldots, \tau_k)}$
(SEL)	$\dfrac{A \supset e : (\tau_1, \ldots, \tau_k) \qquad (1 \leq i \leq k)}{A \supset e.i^{(k)} : \tau_i}$

Table 1: Type inference axioms and rules for ML typing with tuples (Milner Calculus)

We shall denote the representations of the elements of $D^2 = \{d_1, d_2\}$ by **true** and **false**, respectively, and instead of

$$\begin{array}{lll} \textbf{case} & d & \textbf{of} \\ & d_1: & e_1 \\ & d_2: & e_2 \end{array}$$

we will write

$$\textbf{cond}\ d\ e_1 e_2$$

Another possible representation of elements of finite domains is as characteristic functions over D; that is, as tuples of length k where, for d_i, the i-th component is **true** and all the other components are **false**.

This is the conventional representation of the Boolean values in the untyped λ-calculus. The term **cond** represents the conditional if-then-else construct. In the Milner Calculus, the type system for the functional core of ML, these combinators have the following principal types.

$$\begin{array}{rcl} \textbf{true} & : & \forall\alpha\beta.\alpha \to \beta \to \alpha \\ \textbf{false} & : & \forall\alpha\beta.\alpha \to \beta \to \beta \\ \textbf{cond} & : & \forall\alpha\beta\gamma.(\alpha \to \beta \to \gamma) \to \alpha \to \beta \to \gamma \end{array}$$

Note that the computation of the types of applied expressions with the conditional and the Boolean values mirror the computation of the values. We can think of **cond** as taking three *type expressions* as input and producing a *type expression* as output. Consider, for example, the simple expressions below and their types, as output by the Standard ML of New Jersey compiler [AM88].

```
- cond tt 5 7.0;
val it = 5 : int

- cond ff 5 7.0;
val it = 7.0 : real
```

We can see that **cond** maps a pair of types **integer** and **real** to **integer** when given (the type of) **true** as its first input and to **real** when given (the type of) **false** as first input. Note, however, that the type of the first argument to **cond** is "side-effected". That is, whereas **false** has the type $\forall\alpha\beta.\alpha \to \beta \to \beta$, in the context **cond false** 5 7.0 it has the type **integer** $\to$ **real** $\to$ **real**. As a consequence we cannot generally use a λ-bound variable more than once if one of its occurrences is in the first argument position of a **cond**.

Our representation is different than the one chosen in Mairson [Mai90]. There, **false** is represented by a λ-expression of type $\forall\alpha.\forall\beta.\forall\gamma.\alpha \to \beta \to \gamma \to \gamma$, and **true** by a term of type $\forall\alpha.\forall\gamma.\alpha \to \alpha \to \gamma \to \gamma$. The underlying idea in Mairson's representation is that a term of type $\sigma \to \tau \to \upsilon \to \upsilon$ is considered a representation of **true** only if σ and τ are identical, and a representation of **false** only if σ and τ are completely independent.

3.2 Assignments

Single *assignments* or *definitions* can be modeled by a beta redex. Again, this is a representation that follows the "value level". That is, we define

$$x := e_1; e_2 \;=\; (\lambda x.e_2)e_1$$

The effect "in the types" is that if e_1 has type τ_1 and $\lambda x.e_2$ has the type $\tau_1 \to \tau_2$ then the whole expression, $(\lambda x.e_2)e_1$, has type τ_2. In ML a beta redex can also be written as a **let**-expression, **let** $x = e_1$ **in** e_2. Since **let**-expressions have a different typing rule than the corresponding the pure beta redexes, however, we shall use the notation above, $x := e_1; e_2$, instead.

We shall use the "pattern matching" notation

$$\begin{array}{rcl}(x_1, \ldots, x_k) & := & e_1; \\ & & e_2\end{array}$$

for the sequence of assignments

$$\begin{array}{rcl} t & := & e_1; \\ x_1 & := & t.1^{(k)}; \\ & \cdots & \\ x_k & := & t.k^{(k)}; \\ e_2 & & \end{array}$$

4 Preventing Interference

One of the main achievements of Mairson's lower bound proof is the way in which type information is "replicated" in such a fashion that encoding a computation in one copy of type information does not affect the type information of another copy. In particular, using different copies of the same type information, multiple occurrences of values in the first argument position of **cond** and **select** can be emulated without the side-effect on the type of one copy affecting another.

This mechanism is necessary to harness the universal side effect of type equality forced by ML's typing rule for λ-bound variables. We develop, analogously, a way of "copying" type information within our representations.

4.1 Copying Boolean Values

The Booleans **true** and **false** are represented by (the types of) the terms $\lambda xy.x$ and $\lambda xy.y$. We present an expression that, whenever given an expression of either type $\forall\alpha\beta.\alpha \to \beta \to \alpha$ or $\forall\alpha\beta.\alpha \to \beta \to \beta$, returns a *pair*, p, such that both $p.1$ and $p.2$ have the same type as the given input. Given such an expression $\mathbf{copy}_2$, corresponding to Mairson's "fan-out gate", we can devise expressions for producing more than two copies of a type and for producing copies of our representations of finite domains.

At this point we use the constant ω with type $\forall\alpha.\alpha$.

$$\begin{aligned}\mathbf{copy}_2 = \lambda d.\\ &(\lambda x_1 x_2.(d(x_1,\omega)(x_2,\omega)).1^{(2)},\\ &\lambda y_1 y_2.(d(\omega,y_1)(\omega,y_2)).2^{(2)})\end{aligned}$$

The principal type of this combinator is

$$\begin{aligned}&\mathbf{copy}_2 : \forall\alpha_1\alpha_2\beta_1\beta_2\gamma_1\gamma_2.\\ &\quad((\alpha_1,\alpha_2)\to(\beta_1,\beta_2)\to(\gamma_1,\gamma_2))\to(\alpha_1\to\beta_1\to\gamma_1,\alpha_2\to\beta_2\to\gamma_2)\end{aligned}$$

This construction can, of course, be generalized to generate any fixed number of "copies" of independent types. We shall write $\mathbf{copy}_l$ for the λ-expression that generates l copies of a Boolean representation.

Note that, on the untyped level, the effect of $\mathbf{copy}_2$ is completely neutral: $\mathbf{copy}_2(\mathbf{true})$ β-reduces[1] to $(\mathbf{true},\mathbf{true})$; and $\mathbf{copy}_2(\mathbf{false})$ β-reduces to $(\mathbf{false},\mathbf{false})$.

4.2 Copying Elements of Finite Domains

The construction for copying Boolean values can be extended in a straightforward fashion to elements of finite domains of size k. In this case the corresponding copy function is denoted by $\mathbf{copy}_2^k$

$$\begin{aligned}\mathbf{copy}_2^k = \lambda d.\\ &(\lambda x_1\ldots x_k.(d(x_1,\omega)\ldots(x_k,\omega)).1^{(2)},\\ &\lambda y_1\ldots y_k.(d(\omega,y_1)\ldots(\omega,y_k)).2^{(2)})\end{aligned}$$

and its principal type is

$$\begin{aligned}&\mathbf{copy}_2^k : \forall\alpha_1\alpha_2\beta_1\beta_2\gamma_1\gamma_2.\\ &\quad((\alpha_1,\beta_1)\to\ldots\to(\alpha_k,\beta_k)\to(\gamma,\delta))\to(\alpha_1\to\ldots\to\alpha_k\to\gamma,\beta_1\to\ldots\to\beta_k\to\delta)\end{aligned}$$

Again, this construction can be generalized to l copies of k-element domains. We shall denote the corresponding representation $\mathbf{copy}_l^k$.

5 Representing Turing Machines

A Turing Machine (TM) consists of the following components (see [HU79]).

[1] We call tuple redex reduction $(e_1,\ldots,e_k).i^{(k)} \Rightarrow e_i$ also β-reduction for convenience' sake.

Q = $\{q_1, \ldots, q_k\}$: a finite set
of states, with initial state q_1,
and accepting states $F \subset Q$

C = $\{c_1, \ldots, c_l\}$: a finite set of
tape symbols, with blank symbol c_1

D = $\{l, r\}$ or $\{l, 0, r\}$:
two or three "directions" in which the tape head may move
(left, not at all, right)

d : $Q \times C \to Q \times C \times D$:
a partial transition function.

A *configuration (instantaneous description)* is defined as a triple, (l, q, r) where $l, r \in C^*, q \in Q$. The next-move function is induced by the transition function as follows[2]:

```
Conf = {(l,q,r) : l,r ∈ C*, q ∈ Q}
move: Conf → Conf (partial)
move (l,q,r) =
    if r = ε then:
        return (l,q,c1);
    (q',c',d') := d(q,r(1));
    if d' = l then:
        if l = ε then:
            return (l,q_rej,r);
        else:
            return (l(2..),q',l(1)c'r(2..));
    if d' = r then:
        return (lc',q',r(2..));
    if d' = 0 then:
        return (l,q',c'r(2..));
```

5.1 The transition function

We now show how the transition function d is represented. For every combination of state q_i and tape symbol c_j the transition function d returns a triple d_{ij} consisting of the new state, the character to be written and the direction in which the tape head is to move. We assume, w.l.o.g., that d is total, but it is also possible to emulate the Turing Machine with a partial transition function directly.

[2] If the notation doesn't make sense, don't worry. The function move is simply the transition relation $\vdash_M$ from [HU79].

$$
\begin{array}{l}
\mathrm{d} = \lambda qc. \\
\quad (c^1, \ldots, c^k) := \mathbf{copy}_k^l c; \\
\quad \mathbf{case}\ q\ \mathbf{of} \\
\quad q_1 : \mathbf{case}\ c^1\ \mathbf{of} \\
\qquad c_1 : d_{11}; \\
\qquad \ldots \\
\qquad c_l : d_{1l}; \\
\quad q_2 : \mathbf{case}\ c^2\ \mathbf{of} \\
\qquad c_1 : d_{21}; \\
\qquad \ldots \\
\qquad c_l : d_{2l}; \\
\quad \ldots \\
\quad q_k : \mathbf{case}\ c^k\ \mathbf{of} \\
\qquad c_1 : d_{k1}; \\
\qquad \ldots \\
\qquad c_l : d_{kl};
\end{array}
$$

Note that in the above definition both arguments of d are side-effected since they occur in the first positions of case expressions.

5.2 The next-move function

As we have seen, the transition function induces a next-move function on configurations. We can almost transliterate this description into a representation "in the types" *and* "in the values". The generally unbounded tape contents are represented by nested pairs. W.l.o.g. we assume that tape symbol c_1 represents the blank character and c_l a special "end-of-tape" symbol; further, that the simulated TM never attempts to write off the left end of its tape.

$$
\begin{array}{l}
\mathrm{move} = \lambda C. \\
\quad (l, q, r) := C; \\
\quad (c^l, \bar{l}) := l; \\
\quad (c^r, \bar{r}) := r; \\
\quad (q^1, \ldots, q^l) := \mathbf{copy}_l^k q; \\
\quad (q', c', d') := \mathbf{case}\ c^r \mathbf{of} \\
\qquad c_1 : dq^1 c_1; \\
\qquad \ldots \\
\qquad c_{l-1} : dq^{l-1} c_{l-1}; \\
\qquad c_l : (q^l, c_1, 0); \\
\quad \mathbf{case}\ d' \mathbf{of} \\
\qquad l : (\bar{l}, q', (c^l, (c', \bar{r}))); \\
\qquad r : ((c', l), q', \bar{r}); \\
\qquad 0 : (l, q', (c_1, (c_l, \omega)))
\end{array}
$$

5.3 The initial configuration

The initial configuration $C_0 = (\epsilon, q_1, x)$ of a Turing Machine computation is represented by

$$C_0 \;=\; ((c_l,\omega), d_1, (x_1, \ldots, (x_n, (c_l,\omega))\ldots))$$

where $x = x_1 \ldots x_n$ is the input.

5.4 Correctness

The correctness of our simulation follows from the following result.

Theorem 1 *Let $e = move(move(\ldots(move(C_0))\ldots))$, and let e' be a β-reduct of e. If e' has type σ then e has also type σ.*

Proof: (Sketch) Let E be the class of (representations of) configurations; i.e., every e in E is of the form $((x_1, \ldots, (x_m, (c_l,\omega))\ldots), q, (y_1, \ldots, (y_n, (c_l,\omega))\ldots))$ and $x_i, y_j \in C, q \in Q$. Note that E is a class of expressions in β-normal form. Obviously, for every e in E there is e' in E such that move(e) β-reduces to e'. It is sufficient to show that if τ' is the principal type of e' then move(e) has also type τ'. But this follows from the representation considerations above. Then the theorem follows by induction on the number of applications of move from the subject reduction theorem. (End of proof) ■

This theorem is also at the heart of the correctness of Mairson's method. Even though this "invariance" theorem — it is a weak dual to the subject reduction theorem — is only formulated for a specific class of expressions, a much more general definition of a class of β-conversions with invariant typings seems on hand.

5.5 The main loop

So far we have not made use of the polymorphic typing rule for **let**-expressions in ML. We use this rule only once, to encode an exponential number of applications of the move function – regarded as applications "in the types" – to an initial configuration. Our presentation here is pretty much directly from Mairson [Mai90].

Given a Turing Machine T and an input $x = c^1 \ldots c^n$ we encode running T for 2^{cn} on x as follows. The exponential composition of move with itself is accomplished as follows.

sim =
  let $\text{move}_1 = \lambda C.\ \text{move}(\text{move}(C))$ **in**
  let $\text{move}_2 = \lambda C.\ \text{move}_2(\text{move}_2(C))$ **in**
  ...
  let $\text{move}_{cn} = \lambda C.\ \text{move}_{cn-1}(\text{move}_{cn-1}(C))$ **in**
  $(l, q, r) :=$ $\text{move}_{cn}(C_0)$;
  accept := **case** q **of**
    F: **true**;
    $Q - F$: **false**;
  $(\text{accept} \rightarrow K, \lambda xyz.(zx, zy))IK$

Here F is a listing of all accepting states. The invariance of ML typing under **let**-reduction yields the final result.

Lemma 2 *The expression sim is ML typable if and only if the represented Turing Machine T accepts input $x = x_1 \ldots x_n$ within 2^{cn} steps.*

Proof: The **let**-reduct sim2 of sim,

$$
\begin{array}{l}
\text{sim2} = \\
\quad (l, q, r) := \text{move}^{2^{cn}}(C_0); \\
\quad \textbf{accept} := \textbf{case } q \textbf{ of} \\
\qquad F\text{: } \textbf{true}; \\
\qquad Q - F\text{: } \textbf{false}; \\
\quad (\textbf{cond accept } K(\lambda xyz.(zx, zy)))IK
\end{array}
$$

is typable if and only if sim is typable [Dam84]. If T is in an accepting state q_i after 2^{cn} steps then, by Theorem 1,

$$
\begin{array}{l}
(l, q, r) := \text{move}^{2^{cn}}(C_0); \\
q
\end{array}
$$

has the same (principal) type as q_i. Similarly,

$$
\begin{array}{l}
\text{sim2} = \\
\quad (l, q, r) := \text{move}^{2^{cn}}(C_0); \\
\quad \text{accept} := \textbf{case } q \textbf{ of} \\
\qquad F\text{: } \textbf{true}; \\
\qquad Q - F\text{: } \textbf{false}; \\
\quad \text{accept}
\end{array}
$$

has the principal type of **true**: $\forall\alpha\beta.\alpha \to \beta \to \alpha$. Since

$$
\begin{array}{l}
\{\text{accept} : \forall\alpha\beta.\alpha \to \beta \to \alpha \supset\} \\
\quad (\textbf{cond accept } K(\lambda xyz.(zx, zy)))IK : \\
\quad \forall\alpha.\alpha \to \alpha
\end{array}
$$

sim is typable.

If, on the other hand, T is not in an accepting state, the principal type of accept above is the principal type of **false**: $\forall\alpha\beta.\alpha \to \beta \to \beta$. But then

$$(\text{accept} \to K, \lambda xyz.(zx, zy))IK$$

would only be typable if I and K had the same type. But this is impossible. Consequently, sim is not typable. (End of proof) ■

Now we have the main theorem for ML typing.

Theorem 3 *ML typability with tuples and constant ω is complete for DEXPTIME under* log-*space reductions.*

Proof: The theorem follows directly from Lemma 2 since, given Turing Machine description T and input x the program, sim can be constructed in logarithmic space. ML typability is shown to be in DEXPTIME by Kanellakis and Mitchell [KM89]. (End of proof) ■

In this form our theorem is actually weaker than DEXPTIME-completeness results of Mairson [Mai90] and Kfoury, Tiuryn, Urzyczyn [KTU89] since we use tuples and the constant ω. We can strengthen it by getting rid of tuples and treating ω as a free variable [Hen90]. But since, in ML typing, the conventional encoding of tuples — i.e., $(e_1, \ldots, e_k) = \lambda z.ze_1 \ldots e_k$ (with z not free in any of the e_i's) — breaks down, the strengthening comes at the expense of giving up the "double simulation" property (on the "value level" and on the "type level") of our representation. Curiously, transferring this DEXPTIME-hardness proof to the elusive problem of Girard-Reynolds Typability is, on the other hand, only a minor step.

6 The Implicitly Typed Girard-Reynolds Calculus

The explicitly typed Girard-Reynolds Calculus comes by many other names: polymorphic λ-calculus, second order λ-calculus, system F (see, e.g., [Gir71, Rey74, GLT89, Mit88]). In this section we review the *implicitly* typed Girard-Reynolds Calculus (or, for short, *GR-calculus*), which is a type inference system of equally many names. Mitchell calls it *pure typing* [Mit88]; Leivant *type quantification* [Lei83]. The axiom and rule schemes of the implicit GR-calculus are given in Table 2. Note that here, as opposed to ML typing, the type expressions may be arbitrary expressions generated by the grammar

$$\tau \ ::= \ \alpha \mid \tau' \to \tau'' \mid \forall\alpha.\tau'$$

We assume, here as before, standard notions of type inference systems.

The *typability* problem for the GR-calculus is the problem of deciding, given a λ-expression e, where there exist A, τ such that the typing $A \supset e : \tau$ is derivable in the above type inference system. It is also called the full polymorphic type inference problem [Boe85, Pfe88] and the type reconstruction problem for the 2nd order λ-calculus [KT89].

Characterizations of GR typability have been given by Mitchell [Mit88] and Giannani and Ronchi della Rocca [GRDR88]. Restricted or modified GR typability problems have been investigated by McCracken [McC84], Boehm [Boe85, Boe89], Pfenning [Pfe88], O'Toole and Gifford [OJG89], Kfoury and Tiuryn [KT89] and probably many others. Interestingly, partial polymorphic inference [Boe85, Pfe88] and rank-bounded polymorphic inference with suitably typed constants [KT89] have been shown undecidable, yet none of the proofs yield any nontrivial lower bound for full polymorphic type inference. In fact, no nontrivial lower or upper bounds on GR typability were exhibited so far. Our lower bound on full polymorphic type inference is based on techniques developed by Kanellakis and Mitchell [KM89] and Mairson [Mai90] for ML typability.

Let A range over type environments; x over variables; e, e' over λ-expressions; α over type variables; τ, τ' over type expressions. The following are the type inference axiom and rule schemes of the implicitly typed Girard-Reynolds Calculus.

Name	Axiom/rule
(TAUT)	$A\{x : \sigma\} \supset x : \sigma$
(GEN)	$\dfrac{A \supset e : \sigma \quad (\alpha \text{ not free in } A)}{A \supset e : \forall\alpha.\sigma}$
(INST)	$\dfrac{A \supset e : \forall\alpha.\sigma}{A \supset e : \sigma[\tau/\alpha]}$
(ABS)	$\dfrac{A\{x : \tau'\} \supset e : \tau}{A \supset \lambda x.e : \tau' \rightarrow \tau}$
(APPL)	$\dfrac{A \supset e : \tau' \rightarrow \tau \quad A \supset e' : \tau'}{A \supset (ee') : \tau}$

Table 2: Type inference axioms and rules of the implicitly typed Girard-Reynold Calculus (GR-calculus)

7 DEXPTIME-hardness of GR Typability

We shall see that our method of proving ML typability DEXPTIME-hard yields a lower bound of DEXPTIME-hardness for GR typability almost immediately. As far as we know this is the first nontrivial lower bound for GR typbability. We believe, though, that our method of type invariant simulation can be pushed much further and possibly result in nonelementary-recursive lower bounds.

It is easy to see that we can simulate **let**, tuples and selections that occur in ML derivations by pure λ-expressions in the GR-calculus,

$$\begin{aligned} \textbf{let } x = e \textbf{ in } e' &= (\lambda x.e')e \\ (e_1, \ldots, e_k) &= \lambda z.ze_1 \ldots e_k \\ e.i^{(k)} &= e(\lambda x_1 \ldots x_k.x_i) \end{aligned}$$

since their ML typing rules are derived rules in the GR-calculus. E.g.,

$$\text{(LET)} \quad \frac{A \supset e : \sigma \qquad A\{x : \sigma\} \supset e' : \sigma'}{A \supset (\lambda x.e')e : \sigma'}$$

$$\text{(TUPLE)} \quad \frac{A \supset e_1 : \tau_1 \qquad \ldots \qquad A \supset e_k : \tau_k}{A \supset \lambda z.ze_1 \ldots e_k : \forall\gamma.(\tau_1 \to \ldots \tau_k \to \gamma) \to \gamma}$$

$$\text{(SEL)} \quad \frac{A \supset e : \forall\gamma.(\tau_1 \to \ldots \tau_k \to \gamma) \to \gamma \qquad (1 \le i \le k)}{A \supset e(\lambda x_1 \ldots x_k.x_i) : \tau_i}$$

Henceforth we shall consider **let**-expressions, tuples and selections as syntactic sugar for their pure λ-calculus counterparts in the GR-calculus.

Now, consider the expression sim':

sim' = $\lambda\omega$.
  let move$_1$ = λC. move(move(C)) **in**
  let move$_2$ = λC. move$_2$(move$_2$(C)) **in**
  ...
  let move$_{cn}$ = λC. move$_{cn-1}$(move$_{cn-1}$(C)) **in**
  (l, q, r) := move$_{cn}$(C_0);
  accept := **case** q **of**
    F: **true**;
    $Q - F$: **false**;
  (**cond** accept $I(\lambda x.xx))(\lambda x.xx)$

The only differences from sim are the first line and the last line. By λ-abstracting over ω we dispose of ω as a separate constant since λ-bound variables may carry polymorphic types — in particular, $\forall\alpha.\alpha$ — in the GR-calculus, which is not possible in the Milner Calculus. The last line is instrumental since it ensures that sim' has no β-normal form whenever T does not accept its input.

Lemma 4 *The expression sim' is GR typable if and only if the represented Turing Machine T accepts input $x = x_1 \ldots x_n$ within 2^{cn} steps.*

Proof: If T accepts after 2^{cn} steps then, as we have seen in the proof of Lemma 2, there is an ML derivation such that accept has type $\forall\alpha\beta.\alpha \to \beta \to \alpha$. Since every ML derivation can be canonically translated into a GR derivation, there is also a GR derivation such that accept has type $\forall\alpha\beta.\alpha \to \beta \to \alpha$. Since $\lambda x.xx$ has type $(\forall\alpha.\alpha) \to (\forall\alpha.\alpha)$ (amongst others) and I has type $\forall\alpha.\alpha \to \alpha$, we get the following derivation for $A_0 = \{\text{accept} : \forall\alpha\beta.\alpha \to \beta \to \alpha,\ \omega : \forall\alpha.\alpha\}$.

$$
\begin{aligned}
A_0 &\supset \text{accept} : \forall\alpha\beta.\alpha \to \beta \to \alpha \\
A_0 &\supset \text{accept} : (\forall\alpha.\alpha \to \alpha) \to ((\forall\alpha.\alpha) \to (\forall\alpha.\alpha)) \to (\forall\alpha.\alpha \to \alpha) \\
A_0 &\supset \text{accept}\ I(\lambda x.xx) : (\forall\alpha.\alpha \to \alpha) \\
A_0 &\supset \text{accept}\ I(\lambda x.xx) : ((\forall\alpha.\alpha) \to (\forall\alpha.\alpha)) \to ((\forall\alpha.\alpha) \to (\forall\alpha.\alpha)) \\
A_0 &\supset (\text{accept}\ I(\lambda x.xx))(\lambda x.xx) : (\forall\alpha.\alpha) \to (\forall\alpha.\alpha)
\end{aligned}
$$

Consequently, sim' is GR typable.

If, on the other hand, T does not accept after 2^{cn} steps then sim' β-reduces to

$$(\textbf{cond false}\ I(\lambda x.xx))(\lambda x.xx)$$

which, in turn, β-reduces to

$$(\lambda x.xx)(\lambda x.xx).$$

But $(\lambda x.xx)(\lambda x.xx)$. has no β-normal form. Thus sim' is not strongly normalizing, and by Girard's strong normalization theorem [Gir71, GLT89] sim' has no GR typing. (End of proof) ■

We have, as in the ML typability case, the following lower bound for GR typability.

Theorem 5 *GR typability is hard for DEXPTIME under* log*-space reductions.*

Proof: This follows from Lemma 4 and the fact that sim' can be constructed from TM T and input x in logarithmic space. (End of proof) ■

8 Concluding Remarks

We have presented a simulation of Turing Machines up to 2^{cn} steps for inputs of size n in the λ-calculus under β-reduction that has the property that every reduction sequence is invariant under ML typability, respectively GR typability. This yields the first intractibility result for GR typability; in particular, we show that GR typability (also called full polymorphic type inference or type reconstruction for the second order λ-calculus) is DEXPTIME-hard under *log*-space reductions.

Since the class of λ-expressions we use for simulation has the property that, if they have a typing then they have a rank-2 typing (see [Lei83, KT89]), this result is the best we can

achieve since Kfoury and Tiuryn have shown that rank-2 GR typability is equivalent to ML typability [KT89], which is DEXPTIME-decidable.

In pursuing this work we have aimed to identify a "powerful" class of λ-expressions that has the property that

- there is a constant k such that if an expression has a typing derivation of any rank then it has a derivation of rank at most k;
- the class of expressions is closed under β-reduction and GR typability is invariant for any β-reduction sequence (i.e., a weak inverse of the subject reduction theorem).

Much to our surprise a rather obvious and straightforward simulation fits the bill for rank 2. Better bounds can possibly be achieved by considering expressions with higher rank. Mitchell's retyping functions [Mit88] in connection with normalized derivations seem like a promising direction to pursue.

Acknowledgements

I would like to thank Harry Mairson and Hans Leiß for extensive discussions on type inference; in particular Harry for sharing his insights on computing with types with me, not to mention his vibrant enthusiasm and his hilarious jokes.

References

[AM88] A. Appel and D. MacQueen. A Standard ML compiler. Manuscript, 1988.

[Boe85] H. Boehm. Partial polymorphic type inference is undecidable. In *Proc. 26th Annual Symp. on Foundations of Computer Science*, pages 339–345. IEEE, Oct. 1985.

[Boe89] H. Boehm. Type inference in the presence of type abstraction. In *Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 192–206. ACM, ACM Press, June 1989.

[Dam84] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[Gir71] J. Girard. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–92, 1971.

[GLT89] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[GRDR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proc. Symp. on Logic in Computer Sciene*, pages 61–70. IEEE, Computer Society, Computer Society Press, June 1988.

[Hen90] F. Henglein. A simplified proof of DEXPTIME-completeness of ML typing. Manuscript, March 1990.

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

[KM89] P. Kanellakis and J. Mitchell. Polymorphic unification and ML typing (extended abstract). In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages.* ACM, January 1989.

[KT89] A. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order lambda calculus. Technical Report BUCS 89-011, Boston University, Oct. 1989.

[KTU89] A. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. Technical Report BUCS 89-009, Boston University, Oct. 1989. also in Proc. European Symposium on Programming, Copenhagen, Denmark, May 1990.

[Lei83] D. Leivant. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 88–98. ACM, Jan. 1983.

[Mai90] H. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proc. POPL '90.* ACM, Jan. 1990.

[McC84] N. McCracken. The typechecking of programs with implicit type structure. In *Proc. Int'l Symp. on Semantics of Data Types*, pages 301–316. Springer-Verlag, June 1984. Lecture Notes in Computer Science, Vol. 173.

[Mit88] J. Mitchell. Polymorphic type inference and containment. *Information and Control*, 76:211–249, 1988.

[OJG89] J. O'Toole Jr. and D. Gifford. Type reconstruction with first-class polymorphic values. In *Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 207–217. ACM, ACM Press, June 1989.

[Pfe88] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *Proc. 1988 ACM LISP and Functional Programming Conf.*, pages 153–163. ACM, July 1988.

[Rey74] J. Reynolds. Towards a theory of type structure. In *Proc. Programming Symposium*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.

A Lower Bound for Full Polymorphic Type Inference: Girard-Reynolds Typability is DEXPTIME-hard

Fritz Henglein

RUU-CS-90-14
April 1990

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

A Lower Bound for Full Polymorphic Type Inference: Girard-Reynolds Typability is DEXPTIME-hard

Fritz Henglein

RUU-CS-90-14
April 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

A Lower Bound for Full Polymorphic Type Inference: Girard-Reynolds Typability is DEXPTIME-hard

Fritz Henglein
Department of Computer Science
Utrecht University
PO Box 80.089
3508 TB Utrecht
The Netherlands
Internet: henglein@cs.ruu.nl

April 9, 1990

Abstract

The typability problem for the Girard-Reynolds Calculus is an old and elusive problem. Neither nontrivial upper nor lower bounds have been known, not even whether the problem is decidable or not. In this paper we present the, as far as we know, first nontrivial lower bound for Girard-Reynolds typability. Expanding on work by Kanellakis and Mitchell [KM89] and, in particular, Mairson [Mai90] on ML typability we prove that GR typability is DEXPTIME-hard.

Mairson's proof of DEXPTIME-hardness of ML typability relies on an encoding of Boolean values by type schemes with equality between some components of the type scheme. This type equality can be forced in ML due to ML's monomorphic typing rule for λ-bound variables. Since this equality cannot be forced in the Girard-Reynolds Calculus, Mairson's proof cannot naively be transferred to the Girard-Reynolds typability problem.

We show that a conventional (untyped) λ-calculus representation of Boolean values, together with an analogue of Mairson's fan-out gates, results in a straightforward simulation of a deterministic Turing Machine "in the types" for an exponential number of steps. This yields yet another proof, principally based on Mairson's, of DEXPTIME-hardness of ML typability. Since the λ-expression representing a Turing Machine computation simulates the Turing Machine also under β-reduction it is easy to extend this lower bound proof to a DEXPTIME-hardness result for Girard-Reynolds typability by mapping a rejecting computation to a nonnormalizing λ-expression.

The simulation relies on a class of λ-expressions that have a rank-2 typing if they have any typing at all. Since rank-2 typability is DEXPTIME-decidable [KT89] our bound is the best we can achieve using this particular class. It appears possible, though, to achieve better lower bounds by extending our technique of double simulation "in the types" and "in the values" to expressions with (only) higher-rank typings.

1 Introduction

Kanellakis and Mitchell have shown that ML typing is decidable in deterministic exponential time [KM89]. They also gave a reduction of the Quantified Boolean Value problem to ML typing thus giving a lower bound of PSPACE-hardness. Mairson recently improved this result by presenting a generic simulation of deterministic Turing Machines for up to an exponential number of steps, thus proving DEXPTIME-hardness of ML typing. An alternative proof based on characterization by "acyclic" semi-unification was given by Kfoury, Tiuryn and Urzyczyn [KTU89].

Mairson's lower bound can be seen as a generic simulation of deterministic Turing Machines (for up to an exponential number of steps) "in the types" of λ-expressions. By this we mean that the Turing Machine is simulated by the type inference algorithm as it type-checks a λ-expression representing the Turing Machine and its input. Critical use is made of

- a universal "type equality", represented by a λ-expression (see below) that forces, in accordance with the typing rules for ML, the types of some variables to be equal;
- an encoding of the Boolean values **true** and **false** by equality or (possible) inequality of certain types;
- a mechanism ("fan-out gates") for producing several terms with the same Boolean input type from a single Boolean input type to "program around" the side effects of type equality; and
- an encoding of an exponential number of moves by a Turing Machine by a nested **let**-expression of polynomial size, exploiting the polymorphic typing rule of ML only at this point.

The use of universal type equality is at the basis of the simulation as it is used in the definition of the Boolean values, of the fan-out gates, and indirectly also in the remaining steps. We shall, in some sense, simplify Mairson's proof and give a simulation in which another, more "semantic" and more conventional representation of Boolean values is used without resorting to a universal type equality relation. The type equality relation is a consequence of ML's typing rule for λ-bound variables since all occurrences of a λ-bound variable are required to have the same monotype. Since the implicitly typed Girard-Reynolds Calculus makes no such provision, Mairson's technique does not directly extend to it.

The typability problem for the Girard-Reynolds Calculus has been very elusive since its inception while attracting a considerable amount of attention. As far as we know no nontrivial lower or upper bounds have been proved. In this paper we present the first nontrivial lower bound for Girard-Reynolds typability. We show that it is DEXPTIME-hard by adapting a new proof of DEXPTIME-hardness for ML typability, which builds on Mairson [Mai90]. The amazing simplicity of our lower-bound technique for GR-typability derives from the fact that with little change we can adapt a general purpose simulation of Turing Machines in the (untyped) λ-calculus to the typing disciplines imposed by ML and the Girard-Reynolds Calculus.

The outline of the rest of this paper is as follows. In section 2 we present ML with tuples and its typing rules. In section 3 elementary encodings of finite domains are given. Section 4 shows how Mairson's fan-out gates are encoded in our representations. In section 5 we give our version of how exponential-time deterministic Turing Machine computations can be encoded as an ML typability problem. Section 6 introduces the implicitly typed Girard-Reynolds Calculus, and section 7 presents the DEXPTIME-hardness result for Girard-Reynolds typability. Finally, section 8 contains a short summary and some concluding remarks.

2 ML with Tuples

We quickly review ML typing. For technical reasons we add tuples and a constant ω that has any type into our language. The ML typing rules (Milner Calculus) are shown in Table 1. For a review of ML typing the reader is referred to, e.g., [DM82].

3 Elementary Encodings

3.1 Finite Domains

We show that a conventional representation of finite domains in the λ-calculus can be "lifted" to the universe of types. This representation does not rely on the equality of certain type expressions, yet permits an encoding of equality for specific domains, instead of a "universal" encoding.

For the representation of Turing Machines it is necessary to represent and compute with finite domains of values; e.g., states, tape symbols, tape head directions. For this reason we present a general representation of finite domains with a *select* (or *case*) expression. As a special case we get a representation for the Booleans.

Let $D^k = \{d_1, \ldots, d_k\}$ be a given finite domain (set). We represent every element d_i by the i-th k-ary projection function, p_i^k.

$$\begin{aligned} d_1 &= \lambda x_1 \ldots x_k . x_1 \\ &\ldots \\ d_i &= \lambda x_1 \ldots x_k . x_i \\ &\ldots \\ d_k &= \lambda x_1 \ldots x_k . x_k \end{aligned}$$

Using this representation it is simple to give an encoding of a case expression.

$$\begin{aligned} \textbf{case } d \textbf{ of } &= d e_1 \ldots e_k \\ d_1 &: e_1 \\ &\ldots \\ d_k &: e_k \end{aligned}$$

Let A range over type environments; x over variables; e, e' over λ-expressions; α over type variables; $\tau, \tau', \tau_1, \ldots, \tau_k$ over monotypes; σ, σ' over polytypes. The following are the type inference axiom and rule schemes for ML typing with tuples.

Name	Axiom/rule
(CONS)	$A \supset \omega : \sigma$
(TAUT)	$A\{x : \sigma\} \supset x : \sigma$
(GEN)	$\dfrac{A \supset e : \sigma \quad (\alpha \text{ not free in } A)}{A \supset e : \forall\alpha.\sigma}$
(INST)	$\dfrac{A \supset e : \forall\alpha.\sigma}{A \supset e : \sigma[\tau/\alpha]}$
(ABS)	$\dfrac{A\{x : \tau'\} \supset e : \tau}{A \supset \lambda x.e : \tau' \rightarrow \tau}$
(APPL)	$\dfrac{A \supset e : \tau' \rightarrow \tau \quad A \supset e' : \tau'}{A \supset (ee') : \tau}$
(LET)	$\dfrac{A \supset e : \sigma \quad A\{x : \sigma\} \supset e' : \sigma'}{A \supset \textbf{let } x = e \textbf{ in } e' : \sigma'}$
(TUPLE)	$\dfrac{A \supset e_1 : \tau_1 \quad \ldots \quad A \supset e_k : \tau_k}{A \supset (e_1, \ldots, e_k) : (\tau_1, \ldots, \tau_k)}$
(SEL)	$\dfrac{A \supset e : (\tau_1, \ldots, \tau_k) \quad (1 \leq i \leq k)}{A \supset e.i^{(k)} : \tau_i}$

Table 1: Type inference axioms and rules for ML typing with tuples (Milner Calculus)

We shall denote the representations of the elements of $D^2 = \{d_1, d_2\}$ by **true** and **false**, respectively, and instead of

$$\begin{array}{lll} \textbf{case} & d & \textbf{of} \\ & d_1: & e_1 \\ & d_2: & e_2 \end{array}$$

we will write

$$\textbf{cond}\ d\ e_1 e_2$$

Another possible representation of elements of finite domains is as characteristic functions over D; that is, as tuples of length k where, for d_i, the i-th component is **true** and all the other components are **false**.

This is the conventional representation of the Boolean values in the untyped λ-calculus. The term **cond** represents the conditional if-then-else construct. In the Milner Calculus, the type system for the functional core of ML, these combinators have the following principal types.

$$\begin{array}{rcl} \textbf{true} & : & \forall\alpha\beta.\alpha \to \beta \to \alpha \\ \textbf{false} & : & \forall\alpha\beta.\alpha \to \beta \to \beta \\ \textbf{cond} & : & \forall\alpha\beta\gamma.(\alpha \to \beta \to \gamma) \to \alpha \to \beta \to \gamma \end{array}$$

Note that the computation of the types of applied expressions with the conditional and the Boolean values mirror the computation of the values. We can think of **cond** as taking three *type expressions* as input and producing a *type expression* as output. Consider, for example, the simple expressions below and their types, as output by the Standard ML of New Jersey compiler [AM88].

```
- cond tt 5 7.0;
val it = 5 : int

- cond ff 5 7.0;
val it = 7.0 : real
```

We can see that **cond** maps a pair of types **integer** and **real** to **integer** when given (the type of) **true** as its first input and to **real** when given (the type of) **false** as first input. Note, however, that the type of the first argument to **cond** is "side-effected". That is, whereas **false** has the type $\forall\alpha\beta.\alpha \to \beta \to \beta$, in the context **cond false** 5 7.0 it has the type **integer** $\to$ **real** $\to$ **real**. As a consequence we cannot generally use a λ-bound variable more than once if one of its occurrences is in the first argument position of a **cond**.

Our representation is different than the one chosen in Mairson [Mai90]. There, **false** is represented by a λ-expression of type $\forall\alpha.\forall\beta.\forall\gamma.\alpha \to \beta \to \gamma \to \gamma$, and **true** by a term of type $\forall\alpha.\forall\gamma.\alpha \to \alpha \to \gamma \to \gamma$. The underlying idea in Mairson's representation is that a term of type $\sigma \to \tau \to \upsilon \to \upsilon$ is considered a representation of **true** only if σ and τ are identical, and a representation of **false** only if σ and τ are completely independent.

3.2 Assignments

Single *assignments* or *definitions* can be modeled by a beta redex. Again, this is a representation that follows the "value level". That is, we define

$$x := e_1; e_2 \;=\; (\lambda x.e_2)e_1$$

The effect "in the types" is that if e_1 has type τ_1 and $\lambda x.e_2$ has the type $\tau_1 \to \tau_2$ then the whole expression, $(\lambda x.e_2)e_1$, has type τ_2. In ML a beta redex can also be written as a **let**-expression, **let** $x = e_1$ **in** e_2. Since **let**-expressions have a different typing rule than the corresponding the pure beta redexes, however, we shall use the notation above, $x := e_1; e_2$, instead.

We shall use the "pattern matching" notation

$$\begin{array}{lll}(x_1, \ldots, x_k) & := & e_1; \\ & & e_2\end{array}$$

for the sequence of assignments

$$\begin{array}{lll} t & := & e_1; \\ x_1 & := & t.1^{(k)}; \\ & \ldots & \\ x_k & := & t.k^{(k)}; \\ e_2 & & \end{array}$$

4 Preventing Interference

One of the main achievements of Mairson's lower bound proof is the way in which type information is "replicated" in such a fashion that encoding a computation in one copy of type information does not affect the type information of another copy. In particular, using different copies of the same type information, multiple occurrences of values in the first argument position of **cond** and **select** can be emulated without the side-effect on the type of one copy affecting another.

This mechanism is necessary to harness the universal side effect of type equality forced by ML's typing rule for λ-bound variables. We develop, analogously, a way of "copying" type information within our representations.

4.1 Copying Boolean Values

The Booleans **true** and **false** are represented by (the types of) the terms $\lambda xy.x$ and $\lambda xy.y$. We present an expression that, whenever given an expression of either type $\forall\alpha\beta.\alpha \to \beta \to \alpha$ or $\forall\alpha\beta.\alpha \to \beta \to \beta$, returns a *pair*, p, such that both $p.1$ and $p.2$ have the same type as the given input. Given such an expression $\mathbf{copy}_2$, corresponding to Mairson's "fan-out gate", we can devise expressions for producing more than two copies of a type and for producing copies of our representations of finite domains.

At this point we use the constant ω with type $\forall\alpha.\alpha$.

$$\begin{aligned}\mathbf{copy}_2 = \lambda d.\\ &(\lambda x_1 x_2.(d(x_1,\omega)(x_2,\omega)).1^{(2)},\\ &\lambda y_1 y_2.(d(\omega,y_1)(\omega,y_2)).2^{(2)})\end{aligned}$$

The principal type of this combinator is

$$\begin{aligned}&\mathbf{copy}_2 : \forall\alpha_1\alpha_2\beta_1\beta_2\gamma_1\gamma_2.\\ &\quad((\alpha_1,\alpha_2)\to(\beta_1,\beta_2)\to(\gamma_1,\gamma_2))\to(\alpha_1\to\beta_1\to\gamma_1,\alpha_2\to\beta_2\to\gamma_2)\end{aligned}$$

This construction can, of course, be generalized to generate any fixed number of "copies" of independent types. We shall write $\mathbf{copy}_l$ for the λ-expression that generates l copies of a Boolean representation.

Note that, on the untyped level, the effect of $\mathbf{copy}_2$ is completely neutral: $\mathbf{copy}_2(\mathbf{true})$ β-reduces[1] to $(\mathbf{true},\mathbf{true})$; and $\mathbf{copy}_2(\mathbf{false})$ β-reduces to $(\mathbf{false},\mathbf{false})$.

4.2 Copying Elements of Finite Domains

The construction for copying Boolean values can be extended in a straightforward fashion to elements of finite domains of size k. In this case the corresponding copy function is denoted by $\mathbf{copy}_2^k$

$$\begin{aligned}\mathbf{copy}_2^k = \lambda d.\\ &(\lambda x_1\ldots x_k.(d(x_1,\omega)\ldots(x_k,\omega)).1^{(2)},\\ &\lambda y_1\ldots y_k.(d(\omega,y_1)\ldots(\omega,y_k)).2^{(2)})\end{aligned}$$

and its principal type is

$$\begin{aligned}&\mathbf{copy}_2^k : \forall\alpha_1\alpha_2\beta_1\beta_2\gamma_1\gamma_2.\\ &\quad((\alpha_1,\beta_1)\to\ldots\to(\alpha_k,\beta_k)\to(\gamma,\delta))\to(\alpha_1\to\ldots\to\alpha_k\to\gamma,\beta_1\to\ldots\to\beta_k\to\delta)\end{aligned}$$

Again, this construction can be generalized to l copies of k-element domains. We shall denote the corresponding representation $\mathbf{copy}_l^k$.

5 Representing Turing Machines

A Turing Machine (TM) consists of the following components (see [HU79]).

[1] We call tuple redex reduction $(e_1,\ldots,e_k).i^{(k)} \Rightarrow e_i$ also β-reduction for convenience' sake.

Q = $\{q_1, \ldots, q_k\}$: a finite set
of states, with initial state q_1,
and accepting states $F \subset Q$

C = $\{c_1, \ldots, c_l\}$: a finite set of
tape symbols, with blank symbol c_1

D = $\{l, r\}$ or $\{l, 0, r\}$:
two or three "directions" in which the tape head may move
(left, not at all, right)

d : $Q \times C \to Q \times C \times D$:
a partial transition function.

A *configuration (instantaneous description)* is defined as a triple, (l, q, r) where $l, r \in C^*, q \in Q$. The next-move function is induced by the transition function as follows[2]:

```
Conf = {(l,q,r) : l,r ∈ C*, q ∈ Q}
move: Conf → Conf (partial)
move (l,q,r) =
    if r = ε then:
        return (l,q,c1);
    (q',c',d') := d(q,r(1));
    if d' = l then:
        if l = ε then:
            return (l,q_rej,r);
        else:
            return (l(2..),q',l(1)c'r(2..));
    if d' = r then:
        return (lc',q',r(2..));
    if d' = 0 then:
        return (l,q',c'r(2..));
```

5.1 The transition function

We now show how the transition function d is represented. For every combination of state q_i and tape symbol c_j the transition function d returns a triple d_{ij} consisting of the new state, the character to be written and the direction in which the tape head is to move. We assume, w.l.o.g., that d is total, but it is also possible to emulate the Turing Machine with a partial transition function directly.

[2]If the notation doesn't make sense, don't worry. The function move is simply the transition relation $\vdash_M$ from [HU79].

$$
\begin{array}{l}
\mathrm{d} = \lambda qc. \\
\quad (c^1, \ldots, c^k) := \mathbf{copy}_k^l c; \\
\quad \mathbf{case}\ q\ \mathbf{of} \\
\quad q_1 : \mathbf{case}\ c^1\ \mathbf{of} \\
\qquad\quad c_1 : d_{11}; \\
\qquad\quad \ldots \\
\qquad\quad c_l : d_{1l}; \\
\quad q_2 : \mathbf{case}\ c^2\ \mathbf{of} \\
\qquad\quad c_1 : d_{21}; \\
\qquad\quad \ldots \\
\qquad\quad c_l : d_{2l}; \\
\quad \ldots \\
\quad q_k : \mathbf{case}\ c^k\ \mathbf{of} \\
\qquad\quad c_1 : d_{k1}; \\
\qquad\quad \ldots \\
\qquad\quad c_l : d_{kl};
\end{array}
$$

Note that in the above definition both arguments of d are side-effected since they occur in the first positions of case expressions.

5.2 The next-move function

As we have seen, the transition function induces a next-move function on configurations. We can almost transliterate this description into a representation "in the types" *and* "in the values". The generally unbounded tape contents are represented by nested pairs. W.l.o.g. we assume that tape symbol c_1 represents the blank character and c_l a special "end-of-tape" symbol; further, that the simulated TM never attempts to write off the left end of its tape.

$$
\begin{array}{l}
\mathrm{move} = \lambda C. \\
\quad (l, q, r) := C; \\
\quad (c^l, \bar{l}) := l; \\
\quad (c^r, \bar{r}) := r; \\
\quad (q^1, \ldots, q^l) := \mathbf{copy}_l^k q; \\
\quad (q', c', d') := \mathbf{case}\ c^r\ \mathbf{of} \\
\qquad\quad c_1 : dq^1 c_1; \\
\qquad\quad \ldots \\
\qquad\quad c_{l-1} : dq^{l-1} c_{l-1}; \\
\qquad\quad c_l : (q^l, c_1, 0); \\
\quad \mathbf{case}\ d'\mathbf{of} \\
\qquad\quad l : (\bar{l}, q', (c^l, (c', \bar{r}))); \\
\qquad\quad r : ((c', l), q', \bar{r}); \\
\qquad\quad 0 : (l, q', (c_1, (c_l, \omega)))
\end{array}
$$

5.3 The initial configuration

The initial configuration $C_0 = (\epsilon, q_1, x)$ of a Turing Machine computation is represented by

$$C_0 \;=\; ((c_l, \omega), d_1, (x_1, \ldots, (x_n, (c_l, \omega))\ldots))$$

where $x = x_1 \ldots x_n$ is the input.

5.4 Correctness

The correctness of our simulation follows from the following result.

Theorem 1 *Let $e = move(move(\ldots(move(C_0))\ldots))$, and let e' be a β-reduct of e. If e' has type σ then e has also type σ.*

Proof: (Sketch) Let E be the class of (representations of) configurations; i.e., every e in E is of the form $((x_1, \ldots, (x_m, (c_l, \omega))\ldots), q, (y_1, \ldots, (y_n, (c_l, \omega))\ldots))$ and $x_i, y_j \in C, q \in Q$. Note that E is a class of expressions in β-normal form. Obviously, for every e in E there is e' in E such that move(e) β-reduces to e'. It is sufficient to show that if τ' is the principal type of e' then move(e) has also type τ'. But this follows from the representation considerations above. Then the theorem follows by induction on the number of applications of move from the subject reduction theorem. (End of proof) ■

This theorem is also at the heart of the correctness of Mairson's method. Even though this "invariance" theorem — it is a weak dual to the subject reduction theorem — is only formulated for a specific class of expressions, a much more general definition of a class of β-conversions with invariant typings seems on hand.

5.5 The main loop

So far we have not made use of the polymorphic typing rule for **let**-expressions in ML. We use this rule only once, to encode an exponential number of applications of the move function – regarded as applications "in the types" – to an initial configuration. Our presentation here is pretty much directly from Mairson [Mai90].

Given a Turing Machine T and an input $x = c^1 \ldots c^n$ we encode running T for 2^{cn} on x as follows. The exponential composition of move with itself is accomplished as follows.

sim =
  let $\text{move}_1 = \lambda C.\ \text{move}(\text{move}(C))$ **in**
  let $\text{move}_2 = \lambda C.\ \text{move}_2(\text{move}_2(C))$ **in**
  ...
  let $\text{move}_{cn} = \lambda C.\ \text{move}_{cn-1}(\text{move}_{cn-1}(C))$ **in**
  $(l, q, r) :=$ $\text{move}_{cn}(C_0)$;
  accept := **case** q **of**
    F: **true**;
    $Q - F$: **false**;
  $(\text{accept} \rightarrow K, \lambda xyz.(zx, zy))IK$

Here F is a listing of all accepting states. The invariance of ML typing under let-reduction yields the final result.

Lemma 2 *The expression sim is ML typable if and only if the represented Turing Machine T accepts input $x = x_1 \ldots x_n$ within 2^{cn} steps.*

Proof: The let-reduct sim2 of sim,

$$\begin{array}{l}\text{sim2} = \\ \quad (l, q, r) := \text{move}^{2^{cn}}(C_0); \\ \quad \text{accept} := \textbf{case } q \textbf{ of} \\ \quad\quad F\text{: } \textbf{true}; \\ \quad\quad Q - F\text{: } \textbf{false}; \\ \quad (\textbf{cond accept } K(\lambda xyz.(zx, zy)))IK\end{array}$$

is typable if and only if sim is typable [Dam84]. If T is in an accepting state q_i after 2^{cn} steps then, by Theorem 1,

$$\begin{array}{l}(l, q, r) := \text{move}^{2^{cn}}(C_0); \\ q\end{array}$$

has the same (principal) type as q_i. Similarly,

$$\begin{array}{l}\text{sim2} = \\ \quad (l, q, r) := \text{move}^{2^{cn}}(C_0); \\ \quad \text{accept} := \textbf{case } q \textbf{ of} \\ \quad\quad F\text{: } \textbf{true}; \\ \quad\quad Q - F\text{: } \textbf{false}; \\ \quad \text{accept}\end{array}$$

has the principal type of **true**: $\forall\alpha\beta.\alpha \to \beta \to \alpha$. Since

$$\begin{array}{l}\{\text{accept} : \forall\alpha\beta.\alpha \to \beta \to \alpha \supset\} \\ \quad (\textbf{cond accept } K(\lambda xyz.(zx, zy)))IK : \\ \quad \forall\alpha.\alpha \to \alpha\end{array}$$

sim is typable.

If, on the other hand, T is not in an accepting state, the principal type of accept above is the principal type of **false**: $\forall\alpha\beta.\alpha \to \beta \to \beta$. But then

$$(\text{accept } \to K, \lambda xyz.(zx, zy))IK$$

would only be typable if I and K had the same type. But this is impossible. Consequently, sim is not typable. (End of proof) ■

Now we have the main theorem for ML typing.

Theorem 3 *ML typability with tuples and constant ω is complete for DEXPTIME under* log-*space reductions.*

Proof: The theorem follows directly from Lemma 2 since, given Turing Machine description T and input x the program, sim can be constructed in logarithmic space. ML typability is shown to be in DEXPTIME by Kanellakis and Mitchell [KM89]. (End of proof) ■

In this form our theorem is actually weaker than DEXPTIME-completeness results of Mairson [Mai90] and Kfoury, Tiuryn, Urzyczyn [KTU89] since we use tuples and the constant ω. We can strengthen it by getting rid of tuples and treating ω as a free variable [Hen90]. But since, in ML typing, the conventional encoding of tuples — i.e., $(e_1, \ldots, e_k) = \lambda z.ze_1 \ldots e_k$ (with z not free in any of the e_i's) — breaks down, the strengthening comes at the expense of giving up the "double simulation" property (on the "value level" and on the "type level") of our representation. Curiously, transferring this DEXPTIME-hardness proof to the elusive problem of Girard-Reynolds Typability is, on the other hand, only a minor step.

6 The Implicitly Typed Girard-Reynolds Calculus

The explicitly typed Girard-Reynolds Calculus comes by many other names: polymorphic λ-calculus, second order λ-calculus, system F (see, e.g., [Gir71, Rey74, GLT89, Mit88]). In this section we review the *implicitly* typed Girard-Reynolds Calculus (or, for short, *GR-calculus*), which is a type inference system of equally many names. Mitchell calls it *pure typing* [Mit88]; Leivant *type quantification* [Lei83]. The axiom and rule schemes of the implicit GR-calculus are given in Table 2. Note that here, as opposed to ML typing, the type expressions may be arbitrary expressions generated by the grammar

$$\tau \ ::= \ \alpha \mid \tau' \to \tau'' \mid \forall\alpha.\tau'$$

We assume, here as before, standard notions of type inference systems.

The *typability* problem for the GR-calculus is the problem of deciding, given a λ-expression e, where there exist A, τ such that the typing $A \supset e : \tau$ is derivable in the above type inference system. It is also called the full polymorphic type inference problem [Boe85, Pfe88] and the type reconstruction problem for the 2nd order λ-calculus [KT89].

Characterizations of GR typability have been given by Mitchell [Mit88] and Giannani and Ronchi della Rocca [GRDR88]. Restricted or modified GR typability problems have been investigated by McCracken [McC84], Boehm [Boe85, Boe89], Pfenning [Pfe88], O'Toole and Gifford [OJG89], Kfoury and Tiuryn [KT89] and probably many others. Interestingly, partial polymorphic inference [Boe85, Pfe88] and rank-bounded polymorphic inference with suitably typed constants [KT89] have been shown undecidable, yet none of the proofs yield any nontrivial lower bound for full polymorphic type inference. In fact, no nontrivial lower or upper bounds on GR typability were exhibited so far. Our lower bound on full polymorphic type inference is based on techniques developed by Kanellakis and Mitchell [KM89] and Mairson [Mai90] for ML typability.

Let A range over type environments; x over variables; e, e' over λ-expressions; α over type variables; τ, τ' over type expressions. The following are the type inference axiom and rule schemes of the implicitly typed Girard-Reynolds Calculus.

Name	Axiom/rule
(TAUT)	$A\{x:\sigma\} \supset x:\sigma$
(GEN)	$\dfrac{A \supset e:\sigma \quad (\alpha \text{ not free in } A)}{A \supset e:\forall\alpha.\sigma}$
(INST)	$\dfrac{A \supset e:\forall\alpha.\sigma}{A \supset e:\sigma[\tau/\alpha]}$
(ABS)	$\dfrac{A\{x:\tau'\} \supset e:\tau}{A \supset \lambda x.e:\tau' \to \tau}$
(APPL)	$\dfrac{A \supset e:\tau' \to \tau \quad A \supset e':\tau'}{A \supset (ee'):\tau}$

Table 2: Type inference axioms and rules of the implicitly typed Girard-Reynold Calculus (GR-calculus)

7 DEXPTIME-hardness of GR Typability

We shall see that our method of proving ML typability DEXPTIME-hard yields a lower bound of DEXPTIME-hardness for GR typability almost immediately. As far as we know this is the first nontrivial lower bound for GR typbability. We believe, though, that our method of type invariant simulation can be pushed much further and possibly result in nonelementary-recursive lower bounds.

It is easy to see that we can simulate **let**, tuples and selections that occur in ML derivations by pure λ-expressions in the GR-calculus,

$$\begin{aligned} \textbf{let } x = e \textbf{ in } e' &= (\lambda x.e')e \\ (e_1, \ldots, e_k) &= \lambda z.ze_1 \ldots e_k \\ e.i^{(k)} &= e(\lambda x_1 \ldots x_k.x_i) \end{aligned}$$

since their ML typing rules are derived rules in the GR-calculus. E.g.,

$$\text{(LET)} \quad \frac{A \supset e : \sigma \qquad A\{x : \sigma\} \supset e' : \sigma'}{A \supset (\lambda x.e')e : \sigma'}$$

$$\text{(TUPLE)} \quad \frac{A \supset e_1 : \tau_1 \qquad \ldots \qquad A \supset e_k : \tau_k}{A \supset \lambda z.ze_1 \ldots e_k : \forall\gamma.(\tau_1 \to \ldots \tau_k \to \gamma) \to \gamma}$$

$$\text{(SEL)} \quad \frac{A \supset e : \forall\gamma.(\tau_1 \to \ldots \tau_k \to \gamma) \to \gamma \qquad (1 \le i \le k)}{A \supset e(\lambda x_1 \ldots x_k.x_i) : \tau_i}$$

Henceforth we shall consider **let**-expressions, tuples and selections as syntactic sugar for their pure λ-calculus counterparts in the GR-calculus.

Now, consider the expression sim':

```
sim' = λω.
    let move_1 = λC. move(move(C)) in
    let move_2 = λC. move_2(move_2(C)) in
    ...
    let move_cn = λC. move_cn−1(move_cn−1(C)) in
    (l, q, r) := move_cn(C_0);
    accept := case q of
        F: true;
        Q − F: false;
    (cond accept I(λx.xx))(λx.xx)
```

The only differences from sim are the first line and the last line. By λ-abstracting over ω we dispose of ω as a separate constant since λ-bound variables may carry polymorphic types — in particular, $\forall\alpha.\alpha$ — in the GR-calculus, which is not possible in the Milner Calculus. The last line is instrumental since it ensures that sim' has no β-normal form whenever T does not accept its input.

Lemma 4 *The expression sim' is GR typable if and only if the represented Turing Machine T accepts input $x = x_1 \ldots x_n$ within 2^{cn} steps.*

Proof: If T accepts after 2^{cn} steps then, as we have seen in the proof of Lemma 2, there is an ML derivation such that accept has type $\forall\alpha\beta.\alpha \to \beta \to \alpha$. Since every ML derivation can be canonically translated into a GR derivation, there is also a GR derivation such that accept has type $\forall\alpha\beta.\alpha \to \beta \to \alpha$. Since $\lambda x.xx$ has type $(\forall\alpha.\alpha) \to (\forall\alpha.\alpha)$ (amongst others) and I has type $\forall\alpha.\alpha \to \alpha$, we get the following derivation for $A_0 = \{\text{accept} : \forall\alpha\beta.\alpha \to \beta \to \alpha,\ \omega : \forall\alpha.\alpha\}$.

$$
\begin{array}{lll}
A_0 & \supset & \text{accept} : \forall\alpha\beta.\alpha \to \beta \to \alpha \\
A_0 & \supset & \text{accept} : (\forall\alpha.\alpha \to \alpha) \to ((\forall\alpha.\alpha) \to (\forall\alpha.\alpha)) \to (\forall\alpha.\alpha \to \alpha) \\
A_0 & \supset & \text{accept}\ I(\lambda x.xx) : (\forall\alpha.\alpha \to \alpha) \\
A_0 & \supset & \text{accept}\ I(\lambda x.xx) : ((\forall\alpha.\alpha) \to (\forall\alpha.\alpha)) \to ((\forall\alpha.\alpha) \to (\forall\alpha.\alpha)) \\
A_0 & \supset & (\text{accept}\ I(\lambda x.xx))(\lambda x.xx) : (\forall\alpha.\alpha) \to (\forall\alpha.\alpha)
\end{array}
$$

Consequently, sim' is GR typable.

If, on the other hand, T does not accept after 2^{cn} steps then sim' β-reduces to

$$(\textbf{cond false}\ I(\lambda x.xx))(\lambda x.xx)$$

which, in turn, β-reduces to

$$(\lambda x.xx)(\lambda x.xx).$$

But $(\lambda x.xx)(\lambda x.xx)$. has no β-normal form. Thus sim' is not strongly normalizing, and by Girard's strong normalization theorem [Gir71, GLT89] sim' has no GR typing. (End of proof) ■

We have, as in the ML typability case, the following lower bound for GR typability.

Theorem 5 *GR typability is hard for DEXPTIME under* log*-space reductions.*

Proof: This follows from Lemma 4 and the fact that sim' can be constructed from TM T and input x in logarithmic space. (End of proof) ■

8 Concluding Remarks

We have presented a simulation of Turing Machines up to 2^{cn} steps for inputs of size n in the λ-calculus under β-reduction that has the property that every reduction sequence is invariant under ML typability, respectively GR typability. This yields the first intractibility result for GR typability; in particular, we show that GR typability (also called full polymorphic type inference or type reconstruction for the second order λ-calculus) is DEXPTIME-hard under *log*-space reductions.

Since the class of λ-expressions we use for simulation has the property that, if they have a typing then they have a rank-2 typing (see [Lei83, KT89]), this result is the best we can

achieve since Kfoury and Tiuryn have shown that rank-2 GR typability is equivalent to ML typability [KT89], which is DEXPTIME-decidable.

In pursuing this work we have aimed to identify a "powerful" class of λ-expressions that has the property that

- there is a constant k such that if an expression has a typing derivation of any rank then it has a derivation of rank at most k;
- the class of expressions is closed under β-reduction and GR typability is invariant for any β-reduction sequence (i.e., a weak inverse of the subject reduction theorem).

Much to our surprise a rather obvious and straightforward simulation fits the bill for rank 2. Better bounds can possibly be achieved by considering expressions with higher rank. Mitchell's retyping functions [Mit88] in connection with normalized derivations seem like a promising direction to pursue.

Acknowledgements

I would like to thank Harry Mairson and Hans Leiß for extensive discussions on type inference; in particular Harry for sharing his insights on computing with types with me, not to mention his vibrant enthusiasm and his hilarious jokes.

References

[AM88] A. Appel and D. MacQueen. A Standard ML compiler. Manuscript, 1988.

[Boe85] H. Boehm. Partial polymorphic type inference is undecidable. In *Proc. 26th Annual Symp. on Foundations of Computer Science*, pages 339–345. IEEE, Oct. 1985.

[Boe89] H. Boehm. Type inference in the presence of type abstraction. In *Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 192–206. ACM, ACM Press, June 1989.

[Dam84] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th Annual ACM Symp. on Principles of Programming Languages*, pages 207–212, Jan. 1982.

[Gir71] J. Girard. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–92, 1971.

[GLT89] J. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1989.

[GRDR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proc. Symp. on Logic in Computer Sciene*, pages 61–70. IEEE, Computer Society, Computer Society Press, June 1988.

[Hen90] F. Henglein. A simplified proof of DEXPTIME-completeness of ML typing. Manuscript, March 1990.

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

[KM89] P. Kanellakis and J. Mitchell. Polymorphic unification and ML typing (extended abstract). In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages.* ACM, January 1989.

[KT89] A. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order lambda calculus. Technical Report BUCS 89-011, Boston University, Oct. 1989.

[KTU89] A. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. Technical Report BUCS 89-009, Boston University, Oct. 1989. also in Proc. European Symposium on Programming, Copenhagen, Denmark, May 1990.

[Lei83] D. Leivant. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 88–98. ACM, Jan. 1983.

[Mai90] H. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proc. POPL '90.* ACM, Jan. 1990.

[McC84] N. McCracken. The typechecking of programs with implicit type structure. In *Proc. Int'l Symp. on Semantics of Data Types*, pages 301–316. Springer-Verlag, June 1984. Lecture Notes in Computer Science, Vol. 173.

[Mit88] J. Mitchell. Polymorphic type inference and containment. *Information and Control*, 76:211–249, 1988.

[OJG89] J. O'Toole Jr. and D. Gifford. Type reconstruction with first-class polymorphic values. In *Proc. SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 207–217. ACM, ACM Press, June 1989.

[Pfe88] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *Proc. 1988 ACM LISP and Functional Programming Conf.*, pages 153–163. ACM, July 1988.

[Rey74] J. Reynolds. Towards a theory of type structure. In *Proc. Programming Symposium*, volume 19 of *LNCS*, pages 408–425. Springer-Verlag, 1974.