Relating Typability and Expressiveness in Finite-Rank Intersection Type Systems (Extended Abstract)

Assaf J. Kfoury*
Boston University
http://www.cs.bu.edu/~kfoury

Harry G. Mairson†
Brandeis University
http://www.cs.brandeis.edu/~mairson

Franklyn A. Turbak‡
Wellesley College
http://www.wellesley.edu/CS/fturbak.html

J. B. Wells §
Heriot-Watt University
http://www.cee.hw.ac.uk/~jbw

Abstract

We investigate finite-rank intersection type systems, analyzing the complexity of their type inference problems and their relation to the problem of recognizing semantically equivalent terms. Intersection types allow something of type $\tau_1 \wedge \tau_2$ to be used in some places at type τ_1 and in other places at type τ_2. A *finite-rank* intersection type system bounds how deeply the $\wedge$ can appear in type expressions. Such type systems enjoy strong normalization, subject reduction, and computable type inference, and they support a pragmatics for implementing parametric polymorphism. As a consequence, they provide a conceptually simple and tractable alternative to the impredicative polymorphism of System F and its extensions, while typing many more programs than the Hindley-Milner type system found in ML and Haskell.

While type inference is computable at every rank, we show that its complexity grows exponentially as rank increases. Let $\mathbf{K}(0,n) = n$ and $\mathbf{K}(t+1,n) = 2^{\mathbf{K}(t,n)}$; we prove that recognizing the pure λ-terms of size n that are typable at rank k is complete for $\text{DTIME}[\mathbf{K}(k-1,n)]$. We then consider the problem of deciding whether two λ-terms typable at rank k have the same normal form, generalizing a well-known result of Statman from simple types to finite-rank intersection types. We show that the equivalence problem is $\text{DTIME}[\mathbf{K}(\mathbf{K}(k-1,n),2)]$-complete. This relationship between the complexity of typability and expressiveness is identical in well-known decidable type systems such as simple types and Hindley-Milner types, but seems to fail for System F and its generalizations. The correspondence gives rise to a conjecture that if $\mathcal{T}$ is a predicative type system where typability has complexity $t(n)$ and expressiveness has complexity $e(n)$, then $t(n) = \Omega(\log^* e(n))$.

1 Introduction

With intersection types, something of type $\tau_1 \wedge \tau_2$ must satisfy both types, so it can be used in some places at type τ_1 and in other places at type τ_2. Most prior work has obtained this kind of *type polymorphism* with universal quantifiers, where something of type $\forall\alpha.\tau$ satisfies the instantiated type $\tau[\alpha{:=}\tau']$ for every type τ' and hence can be used at each instantiated type. Polymorphism of types is essential for code reuse, e.g., in implementing operations on generic containers such as lists, trees, etc.

It is well known that it is undecidable whether a λ-term M is typable with intersection and function types, because this holds exactly when M is strongly normalizable. However, the *rank-2* restriction of intersection types, studied by van Bakel [vB93] and Jim [Jim96], has computable type inference, and recognizes strictly more typable terms than the Hindley-Milner system at the core of ML and Haskell type inference. The *finite-rank* restriction on intersection types bounds how deeply the $\wedge$ can appear in type expressions, counting nesting in the left arguments of the $\to$ type constructor. Recently it has been shown that finite-rank intersection types have computable type inference for any rank [KW99]. As such, they provide a conceptually simple alternative to the impredicative polymorphism of System F and its extensions, supporting a pragmatics for implement-

*Supported by NATO grant CRG 971607, NSF grant CCR–9417382, and NSF grant EIA–9806747.
†Supported by ONR Grant N00014-93-1-1015, NSF Grant CCR-9619638, and the Tyson Foundation.
‡Supported by NSF grant EIA–9806747.
§Supported by EPSRC grant GR/L 36963, and NSF grant EIA–9806747.

ing parametric polymorphism. Finite-rank intersection types are being investigated in the context of the Church Project's experimental flow-type compiler.[1]

We analyze the complexity of two decision problems concerning the rank-k-bounded intersection type system for the λ-calculus:

Typability: Given a pure λ-term of size n, can it be typed?

Expressiveness: Given two typable pure λ-terms of size n, do they have the same normal form? (This is a simple form of detecting program equivalence.)

The Kalmar-elementary functions $\mathbf{K}(t,-)$ are defined so that $\mathbf{K}(0,n) = n$ and $\mathbf{K}(t+1,n) = 2^{\mathbf{K}(t,n)}$. We prove that rank-$k$ typability is $\mathrm{DTIME}[\mathbf{K}(k-1,n)]$-complete and that rank-$k$ expressiveness is $\mathrm{DTIME}[\mathbf{K}(\mathbf{K}(k-1,n),2)]$-complete.[2]

The upper bound on typability is proven purely syntactically, by giving a simplifier $\mathsf{Simplify}(k)$ such that the pure λ-term M is typable at rank k iff the λ-term $\mathsf{Simplify}(k)(M)$ satisfies a polynomial-time test, where $\mathsf{size}(\mathsf{Simplify}(k)(M)) \le \mathbf{K}(k-1, p(\mathsf{size}(M)))$, and p is a polynomial function. The upper bound is complemented by a tight lower bound, borrowing techniques for proving bounds on typability for F_ω, where computation time bounded by Kalmar-elementary functions is simulated by any correct type-inference algorithm.

The bounds on expressiveness are both good and bad, because as the expressiveness of a language grows, so does the difficulty of reasoning about it. Enormous lower bounds for expressiveness means that the language really is expressive. Enormous lower bounds on typability indicates computation that we think of as being at runtime can be simulated at compile time.

These results are interesting from the perspectives of theory as well as practice. Obviously, the results on typability make clear that the delineation between "decidable" and "undecidable" type inference is hardly the place to draw the line—computations taking $\mathbf{K}(10,n)$ steps are no more desirable than ones that do not terminate. However, similar complexity bounds on typability in ML are not in agreement with the practical experience of ML programmers, who experience ML type inference as being efficient. The precise impact of these astronomical bounds in practice remains to be seen.

What becomes very clear in this analysis is how bounds on typability and expressiveness are intimately related. In particular, the key unifying idea is the construction of large polymorphic iterators which, while minimizing the initial program size, maximize the number of times they can compose a function with itself. These iterators are polymorphic in that each iteration can use a different type. We can give identical examples of this relationship for well-known decidable type systems, such as simple types and ML types, but the relationship seems to fail for System F and its extensions. The correspondence gives rise to a conjecture[3] concerning the essential relationship between deciding typability and language expressiveness: if $\mathcal{T}$ is a predicative type system where typability has complexity $t(n)$ and expressiveness has complexity $e(n)$, then $t(n) = \Omega(\log^* e(n))$.[4]

2 Intersection Type Systems

Untyped Contexts and Terms:

$$
\begin{array}{rcl}
w,x,y,z \in \mathsf{TermVar} & & \\
C \in \mathsf{Context} & ::= & \Box \mid x \mid (\lambda x.C) \mid (C_1\, C_2) \\
M,N,P,Q \in \mathsf{Term} & = & \{C \in \mathsf{Context} \mid C \text{ does not mention } \Box\}
\end{array}
$$

Untyped Notions of Reduction (contextually closed in the usual manner):

$$
\begin{array}{lcl}
(\lambda x.M)N & \to_\beta & M[x := N] \\
((\lambda x.N)P)Q & \to_{\mathrm{T}} & (\lambda x.NQ)P \\
(\lambda_{\mathrm{I}} x.M)N & \to_{\beta_{\mathrm{I}}} & M[x := N] \\
(\lambda_{\mathrm{K}} x.M)N & \to_{\beta_{\mathrm{K}}} & M
\end{array}
$$

Syntactic Sugar:

$$
\begin{array}{lcll}
(\lambda_{\mathrm{I}} x.M) & \equiv & (\lambda x.M) & \text{where } x \in \mathsf{FV}(M) \\
(\lambda_{\mathrm{K}} x.M) & \equiv & (\lambda x.M) & \text{where } x \notin \mathsf{FV}(M) \\
(\mathbf{let}\ y = N\ \mathbf{in}\ P) & \equiv & (\lambda y.P)N & \\
(\mathbf{let}_{\mathrm{I}}\ y = N\ \mathbf{in}\ P) & \equiv & (\lambda_{\mathrm{I}} y.P)N & \\
(\mathbf{let}_{\mathrm{K}}\ y = N\ \mathbf{in}\ P) & \equiv & (\lambda_{\mathrm{K}} y.P)N &
\end{array}
$$

Figure 1: Untyped calculus.

In figure 1, we define the λ-calculus (untyped) and an additional notion of reduction, T, the purpose of which is explained in section 3. We call the terms of the λ-calculus *untyped* to indicate they contain no type annotations. Note that terms are defined as holeless contexts. The notation $C[M_1, \ldots, M_n]$ denotes the term that results from filling the n holes of context C from left to right. We ignore issues of variable capture, renaming, and α-conversion by following Barendregt's conventions [Bar84].[5] Let $\mathsf{FV}(M)$ denote the set of free variables of term M. Notice our conventions for distinguishing I- and K- abstractions and redexes.

The syntax for types appears in figure 2. There are two mutually defined type domains: $\mathcal{T}^\to$ includes type variables and function types between $\mathcal{T}^\wedge$ and $\mathcal{T}^\to$, while $\mathcal{T}^\wedge$ includes intersection types, which are sets of elements in $\mathcal{T}^\to$. This restricted grammar simplifies type

[1] ⟨URL:http://www.cs.bu.edu/groups/church/⟩.

[2] Note that $\mathbf{K}(4,2) \approx 10^{19{,}500}$, thus $\mathbf{K}(\mathbf{K}(4,2),2)$ is a stack of 2s about $10^{19{,}500}$ high.

[3] This conjecture was advanced informally by one of us (Mairson) several years ago. It may well need modification and amendments, and it may even be false. But the results of this paper show it is a reasonable first guess as to how one relates typability and expressiveness in complexity-theoretic terms.

[4] The expression $\log^* n$ is the minimal natural number m such that $\log^{(m)} n \le 1$, where $\log^{(i)}$ is the i-fold self-composition of the log function. The function $\log^*$ serves as an inverse to $\mathbf{K}(-,2)$.

[5] For example, in the definition of T-reduction in figure 1, these conventions guarantee that x does not appear free in Q.

Types:

$$
\begin{aligned}
\alpha, \beta, \gamma &\in \mathsf{TyVar} \\
\tau, \rho &\in \mathcal{T}^{\rightarrow} && ::= \alpha \mid (\sigma \rightarrow \tau) \\
\sigma, \rho &\in \mathcal{T}^{\wedge} && ::= \wedge\{\tau_1, \ldots, \tau_n\} \quad \text{where } n \geq 1
\end{aligned}
$$

Type Environments:

$$
\begin{aligned}
\mathcal{A} \in \mathsf{TypeEnv} &= \mathcal{P}(\mathsf{TermVar} \times \mathcal{T}^{\rightarrow}) \\
\mathcal{A}(x) &= \wedge\{\tau_1, \ldots, \tau_n\} && \text{where } \{\tau_1, \ldots, \tau_n\} = \{\, \tau \mid (x : \tau) \in \mathcal{A} \,\} \\
\mathcal{A}/x &= \{\, (y : \tau) \mid (y : \tau) \in \mathcal{A} \text{ and } y \neq x \,\} \\
\mathsf{Dom}(\mathcal{A}) &= \{\, x \mid (x : \tau) \in \mathcal{A} \,\} \\
\mathsf{Ran}(\mathcal{A}) &= \{\, \sigma \mid x \in \mathsf{Dom}(\mathcal{A}),\ \mathcal{A}(x) = \sigma \,\}
\end{aligned}
$$

Syntax for Type-Annotated Contexts and Terms:

$$
\begin{aligned}
C^{\tau} \in \mathsf{PreTContext} &::= \Box^{\tau} \mid x^{\tau} \mid (\lambda x^{\sigma}.C^{\tau'})^{\tau} \mid (C^{\tau_0}\,\{C_1^{\tau_1}, \ldots, C_n^{\tau_n}\})^{\tau} \\
M^{\tau}, N^{\tau} \in \mathsf{PreTTerm} &= \{\, C^{\tau} \in \mathsf{PreTContext} \mid C^{\tau} \text{ does not mention } \Box \,\}
\end{aligned}
$$

System $\boldsymbol{\mathcal{I}}$ (Typing Rules):

$$
\text{VAR} \quad \frac{}{\mathcal{A} \cup \{x : \tau\} \vdash x,\, x^{\tau} : \tau} \qquad\qquad \text{HOLE} \quad \frac{}{\mathcal{A} \vdash \Box,\, \Box^{\tau} : \tau}
$$

$$
\text{ABS} \quad \frac{\mathcal{A} \vdash C,\, \hat{C}^{\tau'} : \tau'; \qquad \sigma = \mathcal{A}(x) \text{ if } x \in \mathsf{Dom}(\mathcal{A}); \qquad \tau = (\sigma \rightarrow \tau')}{\mathcal{A}/x \vdash (\lambda x.C),\, (\lambda x^{\sigma}.\hat{C}^{\tau'})^{\tau} : \tau}
$$

$$
\text{APP} \quad \frac{\mathcal{A} \vdash C,\, \hat{C}^{\tau_0} : \tau_0; \qquad \forall_{i=1}^{n}.\, \mathcal{A}_i \vdash C',\, C_i^{\tau_i} : \tau_i; \qquad \tau_0 = (\wedge\{\tau_1, \ldots, \tau_n\} \rightarrow \tau)}{\mathcal{A} \cup \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n \vdash (C\,C'),\, (\hat{C}^{\tau_0}\,\{C_1^{\tau_1}, \ldots, C_n^{\tau_n}\})^{\tau} : \tau}
$$

Type-Annotated Contexts, Terms, and Parallel Contexts:

$$
\begin{aligned}
\mathsf{TContext} &= \{\, \hat{C}^{\tau} \in \mathsf{PreTContext} \mid \mathcal{A} \vdash C,\, \hat{C}^{\tau} : \tau \text{ derivable in System } \boldsymbol{\mathcal{I}} \,\} \\
\mathsf{TTerm} &= \{\, \hat{M}^{\tau} \in \mathsf{PreTTerm} \mid \mathcal{A} \vdash M,\, \hat{M}^{\tau} : \tau \text{ derivable in System } \boldsymbol{\mathcal{I}} \,\} \\
Cp^{\tau} \in \mathsf{ParContext} &= \{\, \hat{C}^{\tau} \in \mathsf{TContext} \mid \mathcal{A} \vdash C,\, \hat{C}^{\tau} : \tau \text{ derivable in System } \boldsymbol{\mathcal{I}},\ C \text{ has one hole} \,\}
\end{aligned}
$$

Typed Notions of Reduction:

Redex/contractum relations (β_{I} and β_{K} handled like β):

$$
\begin{aligned}
&((\lambda x^{\sigma}.M^{\tau})^{\tau'}\{N_1^{\tau_1}, \ldots, N_m^{\tau_m}\})^{\tau} && \rightsquigarrow_{\beta} M^{\tau}[x^{\tau_1}{:=}N_1^{\tau_1}, \ldots, x^{\tau_m}{:=}N_m^{\tau_m}] \\
&(((\lambda x^{\sigma}.N^{\tau_1})^{\tau_2}\{P_1^{\tau_1'}, \ldots, P_n^{\tau_n'}\})^{\tau_1}\{Q_1^{\tau_1''}, \ldots, Q_m^{\tau_m''}\})^{\tau_3} && \rightsquigarrow_{\mathrm{T}} ((\lambda x^{\sigma}.(N^{\tau_1}\{Q_1^{\tau_1''}, \ldots, Q_m^{\tau_m''}\})^{\tau_3})^{(\sigma \rightarrow \tau_3)}\{P_1^{\tau_1'}, \ldots, P_n^{\tau_n'}\})^{\tau_3}
\end{aligned}
$$

Contextual closure, for $X \in \{\beta, \mathrm{T}\}$:

$$
M^{\tau} \rightarrow_X N^{\tau} \iff M^{\tau} \equiv Cp^{\tau}[M_1^{\tau_1}, \ldots, M_j^{\tau_j}],\ N^{\tau} \equiv Cp^{\tau}[N_1^{\tau_1}, \ldots, N_j^{\tau_j}], \text{ and } \forall_{i=1}^{j}.\, M_i^{\tau_i} \rightsquigarrow_X N_i^{\tau_i}
$$

Recovering the Type Environment:

$$
\begin{aligned}
\mathsf{TEnv}(x^{\tau}) &= \{x : \tau\} \\
\mathsf{TEnv}((\lambda x^{\sigma}.M^{\tau})^{\tau'}) &= \mathsf{TEnv}(M^{\tau})/x \\
\mathsf{TEnv}((M^{\tau}\,\{N_1^{\tau_1}, \ldots, N_m^{\tau_m}\})^{\tau'}) &= \mathsf{TEnv}(M^{\tau}) \cup \mathsf{TEnv}(N_1^{\tau_1}) \cup \cdots \cup \mathsf{TEnv}(N_m^{\tau_m})
\end{aligned}
$$

Figure 2: System $\boldsymbol{\mathcal{I}}$.

inference but does not reduce the expressive power of the intersection type system. Instead of the traditional $\tau_1 \wedge \cdots \wedge \tau_n$, we write $\wedge\{\tau_1, \ldots, \tau_n\}$ to aid in treating $\wedge$ in an *associative*, *commutative*, and *idempotent* (ACI) manner. We say that members of $\mathcal{T}^{\rightarrow}$ are $\rightarrow$-types and members of $\mathcal{T}^{\wedge}$ are $\wedge$-types. Note that $\rightarrow$-types include type variables in addition to function types. We identify $\wedge$-types that differ only in the order of their members. By convention, whenever an intersection type $\wedge\{\tau_1, \ldots, \tau_n\}$ appears in a formal statement, we require all of $\tau_1, \ldots, \tau_n$ to be distinct. Let $\mathsf{TV}(X)$ denote the set of type variables occurring in X, where may be a type or something containing types.

A $\wedge$-type is *trivial* if it contains exactly one $\rightarrow$-type. We may omit the $\wedge\{\ \}$ notation for trivial intersection types when the context can distinguish them from $\rightarrow$-types. Thus, the type $\wedge\{\wedge\{\alpha\} \rightarrow \beta\} \rightarrow \alpha$ can be abbreviated as $\alpha \rightarrow \beta \rightarrow \alpha$.

Type environments are sets of bindings, where each binding associates a term variable with a $\rightarrow$-type. The same term variable can be associated with multiple types. The lookup function $\mathcal{A}(x)$ returns a $\wedge$-type containing all the bindings of x in $\mathcal{A}$. This representation simplifies type inference.

Figure 2 also introduces a syntax for typed (pre)contexts and (pre)terms. As in the untyped case, (pre)terms are holeless (pre)contexts. As in most explicitly typed syntaxes, variables are annotated with their types. For convenience, we also annotate all subterms with their derived type. Sometimes, we will omit some of the type annotations when the types are not relevant or can be uniquely reinserted.

An unusual feature of the explicitly typed term syntax is that the syntax for applications allows repeating the argument many times. This is necessary to account for the *finitary polymorphism* of intersection types, where a polymorphic function explicitly lists the different types at which its argument will be used. For each argument type, we need a typing derivation proving the argument to be of that type. Our explicitly typed presentation merely gives a term syntax for these multiple typing derivations of the argument. As an example, consider the term $\hat{M}^{\tau_{\mathrm{a}}}$ below, in which two distinct type-annotated versions of $\lambda x.x$ are provided as the argument to a type-annotated version of $\lambda w.ww$:

$$\begin{aligned}
\tau_{\mathrm{id1}} &= \wedge\{\alpha\} \rightarrow \alpha \\
\tau_{\mathrm{id2}} &= \wedge\{\tau_{\mathrm{id1}}\} \rightarrow \tau_{\mathrm{id1}} \\
\tau_{\mathrm{a}} &= \wedge\{\tau_{\mathrm{id2}}, \tau_{\mathrm{id1}}\} \rightarrow \tau_{\mathrm{id1}} \\
\hat{M}^{\tau_{\mathrm{a}}} &= ((\lambda w^{\wedge\{\tau_{\mathrm{id2}},\tau_{\mathrm{id1}}\}}.(w^{\tau_{\mathrm{id2}}}\{w^{\tau_{\mathrm{id1}}}\})^{\tau_{\mathrm{id1}}})^{\tau_{\mathrm{a}}} \\
&\qquad \{(\lambda x^{\wedge\{\tau_{\mathrm{id1}}\}}.x^{\tau_{\mathrm{id1}}})^{\tau_{\mathrm{id2}}}, \\
&\qquad\ (\lambda x^{\wedge\{\alpha\}}.x^{\alpha})^{\tau_{\mathrm{id1}}}\})^{\tau_{\mathrm{id1}}}
\end{aligned}$$

As with intersection types, we will sometimes omit the $\{\}$ for singleton argument sets.

A context (resp. term) is a pre-context (resp. preterm) that is typable in System $\boldsymbol{\mathcal{I}}$, the unrestricted system of intersection types. In the typing rules of System $\boldsymbol{\mathcal{I}}$ (see figure 2) the judgements are quadruples instead of the usual triples because the typing rules work simultaneously on both untyped and explicitly typed terms. We define all of the following when a judgement $\mathcal{A} \vdash M,\ \hat{M}^{\tau} : \tau$ is derivable in System $\boldsymbol{\mathcal{I}}$.

1. The *untyped* λ*-term* M is *typable* with *type environment* $\mathcal{A}$ and *derived type* τ.
2. The *(explicitly) typed term* $\hat{M}^{\tau}$ is *well typed* with the same type environment and derived type.
3. The *type erasure* of $\hat{M}^{\tau}$, written $|\hat{M}^{\tau}|$, is the untyped term M.

Here are two examples of System $\boldsymbol{\mathcal{I}}$ judgements, using τ_{id1}, τ_{id2}, τ_{a}, and $\hat{M}^{\tau_{\mathrm{a}}}$ from above:

$$\begin{aligned}
&\{w : \tau_{\mathrm{id1}}, w : \tau_{\mathrm{id2}}\} \vdash ww,\ (w^{\tau_{\mathrm{id2}}}\{w^{\tau_{\mathrm{id1}}}\})^{\tau_{\mathrm{id1}}} : \tau_{\mathrm{id1}} \\
&\varnothing \vdash (\lambda w.ww)(\lambda x.x),\ \hat{M}^{\tau_{\mathrm{a}}} : \tau_{\mathrm{a}}
\end{aligned}$$

It is easy to extract (triple-based) derivations in either the implicit or explicit typing style from a System $\boldsymbol{\mathcal{I}}$ derivation.

In System $\boldsymbol{\mathcal{I}}$, $\wedge$-introduction only occurs at arguments of applications, and so it is folded into the APP rule. This is consistent with both the type syntax, where the $\wedge$ is only used to the left of the $\rightarrow$, and the ABS rule, where the parameter must have a $\wedge$-type. Elimination of $\wedge$-types only occurs at λ-term variables, and so is absorbed into the VAR rule. The restricted placement of $\wedge$-introduction and $\wedge$-elimination simplifies type inference, but types the same set of (untyped) terms as other intersection type systems, the set of strongly β-normalizing terms [CDCV80, CDCV81, RDRV84, vB93].

The APP rule requires each of the alternatives in the argument of an application to be a typed version of the *same* untyped λ-term. This is similar to the formulation of λ^{CIL}, the flow typed intermediate language being used in the Church Project [WDMT9X][6]. Key differences are that (1) $\wedge$ in λ^{CIL} is not ACI and (2) λ^{CIL} allows intersection introduction and elimination at arbitrary positions.

In the VAR rule, the use of $\mathcal{A}$ permits the introduction of unused bindings, which are necessary for subject reduction to hold.[7] In the ABS rule, if $x \notin \mathsf{Dom}(\mathcal{A})$, then it may be assigned an arbitrary $\wedge$-type. In the absence of unused bindings, the test $x \in \mathsf{Dom}(\mathcal{A})$ distinguishes between λ_{I} and λ_{K} abstractions.

The syntax for typed applications, which allows multiple representatives of what is a single argument subterm in the untyped version, makes it impossible to use the ordinary definition of reduction on explicitly typed terms. For example, the untyped λ-term $M = (\lambda x.w(x\,P_1)(x\,P_2))(\lambda z.(\lambda y.y)z)$ β-reduces in one step to $N = (\lambda x.w(x\,P_1)(x\,P_2))(\lambda z.z)$. However, the

[6] Contains a discussion of various approaches to explicitly typed systems of intersection types.

[7] E.g. consider $\{y : \alpha, z : \gamma\} \vdash (\lambda x.y)z,\ (\lambda x^{\gamma}.y^{\alpha})z^{\gamma} : \alpha$ and $(\lambda x^{\gamma}.y^{\alpha})z^{\gamma} \rightarrow_{\beta} y^{\alpha}$. If VAR did not allow unused bindings, then we could show $\{y : \alpha\} \vdash y,\ y^{\alpha} : \alpha$ but not $\{y : \alpha, z : \gamma\} \vdash y,\ y^{\alpha} : \alpha$. But the latter's unused binding $(z : \gamma)$ is needed for subject reduction.

typed term corresponding to M might be (including only the important types):

$$\begin{aligned}\hat{M} = \ & (\lambda x^{\wedge\{\wedge\{\tau_1\}\to\tau_1, \wedge\{\tau_2\}\to\tau_2\}}.w(x\,\hat{P}_1)(x\,\hat{P}_2)) \\ & \{(\lambda z^{\wedge\{\tau_1\}}.(\lambda y.y)z), (\lambda z^{\wedge\{\tau_2\}}.(\lambda y.y)z)\}\end{aligned}$$

It takes *two* ordinary β-reduction steps to transform $\hat{M}$ into the term which corresponds to N:

$$\begin{aligned}\hat{N} = \ & (\lambda x^{\wedge\{\wedge\{\tau_1\}\to\tau_1, \wedge\{\tau_2\}\to\tau_2\}}.w(x\,\hat{P}_1)(x\,\hat{P}_2)) \\ & \wedge\{(\lambda z^{\wedge\{\tau_1\}}.z), (\lambda z^{\wedge\{\tau_2\}}.z)\}\end{aligned}$$

Furthermore, if these steps are performed sequentially, the intermediate result is ill-typed and corresponds to *no* untyped λ-term. To solve this problem, in figure 2 *parallel contexts* (members of the set $\mathsf{ParContext}$) are used in defining the contextual closure of reduction to force each reduction step at the typed level to correspond to a reduction step at the untyped level [KW94, WDMT9X]. Additionally, the typed β-reduction rule models a single untyped β step by simultaneously substituting each typed argument representative into the occurrences of the bound variable that share the same type.

An explicitly typed term contains the entire information of a derivation which proves it to be well typed except for any unused type assumptions in the type environment. This result is formalized in the following theorem, which uses the TEnv function defined in figure 2 to recover a type environment from a term.

Theorem 2.1 (Typed Terms are Derivations). *If $\hat{M}^\tau \in \mathsf{TTerm}$, then $\mathsf{TEnv}(\hat{M}^\tau) \vdash |\hat{M}^\tau|, \hat{M}^\tau : \tau$ is derivable in System $\boldsymbol{\mathcal{I}}$.* □

Theorem 2.1 allows us to identify typed terms with typings in which every type assumption is used. System $\boldsymbol{\mathcal{I}}$ also enjoys other important properties:

Theorem 2.2 (Subject Reduction). *Let X range over $\{\beta, T\}$. If $\mathcal{A} \vdash M, \hat{M}^\tau : \tau$ holds then (1) $M \to_X N$ implies the existence of an $\hat{N}^\tau$ such that $\hat{M}^\tau \to_X \hat{N}^\tau$ and $\mathcal{A} \vdash N, \hat{N}^\tau : \tau$ holds; (2) $\hat{M}^\tau \to_X \hat{N}^{\tau'}$ implies that $\tau' = \tau$, $\mathcal{A} \vdash |\hat{N}^\tau|, \hat{N}^\tau : \tau$, and $|\hat{M}^\tau| \to_X |\hat{N}^\tau|$.* □

Theorem 2.3 (Strong Normalization is Typability). *M is typable in System $\boldsymbol{\mathcal{I}}$ iff M is strongly β-normalizing, i.e., there are no infinite β-reduction paths starting at M.* □

By Theorem 2.3, the term $(\lambda w.ww)(\lambda x.x)$ is typable in System $\boldsymbol{\mathcal{I}}$ because its only β-reduct is $(\lambda x.x)$; we saw a typing for this term above. But $(\lambda w.ww)(\lambda x.xx)$ is not typable in System $\boldsymbol{\mathcal{I}}$ because it has no β-normal form.

We introduce the definitions for the *rank* of types and explicitly typed terms in figure 3. The rank of a type measures how "deeply" non-trivial $\wedge$-types are nested to the left of $\to$'s. The rank of a $\to$-type is 0 if all $\wedge$-types occurring within it are trivial. The rank of a $\wedge$-type is 0 if it is a set containing a single $\to$-type with rank 0, and it is 1 if it is a set containing two or more $\to$-types each with rank 0. Otherwise, the rank of either a $\to$-type or a $\wedge$-type is one more than the maximum number of "lefts" taken at arrows in all paths from the root of the type to non-trivial $\wedge$-types occuring within the type. Rank 0 types correspond to the traditional notion of *simple types*. The notion of rank extends to typed terms. Given a typed term $\hat{M}^\tau$, the *environment rank* of a bound variable is the rank of the $\wedge$-type annotating the abstraction parameter binding the variable, and the environment rank of a free variable z is the rank of the $\wedge$-type containing all the $\to$-types at which z is used in $\hat{M}^\tau$. The environment rank of $\hat{M}^\tau$ is the maximum of the environment ranks of all its free and bound variables. The rank of $\hat{M}^\tau$ is 0 if its environment rank is 0, and otherwise is one more than its environment rank. Note that the rank of a typed term is never 1. Rank 0 terms correspond to the traditional notion of *simply typable terms*. It can be shown that if ρ is a $\wedge$-type or $\to$-type occurring in $\hat{M}^\tau$, then $\mathsf{rank}(\rho) \le rank(\hat{M}^\tau)$.

Given an untyped term M, define $\mathsf{minrank}(M)$ as the rank of the minimum rank typing for M (if one exists). For example:

term	minrank	term	minrank
$I = (\lambda x.x)$	0	$(II)R_i$	$2i+2$
$R_1 = (\lambda w.ww)$	2	$(\lambda z.z(IR_1)R_2)$	4
$R_i = (\lambda y.yR_{i-1})$	$2i$	$(\lambda z.z(IR_2)R_1)$	5
IR_i	$2i+1$	(R_1R_1)	undefined

Our definition of rank is equivalent to others found in the literature [Lei83, Jim96, vB93, KT92], except that we consider the rank of typed terms instead of the rank of derivations, and the rank of a typed term is not affected by unused type assumptions.

Auxiliary Functions:

$$\begin{aligned}&\mathsf{inc}(0) = 0 && \mathsf{dec}(0) = 0 \\ &\mathsf{inc}(n) = n+1 \ \text{, if } n \ge 1 && \mathsf{dec}(n) = n-1 \ \text{, if } n \ge 1\end{aligned}$$

Rank of Types:

$$\begin{aligned}\mathsf{rank}(t) &= 0 \\ \mathsf{rank}(\sigma \to \tau) &= \max\{\mathsf{inc}(\mathsf{rank}(\sigma)), \mathsf{rank}(\tau)\} \\ \mathsf{rank}(\wedge\{\tau\}) &= \mathsf{rank}(\tau) \\ \mathsf{rank}(\wedge\{\tau_1, \ldots, \tau_n\}) &= \max\{1, \mathsf{rank}(\tau_1), \ldots, \mathsf{rank}(\tau_n)\} \quad \text{, if } n \ge 2 \\ \mathsf{rank}(\{\sigma_1, \ldots, \sigma_n\}) &= \max\{\mathsf{rank}(\sigma_1), \ldots, \mathsf{rank}(\sigma_n)\}\end{aligned}$$

Extracting Variable Types from Typed Terms:

$$\begin{aligned}\mathsf{bind\text{-}types}(x^\tau) &= \varnothing \\ \mathsf{bind\text{-}types}((\lambda x^\sigma.\hat{M}^{\tau'})^\tau) &= \{\sigma\} \cup \mathsf{bind\text{-}types}(\hat{M}^{\tau'}) \\ \mathsf{bind\text{-}types}((\hat{M}^{\tau'}\,\{\hat{N}_1^{\tau_1}, \ldots, \hat{N}_m^{\tau_m}\})^\tau) &= \mathsf{bind\text{-}types}(\hat{M}^{\tau'}) \cup \textstyle\bigcup_{1 \le i \le m} \mathsf{bind\text{-}types}(\hat{N}_i^{\tau_i}) \\ \mathsf{env\text{-}types}(\hat{M}^\tau) &= \mathsf{bind\text{-}types}(\hat{M}^\tau) \cup \mathsf{Ran}(\mathsf{TEnv}(\hat{M}^\tau))\end{aligned}$$

Rank of Typed Terms:

$$\mathsf{rank}(\hat{M}^\tau) = \mathsf{inc}(\mathsf{rank}(\mathsf{env\text{-}types}(\hat{M}^\tau)))$$

Figure 3: Definitions of rank used for System $\boldsymbol{\mathcal{I}}_k$.

The restriction of System $\mathcal{I}$ to finite-rank k, called System $\mathcal{I}_k$, is the restriction deriving only judgements of the form $\mathcal{A} \vdash M, \hat{M}^\tau : \tau$ where $\mathsf{rank}(\hat{M}^\tau) \leq k$. Theorems 2.1 and 2.2 carry over to System $\mathcal{I}_k$, but theorem 2.3 does not.

3 A Kalmar-elementary Upper Bound for System $\mathcal{I}_k$ Typability

This section establishes a Kalmar-elementary upper bound for the complexity of type inference in System $\mathcal{I}_k$ (for every k). The type inference problem for System $\mathcal{I}_k$ is as follows: given an untyped term M of size n, either find a typed term $\hat{M}^\tau$ in System $\mathcal{I}_k$ such that $|\hat{M}^\tau| = M$ or determine that such a term does not exist.

Theorem 3.1 (Upper bound for System $\mathcal{I}_k$ Typability). *The type inference problem for System $\mathcal{I}_k$ is in* $\mathrm{DTIME}[\mathbf{K}(k-1, p(n))]$, *where n is the size of the given term, and p is a polynomial function.* □

We prove the theorem by developing a naive type inference algorithm. Our main criteria for this algorithm is that we can analyze its time complexity and show that it has a Kalmar-elementary upper bound.

Our type inference algorithm for System $\mathcal{I}_k$ is inspired by the following idea. The **let**-style polymorphism supported in Hindley-Milner type systems is equivalent to inlining the **let**-bound definitions and typing the resulting program with simple types. Because this expands the program at most exponentially [Mit96] and type inference for simple types takes polynomial time, this implementation of Hindley-Milner polymorphism gives an exponential-time type inference algorithm and thus an upper bound on the complexity of Hindley-Milner type inference. We extend this strategy to type inference for finite-rank intersection types. Along the way, we encounter a few snags, but the basic approach works.

Our type inference algorithm $\mathsf{Infer}\text{-}\mathcal{I}(k)$ for System $\mathcal{I}_k$ proceeds in three stages. First, we reduce the problem of System $\mathcal{I}_k$ type inference for an arbitrary untyped term to a restricted form of type inference for a "simplified" (and potentially much larger) term. Given a untyped term M of size n, we apply a transformation $\mathsf{Simplify}(k)$ to obtain a untyped term N whose size is less than $\mathbf{K}(k-1, p(n))$, where p is a polynomial function. This transformation preserves System $\mathcal{I}_k$ typability (with any derived type and type environment). Furthermore, the minimal-rank typing $\hat{N}^\tau$ of N satisfies a predicate $\mathsf{B\text{-}Simple}_k$ iff M is typable in System $\mathcal{I}_k$. Second, we apply a polynomial-time type inference algorithm $\mathsf{B\text{-}Typing}$ to N that yields (if it exists) a System $\mathcal{I}_k$ typing $\hat{N}^\tau$ for N such that $\mathsf{B\text{-}Simple}_k(\hat{N}^\tau)$. If $\mathsf{B\text{-}Typing}$ fails on N, then M is not typable in System $\mathcal{I}_k$. Finally, if $\mathsf{B\text{-}Typing}$ yields a typing $\hat{N}^\tau$, then the typing can be pulled back through $\mathsf{Simplify}(k)$ to yield a System $\mathcal{I}_k$ typing for M. This algorithm runs in time $O(\mathbf{K}(k-1, p(\mathsf{size}(M))))$. Section 4 shows that this bound is tight: there are terms for which type inference must take Kalmar-elementary time.

3.1 The Simplification Stage

Intuitively, a term typable in System $\mathcal{I}_k$ harbors a certain "potential" for finitary polymorphism that can be reduced whenever a β reduction copies a polymorphic argument. For example, consider the reduction sequence $M_1 \rightarrow_\beta M_2 \twoheadrightarrow_\beta M_3$, with M_1, M_2, and M_3 as follows:

$$\begin{aligned}
M_1 &\equiv (\lambda f.v(f\lambda x.x)(f(\lambda y.\lambda z.y)))(\lambda w.ww) \\
M_2 &\equiv v((\lambda w.ww)(\lambda x.x))((\lambda w.ww)(\lambda y.\lambda z.y)) \\
M_3 &\equiv v((\lambda x.x)(\lambda x.x))((\lambda y.\lambda z.y)(\lambda y.\lambda z.y))
\end{aligned}$$

It holds that $\mathsf{minrank}(M_1) = 3$, $\mathsf{minrank}(M_2) = 2$, and $\mathsf{minrank}(M_3) = 0$. In this example, the minimum rank shrinks with each reduction as polymorphic functions like $\lambda w.ww$, $\lambda x.x$, and $\lambda y.\lambda z.y$ are duplicated into monomorphic copies.

We will construct $\mathsf{Simplify}(k)$ out of a sequence of simpler $\mathsf{Simplify1}$ transformations. $\mathsf{Simplify1}$ takes advantage of the above-described decrease in polymorphism due to β-reduction. Because β-reduction alone is insufficient, the $\mathsf{Simplify1}$ algorithm has two major components: (1) reducing to normal form w.r.t. the let-lifting transformation T and (2) performing the complete development of all β_{I}-redexes.

Let us first review the notion of complete development. A *development* of a family $\mathcal{F}$ of redexes in a term M is a reduction sequence beginning with M contracting only redexes which are in $\mathcal{F}$ or are residuals of redexes in $\mathcal{F}$. This can be viewed as marking the redexes in $\mathcal{F}$ and then only allowing contractions of marked redexes. A *complete development* of a redex family leaves no marked redexes in its result. It is well known that the complete development of any β-redex family $\mathcal{F}$ in the λ-calculus ends in a uniquely determined term [Bar84]. Let $\mathsf{CD}_{\beta_{\mathrm{I}}}$ be the function such that $\mathsf{CD}_{\beta_{\mathrm{I}}}(M)$ is the result of a complete development of the family of all β_{I}-redexes in M.

The notion of T-reduction is necessary because merely performing a complete development of all β_{I}-redexes in a term will not always lower the minimal rank. For example, consider $M = ((\lambda x.\lambda y.xyy)w)(\lambda z.z)$, which has $\mathsf{minrank}(M) = 2$, and $N = \mathsf{CD}_{\beta_{\mathrm{I}}}(M) = (\lambda y.wyy)(\lambda z.z)$, which also has $\mathsf{minrank}(N) = 2$. In this case, it is necessary to perform a second complete development, $P = \mathsf{CD}_{\beta_{\mathrm{I}}}(N) = w(\lambda z.z)(\lambda z.z)$, to reduce the minimal rank to $\mathsf{minrank}(P) = 0$. The problem in this example is that M has an "implicit" redex of $(\lambda y.xyy)$ applied to $(\lambda \mathrm{z.z})$ that is blocked until the outermost redex of M is contracted. The function and argument of such an implicit redex are *companions*.

The purpose of the let-lifting transformation $((\lambda x.N)P)Q \rightarrow_{\mathrm{T}} (\lambda x.NQ)P$ is to join the companions of an implicit β-redex in an explicit one. It works by moving Q, the potential argument companion of an implicit redex, through the blocking redex $(\lambda x.\ldots)P$ to become the argument of N, where it is closer to any function companion that might be in N. It is easy to show that T-reduction is both confluent and strongly normalizing, so that repeated T-reductions will eventually expose all implicit redexes. Let T-nf denote the function such that $\mathsf{T\text{-}nf}(M)$ is the (unique) T-normal form of M. See [KW95] for a discussion of similar notions of reduction in the literature. There is a close connection with notions of *generalized β-reduction* which can contract the implicit redexes directly [Kam96, KRW98].

Making all implicit redexes into explicit redexes serves two major purposes in System $\boldsymbol{\mathcal{I}}_k$ type inference: (1) it allows postponing β_{K}-redexes forever, and (2) it makes a subsequent complete development of all β_{I}-redexes reduce (a component of) the rank of a System $\boldsymbol{\mathcal{I}}_k$ typing. Purpose (1) is important because contracting β_{K}-redexes can alter typability by discarding an untypable argument. For example, $(\lambda x.y)(\lambda w.ww)$ is not typable in System $\boldsymbol{\mathcal{I}}$, but it β_k-reduces to the simply-typable term y. As an example of purpose (2), reconsider the term M from above. Observe that $\mathsf{T\text{-}nf}(M) = M' = (\lambda x.(\lambda y.xyy)(\lambda z.z))w$ and $\mathsf{CD}_{\beta_{\mathrm{I}}}(M') = N' = w(\lambda z.z)(\lambda z.z)$ and notice that $\mathsf{minrank}(M') = 2$ but $\mathsf{minrank}(N') = 0$.

Define $\mathsf{Simplify1} = \mathsf{CD}_{\beta_{\mathrm{I}}} \circ \mathsf{T\text{-}nf}$. Because Simplify1 is composed out of β_{I}- and T-reduction steps, it is defined on both untyped and typed terms. Based on the above discussion, we might hope that Simplify1 lowers the minimal rank of a term. Unfortunately, free variables, λ_{K} abstractions, and unapplied λ_{I} abstractions contribute to the rank of a typed term but are not removed by Simplify1. For example, Simplify1 acts as the identity on the following terms, even though they have non-zero minimal rank:

$$
\begin{array}{ll}
M_4 = (\lambda w.ww) & \mathsf{minrank}(M_4) = 2 \\
M_5 = (\lambda x.y)(\lambda w.ww) & \mathsf{minrank}(M_5) = 3 \\
M_6 = w(\lambda x.x)(w(\lambda y.\lambda z.y)) & \mathsf{minrank}(M_6) = 2
\end{array}
$$

Extracting Types of Variables from a Typed Term:

$$
\begin{array}{l}
\mathsf{AB}(i, x^\tau) \;=\; \varnothing \\
\mathsf{AB}(0, (\lambda x^\sigma.\hat{M}^{\tau'})^\tau) \;=\; \{(\mathrm{B},\sigma)\} \cup \mathsf{AB}(0, \hat{M}^{\tau'}) \\
\mathsf{AB}(i+1, (\lambda_I x^\sigma.\hat{M}^{\tau'})^\tau) \;=\; \{(\mathrm{A},\sigma)\} \cup \mathsf{AB}(i, \hat{M}^{\tau'}) \\
\mathsf{AB}(i+1, (\lambda_K x^\sigma.\hat{M}^{\tau'})^\tau) \;=\; \{(\mathrm{B},\sigma)\} \cup \mathsf{AB}(i, \hat{M}^{\tau'}) \\
\mathsf{AB}(i, (\hat{M}^{\tau'}\ \{\hat{N}_1^{\tau_1}, \ldots, \hat{N}_m^{\tau_m}\})^\tau) \;= \\
\qquad \mathsf{AB}(i+1, \hat{M}^{\tau'}) \cup \bigcup_{1 \le i \le m} \mathsf{AB}(0, \hat{N}_i^{\tau_i})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{As}(\hat{M}^\tau) = \{\, \sigma \mid (\mathrm{A},\sigma) \in \mathsf{AB}(0, \hat{M}^\tau) \,\} \\
\mathsf{Bs}(\hat{M}^\tau) = \{\, \sigma \mid (\mathrm{B},\sigma) \in \mathsf{AB}(0, \hat{M}^\tau) \,\} \cup \mathsf{Ran}(\mathsf{TEnv}(\hat{M}^\tau))
\end{array}
$$

Rank of Typed Terms:

$$
\begin{array}{c}
\mathsf{A\text{-}rank}(\hat{M}^\tau) = \mathsf{inc}(\mathsf{rank}(\mathsf{As}(\hat{M}^\tau))) \\
\mathsf{B\text{-}rank}(\hat{M}^\tau) = \mathsf{inc}(\mathsf{rank}(\mathsf{Bs}(\hat{M}^\tau)))
\end{array}
$$

Figure 4: Definitions of A-rank and B-rank.

To show that Simplify1 somehow makes type inference simpler, we decompose the rank of a typed term into components we call *A-rank* and *B-rank*, computed by functions A-rank and B-rank defined in figure 4. The A-rank of a typed term summarizes the contribution of (possibly implicitly) $\mathbf{let}_{\mathrm{I}}$-bound variables to the rank, while the B-rank summarizes the contribution of all other variables. The types of λ-bound variables are partitioned by the function AB, which maintains a count of potential unpaired argument companions as it descends into a term. Any λ_{I} abstraction encountered while the count is a non-zero is the operator of an explicit or implicit β_{I} redex, and its parameter type is marked as contributing to the A-rank. All other variable declarations and free variables contribute to the B-rank. It holds that the A-rank and B-rank of a typed term are preserved by T-reduction and that $\mathsf{rank}(\hat{M}^\tau) = \max\{\mathsf{A\text{-}rank}(\hat{M}^\tau), \mathsf{B\text{-}rank}(\hat{M}^\tau)\}$. Here are some examples of terms with their minimal ranks and the A-ranks and B-ranks of their minimally ranked typings:

term	minrank	A-rank	B-rank
$I = (\lambda x.x)$	0	0	0
$R_1 = (\lambda w.ww)$	2	0	2
$R_2 = (\lambda y.yR_1)$	4	0	4
$(\lambda y.yI)R_1$	3	3	2
$(\lambda z.z(IR_1)R_2)$	4	4	3
$(\lambda z.z(IR_2)R_1)$	5	5	4

The following lemma shows that A-rank, not rank, is the real measure of the polymorphic 'potential" of a derivation reduced by every application of Simplify1.

Lemma 3.2.

1. *If $\hat{M}^\tau$ is a typed term in System $\boldsymbol{\mathcal{I}}$, then* $\mathsf{A\text{-}rank}(\hat{M}^\tau) \ge \mathsf{inc}(\mathsf{A\text{-}rank}(\mathsf{Simplify1}(\hat{M}^\tau)))$, $\mathsf{B\text{-}rank}(\hat{M}^\tau) \ge \mathsf{B\text{-}rank}(\mathsf{Simplify1}(\hat{M}^\tau))$, *and* $\mathsf{size}(\mathsf{Simplify1}(\hat{M}^\tau)) \le 2^{p(\mathsf{size}(\hat{M}^\tau))}$, *where p is a polynomial function.*

2. *If* $\mathsf{Simplify1}(M) = N$, *$\hat{N}^\tau$ is a typed term in System $\boldsymbol{\mathcal{I}}$, and $|\hat{N}^\tau| \equiv N$, then there is a well typed term $\hat{M}^\tau$ such that $|\hat{M}^\tau| \equiv M$ and* $\mathsf{Simplify1}(\hat{M}^\tau) = \hat{N}^\tau$. *Moreover, $\hat{M}^\tau$ can be constructed in time polynomial in the size of $\hat{N}^\tau$.* □

Let the predicate $\mathsf{B\text{-}Simple}_k(\hat{M}^\tau)$ hold iff $\mathsf{A\text{-}rank}(\hat{M}^\tau) = 0$ and $\mathsf{B\text{-}rank}(\hat{M}^\tau) \le k$. Let $\mathsf{Simplify}(k) = \mathsf{Simplify1}^{k-1}$ (the $(k-1)$-fold self-composition of Simplify1). Because $\mathsf{Simplify}(k)$ is composed out of β_{I}- and T-reduction steps, it is defined on both untyped and typed terms. The reason for $k-1$ rather than k in the definition of $\mathsf{Simplify}(k)$ is an artifact of arithmetic on ranks — simplifying a rank-2 derivation reduces the A-rank to 0 in one step.

Theorem 3.3.

1. *If $\hat{M}^\tau$ is a typed term in System $\boldsymbol{\mathcal{I}}$, then (1)* $\mathsf{B\text{-}Simple}_k(\mathsf{Simplify}(k)(\hat{M}^\tau))$ *iff $\hat{M}^\tau$ is a term in System $\boldsymbol{\mathcal{I}}_k$ and (2)* $\mathsf{size}(\mathsf{Simplify}(k)(\hat{M}^\tau)) < \mathbf{K}(k-1, p(\mathsf{size}(\hat{M}^\tau)))$, *where p is a polynomial function.*

2. *If* Simplify$(k)(M) = N$, $\hat{N}^\tau$ *is a typed term in System* $\boldsymbol{\mathcal{I}}_k$, $|\hat{N}^\tau| = N$, *and* B-Simple$_k(\hat{N}^\tau)$, *then there is a typed term* $\hat{M}^\tau$ *in System* $\boldsymbol{\mathcal{I}}_k$ *such that* $|\hat{M}^\tau| \equiv M$ *and* Simplify$(k)(\hat{M}^\tau) = \hat{N}^\tau$. *Moreover,* $\hat{M}^\tau$ *can be constructed in time polynomial in the size of* $\hat{N}^\tau$. □

This theorem justifies the use of Simplify(k) in System $\boldsymbol{\mathcal{I}}_k$ type inference. M is typable in System $\boldsymbol{\mathcal{I}}_k$ iff the expanded term Simplify$(k)(M)$ has a System $\boldsymbol{\mathcal{I}}_k$ typing that is B-Simple$_k$. Morever, any typing inferred for Simplify$(k)(M)$ can be pulled back through Simplify(k) to get a typing for N.

3.2 The Type Inference Stage

We have developed an algorithm B-Typing that, for a given k, is a type inference algorithm for the subset of untyped terms with typings in System $\boldsymbol{\mathcal{I}}_k$ that satisfy the predicate B-Simple$_k$. Given a term M in T-nf, the invocation B-Typing(M, k) either succeeds with a System $\boldsymbol{\mathcal{I}}_k$ typed term whose erasure is M or it fails.

Space limitations preclude us from presenting the algorithm in detail. Instead, we give a brief sketch of the algorithm and summarize its properties.

3.2.1 Sketch of Algorithm B-Typing

B-Typing(M, k) is defined in terms of the auxiliary algorithm BT(N, e, d, k), whose parameters have the following meaning:

- N is a subterm of the term M.
- e is a *rank environment* that maps each free variable in N to the upper bound on the rank that can be inferred for the binding type of the variable. We shall call this quantity the env-rank of the variable. If $e(x) = 0$, then all occurrences of x are constrained to have the same simple type. If $e(x) = 1$, then all occurrences of x must have simple types, but they need not be the same. If $e(x) = h \geq 2$, then all occurrences of x must have types whose rank is less than h. The initial rank environment e_{init} binds all free variables of the top-level term M to dec(k).
- d is an upper bound on the rank of the derived type inferred for N.
- k is the upper bound on the B-rank of N, as well as on the whole term M. This parameter never changes.

The call BT(N, e, d, k) either fails, or it succeeds by returning a typed term $\hat{N}^\tau$ whose erasure is N and which satisfies the following properties: (1) rank$(\tau) \leq d$; and (2) rank(TEnv$(\hat{N}^\tau)(x)) \leq e(x)$, for all x in FV(M)

The BT algorithm extends the bottom-up type inference algorithm for simple types (such as algorithm PT in [Mit96]) to handle non-zero B-rank. At an application site (including $\mathbf{let}_{\mathrm{I}}$ and $\mathbf{let}_{\mathrm{K}}$), the types inferred for free variables whose env-rank is 0 and that also appear in both the operator and operand must be unified, just as in the algorithm for simple type inference. However, the types inferred for a free variable x having an env-rank rank greater than 0 are *not* unified; instead, they are implicitly unioned together in the resulting typed term, and later appear as components of an intersection type in the enclosing $\lambda_{\mathrm{I}} x$ binding construct.

3.2.2 Properties of Algorithm B-Typing

The important properties of algorithm B-Typing are summarized in the following theorem:

Theorem 3.4.

1. B-Typing(M, k) *succeeds with a System* $\boldsymbol{\mathcal{I}}_k$ *typed term* $\hat{M}^\tau$ *such that* $|\hat{M}^\tau| = M$ *iff* B-Simple$_k(M)$ *holds.*
2. B-Typing(M, k) *succeeds or fails in time polynomial in* size(M). □

B-Typing runs in polynomial time because it is a *monovariant* analysis, i.e., it analyzes each subterm at most once. The intuition behind why B-Typing can be monovariant is the following. If B-Simple$_k(\hat{M}^\tau)$ holds, then there is a well typed term $\hat{N}^\tau$ such that B-Simple$_k(\hat{N}^\tau)$ holds, $|\hat{M}^\tau| \equiv |\hat{N}^\tau|$, and every use of the APP rule in deriving $\hat{N}^\tau$ can be replaced by a use of the following restricted rule:

$$\frac{\mathcal{A} \vdash M,\ \hat{M}^\tau : \tau;\ \mathcal{A}' \vdash N,\ \hat{N}^{\tau_1} : \tau_1;\ \tau = (\wedge\{\tau_1\} \to \tau_2)}{\mathcal{A} \cup \mathcal{A}' \vdash (M\, N),\ (\hat{M}^\tau \{\hat{N}^{\tau_1}\})^{\tau_2} : \tau_2}$$

In other words, even though the typing $\hat{N}^\tau$ may mention non-trivial $\wedge$-types, each application argument need be typed with only one subderivation. For example, the types of the terms R_1, and R_2 introduced at the end of Section 2 can have typings inferred by B-Typing even though they are not simply typable.

The fact that System $\boldsymbol{\mathcal{I}}_k$ type inference takes polynomial time for B-Simple$_k$ terms but can take Kalmar-elementary time in general indicates that requiring an A-rank of 0 is a strong constraint and that the high complexity of general type inference for System $\boldsymbol{\mathcal{I}}_k$ results from performing multiple typing derivations for the terms that are $\mathbf{let}_{\mathrm{I}}$-bound to variables.

4 Recognizing Typable System $\boldsymbol{\mathcal{I}}_k$ Terms is Kalmar-elementary

Given an untyped λ-term of size n, how hard is it to determine whether the term has a type in System $\boldsymbol{\mathcal{I}}_k$? We prove that this problem is DTIME$[\mathbf{K}(k-1, n)]$-hard, by showing the following:

Theorem 4.1 (Hardness). *Let* $\mathcal{M}$ *be a deterministic Turing Machine accepting or rejecting its input* x *of size* n *in* $\mathbf{K}(k-1, n)$ *steps. Then we can construct in* $O(\log n)$ *space a closed, typed term* $P_{\mathcal{M},x}$ *in System* $\boldsymbol{\mathcal{I}}_k$, *such that $P_{\mathcal{M},x}$ reduces to true $= \lambda x.\lambda y.x : \alpha \to \beta \to \alpha$ if $\mathcal{M}$ accepts x, and reduces to false $= \lambda x.\lambda y.y : \alpha \to \beta \to \beta$ if $\mathcal{M}$ rejects x. In addition, there exists a fixed context $\hat{C}$ not depending on $\mathcal{M}$ or x such that $\hat{C}[P_{\mathcal{M},x}]$ is typable in System $\boldsymbol{\mathcal{I}}_k$ iff $\mathcal{M}$ accepts x.* □

Following the earlier, similar proof for ML and bounded-kind generalizations of System F [KHM94, HM94], we are thus able to simulate computations faithfully "in the types." The key idea is that, in the λ-calculus simulation of circuits, the terms are essentially *linear*—arguments are not used more than once. The correspondence with linear logic is most likely more than surface-deep, since the idea of *fanout*, implemented by explicit copying, is simulating the function of exponentials and sharing.

Combining the lower bound of the hardness result with the upper bound of the previous section yields:

Theorem 4.2 (Completeness). *Recognizing the untyped terms of size n that are typable in System $\boldsymbol{\mathcal{I}}_k$ is complete for* DTIME$[\mathbf{K}(k-1,n)]$. □

4.1 The Polymorphic Iteration Lemma

The central problem in deriving this result is to transfer the simulations in [HM94] for F_k, the generalization of System F with k-th order type constructors, to the much simpler intersection type discipline. The key technical construction needed to derive this lower bound is the following:

Lemma 4.3 (Polymorphic Iteration). *Let $N = \mathbf{K}(k-1,n)$, $\overline{m}$ denote the Church numeral $\lambda s.\lambda z.s^m z$, and let Φ be a simply-typable term that can be given any of the simple types $\tau(i) \to \tau(i+1)$ for $0 \le i < N$, where the $\tau(i)$ are arbitrary. Let polyit$_{n,k} = (\lambda z.\overline{2}^n z)\, \underbrace{\overline{2}\,\overline{2}\cdots\overline{2}}_{k-2}$.*

Then the term (polyit$_{n,k}\,\Phi$) normalizes to $\lambda x.\Phi^N x$, and has simple type $\tau(0) \to \tau(N)$ in System $\boldsymbol{\mathcal{I}}_k$. □

The finite-rank intersection type systems satisfy subject reduction, and observe that term $\overline{n}\, \underbrace{\overline{2}\,\overline{2}\cdots\overline{2}}_{k-1}\,\Phi$, which reduces to $(\lambda z.\overline{2}^n z)\, \underbrace{\overline{2}\,\overline{2}\cdots\overline{2}}_{k-2}\,\Phi = \mathit{polyit}_{n,k}\,\Phi$ by contracting $\overline{n}\,\overline{2}$ to $\lambda z.\overline{2}^n z$, further reduces to $\lambda x.\Phi^N x$, where each *polymorphic* instance of Φ has a different type. By substituting a term coding the transition function of a Turing Machine for Φ, we can simulate any computation in DTIME$[\mathbf{K}(k-1,n)]$ dually in terms and in the type system. By substituting the exponentiation function $\lambda x.x\overline{2}$ for Φ, we note that $\mathit{polyit}_{n,k}\,(\lambda x.x\overline{2})\,\overline{2}$ is a term of size $O(n)$ that reduces to the astronomical $\overline{\mathbf{K}(\mathbf{K}(k-1,n),2)}$—a number that would have made even Carl Sagan blush. By instead substituting the *powerset* function for Φ, we can (after a little technical work) simulate any computation in DTIME$[\mathbf{K}(\mathbf{K}(k-1,n),2)]$ by a term having the simple type $\mathsf{Bool} = \alpha \to \alpha \to \alpha$; our later expressiveness results follow from this construction.

Proof. Some abbreviations: if $\sigma' = \wedge\{\tau_1,\ldots,\tau_m\}$, we write $\sigma \Rightarrow \sigma'$ for $\wedge\{\sigma \to \tau_1,\ldots,\sigma \to \tau_m\}$; we write $\wedge\{\sigma_1 \Rightarrow \sigma_1',\ldots,\sigma_n \Rightarrow \sigma_n'\}$ for $\wedge\{\tau | \tau \in \sigma_i \Rightarrow \sigma_i', 1 \le i \le n\}$; we say closed term M has type $\wedge\{\tau_1,\ldots,\tau_m\}$ if $\vdash M, \hat{M}^{\tau_i} : \tau_i$ is derivable for each τ_i.

To prove the lemma, we begin by defining the following rank-1 types:

$$\begin{aligned}
\sigma(1,0) &= \wedge\{\, \tau(0) \to \tau(1), \tau(1) \to \tau(2), \\
&\qquad \ldots, \tau(N-1) \to \tau(N)\} \\
&\cdots \\
\sigma(1,j) &= \wedge\{\, \tau(0) \to \tau(2^j), \tau(2^j) \to \tau(2\cdot 2^j), \\
&\qquad \ldots, \tau(N-2^j) \to \tau(N)\} \\
&\cdots \\
\sigma(1,\log N) &= \wedge\{\tau(0) \to \tau(N)\}
\end{aligned}$$

Observe that Φ has type $\sigma(1,0)$ and that for every $0 \le j < \log N$, the Church numeral $\overline{2}$ has type $\wedge\{\sigma(1,j) \Rightarrow \sigma(1,j+1)\}$. Inductively, we then define a set of rank $t+1$ types $\sigma(t+1,j)$ for $1 \le t \le k-2$:

$$\begin{aligned}
\sigma(t+1,0) &= \wedge\{\, \sigma(t,0) \Rightarrow \sigma(t,1), \\
&\qquad \sigma(t,1) \Rightarrow \sigma(t,2), \ldots, \\
&\qquad \sigma(t,\log^{(t)} N - 1) \Rightarrow \sigma(t,\log^{(t)} N)\} \\
&\cdots \\
\sigma(t+1,j) &= \wedge\{\, \sigma(t,0) \Rightarrow \sigma(t,2^j), \\
&\qquad \sigma(t,2^j) \Rightarrow \sigma(t,2\cdot 2^j), \ldots, \\
&\qquad \sigma(t,\log^{(t)} N - 2^j) \Rightarrow \sigma(t,\log^{(t)} N)\} \\
&\cdots \\
\sigma(t+1, \log^{(t+1)} N) &= \wedge\{\sigma(t,0) \Rightarrow \sigma(t,\log^{(t)} N)\}
\end{aligned}$$

(We write $\log^{(i)}$ for the i-th iterated logarithm.) Notice that by induction, $\overline{2}$ has type $\sigma(t+1,0)$. Finally, at rank k, we define only

$$\begin{aligned}
\sigma(k,0) &= \wedge\{\, \sigma(k-1,0) \Rightarrow \sigma(k-1,1), \\
&\qquad \sigma(k-1,1) \Rightarrow \sigma(k-1,2), \ldots, \\
&\qquad \sigma(k-1,n-1) \Rightarrow \sigma(k-1,n)\} \\
\sigma(k,\log n) &= \wedge\{\sigma(k-1,0) \Rightarrow \sigma(k-1,n)\}
\end{aligned}$$

(Recall that $\log^{(k-1)} N = n$.) Again, $\overline{2}$ has type $\sigma(k,0)$. Now we make the following straightforward observations: that $\overline{n}$ has rank $k+1$ type $\sigma(k,0) \Rightarrow \sigma(k,\log n)$, and that

$$\begin{aligned}
\sigma(k,\log n) = \sigma(k-1,0) &\to \sigma(k-2,0) \to \cdots \\
&\to \sigma(1,0) \to \tau(0) \to \tau(N)
\end{aligned}$$

We then derive that $\overline{n}\,\overline{2}\,\overline{2}\cdots\overline{2}\,\Phi$ has type $\tau(0) \to \tau(N)$. This term has a type of rank $k+1$, since $\overline{n}$ takes an argument of rank k, but we can decrease the rank by reducing $\overline{n}\,\overline{2}$ to $\lambda z.\overline{2}^n z$; the latter term takes an argument of rank $k-1$, and is then typable in rank k. Note that this trick of reducing a term M to M', where the rank of the type derivation for M' is less than that for M, is the fundamental technique used in proving upper bound. In this case, it only makes the term size of the iterator grow by a constant factor. □

4.2 The Polymorphic Iteration Lemma Makes Typability Bounds Easy

Now it is fairly straightforward to prove Theorem 4.1. Let δ be the simply-typable term encoding the transition function of Turing Machine $\mathcal{M}$, ID_0 be the simply-typable term encoding of the initial ID, and *accept?* be a simply-typable term extracting a Boolean value from the final ID indicating whether $\mathcal{M}$ accepted its input, as in [HM94]. (Due to space limitations, we do not present the encodings here.) Then using the polymorphic iteration lemma,

$$P_{\mathcal{M},x} \equiv \mathit{accept?}\ (\mathit{polyit}_{n,k}\ \delta\ ID_0)$$

reduces to $\mathit{true} = \lambda x.\lambda y.x : \alpha \to \beta \to \alpha$ if $\mathcal{M}$ accepts x, and reduces to $\mathit{false} = \lambda x.\lambda y.y : \alpha \to \beta \to \beta$ if $\mathcal{M}$ rejects x. By losing an exponential, we can then derive a very easy lower bound for $\mathrm{DTIME}[\mathbf{K}(k-1,n)]$-hardness, by using the term $P_{\mathcal{M},x}$ to choose between a strongly normalizing computation typable in rank $k+1$, and a divergent computation that is not typable in any rank:

$$\hat{C}[P_{\mathcal{M},x}] \equiv (\lambda z.zz)(P_{\mathcal{M},x}\ (\lambda w.w)\ (\lambda w.ww))$$

(An exponential is lost because $\lambda w.ww$ has rank-2 type $\wedge\{a, a \to b\} \to b$, driving up the rank of $P_{\mathcal{M},x}$ by 1.)

A tighter bound comes from the following construction. Let $I = \lambda x.x$, $K = \lambda x.\lambda y.x$, and $\langle x,y\rangle = \lambda w.wxy$. Define $B = \lambda x.\lambda y.Ky\langle y(xIK), yI\rangle$. Notice that $B\,\mathit{true}$ has rank-0 type $\wedge\{\sigma \Rightarrow \sigma\}$ where $\sigma = \wedge\{(a \to a) \to b\}$, and $B\,\mathit{false}$ has rank-2 type $\wedge\{\sigma' \Rightarrow \sigma'\}$, where $\sigma' = \wedge\{(a \to a) \to b, (c \to d \to c) \to e\}$.

Now we can use a gadget that drives the minimal rank of $B\,\mathit{false}$ up to $k+1$ but leaves the minimal rank of $B\,\mathit{true}$ at 0. Define $G_1 = I$ and $G_i = (G_{i-1}I), i > 1$. Suppose the minimal rank typing of a term Q is $\hat{Q}^\tau$, where $\mathsf{rank}(\hat{Q}) = k$ and $\mathsf{rank}(\tau) = j$. Then it turns out that $\mathsf{minrank}(G_nQ)$ is $\max\{k, \mathsf{inc}^n(j)\}$. So $\mathsf{minrank}(G_{k-1}(BP_{\mathcal{M},x}))$ is k if $P_{\mathcal{M},x}$ normalizes to *true*[8] and is $k+1$ if $P_{\mathcal{M},x}$ normalizes to *false*. Thus, the term $G_{k-1}(BP_{\mathcal{M},x})$ is typable in System $\mathcal{I}_k$ iff $P_{\mathcal{M},x}$ normalizes to *true*.

5 Polymorphic Iteration and Expressiveness Theorems

Given two λ-terms of size n that are both typable in System $\mathcal{I}_k$, how hard is it to decide if they have the same normal form? In this paper, we omit many technical details relating to the complexity analysis of this decision problem; see [Sta79, Mai92, AM98]. Instead, we outline the analysis at a high level. The technical details are not difficult, and amount to a fairly mundane form of functional programming.

The above decision problem is a simple form of detecting program equivalence. We can use the polymorphic iteration lemma to get lower bounds on the problem by reworking certain technical machinery in the analysis of a closely related problem, called the *decision problem for type theory*, due to Rick Statman and Albert Meyer [Mey74, Sta79]:

> Let $\mathcal{D}_0 = \{\mathbf{true}, \mathbf{false}\}$, and define $\mathcal{D}_{k+1} = \mathit{powerset}(\mathcal{D}_k)$. Let x^k, y^k, z^k be variables allowed to range over $\mathcal{D}_k$; we define the *prime formulas* as x^0, $\mathbf{true} \in y^1$, $\mathbf{false} \in y^1$, and $x^k \in y^{k+1}$. Now consider a formula Φ built up out of prime formulas, the usual logical connectives $\vee, \wedge, \to, \neg$, and the quantifiers $\forall$ and $\exists$: is Φ true under the usual interpretation?

Define the *length of a formula* to be the number of variables, logical connectives, and quantifier symbols in it; Statman and Meyer proved the following:

Theorem 5.1. *For any Turing Machine M that accepts or rejects its input x of length n in $\mathbf{K}(t,n)$ steps, there is a logspace constructible formula $\Phi_{M,x}$ of length $O(n(t + \log^* n))$, using only variables x^j where $j \le t + c + \log^* n$ for some small integer constant c, such that $\Phi_{M,x}$ is true iff M accepts x.* □

(The Boolean universe $\mathcal{D}_{t+c+\log^* n}$ is just big enough to code such a complex computation.) Statman went on to show that in the simply typed λ-calculus, one could use β-reduction to implement quantifier elimination for this logic, a point clarified in [Mai92], where the quantifier elimination uses a style of primitive recursion that is obvious to any functional programmer. As a consequence, deciding if two simply-typed terms of size n (including type annotations) have the same normal form was not Kalmar-elementary—the decision problem was $\mathrm{DTIME}[\mathbf{K}(t,n)]$-hard for any integer constant $t \ge 0$. The two terms were of type Bool; one term coded a Turing Machine computation via a short formula that said "Does M accept x in $\mathbf{K}(t,n)$ steps?" and the other term coded "true."[9] When we consider typed λ-calculi with more sophisticated type systems, like finite rank intersection types, it becomes possible to implement more powerful quantifier elimination procedures—that is, shorter terms that implement quantifier elimination over bigger Boolean universes. As a warmup, we consider the problem of detecting the equivalence of two Core ML terms, consisting of the simply-typed λ-calculus with a polymorphic `let` construction.

Theorem 5.2. *Given two* core ML *terms, where we augment λ-terms with a polymorphic* `let`*, but do not add fixpoints, deciding if the terms have the same normal form is $\mathrm{DTIME}[\mathbf{K}(2^{n^t}, 2)]$-hard for every integer $t \ge 0$.* □

Proof. (sketch) The main technical problem is to implement quantifier elimination in $\mathcal{D}_j$ by terms of size $O(\log j)$. In the λ-calculus simulation of the logic, sets

[8] This is k rather than 0 because $P_{\mathcal{M},x}$ contributes k to the minimal rank.

[9] This bound can be improved to $\mathrm{DTIME}[\mathbf{K}(\frac{t \log n}{\log\log n}, n)]$-hardness for any fixed integer t. The main technical difficulty is that the universe $\mathcal{D}_j$ is coded by a type having size $O((2j)!)$, which under a logspace transduction must be polynomial in n. The explosion in type size can be handled properly by a transducer if we use sharing (i.e., directed acyclic graphs) to code types.

are represented as lists, abstracting over the constructors ("Church lists"), and functions on sets are implemented via primitive recursion. Here is how to construct large Boolean universes:

$\texttt{let}\ insert = \lambda x.\lambda s.\lambda c.\lambda n.(c\ x\ (s\ c\ n))\ \texttt{in}$
$\texttt{let}\ double = \lambda x.\lambda \ell.\lambda c.\lambda n.\ell\ (\lambda e.c\ (insert\ x\ e))\ (\ell\ c\ n)\ \texttt{in}$
$\texttt{let}\ powerset = \lambda L.L\ double\ (\lambda c.\lambda n.c(\lambda c'.\lambda n'.n')n)\ \texttt{in}$
$\texttt{let}\ p_1 = \lambda y.powerset(powerset\ y)\ \texttt{in}$
$\texttt{let}\ p_2 = \lambda y.p_1(p_1 y)\ \texttt{in}$
$\ldots$
$\texttt{let}\ p_{\log j} = \lambda y.p_{\log j-1}(p_{\log j-1} y)\ \texttt{in}$
$\texttt{let}\ D_j = p_{\log j}\ (\lambda c.\lambda n.c\ true\ (c\ false\ n))\ \texttt{in}\ D_j$

The *double* function takes element x and list of sets ℓ, and constructs the new set $\ell \cup \{e \cup \{x\} \mid e \in \ell\}$; *powerset* iterates this over elements of a set, starting with a set containing the empty set. (Doubling is to exponentiation as *double* is to *powerset*.) Similarly, here is how to implement equality at level j:

$\texttt{let}\ eq_0\ x\ y = or\ (and\ x\ y)\ (and\ (not\ x)\ (not\ y))\ \texttt{in}$
$\texttt{let}\ member\ eq\ x\ \ell = \ell\ (\lambda y.or\ (eq\ x\ y))\ false\ \texttt{in}$
$\texttt{let}\ subset\ mem\ x\ y = x\ (\lambda x'.and\ (mem\ x'\ y))\ true\ \texttt{in}$
$\texttt{let}\ neweq\ subs\ x\ y = and\ (subs\ x\ y)\ (subs\ y\ x)\ \texttt{in}$
$\texttt{let}\ lifteq_0\ eq = neweq\ (subset\ (member\ eq))\ \texttt{in}$
$\texttt{let}\ lifteq_1\ eq = lifteq_0\ (lifteq_0\ eq)\ \texttt{in}$
$\texttt{let}\ lifteq_2\ eq = lifteq_1\ (lifteq_1\ eq)\ \texttt{in}$
$\ldots$
$\texttt{let}\ lifteq_{\log j}\, eq = lifteq_{\log j-1}\ (lifteq_{\log j-1}\ eq)\ \texttt{in}$
$\texttt{let}\ eq_j = lifteq_{\log j}\ eq_0\ \texttt{in}\ eq_j$

In the above construction, eq_0 implements equality over $\mathcal{D}_0 = \{\mathbf{true}, \mathbf{false}\}$. The definition of *member* for $\mathcal{D}_i$ needs as input *eq* for $\mathcal{D}_{i-1}$; similarly, *subset* for $\mathcal{D}_i$ needs *member* for $\mathcal{D}_i$, and *eq* for $\mathcal{D}_i$ needs *subset* for $\mathcal{D}_i$. Then $lifteq_0$ can define *eq* for $\mathcal{D}_i$ given *eq* for $\mathcal{D}_{i-1}$, and by polymorphic composition, $lifteq_j$ can define *eq* for $\mathcal{D}_{i+2^j}$ given *eq* for $\mathcal{D}_i$. □

We can `let`-reduce any core ML term of size n to a simply-typed term of size 2^n. A well-known theorem of Schwichtenberg [Sch81] states that a simply-typed term of size ℓ has a normal form of size at most $\mathbf{K}(\ell, 2)$; we then have a corresponding upper bound for the core ML equivalence problem:

Theorem 5.3. *Given two* core ML *terms of size n, deciding if the terms have the same normal form can be determined in $O(\mathbf{K}(2^n, 2))$ steps.* □

The above lower bound for ML generalizes easily to finiterank intersection type systems:

Theorem 5.4. *Let M be a Turing Machine that accepts or rejects its input x of size n in $\mathbf{K}(\mathbf{K}(k-1, n), 2)$ steps. Then there is a closed, typed term in System $\boldsymbol{\mathcal{I}}_k$ of type* Bool*, constructible in $O(\log n)$ space, that normalizes to $true = \lambda x.\lambda y.x$ when M accepts x, and normalizes to $false = \lambda x.\lambda y.y$ otherwise.* □

Proof. (sketch) The same proof techniques from the ML case carry over, except a different and more powerful kind of polymorphic iteration is used. For example, to construct $\mathcal{D}_j$ where $j = \mathbf{K}(k-1, n)$, we use the λ-term $D_j \equiv polyit_{n,k}\, powerset\ (\lambda c.\lambda n.c\ true\ (c\ false\ n))$; to implement equality for this iterated Boolean universe, we use $eq_j \equiv polyit_{n,k}\, lifteq_0\, eq_0$. □

By standard diagonalization arguments used to prove time-hierarchy theorems, we then have the following expressiveness theorem:

Theorem 5.5. *Let E_1 and E_2 be two terms of size n typable in System $\boldsymbol{\mathcal{I}}_k$. Then deciding if E_1 and E_2 have the same normal form is* DTIME$[\mathbf{K}(\mathbf{K}(k-1, n^t), 2)]$*-hard for every integer $t \geq 0$.* □

In the case of ML, an upper bound on expressiveness came from `let`-reduction, a finite development increasing term size at most exponentially. In System $\boldsymbol{\mathcal{I}}_k$, the analogous operation is given by Simplify(k) from the upper bound construction; this generalization of finite developments increases terms of size n to at most size $\mathbf{K}(k-1, n)$. By removing the λ_{K}-redexes and again applying Schwichtenberg's analysis to the resultant simply-typed term, we can derive a normal form of size at most $\mathbf{K}(\mathbf{K}(k-1, n), 2)$. From this bound on the size of normalized terms, we have:

Theorem 5.6. *Given two terms of size n typable in System $\boldsymbol{\mathcal{I}}_k$, deciding if they have the same normal form is* DTIME$[\mathbf{K}(\mathbf{K}(k-1, n^t), 2)]$*-complete for every integer $t \geq 0$.* □

5.1 Relating the Complexity of Typability and Expressiveness

What is the biggest Church numeral that can be expressed as a typable λ-term of size n? Or put more crudely, what is the largest number that can be specified in n bits? This question is central in understanding bounds on the decidability of program equivalence. In the simply-typed λ-calculus, a term of size n can normalize to the Church numeral for $\mathbf{K}(n, 2)$; in ML, a term of size n can normalize to the Church numeral for $\mathbf{K}(2^n, 2)$; in System $\boldsymbol{\mathcal{I}}_k$, a term of size n can normalize to the Church numeral for $\mathbf{K}(\mathbf{K}(k-1, n), 2)$.

Now consider the relationship between deciding typability, and deciding the equivalence of typed terms. Let $t \geq 2$ be some fixed integer. In the simply-typed λ-calculus, recognizing if a λ-term of size n is typable can be decided in $O(n^t)$ time; deciding equivalence is DTIME$[\mathbf{K}(n^t, 2)]$-hard. Recognizing if a core ML term of size n is typable can be decided in $O(2^{n^t})$ time; deciding equivalence is DTIME$[\mathbf{K}(2^{n^t}, 2)]$-hard. In System $\boldsymbol{\mathcal{I}}_k$, recognizing if a λ-term of size n is typable can be decided in $O(\mathbf{K}(k-1, n^t))$ time; deciding equivalence is DTIME$[\mathbf{K}(\mathbf{K}(k-1, n^t), 2)]$-hard. It is not what happens in the case of System F and its higher-kinded generalization. Typability is undecidable, but nothing is known about the corresponding equivalence problem.

As a consequence of these observations, we make the following conjecture. Let $\mathcal{T}$ be a predicative type system where typability has complexity $t(n)$ and expressiveness has complexity $e(n)$; then $t(n) = \Omega(\log^* e(n))$. This conjecture may well need modification and amendments, and it may even be false. But it is a reasonable first guess as to how one relates typability and expressiveness in complexity-theoretic terms.

If nothing else, the symmetry of these relationships confirm the comments in [Mai90] that "lower bounds on deciding typability for extensions to the [simply] typed λ-calculus can be regarded precisely in terms of this expressive capacity for succinct function composition," and gives evidence in support of a long-held belief that studying the complexity of type inference is important because it is a measure of language expressiveness.

6 Future Work

Despite scary worst-case complexity results, type inference for finite-rank intersection types may be acceptable in practice.We are implementing type inference algorithms for finite-rank intersection types and evaluating their tractability in practice.

The Infer-$\mathcal{I}(k)$ algorithm infers types for a type system with ACI intersection types. In contrast, the finite-rank inference algorithm presented in [KW99] uses non-ACI intersection types, which is not as flexible. However, the system of [KW99] is substitution/unification-based and guarantees principal typings. We are investigating an algorithm that has these advantages for ACI intersection types.

References

[AM98] A. Asperti and H. G. Mairson. Parallel beta reduction is not elementary recursive. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 303–315, San Diego, California, 19–21 Jan. 1998.

[vB93] S. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems.* PhD thesis, Catholic University of Nijmegen, 1993.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[CDCV80] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In J. P. Seldin and J. R. Hindley, eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pp. 535–560. Academic Press, 1980.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematishche Logik und Grundlagen der Mathematik*, 27:45–58, 1981.

[HM94] F. Henglein and H. G. Mairson. The complexity of type inference for higher-order typed lambda calculi. *J. Funct. Prog.*, 4(4):435–478, Oct. 1994.

[Jim96] T. Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Principles of Prog. Languages*, 1996.

[Kam96] F. Kamareddine. A reduction relation for which postponement of k-contractions, conservation and preservation of strong normalisation hold. Technical Report TR-1996-11, Univ. of Glasgow, Glasgow G12 8QQ, Scotland, Mar. 1996.

[KHM94] P. C. Kanellakis, G. G. Hillebrand, and H. G. Mairson. An analysis of the Core-ML language: Expressive power and type reconstruction. In *21st Int'l Colloq. on Automata, Languages, and Programming*, vol. 820 of *LNCS*, pp. 83–106, 1994. Invited paper.

[KRW98] F. Kamareddine, A. Ríos, and J. B. Wells. Calculi of generalised β-reduction and explicit substitutions: The type free and simply typed versions. *J. Functional & Logic Programming*, 1998(5), June 1998.

[KT92] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite-rank fragments of the second-order λ-calculus. *Inf. & Comput.*, 98(2):228–257, June 1992.

[KW94] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, 1994.

[KW95] A. J. Kfoury and J. B. Wells. Addendum to "New notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi". Tech. Rep. 95-007, Comp. Sci. Dept., Boston Univ., 1995.

[KW99] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Principles of Prog. Languages*, 1999.

[Lei83] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Principles of Programming Languages*, pp. 88–98, 1983.

[Mai90] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conf. Rec. 17th Ann. ACM Symp. Principles of Programming Languages*, pp. 382–401, 1990.

[Mai92] H. G. Mairson. A simple proof of a theorem of Statman. *Theoretical Computer Science*, 103(2):387–394, Sept. 1992.

[Mey74] A. R. Meyer. The inherent computational complexity of theories of ordered sets. In *Proceedings of the International Congress of Mathematicians.*, pp. 477–482, 1974.

[Mit96] J. C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[RDRV84] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theor. Comp. Sc.*, 28:151–169, 1984.

[Sch81] H. Schwichtenberg. Complexity of normalization in the pure typed lambda-calculus. In A. S. Troelstra and D. van Dalen, eds., *Proceedings L. E. J. Brouwer Centenary Symp.*, vol. 110 of *Studies in Logic and the Foundations of Mathematics*, pp. 453–457, Noordwijkerhout, The Netherlands, June8–13, 1981. North-Holland, Amsterdam. Published in 1982.

[Sta79] R. Statman. The typed lambda -calculus is not elementary recursive. *Theoretical Computer Science*, 9(1):73–81, July 1979.

[WDMT97] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A typed intermediate language for flow-directed compilation. In *Proc. 7th Int'l Joint Conf. Theory & Practice of Software Development*, 1997. Superseded by [WDMT9X].

[WDMT9X] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Prog.*, 199X. To appear after revisions. Supersedes [WDMT97].