Types, Potency, and Idempotency: Why Nonlinearity and Amnesia Make a Type System Work

Peter Møller Neergaard*
Computer Science Department
Brandeis University
Waltham, MA 02454, USA
turtle@achilles.linearity.org

Harry G. Mairson†
Computer Science Department
Brandeis University
Waltham, MA 02454, USA
mairson@cs.brandeis.edu

Abstract

Useful type inference must be faster than normalization. Otherwise, you could check safety conditions by running the program. We analyze the relationship between bounds on normalization and type inference. We show how the success of type inference is fundamentally related to the *amnesia* of the type system: the *nonlinearity* by which all instances of a variable are constrained to have the same type.

Recent work on *intersection types* has advocated their usefulness for static analysis and modular compilation. We analyze System-$\mathbb{I}$ (and some instances of its descendant, System E), an intersection type system with a type inference algorithm. Because System-$\mathbb{I}$ lacks *idempotency*, each occurrence of a variable requires a distinct type. Consequently, type inference is equivalent to normalization in *every single case*, and time bounds on type inference and normalization are identical. Similar relationships hold for other intersection type systems without idempotency.

The analysis is founded on an investigation of the relationship between *linear logic* and intersection types. We show a lockstep correspondence between normalization and type inference. The latter shows the promise of intersection types to facilitate static analyses of varied granularity, but also belies an immense challenge: to add amnesia to such analysis without losing all of its benefits.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Program and recursion schemes*; F.4.1 [**Computation by Abstract Devices**]: Mathematical Logic—*Computability theory, Computational logic, Lambda calculus and related systems*

General Terms

Languages, theory, performance

Keywords

Intersection types, type inference, proofnet, normalization, complexity, idempotence

1 Introduction

Those who forget the past are condemned to repeat it.
–George Santayana

Type inference is a true success story of programming language theory and pragmatics: added to languages like ML, Haskell, OCaml, it works without the programmer worrying (too much) how. Type inference works so well that it is easy to forget the ingenuity which lead to its success.

Like other forms of static safety analysis, type inference predicts at *compile-time* the behavior of the program at *run-time*—recall Milner's "well-typed programs do not go wrong" [35]. A crucial requirement for type inference is that it is substantially faster than running the program—otherwise, you could get the same result as follows: run the program, and watch! This efficiency has been achieved for simple types [43], the Hindley/Milner ML type system [18, 35, 32, 25], and some intersection type systems [24]. In these systems, there is an exponential or super-exponential leap from the worst-case bound on type inference to the worst-case bound on normalization (running time). As a nonobvious consequence, even programs with small types can run long enough to do something interesting; this explains why ML works so well.

We explain the super-exponential gap via an investigation of *intersection types*, where polymorphism is itemized explicitly using a type constructor $\wedge$: a term of type $\tau \wedge \sigma$ can have both type τ and type σ. Intersection types have gained recent interest for such varied purposes as flow analysis [4], strictness analysis [20], dead-code elimination [10, 13], and totality analysis [8]. Moreover, intersection types have been advocated for modular compilation [6, 26, 12, 4] as they (usually) have principal typings [21]—that is, there exists a single typing from which all other typings of a

*Supported by the Danish Research Agency grants 1999-114-0027 and 642-00-0062 and the NSF grant CCR-9806718.

†Supported by NSF Grants CCR-9619638, CDA-9806718, and the Tyson Foundation.

ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.

term can be derived.[1] A significant, interesting, and representative example of an intersection type system is the prototype system System-$\mathbb{I}$ [28, 26] (and its descendant, System E [6]), which has both principal typings, and automatic type inference [26, 7]. These type systems serve as a foundation of the Church compiler project [48].

We analyze here the relationship in System-$\mathbb{I}$ between the cost of type inference, and the cost of normalization. We focus on the crucial fact that System-$\mathbb{I}$ *lacks type idempotency:* $a \wedge a \neq a$. The analysis establishes the following:

A formal connection between intersection types and *linear logic* originally suggested by Regnier [39]. Most notably, the *expansion variables* [28, 6] of System-$\mathbb{I}$ are shown to be a syntactic rendition of the *boxes* from proofnets [29]. Further, the $\wedge$ in types are sharing nodes, and normalization bounds mimic normalization techniques from *elementary linear logic*.

An exact complexity analysis of the type inference required for System-$\mathbb{I}$. The analysis formalizes the relationship between type inference and normalization, since the inference rules correspond to the reduction rules for proofnets.

A corollary of this analysis is that normalization and type inference are identical in *every single case*. This is evidenced by the Church Project's experimentation tool [46]: inference for a seemingly innocent ML-program like `(fn s=>fn z=> s(s(s z)))(fn s=>fn z=> s(s z))(fn s=>fn z=>s(s z))` times out. The I in System-$\mathbb{I}$ therefore reminds us of its computational *impotence*. Without idempotency, you need a much bigger type to get a much longer runtime—in fact, a type potentially as big as the runtime. This holds for other intersection type systems without idempotency.

Compelling evidence that *amnesia* is crucial for type systems. Contrasting the expressiveness results for System-$\mathbb{I}$ with the results mentioned above for simply-typed λ-calculus and ML, the fundamental difference is in the *simply-typable* terms of the systems. Simply-typed λ-calculus has amnesia (idempotence) and normalization is not bound by any elementary function [43], i.e., the simply-typed λ-calculus has nonelementary power.[2] In contrast, simple ($\wedge$-free) types in System-$\mathbb{I}$ have only *linear* power as normalization can be done in linear time in the size of the term. More complex types—for example, ML polymorphism versus the introduction of $\wedge$-types—amplify this distinction.

In the rest of this section, we introduce the concepts mentioned above, yet still using a fairly broad, informal brush. In Sec. 2 we give some required technical preliminaries. In Sec. 3 and 4 we prove that without amnesia (idempotency), intersection type systems have identical bounds on normalization and type inference. In Sec. 5 we go one step further and show that for System-$\mathbb{I}$ type inference *is* normalization—a result that should hold for other intersection type systems without idempotency.

1.1 Intersection types

Intersection types provide an alternative to the ML/System F paradigm of *parametric* polymorphism. They implement a *finite* polymorphism which is captured by the following typing rules:

$$\frac{}{x : a_1 \wedge a_2 \vdash x : a_i} \qquad \frac{\Gamma_1 \vdash M : a_1 \quad \Gamma_2 \vdash M : a_2}{\Gamma_1 \wedge \Gamma_2 \vdash M : a_1 \wedge a_2} \ .$$

The first rule states that if x has type $a_1 \wedge a_2$, any *occurrence* of x may have either type a_1 or type a_2. The second rule states that if term M can have both type a_1 and type a_2 in environments Γ_1 and Γ_2, then M also has intersection type $a_1 \wedge a_2$ in the combined environment $\Gamma_1 \wedge \Gamma_2$. When combining types and environments a seemingly technical question is whether $\wedge$ is *associative, commutative,* or *idempotent* (ACI). In other words, which of the following equations hold?

$$a \wedge (b \wedge c) = (a \wedge b) \wedge c \qquad a \wedge b = b \wedge a \qquad a \wedge a = a$$

Even though leaving out ACI simplifies some matters (such as type inference), the choice, as we shall see, has substantial consequences for normalization bounds.

The recent interest in intersection types is motivated by 3 concerns:

Typing more terms: In full generality, type inference with both universal types and intersection types is undecidable—Wells proves this for universal types [47]; for intersection types, observe that such types characterize exactly the strongly normalizing terms [2][3]; see also Cor. 5.9. Practical implementations therefore employ a restriction. For example, ML allows only outermost quantification and therefore rejects `fn f => (f 3, f true)` since `f` must have monomorphic type, but type `int` does not unify with type `bool`. For intersection types, *rank-bounds*, which bound the depth of a $\wedge$-type in an $\rightarrow$-type by a constant (the *rank)*, are used. Already with rank 2—which, by an oddity in the way of counting, is the lowest non-trivial rank—all ML programs and more can be typed, e.g., the program above has intersection type $((\texttt{int} \rightarrow a) \wedge (\texttt{bool} \rightarrow b)) \rightarrow a \times b$. This is well-studied by Damiani [11].

Modular compilation, where the different parts of a software system are developed and compiled independently, becomes increasingly important as program size increases. Intersection types are a strong candidate for supporting type inference under modular compilation. To avoid reanalyzing the code, we make use of the principal typing. Intersection type systems are advantageous because they—in most cases—have a principal typing for all programs.

Static analysis: Flow analysis is a typical kind of static analysis—does some fixed call site in a program ever call some other procedure? Since each variable occurrence has a unique non-ACI intersection type, typed-based flow analysis between call sites and functions can optimize specific procedure applications. This has been exploited for instance by Mossin [37].

1.2 Comparing expressiveness, type inference

We observe that without idempotency, *linearity* is enforced in the types: if a variable occurs twice, it has type $a \wedge b$, even if $a = b$. Consequently, in a term with a *simple* type (i.e., without $\wedge$), any variable (free or bound) occurs at most once. Such terms normalize in *linear time*—vastly less powerful than the nonelementary bounds found in the simply-typed λ-calculus. We use this observation as the base case of a variant of Statman's theorem [43]. The linearity

[1] This should not be confused with the weaker notion of principal types which is present in the Hindley/Milner type system [35]. With principal types the type environment is fixed.

[2] Recall that an *elementary function* is any function bounded by a stack of exponentials, i.e., $2^{2^{\cdots^{2^x}}}$.

[3] The statement has been presented many places in the literature, but with incorrect proofs. B. Venneri in unpublished work made the first correct proof. The reference here appears to be the first correct proof in the published literature [23, Footnote 8].

spills over when we introduce $\wedge$, and we prove that normalization of *rank-bounded* fragments of intersection type systems without idempotency take at most elementary time.

In lower bounds on type inference, the fundamental technical question is how to iterate a linear function with different domain and range types. As shown by Henglein and Mairson [17], any Turing machine transition function can be encoded as a linear λ-term; we can therefore represent the transition function by a simple System-$\mathbb{I}$ term. Since the Turing machine state is implicitly coded in the type, the domain and range types must be different. By *polymorphically* iterating this function, we can prove that for every *rank* we add to the type system, expressiveness is increased by an exponential, as is the cost of type inference.

1.3 Relating type inference, expressiveness in every case

The shared upper bound on type inference and normalization is however only the start: unlike most languages you can think of, in System-$\mathbb{I}$ type inference is *equal* to normalization for every single term.

To relate type inference and normalization, we recast λ-terms as *sharing graphs* for λ-calculus [15]. We refer to the graphs as *interaction nets* (or just *nets*) when untyped and *proofnets* when containing typing information. The *sharing nodes* of nets (indicating *contraction* in linear logic) represent intersections.

System-$\mathbb{I}$ uses the technology of *expansion variables* to facilitate *compositionality*, by delaying the application of a typing rule. Expansion variables are the System-$\mathbb{I}$ syntax for *boxes* from interaction nets (though they are not identical with the *exponentials* of linear logic). The boxes delimite a piece of code where we later may apply an intersection typing rule: if a box has typing $\langle \Gamma; a \rangle$, we might decide later that it has typing $\langle \Gamma_1 \wedge \cdots \wedge \Gamma_n; a_1 \wedge \cdots \wedge a_n \rangle$ where Γ_i (a_i) are derived from Γ (a) by picking fresh names for the type variables. Furthermore, by simply ignoring the expansion variables, we have a standard ACI-less intersection type systems. Using standard interaction net machinery, associativity and commutativity can be added to the system. The results therefore speak not only about System-$\mathbb{I}$, but in general about intersection type systems without idempotency.

The delaying of typing rules seems a promising way to avoid analyzing a term more than once. Nonetheless, the lack of idempotency forces the exact number of copies needed to be made explicit when the box is passed in as an argument. The end result is that the set of typings is invariant under reduction. Principal typings and normal forms are consequently *isomorphic*, following the intuitive understanding all functional programmers have of reading a function from its type [45, 33] and *vice versa*. Combining these insights, we conclude that type inference *is* normalization: we can either normalize a term (which terminates due to the normalization bounds) and read back the principal typing, or find the principal typing and return the normal form. On the operational level, this correspondence is realized by observing that the unification rules of the original presentation of System-$\mathbb{I}$ [27] are exactly the rules for global reduction of interaction nets.

How may we interpret these technical results to yield practical insights? Worst-case lower bounds (in ML, for example) have traditionally been wished away since type inference works in practice: in other words, programs with small types are good enough. Why? Programmers can usually keep the type in their heads, goes one saying—because the bounds on normalization show that there is a tremendous expressive power, even with those small types. But you cannot keep a System-$\mathbb{I}$ type in your head, because you would then know—in advance—exactly what your program is computing.

The result is however not all negative: It stresses that intersection types facilitate an exact program analysis—you do not get more precise than running the program. If combined with imprecise analysis—as suggested by the non-linear ! present in System E—it provides a powerful tool that allows the programmer to get the precision that she needs in the analysis of her program.

1.4 Relating flow analysis and the geometry of interaction

We observe that in the interpretation of intersection-typed programs as nets, flow analysis for programs becomes essentially a reformulation of Girard's *geometry of interaction* (GoI), made mundane for computer scientists in the guise of *context semantics* [16, 15, 34]. From the context semantics of a term, we may recover intersection typing information. The method is an elaboration of a *readback algorithm* we have used to prove the correctness of optimal reduction [30, 15, 3, 34], where we not only want to read back a λ-term, but also the location of linear logic *boxes* (representing expansion variables), and the existence, commutativity, and associativity of sharing nodes. For space reasons we cannot provide further details, but hope to in a longer version of this paper.

1.5 Related work

Coppo, Dezani, and Venneri [9], Ronchi della Rocca and Venneri [41], van Bakel [44], and Sayag and Mauny [42] have all established a connection between inferring the principal intersection typing and normalization as they (essentially) infer the principal typing through normalization. In fact, we share the overall idea with Sayag and Mauny: establish a connection on normal forms and show subject reduction and expansion. However, as we—through the connection to linear logic—also account for the expansion variables introduced by System-$\mathbb{I}$, our result is significantly stronger than the previous results.

The relationship between intersection types and linear logic was first observed by Regnier [39]. The present work formalizes his observations and extends his contribution, by relating normalization and type inference from complexity-theoretic as well as observational perspectives.

Kfoury et al. [24] analyze the computational difficulty of type inference for intersection types with ACI, comparing it to the expressive power of the language. They develop a naïve type inference algorithm based on `let`-expansion [36]. Let the *Kalmar elementary functions* be $\mathbf{K}(0,n) = n$ and $\mathbf{K}(t+1,n) = 2^{\mathbf{K}(t,n)}$; they prove that type-inference for a term M of rank r is in $\textsc{dtime}[\mathbf{K}(r,|M|)]$, while the expressiveness is in $\textsc{dtime}[\mathbf{K}(\mathbf{K}(r,|M|),1)]$. In this paper, we show that when idempotency is removed from the type system, the expressiveness bound reverts to the bound for type inference.

Carlier et al. have recently supplanted System-$\mathbb{I}$ by System E [6], which overcomes a number of technical deficiencies of System-$\mathbb{I}$. It also introduces a !-modality to allow inexact analysis. The current type inference for System E [7, 6], however, uses only the linear fragment of System E. It therefore corresponds to the type inference in System-$\mathbb{I}$. Since the present work was first presented at the

working meeting on intersection types in the Summer 2003, Carlier and Wells have subsequently proved the step-wise correspondence between type inference and normalization in the setting of System E [7]. Carlier et al. also show how System E can do the analysis corresponding to Lévy labels [31].

1.6 Acknowledgements

We thank Joe Wells for explaining the intricate details of System-$\mathbb{I}$, and for his invitation to visit Heriot-Watt University during summer 2003, as well as Sébastien Carlier for a very inspiring seminar on β-unification and proofnets—both have done a tremendous effort to answer our questions. For their technical comments and criticisms, we thank Alan Bawden, Vincent Danos, Jakob Grue Simonsen, Joe Hallett, Assaf Kfoury, Laurent Regnier, Simona Ronchi Della Rocca, Sebastian Skalberg, Franklyn Turbak, Paweł Urzyczyn, and Steffen van Bakel. We thank the anonymous referees, in particular #4, whose detailed astute technical comments caught several howlers and vastly improved the technical presentation.

2 Preliminaries

Before starting the real party proving the relationship between type inference for intersection types and normalization, let us do a warm-up waltz recalling the central concepts. This is not self-contained, but aims at getting the reader back in rhythm with the main concepts of this paper.

We use $\mathbb{N}$ to denote the positive integers and $\mathbb{N}_0$ for the non-negative integers. We use $f : A \to B$ to denote a complete function f from A to B and $f : A \hookrightarrow B$ to denote a partial function. Disjoint union is described with $\uplus$. We use log for logarithm base 2.

We then in turn introduce the two major systems we will use in this paper—System-$\mathbb{I}$, a rigid (non-ACI) intersection type system for the λ-calculus, and *interaction nets*, an alternative, more precise syntax for the λ-calculus. We consider the typing problem of the basic λ-calculus. The set of λ-terms Λ is given by following grammar:

$$\Lambda \ni M,N,P,Q ::= x \mid \lambda x.M \mid MN \tag{1}$$

where the variables x are drawn from the countable set V_Λ. We adopt Barendregt's variable convention [5]. We use $\#_x(M)$ for the number of free occurrences of the variable x in M. We use the standard size measure $|\cdot|$ on terms. We use $x^n y$ to denote n applications of x to y; e.g., $s^3 z = s(s(sz))$; the Church numeral $\overline{n}$ of n is $\lambda s.\lambda z.s^n z$. The capture-free substitution of Q for x in P is written $P[Q/x]$. We define the usual notion of β-reduction: $(\lambda x.P)\,Q \;\beta\; P[Q/x]$. When $\mathbf{R}$ is a notion of reduction (e.g., β) we use $\to_{\mathbf{R}}$ for the compatible closure and $\twoheadrightarrow_{\mathbf{R}}$ for the reflexive transitive closure of $\to_{\mathbf{R}}$.

We define subterm indirectly to have a unique path identifying the subterm: $(M,\varepsilon) \Subset M$, $(P,\mathrm{L}p) \Subset \lambda x.M$ when $(P,p) \Subset M$, $(P,\mathrm{L}p) \Subset MN$ when $(P,p) \Subset M$, and $(P,\mathrm{R}p) \Subset MN$ when $(P,p) \Subset N$. We call P a subterm of M, written $P \subseteq M$, if, and only if, $(P,p) \Subset M$ for some p. The subterm of path p is $\mathsf{sub}(M,p) = P$ when $(P,p) \Subset M$ for some P and undefined otherwise.

2.1 System-$\mathbb{I}$: types, expansions, rank

We motivate System-$\mathbb{I}$ by first considering an intersection type system in the style of van Bakel [44]. Types $\tau,\sigma,\rho \in T$ and simple types $\overline{\tau} \in \overline{T}$ are given by the grammar

$$\overline{\tau} ::= a \mid \tau \to \overline{\tau} \qquad \tau ::= \overline{\tau} \mid \tau \wedge \tau$$

where $a,b,c,\ldots$ are type variables. Idempotency, associativity, and commutativity are the equivalences $\tau \wedge \tau = \tau$, $\tau \wedge (\tau' \wedge \tau'') = (\tau \wedge \tau') \wedge \tau''$, $\tau \wedge \tau' = \tau' \wedge \tau$; when these do not hold, we call the $\wedge$ operator *rigid*. The typing rules are

$$\frac{}{x : \overline{\tau} \vdash_{\mathbb{I}} x : \overline{\tau}}\,\mathrm{Var}$$

$$\frac{\Gamma, x : \tau \vdash_{\mathbb{I}} P : \overline{\tau}}{\Gamma \vdash_{\mathbb{I}} \lambda^I x.P : \tau \to \overline{\tau}}\,\lambda\mathrm{I} \qquad \frac{\Gamma \vdash_{\mathbb{I}} P : \overline{\tau}'}{\Gamma \vdash_{\mathbb{I}} \lambda^K x.P : \tau \to \overline{\tau}'}\,\lambda\mathrm{K}$$

$$\frac{\Gamma \vdash_{\mathbb{I}} P : \tau \quad \Gamma' \vdash_{\mathbb{I}} P : \tau'}{\Gamma \wedge \Gamma' \vdash_{\mathbb{I}} P : \tau \wedge \tau'}\,\wedge \qquad \frac{\Gamma \vdash_{\mathbb{I}} P : \tau \to \overline{\tau} \quad \Gamma' \vdash_{\mathbb{I}} Q : \tau}{\Gamma \wedge \Gamma' \vdash_{\mathbb{I}} PQ : \overline{\tau}}\,@$$

Rule λ^I (λ^K) types abstraction $\lambda x.P$ with $x \in \mathsf{fv}(P)$ ($x \notin \mathsf{fv}(P)$), and

$$\Gamma_0 \wedge \Gamma_1 = \{x : \tau \mid x : \tau \in \Gamma_i, x \notin \mathrm{dom}\,\Gamma_{1-i}\} \cup \{x : \tau_0 \wedge \tau_1 \mid x : \tau_i \in \Gamma_i\}\ .$$

Without the $\wedge$-rule, the typing rules would be syntax-directed. Note that the rigidity of $\wedge$ and the formulation of rule @ implies that a variable x with n occurrences in a term has at least $n-1$ top-level $\wedge$. The duplication of the term P in the premises of $\wedge$ highlights a verbose feature of the type system. We call $\langle \Gamma;\tau\rangle \in (V_\Lambda \hookrightarrow T) \times T$ a *typing* of a term M if, and only if, $\Gamma \vdash M : \tau$. Due to the rule $\wedge$ there might be several derivations of a type for a subterm in a type derivation. We use $\mathsf{typ}(\Delta,p)$ for all the types derived for the subterm identified by the subterm path p in the type derivation Δ; when the subterm N is clear from the context we write $\mathsf{typ}(\Delta,N)$.

System-$\mathbb{I}$ attempts to restrain the above verbosity, while retaining principal typings [28]. It avoids the redundancy of rule $\wedge$ by inferring a *principal typing* once, and deferring the choice of how many instances until a reasonable choice can be made. This is accomplished using expansion variables.

Expansion variables are motivated by the fact that for intersection types, substitution is often not enough to obtain an instance of a principal typing. For instance, in the term $M \equiv (\lambda x.xx)(\lambda y.y)$ the subterm $N \equiv \lambda y.y$ has principal type $\tau_N \equiv a \to a$. In a principal typing for M, the subterm N has type $(\tau_N \to \tau_N) \wedge \tau_N$ which cannot be obtained by substitution into τ_N as the top-level type constructors are different. Instead, we first *expand* by duplicating (parts of) the principal type with fresh names to obtain $(a^\circ \to a^\circ) \wedge (a^\bullet \to a^\bullet)$. We then apply the simultaneous substitution $\{a^\circ = \tau_N, a^\bullet = a\}$. This is formalized in numerous ways [41, 40, 44, 42], but unfortunately with a very heavy notation. Kfoury and Wells made a significant simplification by extending the types with *expansion variables* which marks where the expansions should take place.

Concretely, System-$\mathbb{I}$ introduces *expansion variables* F,G ranging over *expansions* $e \in E$, and extends the definition of types to include expansion variables:

$$e \in E ::= \Box \mid e \wedge e \mid \mathrm{F}\,e \qquad\qquad \sigma,\tau \in T ::= \overline{\tau} \mid \tau \wedge \tau \mid \mathrm{F}\,\tau\ .$$

and extend the typing rules with

$$\frac{\Gamma \vdash_{\mathbb{I}} P : \tau}{\mathrm{F}\,\Gamma \vdash_{\mathbb{I}} P : \mathrm{F}\,\tau}\,\mathrm{E}$$

where $\mathrm{F}\,\Gamma = \{x : \mathrm{F}\alpha \mid x : \alpha \in \Gamma\}$.

The expansion variables come to use when doing type inference. The type inference first derives a typing $\langle \Gamma; \overline{\tau} \rangle$ and a set of constraints C on the form $\tau = \tau'$. It then finds a substitution S that solves the constraints C. The inferred type is obtained by applying S to $\langle \Gamma; \overline{\tau} \rangle$. The inference is syntax-directed and straightforward except for the application where we use[4]

$$\frac{\Gamma \vdash_{\mathbb{I}} P : \overline{\tau} \quad \Gamma' \vdash_{\mathbb{I}} Q : \tau}{\Gamma \wedge (\mathrm{F}\,\Gamma') \vdash_{\mathbb{I}} P\,Q : \overline{\tau}'} \text{ AppInfer}$$

where F and $\overline{\tau}'$ are fresh. We add the constraint that $\overline{\tau} = \mathrm{F}\,\tau \to \overline{\tau}'$ and modify each constraint $\tau = \tau'$ found for Q to be $\mathrm{F}\,\tau = \mathrm{F}\,\tau'$.

To meaningfully talk about solutions, we need to specify what it means to substitute an expansion for an expansion variables. The idea is that when e is substituted for F, applications $\mathrm{F}\,\tau$ are replaced by e with each hole filled with a fresh instance of τ. The fresh instances are chosen such that if the same expansion is applied to different instances of a type variable, it gets the same fresh name. A formal definition and some further constraints necessary for technical reasons are in the formal definition of System-$\mathbb{I}$ [28, 26].

Intersection type systems (and System-$\mathbb{I}$) type exactly the strongly normalizing λ-terms, so type inference is undecidable. We obtain a tractable fragment by limiting the depth of $\wedge$ under $\to$. Usually [24] rank is defined as the maximal depth of $\wedge$ in the left child of a $\to$. We use a tighter definition where we only count the *alternation* of $\wedge$ on the "domain" (i.e., argument) side of a $\to$ in types. Formally, we use the functions $\mathsf{rank} : (\overline{T} \cup T) \to \mathbb{N}_0$ and $\mathsf{rank'} : T \to \mathbb{N}_0$:

$$\begin{aligned} \mathsf{rank}(a) &= 0 & \mathsf{rank}(\tau \wedge \tau') &= \max\{\mathsf{rank}\,\tau, \mathsf{rank}\,\tau'\} \\ \mathsf{rank}(\mathrm{F}\,\tau) &= \mathsf{rank}\,\tau & \mathsf{rank}(\tau \to \overline{\tau}) &= \max\{\mathsf{rank'}\,\tau, \mathsf{rank}\,\overline{\tau}\} \\ \mathsf{rank'}(\overline{\tau}) &= \mathsf{rank}\,\overline{\tau} & \mathsf{rank'}(\mathrm{F}\,\tau) &= \mathsf{rank'}\,\tau \\ \mathsf{rank'}(\tau \wedge \tau') &= 1 + \max\{\mathsf{rank}\,\tau, \mathsf{rank}\,\tau'\} \end{aligned} \tag{2}$$

A term M with typing $\langle \Gamma; \tau \rangle$ derived by Δ has as *rank* the maximal rank of an abstracted variable (either explicitly or in the environment): $\mathsf{rank}(\Delta) = \max(\{\mathsf{rank'}\,\sigma \mid z : \sigma \in \Gamma\} \cup \{\mathsf{rank}\,\sigma \mid \exists p.(\sigma \in \mathsf{typ}(\Delta, p)\ \&\ \mathsf{sub}(M, p) = \lambda z.Q)\})$.

2.2 Interaction nets and proofnets

Interaction nets provide a graphical representation of λ-calculus terms, while avoiding problems of variable capture, preserving subject reduction, and providing a link to concepts in linear logic. The λ-calculus can be represented using nets (here, we have used the *call-by-name* encoding $\lceil \cdot \rceil$, presented inductively in Fig. 1); to represent System-$\mathbb{I}$ typings we add typing information to the nets and obtain *proofnets*.

The external vertices of a net are either *free ports* (◯) representing the free variables, or the distinguished *root port* for the whole term. *Weakening nodes* (⊗) mark unused function arguments; *application (@) and function nodes (λ)* mark the definition and use of procedures; *sharing nodes* (▽) code the multiplicity of variable occurrences; and *croissant nodes* (⌢) mark variable occurrences. A *global* construction, the *box*, delineates the (sharable) arguments of an application. Edges are *wires*; their endpoints are attached to

[4] Strictly speaking, the System-$\mathbb{I}$ algorithm has an extra step where the term is decorated with expansion variables to form a so-called *skeleton*. In principle this can be done in many ways, but the algorithm decorates the arguments to applications as shown here.

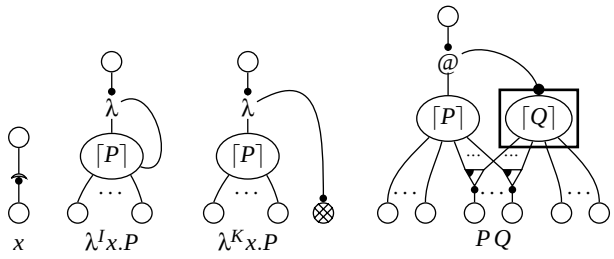

Figure 1. The inductive encoding $\lceil \cdot \rceil$ of a λ-term as a net. The port (◯) on top is the root, representing the whole term, while the ports in the bottom correspond to the free variables. In the application case the left group of wires is the variables solely in P, the middle group the variables occurring in both P and Q, and the right group is the variables solely in Q.

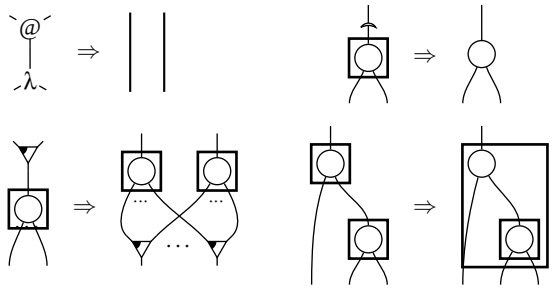

Figure 2. Reduction rules for interaction nets.

ports of either a node, or the entire graph. Each node has a principal port (marked with a black dot), and possibly other *auxiliary ports*.

Equivalent to doing β-reduction, a net can be reduced using the rules presented in Fig. 2. An interaction takes place between an @- and λ-node connected on their principal ports, between a sharing node and box connected on their principal ports, or between two boxes, connected from a principal to an auxiliary port. These rules are the graphical equivalent of β-unification, the constraint solving rules used in System-$\mathbb{I}$ [27].

We omit discussion of necessary *weakening* rules, since our primary concern is type inference where erasing an argument can change typability. The system is Church-Rosser, and we write $\mathrm{NF}_{\Rightarrow}$ for its normal forms; it simulates β-reduction up to trivial permutations on the sharing nodes, as in the following proposition.

Proposition 2.1. *Let M and N be λ-terms such that $M \to_\beta N$. There is a net I such that $\lceil M \rceil \Rightarrow I$, and I is equivalent to the nodes of $\lceil N \rceil$ reachable from the root up to left/right orientation of auxiliary ports on sharing nodes, and their location inside or outside boxes.* □

We restrict our consideration to the nets that arise from encoding and reducing λ-terms.

At this point, we have a language corresponding to the untyped λ-calculus. We recover the equivalent of a type system by annotating the wires of the net with types, oriented in a fixed direction along the wire, where the type system enforces constraints around the ports of nodes and boxes. We call an annotated net a *proofnet*. For example, with simple types, the wire on the principal port of an @-node has an incoming type $\alpha \to \beta$, and the right (left) auxiliary port has an

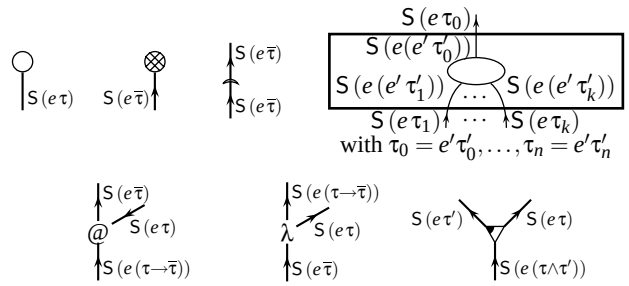

Figure 3. The constraints on proofnet nodes in order to be well-typed. Note that $\mathsf{S}(e(e'\tau_i'))$ is the same as $\mathsf{S}(\underline{e}\tau_i')$ where $\underline{e}$ is e with e' (with renamed expansion variables) substituted for each $\square$. The annotations $\mathsf{S}(e(e'\tau_i'))$ thus consist of the substitution S, the expansion $\underline{e}$ and the type τ_i' so the case for box is well-defined.

incoming (outgoing) type α (β). Observe that each of the typing rules Var, λI, λK, and AppInfer correspond naïvely and directly to our inductive cases in the encoding. However, the $\wedge$-rule is a little problematic; we have two options—either duplicate the structure of the box contents, or figure out a way to share them.

The first option amounts to typing a boxed net (with output of type $\alpha \wedge \beta$) by explicitly representing the two subderivations as distinct proofnets, and pairing their input to the application.[5] But then the sharing nodes are *unpairing* nodes (with the linear logic equivalent of $\parr$), and the multiple proofnets are paired together with $\otimes$. However, the resulting proofnet would be the same size as that of the normal form, and all reductions would be linear. We therefore take the alternative option, where we do not duplicate the term. Instead, we use *expansions* in the spirit of System-$\mathbb{I}$ to represent the different typings. With these considerations in mind, we define the intersection typing of a proofnet *à la* System-$\mathbb{I}$:

Definition 2.2. A *proofnet* is an interaction net where

(1). Each wire is oriented, and annotated with a triple consisting of a type τ, an expansion e, and a variable substitution S. The expansion of the type followed by the substitution, $\mathsf{S}(e\tau)$, gives the different types of the wire; we therefore write the annotations as $(\mathsf{S}(e\tau))$.

(2). Exactly one free port has its wire oriented toward the port, denoting the root of the λ-term of interest. This port is the *conclusion* of the proof Π (holding the type of the λ-term); the remaining ports are the *assumptions* (corresponding to the types of the free variables).

(3). The triples on the wires incident with each node and box satisfies the constraints in Fig. 3.

(4). The conclusion and the assumptions have the same expansion e.

(5). A *switching* is the graph derived by replacing each λ-node with a wire from the principal port to one of the auxiliary ports. Every switching results in a forest of connected, acyclic graphs containing either the root port, or the port marking a weakening from a K-abstraction. (This is in essence the correctness criterion of Danos and Regnier [14].) □

If Π without annotations and orientations of the wires is the net I, we call Π a *typing* of I. We say that I has typing $\langle x_1 : \tau_1, ..., x_n : \tau_n; \tau\rangle$ where τ is the type of the conclusion of Π and τ_i is the type on the free port corresponding to the variable x_i.

Theorem 2.3. *Let M be a λ-term. Then M has System-$\mathbb{I}$-typing $\langle\Gamma;\tau\rangle$ if, and only if, $\lceil M\rceil$ has typing $\langle\Gamma;\tau\rangle$.* □

Every System-$\mathbb{I}$ typable term has a *principal typing* [28]:

Lemma 2.4 (implicit in [28]). *In a principal System-$\mathbb{I}$ typing, every type variable occurs at most twice; if it occurs twice it occurs once positively and once negatively.*[6] *In the syntax tree of any type, each occurrence of a type variable has the same expansion variables along any path from the root of the syntax tree to the occurrence. Every expansion variable occurs at most once positively.* □

The lemma facilitates a translation to the net formulation:

Definition 2.5. Let Π be a proofnet typing a net I; Π is *principal* if for any other proofnet Π' typing I there exists a substitution S' and an expansion substitution E' such Π' can be obtained from Π by replacing each of the annotations $\mathsf{S}(e\tau)$ with $(\mathsf{S}'\circ\mathsf{S})((E'e)\tau)$. □

3 Expressiveness bounds

We can now engage in our main motif of understanding what makes type inference worthwhile for intersection type systems. In this section we derive bounds on the expressiveness of rigid intersection types. We consider this decision problem: how hard is it to tell whether two typed programs are equivalent? (A decision problem avoids any question of irrelevant costs due to the choice of output format). We prove that given two λ-terms M and N typable with rank at most r it is $\text{DTIME}[(\mathbf{K}(r,|M|))]$-hard to decide whether the two terms are equal. This should be compared with the time $\Omega(\mathbf{K}(r,|M|))$ lower bound on deciding whether term M is typable in rank r which we derive in the following section. We shall see that the function $\mathbf{K}(r,n)$ appears in both bounds because of the lack of idempotency: simply-typed terms have to be *linear*. These worst-case bounds are a forewarning for the last section, where we show that specifically for System-$\mathbb{I}$, it is not merely worst case, but for *every case*, type inference and normalization are the same.

We obtain the upper bound on normalization by tracing how the typing of a term is transformed into a typing of the contractum. Recall the insight of Kfoury et al. [24] that a *complete development* reduces the rank of the redexes in a term by at least one; consequently a term with redexes of rank r is turned into a term where all redexes are rank 0 in $r-1$ complete developments; a term where all redexes are rank 0 term is affine and can therefore be reduced in linear time. The key insight comes from observation of a redex of form $((\lambda x_{\cdot}.P^{\overline{\tau}})^{(\overline{\tau}_1\wedge\cdots\wedge\overline{\tau}_n)\to\overline{\tau}}\,Q^{\overline{\tau}_1\wedge\cdots\wedge\overline{\tau}_n})^{\overline{\tau}}$; it contracts to $P^{\overline{\tau}}[Q/x]$ where each occurrence of Q has one of the types $\overline{\tau}_1,\ldots,\overline{\tau}_n$. New redexes are only created when Q is an abstraction, and substituted as the operator of an application; all such redexes satisfy $\mathsf{rank}(\overline{\tau}_i) < \mathsf{rank}((\overline{\tau}_1\wedge\cdots\wedge\overline{\tau}_n)\to\overline{\tau})$ for some i. Only when the re-

[5] Note that the type environment of System-$\mathbb{I}$ contains exactly the free variables of the term—consequently the two subderivations have the same variables in the type environment.

[6] Positive and negative occurrence are used in the standard way: count the number of times we choose the left child (the argument) of an arrow in the syntax tree when going from the root to the occurrence. If the number is even, the occurrence is positive; if it is odd, the occurrence is negative.

dex is linear, i.e., $n = 1$, does reduction fail to reduce the rank of the redex. We therefore remove all linear redexes before each round of complete developments.

Since System-$\mathbb{I}$ does not have subject reduction due to the rigidity of $\wedge$, it is non-trivial to relate the System-$\mathbb{I}$ derivations of a term and its contractum. We therefore introduce a relaxed type system $\vdash_{\mathrm{ACW}}$.

Definition 3.1. The ACW type system is as System-$\mathbb{I}$ with the following changes:

(1). The types are $\overline{\tau} ::= a \mid \tau \to \overline{\tau}$ and $\tau ::= \{\overline{\tau}, \ldots, \overline{\tau}\}$.

(2). For $n, m \geq 1$, take $\{\overline{\tau}_1, \ldots, \overline{\tau}_n\} \wedge \{\overline{\tau}_{n+1}, \ldots, \overline{\tau}_{n+m}\} = \{\overline{\tau}_1, \ldots, \overline{\tau}_{n+m}\}$.

(3). The variable rule is

$$\frac{}{\Gamma, x : \{\overline{\tau}_1, \ldots, \overline{\tau}_n\} \vdash x : \overline{\tau}_i}\ \mathrm{Var} \quad \text{where } 1 \leq i \leq n$$

(4). The rank of an ACW-type is:

$$\begin{aligned} \mathsf{rank}(\mathrm{a}) &= 0 \\ \mathsf{rank}(\tau \to \overline{\tau}) &= \max\{\mathsf{rank'}\,\tau, \mathsf{rank}\,\overline{\tau}\} \\ \mathsf{rank}(\{\overline{\tau}_1, \ldots, \overline{\tau}_n\}) &= \max\{\mathsf{rank}\,\overline{\tau}_1, \ldots, \mathsf{rank}\,\overline{\tau}_n\} \\ \mathsf{rank'}(\{\overline{\tau}\}) &= \mathsf{rank}\,\overline{\tau} \\ \mathsf{rank'}(\{\overline{\tau}_1, \overline{\tau}_2, \ldots, \overline{\tau}_n\}) &= 1 + \\ &\quad \max\{\mathsf{rank}\,\overline{\tau}_1, \ldots, \mathsf{rank}\,\overline{\tau}_n\} \end{aligned}$$

(5). We define the following mappings from System-$\mathbb{I}$-types to ACW-types: $\|a\|_{\overline{T}} = a$, $\|\tau \to \overline{\tau}\|_{\overline{T}} = \|\tau\| \to \|\overline{\tau}\|_{\overline{T}}$, $\|\overline{\tau}\| = \{\|\overline{\tau}\|_{\overline{T}}\}$, and $\|\tau \wedge \tau'\| = \|\tau\| \wedge \|\tau'\|$ and extend it trivially to environments. □

REMARK 3.2. The differences between $\vdash_{\mathbb{I}}$ and $\vdash_{\mathrm{ACW}}$ are that the latter allows weakening and strengthening, has an associative and commutative $\wedge$, and that we allow intersections as the argument type of a K-abstraction. Moreover, it is straightforward to prove that typability and rank are preserved under $\|\cdot\|$. □

We now introduce linear reductions and the necessary machinery to expose implicit redexes.

Definition 3.3. We define the following notions of reduction

$$\begin{array}{lcll} (\lambda x.P)\,Q & \mathrm{l} & P[Q/x] & \text{when } \#_x(P) = 1 \\ (\lambda x.P)\,Q\,R & \mathrm{T} & (\lambda x.P\,R)\,Q & \end{array}$$

□

Lemma 3.4. *(1)* Tl*-reduction is strongly normalizing, (2) The* Tl*-normal form can be computed in* $\mathrm{DTIME}[|M|^3]$. *(3) If* $M \to_{\mathrm{Tl}} N$ *for terms* M *and* N *then there is a term* O *such that* $M \to_\beta O$ *and* $M \twoheadrightarrow_\beta O$ □

We will trace how the rank of redexes in the term is decreased by reduction; we define ι as the maximal rank of a redex.

Definition 3.5. Given a derivation Δ of $\Gamma \vdash_{\mathrm{ACW}} M : \tau$, the maximal rank of a β-redex is $\iota(\Delta)$. Formally, we take

$$\begin{aligned} \iota(\Delta) = \max\{\,\mathsf{rank}\,\tau \mid \exists p.\,(\tau \in \mathsf{typ}(\Delta, p_{\mathrm{L}}) \\ \&\ \mathsf{sub}(M, p) = (\lambda x.P)\,Q)\,\} \end{aligned}$$

when $M \notin \mathrm{NF}_\beta$ and $\iota(M) = 0$ otherwise. □

Clearly, $\iota(\Delta) \leq \mathsf{rank}(\Delta)$. We first establish a substitution lemma:

Lemma 3.6. *Let* Δ_0 *derive* $\Gamma_0, x : \{\overline{\tau}_1, \ldots, \overline{\tau}_n\} \vdash M : \tau_0$ *and for* $i = 1, \ldots, n$ *let* Δ_i *derive* $\Gamma_i \vdash N : \overline{\tau}_i$. *We then have an* ACW*-derivation* Δ' *of* $\Gamma_0 \wedge \cdots \wedge \Gamma_n \vdash M[N/x] : \tau_0$ *where* $\iota(\Delta') \leq \max\{\iota(\Delta_0), \ldots, \iota(\Delta_n), \mathsf{rank}\,\overline{\tau}_1, \ldots, \mathsf{rank}\,\overline{\tau}_1\}$. □

We now show that Tl-reductions have subject reduction and only affects the rank in benign way:

Lemma 3.7. *Let* M *and* N *be terms such that* $M \to_{\mathrm{Tl}} N$ *and let* M *have the* ACW*-typing* $\langle \Gamma; \tau \rangle$ *derived with the derivation* Δ. *(1) There is an* ACW*-derivation* Δ' *of* $\Gamma \vdash N : \tau$. *(2)* $\iota(\Delta') \leq \iota(\Delta)$. *(3) If* $M \not\equiv \lambda z.S$ *and* $N \equiv \lambda z.T$, *then* $\mathsf{rank}\,\tau \leq \iota(\Delta)$. □

To bound overall normalization, we follow a fairly standard calculation that pairs Tl *developments* (maximal sequences of Tl reductions) with *complete developments*; after a Tl-development, a complete development must reduce the redex rank by 1, while increasing term size by at most an exponential.

A *complete development* of a term is intuitively this: take a term, and underline every existing redex. Now β-reduce each of them, where redexes *copied* by a β-step (called *residuals*) remain underlined, but *new* redexes (caused by substituting a λ-abstraction for x in some application xP) are not underlined. More formally, define $D(\lambda x.P) = \lambda x.D(P)$, $D(x\,P_1 \cdots P_n) = x\,D(P_1) \cdots D(P_n)$, and the *interesting case* (where $n \geq 0$ is chosen maximally)

$$D((\lambda x.P)\,Q_0 \cdots Q_n) = D(P)[D(Q_0)/x]\,D(Q_1) \ldots D(Q_n)\ .$$

Proposition 3.8. *Let* $M \in \mathsf{NF}_{\mathrm{Tl}}$ *be* λ*-term without* Tl*-redexes and with an* ACW*-derivation* Δ *of the typing* $\langle \Gamma; \tau \rangle$. *We have (1) There is a* ACW*-derivation of* $\Gamma \vdash D(M) : \tau$. *(2)* $\iota(\Delta') < \iota(\Delta)$ *if, and only if,* $\iota(\Delta) > 0$. *(3)* $|D(M)| \leq 2^{|M|}$ *(4)* $M \twoheadrightarrow_\beta D(M)$. □

Proof. Part (1) and Part (2) are proven by induction on the height of the derivation. The only interesting case is the last one above for complete developments: As the term is in Tl-normal form, we can only have $D((\lambda x.P)\,Q) = D(P)[D(Q)/x]$ with $\#_x(P) \geq 2$. All $\tau \in \mathsf{typ}(\Delta, (\lambda x.P)\,Q)$ are therefore on the form $\tau \equiv \{\overline{\tau}_1, \ldots, \overline{\tau}_n\} \to \overline{\tau}'$ and new redexes will have $\mathsf{rank}\,\overline{\tau}_i < \mathsf{rank}\,\tau$ for some i. Part (3) is proven by induction on M; in the only *interesting case* (see above) we have

$$\begin{aligned} |D((\lambda x.P)\,Q)| = |D(P)[D(Q)/x]| &\leq |D(P)||D(Q)| \\ &\leq 2^{|P|+|Q|} \leq 2^{|(\lambda x.P)\,Q|}\ . \end{aligned}$$

The last part is [5, Lem. 11.2.28(i)]. □

Corollary 3.9. *Given a term* M, *its complete development* $D(M)$ *can be computed in* $\mathrm{DTIME}[2^{|M|}]$. □

Theorem 3.10. *Let* M *be a term typable in rank* r. *Its normal form has length* $O(\mathbf{K}(r-1, |M|))$ *and can be computed in* $\mathrm{DTIME}[\mathbf{K}(r-1, |M|)]$. □

Proof. It follows from Lem. 3.7 and Prop. 3.8 that Tl-normalization followed by a complete development of a term P produces a term Q with $|Q| \leq 2^{|P|}$. There is a term R such that $P \twoheadrightarrow_\beta R \twoheadleftarrow_\beta Q$. If P is typable in redex rank r, then Q is typable in redex rank $k-1$. By induction, at most $r-1$ repetitions produce a redex rank 0 type which by Tl-normalization becomes a β normal form N. Using the Church-Rosser property, $N = \mathsf{NF}_\beta(M)$. □

This theorem shows how to construct untypable terms: just violate the normalization bounds. We recall $\overline{n}$ as the Church numeral for n, the application $\overline{2}\,\overline{2}\,\overline{2}$ is not typable in rank 3. Note that $P = (\lambda x.x\,\overline{2})\,\overline{n} \twoheadrightarrow_\beta \overline{2^n}$: observe that for any rank, $\lambda k.k\,P\,\overline{2}$ cannot be typed for almost all $\overline{k}$. Terms untypable in ML, System F, etc. are often anomalous and peculiar; those untypable in rank-bounded[7] System-$\mathbb{I}$ are the standard ones that are the mainstay of programming with inductive datatypes in the absence of fixpoint recursion—see the Conclusions.

4 Lower bounds on type inference

To derive a lower bound on type inference, we simulate a Turing Machine in the type system. Following an idea by Henglein and Mairson [17], we observe that when a term is linear, its principal typing is isomorphic to its normal form.

Consider for instance linear λ-terms representing boolean `true`, `false`, and `not`: $\lambda x\lambda y.x$, $\lambda x\lambda y.y$, and $\lambda p\lambda x\lambda y.p\,y\,x$. Their principal types are $a \to b \to a$, $a \to b \to b$, and $(a \to b \to c) \to b \to a \to c$; it is easy to see that there is only one linear normal form with each of these types. More interesting is to consider the application `not true`; it has principal type $b \to a \to a$ which is a principal type of `false`. Incidentally, $(\lambda p\lambda x\lambda y.p\,y\,x)\,\lambda x\lambda y.x \twoheadrightarrow_\beta \lambda x\lambda y.y$. Type inference has effectively simulated normalization—a technique further elaborated in Section 5

In order to derive a lower bound on type inference, we represent Turing machine IDs as closed, linear λ-terms. A Turing machine going through IDs $s_0, s_1, \ldots, s_n$ is then simulated by terms $M_0, M_1, \ldots, M_n$ of types $\overline{\tau}_0, \overline{\tau}_1, \ldots, \overline{\tau}_n$. We represent the state transition function as a closed, linear λ-term δ where $\delta M_i \twoheadrightarrow_\beta M_{i+1}$ and similar in types $\delta M_i : \overline{\tau}_{i+1}$. We tie the knot by proving the following lemma:

Lemma 4.1 (Polymorphic Iteration). *Let $N = \mathbf{K}(r,n)$, and $\overline{m}$ be the Church numeral for m. Let ϕ be a term that can be given any of the simple types $\overline{\tau}_i \to \overline{\tau}_{i+1}$ for $0 \le i < N$, where the $\overline{\tau}_i$ are arbitrary. Define $J_n = \lambda z.\overline{2}^n\,z$; the term $\Phi = J_n\,\overline{2} \cdots \overline{2}\,\phi$ (with $r-1$ applications on $\overline{2}$s) reduces to $\lambda x.\phi^N\,x$, and has simple type $\overline{\tau}_0 \to \overline{\tau}_N$ in rank r.* □

The proof is essentially as by Kfoury et al. [24] with a more tedious packing of the types due to the rigidity. When we choose $\phi = \delta$ the term ΦM_0 of length $O(r+n)$ simulates a Turing machine computation of up to N steps in *both the types and in the terms*. It therefore provides a lower bound on both type inference and normalization; this shows that our upper bounds given above are tight.

We use the following conventions regarding Turing machines; it should be clear to the reader how to represent her favorite machine.

Convention 4.2. Consider Turing machines which decide predicates, coded on a right-infinite tape over alphabet ␣, `0`, and `1`. The program never goes beyond the leftmost symbol, never writes a blank symbol, and moves the head exactly when it reads a `0` or a `1`. A given Turing machine has l states $q_1, \ldots, q_l$. When the program is done, it loops in the designated failure state q_{l-1} or acceptance state q_l. □

We build the transition function and states with the linear λ-terms in Fig. 4: We use the standard representation of pairs to build lists as pairs (in contrast with the usual inductive representation); this implies that there is no uniform type for list. We represent a machine state as a tuple $\langle q, \mathbf{L}, \mathbf{R}\rangle$ consisting of the program counter q, the tape $\mathbf{L}$ to the left of the head in reverse, and the tape $\mathbf{R}$ to the right of the head. The transition function is a table lookup: using q and the symbol under the head r, find the new state q', the new symbol under the head r, and the direction of the head d. This is represented by δ: by our conventions, a blank is only read at the right end of the tape; in this case the tape is explicitly extended with a blank (the third line of δ).

Pairs and lists:

$$\langle \psi;\phi\rangle = \lambda z z\,\psi\,\phi$$
$$\mathrm{nil} = \lambda z.z$$
$$[x_1,\ldots.x_k] = \langle x_1, \langle x_2, \langle \cdots \langle x_k, \mathrm{nil}\rangle \cdots\rangle\rangle$$

Sequencing (right associative):

$$(\langle v_1,\ldots,v_k\rangle := \psi;\phi) = \psi\,(\lambda v_1.\cdots\lambda v_k.\phi)$$

State, symbols, head movement:

$$\mathrm{PC}_i = \lambda q_0.\cdots\lambda q_l.q_i$$
$$␣ = \lambda B\lambda Z\lambda O.B \qquad \mathtt{0} = \lambda B\lambda Z\lambda O.Z \qquad \mathtt{1} = \lambda B\lambda Z\lambda O.O$$
$$\rightarrow = \lambda x\lambda y\lambda z.x \qquad \downarrow = \lambda x\lambda y\lambda z.y \qquad \leftarrow = \lambda x\lambda y\lambda z.z$$

State transition function δ

$$\begin{aligned}&\lambda I.\langle q,\mathbf{L},\mathbf{R}\rangle := I;\langle l,L\rangle := \mathbf{L};\langle r,R\rangle := \mathbf{R};\langle q',r',d'\rangle := \delta\,q\,r;\\ &\quad (d'\ (\lambda q'\lambda L\lambda l\lambda r'\lambda R.\langle q',L\rangle,\langle l,\langle r',R\rangle\rangle)\\ &\qquad (\lambda q'\lambda L\lambda l\lambda r'\lambda R.\langle q',\langle l,L\rangle,\langle r',\langle ␣,\mathrm{nil}\rangle\rangle)\\ &\qquad (\lambda q'\lambda L\lambda l\lambda r'\lambda R.\langle q',\langle r',\langle l,L\rangle\rangle,R\rangle))\,q'\,L\,l\,r',R\end{aligned}$$

Figure 4. The building blocks of a Turing machine encoding. The types are omitted: as the terms are linear they are isomorphic to the terms.

The Polymorphic Iteration Lemma is originally proved in a setting with ACI [24]. One application of this lemma *with* ACI is to let Φ be the term $\lambda x.x\overline{2}$, which is *not* linear—then the normal form takes an *enormous* leap to $\mathbf{K}(\mathbf{K}(t,n),1)$. But without ACI, the "base" calculus is the *linear* λ-calculus, not the *simply-typed* one, and there is no such boost. Finally, using standard machinery (e.g., [22, 17]) we may conclude:

Theorem 4.3. *Let P be a λ-term of length n; then deciding if P is typable in rank t is complete for* DTIME$[\mathbf{K}(t,n)]$. □

5 Type inference is never cheaper than normalization

We have now reached the high tide of the paper, and prove that due to the lack of idempotency, type inference is the same as normalization. Thus type inference *cannot be faster* than running the program, and types can be inferred without nominally solving constraints—they can be read off of the normal form. It also suggests a Curry-Howard isomorphism for System-$\mathbb{I}$ (and possible intersection types in general) through interaction nets. Furthermore, it shows that the type inference bounds are, literally, Statman's theorem for this type system.

[7] Recall from the introduction that rank-unbounded System-$\mathbb{I}$ is irrelevant for practice as type inference is undecidable.

The result is obtained through an isomorphism between the normal form of a net, and its principal typing.[8] We first establish the isomorphism for the restricted case of normal nets. In this case, we can read the net as its own principal typing, and dually, construct a normal net from a given principal typing; we obtain algorithms similar to Sayag and Mauny [42]. In the general case, we show that the set of typings is unchanged under reduction. As System-$\mathbb{I}$ is strongly normalizing for fixed rank, it follows that any net has the same principal typing as its normal form.

5.1 Principal typing from a normal form

We obtain the principal typing from the normal form by reading sharing nodes as $\wedge$, function and application nodes as $\to$, and boxes as expansion variables. We collect the principal typing using the recursive algorithm outlined in Fig. 5; it is called $\mathsf{tp} : I \hookrightarrow T$. The base case corresponds to the normal form $\lambda x_1.\cdots\lambda x_k.v$ with the typing $(\emptyset, \gamma_1 \to \cdots \to \gamma_k \to \gamma_i)$ when $v = x_i$ and $(v : \alpha, \gamma_1 \to \cdots \to \gamma_k \to \alpha)$ when v is free. The algorithm returns the typing by annotating the root and the free ports with the type information, e.g., in the second case, the root gets type $\gamma_1 \to \cdots \to \gamma_k \to \alpha$, and the free port gets type α.

Otherwise, the graph represents $\lambda x_1.\cdots\lambda x_k.vN_1\cdots N_l$. Apply the algorithm recursively to each subnet Π_i (corresponding to N_i); the result is type annotations of the root and the free ports of each Π_i. We assign a fresh expansion variable F_i to each of the boxes. We assign type $\mathrm{F}_l\,\tau_0^l \to \cdots \to \mathrm{F}_1\,\tau_0^1 \to \alpha$ to the head variable and type $\mathrm{F}_i\,\pi_j^i$ to each wire from Π_i with π_j^i the type returned by the recursive call. Finally, we find the intersection type of each shared variable, by a bottom-up traversal of the sharing forest. Each sharing node is given the principal port type $\tau_\circ \wedge \tau_\bullet$, where $\tau_\circ$ ($\tau_\bullet$) is the type of the white (black) auxiliary port.

5.2 Normal form from a principal typing

Dually, the function $\mathsf{net} : (V_\Lambda \hookrightarrow T) \times T \to \mathrm{NF}_\Rightarrow$ produces a net normal form of any given principal System-$\mathbb{I}$ typing.

Algorithm net is founded on the intuition all functional programmers have when gazing the structure of a function from its type: she immediately knows that a function of type $(\alpha \to \beta) \to \alpha \to \beta$ has two outermost abstractions and that the first argument appears in the function position of an application. What sets System-$\mathbb{I}$ apart from other functional languages is the *rigidity:* without idempotency, we have a type specification for each variable occurrence; lack of commutativity gives the orientation of sharing nodes, and non-associativity tells us the boxes; finally, the principal typing reveals the sources of the arguments of applications. For instance, $\overline{2}$ can be typed $(\alpha \to \alpha) \wedge (\alpha \to \alpha) \to \alpha \to \alpha$; it is not clear whether it is the first or the second occurrence of the first argument that is applied to the second argument. However, when looking at the principal type, $((\mathrm{F}\,\beta \to \gamma) \wedge \mathrm{F}\,(\mathrm{G}\,\alpha \to \beta)) \to \mathrm{F}\,\mathrm{G}\,\alpha \to \gamma$, it is clear that it is the second occurrence.

Definition 5.1. The function $\mathsf{net} : (V_\Lambda \hookrightarrow T) \times T \hookrightarrow \mathrm{NF}_\Rightarrow$ is defined on any principal typing $\langle \Gamma; \tau \rangle$ (see Lemma 2.4). We proceed as follows: (1) Let net^+ and net^- be the mutual recursive functions in Fig. 6. Produce a forest of graphs by applying net^+ to τ and net^- to each type in Γ. For example, $\mathsf{net}^+(\tau \to \tau')$ constructs

[8] Strictly speaking, there is a plethora of principal typings. They differ, however, only in the choice of names for the type and expansion variables.

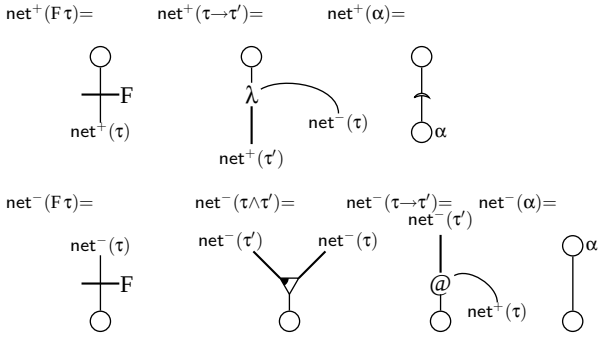

Figure 6. The mutual recursive definitions of the functions net^+ and net^- building the skeleton of a net from a type.

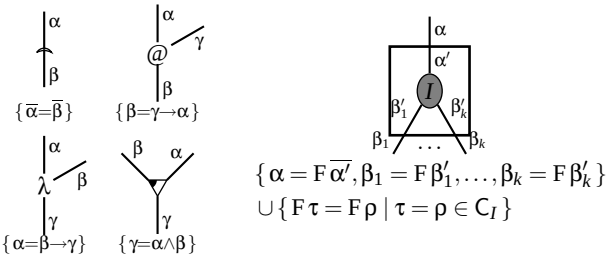

Figure 7. The typing constraints for the various interaction nodes. In the right subfigure, F is the expansion variable of the box.

proofnets from $\mathsf{net}^+(\tau')$ and $\mathsf{net}^-(\tau)$, connecting them with a λ-node. (2) Connect the ports of each type variable that is mentioned twice. Connect the remaining ports to a weakening node. □

Proposition 5.2. *Let $I \in \mathrm{NF}_\Rightarrow$ be a net with principal typing $\langle \Gamma; \tau \rangle$. Then $I = \mathsf{net}(\langle \Gamma; \tau \rangle)$ and $\mathsf{tp}(I) = \langle \Gamma; \tau \rangle$.* □

5.2.1 Typings Are Preserved Under Reduction

Going from the easier case of normal forms to arbitrary nets, we establish that the set of typings is invariant under reduction. From this analysis we derive both subject reduction, and subject expansion.[9]

We first recast the question of typability as a constraint problem. We show that every interaction net has a set of *typing constraints*, which exactly characterizes its typings. Second, we show that the set of solutions is invariant under reduction. It follows that a net and its normal form have the same set of typings. As a principal typing is a typing from which all other typings can be obtained, a net and its normal form then have the same principal typing.

Definition 5.3. Let $I \in \mathcal{I}$ be a net. Let each wire in I be labeled with a unique type variable and each box be labelled with a unique expansion variable.

(1). The *typing constraints* C_I of I is a set of equations $\tau = \sigma$; some occurrences of type variable α can be labelled $\overline{\alpha}$ constraining α to types from $\overline{T}$. The nodes induce the *typing*

[9] Recall subject expansion as the feature that if M reduces to N then $N : \langle \Gamma; \tau \rangle$ implies $M : \langle \Gamma, \tau \rangle$.

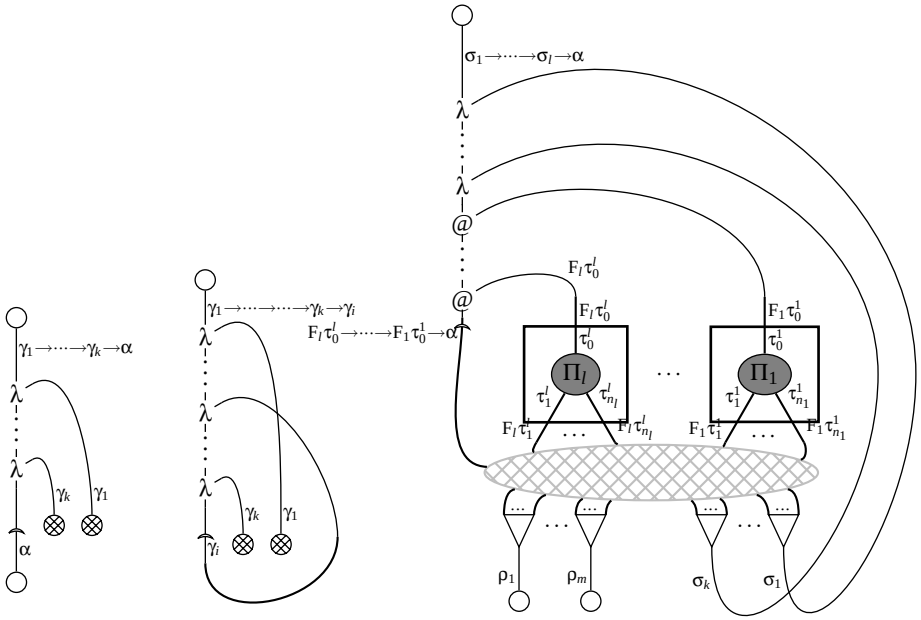

Figure 5. Inductive definition of the function tp that derives a principal typing of a net. In the right subfigure, the hatched oval is the wiring between the sharing forest at the bottom and the wires coming from the croissant and the free ports of the subnets $\Pi_1, \ldots, \Pi_l$.

In the two cases to the left, the variables $\alpha, \gamma_1, \ldots, \gamma_k$ are fresh. In the last case, the variables $\alpha, F_1, \ldots, F_l$ are fresh. The algorithm is applied recursively to the darked subnets $\Pi_1, \ldots, \Pi_l$ to obtain the types $\tau_0^i, \ldots, \tau_{n_i}^i$ for all *i*. The result types $\rho_1, \ldots, \rho_m, \sigma_1, \ldots, \sigma_k$ are built by processing the sharing trees bottom-up as described in the text.

constraint to the left of Fig. 7. For a box labelled F, let C be union of the typing constraints for the nodes and boxes inside the box; the typing constraints for F are $\{\alpha = \mathrm{F}\,\overline{\alpha'}, \beta_1 = \mathrm{F}\,\beta_1', \ldots, \beta_k = \mathrm{F}\,\beta_k'\} \cup \{\mathrm{F}\,\tau = \mathrm{F}\,\sigma \mid \tau = \sigma \in \mathsf{C}\}$ where α (α') is the type variable of the root outside (inside) the box and β_i (β_i') is the type variable of the *i*th auxiliary port outside (inside) the box. C_I is the union of the constraints from the outermost boxes and the nodes outside boxes.

(2). A *solution* U to a set of typing constraints C is pair consisting of an expansion substitution E and a variable substitution T such that $\mathsf{T}(\mathsf{E}\,\tau) = \mathsf{T}(\mathsf{E}\,\sigma)$ for all $\tau = \sigma \in \mathsf{C}$. We restrict E and T to the variables occurring in C. Also, T respects labels, i.e., $\mathsf{T}(\alpha) \in \overline{T}$ if α has a labelled occurrence in C. □

To prove an isomorphism between typings of a fixed net I, and solutions to C_I, we need to relate proofnets with solutions U to C_I. Both specify an intersection type for each wire: The proofnet through the annotation $\mathsf{S}(e\,\tau)$ of each wire. A solution $\mathsf{U} = (\mathsf{E}, \mathsf{T})$ inserts expansions from the substitution E for the expansion variables $\vec{G} = (G_1, \ldots, G_k)$, which annotate boxes surrounding a wire (listed from outermost to innermost); the substitution T provides a type for the expanded derivatives of γ, the wire's type variable (created by applying $\mathsf{E} \circ \vec{G}$ to γ). We consider the proofnet Π and the solution U *equivalent on a wire* exactly when $\mathsf{S}(e\,\tau) = \mathsf{T}(\mathsf{E}(\vec{G}\,\gamma))$.

Proposition 5.4. *Let I be a net and C its typing constraints. There exists a solution (T, E) to C for all proofnets Π typing I such that (T, E) and Π are equivalent on all wires. Furthermore, given a solution (T, E) to C there exists a proofnet Π typing I such that (T, E) and Π are equivalent on all wires.* □

It is now sufficient to prove that the set of solutions are preserved under reduction. At the intuitive level, the preservation holds because C_I capture what happens under reduction. As an example consider a croissant dissolving an outermost box: looking at the wire with the croissant there are 3 relevant type variables in the constraint set: α above the croissant, β between the box and the croissant, and γ on the inside of the box. The constraints are $\overline{\alpha} = \overline{\beta}$ and $\beta = \mathrm{F}\,\overline{\gamma}$. Any solution (T, E) will have $\mathsf{T}(\mathsf{E}\,\alpha) = \mathsf{T}(\mathsf{E}(\mathrm{F}\,\gamma))$; as α, β, and γ are type variables the expansion substitution E has no effect and we have: $\mathsf{T}\,\alpha = \mathsf{T}\,\beta$ and $\mathsf{T}\,\beta = \mathsf{T}(\mathrm{F}\,\gamma)$. To respect the labellings, T must substitute a $\overline{T}$-type for β so we have $\mathrm{F} = \Box$. Consequently, all solutions have $\mathsf{T}\,\alpha = \mathsf{T}\,\beta = \mathsf{T}\,\gamma$. This is in accordance with the fact that after dissolving the box we have a single wire. Furthermore, it allows any solution to the reduced net to be used on the original net. (If the redex is inside a box we have $\mathsf{T}(\mathsf{E}(\vec{G}\,\overline{\alpha})) = \mathsf{T}(\mathsf{E}(\vec{G}\,\overline{\beta}))$ and $\mathsf{T}(\mathsf{E}(\vec{G}\,\beta)) = \mathsf{T}(\mathsf{E}(\vec{G}(\mathrm{F}\,\overline{\gamma})))$ where $\vec{G}$ are the expansion variables of the boxes; we essentially get a list of constraints similar to the outermost one.)

In establishing the preservation, the main technical difficulty is that a redex and its contractum have a different number of wires. We solve this by noting the redundancy of solutions—in the example above, choosing the substitution for the instances of α leaves only one choice for the instances of β and γ. It is thus sufficient to relate the two typing constraints on a subset of variables which can be extended uniquely to a solution. We call such a set a *basis*.

Definition 5.5.

(1). A *basis* B of a set of typing constraints C is a subset of the variables occurring in C such that: For any type substitution R and expansion substitution D specified on the variables of B, there is at most one solution (T, E) of C such that $\mathsf{R}(x) = \mathsf{T}(x)$ for any $x \in \mathrm{dom}\,\mathsf{R}$ and $\mathsf{D}(\mathrm{F}) = \mathsf{E}(\mathrm{F})$ for any $\mathrm{F} \in \mathrm{dom}\,\mathsf{D}$.

(2). Given a basis B and solution $\mathsf{U} = (\mathsf{S}, \mathsf{E})$ to a set of typing

constraints C, we write $\mathsf{U}(B)$ to denote the restriction of a solution to the variables in B.

(3). Let C be a set of typing constraints, B a basis of C, and (R,D) a pair of substitutions on B. If it exists, we write $\lceil(\mathsf{R},\mathsf{D})\rceil_{\mathsf{C}}$ for the unique solution to C coinciding with R and D on B. □

We prove that the set of solutions to the typing constraints is invariant under reduction: for each reduction rule, choose two bases, one for the net and one for the reduced net, with an immediate connection between the wires and boxes in the two bases. The solutions are related by restricting them to the bases, using the mapping, and extending them uniquely.

Proposition 5.6. *Let I and J be nets such that $I \Rightarrow J$. Let C_I (C_J) be the typing constraints of I (J). There exists bases B_I of C_I and B_J of C_J, such that 1) $\lceil \mathsf{U}_I(B_I)\rceil_{\mathsf{C}_J}$ exists and is a solution to C_J when U_I is a solution to C_I and 2) $\lceil \mathsf{U}_J(B_J)\rceil_{\mathsf{C}_I}$ exists and is a solution to C_I for any solution U_J of C_J.* □

Strictly speaking we cannot be sure that the variables in C_I and C_J overlap; we have however without loss of generality assumed that the unchanged wires have the same type variables. Subject reduction and expansion follow as does completeness using the strong normalization result in Sec. 3.

Corollary 5.7. *The set of typings is invariant under proofnet reduction.* □

Corollary 5.8. *If I is a net with principal typing $\langle\Gamma;\tau\rangle$ and $\mathrm{NF}(I) = J$, then $\mathsf{net}(\langle\Gamma;\tau\rangle) = J$, and $\mathsf{tp}(J) = \langle\Gamma;\tau\rangle$.* □

Since net reduction simulates β-reduction, the following folklore theorem follows from subject expansion and the fact that all normal forms are typable.

Corollary 5.9. *Let M be a strongly normalizable λ-term. Then M is typable in System-$\mathbb{I}$.* □

6 Conclusions

Idempotency is crucial. Without idempotency, the rank 0 types form only the *linear* λ-calculus, expressiveness of the language collapses to the same complexity as type inference, and type inference becomes synonymous with normalization. Previously [24], one of us conjectured that the complexity of type inference, and that of normalization, are related by the $\log^*$ function for suitably simple and "reasonable" type systems. The "reasonable" characteristic is clearly whether the type system is built on simply-typed λ-calculus. Without idempotency, it can't be, and so the conjecture fails.

Without idempotency, every Church numeral has a different type, and generalizations of this observation to similar data representations (lists, trees, etc.) are clear. As a consequence, typing *functions* on those datatypes makes no sense. A conceivable rebuttal: normal people don't program with Church numerals—they program with *real* numbers. But iteration is one of the functional programmer's weapons of mass construction: for example `iter 0 s z = z`, `iter n+1 s z = s(iter n s z)`—but now, if `iter` is to have type $\mathsf{Int} \to \alpha$, what is α? Even worse, typing simple programming examples, such as Abelson and Sussman's square-root program via iterative approximation [1], become *compiler* exercises in numerical analysis.

The interaction net vernacular underlines the similarities between intersection types and linear logic. Sharing nodes capture the behavior of non-ACI features of $\wedge$. Boxes capture the essence of expansion variables—the renaming of expansions is similar to the copying of a box. Brackets capture the essence of absorption, and the propagation of an expansion through a type formula.

An obvious question is how to preserve important aspects of this type analysis, while allowing the expressiveness of the language to increase substantially beyond the cost of typing. A very related problem is to design type systems whose expressiveness allow intermediate levels of static analysis, between all and nothing. Could static analysis, for example, learn anything from limited forms of polymorphism?

7 References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] R. Amadio and P.-L. Curien. *Domains and Lambda Calculi*, volume 46 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1998.

[3] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages.* Cambridge University Press, 1998.

[4] A. Banerjee. A modular, polyvariant, and type-based closure analysis. In *Proc. 1997 Int'l Conf. Functional Programming*. ACM Press, 1997.

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

[6] S. Carlier, J. Polakow, J. B. Wells, and A. J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In *Programming Languages & Systems, 13th European Symp. Programming*, volume 2986 of *LNCS*, pages 294–309. Springer-Verlag, 2004.

[7] S. Carlier and J. B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with β-reduction. Technical Report HW-MACS-TR-0012, Heriot-Watt Univ., School of Math. & Comput. Sci., Jan. 2004.

[8] M. Coppo, F. Damiani, and P. Giannini. Strictness, totality, and non-standard type inference. *Theoret. Comput. Sci.*, 272(1-2):69–111, Feb. 2002.

[9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematishche Logik und Grundlagen der Mathematik*, 27(1):45–58, 1981.

[10] F. Damiani. A conjunctive type system for useless-code elimination. *Math. Structures Comput. Sci.*, 13:157–197, 2003.

[11] F. Damiani. Rank 2 intersection types for local definitions and conditional expressions. *ACM Trans. on Prog. Langs. & Systs.*, 25(4):401–451, 2003.

[12] F. Damiani. Rank 2 intersection types for modules. In *Proc. 5th Int'l Conf. Principles & Practice Declarative Programming*, pages 67–78, 2003.

[13] F. Damiani and P. Giannini. Automatic useless-code detection and elimination for HOT functional programs. *J. Funct. Programming*, pages 509–559, 2000.

[14] V. Danos and L. Regnier. The structure of multiplicatives. *Arch. Math. Logic*, 26, 1989.

[15] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Conf. Rec. 19th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 15–26, 1992.

[16] G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *Proc. 7th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 223–34. IEEE Comput. Soc. Press, 1992.

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

[18] J. R. Hindley. The principal type scheme of an object in combinatory logic. *Trans. American Mathematical Society*, 146:29–60, Dec. 1969.

[19] J. R. Hindley and J. P. Seldin, editors. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[20] T. Jensen. Inference of polymorphic and conditional strictness properties. In *Conf. Rec. POPL '98: 25th ACM Symp. Princ. of Prog. Langs.*, 1998.

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

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

[23] A. J. Kfoury. A linearization of the lambda-calculus. *J. Logic Comput.*, 10(3), 2000.

[24] A. J. Kfoury, H. G. Mairson, F. A. Turbak, and J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming*, pages 90–101. ACM Press, 1999.

[25] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is DEXPTIME complete. In *15th Colloq. Trees in Algebra and Programming*, volume 431 of *LNCS*, pages 206–220. Springer-Verlag, 1990.

[26] A. J. Kfoury, G. Washburn, and J. B. Wells. Implementing compositional analysis using intersection types with expansion variables. In *Proceedings of the 2nd Workshop on Intersection Types and Related Systems*, 2002.

[27] 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. Princ. of Prog. Langs.*, pages 161–174, 1999.

[28] A. J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. *Theoret. Comput. Sci.*, 311(1–3):1–70, 2004.

[29] Y. Lafont. From proof-nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic, Proceedings of the 1993 Workshop on Linear Logic*, London Math. Soc. Lecture Note Series 222, pages 225–247. Cambridge University Press, 1995.

[30] J. Lamping. An algorithm for optimal lambda-calculus reductions. In POPL '90 [38], pages 16–30.

[31] J.-J. Lévy. Optimal reductions in the lambda-calculus. In Hindley and Seldin [19], pages 159–191.

[32] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In POPL '90 [38], pages 382–401.

[33] H. G. Mairson. Outline of a proof theory of parametricity. In J. Hughes, editor, *FPCA '91, Conf. Funct. Program. Lang. Comput. Arch.*, volume 523 of *LNCS*, pages 313–327, Cambridge, MA. U.S.A., Aug. 1991. Springer-Verlag.

[34] H. G. Mairson. From Hilbert spaces to Dilbert spaces: Context semantics made simple. In *22nd Conference on Foundations of Software Technology and Theoretical Computer Science*, 2002.

[35] R. Milner. A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17:348–375, 1978.

[36] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 365–458. North-Holland, 1990.

[37] C. Mossin. Exact flow analysis. *Math. Structures Comput. Sci.*, 13:125–156, 2003.

[38] *Conf. Rec. 17th Ann. ACM Symp. Princ. of Prog. Langs.*, 1990.

[39] L. Regnier. *Lambda calcul et réseaux*. PhD thesis, University Paris 7, 1992.

[40] S. Ronchi Della Rocca. Principal type schemes and unification for intersection type discipline. *Theoret. Comput. Sci.*, 59(1–2):181–209, Mar. 1988.

[41] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoret. Comput. Sci.*, 28(1–2):151–169, Jan. 1984.

[42] É. Sayag and M. Mauny. A new presentation of the intersection type discipline through principal typings of normal forms. Technical Report RR-2998, INRIA, Oct. 16, 1996.

[43] R. Statman. The typed lambda-calculus is not elementary recursive. *Theoret. Comput. Sci.*, 9(1):73–81, July 1979.

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

[45] P. Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Program. Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[46] G. Washburn, A. Kfoury, J. Wells, B. Alan, B. Yates, O. Schwartz, and S. Carlier. System I experimentation tool. `http://types.bu.edu/modular/compositional/experimentation-tool/`.

[47] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999.

[48] J. B. Wells, A. Dimock, R. Muller, and F. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Programming*, 12(3):183–227, May 2002.