

Rank Bounded Intersection: Types, Potency, and Idempotency

Peter Møller Neergaard^{*1} and Harry G. Mairson^{**1}

School of Computer Science
Brandeis University
Waltham, MA 02454, USA
Fax: +1 781 736 2741
`{turtle,mairson}@cs.brandeis.edu`

Abstract. Intersection type systems realize a *finite polymorphism* where different types for a term are itemized explicitly. We analyze System- \mathbb{I} , a rank-bounded intersection type system where intersection is not *associative*, *commutative*, or *idempotent* (ACI), but includes a substitution mechanism employing *expansion variables* that facilitates modular program composition and flow analysis. This type system is used in a prototype intersection type compiler for the Church project [15]. We prove that the problem of type inference is exactly as hard as the problem of normalization: the worst-case cost of both is an elementary function, where the iterated exponential depends on the rank. The key to these results is that simply-typed terms must be linear without ACI, but have the usual nonelementary power with ACI. Further, type inference is *always* synonymous with normalization: the cost of computing the principal typing of any term is exactly the cost of computing its normal form. These results do not hold when AC, and particularly I, is added.

1 Introduction

Poking holes in static type systems is a well-understood *modus vivendi* of programming language design and analysis. These type systems are designed so that programs typable at compile-time do not go wrong at run-time. A type system defect occurs when a well-typed program indeed goes wrong; more benign is when a program that goes right is rejected due to typing problems. But just as an oncologist cannot usefully claim to have cured cancer by killing the patient, a compiler cannot usefully claim to reject type-unsafe programs by rejecting all programs. The technical challenge is to design a type system for an expressive language that always rejects unsafe programs, but accepts as many good programs as possible, while using reasonable compile-time computational resources.

* 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.

Intersection types provide an alternative to the ML/System F paradigm of *parametric* polymorphism, which implements instead a *finite* polymorphism. For example, ML rejects `fn f => (f 3, f true)` since `f` must have monomorphic type, and type `int` does not unify with type `bool`; the intersection type of this program is just $((\text{int} \rightarrow 'a) \wedge (\text{bool} \rightarrow 'b)) \rightarrow 'a * 'b$. Recognizing intersection-typable terms is undecidable because they are exactly the ones that strongly normalize. A practical alternative is to implement *rank-bounded* intersection types, which limit the higher-order functionality of intersection types—the depth of a \wedge -type in an \rightarrow -type is bounded by a constant.

In [7], we analyzed the computational difficulty of this inference, comparing it to the expressive power of the language. Here, we examine related problems for System- \mathbb{I} , where the \wedge is not *associative*, *commutative*, or *idempotent* (ACI). System- \mathbb{I} is the foundation of the Church compiler project [15]: as each variable occurrence is typed individually, typed-based flow analysis between call sites and functions can optimize specific procedure applications.

Life without ACI: What do intersection types look like without ACI? Let \bar{k} be the Church numeral for k . Without associativity, subject reduction is lost:

$$\begin{aligned} \bar{2}\bar{2} &: ((d \rightarrow e) \wedge (c \rightarrow d)) \wedge ((b \rightarrow c) \wedge (a \rightarrow b)) \rightarrow (a \rightarrow e) \\ \bar{4} &: ((d \rightarrow e) \wedge ((c \rightarrow d) \wedge ((b \rightarrow c) \wedge (a \rightarrow b)))) \rightarrow (a \rightarrow e) \end{aligned}$$

even though $\bar{2}\bar{2} \triangleright \bar{4}$. (In the normalization of $\bar{2}\bar{2}$, the types show that the first and last two occurrences of the “successor” parameter in the normal form are copies of each other.) Without commutativity, subject reduction is lost again:

$$\begin{aligned} \lambda x.(\lambda y.\lambda z.\lambda w.wzy)xx &: (b \wedge a) \rightarrow (a \rightarrow b \rightarrow c) \rightarrow c \\ \lambda x.\lambda w.wxx &: (a \wedge b) \rightarrow (a \rightarrow b \rightarrow c) \rightarrow c \end{aligned}$$

(The inversion of y, z in the pairing is detected in the type; even if $a = b$ there is a *twisting*.) Without idempotency, subject reduction is lost once more:

$$\begin{aligned} \lambda z.(\lambda x.\lambda y.x)zz &: (a \wedge a) \rightarrow a \\ \lambda z.z &: a \rightarrow a \end{aligned}$$

The absence of I also means that every Church numeral has a different type, with generalizations of this observation to similar data representations (lists, trees, etc.). As a consequence, typing *functions* on these datatypes is not possible: how can addition on Church numerals be typed as $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ if there is no uniform representation of Int ? The addition function *can* be typed, but in a useless way: one typing for each of the additions of \bar{m} to \bar{n} . Thus we can say in types, “ $\bar{5} + \bar{7} = \bar{12}$ ” but not “if $\bar{m}, \bar{n} : \text{Int}$, then $\bar{m} + \bar{n} : \text{Int}$.”

Normalization bounds: We analyze the computational difficulty of recognizing typable terms and type inference in System- \mathbb{I} , as well as computing bounds on normalization of typable terms. Let $\mathbf{K}(t+1, n) = 2^{\mathbf{K}(t, n)}$, with basis $\mathbf{K}(0, n) = n$;

given a pure (i.e., not type-annotated) λ -term of length n , for System- \mathbb{I} at *rank*¹ t , we prove that the worst-case complexity of *all* of the above problems is $\mathbf{K}(t, n)$. (Worst-case bounds are not the end of the story.) This answers (negatively) an open question about System- \mathbb{I} one of us conjectured in [7]: that in a predicative type system, the respective complexity of recognizing term equivalence, and recognizing typable terms, are related by the \log^* function.

The lack of \mathbb{I} enforces *linearity*, and for this reason, the \mathbb{I} in System- \mathbb{I} is a reminder of its computational impotence. In each term with a *simple* type (i.e., without \wedge), no free or bound variable can occur more than once. These terms normalize in linear time—vastly less powerful than terms with the same types in the simply-typed λ -calculus. The expressiveness bound is a variant of Statman’s theorem [13], but so is the bound on type inference. Both derive from understanding how much a function with different domain and range can be *polymorphically* iterated.

Relating type inference and expressiveness in every case: Unlike almost any language you can think of, in System- \mathbb{I} the difficulty of typing is *equal* to the expressive power. By contrast, simply-typed λ -calculus has polynomial-time type inference and a $\mathbf{K}(n, 2)$ bound on expressiveness; core-ML has a $\mathbf{K}(1, n)$ bound on type inference and a $\mathbf{K}(\mathbf{K}(1, n), 2)$ bound on expressiveness.

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 can’t keep a System- \mathbb{I} type in your head, because you would then know—in advance—exactly what your program was computing.²

To relate type inference and normalization, we use *sharing graphs* for λ -calculus described in [3], which we call *interaction nets* (or just *nets*) when untyped and *proofnets* when containing typing information. The *sharing nodes* of interaction nets (indicating *contraction* in linear logic) represent intersections. We believe, following Regnier [12], that interaction nets are the right way to talk about intersection type systems. Though System- \mathbb{I} lacks subject reduction, the analogous interaction net reductions preserve subject reduction.

To show type inference is synonymous with normalization for each term, we analyze type inference for System- \mathbb{I} including its introduction of *expansion variables*, proposed to facilitate the engineering of *compositionality* between independently compiled pieces of code. Expansion variables are the System- \mathbb{I} syntax for *boxes* from interaction nets.

¹ In this analysis, *rank* is defined differently from that in [7] since there is no ACI: we measure instead the *alternation* of \rightarrow and \wedge in types.

² The lower bound on ML typing resulted from a perverse form of computing where type inference and term normalization were “doing the same thing”—in contrast, we prove that rank-bounded intersection without ACI is, to misquote Freud, an example of *pervasively* polymorphous perversion: inference is *always* synonymous with normalization.

We prove that a principal System- \mathbb{I} typing of a λ -term can be computed by coding the term as a sharing graph, normalizing it, and reading off the type (with expansion variables) from the interaction net of the normal form, *without* any constraint solving. (Note that termination follows from the normalization bounds.) Dually, we also show that from a principal System- \mathbb{I} type including expansion variables, we can reconstruct the normal form.

Flow analysis, semantics, intersection types: System- \mathbb{I} facilitates *flow analysis*: does some fixed call site in a program ever call some other procedure? Formulating System- \mathbb{I} programs as interaction nets, flow analysis for intersection-typed programs is essentially a reformulation of Girard’s *geometry of interaction* (GoI), made mundane for computer scientists in the guise of *context semantics* [4,3,11]. And from the context semantics of a term, we may recover its principal System- \mathbb{I} typing with expansion variables. The method is an elaboration of a *readback algorithm* we have used to prove the correctness of optimal reduction [10,3,1,11], 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.

2 Preliminaries

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

We motivate System- \mathbb{I} by first considering a vanilla intersection type system in the style of [14], with types $\tau, \sigma, \rho \in \mathcal{T}$ and $\bar{\tau} \in \bar{\mathcal{T}}$ given by the grammar

$$\bar{\tau} ::= \alpha \mid \tau \rightarrow \bar{\tau} \quad \tau ::= \bar{\tau} \mid \tau \wedge \tau$$

where $\alpha, \beta, \gamma, \dots$ are type variables, and inference rules that introduce \wedge when a variable occurs multiple times:

$$\frac{}{x : \bar{\tau} \vdash x : \bar{\tau}} \text{Var}$$

$$\frac{\Gamma, x : \tau \vdash P : \bar{\tau}}{\Gamma \vdash \lambda^I x.P : \tau \rightarrow \bar{\tau}} \lambda I \quad \frac{\Gamma \vdash P : \bar{\tau}'}{\Gamma \vdash \lambda^K x.P : \bar{\tau} \rightarrow \bar{\tau}'} \lambda K$$

$$\frac{\Gamma \vdash P : \tau \quad \Delta \vdash P : \tau'}{\Gamma \wedge \Delta \vdash P : \tau \wedge \tau'} \wedge \quad \frac{\Gamma \vdash P : \tau \rightarrow \bar{\tau} \quad \Delta \vdash Q : \tau}{\Gamma \wedge \Delta \vdash P Q : \bar{\tau}} \text{App}$$

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

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

Without the \wedge -rule, the inference rules are syntax-directed. Duplication of term P in the premises of \wedge highlights a verbose feature of the type system.

System- \mathbb{I} attempts to restrain this verbosity while retaining *principality* [9]. Avoiding the redundancy of rule \wedge , we infer a *principal typing* once, and obtain other typings by a kind of substitution called *expansion*. Unfortunately, we show here that type inference is inexorably equivalent to normalization: the duplication of boxes negates the “efficient” sharing of type inference.

In System- \mathbb{I} , we introduce *expansion variables* F, G ranging over *expansions* $e \in \mathcal{E}$, and extend the definition of types to include expansion variables:

$$e \in \mathcal{E} ::= \square \mid e \wedge e \mid F e \qquad \tau \in \mathcal{T} ::= \bar{\tau} \mid \tau \wedge \tau \mid F \tau$$

Intuitively, expansions are binary trees (\wedge serving as an internal node) that can be substituted for expansion variables in a type. In such a substitution, applications $F \tau'$ in τ are replaced by e with each of the holes filled with a fresh instance of τ' .

The \wedge -rule types each parameter occurrence in a λ -abstraction. Instead, we use rule E below to type the parameter *once*, and then use expansions to specialize each occurrence. Alternatively, we could use the rule AppAlt for application:

$$\frac{\Gamma \vdash P : \alpha}{F \Gamma \vdash F P : F \alpha} \text{E} \qquad \frac{\Gamma \vdash P : F \tau \rightarrow \bar{\tau} \quad \Delta \vdash Q : \tau}{\Gamma \wedge F \Delta \vdash P Q : \bar{\tau}} \text{AppAlt}$$

Note $F \Gamma = \{x : F\alpha \mid x : \alpha \in \Gamma\}$. The latter choice commits type derivation to an analogue of the linear logic translation $[a \rightarrow b] = ![a] \multimap [b]$ where the F plays the role of the $!$.

Intersection type systems (and System- \mathbb{I}) type exactly the strongly normalizing λ -terms, so type inference is undecidable. We thus use only a tractable fragment by limiting the *alternation* of \wedge on the “domain” (i.e., argument) side of a \rightarrow in types, using a definition which differs critically from that in [9]. The *rank* of a type is defined as:

$$\begin{aligned} \text{rank } a &= 0 & \text{rank}(F \tau) &= \text{rank } \tau \\ \text{rank}(\bar{\tau} \rightarrow \bar{\tau}') &= \max\{\text{rank } \bar{\tau}, \text{rank } \bar{\tau}'\} & \text{rank}(F \tau \rightarrow \bar{\tau}') &= \max\{\text{rank } \tau, \text{rank } \bar{\tau}'\} \\ \text{rank}(\tau \wedge \tau') &= \max\{\text{rank } \tau, \text{rank } \tau'\} \\ \text{rank}((\tau \wedge \tau') \rightarrow \bar{\tau}) &= \max\{1 + \text{rank } \tau, 1 + \text{rank } \tau', \text{rank } \bar{\tau}\} \end{aligned}$$

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 interaction nets (here, we have used the *call-by-name* encoding, 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 interaction nets are either *free ports* (\circ) representing the free variables, or the distinguished *root port* for the whole term. *Weakening nodes* (\otimes) 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* (\frown) mark variable occurrences. A *global* construction, the *box*, delineates the (sharable) argument of an application. Edges are called *wires*; their endpoints are attached to *ports* of either a node, or the entire graph. Each node has one principal port (marked with a black dot), and possibly other *auxiliary ports*.

An interaction 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. We will see that these rules are the graphical equivalent of the constraint solving rules in [9]. As a consequence, β -unification is just the reinvention of global reduction, though a type inference algorithm requires the additional ingredient of strong normalization.

The encoding is call-by-name since when an application is reduced, the argument is duplicated until each variable occurrence (marked by a croissant) has a copy. 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 NF_{\Rightarrow} for its normal forms; it simulates β -reduction up to trivial permutations on the sharing nodes, as in the following proposition.

Proposition 1. *Let M and N be λ -terms such that $M \rightarrow_{\beta} N$, and let $\lceil - \rceil$ denote the coding of λ -terms as interaction net. Then there is a net I such that $\lceil M \rceil \Rightarrow I$, and I is equivalent to $\lceil N \rceil$ up to left/right orientation of auxiliary ports on sharing nodes, and their location inside or outside boxes. \square*

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 interaction 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 interaction net a *proofnet*. For example, with simple types, the wire on the principal port of an @-node has an incoming type $\alpha \rightarrow \beta$, and the right (left) auxiliary port has an incoming (outgoing) type α (β). Observe that each of the typing rules Var, λ I, λ K, and AppAlt correspond naively 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.³ But then the sharing nodes are *unpairing* nodes

³ Note that the type environment of System-II contains exactly the free variables of the term—consequently the two subderivations have the same variables in the type environment.

(with the linear logic equivalent of \wp), 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-I to represent the different typings. With these considerations in mind, we define the intersection typing of a proofnet *à la* System-II:

Definition 1. *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, $S(e\tau)$, give the different typings of the wire; we therefore write the annotations as $(S(e\tau))$.*
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. Then every switching results in a forest of connected, acyclic graphs containing either the root port, or the port marking a weakening from a K -redex. (This is essentially the Danos-Regnier criterion [2].) \square*

If Π without annotations and orientations of the wires is the interaction net I , we call Π a typing of I . We say that I has typing $\langle x_1 : \tau_1, \dots, x_n : \tau_n, \tau \rangle$ 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 1. *Let M be a λ -term and $I = \lceil M \rceil$ its encoding as an interaction net. Then M has System-II-typing $\langle \Gamma, \tau \rangle$ if, and only if, I has System-II-typing $\langle \Gamma, \tau \rangle$. \square*

Every System-II typable term has a *principal typing* [9,8]; a similar assertion holds for the analogous interaction net:

Definition 2. *Let Π be a proofnet typing an interaction 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 $S(e\tau)$ with $(S' \circ S)((E' e)\tau)$. \square*

3 Normalization bounds, type inference bounds

Since type inference for System-II is realized by normalization, we want to derive upper bounds on normalization. Otherwise, it is not clear whether inference by

normalization terminates. Let $\mathbf{K}(0, n) = n$ and $\mathbf{K}(t + 1, n) = 2^{\mathbf{K}(t, n)}$; given a λ -term of rank t and length n , we prove that its normal form has length $O(\mathbf{K}(t, n))$. Also, given a λ -term of length n , deciding if it is typable in rank t requires time $\Omega(\mathbf{K}(t, n))$. The function $\mathbf{K}(t, n)$ appears in both bounds because of a simple observation: without ACI, simply-typed terms are *linear*—a variable can occur at most once. For simplicity, we carry out this analysis without expansion variables.

Define a *goofy redex* to be $(\lambda x_1. \dots \lambda x_k. P)^{\alpha_1 \rightarrow \dots \rightarrow \alpha_k \rightarrow \beta} Q_1 \dots Q_k$, with $\alpha_k \neq \gamma \wedge \delta$, which becomes $((\lambda x_1. \dots \lambda x_{k-1}. P)[Q_k/x_k])^{\alpha_1 \rightarrow \dots \rightarrow \alpha_{k-1} \rightarrow \beta} Q_1 \dots Q_{k-1}$ by goofy reduction. We make the casual but crucial observation that in a goofy redex, the bound variable x_k can occur at most *once* in P —were it to occur twice, x_k would have a \wedge -type. So a goofy normalization is *inherently linear* (thus its name): it is guaranteed to terminate *without* increasing term size, nor does it change the normal form. To bound overall normalization, we follow a fairly standard calculation that pairs *goofy developments* (maximal sequences of goofy reductions) with *complete developments*; after a goofy development, a complete development must reduce the 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 xP) are not underlined. More formally, define $\mathcal{D}(x) = x$, $\mathcal{D}(\lambda x. P) = \lambda x. \mathcal{D}(P)$, $\mathcal{D}(x P_1 \dots P_n) = x \mathcal{D}(P_1) \dots \mathcal{D}(P_n)$, and the *interesting case* (where n is chosen maximally)

$$\mathcal{D}((\lambda x_1. \dots \lambda x_n. P) Q_1 \dots Q_n) = \mathcal{D}(P)[\mathcal{D}(Q_1)/x_1, \dots, \mathcal{D}(Q_n)/x_n]$$

Lemma 1. *Assume λ -term T is in goofy normal form, and let $\iota(-)$ give the maximal rank of any redex in a term. Then $\iota(\mathcal{D}(T)) < \iota(T)$ \square*

The proof is by induction on T , and the only interesting case is the last one above for complete developments: if the function has type $\sigma = \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta$, the reduction $(\lambda x_1. \dots \lambda x_n. P)^\sigma Q_1 \dots Q_n \triangleright P[Q_1/x_1, \dots, Q_n/x_n]$ can *only* create redexes of index $\alpha_1, \dots, \alpha_n$ but *not* β , and since each $\alpha_i = \beta_i \wedge \beta'_i$ we know rank $\alpha_i < \text{rank } \sigma$.

Theorem 2. $|\mathcal{D}(T)| \leq 2^{|T|}$; thus if T is typable in rank t , its normal form has length $O(\mathbf{K}(t, |P|))$. \square

The proof is by induction on t and repeated induction on $\mathcal{D}^i(T)$, $0 \leq i \leq t$; in the only *interesting case* (see above), bound $|\mathcal{D}(P)|$ and $|\mathcal{D}(Q_i)|$ as $2^{|P|}$ and $2^{|Q_i|}$; then the derived term has length at most $2^{|T|}$.

This theorem shows how to construct untypable terms: just violate the normalization bounds. Let \bar{n} be the Church numeral for n ; then $\bar{2}\bar{2}\bar{2}$ is not typable in rank 3. Note $P = \lambda x. x\bar{2}$ computes $f(n) = 2^n$ on Church numerals: observe that for any rank, $\lambda k. k P \bar{2}$ cannot be typed for almost all \bar{k} . Terms untypable in ML, System F, etc. are often anomalous and peculiar; those untypable in 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.

To derive a lower bound on the complexity of type inference, observe that the standard codings of λ -terms for Boolean values and functions are all linear terms. Thus Boolean circuits can be given a rank 0 type, as well as the transition function of a Turing Machine (which combines circuitry with some list processing)—as long as the latter is only used on *one* configuration. To iterate the transition function, we use the following:

Lemma 2 (Polymorphic Iteration). *Let $N = \mathbf{K}(t, n)$, \bar{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 $\tau(i) \rightarrow \tau(i+1)$ for $0 \leq i < N$, where the $\tau(i)$ are arbitrary. Then the term $(\lambda z.\bar{2}^n z)\bar{2} \cdots \bar{2} \Phi (t-1 \bar{2}s)$ normalizes to $\lambda x.\Phi^N x$, and has simple type $\tau(0) \rightarrow \tau(N)$ in rank t . \square*

A version of this lemma was originally proved in [7]; it can also be proved with no use of ACI. One of the applications of this lemma *with* ACI was to let Φ be the term $\lambda x.x\bar{2}$, 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 (see, i.e., [6,5]) we may conclude:

Theorem 3. *Let P be a λ -term of length n ; then deciding if P is typable in rank t is complete for $\text{DTIME}[\mathbf{K}(t, n)]$. \square*

Corollary 1. *Let P be an untyped λ -term. Compute t successive iterations of goofy and complete developments on P , where in the case of a K -redex, we develop discarded arguments as well, producing a set of terms. Then P is typable in rank t iff normalization of these terms never creates a redex with a nonlinear function. \square*

4 Typing is never cheaper than normalization

4.1 Naive type inference via normalization

Given a fixed rank, we can construct a naive algorithm for type inference, similar to many for other type systems, as follows: for an untyped term P , construct a net for P , annotating each edge with a unique type variable. Begin normalization, generating new type variables and constraints; solving the constraints generates the type. By checking the rank of incremental solutions, we can ensure that the algorithm terminates.

The initial constraints are just those around λ - and $@$ -nodes, and around sharing nodes: (1) that a function must have type $\alpha \rightarrow \beta$, an argument type α , the result type β ; (2) that sharing a datum of type t , with uses at type u, v , incurs the constraint $t = u \wedge v$. Using the call-by-name coding, every argument of an application is in a *box*; each variable occurrence is marked by a *croissant*. Constraints are modified when (1) a croissant of type t opens a box with type u (add $t = u$), and (2) a box is duplicated: make two copies of the box, replacing

each type variable u within by u' in one copy, and u'' in the other, and add the constraint $u = u' \wedge u''$.

We observe that the technology of *expansion variables* in [9] implicitly associates a variable F with each box. A type $\alpha \rightarrow \beta$ occurring in the box is then written $F(\alpha \rightarrow \beta)$, and various formal rules are introduced to implement the duplication we have described in graphical shorthand. Solving the constraints that are introduced by reduction is easy: $=$ is an equivalence relation, and from either $a \rightarrow b = c \rightarrow d$ or $a \wedge b = c \wedge d$ we derive $a = c$ and $b = d$. Observe that the latter derivation only makes sense because \wedge is not commutative.

Lemma 3. (1) *The constraints generated are finite for strongly normalizing terms, and every such finite set has a most general (principal) solution.* (2) *Subject reduction holds, modulo the existence of K -redexes. (In this latter case, we may continue to normalize discarded arguments.)* \square

In the next section, we detail the truth of (2): that the solution of syntax-generated typing constraints is fundamentally invariant, and preserved by reduction. Observe that upper bounds on the length of normal forms of rank t λ -terms then serve as upper bounds on type inference. Furthermore, the combination of normalization and constraint solving is redundant—let’s get rid of the latter.

4.2 Type inference without solving constraints

We have now reached the high tide of the paper: we 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.

We obtain the result through an isomorphism between the normal form of an interaction net, and its principal typing.⁴ 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. 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. Since the isomorphism holds for normal forms, normalization and type inference are simply two sides of the same coin.

Principal Typing From a Normal Form In obtaining the principal typing from the normal form, we read sharing nodes as \wedge , function and application nodes as \rightarrow , and boxes as expansion variables. We collect the principal typing

⁴ Strictly speaking there is a plethora of principal typings. The difference between them is however only in the choice of names for the type and expansion variables.

using the recursive algorithm outlined in Fig. 4; it is called **typing** : $\mathcal{I} \leftrightarrow \mathcal{T}$. The base case corresponds to the normal form $\lambda x_1. \dots \lambda x_k. v$ with the typing $(\emptyset, \gamma_1 \rightarrow \dots \rightarrow \gamma_k \rightarrow \gamma_i)$ when $v = x_i$ and $(v : \alpha, \gamma_1 \rightarrow \dots \rightarrow \gamma_k \rightarrow \alpha)$ 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 $\gamma_1 \rightarrow \dots \rightarrow \gamma_k \rightarrow \alpha$ and the free port α .

Otherwise the graph represents the term $\lambda x_1. \dots \lambda x_k. v N_1 \dots N_\ell$. We apply the algorithm recursively to each of the subnets II_i (corresponding to N_i); the result will be type annotations of the root and the free ports of each subnet. We assign a fresh expansion variable F_i to each of the boxes around II_i . We find the intersection type of each shared variable by processing the sharing forest bottom-up. The leaves are the wires coming from II_i and the croissant corresponding to the head variable. We initialize by giving the head variable type $F_l \tau_0^l \rightarrow \dots \rightarrow F_1 \tau_0^1 \rightarrow \alpha$, and each wire from II_i type $F_i \pi_j^i$, where π_j^i is the type returned by the recursive call. For each sharing node we give the principal port type $\tau_\circ \wedge \tau_\bullet$ where τ_\circ (τ_\bullet) is the type of the white (black) auxiliary port.

Computing a normal form from a principal typing Dually, the function $\text{net} : \mathcal{T}^* \times \mathcal{T} \rightarrow \text{NF}_{\Rightarrow}$ produces an interaction net normal form of any given principal System-II typing.

In its onset **net** uses the intuition all functional programmers have about gazing the structure of a function from its type: she immediately knows that a function of type $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$ has two outermost abstractions and that the first argument appears in the function position of an application. What sets System-II apart from other functional languages is the lack of ACI: without idempotency we have a type specification for each variable occurrence; lack of commutativity gives the orientation of sharing nodes, non-associativity tells us the boxes, and, finally, the principal typing reveals the sources of the arguments of applications. For instance, $\bar{2}$ can be typed $(\alpha \rightarrow \alpha) \wedge (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$; it is not clear whether it is the first or the second occurrence of the first argument that is applied to the second argument. The principal typing, $(F \beta \rightarrow \gamma) \wedge F (G \alpha \rightarrow \beta) \rightarrow F G \alpha \rightarrow \gamma$, makes it clear that it is the second occurrence. This insight is captured in a well-known characterization of the form of a principal intersection types:

Lemma 4. *In a principal System-II typing, every type variable occurs at most twice; if it occurs twice it occurs once positively and once negatively.⁵ Each occurrence of a type variable has the same expansion variables as one goes from the root of the type's syntax tree to the occurrence. Every expansion variable occurs at most once positively.* \square

⁵ 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.

Definition 3. The function $\text{net} : \mathcal{T}^* \times \mathcal{T} \hookrightarrow \text{NF}_{\Rightarrow}$ is defined on any typing meeting the conditions in Lemma 4. Let τ be a type and Γ a type environment; we proceed as follows:

1. Let net^+ and net^- be the mutual recursive functions in Fig. 5. Produce a forest of graphs by applying net^+ to τ and net^- to each type in Γ . For example, the computation of $\text{net}^+(\tau \rightarrow \tau')$ constructs proofnets from $\text{net}^+(\tau')$ and $\text{net}^-(\tau)$, connecting them (as in Fig. 5) with a λ -node.
2. Connect the ports of each type variable that is mentioned twice. Connect the remaining ports to a weakening node. \square

Proposition 2. Let I be a normal form with principal typing $\langle \Gamma, \tau \rangle$. We then have $I = \text{net}(\langle \Gamma, \tau \rangle)$ and $\text{typing}(I) = \langle \Gamma, \tau \rangle$. \square

Typings Are Preserved Under Reduction In proceeding from the sandbox of normal forms to arbitrary interaction nets, we establish that the set of typings is invariant under reduction. We have thus not only subject reduction, but also subject expansion.⁶

We achieve this goal in two steps: First, we recast the question of typability as a constraint problem. We show that every interaction net has a set of *the 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 have the same principal typing. First the notion of typing constraints:

Definition 4. Let $I \in \mathcal{I}$ be an interaction 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 set of typing constraints \mathbf{C}_I of I is a set of equations $\tau = \sigma$, where some occurrences of type variable α can be labelled $\bar{\alpha}$ constraining α to types from $\bar{\mathcal{T}}$. The nodes induce the typing constraint to the left of Fig. 6. For a box labelled F , let \mathbf{C} be union of the typing constraints for the nodes and boxes inside the box; the typing constraints for F are $\{\alpha = F \bar{\alpha}', \beta_1 = F \beta'_1, \dots, \beta_k = F \beta'_k\} \cup \{F \tau = F \sigma \mid \tau = \sigma \in \mathbf{C}\}$ 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. The constraints \mathbf{C}_I is the union of the constraints from the outermost boxes and the nodes outside boxes.
2. A solution \mathbf{U} to a set of typing constraints \mathbf{C} is an expansion substitution \mathbf{E} and a variable substitution \mathbf{T} such that for all equations $\tau = \sigma \in \mathbf{C}$ we have $\mathbf{T}(\mathbf{E} \tau) = \mathbf{T}(\mathbf{E} \sigma)$. The substitutions are restricted to the variables occurring in \mathbf{C} , and \mathbf{T} respects labels, i.e., type variables $\bar{\alpha}$ which have a labelled occurrence in \mathbf{C} are only substituted types from $\bar{\mathcal{T}}$. \square

⁶ Recall that subject expansion is the feature that M has the same typings as its reducts, i.e., if M reduces to N then $N : \langle \Gamma, \tau \rangle$ implies $M : \langle \Gamma, \tau \rangle$.

Given a fixed net, to prove an isomorphism between typings and solutions to the typing constraints, we need to relate proofnets and solutions to the typing constraints. Both specify an intersection type for each wire—the proofnet does so through the annotation $S(e\tau)$ of each wire. A solution $U = (E, T)$ to the constraints inserts expansions from the substitution E for the expansion variables G , 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 $E \circ G$ to γ). We consider the proofnet Π and the solution U *equivalent on a wire* exactly when $S(e\tau) = T(E(G\gamma))$.

Proposition 3. *Let I be an interaction 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. \square*

It is now sufficient to prove that the set of solutions are preserved under reduction. Intuitively, solutions are preserved because the constraints 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. As there are no surrounding boxes, the constraints are $\bar{\alpha} = \bar{\beta}$ and $\beta = F\bar{\gamma}$. Any solution (T, E) will have $T(E\alpha) = T(E(F\bar{\gamma}))$; as α , β , and γ are type variables the expansion substitution E has no effect and we have: $T\alpha = T\beta$ and $T\beta = T(F\bar{\gamma})$. To respect the labellings, T must substitute a \bar{T} -type for β so we have $F = \square$. Consequently, all solutions have $T\alpha = T\beta = T\gamma$. 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 $T(E(G\bar{\alpha})) = T(E(G\bar{\beta}))$ and $T(E(G\beta)) = T(E(G(F\bar{\gamma}))$) where 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 reduct have a different number of wires. This is solved by noting that there is redundancy in a solution, e.g., in the example above knowing 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 to a unique solution. Call a *basis* any set of variables that is sufficiently large so that any solution specified on these variables extends uniquely to a solution for the entire set of variables occurring in the typing constraints.

Definition 5.

1. A basis B of a set of typing constraints C is a subset of the variables occurring in C that satisfies the following condition: 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 $R(x) = T(x)$ for any $x \in \text{dom } R$ and $D(F) = E(F)$ for any $F \in \text{dom } D$.

2. Given a basis B and solution $U = (S, E)$ to a set of typing constraints C , we write $U(B)$ 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 specified on B . We write $[(R, D)]_C$ for the unique solution to C if it exists. \square

We can now prove that the set of solutions to the typing constraints is invariant under reduction. For each reduction rule, we establish that we can choose two bases, one for the net and one for the reduced net, with an immediate connection between the wires and the boxes in the two bases. A solution for the original net can then be transformed to solution for the reduced net (and vice versa): restrict the solution to the basis of the original net, map it to the basis of the reduced net, and extend it uniquely

Proposition 4. *Let I and J be interaction 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) $[U_I(B_I)]_{C_J}$ exists and is a solution to C_J when U_I is a solution to C_I and 2) $[U_J(B_J)]_{C_I}$ exists and is a solution to C_I for any solution U_J of C_J . \square*

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:

Corollary 2. *The set of typings is invariant under proofnet reduction. \square*

Another easy corollary follows from Lemma 2 in combination with the strong normalization result in Sec. 3.

Corollary 3. *If I is an interaction net with principal typing $\langle \Gamma, \tau \rangle$ and normal form J , then $\text{net}(\tau) = J$, and $\text{typing}(J) = \langle \Gamma, \tau \rangle$. \square*

5 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. 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 $\text{Int} \rightarrow \alpha$, what is α ?

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. And

not too surprisingly, from the context semantics of a term, we can recover its System- \mathbb{I} typing. The 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.

Acknowledgements: Thanks to 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, who gave a very inspiring seminar on β -unification and proofnets. For their technical comments and criticisms, we also want to thank Alan Bawden, Joe Hallett, Franklyn Turbak, Laurent Regnier, and Pawel Urzyczyn.

References

1. A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998. Referenced on pp. [4](#)
2. V. Danos and L. Regnier. The structure of multiplicatives. *Arch. Math. Logic*, 26, 1989. Referenced on pp. [7](#)
3. 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. Referenced on pp. [3](#), [4](#)
4. 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. Referenced on pp. [4](#)
5. 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. Referenced on pp. [9](#)
6. 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. Referenced on pp. [9](#)
7. 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. Referenced on pp. [2](#), [3](#), [9](#)
8. 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. Referenced on pp. [7](#)
9. 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. Referenced on pp. [5](#), [6](#), [7](#), [10](#)
10. J. Lamping. An algorithm for optimal lambda-calculus reductions. In *Conf. Rec. 17th Ann. ACM Symp. Princ. of Prog. Langs.*, pages 16–30, 1990. Referenced on pp. [4](#)
11. 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. Referenced on pp. [4](#)
12. L. Regnier. *Lambda calcul et réseaux*. PhD thesis, University Paris 7, 1992. Referenced on pp. [3](#)

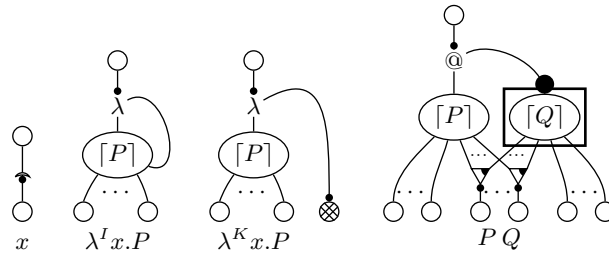


Fig. 1. The inductive encoding $[\cdot]$ of a λ -term as an interaction net. The port (\circ) on top is the root, representing the whole term, while the ports in the bottom correspond to the free variables. In the application case the left group of wires is the variables solely in P , the middle group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

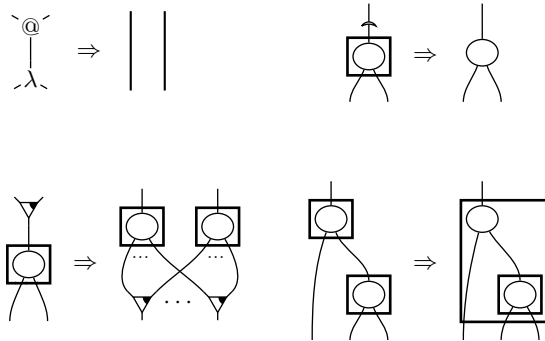


Fig. 2. Reduction rules for interaction nets.

13. R. Statman. The typed lambda-calculus is not elementary recursive. *Theoret. Comput. Sci.*, 9(1):73–81, July 1979. Referenced on pp. 3
14. S. J. van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Catholic University of Nijmegen, 1993. Referenced on pp. 4
15. 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. Referenced on pp. 1, 2

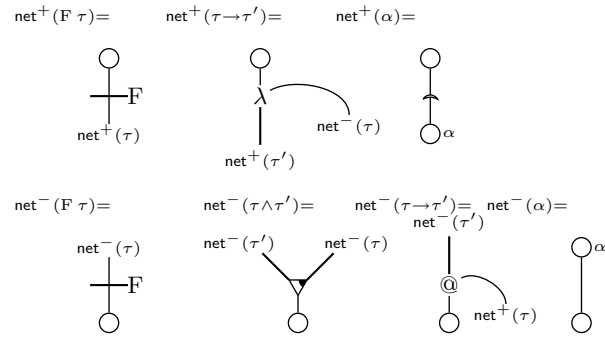


Fig. 5. The mutual recursive definitions of the functions net^+ and net^- building the skeleton of an interaction net from a type.

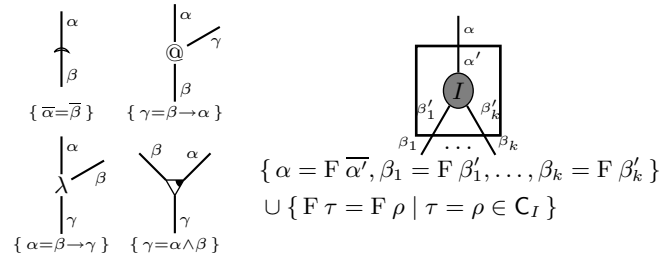


Fig. 6. The typing constraints for the various interaction nodes. In the right subfigure, F is the expansion variable of the box.