# Polymorphic type inference

*Daniel Leivant*
Computer Science Department
Carnegie-Mellon University

## Introduction.

The benefits of strong typing to disciplined programming, to compile-time error detection and to program verification are well known. Strong typing is especially natural for functional (applicative) languages, in which function application is the central construct, and type matching is therefore a principal program correctness check. In practice, however, assigning a type to each and every expression in a functional program can be prohibitively cumbersome. As expressions are compounded, the task of assigning a type to each expression and subexpression becomes practically impossible, even more so because the type-expressions themselves grow longer. It becomes imperative therefore to design friendly programming environments that permit the user type-free programming, but that generate fully typed programs in which the types of all expressions are inferred by the system from the program. For interactive functional programming environments of the kind implemented for the Edinburgh functional programming language ML, a type-inference system is an invaluable tool for on-line parse-time error detection and debugging.

The issue of type inference leads naturally to *type-schemes.* If all one has of a functional program is a definition statement such as $f(x)=3$, then all one can say of the type of $f$ is that it must be of the form $\tau\rightarrow\mathbf{int}$, i.e. — an instance of the type scheme $t\rightarrow\mathbf{int}$, where $t$ is a type variable. As more of the program appears, say $x(\mathrm{true})=z$, $t$ may be restricted, to $\mathbf{bool}\rightarrow s$. We are therefore interested in *type-schemes*, using *type parameters* (free type variables). All one needs then to obtain a rudimentary form of generic procedures is a device for type instantiation. This function is fulfilled in ML by the construct **let**. Other methods are described in §4 below.

During the process of inferring a type for a given program, one would like at each step to find a *most general* (so-called *principal*) type scheme for each subexpression, so that types to be assigned to variables and subexpressions in the sequel may be assumed to be instances of those assigned earlier. The notion of principal type was introduced by Curry and Feys [CF] for Combinatory Logic, and Hindley [Hin] showed that the principal type exists for any Combinatory Logic expression and that it can be found using J.A. Robinson's Unification algorithm [Rob]. The seminal paper on type-inference for functional programs is Milner's [Mil], where an algorithm W is defined for inferring a type for any ML program (without recursively defined types). More recently, Damas and Milner [DM] defined a simple formal calculus of type inference, with respect to which W is proved complete.

In this paper we study several aspects of type inference for polymorphic type disciplines. First, we deal with type inference for parametric type systems, that is — type systems that include simple types defied over constant types and type variables. We recast Milner's algorithm W in a general "algebraic" form, which applies to a variety of generic type disciplines (§ 2). In §3 we present an alternative algorithm V for type inference, fundamentally different from W, which easily accommodates type coercion and overloading, allows efficient local updates to the user program, and is more efficient than W for implementations that allow concurrency.

In §4 we briefly describe four polymorphic disciplines: type abstraction, type quantification, type conjunction (intersection) and the ML construct **let**.

In §5 type inference for the abstraction and quantification disciplines is discussed. We point out that the two disciplines are combinatorially isomorphic, and we outline a type inference algorithm for them.

§6 is devoted to type inference for the conjunctive discipline. We mention some limitations to type inference here, in particular — the fact that the set of typable expressions is not effectively decidable. Moreover, there is no feasible algorithm that would type expressions even when these are given with a quantificational typing. This kind of non-effectiveness motivates our considering a restriction to types where type conjunction is permitted only at low levels of functionality. Here the level of functionality of a function (or procedure) is recursively defined as one plus the greatest level of functionality of its arguments. This gives rise to a hierarchy on polymorphic types, a notion that is equally natural for the type abstraction and type quantification disciplines. We show that our algorithm V (of §3) can be naturally extended to the restriction of the conjunctive discipline to types of rank 2 in the hierarchy, and we outline an argument showing that already for a low rank, seemingly 3, type inference is not effectively decidable.

In §7 we discuss the ML construct let in light of our previous results. We point out that the polymorphic discipline of ML is isomorphic to the restriction to rank 2 of each one of the other disciplines. This gives weight to the particular choice of polymorphism in ML, and to the completeness of Milner's algorithm W for that choice. At the same time, the equivalence suggests more flexible representations of this discipline (namely — either the conjunctive or the quantificational disciplines restricted to rank 2) in which the anomaly in ML, of legal expressions with well-formed illegal subexpressions, is avoided.

## 1. Parametric type inference.

Type inference is a purely combinatorial problem, and in order to study it we assume that expressions other than constants are given to us type-free (at least to a point), and are then assigned types "from the exterior", although the intent is that expressions are thought of as well-typed objects from the outset. Contrary to the construction of usual term-algebras over a set of functions, we have to deal here with constructs that involve binding of variables. Lambda abstraction is an obvious example, but there are others, sometime binding more than a single variable. In Martin-Lof's Programming Language [Mar] the constructs $E$, $D$ and $R$ bind two variables simultaneously, and the construct $T$ binds three variables. The fixed-point operator $\mu$ is another variable binding construct. Each functional letter $\psi$ is therefore given with an arity $r_\psi = (p_\psi,q_\psi)$, where $p_\psi$ is the number of variables being bound by $\psi$, and $q_\psi$ is the number of arguments.

A *term system* is determined by a pair $\langle \mathrm{C},\Psi\rangle$, where $\mathrm{C} = \{C_a\}_{a\in A}$, with $A$ a finite set of *atomic types*, and $C_a$ the set of *constants of type* $a$, and where $\Psi$ is a finite set of *functional letters*, with each $\psi \in \Psi$ assigned an *arity* $r_\psi = (p_\psi,q_\psi)$. Let $c$ range over $\mathrm{C} = \bigcup_{a\in A} C_a$, $\psi$ range over $\Psi$, and $x$ range over a countable list X of variables; the *expressions* $e \in \mathrm{E}$ of the system are then given by

$e ::= c \mid x \mid \psi[x_1,...,x_{p_\psi}](e_1,...,e_{q_\psi})$, where $e_i \in \mathrm{E}$.

Let $K$ be a set of *type constructors*. Assume for now that no type constructor binds variables. Then the set $\mathrm{T}$ of *type-expressions* is generated inductively as usual from a set of type variables, the set $A$ of type constants, and $K$.

A *type statement* is an expression $e{:}\tau$ where $e \in \mathrm{E}$, $\tau \in \mathrm{T}$ (read: $e$ is of type $\tau$). If $\bar{e} = (e_1,...,e_k)$ and $\bar{\tau} = (\tau_1,...,\tau_k)$ then $\bar{e}{:}\bar{\tau}$ stands for $\{e_i{:}\tau_i\}_i$. A type assigned to an expression depends on the type of the free variables therein. This leads to the following definition. A *basis* $B$ is a set $\bar{x}{:}\bar{\tau}$ where $\bar{x} \in \mathrm{X}^k$, $\bar{\tau} \in \mathrm{T}^k$, and $\bar{x}$ is without repetitions. Write $B,e{:}\tau$ for $B \cup \{e{:}\tau\}$. A *typing* is a pair $B \vdash e{:}\tau$ of a basis and a type statement. We are interested only in typings where $B$ assigns a type to each variable free in $e$. If this is the case, then we say that $\langle B;\tau\rangle$ is a *typing for* $e$. Also, type statements in $B$ for variables not free in $e$ play no role. If no such variable occurs in $B = \bar{x}{:}\bar{\sigma}$, we contract $\langle B;\tau\rangle$ to $T = \langle\bar{\sigma};\tau\rangle$, where the order in $\bar{\sigma}$ is the order in X of the corresponding variables. $T$ is also said to be a *typing for* $e$. $B$ and $\sigma$ are said to be *antecedent* of the typing, and $\tau$ the *succedent*.

A *substitution* is an operation S of simultaneously replacing free occurrences of type variables by type expressions. If $\bar{\tau} = (\tau_1,...,\tau_k)$ then $\mathrm{S}\bar{\tau} = (\mathrm{S}\tau_1,...,\mathrm{S}\tau_k)$. SR denotes the composition of substitutions S and R. If $\bar{\mathrm{R}}$ is $(\mathrm{R}_1,...,\mathrm{R}_n)$, then $\mathrm{S}\bar{\mathrm{R}} = (\mathrm{SR}_1,...,\mathrm{SR}_n)$.

A *type inference calculus assigns to each* $\psi \in \Psi$ *an inference rule* $R_\psi \equiv \langle\bar{\sigma}_\psi;\bar{\rho}_\psi;\tau_\psi\rangle$, *where* $\bar{\sigma}_\psi \in \mathrm{T}^{p_\psi}$, $\bar{\rho}_\psi \in \mathrm{T}^{q_\psi}$, $\tau_\psi \in \mathrm{T}$. A set-up of the form

$$\frac{\langle\, B,\ \bar{x}{:}\mathrm{S}\bar{\sigma}_\psi \vdash e_i{:}\mathrm{S}\rho_i \,\rangle_{i \le q_\psi}}{B \vdash \psi[\bar{x}](\bar{e}){:}\mathrm{S}\tau_\psi}\ ,$$

where $\langle\rho_i\rangle_i = \bar{\rho}_\psi$ and S is a substitution, is an *instance* of $R_\psi$. For example, the inference rule for $\lambda$-abstraction is $\langle t;s;t\to s\rangle$, and each instance thereof is of the form

$$\frac{B,x{:}\tau \vdash e{:}\sigma}{B \vdash \lambda x.e{:}\tau\to\sigma}\ .$$

A *derivation* of $S$ is a tree D of statements, where each leaf is either $B \vdash c{:}a$ with $c \in C_a$, or $B \vdash x{:}\sigma$ with $(x{:}\sigma) \in B$, and where each furcation node is an instance of an inference rule. We also write $B \vdash e{:}\tau$ when this is the root of some derivation D, in which case D *derives* $B \vdash e{:}\tau$.

A typing $T$ is a *principal typing for* $e$ if the typings for $e$ are precisely those of the form $\mathrm{S}T$ for some substitution S. Note that if S is the empty substitution then $\mathrm{S}T$ is $T$.

It will be convenient to generalize the notions above to permit the assignment of several distinct types to one and the same variable. In a discipline in which a variable can have only one type, the intent is that the real type of a variable $x$ is an instance of each one of the type schemes assigned to $x$. For a set $A$, let $\tilde{A}$ denote the collection of finite subsets of $A$. A set of statements $x{:}\Sigma = \{x{:}\sigma \mid \sigma \in \Sigma\}$, where $\Sigma \in \tilde{\mathrm{T}}$, is a *multi-type statement.* For a substitution S, let $\mathrm{S}\Sigma \equiv \{\mathrm{S}\sigma \mid \sigma \in \Sigma\}$. If $\bar{e} = \langle e_1,...,e_k\rangle$, $\bar{\Sigma} = \langle\Sigma_1,\ldots,\Sigma_k\rangle$, $\Sigma_i \in \tilde{\mathrm{T}}$, let $\bar{e}{:}\bar{\Sigma} = \{e_i{:}\Sigma_i\}_i$. A *multi-basis* $B$ is a set $\bar{x}{:}\bar{\Sigma}$ where $\bar{x} \in \mathrm{X}^k$, $\bar{\Sigma} \in \tilde{\mathrm{T}}^k$ for some k, and $\bar{x}$ is without repetitions. If each $\Sigma \in \bar{\Sigma}$ is a singleton, then $B$ is isomorphic to a basis. A *multi-typing* is a pair $B \mid\!- e{:}\tau$ of a multi-basis and a type statement. Again, this can be represented by a tuple $\langle\bar{\Sigma};\tau\rangle$. A multi typing $\langle\Sigma_1',\ldots,\Sigma_k';\tau\rangle$ is a *thinning* of $\langle\Sigma_1,\ldots,\Sigma_k;\tau\rangle$ *if* $\Sigma_i' \supset \Sigma_i$, *(*$i = 1,...,k$*). A multi-typing* $T$ *is a principal multi typing for* $e$ if the multi-typings for $e$ are exactly the thinnings of the substitution instances $\mathrm{S}T$ of $T$.

If $B = \{x_i{:}\Sigma_i\}_i$ is a multi-basis, let $B|x_i \equiv \Sigma_i$, let $B|\langle y_1,...,y_k\rangle \equiv \langle B|y_1,\ldots,B|y_k\rangle$, and let $exclude(B,\bar{y}) \equiv B - \bigcup_{j\le k} y_j(B|y_j)$.

A *multi-type inference calculus* assigns to each $\psi \in \Psi$ an *inference rule* $R_\psi = \langle\bar{\Sigma}_\psi;\bar{\rho}_\psi;\tau_\psi\rangle$, where $\bar{\Sigma}_\psi \in \tilde{\mathrm{T}}^{p_\psi}$, $\bar{\rho}_\psi \in \mathrm{T}^{q_\psi}$, $\tau_\psi \in \mathrm{T}$. A set-up

$$\frac{\langle\, B,\ \bar{x}{:}\mathrm{S}\bar{\Sigma}_\psi \vdash e_i{:}\mathrm{S}\rho_i \,\rangle_{i \le q_\psi}}{B \vdash \psi[\bar{x}](\bar{e}){:}\mathrm{S}\tau_\psi}$$

where $\langle \rho_i \rangle_i = \overline{\rho}_\psi$ and S is a substitution is an *instance* of $R_\psi$. A *derivation of S* is defined as for a type inference calculus (except, of course, that the condition $(x{:}\sigma) \in B$ for a leaf is replaced by $\sigma \in B|x$). Clearly, if each $\Sigma_\psi$ is a singleton then the calculus is essentially the same as a type inference calculus.

## 2. Milner's type inference algorithm W for parametric type systems.

In this section we define the algorithm W of [Mil] to arbitrary type inference calculi of the kind defined in §1. We also prove soundness and syntactic completeness for the type assignment defined by W. The generalization to arbitrary inference rules renders both definition and proofs simpler and more transparent. We postpone discussion of type binding mechanisms, such as the ML construct **let**, to §§4-7.

### 2.1 Type inference strategies.

Consider an expression $e = fg$ in the λ-calculus. The type of $f$ is constrained by the context $e$ in which $f$ appears ("outer constraint"), as well as by the structure of $f$ ("inner constraint"). The two are related by the occurrences of the same variables in $f$ and in $g$. Essential to the matching of these constraints is J. Robinson's Unification Lemma [Rob].

The definition of a type inference algorithm proceeds naturally by structural induction, where the typing of $f = \psi[\overline{x}](\overline{e})$ is determined from the typing of the expression(s) $\overline{e}$ and the rule $R_\psi$. The context in which an expression (say $e \in \overline{e}$) appears constrains the possible types of both $e$ and the tuple $\overline{fv}(e)$ of the variables free in $e$. A typing algorithm must deduce, given an expression $e$, a sequence of types $\overline{\sigma}$ and a type $\tau$ such that $\overline{fv}(e){:}\overline{\sigma}$ and $e{:}\tau$ are imposed by the sructure of $e$. There are therefore four possible typing strategies, according to whether both, one or none of $\overline{\sigma}$ and $\tau$ are passed as paramters in recursive calls.

A parameter $\tau$ as above is used in a recursive call only in a last-step matching of types. This step can equally be performed by the calling procedure, thereby allowing a simplification of the algorithm for atomic expressions. Thus, only two strategies are of interest:

(1) $\overline{\sigma}$ is passed as parameter; the algorithm finds S and $\tau$ such that $\overline{x}{:}S\overline{\sigma} \vdash e{:}\tau$.

(2) No parameter passed; find $\overline{\sigma}$ and $\tau$ such that $\overline{x}{:}\overline{\sigma} \vdash e{:}\tau$.

Strategy (1) is the one adopted for W in [Mil]. Strategy (2) is inefficient when only simple bases are used, since the types $\overline{\sigma}$ assigned to $\overline{fv}(e)$ have to be recalculated for each recursive call. However, for multi-bases only $\overline{\sigma}_i$'s for freshly bound $x_i$'s need be matched. This use of multi-bases allows the definition of an algorithm based on (2), which we call V. This alternative to W seems to be conceptually simpler, and it applies to the more general multi-typing systems, which are of independent interest in relation to type disciplines based on a type-conjunction constructor as described in § 4.3 below. Computationally V is as efficient as W for sequential implementations, but is more efficient when concurrency is used. In addition, the fact that the typing of one subexpression is computationally unrelated to the typing of any subexpression disjoint therefrom permits a more efficient update of typing when a subexpression is altered (I am grateful to Robin Milner for this observation).

### 2.2 Definition of W for parametric type inference calculi.

Fix a type inference calculus. The algorithm W takes as input an expression $e$ and a basis $B$ (with the intention that $B \supset (\overline{fv}(e){:}\overline{\sigma})$ for some $\overline{\sigma}$). The output, when W succeeds (that is — does not abort) is a substitution S and a type $\tau$. The intended properties of W are:

(1) (Soundness) If $W(B,e)$ succeeds, with result $(S,\tau)$, then $SB \vdash e{:}\tau$.

(2) (Syntactic completeness) If $SB \vdash e{:}\tau$ then $W(B,e)$ is a principal typing for $e$, i.e., $W(B,e)$ succeeds with result $(S_0,\tau_0)$, and $S = RS_0$, $\tau = R\tau_0$ for some substitution R.

(3) W runs in time $O(n^2)$, where $n = |B|+|e|$.

By J. Robinson's Unification Lemma [Rob] there is a linear-time procedure *match* such that, if $\overline{\sigma}, \overline{\tau} \in T^*$ and $K\overline{\sigma} = M\overline{\tau}$, then $(K_0,M_0) = match(\overline{\sigma};\overline{\tau})$ succeeds, $K_0\overline{\sigma} = M_0\overline{\tau}$, and $K = RK_0$, $M = RM_0$ for some substitution R. For example, if $(K,M) = match(\tau,\sigma;s\rightarrow t,s)$, and U is a principal unifier for $\tau,\sigma\rightarrow a$ ($a$ new), then K is (modulo variable renaming) the restriction of U to variables occurring in $\tau$ and $\sigma$, and $Mt = Ua$.

*Definition of* W.

```
procedure W  ($B$:basis, $e$:E)
                returns (S:substitution, $\tau$:T);
    S := $\emptyset$;
    if        $e \in C_a$  →  $\tau := a$;
          □  $e \in X$ & $(e{:}\sigma) \in B$  →  $\tau := \sigma$;
          □  $e \equiv \psi[\overline{x}](e_1,\ldots,e_k)$  →
                 $\theta := \langle\rangle$;  $C := (B,\overline{x}{:}\overline{\sigma}_\psi)$;
                 do  i := 1 to k;
                     $(T,\xi) := W(SC,e_i)$;
                     $\theta := (T\theta \leftarrow \xi)$
                     S := TS;
                 od;
                 $(K,M) := match(S\overline{\sigma}_\psi,\theta;\overline{\sigma}_\psi,\overline{\rho}_\psi)$;
                 S := KS;  $\tau := M\tau_\psi$;
    fi;
end W;
```

Here the **if** construct is Dijkstra's; in particular, if no □ clause applies then the procedure aborts with a failure message.

For example, consider a call of W for $e = \lambda[x](f) \equiv \lambda x.f$.

```
    $C := (B,x{:}s)$;
    $(T,\xi) := W(C,f)$;
                            Note:  here $\theta := \langle\xi\rangle$,  S := T
    $(K,M) := match(Ts,\xi;s,t)$;
    S := KT;
    $\tau := M(s\rightarrow t)$;
```

### 2.3 Soundness of W.

Theorem 2.1. If $(S,\tau) = W(B,e)$ succeeds, then $SB \vdash \tau{:}e$.

Proof. By induction on $e$. The induction basis is trivial. For the induction step, suppose $e \equiv \psi[\overline{x}](e_1,\ldots,e_k)$, and let $\overline{o} \equiv \overline{\sigma}_\psi$, $\overline{\rho} \equiv \langle\rho_i\rangle_i \equiv \overline{\rho}_\psi$. Let $T_i$, $\xi_i$, $S_i$ be the values of T, $\xi$ and S defined in the $i$th run of the do-loop of W; so $S_i \equiv T_i \cdots T_1$. Let $R_i \equiv T_k \cdots T_{i+1}$, so $R_iS_i \equiv S_k$, and $\theta \equiv \langle R_1\xi_1, \ldots, R_k\xi_k\rangle$. By assumption $(T_i,\xi_i) := W(S_{i-1}C,e_i)$ succeeds, so by induction assumption $T_iS_{i-1}C \vdash e_i:\xi_i$, i.e. $S_iC \vdash e_i:\xi_i$, whence $R_iS_iC \vdash e_i:R_i\xi_i$, and so

(1) $S_kB, \bar{x}:S_k\bar{\sigma} \vdash e_i:R_i\xi_i$

since $R_iS_i \equiv S_k$ and $C \equiv (B.\bar{x}:\bar{\sigma})$. By assumpion

$(K,M) := match(\langle S_k\bar{\sigma}; R_1\xi_1, \ldots, R_k\xi_k\rangle,\langle\bar{\sigma};\bar{\rho}\rangle)$

succeeds. Also, from (1) we have trivially

$KS_kB, \bar{x}:KS_k\bar{\sigma} \vdash e_i:KR_i\xi_i$.

Since, by the definition of *match* $KS_k\bar{\sigma} \equiv M\bar{\sigma}$, $KR_i\xi_i \equiv M\rho_i$, it follows that

$SB, \bar{x}:M\bar{\sigma} \vdash e_i:M\rho_i$,

where $S \equiv KS_k$. By $R_\psi$ this implies $SB \vdash \psi[\bar{x}](\bar{e}):\tau$, where $\tau \equiv M\tau_\psi$. ■

## 2.4 Syntactic completeness of W.

**Theorem 2.2.** Suppose $SB \vdash e:\tau$; then $(S_0,\tau_0) := W(B,e)$ succeeds, and $S \equiv PS_0$, $\tau \equiv P\tau_0$ for some substitution P.

**Proof.** By induction on the derivation of $SB \vdash e:\tau$. The induction basis is trivial. For the induction step consider

$$\frac{\langle SB, \bar{x}:N\bar{\sigma} \vdash e_i:N\rho_i\rangle_i}{SB \vdash \psi[\bar{x}](\bar{e}):N\tau_\psi}$$

where $\bar{\sigma} \equiv \bar{\sigma}_\psi$, $\bar{\rho} \equiv \langle\rho_i\rangle_i \equiv \bar{\rho}_\psi$. We may assume without loss of generality that $N = S$. (For suppose $\bar{t}$ are the type-variables occurring in all three of $\bar{\sigma}$, $B$ and *domain*(S); let $\bar{t}'$ be a fresh tuple of variables, $B' \equiv B[\bar{t}'/\bar{t}]$, $S' = S[\bar{t}/\bar{t}']$, and $Q \equiv S' \cup N$. Then $SB = S'B' = QB'$ and $N\bar{\sigma} = Q\bar{\sigma}$.)

Let $T_i$, $\xi_i$ and $S_i$ be defined as in 2.3, and let $R_{ji} \equiv T_j \cdots T_{i+1}$ (so $R_i \equiv R_{ki}$).

**Lemma.** Let $j \leq k$. There are substitutions $P_1,\ldots,P_j$ such that:

$(1)_j$ $S\rho_j = P_j\xi_j$;

$(2)_j$ $S = P_jS_j$;

$(3)_{ji}$ $P_i = P_jR_{ji}$ (for all $i \leq j$).

**Proof of lemma.** Induction on j. *Basis:* By the theorem's induction assumption $(T_1,\xi_1) := W(S_1C,e_1)$ succeeds, and $S = P_1S_1$, $S\rho_1 = P_1\xi_1$ for some $P_1$, hence $(1)_1$, $(2)_1$, while $(3)_{11}$ is trivial.

*Induction Step:* By induction assumption $S = P_{j-1}S_{j-1}$, and by assumption $SC \vdash e_j:S\rho_j$, i.e. $P_{j-1}(S_{j-1}C) \vdash e_j:S\rho_j$. By the theorem's induction assumption $(T_j,\xi_j) := W(S_{j-1}C,e_j)$ succeeds, with $P_{j-1} = P_jT_j$, $S\rho_j = P_j\xi_j$ for some substitution $P_j$. The latter is $(1)_j$, and it also follows

$S = P_{j-1}S_{j-1}$ (assuming $(2)_{j-1}$)

$= P_jT_jS_{j-1} = P_jS_j$, i.e. $(2)_j$.

Also, for $i < j$,

$P_i = P_{j-1}R_{j-1,i}$ (assuming $(3)_{j-1,i}$)

$= P_jT_jR_{j-1,i} = P_jR_{ji}$, i.e. $(3)_{ji}$.

$(3)_{jj}$ is trivial. ▣ **lemma**

**Proof of theorem concluded.**

The lemma implies

$S = P_kS_k$, $S\rho_j = P_j\xi_j = P_kR_j\xi_j$;

hence

(4) $P_k\langle S_k\bar{\sigma}; R_1\xi_1, \ldots, R_k\xi_k\rangle = S\langle\bar{\sigma};\bar{\rho}\rangle$.

Therefore $(K,M) := match(\langle S_k\bar{\sigma},\theta\rangle,\langle\bar{\sigma},\bar{\rho}\rangle)$ succeeds, with

(5) $P_k = PK$, $S = PM$,

for some substitution P. Hence $(S_0,\tau_0) := W(B,e)$ succeeds, and

$S = P_kS_k$ by (4)

$= PKS_k$ by (5)

$= PS_0$ by the definition of $S_0$,

and

$\tau = S\tau_\psi$ by assumption

$= PM\tau_\psi$ by (5)

$= P\tau_0$ by the definition of $\tau_0$. ■

## 3. A type inference algorithm V with no context-parameters.

### 3.1. Definition of the algorithm V.

We define an algorithm V following strategy (2) of §2.1, with the advantages listed there. The input is an expression $e \in E$, and the output is a multi-typing for $e$. The intended properties of V are:

(1) (Soundness) If $V(e)$ succeeds, with result $(B,\tau)$, then $B \vdash e:\tau$.

(2) (Syntactic completeness) If $B \vdash e:\tau$ for a multi-base $B$, then $V(e)$ succeeds, with result $(B_0,\tau_0)$, and $(B,\tau) = S(B_0,\tau_0)$ for some substitution S.

(3) $V(e)$ runs in time $O(n^2)$, where $n = |e|$.

The linear-time procedure *mmatch* (for "multi-match") is an inessential generalization of *match*. Suppose $\bar{A}$, $\bar{\Sigma} \in \tilde{T}^m$, $\bar{\xi}$, $\bar{\rho} \in T^k$. If $K(\bar{A},\bar{\xi}) = M(\bar{\Sigma},\bar{\rho})$, then $(K_0,M_0) := mmatch(\bar{A},\bar{\xi};\bar{\sigma},\bar{\rho})$ succeeds, $K_0(\bar{A},\bar{\xi}) = M_0(\bar{\Sigma},\bar{\rho})$, and $(K,M) = S(K_0,M_0)$ for some substitution S.

The linear-time procedure *join* is another off-spring of the Unification Algorithm. If $\bar{M} \equiv (M^1,\ldots,M^k)$ is a tuple of substitutions, and $P^iM^i = Q$ for $i = 1,\ldots,k$, then $(\bar{P}_0,Q_0) := join(\bar{M})$ succeeds, $\bar{P}_0 = \langle\bar{P}_0^1, \ldots, \bar{P}_0^k\rangle$, $P_0^iM^i = Q_0$, and $(\bar{P},Q) \equiv S(\bar{P}_0,Q_0)$ for some substitution S.

For each $e \in E$ let $t_e$ be a type variable distinct from all others in the context.

• *Definition of* V.

```
procedure V ($e$:E)
            returns ($B$:$multibase$, $\tau$:T);
  if      $e \in C_a \rightarrow \tau := a$;
     □ $e \in X \rightarrow \tau := t_e$;
     □ $e \equiv \psi[\bar{x}](e_1,\ldots,e_k) \rightarrow$
       parallel.do i= 1 to k;
          $(C_i,\xi_i) := V(e_i)$;
          $A_i := C_i|\bar{x}$;
          $(K_i,M_i) := mmatch(A_i,\xi_i;\bar{\Sigma}_\psi,\rho_i)$;
          $B_i := K_i exclude(C_i,\bar{x})$;
       od.lellarap;
       $(\bar{P},Q) := join(\bar{M})$;
       $B := \bigcup_i P_iB_i$;  $\tau := Q\tau_\psi$;
  fi;
end V;
```

For example, consider a call of V for an expression $e = ap(f,g) \equiv f(g)$.

```
                    cobegin
$(B_1,\xi_1) := V(f)$;                       $(B_2,\xi_2) := V(g)$;
$(K_1,M_1) := match(\xi_1, s\to t)$          $M_2 := [\xi_2/s]$;
                    coend
        $(P_1,P_2,Q) := join(M_1,M_2)$;
        $B := P_1K_1B_1 \bigcup P_2B_2$;
        $\tau := Qt$;
```

## 3.2. Soundness of V.

**Theorem 3.1.** If $(B,\tau) := V(e)$ succeeds, then $B \vdash e{:}\tau$.

**Proof.** By induction on $e$. The induction basis is trivial. *Induction Step:* Suppose $e = \psi[\bar{x}](e_1,...,e_k)$. By assumption $(C_i,\xi_i) := V(e_i)$ succeeds, and $C_i \vdash e_i{:}\xi_i$. Let $A_i = C_i|\bar{x}$. By assumption $(K_i,M_i) := mmatch(A_i,\xi_i;\bar{\Sigma}_\psi,\rho_i)$ succeeds. Let $B_i = K_i exclude(C_i,\bar{x})$. We have $K_iC_i \vdash e_i{:}K_i\xi_i$, $K_iC_i = B_i \cup (\bar{x}{:}K_iA_i) = B_i \cup (\bar{x}{:}M_i\bar{\Sigma}_\psi)$, and $K_i\xi_i = M_i\rho_i$. Hence $B_i,\bar{x}{:}\bar{\Sigma}_\psi \vdash e_i{:}M_i\rho_i$.

By assumption $(\bar{P},Q) := join(\bar{M})$ succeeds, with $P_iM_i = Q$ for all i, so $B,\bar{x}{:}Q\bar{\Sigma}_\psi \vdash e_i{:}Q\rho_i$, from which $B \vdash e{:}Q\tau_\psi$ follows by $R_\psi$. ■

## 3.3. Syntactic completeness of V.

**Theorem 3.2.** If $B \vdash e{:}\tau$ where $B$ types $\overline{fv}(e)$, then $(B_0,\tau_0) := V(e)$ succeeds, and $B = SB_0$, $\tau = S\tau_0$ for some substitution S.

**Proof.** By induction on the derivation of $B \vdash e{:}\tau$. The induction basis is trivial. For the induction step consider the inference

$$\frac{\langle B,\bar{x}{:}N\bar{\Sigma}_\psi \vdash e_i{:}N\rho_i\rangle_i}{B \vdash \psi[\bar{x}](e_1,...,e_k){:}N\tau_\psi}$$

By induction assumption, for each $i = 1,...,k$ $(C_i,\xi_i) := V(e_i)$ succeeds, and there is a substitution $S_i$ such that $N\rho_i = S_i\xi_i$, and $(B,\bar{x}{:}N\bar{\Sigma}_\psi) = S_iC_i$, whence $N\bar{\Sigma}_\psi = S_iA_i$, and $B = S_i exclude(C_i,\bar{x})$. Thus $S_i(A_i,\xi_i) = N(\bar{\Sigma}_\psi,\rho_i)$. Therefore $(K_i,M_i) := mmatch(A_i,\xi_i;\bar{\Sigma}_\psi,\rho_i)$ succeeds, and $S_i = T_iK_i$, $N = T_iM_i$ for some substitution $T_i$. Hence $(\bar{P},Q) := join(\bar{M})$ succeeds, with $T_i = SP_i$, $N = SQ$ for some sustitution S. Let $C_i' = exclude(C_i,\bar{x})$. Then $B = S_iC_i' = T_iK_iC_i' = SP_iK_iC_i'$ for all i. So $B = S\{\bigcup_i P_iK_iC_i'\} = SB_0$ and $N\tau_\psi = SQ\tau_\psi = S\tau_0$. ■

## 4. Polymorphic type disciplines.

Parametric type disciplines of the kind discussed so far permit the definition of an expression (procedure) $e$ whose arguments are of parametric types. There remains to control generic *uses* of such expressions. For example, the expression $I = \lambda x.x$, of type $t\to t$, is used twice in $\langle I5, I\mathrm{true}\rangle$, as well as in $II$. The issue is then to type expressions like $(\lambda z.e)I$ with $e = \langle z5, z\mathrm{true}\rangle$ or $e = zz$, where the genericity of $I$ has to be conveyed explicitly in typing the variable $z$.

We describe four existing mechanisms for syntactically controlling the typing of an expression that is applied to expressions of different types: type abstraction (Reynolds [Rey], Girard [Gir]), type quantification (McQueen and Sethi [MS]), type conjunction (Coppo and Dezani-Ciancaglini [C, CD, CDV81, Sal, BCD]), and the construct let of the programming language ML (Milner [Mil]). We also mention some properties that we use in the sequel.

### 4.1 Type abstraction.

Consider an expression $e$ of generic type $\tau[t]$ (where $t$ is a free type variable). Under this approach the dependency of the meaning of $e$ on the instances $\tau[\sigma/t]$ of the type is viewed as a functional dependency. To make this functionality explicit one abstracts over the type variable $t$, say $\Lambda t.e$, in analogy to the use of $\lambda$-abstraction to render usual functionality explicit. Thus, the expression $\Lambda t.e$ denotes a function from types $\sigma$ to objects, denoted by $e[\sigma/t]$, of type $\tau[\sigma/t]$. The expression $\Lambda t.e$ can now be applied to any type $\rho$, yielding $(\Lambda t.e)\rho$. The type of $\Lambda t.e$ is denoted by $\Delta t.\tau$. Thus $\Delta$ is a variable binding type constructor, and if $e$ is of type $\Delta t.\tau$ then $e\rho$ is of type $\tau[\rho/t]$. Thus, if $\iota$ denotes the type $\Delta t.t\to t$, then $x^\iota$ is of type $\iota\to\iota$, so $x^\iota\iota x^\iota$ is of type $\iota$.

This type discipline departs from simple and parametric typing in its use of types both as conditions on objects and as arguments to objects. Syntactically this duality is represented by the two-tier use of type expressions.

The description above is conveyed by two inference rules on typing, analogous to the rule for $\lambda$-abstraction and for object application:

$$\frac{B \vdash e:\tau}{B \vdash \Lambda t.e : \Delta t.\tau}$$

($t$ not free in $B$), and

$$\frac{B \vdash e : \Delta t.\tau}{B \vdash e\sigma : \tau[\sigma/t]}$$

($\sigma$ free for $t$ in $\tau$).

Using common techniques of Proof Theory it can be shown that every expression of the $\lambda$-calculus that can be typed in this fashion reduces to a normal expression [FLO]. However, this statement cannot be proved in a formal theory as strong as full (impredicative) Mathematical Analysis (Second Order Arithmetic) [Lei].

The representation of functions over the natural numbers by $\lambda$-expressions typed in this discipline was initiated by S. Fortune and M. O'Donnell [For, ODo, FLO]. Church's definition of the $n$'th numeral, as

$\bar{n} = \lambda f.\lambda x.f...fx$ ($n$ iterations of $f$),

yields numerals of generic type $(t\to t)\to(t\to t)$. Using type abstraction this representation is further developed to

$\bar{n} = \Lambda t.\lambda f^{t\to t}.\lambda x^t.f...fx$,

of type $\Delta t.((t\to t)\to(t\to t))$. Numerals can now be used to iterate the unfolding of expressions with successively different types.

From the proof of the normalization theorem quoted above it follows that every function represented by a typed expression can be proved total in Mathematical Analysis. Statman [Sta] showed the converse: every recursive function $F$ that can be proved in Mathematical Analysis to be total is represented, modulo the Fortune-O'Donnell representation of numerals, by some typed expression $e$; that is, $e\bar{n}$ reduces to $\overline{Fn}$ for all positive integers $n$.

## 4.2 Type quantification.

This approach [MS] is based on the view that an expression like $I$ is of type $\tau\rightarrow\tau$ for all types $\tau$, and that no functionality over types has to be introduced to the level of objects to express this property. To permit multiple uses of generic expressions this implicit quantification has to be made explicit. Thus $I$ is assigned type $\Pi t.t\rightarrow t$ (read "for all $t$, $t\rightarrow t$"). The deduction rules for type quantification are therefore

$$\frac{B \mid\!- e:\tau}{B \mid\!- e:\Pi t.\tau}$$

($t$ not free in $B$), and

$$\frac{B \mid\!- e:\Pi t.\tau}{B \mid\!- e:\tau[\sigma/t]}$$

($\sigma$ free for $t$ in $\tau$).

Semantically this is indeed an approach altogether different from type abstraction. Types are no longer object-arguments, but at the cost of admitting objects that are not uniquely typed. Also, the discipline of type abstraction can be extended with primitive objects that take types as arguments; for instance, an object $d$ of type $\Delta t.\mathbf{bool}$ such that $d\tau$ = true iff $\tau$ is an atomic type. This, however, is not compatible with the discipline of type quantification.

Although differently motivated, the discipline of type quantification is, from the combinatorial viewpoint of type deduction, merely a notational variant of type abstraction. In fact, the following theorem follows immediately from the definition of the inference rules for type abstraction and for type quantification.

**Theorem 4.1.** Let $e$ be an expression of the type abstraction discipline. Let $e'$ be the untyped expression that results by deleting all references to types in $e$, as type superscipts, arguments or abstraction. Then $e$ is typed in the type abstraction discipline iff $e'$ is typed in the type quantification discipline.

Moreover, if D is a deduction of $\vec{x}:\vec{\sigma} \mid\!- e:\tau$ in the type abstraction discipline, and D' results from suppressing in D type operations in expressions and replacing $\Delta$ by $\Pi$, then D' is a deduction of $\vec{x}':\vec{\sigma}' \mid\!- e':\tau'$ in the type quantification discipline, where $\rho'$ is $\rho$ with $\Delta$ replace by $\Pi$. Conversely, if D is a deduction of $\vec{x}:\vec{\sigma} \mid\!- e:\tau$ in the type quantification discipline, and $D^+$ results from D by replacing $\Pi$ by $\Delta$, superscripting variables with the type assigned to them, and introducing type abstraction and type application into expressions corresponding to quantification rules in D, then $D^+$ is a deduction of $\vec{x}^+:\vec{\sigma}^+ \mid\!- e^+:\tau^+$ in the type abstraction discipline, where $e^+$ is $e$ appropriately decorated with a type structure, and $\tau^+$ is $\tau$ with $\Pi$ replaced by $\Delta$. ■

## 4.3 Type conjunction.

This approach is conceptually akin to type quantification (which it historically precedes), but combinatorially it is somewhat different. The new type constructor here is *type conjunction* (or *type intersection*): if $\tau$ and $\sigma$ are types then so is $\tau\&\sigma$, and an expression $e$ is of type $\tau\&\sigma$ iff $e$ is both of type $\tau$ and of type $\sigma$. Again, an expression, say $I$, is thought of as being of type $\tau\rightarrow\tau$ for every type $\tau$, but only finitely many of those can be mentioned at any given time, by considering the conjunctive type $(\tau_1\rightarrow\tau_1)\ \&\ \cdots\ \&\ (\tau_k\rightarrow\tau_k)$. The choice of the conjuncts would depend on the context. For instance, if $e$ is $\langle z5, z\mathbf{true}\rangle$, then $z$ may be assigned type $(\mathbf{int}\rightarrow p)\&(\mathbf{bool}\rightarrow q)$, and in $(\lambda z.e)I$ the variable $z$ might be assigned type $(\mathbf{int}\rightarrow\mathbf{int})\&(\mathbf{bool}\rightarrow\mathbf{bool})$. A more interesting example: to correctly type $zz$ the variable $z$ may be assigned type $s\&.s\rightarrow t$; and in $(\lambda z.zz)I$ $z$ may be assigned type $(q\rightarrow q)\&((q\rightarrow q)\rightarrow(q\rightarrow q))$.

The type inference rules conveying the intended properties of type conjunction are naturally

$$\frac{B \mid\!- e:\tau_1 \quad B \mid\!- e:\tau_2}{B \mid\!- e:\tau_1\&\tau_2}$$

and

$$\frac{B \mid\!- e:\tau_1\&\tau_2}{B \mid\!- e:\tau_i}$$

($i$ = 1,2).

The salient property of the conjunctive discipline is the following

**Theorem 4.2.** [Cop, CDV81, Pot] The typable expressions are precisely the $\lambda$-expressions that are strongly normalizable (i.e. for which every reduction sequence terminates).

**Proof.** Each typable expression is strongly normalizabe because the type deduction calculus is isomorphic to Gentzen natural deduction calculus for positive propositional logic [Pra], appropriately restricted (&-introduction is permitted only when the two subderivations are isomorphic); the restriction is preserved under reductions, and the strong normalization theorem for the calculus in question is well-known [Pra] (for a direct proof see [Cop, theorem 3]).

To see that each strongly normalizable expression $e$ is typable proceed by induction on the length of the shortest reduction sequence on $e$. The induction basis, with $e$ normal, is handled by a straightforward induction on the size of $e$; e.g., given $e = xe_1\cdots e_k$, with $e_i$ already assigned type $\tau_i$, and with the free variable $x$ assigned type $\sigma_i$ in $e_i$, assign to $x$ in $e$ the type $\sigma_1\&\cdots\sigma_k\&(\tau_1\rightarrow\tau_2\rightarrow\cdots\rightarrow\tau_k\rightarrow t)$ (where $t$ is fresh). The induction step consists in showing that an expression $(\lambda x.a)b$ is typable whenever its $\beta$-redex $a[b/x]$ is, simply by assigning to $x$ the conjunction of all types that are assigned to $b$ within the redex. ■

A methodological flaw of the conjunctive discipline is the non-uniform character it attaches to polymorphism. For instance, assume that $e_1,\ldots,e_k$ have types $\tau_1,\ldots,\tau_k$ respectively; then the variable $x$ in $\langle xe_1,\ldots,xe_k\rangle$ has to be assigned a conjunctive type with types of the forms $\tau_1\rightarrow\sigma_1,\ldots,\tau_k\rightarrow\sigma_k$ among its conjuncts, and these conjuncts would have to be modified and multiplied with changes made in the tuple of $e_i$'s.

## 4.4 The ML construct *let*.

Here the issue of assigning an explicitly generic type to the variable $z$ in $\lambda z.e$, where $z$ is used generically in $e$, is avoided altogether. The driving idea is that such abstraction expressions should only appear applied to a generic expression, as in $(\lambda z.e)d$, and that the compound expression can be viewed as a notational variant of the reduct $e[d/z]$, with $z$ serving as a mere placeholder. In fact, to distinguish compound expressions like this a new construct **let** is introduced, with $(\lambda z.e)d$ being written as **let** $z = d$ **in** $e$.

An anomalous aspect of this mechanism is that it permits legal expressions (e.g. **let** $z = \lambda x.x$ **in** $zz$) with illegal, though well-formed, sub-expressions $(zz)$.

A fundamental limitation of this approach is that essentially only parametric polymorphism is being allowed here: an expression that is defined by abstraction over polymorphic objects cannot appear as an argument of another expression. Put in the parlance of procedure calls: a procedure calling generic procedures cannot be called. While this restriction is accommodated by the more naive uses of polymorphism, it is rather easy to construct natural examples that violate it. For instance, one might wish to have some procedure that transforms arrays of objects of type $t$ (partial-sorting, doubling, padding, thinning...) called by a procedure that calls a variety of such generic array-transformers (iteration, composition, piecemeal application to arrays that mix types...). However, this limitation had been consciously motivated by the wish the designers of ML had to provide an efficient type inference facility to the user of the language. We return to this point in §7.

## 5. Type inference for the abstraction and quantification disciplines.

By theorem 4.1 the type abstraction and the type quantification disciplines are combinatorially identical as deductive systems in general, and for type inference in particular.

The typability in the conjunctive discipline of all strongly normalizable expressions implies the impossibility of effective type inference for the conjunctive discipline, as we discuss in §6. We note from the outset that the quantificational discipline is much tamer. It clearly *is* possible to type every normal expression, for instance by assigning to each variable the type $\Pi t.t$. However, not every strongly normalizable expression is typable. Consider, for example, the expression $e = d(\lambda x.x)(uu)$, where $d$ is $\lambda z.(zu)(z\lambda x.x)$. $e$ reduces to the normal expression $u(\lambda x.x)(uu)$ by any reduction sequence. Since $d$ is applied to an abstraction expression, $z$ must be assigned a type of the form $\sigma \rightarrow \tau$. The presence of the sub-expression $z\lambda x.x$ implies that $\sigma$ must itself be an implicational type $\sigma_1 \rightarrow \sigma_2$. On the other hand, the presence of $zu$ implies that $\sigma$ must also be a type for $u$, while the type of $u$ must be of the form $\Pi t.\rho$, because of the presence of $uu$. This is a contradiction.

In fact, the proliferation of strongly normalizable but non-typable expressions, indeed the fact that the set of such expressions is non-recursive, follows from the following

**Theorem 5.1.** The set of $\lambda$-expressions typable in the quantificational discipline is recursive.

**Proof.** We outline a (rather inefficient) type inference algorithm for the quantificational discipline, restricted to the paradigmatic constructs of $\lambda$-abstraction and $\lambda$-application. In an expanded version of the present paper, to be published, we present a somewhat more feasible algorithm, discuss its implementation, and treat type systems with (almost) arbitrary inference rules. Since we are not interested in efficiency at this point, we shall use unmodified strategy (2) described in §2.1, which is conceptually and technically the simplest.

**Definition.** A *cut-off quantification* of a type expression $\tau$ is an expression $\Pi t.\tau[t/\sigma]$ for some $\sigma$ (NB: $t$ is substituted for $\sigma$). Thus $\tau'$ is a cut-off quantification of $\tau$ exactly when $\tau$ is a quantifier-elimination instance of $\tau'$.

**Lemma 1.** A typing $x{:}\tau \mid\!- x{:}\sigma$ is derived iff there is a type $\rho$ such that $\tau$ is a cut-off quantification of $\rho$ and $\sigma$ is a quantification of $\rho$ over variables not free in $\tau$.

**Proof of lemma 1.** A natural deduction derivation of the typing above is a linear list $x{:}\tau_0, \ldots, x{:}\tau_k$, with $\tau_0 \equiv \tau$ and $\tau_k \equiv \sigma$, where $\tau_{i+1}$ is obtained from $\tau_i$ by $\Pi$-introduction or $\Pi$-elimination. Without loss of generality all $\Pi$-eliminations precede all $\Pi$-introductions. Let $\rho$ be the first $\tau_j$ such that the $j$'th typing in the list is the first one that is not the premise of $\Pi$-elimination (i.e., the "bottom" of the chain, in a natural sense). ■ lemma 1

**Definitions.** Let $\sigma$ be an (occurrence of a) subexpression of $\tau$. $\sigma$ is *simply positive* in $\tau$ if $\sigma$ is not in the negative (i.e. left) scope of any instance of $\rightarrow$. $\rho$ is *simply negative* in $\tau$ if there is a simply positive subexpression of $\tau$ where $\rho$ falls in the scope of exactly one instance of $\rightarrow$, negatively.

Let $\tau' = S\tau$. A subexpression $\sigma$ of $\tau'$ is *originally rooted (with respect to* S*)* if the main symbol of $\sigma$ is an original symbol-occurrence of $\tau$.

Let $T = (\bar{\sigma}, \tau)$ be a typing. A typing $T'$ is a *general instance of* $T$ *(by substitution* S) if it is obtained by:

(a) applying S; then

(b) cut-off quantifying any number of $\sigma_i \in \bar{\sigma}$ and of simply negative originally rooted (with respect to S) subexpressions of the substitution instance of $\tau$; and finally

(c) binding any number of simply positive originally rooted subexpressions of $\tau$ with any number of non-vacuous quantifiers over variables not occurring free out of that scope.

Examples: By lemma 1 $x{:}\tau \mid\!- x{:}\sigma$ is derived iff $\langle\tau;\sigma\rangle$ is a general instance of $\langle t;t\rangle$.

$\langle \Pi t.t \rightarrow s; \Pi t.t \rightarrow s\rangle$ is a general instance of both $\langle t;t\rangle$ and $\langle t \rightarrow s; t \rightarrow s\rangle$, because it is obtained from $\langle t;t\rangle$ by (a) and from $\langle t \rightarrow s; t \rightarrow s\rangle$ by (b) and (c).

**Lemma 2.** Say that typing $\langle B_1; \tau_1\rangle$ *matches* $\langle B_2; \tau_2\rangle$ if $B_1$ and $B_2$ agree on the common variables, and $\tau_1$ is of the form $\tau_2 \rightarrow \sigma$. There is an algorithm *match* that, given typings $T_1$ and $T_2$, yields a finite set $R \equiv match(T_1, T_2)$ of pairs $S_1, S_2$ of typings, such that general instances $T_1', T_2'$ of $T_1, T_2$ match (in the sense above) iff $T_i'$ is a general instance of $S_i$ $(i = 1,2)$ for some $(S_1, S_2) \in R$.

**Proof of lemma 2.** By a direct modification of (first order) unification. Essential to the proof is that the quantifications involved in operations (b) and (c) above are only over originally rooted subexpressions. ■ lemma 2

**Proof of the theorem.** We define an algorithm that, given an expression $e$, yields a finite set $Q_e$ of typings such that $T$ is a typing of $e$ iff $T$ is a general instance of some element of $Q_e$. Proceed by structural recursion on $e$.

For the basis, $e$ a variable, take $Q_e := \{(t,t)\}$. This is all right by lemma 1.

For the case $e = ab$ (application) let $Q_e$ be the union of the sets obtained by applying *match* to all pairs $T_1$, $T_2$ with $T_1 \in Q_a$ and $T_2 \in Q_b$. Since $Q_a$ and $Q_b$ satisfy the required property by induction assumption, it easily follows that $Q_e$ does.

For the case $e = \lambda x.d$, let $Q_e$ be obtained by replacing each typing $\langle B, x{:}\sigma \,;\, \tau \rangle$ in $Q_d$ by $\langle B \,;\, \sigma \to \tau \rangle$. Again, it follows immediately from the induction assumption for $d$ that the required property holds for $Q_e$. ■

The reader familiar with the formula-as-type analogy from Proof Theory (see e.g. [How], and for the present context [FLO, Lei]) may find it useful to reformulate the statement of the theorem in terms of natural deduction proofs for purely implicational second order propositional logic: given a tree $e$ of instances of implication-introduction and of implication-elimination, it is decidable whether it can be completed into a correct derivation of *some* formula (possibly with instances of the quantification rules added).

At the risk of dwelling on the obvious we caution that the effective decidability of typability in the quantificational discipline is lost if we close types under $\beta$-equality:

**Proposition 5.2.** There is no effective procedure that decides, given a $\lambda$-expression $e$, whether $e$ is $\beta$-reducible to a typable expression. The same holds for $\beta$-equality.

**Proof.** Since the typable expressions are closed under $\beta$-reduction, it follows from the Church-Rosser Theorem that the statement for $\beta$-reducibility is equivalent to the statement for $\beta$-equality. The latter is an immediate consequence of Scott's Theorem [Bar, p. 140]. For an *ad hoc* reductive proof, of the statement for $\beta$-reducibility, recall that all typable expressions are normalizable, and that all normal expressions are typable. Thus, $e$ is reducible to a typable expression iff $e$ is normalizable, and the set of normalizable expressions is well-known to be non-recursive [Bar]. ■

## 6. Type inference for the conjunctive discipline.

### 6.1. Limitations on type inference.

**Theorem 6.1.** The set of typable expressions in the conjunctive discipline is not effectively decidable.

**Proof.** Recall that the typable expressions are precisely the strongly normalizable ones. That the set of strongly normalizable expressions is not recursive can be seen, for instance, from Scott's Theorem for the (closed expressions) of the $\lambda$I-calculus [Bar p. 141]. ■

Actually, even more is true:

**Corollary 6.2.** For any type $\tau$, the set of closed expressions $e$ such that $\vdash e{:}\tau$ is not effectively decidable.

**Proof.** A closed expression $d = xe$ (where $x$ is a variable not in $e$) is of type $\tau$ iff $e$ is typable. Thus, the typability problem effectively reduced to deciding $\vdash d{:}\tau$. By the theorem it follows that the latter is not recursively solvable. ■

In fact, it seems that a symmetric statement is also true: there is a closed $\lambda$-expression $e$ for which the set of types $\tau$ such that $\vdash e{:}\tau$ is not effectively decidable. To see this observe that, for Church's numerals,

$$\tau_n = (t_0 \to t_1) \& (t_1 \to t_2) \& \cdots \& (t_{n-1} \to t_n) \to t_0 \to t_n$$

is a simple typing of $\bar{n}$ that distinguishes between the numerals. Church's representation of numerals is used (for $n > 0$) also in the $\lambda$I-calculus. In general the numeral $\bar{n}$ can be typed by a multitude of other types. However, inspection of the standard representation of partial recursive functions in the $\lambda$I-calculus [Bar] seems to reveal that, if $e$ is an expression representing a unary recursive function, then $e\bar{n} =_\beta \bar{m}$ iff $e{:}\, \tau_n \to \tau_m$ is derived.

Let $F$ be a partial recursive function with a non-recursive domain, taking only 1 as value. Let $e$ be a $\lambda$-expression representing $F$ in the $\lambda$I-calculus. Then (assuming our statement above to be accurate) $e\bar{n}$ is typable iff $\tau_n \to \tau_1$ is one of the types of $e$. Hence, if the set of types of $e$ were recursive, then so would be the domain of F, contradicting the choice of $F$.

From theorems 5.1 and 6.1 it follows that there is no effective procedure that translates a typing for an expression $e$ in the conjunctive discipline into a typing for $e$ in the quantificational discipline. A translation in the opposite direction is clearly possible: since all quantificationally typed expressions are strongly normalizable, whence typable in the conjunctive discipline, it follows that a quantificational typing can be transformed into a conjunctive typing by exhaustive search. However, the following theorem shows that not much better can be done.

**Theorem 6.3.** There is no recursive function, proved total in Mathematical Analysis, that maps a typing of a $\lambda$-expression $e$ in the quantificational discipline to a bound on the size of types necessary to type $e$ in the quantificational discipline.

**Proof.** Arguing by contradiction, suppose that $\varphi$ is a function of the kind excluded by the theorem. Since $\varphi$ is provably total in Mathematical Analysis, there is a recursive function $\psi$, proved total in Mathematical Analysis, but not provably total in First Order Arithmetic, PA, extended with the (canonically coded) statement "$\varphi$ is total". By [Sta] there is a $\lambda$-expression $e$, typable in the quantificational discipline, that represents $\psi$. By our assumption on $\varphi$, $\varphi\lceil e\bar{n}\rceil$ , is a bound on the size of type deductions for $e\bar{n}$ in the conjunctive discipline. Since the proof of theorem 4.2 is formalizable in PA, it follows that the function $\psi n$ = "the normal form of $e\bar{n}$" is provably total in PA extended with the statement "$\varphi$ is total", contradicting our choice of $\psi$. ■

The parametric type systems, in addition to admitting effective type inference, also have the principal-typing property. This property is lost in passing to the conjunctive discipline: Already $I = \lambda x.x$ admits any type $\tau$ of the form $(t_1 \to t_1) \& \cdots \& (t_k \to t_k)$. If this seems too trivial, consider $\lambda z.zI$ [Cop] which admits all types $\tau \to s \to s$ with $\tau$ as above, for which it is easy to see that no principal type exists. A principal type property can be restored [CDV80], if in addition to substitution one admits a certain "expansion" operation. Roughly, an expansion of $\sigma$ over a type variable $t$ is obtained by choosing distinct subexpressions $\rho$ of $\sigma$ jointly containing all occurrences of $t$, choosing fresh variables $t_1, \ldots, t_k$, and replacing each $\rho$ by the conjunction of all $\rho[t_j/t]$, $j = 1,\ldots,k$.

## 6.2 Ranking of types, and type inference of bounded rank.

For simplicity we consider a type discipline where the type-inference rule for λ-application is the only rule that uses the same variable in distinct premises (in the schematic statement of the rule!), and where λ-abstraction is the only variable binding rule. More general type systems can be easily accommodated.

**Definition.** Let $\kappa$ be $\Delta$, $\Pi$ or $\&$ (according to the discipline). A type $\tau$ is of *rank* $r$ if there is no instance of $\kappa$ falling in $\tau$ in the negative scope (i.e. left hand side) of $r$ instances of $\rightarrow$. A typing $\langle\bar{\sigma};\tau\rangle$ is of rank $r$ if $\bar{\sigma}\rightarrow\tau$ (curried) is of rank $r$. The *restriction to rank* $r$ of a type deductive system is that system with all typings of rank $\leq r$. Thus, the restriction to rank 0 of a type system, in any one of the polymorphic type disciplines, is the corresponding parametric type system.

For example, let $\tau$, $\sigma$ and $\rho$ be conjunction free. Then the types $\tau\&\sigma$ and $\rho\rightarrow(\tau\&\sigma)$ are of rank 1, the type $(\tau\&\sigma)\rightarrow\rho$ is of rank 2, and $((\tau\&\sigma)\rightarrow\rho)\rightarrow\tau)$ is of rank 3. A generic procedure that has no generic procedure as formal parameter is of rank 1. If a procedure has as formal parameter a procedure of rank 1 (but no procedure of higher rank) then it is of rank 2.

The importance of negative nesting of $\rightarrow$ is well known from Proof Theory and from the uses of functionality in Arithmetic in all finite types. For instance, every functional of finite type of rank $n$ can be interpreted, using currying, by a functional of *the pure* $n$'th type, $\tau_n$, where the pure types are defined by $\tau_0 = \mathrm{N}$, $\tau_{n+1} = \tau_n\rightarrow\mathrm{N}$, and are themselves of rank $n$.

**Theorem 6.4.** If an expression $e$ is typable in any polymorphic discipline restricted to rank 1, then $e$ is typable in the corresponding parametric system.

**Proof.** Straightforward by induction on length of type derivations. ■

The *reduced form* $\tau_r$ of a conjunctive type $\tau$ is obtained by recursively replacing each subtype of the form $\rho\rightarrow(\sigma_1\& \cdots \&\sigma_k)$ by $(\rho\rightarrow\sigma_1)\&...\&(\rho\rightarrow\sigma_k)$. Thus in $\tau_r$ no conjunction occurs immediately on the right of an arrow (this is the conjunctive discipline as formulated originally in [Cop, CD, Sal]; unrestricted conjunction is introduced in [BCD]). It is easy to verify, by induction on type derivations, that $\langle\bar{\sigma};\tau\rangle$ is a derived typing for an expression $e$ iff $\langle\bar{\sigma}_r;\tau_r\rangle$ is derived using reduced types only. We therefore assume all types to be reduced. In particular, a reduced typing of rank 2 is of the form $\langle\bar{\sigma};\tau\rangle$, where each $\sigma$ is the conjunction of $\&$-free types, and $\tau$ is the conjunction of types of the form $\rho_1\rightarrow \cdots \rightarrow\rho_k\rightarrow\pi$, with each $\rho_i$ a conjunction of $\&$-free types, and $\pi$ $\&$-free.

We outline a modified version $V_2$ of the algorithm V that applies to the conjunctive polymorphic discipline restricted to rank 2. $V_2$ derives typings $\langle\bar{\sigma};\tau\rangle$ as above, but with $\tau$ of the form $\rho_1\rightarrow \cdots \rightarrow\rho_k\rightarrow\pi$ (only one conjunct).

Recall that, when applied to an expression $\lambda x.e$, V calls itself for $e$, then unifies the various types obtained for $x$. By contrast, $V_2$ takes the conjunction of these types.

Suppose that $V_2$ is to be applied to an expression $ed$. The type $\delta$ derived by $V_2$ for $d$ may be (strictly) of rank 2, implying rank 3 for the type of $e$. Therefore, $V_2$ starts by unifying, for each type-conjunction occurring negatively in $\delta$, all conjuncts thereof. (Recall that these conjuncts are $\&$-free). The resulting type, $\delta'$, is $\&$-free. The type $\varepsilon$ derived by $V_2$ for $e$ is of the form $\rho\rightarrow\pi$, where $\rho$ is the conjunction of $\&$-free types (except for the trivial case where $\varepsilon$ is a type variable or constant). $V_2$ proceed by unifying the conjuncts of $\rho$ with $\delta'$.

In all other respects $V_2$ is identical to V.

**Theorem 6.5.** The algorithm $V_2$, applied to an expression $e$ typable in rank 2, yields a typing $T$ derived for $e$ in the conjunctive discipline restricted to 2.

**Proof.** Straightforward by induction on $e$. ■

In fact, the typing obtained here is "principal" modulo expansion as defined above, redundant conjuncts on the left of $\rightarrow$, and transformation to a reduced form.

It seems that the result above is optimal within the rank hierarchy. In fact, it appears that already the conjunctive discipline restricted to rank 3 is undecidable, by the following argument. An inspection of the standard representations of partial recursive functions in the λI-calculus [Bar], seems to reveal that (modulo inessential modifications) only expressions of type restricted to rank 3 need to be used to type any normalizable "recursion theoretic" expression.

Referring to the exposition in [Bar pp. 184-187], the only points where simple typing fails are in the treatment of closure under primitive recursion and of closure under minimalization. The only point where simple typing would not do for closure under primitive recursion is the definition of the projection functions $\mathrm{U}^k_{\mathrm{i}}$, which are clearly of rank 2 ($x_i$ must be conjunctive), implying rank 3 for the tuple-projection functions $\mathrm{P}^k_{\mathrm{i}}$, but no other effect.

In the treatment of minimalization, the iteration-like operation $A_1$ is of rank 2 ($w$ must be conjunctive), whence W is of rank 3, but no other effect.

Let now $e$ represents a partial recursive function with a non-recursive domain. Then $e\bar{n}$ is typable in the conjunctive discipline restricted to rank 3 iff $e\bar{n}$ has a normal form. Since we chose $e$ so that the latter is undecidable, it follows that typability in the discipline restricted to rank 3 is also undecidable.

## 7. Type inference for the ML construct *let*.

Milner's [Mil] original definition of the algorithm W includes a clause for let, which corresponds naturally [DM] to the type inference rule

$$\frac{B \mid\!- e{:}\tau \qquad B,x{:}\Pi t.\tau \mid\!- d{:}\sigma}{B \mid\!- (\textbf{let } x{=}e \textbf{ in } d) : \sigma}$$

($t$ not free in $B$). This rule is derived in the quantificational discipline (restricted to rank 2):

$$\frac{\dfrac{B,\, x{:}\Pi t.\tau \mid\!- d{:}\sigma}{B \mid\!- \lambda x.d{:}\ \Pi t.\tau\rightarrow\sigma} \qquad \dfrac{B \mid\!- e{:}\tau}{B \mid\!- e{:}\Pi t.\tau}}{B \mid\!- (\lambda x.d)e : \sigma}$$

In [DM] the type inference algorithm of [Mil] is proved complete for this rule. This is satisfactory only to the extent that the ML inference rule for **let** given above is accepted as fully capturing the intended use of polymorphism. Our analysis of richer polymorphic disciplines enables us to better articulate and delineate the nature of polymorphism in ML.

**Theorem 7.1.** Let $e$ be a closed ML expression, and let $e^{\lambda}$ result from recursively replacing in $e$ each subexpression of the form **let** $c = x$ **in** $d$ by $(\lambda x.d)c$. Then a parametric type $\tau$ is derived for $e$ in ML iff $\tau$ is derived for $e$ in the restriction to rank 2 of either one of the quantification discipline, the conjunctive discipline, or the two combined. ■

One conclusion to be drawn from this theorem is that the type deduction rule for **let** in ML is indeed intrinsically satisfactory. Another conclusion is that the same effect would be achieved by working in the restriction to rank 2 of any one of the other polymorphic disciplines, with the advantage of avoiding the anomaly of legal expressions with illegal well-formed subexpressions.

From the viewpoint of type inference, the treatment of **let** in Milner's W is analogous to our algorithm for the conjunctive discipline restricted to rank 2 in §6. The difference is the combinatorial one between W and V. Our algorithm in §5 for the full quantificational discipline is naturally more complex, as it is formulated to accommodate types of larger complexity in recursive calls. When a restriction to typings of rank 2 is guaranteed, the algorithm can be easily simplified to resemble $V_2$.

The fact that the same algorithm, modulo order of unification, is naturally obtained for the different polymorphic disciplines restricted to rank 2, is a further indication of the naturalness of implementing polymorphism at that rank. Type inference is then executable in quadratic time, just as for the parametric discipline. Our treatment of the quantificational discipline in §5 shows that effective type inference is also possible for a less restrictive discipline. We conjecture, however, that in analogy to the breakdown at rank 3 of effective type inference for the conjunctive discipline, type inference for the quantificational discipline restricted to rank 3 is at a substantially higher level in the time/space complexity hierarchy than the quadratic time algorithms we have for all disciplines restricted to rank 2.

**References.**

[Bar] H.P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North Holland, Amsterdam, 1981, xiv+615pp.

[BCD] H. Barendregt, M. Coppo and M. Dezani-Ciancaglini, "A filter lambda model and the completeness of type assignment," to appear in the *Journal of Symbolic Logic.*

[CD] M. Coppo and M. Dezani-Ciancaglini, "A new type assignment for λ-terms," *Archive f. math. Logik u. Grundlagenforschung* **19** (1979) 139-156.

[CDV80] M. Coppo, M. Dezani-Ciancaglini and B. Venneri, "Principal type schemes and lambda-calculus semantics," pp. 535-560 in [SH].

[CDV81] M. Coppo, M. Dezani-Ciancaglini and B. Venneri, "Functional characters of solvable terms." *Zeitschrift fur Mathematische Logik und Grundlagen der Matematik*, **27** (1981) 45-58.

[Cop] M. Coppo, "An extended polymorphic type system for applicative languages," pp. 194-204 in *Mathematical Foundations of Computer Science, Proceedings 1980*, editor P. Dembinski, Springer-Verlag (Lecture Notes in Computer Science **88**), Berlin, 1980, 194-204.

[CF] H.B. Curry and R. Feys, *Combinatory Logic*, North-Holland, Amsterdam, 1958.

[DM] L. Damas and R. Milner, "Principal type-schemes for functional programs," *Conference Records of the Ninth Annual ACM Symposium on Principles of Programming Languages* (1982) 207-212.

[FLO] S. Fortune, D. Leivant, and M. O'Donnell, "The expressiveness of simple and second order type structures," IBM Research Report RC 8542, 1980, 73pp; to appear in the *Journal of the ACM.*

[For] S. Fortune, *Topics in Computational Complexity*, PhD Dissertation, Cornell University, 1979, 125pp.

[Gir] J.-Y. Girard, *Interpretation fonctionelle et elimination des coupures dans l'arithmetique d'ordre superieur*, These de Doctorat d'Etat, 1972, Paris.

[Hin] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Trans. Amer. Math. Society* **146** (1969) 29-60.

[How] W.A. Howard, "The formula as type notion of construction," pp. 479-490 in [SH].

[Lei] D. Leivant, "The complexity of parameter passing in polymorphic procedures," *Thirteenth Annual Symposium on Theory of Computing (SIGACT)*, 1981, 38-45.

[Mar] P. Martin-Lof, "Constructive mathematics and computer programming," *Proceedings of the Sixth (1979) International Congress for Logic, Methodology and Philosophy of Science*, North-Holland, Amsterdam, 1979.

[McC] N.J. McCracken, *An Investigation of a Programming Language with a Polymorphic Type Structure*, PhD Disseration, Syracuse University, 1979, 125pp.

[MS] D.B. MacQueen and R. Sethi, A semantic model of types for applicative languages, *ACM Symposium on LISP adn Functional Programming*, 1982, 243-252.

[Mil] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences* **17** (1978), 348-375.

[O'D] M.J. O'Donnell, "A programming language theorem which is independent of Peano Arithmetic," *Eleventh Annual ACM Symposium on Theory of Computing*, 1979, 176-188.

[Pot] G. Pottinger, "A type assignment for the strongly normalizable λ-terms," pp. 561-577 in [SH].

[Pra] D. Prawitz, *Natural Deduction*, Almqvist and Wiksell, Uppsala, 1965, 113pp.

[Rey] J.C. Reynolds, "Towards a theory of type structures," in *Programming Symposium (Colloque sur la Programmation, Paris)*, Springer (Lecture Notes in Computer Science 19), Berlin, 1974, 408-425.

[Rob] J.A. Robinson, "A machine-oriented logic based on the resolution principle," *Jour. Assoc. Comp. Mach.* **12** (1965) 23-41.

[Sal] P. Salle, "Une extension de la theorie des types," in *Automata, Languages and Programming* (Edited by G. Ausiello and C. Bohm), Springer-Verlag (Lecture Notes in Computer Science 62), 1978, 398-410.

[SH] J.P. Seldin and J.R. Hindley (editors), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, London, 1980, 606pp.*

[Sta] R. Statman, "Number theoretic functions computable by polymorphic programs," in *Twenty Second Annual Symposium on Foundations of Computer Science*, 1981, 279-282.