CMU-CS-85-187

TYPE CHECKING AND TYPE DEDUCTION TECHNIQUES FOR POLYMORPHIC PROGRAMMING LANGUAGES

Paola Giannini
Carnegie-Mellon University

December 1985

Abstract

In this paper we present some of the syntactic issues that arise in polymorphic programming languages. In particular we examine type checking and deduction in two different polymorphic type structures: the parametric lambda-calculus (with *let* construct) and the polymorphic or second-order lambda-calculus. In both approaches the behavior of types is formalized with type inference rules. Examples of programming languages following those approaches are presented and some of their specific problems studied.

The research reported in this paper was supported in part by funds from the Computer Science Department of Carnegie-Mellon University, and by the Defence Advanced Research Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions contained in it are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defence Advanced Research Projects Agency or the US Government.

Introduction

Some constructs of a programming language are meaningful only if the expressions in them yield suitable results. For example $E_1 + E_2$ (where E_1 and E_2 are expressions) does not usually make sense if E_1 is a boolean expression and E_2 is an integer expression. All the values that are suitable for every context in which a given value is suitable are usually said to have a common *type*. The set of types allowed varies considerably from language to language; however, a common characteristic of programming languages is the provision of a set of primitive types (integers, booleans, etc.) with primitive operations, and of mechanisms for constructing more complex types out of primitive or existing ones. Examples of such mechanisms are *records* in PASCAL, *structures* in ALGOL68, *forms* in ALPHARD and so on.

In *strongly typed* (or *statically typed*) languages each expression in a program is assigned a type that can be determined by analyzing the *syntactic* structure of the context in which the expression appears, and programs are well typed with respect to the type restriction of the language. Types provide a basic syntactic verification of correctness of programs, comparable to the checking of dimensional equations in Physics, which insures that the result of a calculation is "dimensionally" correct (even though the numerical result itself may be wrong).

The advantages of strongly typed languages for improving program reliability and programming style are widely recognized; however, the limitations imposed by the type structure are sometimes too restrictive. A frequently quoted example is the impossibility of writing a function in ALGOL68 or PASCAL that, given an array A, of elements of some type, and a total order relation C on such a type, returns A sorted w.r.t. the given order. The declaration of such a function for a fixed type, say integer, and N would look like:

$$fun\ Sort(A\colon array[1..N]of\ Int, C\colon (Int \times Int) \rightarrow Bool)\colon array[1..N]of\ Int...$$

and its type:

$$((array[1..N]of\ Int) \times ((Int \times Int) \rightarrow Bool)) \rightarrow (array[1..N]of\ Int).$$

However, there is no way of using the implicit independence of *Sort* from the type *Int* for writing a "polymorphic" sorting function, i.e., a function that can be applied to arrays regardless of the type of their elements.

On the other hand the type structure of the programming language EL1, [42], which includes disjoint union types and a special type identifier, *ANY*, that stands for any type, is not a viable solution in a strongly typed language. We can certainly write a polymorphic sorting function (just substitute *ANY* for *Int* in the declaration of *Sort*), but then we are not able, when checking an application of *Sort* to some parameters, to enforce the constraint that the type of the elements of the array has to match the type of the parameters of the function C. Moreover it is not clear what

is the type of the elements of the result array. So a program, which is type correct, could produce a run-time error or, if no run-time checking is performed, return incorrect answers.

A simple solution to the problem is provided in the programming languages ML, [15], and HOPE, [3]. The observation there is that type constraints, like the ones a generic sorting function has to satisfy, can be expressed by the use of type variables. A polymorphic sorting function in such languages could be defined by

$$fun\ GenSort(A\text{: }array[1..N]of\ t, C\text{: }(t \times t) \rightarrow Bool)\text{: }array[1..N]of\ t...$$

and its type:

$$((array[1..N]of\ t) \times ((t \times t) \rightarrow Bool)) \rightarrow (array[1..N]of\ t), \tag{1}$$

where t is a type variable. The use of different instances of the same type variable expresses contextual dependences; in this example between the type of the elements of the arrays and the type of C.

The type-checking process can be divided into two (mutually dependent) parts. On one hand, the type checker has to verify that all the occurrences of the parameters, A and C, in the body of *GenSort* are in contexts that do not make any restriction on the type of the elements of the arrays and the function. On the other, for an application of *GenSort* to some actual parameters, the type checker has to match the type operators, ($\rightarrow$, $\times$ and $array[1..N]of$ in *GenSort*) of the actual and formal parameters and instantiate type variables. Whenever an occurrence of a type variable is instantiated, all the other occurrences of the same variable have to be instantiated to the same type. We can think of *GenSort* as simply an abbreviation for the totality of the sorting functions

$$fun\ GenSort(A\text{: }array[1..N]of\ \tau, C\text{: }(\tau \times \tau) \rightarrow Bool)\text{: }array[1..N]of\ \tau...$$

where τ is some type. (The type of the function is obtained by substituting τ for t in (1).) Type checking an application of *GenSort* is essentially chosing one of such functions (if any) that can be correctly applied to the actual parameters. The choice determines the type of the result array. The nice point about this approach is that, once it has been established what type constraints the functions provided by the language have to satisfy, the type checker can deduce most-general (least-constrained) types for user-defined functions. Therefore, types can be omitted and programs are still sure not to produce run-time type errors.

This simple approach to polymorphism, however, does not allow the polymorphic use of parameters. Consider defining a function, f, one of whose parameters is a polymorphic function, say *GenSort*, with type defined as in (1). In each instance of f all the occurrences of *GenSort* have the same type. So, even though the parameter is polymorphic, in the body of f there cannot be applications of *GenSort* to arrays (and functions) of different types.

A different analysis of a language in which polymorphism and strong typing are combined, and polymorphic functions can be specified as parameters, is presented by Reynolds in [37]. Types are

introduced as expressions of the language and can be passed as parameters to functions. Reynolds' language is an extension of the lambda-calculus in which a special type binding operator "Λ" is introduced. "Λ" binds type variables to create functions from types to values, the *polymorphic* functions. Injecting Reynolds' "Λ" into the example above, we may define a truly polymorphic sorting function by

$$fun\ PolSort = \Lambda t.(A{:}\ array[1..N]of\ t, C{:}(t \times t) \rightarrow Bool){:}\ array[1..N]of\ t...).$$

The type of *PolSort* is

$$\forall t.((array[1..N]of\ t) \times ((t \times t) \rightarrow Bool)) \rightarrow (array[1..N]of\ t).$$

$\forall t.\tau$ is the type of a function polymorphic in t (where t is a type variable) which for type-argument t returns values of type τ. A polymorphic function can be applied to a type expression to produce a value. Applying *PolSort* to the type *Int* (the type of integers) we obtain the *Sort* function given previously.

The introduction of polymorphism via type parameters is common to several programming languages. The most notable examples are ALPHARD, [40], RUSSELL, [10], and CLU, [23]. However, these languages do not allow the full generality of Reynolds' language. In particular they identify types with sets of named operations so that a function can be polymorphic in the operations used.

In this paper we analyze the two paradigms of polymorphism previously mentioned. We look at applicative languages and identify extensions of typed lambda-calculus as the core of such languages. We are mainly interested in the syntactic aspects of the language and particularly in the type-checking and, for ML, type-deduction techniques.

Section 1 surveys ML and HOPE. We give the type-deduction and type-checking algorithms for those languages and look at the specific problems that arise in such contexts. At the end of the section we mention some extensions of the basic type system that might be introduced in such languages.

In Section 2 we present Reynolds' polymorphic lambda-calculus and its associated type system that includes type quantification. Type checking is quite a simple problem for this language. However, almost nothing is known about type deduction. We give some general results about the language and its type structure. Next the RUSSELL type system and its type-checking algorithm are explained. The choice of RUSSELL is motivated by its clean treatment of user-defined types, which was also the main motivation for Reynolds' polymorphic lambda-calculus. Some of the differences between the two type systems will be discussed.

The main tool used in type checking and deduction for ML and HOPE is unification. In an Appendix we present some unification algorithms and discuss their complexity.

A style of polymorphism which is not surveyed in this paper is the conjunctive discipline of [1], [5], and [6]). This discipline is theoretically very interesting as it allows one to characterize various classes of lambda-calculus terms. However, it is impractical from a programming language point of view as its type checking is unsolvable.

1. Type Checking and Type Deduction in Parametric Lambda-Calculus

In this section we present the results underlying ML type deduction and HOPE type checking. Both languages can be abstractly modelled using the parametrically typed lambda-calculus: a typed lambda-calculus whose syntax is augmented with type variables. Since types can be omitted from ML expressions, it may seem strange to consider typed lambda-calculus as the abstract model underlying ML. It is, however, the case that only expressions that can be successfully filled with types as to otain (parametrically) typed lambda-calculus expressions are ML type-correct expressions. Therefore, we can regard ML as a typed language, which type structure allows automatic recovering of the (possibly) missing type information.

The syntactic structure of types, and its relation to the expressions of the language, provide the main tool for defining an algorithm for polymorphic type checking and for deducing types for untyped expressions.

The first subsection introduces the language and a general treatment of the topics above. It starts by considering a simply typed lambda-calculus and defining what it means for expressions to be well typed in such a language. The only constraint is in applications, $e(e')$, where the expression e must have a functional type and its input type has to be equal to the type of e'. The basic idea in type deduction is that types do not have to be specified in expressions in order to verify that this constraint is satisfied. However, in general more than one type can be deduced for an expression. (Consider the identity function $\lambda x.x$). To express the type of such expressions type variables are introduced. The type assigned to $\lambda x.x$, $t \rightarrow t$ (where t is a type variable), is a schema from which it can be derived that $\lambda x.x$ has type $\tau \rightarrow \tau$ for every type τ. Type deduction is introduced by a set of natural deduction rules, and an algorithm, W, is shown to be sound and complete for such deduction rules.

In HOPE the situation is slightly different as types cannot be omitted from expressions. By allowing type variables, however, a similar effect can be achieved. Indeed the polymorphic type checker that we will present is similar to the type-deduction algorithm W.

The second subsection analyzes ML and HOPE in greater detail and discusses the specific constructs of those languages. For ML the *let* construct is introduced and type deduction extended to allow typing of *let* expressions. The algorithm W, and the proof of its soundness and completenes are generalized to the new deduction system.

HOPE's characteristic function definition, which relies on a very syntactic semantics of the data types, is then presented. We study the type-checking problem that arises from this style of definition, and present a result that establishes the decidability of the well-definedness predicate.

At the end of the section some extensions to the basic type structure relevant to programming languages are mentioned.

Types

We start by defining the types of the language and analyzing their syntactic structure. Let κ range in K, where K is a set of constant types, and t range in V, where V is a set of type variables. (The constant types Integers, *Int*, and Booleans, *Bool*, will be used in the examples.) The types $\tau \in T$ of the language are defined by

$$\tau ::= \kappa \mid t \mid \tau \times \tau' \mid \tau \rightarrow \tau'.$$

$V(\tau)$ is the set of type variables contained in τ. Types not containing type variables are called *monotypes*. In order to avoid confusion, we denote these by $\bar{\tau}$, with subscripts and superscripts if necessary.

A *substitution* S is a function from V to T which is the identity function almost everywhere. S can be extended to a function from T to T in the obvious way. For substitutions S and S', SS' denotes the composition of the two functions. Given the associativity of composition, in $SS'S''$, parentheses can be omitted ($(SS')S'' = S(S'S'')$). We use the following notations:

- $S[t_1 \mapsto \tau_1, ..., t_n \mapsto \tau_n]$ for the substitution equal to S on all the type variables except that the t_i's , for $1 \leq i \leq n$, are mapped to τ_i (the t_i's are distinct variables).
- $S =_R S'$, where $R \subseteq V$, for: $St = S't$ for all $t \in R$.

An *instance* of a type τ is a type τ' such that $\tau' = S\tau$ for some substitution S.

Polytypes are types containing type variables; intuitively they stand for the set of types that can be obtained by substituting monotypes for their type variables. This syntactic interpretation of polytypes induces a preorder on the set T:

$$\tau \leq \tau' \text{ if and only if } \mathit{MonoInst}(\tau') \subseteq \mathit{MonoInst}(\tau), \tag{2}$$

where $\mathit{MonoInst}(\tau)$ is the set of monotypes that are instances of τ. Note that, for monotypes $\bar{\tau}$ and $\bar{\tau}'$, $\bar{\tau} \leq \bar{\tau}'$ if and only if $\bar{\tau} = \bar{\tau}'$. If $\tau \leq \tau'$ we say that τ is *more general* than τ'. The preorder can be made a partial order by introducing an equivalence relation (denoted by $\cong$) between τ and τ' whenever $\tau \leq \tau'$ and $\tau' \leq \tau$. The set of all variables V is the minimum element of $T_{\cong}$.

The preorder in (2) is equivalent to:

$$\tau \leq \tau' \text{ if and only if } \tau' = S\tau \text{ for some substitution } S. \tag{3}$$

It is obvious that (3) implies (2). The other implication can be proved by structural induction on τ' and case analysis on τ. The only difficult case is when, for some τ'_1 and τ'_2, τ' is equal to $\tau'_1 \rightarrow \tau'_2$ (or $\tau'_1 \times \tau'_2$), and $\tau \notin V$. By (2), for some τ_1 and τ_2, $\tau = \tau_1 \rightarrow \tau_2$. Applying the inductive hypothesis to τ'_1 and τ'_2 we derive that $\tau'_1 = S_1\tau_1$ and $\tau'_2 = S_2\tau_2$, for some substitutions S_1 and S_2. To prove the result we have to show that $S_1 =_{V(\tau_1)} {}_{V(\tau_2)} S_2$. (Therefore S_1 and S_2 can be combined to get

the desired substitution.) Let $t \in V(\tau_1) \cap V(\tau_2)$. By (2), $\bar{S}S_1t = \bar{S}S_2t$ for all $\bar{S}$ such that $\bar{S}S_1t$ and $\bar{S}S_2t$ are monotypes. This fact can be proved to imply that $S_1t = S_2t$.

Given the definition of $\leq$ of (3) we derive that $\tau \cong \tau'$ if and only if τ is equal to τ' up to renaming of variables.

A *unifier* for τ and τ' is a substitution U such that $U\tau = U\tau'$. The important property of the partial order on $T_{\cong}$ is that if τ and τ' are compatible (that is, there is τ'' such that $\tau \leq \tau''$ and $\tau' \leq \tau''$) then τ and τ' have a least upper bound w.r.t. this order. Moreover, to a least upper bound of two compatible types there corresponds a unifier U such that: for any unifier U' of τ and τ' there is a substitution S and $U' = SU$. Such an unifier is called a *most-general unifier*. (See the Appendix for details about some unification algorithms and a discussion of their complexity.)

Remark. The set of types T is a special case of the class of first-order term languages. The existence of a most-general unifier (if any) for such class of languages was first shown in Herbrand's thesis. It was, however, explicitly stated as the Unification Lemma in Robinson [39]. Huet in [20] chapter 5 gives a detailed presentation of the properties of the partial order on $T_{\cong}$ and its relation to the unification problem.

Expressions

Next we define the simply typed lambda-expressions. The expressions of the language $E_{\bar{T}}$ are typed lambda-calculus terms $e \in E_{\bar{T}}$ defined by

$$e ::= \lambda x : \bar{\tau}.e \mid e(e') \mid x \mid c_{\bar{\tau}}$$

where $\bar{\tau}$ is a monotype, $x \in X$ a set of variables, and $c_{\bar{\tau}} \in C$ a set of typed constants. (The subscript $\bar{\tau}$ of c indicates its type). We assume that the set C contains, for all types $\bar{\tau}$ and $\bar{\tau}'$, the following typed selectors and constructors for products:

$$\begin{array}{l} fst_{(\bar{\tau}\times\bar{\tau}')\to\bar{\tau}} \\ snd_{(\bar{\tau}\times\bar{\tau}')\to\bar{\tau}'} \\ \langle\cdot,\cdot\rangle_{\bar{\tau}\to\bar{\tau}'\to(\bar{\tau}\times\bar{\tau}')} \end{array}$$

In the following examples, type subscripts are omitted and $\langle e, e'\rangle$ stands for $\langle\cdot,\cdot\rangle(e)(e')$. (However, in the definition of well typed expressions we assume they are typed.) The axioms that the previous constants satisfy are:

$$\begin{aligned} fst(\langle e, e'\rangle) &= e \\ snd(\langle e, e'\rangle) &= e' \\ \langle fst(e), snd(e)\rangle &= e. \end{aligned}$$

Examples. We use the notations: *let* $x : \tau = e$ *in* e' for $(\lambda x : \tau.e')(e)$, and *lettype* $I = \tau$ *in* e for the syntactic substitution of τ for I in e.

The *let* notation should not be confused with the ML *let* construct, which will be discussed later. *Lettype* allows us to denote a type by an identifier I, and is only for convenience in the example.

1. The product of two rational numbers, represented as pairs of integers, is computed by the following expression.

$$\begin{array}{l} \mathit{lettype}\ \mathit{rat} = \mathit{Int} \times \mathit{Int}\ \mathit{in} \\ \quad \lambda x : \mathit{rat}.\lambda y : \mathit{rat}.\langle \mathit{fst}(x) \cdot \mathit{fst}(y), \mathit{snd}(x) \cdot \mathit{snd}(y) \rangle \end{array}$$

2. The following expression is equivalent to the identity function on rationals (represented as pairs of integers).

$$\begin{array}{l} \mathit{lettype}\ \mathit{rat} = \mathit{Int} \times \mathit{Int}\ \mathit{in} \\ \quad \mathit{let}\ \mathit{swap} : \mathit{rat} \to \mathit{rat} = \lambda x : \mathit{rat}.\langle \mathit{snd}(x), \mathit{fst}(x) \rangle\ \mathit{in} \\ \quad\quad \lambda y : \mathit{rat}.\mathit{swap}(\mathit{swap}(y)) \end{array}$$

Well-Typed Expressions and Type Checking

The syntax of $E_{\bar{T}}$ allows expressions that do not respect the intuitive type restrictions imposed by the use of types. For example, $\mathit{fst}(n)$, where n is a constant of type *Int*, and *fst* is the selector of the first component of a pair, does not always make sense. To express formally such type constraints we introduce a definition of *meaningful* or *well-typed* expressions. In this functional language the only limitation imposed on the expressions is the matching of the types of the actual and the formal parameters in a function application. Such matching, in the simple case of monotypes, means that the two types are equal. (We think of constant types as denoting disjoint sets of values.)

A *context* A on $\bar{T}$ is a partial function (with finite domain) from variables X to types in $\bar{T}$. A can be represented as a finite set of pairs $\{(x_i, \bar{\tau}_i) \mid 0 \leq i \leq n\}$, where $x_i \neq x_j$ for $i \neq j$. $\emptyset$ denotes the empty context, and A_x the context $\{(x', \bar{\tau}) \in A \mid x \neq x'\}$.

Definition 1: Let A be a context on $\bar{T}$, e an expression in $E_{\bar{T}}$, and $\bar{\tau}$ a monotype. The relation $A \supset e : \bar{\tau}$ (*e has type* $\bar{\tau}$ *in* A) is the smallest relation satisfying:

1. $A \supset c_{\bar{\tau}} : \bar{\tau}$ for all $c_{\bar{\tau}}$;
2. if $A(x) = \bar{\tau}$, then $A \supset x : \bar{\tau}$;
3. if $A_x \cup \{(x, \bar{\tau})\} \supset e : \bar{\tau}'$, then $A \supset (\lambda x : \bar{\tau}.e) : \bar{\tau} \to \bar{\tau}'$;
4. if $A \supset e : \bar{\tau} \to \bar{\tau}'$ and $A \supset e' : \bar{\tau}$, then $A \supset e(e') : \bar{\tau}'$.

We say that e is *well typed* w.r.t. A if there is a $\bar{\tau}$ such that $A \supset e : \bar{\tau}$. □

By structural induction on e we can prove that:

1. for any A, if $A \supset e : \bar{\tau}$ and $A \supset e : \bar{\tau}'$ then $\bar{\tau} = \bar{\tau}'$ (the same expression cannot have more than one type for any given context);
2. the type of a closed expression does not depend on the context;
3. if an expression e is well typed w.r.t. A, then every subexpression of e is well typed w.r.t. some context A'.

Let $e \in E_T$ and let A be a context defined for all the free variables of e. The definition of the relation gives a linear algorithm for finding (if any) a type $\bar{\tau}$ such that $A \supset e : \bar{\tau}$.

Type Deduction

Strong typing is a sound principle that eliminates the run-time cost of type checking (if the language is typed) and permits some degree of mechanical verification of the program. However, for a functional language, like the one considered in this section, the required typing information is frequently redundant. Consider the second example given at page 1. The types explicitly assigned to x and y can be derived by the type assigned to *swap*.

Having a definition of the well-typed expressions of the language, we can ask how much type information is needed in order verify that, indeed, expressions are well typed. It turns out that the relation between types and expressions in this language (and in many extensions of both the language and the type system) is such that we do not need any type information (types of lambda variables) at all.

To formally introduce type deduction we define the language E obtained by $E_{\bar{\tau}}$ erasing types from lambda-abstractions, i.e., $e \in E$ is defined by

$$e ::= \lambda x.e \mid e(e') \mid x \mid c_\tau$$

In c_τ, τ is a *most-general* or *principal* type for c, i.e., for all $c_{\tau'} \in C$, τ' is an instance of τ. The choice of the principal type for constants, as we will see from the deduction rules, is irrelevant. The constructors and selectors for product could be defined as follows.

$$\begin{array}{l} fst_{t\times t'\to t} \\ snd_{t\times t'\to t'} \\ \langle\cdot,\cdot\rangle_{t\to t'\to(t\times t')} \end{array}$$

Expressions of $E_{\bar{\tau}}$ are mapped in expression of E by the function *bare* defined as follows:

$$\begin{aligned} bare(\lambda x:\bar{\tau}.e) &= \lambda x.bare(e) \\ bare(e(e')) &= bare(e)(bare(e')) \\ bare(x) &= x \\ bare(c_{\bar{\tau}}) &= c_\tau \end{aligned}$$

where τ, in the last clause, is a principal type for c. For instance (omitting the principal types for the product functions) applying *bare* to the second example page 6 we get the following expression.

$$let\ swap = \lambda x.\langle snd(x), fst(x)\rangle\ in\ \lambda y.swap(swap(y)) \tag{4}$$

The type constraints that expressions of E have to satisfy are expressed by inference rules.

Definition 2: Let A be a context on T and $e \in E$,

$$A \vdash e : \tau$$

if $e : \tau$ can be derived from A using the following rules:

$$TAUT: \quad A_x \cup \{(x,\tau)\} \vdash x : \tau$$

$$\rightarrow IN: \quad \frac{A_x \cup \{(x,\tau)\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \rightarrow \tau'}$$

$$\rightarrow EL: \quad \frac{A \vdash e : \tau \rightarrow \tau' \quad A \vdash e' : \tau}{A \vdash e(e') : \tau'}$$

$$CONST: \quad A \vdash c_\tau : \tau' \quad \text{for } \tau' \leq \tau \quad \square$$

The type deduced for (4) can be easily seen to be:

$$(t \times t) \rightarrow (t \times t)$$

which is more general than the type of the original expression.

Let A be a context on T.

- $V(A) = \bigcup_{(x,\tau)\in A} V(\tau)$ is the set of type variables of A, and
- $SA = \{(x, S\tau) \mid (x,\tau) \in A\}$, the result of the application of the substitution S to A.

The relation between type checking of expressions in $E_{\bar{T}}$ and deduction of types for expressions in E is expressed by the following proposition.

Proposition 3: 1. Let $\bar{A}$ be contexts on $\bar{T}$, $\bar{e} \in E_{\bar{T}}$, and $\bar{\tau} \in \bar{T}$. If $\bar{A} \supset \bar{e} : \bar{\tau}$, then, for some $\tau \in T$, $\bar{A} \vdash bare(\bar{e}) : \tau$ and $\bar{\tau} \leq \tau$.

2. Let A be a context on T, $e \in E$, and $\tau \in T$. If $A \vdash e : \tau$, then, for some substitution S and $\bar{e} \in \bar{E}$ such that $bare(\bar{e}) = e$, $SA \supset \bar{e} : S\tau$. (All the types in SA and $S\tau$ are monotypes.)

Proof. The proposition can be easily derived by structural induction on expressions. $\square$

In a derivation of $e : \tau$ from A there is a one-to-one correspondence between the structure of the expression e and the structure of the deduction. Therefore for each occurrence of a subexpression e' of e there is exactly one $e' : \tau'$ for some τ' that is the conclusion of one of the rules. Therefore, we can easily prove that: if

$$A \vdash (\lambda x.e)(e') : \tau$$

then

$$A \vdash e[e'/x] : \tau.$$

This is the subject-reduction theorem (see [8] p. 313) for lambda-calculus that asserts that types are preserved by β-reductions.

The type-deduction system does not provide an easy way to compute the type of an expression; in fact it does not even ensure that the problem is decidable. To show that the problem of deducing types for expressions in E is solvable we prove the following theorem.

Theorem 4: Let A be a context on T and $e \in E$. There is an algorithm, W, which takes as input e and A and

1. either fails, or returns a substitution S and a type τ such that

$$SA \vdash e : \tau.$$

This asserts the *soundness* of W.

2. Moreover,

$$SA \vdash e : \tau, \text{ for some } \tau \in T \text{ and } S, \text{ implies } W(A, e) = (S', \tau'),$$

and, for some $\bar{S}$, $S =_{V(A)} \bar{S}S'$ and $\tau = \bar{S}\tau'$. This means that W is *complete* for $\vdash$.

Proof. Let W be defined by

$W(A, e)$
case e **of**

$\lambda x.e'$	**then**	$(S, \tau) := W(A_x \cup \{(x, t)\}, e')$	(t is a new variable)
		$return(S, St \to \tau)$	
$e_1(e_2)$	**then**	$(S_1, \tau_1) := W(A, e_1)$;	
		$(S_2, \tau_2) := W(S_1 A, e_2)$;	
		$U = Unify(S_2\tau_1, \tau_2 \to t)$;	(t is a new variable)
		$return(US_2S_1, Ut)$	
x	**then**	$return(I, A(x))$	(I is the identity substitution)
c_τ	**then**	$return(I, \tau[t'_1/t_1, ..., t'_n/t_n])$	($\{t_1, ..., t_n\} = V(\tau)$, t'_i's are new variables)

The function $Unify(\tau, \tau')$ either fails (if τ and τ' are not compatible) or returns a most-general unifier for τ and τ'. If *Unify* fails then W fails.

1. The proof of *soundness* is by structural induction on e. The base cases x and c are obvious.

For abstraction: if $W(A, \lambda x.e')$ succeeds, then $W(A_x \cup \{(x, t)\}, e') = (S, \tau)$ for some S and τ. By inductive hypothesis,

$$SA_x \cup \{(x, St)\} = S(A_x \cup \{(x, t)\}) \vdash e' : \tau.$$

For an application $e_1(e_2)$ assume $W(A, e_1(e_2))$ succeeds. Then $W(A, e_1) = (S_1, \tau_1)$ and $W(S_1 A, e_2) = (S_2, \tau_2)$ for some S_1, S_2, τ_1 and τ_2 (both calls succeed). By inductive hypothesis,

$$S_1 A \vdash e_1 : \tau_1 \tag{5}$$

and

$$S_2 S_1 A \vdash e_2 : \tau_2. \tag{6}$$

Applying the substitution S_2 to all the types in the derivation of (5) we get:

$$S_2 S_1 A \vdash e_1 : S_2 \tau_1. \tag{7}$$

Moreover, since $W(A, e_1(e_2))$ succeeds we know that $US_2\tau_1 = US_2(\tau_2 \to t) = (U\tau_2) \to (Ut)$. Applying the substitution U to both (7) and (6), we get:

$$US_2S_1A \vdash e_1 : (U\tau_2) \to (Ut) \quad \text{and} \quad US_2S_1A \vdash e_2 : U\tau_2,$$

which is the premises of the rule $\to EL$. Therefore

$$US_2S_1A \vdash e_1(e_2) : Ut.$$

This completes the proof of soundness.

2. Let N be the set of new variables generated by $W(A, e)$. To prove *completeness*, we prove 2 where $S =_{V-N} \bar{S}S'$ replaces $S =_{V(A)} \bar{S}S'$. As $N \cap V(A) = \emptyset$, this statement implies completeness. The stronger result is needed in the inductive step. The proof is again by structural induction on e.

For variables is immediate. (Just take $\bar{S} = S$.)

Let e be c_τ. $SA \vdash c_\tau : \tau'$ where $\tau' = S'\tau$ for some S'. Let $\bar{S}$ be $S[t'_1 \mapsto S't_1, ..., t'_n \mapsto S't_n]$, where the t'_i's are the new variables in $W(A, c_\tau)$. $\bar{S}$ verifies the required conditions.

Let e be $\lambda x.e'$ and assume that $SA \vdash e : \tau$. From $\to IN$,

$$SA_x \cup \{(x, \tau_1)\} \vdash e' : \tau_2$$

for some τ_1 and τ_2 such that $\tau = \tau_1 \to \tau_2$. Let S_1 be $S[t \mapsto \tau_1]$, where t is the new variable chosen in $W(A, e)$ $(t \notin V(A))$,

$$S_1(A_x \cup \{(x, t)\}) \vdash e' : \tau_2.$$

From the inductive hypothesis, $W(A_x \cup \{(x, t)\}, e') = (S', \tau')$, and, for some S'', $\tau_2 = S''\tau'$, and

$$S_1 =_{V-N} S''S'. \tag{8}$$

N is the set of new variables generated in $W(A_x \cup \{(x, t)\}, e')$, and $t \notin N$. Hence, from the definition of W,

$$W(A, \lambda x.e') = (S', S't \to \tau').$$

To complete the proof we have to show that $S =_{V-(N\cup\{t\})} \bar{S}S'$ and $\tau = \bar{S}(S't \to \tau')$, for some $\bar{S}$. Let $\bar{S}$ be S'', the first equality derives from the fact that S and S_1 are equal on all the type variables but t and (8). Again by (8), $\bar{S}S't = S_1t = \tau_1$. Hence $\bar{S}(S't \to \tau')$ is equal to $\tau_1 \to \tau_2$.

Let e be $e_1(e_2)$ and assume that $SA \vdash e : \tau$. From $\rightarrow EL$, for some τ',

$$SA \vdash e_1 : \tau' \rightarrow \tau$$

and

$$SA \vdash e_2 : \tau'.$$

By the inductive hypothesis applied to e_1, $W(A, e_1) = (S_1, \tau_1)$, and, for some $\bar{S}_1$, $S =_{V-N_1} \bar{S}_1 S_1$ and $\tau' \rightarrow \tau = \bar{S}_1 \tau_1$. ($N_1$ are the new variables of $W(A, e_1)$). Since $V(A) \cap N_1 = \emptyset$,

$$\bar{S}_1 S_1 A \vdash e_2 : \tau'.$$

By inductive hypothesis applied to e_2, $W(S_1 A, e_2) = (S_2, \tau_2)$, and $\bar{S}_1 =_{V-N_2} \bar{S}_2 S_2$ and $\tau' = \bar{S}_2 \tau_2$, for some $\bar{S}_2$ (N_2 are the new variables of $W(S_1 A, e_2)$, hence $N_1 \cap N_2 = \emptyset$). Let S' be $\bar{S}_2[t \mapsto \tau]$, where t is the new variable chosen in the algorithm:

- $t \notin V(\tau_2)$ implies $S'(\tau_2 \rightarrow t) = \tau' \rightarrow \tau$, and
- $t \notin \tau_1$ and $V(\tau_1) \cap N_2 = \emptyset$ imply $S' S_2 \tau_1 = \bar{S}_2 S_2 \tau_1 = \bar{S}_1 \tau_1 = \tau' \rightarrow \tau$.

Hence $Unify(\tau_2 \rightarrow t, S_2 \tau_1)$ succeeds and returns a U such that, for some S'', $S' = S''U$. The result of $W(A, e)$ is then $(US_2 S_1, Ut)$. Let $N = N_1 \cup N_2 \cup \{t\}$. From the previous statements, it is easy to prove that $S =_{V-N} S''US_2 S_1$ and $\tau = S''Ut$. □

Remark. The original proof of 4.2 is due to Hindley, [17] theorem 1. Hindley's proof, formulated for combinatory logic, does not mention explicitly an algorithm for deducing types. However, an algorithm similar to W can be extracted from the proof. A more complex version of the algorithm W, which will be presented in the following subsection as W_L, is due to Damas and Milner, see [30] and [9]. *Soundness* of this algorithm is proved in [30] and the statement of *completeness* is presented in [9]. In [4], the author describes a relatively efficient implementation of the extended algorithm.

Polymorphic Type Checking

In HOPE, [3], function declarations have to mention explicitly the type of parameters and result. The types that can be specified may contain type variables, which specify contextual dependences between types. Type checking has to use the constraints given by the explicit typing, and the ones given by the syntactic structure of the expressions.

The language can be abstractly defined as follows. The set of types of the language is T. The expressions $e \in E_H$ are defined by

$$e ::= \lambda x : \tau . e_{\tau'} \mid e(e') \mid x \mid c_\tau$$

The expression $\lambda x : \tau.e_{\tau'}$ is the normal function definition of programming languages in which both the type of the parameter x, τ, and the type of the result, τ', are specified. Expressions with (without) occurrences of type variables in the type of their subexpressions $\lambda x : \tau.e_{\tau'}$ and c_τ are called polymorphic (monomorphic) expressions.

The definition of *well typed* for monomorphic expressions of E_H is obtained by replacing condition 3 of definition 1 by

3'. if $A_x \cup \{(x, \bar{\tau})\} \supset e : \bar{\tau}'$, then $A \supset (\lambda x : \bar{\tau}.e_{\bar{\tau}'}) : \bar{\tau} \rightarrow \bar{\tau}'$.

The application of a substitution S to an expression $e \in E_H$, Se, is the expression resulting from applying S to all the types of e, i.e.,

$$Se = \begin{cases} Se'(Se'') & \text{if } e = e'(e'') \\ \lambda x : S\tau.Se'_{S\tau'} & \text{if } e = \lambda x : \tau.e'_{\tau'} \\ c_{S\tau} & \text{if } e = c_\tau \\ x & \text{if } e = x \end{cases}$$

We extend the definition of well typed for monomorphic expressions to polymorphic expressions as follows.

Definition 5: Given A and e we say that e is well typed w.r.t. A if, for some S such that SA is a context on $\bar{T}$ and Se is a monomorphic expression, Se is well typed w.r.t. SA. So a polymorphic expression is well typed if we can make sense of some of its monomorphic instances. □

Let e be the expression

$$\lambda y : t \rightarrow t.\underbrace{(\lambda x : Int \times t.(y(snd(x)))_t(\langle 3, 4\rangle))_{t'}}_{e_1}(e_2)$$

where extra parentheses are added for readability. Consider first the subexpression e_1. Let A be $\{(y, t \rightarrow t)\}$. The substitution $S = I[t \mapsto Int, t' \mapsto Int]$ is such that Se_1 is well typed w.r.t. SA and $SA \supset Se_1 : Int$. Moreover, any substitution S' such that $S'e_1$ is well typed w.r.t. $S'A$, has to be such that $S't = S't' = Int$. Assume e_2 is a closed expression, for e to be well typed, there has to be a substitution S'' such that $S''t = S''t' = Int$, and $\emptyset \supset S''e_2 : Int$.

The types specified in an expression (e.g., $y : t \rightarrow t$ in e) impose additional constraints on the type of the expression and type checking verifies that there is an interpretation (assignment of types to type variables) that satisfies them and is consistent with the structure of the expression.

A simple modification of Milner's W algorithm provides a polymorphic type checker for this language. (I believe that this is similar to the type checker for HOPE, but unfortunately [3] does not say anything about it.)

The input to the algorithm is a context A on T and an expression $e \in E_H$. The algorithm, which we call W_H, either fails (if *Unify* fails), or returns a substitution S and a type τ whose properties are given by the following Proposition.

Proposition 6: Let A be a context on T and let $e \in E_H$.

1. If $W_H(A, e) = (S, \tau)$ then e is well typed w.r.t. A and for all S' such that $S'Se$ is a monomorphic expression ($S'\tau$ is a monotype) and $S'SA$ is a context on $\bar{T}$

$$S'SA \supset S'Se : S'\tau.$$

2. Let $V(e)$ be the set of type variables of e. If, for some S', $S'A \supset S'e : \tau'$, then, for some S and τ, $W_H(A, e) = (S, \tau)$. Moreover, for some $\bar{S}$, $S' =_{V(A) \cup V(e)} \bar{S}S$ and $\tau' = \bar{S}S\tau$. □

The algorithm W_H is defined as follows.

$W_H(A, e)$
case e **of**

$\lambda x : \tau.e'_{\tau'}$	**then**	$(S_1, \tau_1) := W_H(A_x \cup \{(x, \tau)\}, e')$;
		$U := Unify(S_1\tau', \tau_1)$;
		$return(US, (US\tau) \rightarrow (U\tau_1))$
$e_1(e_2)$	**then**	$(S_1, \tau_1) := W_H(A, e_1)$;
		$(S_2, \tau_2) := W_H(S_1A, Se_2)$;
		$U = Unify(S_2\tau_1, \tau_2 \rightarrow t)$; ($t$ is a new variable)
		$return(US_2S_1, Ut)$
c_τ	**then**	$return(I, \tau)$ (I is the identity substitution)
x	**then**	$return(I, A(x))$

Proposition 6 asserts *soundness* and *completeness* of W_H w.r.t. the definition of well typed given in 5. The proposition can be proved by structural induction on e following the lines of the proof of theorem 4.

1.1. ML and HOPE

The ML Constructs *let* and *letrec*

The only binding mechanism of the typed lambda-calculus, introduced in the previous sections, is the lambda-binding $\lambda x.e$. By the correspondence between the structure of expressions and the structure of a derivation all the instances of x in a derivation have the same type (possibly a polytype). So if we were to consider a definition $f = e$ *in* e' in a program, as usual, as a short-hand for:

$$(\lambda f.e')(e) \tag{9}$$

even though e (and then f) can be assigned a polytype, all the instances of f in e' must have the same type. Consider the example of $f = \lambda x.x$ and $e' = \langle f(n), f(true)\rangle$ where n is a constant of type *Int* and *true* is a constant of type *Bool*. The type assigned to f is $t \rightarrow t$ (where t is arbitrary),

however. the expression $\lambda f.(f(n), f(true))$ (that is $\lambda f.e'$) is not well typed and so the application is not well typed. The point is that, as remarked in Milner, [30], there are applications of $\lambda f.e'$ to functions (for instance the identity on integers) that are not well typed. When the definition of f is given, however, the information about its typing can be used to ensure that the application is correct.

In ML, to allow the use of polymorphic expressions the construct:

$$let\ f = e\ in\ e' \tag{10}$$

is introduced. (10) is an abbreviation for:

$$e'[e/f]$$

that by the β-axiom of lambda-calculus, is equivalent to (9). However, since the expression e replaces f in e', different instances of its type may be used in deriving a type for e'. Therefore, the fact that (10) is typable, whereas (9) is not, should not be a surprise. Types reflect structural properties of the expressions and such properties change in reductions[1]. For instance if f is not free in e' (10) can be typable even when e is not.

Let E_L be the language defined by

$$e ::= \lambda x.e \mid e(e') \mid x \mid c_\tau \mid let\ x = e\ in\ e'$$

A rule for deducing types for expressions in E_L is presented in [9]. The idea is to introduce a new set of types T_L defined by:

$$\sigma := \forall t.\sigma \mid \tau \tag{11}$$

where $\tau \in T$. Let A be a context on T_L. Let $V(A)$ be the set of free variables of A. In the following we assume that the bound variables of A are all distinct and disjoint from $V(A)$. (If such conditions are not satisfied we can rename bound variables). The rules for deducing types for *let* expressions are the following:

$$\forall IN: \quad \frac{A \vdash e : \sigma}{A \vdash e : \forall t.\sigma} \qquad (t \notin V(A))$$

$$\forall EL: \quad \frac{A \vdash e : \forall t.\sigma}{A \vdash e : \sigma[\tau/t]} \qquad (\tau \text{ free for } t \text{ in } \sigma)$$

$$LET: \quad \frac{A \vdash e : \sigma \quad A_x \cup \{(x,\sigma)\} \vdash e' : \tau}{A \vdash let\ x = e\ in\ e' : \tau}$$

[1] It is a well known fact, see [8] that Curry functionality theory (ML type deduction system is modelled on this) is not closed under β-conversion.

(The definition of τ free for t in σ is the usual one for predicate calculus.) Note that in $\rightarrow IN$ the assumption on x, (x, τ), is not quantified. Therefore, also for this extension of language, in a deduction of a type for $\lambda x.e$, none of the variables free in τ changes. (The variables cannot be quantified and then instantiated.)

Definition 7: Let T_L be the set of types defined in (11), A a context on T_L, and $e \in E_L$. We say that

$$A \vdash_L e : \sigma$$

if there is a deduction of $e : \sigma$ from assumptions A using the rules: $TAUT$, $CONST$, $\rightarrow IN$, $\rightarrow EL$, $\forall IN$, $\forall EL$ and LET. □

For $\sigma = \forall t_1 ... t_n.\tau$ define $\sigma > \sigma'$ if $\sigma' = \forall t'_1 ... t'_m.(\tau[\tau_1/t_1, ..., \tau_n/t_n])$ where the t'_i's are not free in σ. Let S be a substitution $S\sigma$ is the result of the substitution S applied to the free variables of σ. In such an application quantified variables may be renamed to avoid clashes with variables of S. (Therefore $\sigma > \sigma'$ implies $S\sigma > S\sigma'$.)

Let A be a context on T_L and let $\tau \in T$. $\forall_A.\tau$ stands for $\forall t_1 ... t_n.\tau$ where t_i are the free variables of τ which are not free in A.

Fact 8: The following facts are used in the proof of theorem 9.

1. If $\sigma > \sigma'$ and $\sigma' > \sigma''$, then $\sigma > \sigma''$.
2. If $\sigma > \tau$ and the variables t_1,...,t_n are not free in σ, then $\sigma > \forall t_1 ... t_n.\tau$.
3. $\{(x, \sigma)\} \vdash_L x : \sigma'$, for some σ and σ', implies $\sigma > \sigma'$. (By standard normalization techniques of proof theory, see [36], any proof of $\{(x, \sigma)\} \vdash_L x : \sigma'$ can be transformed in a proof of the same statement in which all the applications of the $\forall EL$ rule precede the applications of $\forall IN$.)
4. If a proof of $A \vdash_L e : \tau$ is followed by any number of applications of $\forall IN$ and $\forall EL$ rules in any order, then the result is a proof for $A \vdash_L e : \sigma$ where $\forall_A.\tau > \sigma$. (By a similar normalization technique.) □

The type deduction algorithm, W_L, that follows is an extension of W. (This is actually the original W presented in [9].)

$W_L(A, e)$
case e **of**
$\lambda x.e'$, $e_1(e_2)$ and c_τ as for W (with calls to W replaced by calls to W_L)
x **then** $return(I, \tau[t'_1/t_1, ..., t'_n/t_n])$
where $A(x) = \forall t_1 ... t_n.\tau$ and the t'_i's are new variables
$let\ x = e_1\ in\ e_2$ **then** $(S_1, \tau_1) := W_L(A, e_1)$;
$(S_2, \tau_2) := W_L(S_1 A_x \cup \{(x, \forall_{S_1 A}.\tau_1)\}, e_2)$;
$return(S_2 S_1, \tau_2)$

The clause for x combined with the one for *let* gives the desired effect of allowing multiple instantiations of the type of x in e'. For instance, let e be *let* $f = \lambda x.x$ *in* $f(f)$. $W_L(\emptyset, e)$ first calls $W_L(\emptyset, \lambda x.x)$, which result is $(I, t \to t)$, and then $W_L(A, f(f))$, where A is $\{(f, \forall t.t \to t)\}$. The two successive calls of $W_L(A, f)$, since f has a quantified type, result in $(I, t' \to t')$ and $(I, t'' \to t'')$ respectively. Therefore, unification succeeds and $W_L(\emptyset, e) = t'' \to t''$. Note that, $W_L(\emptyset, \lambda f.f(f))$ fails. Hence W_L applied to the expression $(\lambda f.f(f))(\lambda x.x)$, which is equivalent to e also fails.

Soundness and completeness of W_L w.r.t. $\vdash_L$ are stated by the following theorem.

Theorem 9: Let A be a context on T_L and $e \in E_L$.

1. W_L applied to A and e either fails, or returns a substitution S and a type $\tau \in T$ such that

$$SA \vdash_L e : \tau.$$

That is, W_L is ***sound.***

2. Moreover, if, for some $\sigma \in T_L$ and S,

$$SA \vdash_L e : \sigma, \text{ then } W_L(A, e) = (S', \tau),$$

where, for some $\bar{S}$, $S =_{V(A)} \bar{S}S'$ and $\bar{S}\forall_{S'A}.\tau > \sigma$.

This asserts the *completeness* of W_L.

Proof. The proof of both statements is by structural induction on e.

1. Follows easily, by structural induction, using 3. and 4. of fact 8.

2. As for theorem 4 we prove the statement of completeness with $=_{V(A)}$ replaced by $=_{V-N}$, where N are the new variables generated by $W_L(A, e)$.

For constants the result is immediate. For variables derives directly from 8.3.

Let e be *let* $x = e_1$ *in* e_2. If $SA \vdash_L e : \sigma$ then, for some τ,

$$SA \vdash_L e : \tau$$

and, by 8.4, $\forall_{SA}.\tau > \sigma$. Therefore, for some σ_1,

$$SA \vdash_L e_1 : \sigma_1, \tag{12}$$

and

$$SA_x \cup \{(x, \sigma_1)\} \vdash_L e_2 : \tau. \tag{13}$$

By the inductive hypothesis applied to (12), $W_L(A, e_1) = (S_1, \tau_1)$ and, for some $\bar{S}_1$,
$S =_{V-N_1} \bar{S}_1 S_1$ and $\bar{S}_1 \forall_{S_1 A}.\tau_1 > \sigma_1$. N_1 are the new variables generates by $W_L(A, e_1)$, hence $N_1 \cap V(A) = \emptyset$. By 8.3 and $SA = \bar{S}_1 S_1 A$ applied to (13)

$$\bar{S}_1(S_1 A_x \cup \{(x, \forall_{S_1 A}.\tau_1)\}) \vdash_L e_2 : \tau$$

By the inductive hypothesis on e_2, $W_L(S_1A_x \cup \{(x, \forall_{S_1A}.\tau_1)\}, e_2)$ succeeds and returns (S_2, τ_2) such that, for some $\bar{S}_2$, $\bar{S}_1 =_{V-N_2} \bar{S}_2S_2$ and

$$\bar{S}_2\forall_{S_2S_1A}.\tau_2 > \tau, \tag{14}$$

where N_2 are the new variables generated in the call. Observe that, by the previous equality between substitutions, and $N_1 \cap N_2 = \emptyset$, $\bar{S}_2S_2S_1A = SA$. Therefore none of the variables free in τ and not free in SA can be free in $\bar{S}_2\forall_{S_2S_1A}.\tau_2$. From 8.2 and 3 applied to (14), it follows that $\bar{S}_2\forall_{S_2S_1A}.\tau_2 > \sigma$. Let $N = N_1 \cup N_2$. From the previous equalities $S =_{V-N} \bar{S}_2S_2S_1$. Therefore $\bar{S}_2$ satisfies the conditions required for $\bar{S}$.

Let e be $e_1(e_2)$. If $SA \vdash_L e_1(e_2) : \sigma$ then, for some τ and τ',

$$SA \vdash_L e_1 : \tau' \rightarrow \tau$$

and

$$SA \vdash_L e_2 : \tau'$$

and $\forall_{SA}.\tau > \sigma$. By the inductive hypothesis on e_1, $W_L(A, e_1) = (S_1, \tau_1)$, and, for some $\bar{S}_1$, $S =_{V-N_1} \bar{S}_1S_1$ and $\bar{S}_1\forall_{S_1A}.\tau_1 > \tau' \rightarrow \tau$. N_1 is the set of new variables generated in $W_L(A, e_1)$. Therefore, $SA = \bar{S}_1S_1A$ and

$$\bar{S}_1S_1A \vdash_L e_2 : \tau'.$$

By the inductive hypothesis on e_2, $W_L(S_1A, e_2) = (S_2, \tau_2)$, and, for some $\bar{S}_2$, $S_1 =_{V-N_2} \bar{S}_2S_2$ and $\bar{S}_2\forall_{S_2S_1A}.\tau_2 > \tau'$. N_2 is the set of new variables generated in the call. Let $N = N_1 \cup N_2$. As $N_1 \cap N_2 = \emptyset$,

$$\bar{S}_1S_1 =_{V-N} S =_{V-N} \bar{S}_2S_2S_1$$

By the properties of the new variables,

$$\bar{S}_1S_1A = SA = \bar{S}_2S_2S_1A. \tag{15}$$

Let t be the new variable chosen in the algorithm. We want to show that $\tau_2 \rightarrow t$ and $S_2\tau_1$ are unifiable. Let $\forall_{S_1A}.\tau_1 = \forall t''_1...t''_n.\tau_1$, where

$$V'' = \{t''_1, ..., t''_n\} = V(\tau_1) - V(S_1A),$$

and $\forall_{S_2S_1A}.\tau_2 = \forall t'_1...t'_m.\tau_2$, where

$$V' = \{t'_1, ..., t'_m\} = V(\tau_2) - V(S_2S_1A)$$

Without loss of generality we make the following assumptions about V' and V''.

(a) $V' \cap V'' = \emptyset$,

(b) $V'' \cap V(\tau_2) = \emptyset$ and $V' \cap V(\tau_1) = \emptyset$,

(c) $(V' \cup V'') \cap (\{t' \mid S_2 t' \neq t'\} \cup \{t' \mid \bar{S}_2 t' \neq t'\}) = \emptyset$,

(d) $t \notin V' \cup V''$.

From the inductive hypotheses and the definition of application of a substitution to a type, for some $\tau_1'', ..., \tau_n''$,

$$\tau' \rightarrow \tau = (\bar{S}_1[t_1'' \mapsto \tau_1'', ..., t_n'' \mapsto \tau_n''])\tau_1$$

and, for some $\tau_1', ..., \tau_m'$,

$$\tau' = (\bar{S}_2[t_1' \mapsto \tau_1', ..., t_m' \mapsto \tau_m'])\tau_2. \tag{16}$$

Let

$$Q \equiv [t \mapsto \tau, t_1'' \mapsto \tau_1'', ..., t_n'' \mapsto \tau_n'', t_1' \mapsto \tau_1', ..., t_m' \mapsto \tau_m'].$$

(Note that Q is not a substitution; we just use it as an abbreviation for its right hand side.) From assumptions (a), (b) and (d), and (16),

$$(\bar{S}_2 Q)(\tau_2 \rightarrow t) = \tau' \rightarrow \tau.$$

By definition of V' and V'', (15), (a), (b) and (d), $(\bar{S}_1 Q)\tau_1 = (\bar{S}_2 S_2)Q\tau_1$. From assumption (c), $(\bar{S}_2 S_2 Q)\tau_1 = (\bar{S}_2 Q)S_2\tau_1$. Therefore

$$(\bar{S}_2 Q)S_2\tau_1 = \tau' \rightarrow \tau,$$

and $\tau_2 \rightarrow t$ and $S_2\tau_1$ are unifiable. Let U be their most-general unifier. $\bar{S}_2 Q = S'U$, for some S', and $Ut' = t'$ for all $t' \notin V(\tau_2 \rightarrow t) \cup V(S_2\tau_1)$ (see the Appendix). Let

$$\bar{S} = S'[t_1'' \mapsto t_1'', ..., t_n'' \mapsto t_n'', t_1' \mapsto t_1', ..., t_m' \mapsto t_m'].$$

By the assumptions on V' and V', and the property of U, $\bar{S}U = \bar{S}_2[t \mapsto \tau]$. We then derive that $\bar{S}US_2S_1 =_{V-N} S$. By the same argument used at the end of the proof for the *let* construct we can show that $\bar{S}\forall_{SA}(Ut) > \sigma$.

We omit the proof for abstraction, which is similar to the proof for the *let* construct. □

As a corollary of 9 we can derive that: given an expression e it is decidable whether there is a context A and a type σ such that $A \vdash_L e : \sigma$. (Apply W_L to e and the context A in which the free variables of e are assigned distinct type variables.)

If we were to use our simple polymorphic language (without constants) as a programming language there would be little we could do. As shown in [13] each expression of the language is strongly normalizable (any reduction sequence terminates). We therefore would not be able to encode possibly nonterminating computations. That in itself would not be a loss. However, the same paper gives a bound on the length of the reduction sequence of expressions. Such a bound

implies that we cannot compute the Ackermann function, although, we certainly want to be able to express arbitrarily complex functions. In particular, the untyped fixed point operator

$$(\lambda f.(\lambda x.f(x(x)))(\lambda x.f(x(x))))$$

cannot be assigned any type. In ML, to express recursion, the construct

$$\mathit{letrec}\ x = e\ \mathit{in}\ e'$$

is introduced. This is a short-hand for:

$$\mathit{let}\ x = Y(\lambda x.e)\ \mathit{in}\ e' \tag{17}$$

where Y is a family of fixed point operators of type $(\tau \to \tau) \to \tau$ for all τ. No new rule is needed for the construct *letrec*. In a derivation of a type for (17) the occurrences of x in e are then typed following the rule of lambda-bound variables (different instances must have the same type) and in e' following the rule of *let* bound variables (different instances of x may be assigned different types). The result, especially when we consider mutually recursive functions, is not always what is expected, as the following example shows. Consider the following definition:

$\mathit{letrec}\ \ \mathit{map} = \lambda f, l.\mathbf{if}\ \mathit{null}(l)\mathbf{then}\ l\ \mathbf{else}\ \mathit{cons}(f(\mathit{hd}(l)), \mathit{map}(f, \mathit{tl}(l)))$
$\mathit{and}\ \ \ \mathit{squarelist} = \lambda l.\mathit{map}(\lambda x.x^2, l)$

The list manipulation functions are the standard ones provided by the abstract data type *list*. The resulting typing would be:

- $\mathit{map} : (\mathit{Int} \to \mathit{Int}) \times \mathit{Int}(\mathit{list}) \to \mathit{Int}(\mathit{list})$,
- $\mathit{squarelist} : \mathit{Int}(\mathit{list}) \to \mathit{Int}(\mathit{list})$,

whereas for the semantically equivalent program in which only *map* is recursively defined we would get the intuitive typing:

- $\mathit{map} : (t \to t') \times t(\mathit{list}) \to t'(\mathit{list})$,
- $\mathit{squarelist} : \mathit{Int}(\mathit{list}) \to \mathit{Int}(\mathit{list})$.

Function Definition by Patterns in HOPE

Usually in programming languages n-ary functions are defined by giving their behavior on n-tuples of variables of the corresponding domains. A distinctive characteristic of HOPE is its treatment of user defined (possibly recursive) data types. Types denote sets of terms on constructors. Functions on types can then be defined by sets of equations, each equation specifying the function on a subset of the data type denoted by a pattern. Then case analysis can be made explicit in the definition of the function and recursion implicitly packaged. This implicit definition of functions introduces a new kind of type checking problem: is the function really well defined?

To illustrate the idea we give an example. Consider

$$num == \mathbf{0} + \mathbf{S}(num)$$

to be the definition of natural numbers. The set corresponding to *num* is $\{\mathbf{0}, \mathbf{S}(\mathbf{0}), ..., \mathbf{S}(...\mathbf{S}(\mathbf{0})), ..\}$. The constructors **0** and **S** in the definition of *num* are not interpreted (they are considered syntactically as terms) whereas the + operator is interpreted as the union of the the two sets of terms. A more complex example is given by the definition of trees of numbers

$$numtree == \mathbf{empty} + \mathbf{tip}(num) + \mathbf{node}(numtree \times numtree)$$

Here the data type *numtree* is defined from the data type *num*. The operator $\times$ is interpreted as the product of the two sets of terms. (It allows the definition of n-ary constructors.) Both + and $\times$ are assumed to be associative.

In general a data type *dt* is defined from a set of constructors C each one with an associated arity and a specification of the data types of its arguments. For $\mathbf{c} \in C$, $\rho(\mathbf{c})$ is the arity of **c** and $\sigma(\mathbf{c}) = \langle dt_1, \cdots, dt_{\rho(\mathbf{c})} \rangle$ the n-tuple of the data types of its arguments (we assume that the result is *dt*)[2].

And now we get to function definition on data types. Patterns on terms of a given data type *dt* are terms of *dt* possibly containing (*non-repeated*) variables. A pattern denotes the set of terms of a *dt* obtained by instantiating its variables with terms of the right data type. For instance Fibonacci numbers are defined by

$$\begin{aligned} f(\mathbf{0}) &= \mathbf{1} \\ f(\mathbf{S}(\mathbf{0})) &= \mathbf{1} \\ f(\mathbf{S}(\mathbf{S}((x))) &= f(x) + f(\mathbf{S}(x)) \end{aligned}$$

As mentioned at the beginning of the section we are interested in checking the conditions that make a function:

$$f : dt_1 \times \cdots \times dt_n \rightarrow dt \tag{18}$$

---

[2]HOPE data type definitions are more general. However this captures the basic idea and allows the formal definition of the problem of *well definedness* for function definition.

(the type of the result is not important) defined by a set of equations

$$\begin{aligned} f(\langle p_1^1, ..., p_1^n \rangle) &= e_1 \\ &\vdots \\ f(\langle p_k^1, ..., p_k^n \rangle) &= e_k \end{aligned} \tag{19}$$

well defined. The p_i^j's, for $1 \leq i \leq k$ are patterns on dt_j. We may assume that for $m \neq m'$ the set of variables of $\bar{p}_m$ is disjoint from the set of variables $\bar{p}_{m'}$.

Definition 10: Let f be defined as in (18) and (19). We say that f is *well defined* when, for all n-tuple of terms $\bar{t} = \langle t_1, ..., t_n \rangle$ such that, for $1 \leq i \leq n$, $t_i \in dt_i$

1. there is a pattern sequence $\bar{p}_j = \langle p_j^1, ..., p_j^n \rangle$ such that some instance of $\bar{p}_j$ is equal to $\bar{t}$ ($\bar{p}_j$ matches $\bar{t}$), and
2. the matching pattern sequence is unique. □

The first condition of the definition says that the domain of f is $dt_1 \times \cdots \times dt_n$. The function could be undefined on elements of the domain but it must be the case that it is applicable to each element of the domain. The second condition insures that the function behaves deterministically. The choice of which clause of the definition to use is unambiguous. The previous conditions do not directly give a way of deciding if a function f is well defined or not.

A sufficient condition for 10.1 comes from a similar problem that arises in the literature on term rewriting systems, see [21]. The basic idea consists of checking that each column of the pattern matrix covers all the possible terms of the corresponding data type.

Definition 11: A set of n-tuples of patterns $P = \{\bar{p}_1, ..., \bar{p}_k\}$ is *complete* for the data type $dt_1 \times \cdots \times dt_n$ if either $n = 0$ and $P = \{\langle\rangle\}$ or:

1. either the set of $n-1$ tuples $\{\langle p_i^2, ..., p_i^n \rangle \mid p_i^1 \text{ is a variable}\}$ is complete for $dt_2 \times \cdots \times dt_n$,
2. or for all constructors in $\mathbf{c} \in dt_1$, $\rho(\mathbf{c}) = m$ and $\sigma(\mathbf{c}) = \langle dt'_1, \cdots, dt'_m \rangle$ the set:

$$\{\langle p_1, ..., p_m, p_i^2, ..., p_i^n \rangle \mid p_i^1 = \mathbf{c}(p_1, ...p_m)\} \cup \{\langle x_1, ..., x_m, p_i^2, ..., p_i^n \rangle \mid p_i^1 \text{ is a variable}\}$$

is complete for $dt'_1 \times \cdots \times dt'_m \times dt_2 \times \cdots \times dt_n$. □

Consider the set of pairs of natural numbers, ordered lexicographically, whose first component is the number of constructors in P and the second the length of the tuples (number of data types). The previous definition is well founded w.r.t. this order, and therefore provide an algorithm for the corresponding test.

From the assumption that, for $i \neq j$, the set of variables of $\bar{p}_i$ is disjoint from the set of variables of $\bar{p}_j$, 10.2 is equivalent to: for all $i \neq j$, $\bar{p}_i$ $\bar{p}_j$ are not unifiable. We then know that the two patterns do not have any common instance.

The result that provides a sufficent condition for the well-definedness of a HOPE function is given by the following proposition.

Proposition 12: Let f be defined as in (18) and (19), if

1. the set of n-tuples of patterns $P = \{\bar{p}_1, ..., \bar{p}_k\}$ is *complete* for the data type $dt_1 \times \cdots \times dt_n$, and
2. for all $i \neq j$, $\bar{p}_i$ $\bar{p}_j$ are not unifiable,

then f is *well defined.* □

In the example of the Fibonacci numbers 12.2 is verified. To show the completeness of the set of patterns we start with $\{\langle \mathbf{0} \rangle, \langle \mathbf{S}(\mathbf{0}) \rangle, \langle \mathbf{S}(\mathbf{S}(x)) \rangle\}$. 11.2 applies and for the constructor **0** we get a complete set. For **S** we have to prove the completeness of $\{\langle \mathbf{0} \rangle, \langle \mathbf{S}(x) \rangle\}$ that follows by applying 11.2 again and then 1 to $\{\langle x \rangle\}$.

1.2. Extensions to the Type System

Some extensions of the basic type system of 1 have been proposed. We briefly consider: type coercion ([33]) and circular types ([29]), both of which are relevant to programming languages.

Automatic coercion is a feature of most languages providing numeric types. For instance, if reals and integers are provided, functions accepting real values can be given integer input and the compiler automatically inserts the coercion needed. This notion of coercion follows the basic model of set inclusion. We think of integers as a subset of reals and then the set of functions from reals to reals contains as a subset the set of functions from integers to reals. The notion can be defined precisely by considering a partial order on the set of constant types, the set of coercions allowed, and extending this order to functional types as well. Let's say $\tau \leq_C \tau'$ if τ can be coerced to τ'. An intuitive rule for the deduction of types for an application $e(e')$ is the following:

$$\frac{A \vdash e : \tau \to \tau' \quad A \vdash e' : \tau'' \quad \tau'' \leq_C \tau}{A \vdash e(e') : \tau'}$$

A complete set of rules and a deduction algorithm is given in [33]. The same paper also gives a semantic justification of the system by interpreting the language in an untyped lambda-calculus model and representing coercion as set inclusion in the model.

Circular types arise in the context of data type specification. For instance polymorphic lists can be defined as:

$$list = t + (t \times list)$$

Consider an application of $e(e')$ if $list \to \tau$ is the type deduced for e and $t + (t \times list)$ is the type deduced for e'. The ML type checker would produce a type error. (To allow functions on recursively defined types ML provides an *ad hoc* mechanism.) In [29] a type system is proposed that includes types as solutions of recursive equations such as *list*. With circular types the expression $\lambda x.xx$ (as well as the fixed point operator) can be given type $t = t \to t$ ($\equiv \mu t.(t \to t)$ in the notation used

there). Moreover every expression not containing constants can be typed. To deduce circular types we add to the basic system given in the first subsection the rules:

μi: $\dfrac{A \vdash e : \sigma[\mu t.\sigma/t]}{A \vdash e : \mu t.\sigma}$

μe: $\dfrac{A \vdash e : \mu t.\sigma}{A \vdash e : \sigma[\mu t.\sigma/t]}$

Here $\sigma ::= t \mid k \mid \sigma \to \sigma' \mid \sigma \times \sigma' \mid \mu t.\sigma$. The paper focuses on the semantics of such types and the existence of solutions for recursive type equations is proved. No result is given for the deduction system. However, it seems likely that an extension of unification, which does not perform the "occurrency check" (see Appendix) would make possible to define a deduction algorithm.

2. Polymorphic Lambda-Calculus and Type Quantification

The following section introduces a formulation of lambda-calculus with higher order polymorphic type structure. The type-checking problem for polymorphic lambda-calculus is almost trivial, because polymorphism is not an artifact of the use of variables in type specifications but is explicitly represented by the use of abstraction over types.

In the following we first introduce some motivations for going further in the research for polymorphic type structures. The polymorphically typed lambda-calculus, developed by Reynolds [37], and its type structure that includes quantified types is presented. The type-deduction problem for the language is a notational variant of the same problem for the quantificational discipline introduced in [26]. Not much is known about its complexity (decidability). RUSSELL is, then, briefly introduced as an example of a language in which polymorphism is achieved by explicit abstraction over (data) types. RUSSELL is further from the polymorphic lambda-calculus than ML and HOPE are from the parametric. It is, however, a language in which user-defined types are introduced and treated uniformly with the primitive types of the language; that was Reynolds' original motivation for the introduction of polymorphic lambda-calculus.

2.1. Why would we need more than ML?

Even though the approach to polymorphism taken in ML has been shown to be very successful there are still many questions that have to be answered about the *right* underlying type structure for polymorphic languages.

As remarked in 1.1, in ML to allow the use of polymorphic expressions a special construct (*let*) is needed. Such construct introduces an unpleasant non uniformity in the treatment of declared (*let*-bound) and parameter (lambda-bound) variables. Let's explore the implications that a uniform treatment of declarations and parameters, which exploits as much as possible the polymorphism of expressions, would impose on the type structure of the language.

Consider a declaration

$$let\ x = ...\ in\ e$$

as an abbreviation of the semantically equivalent expression

$$(\lambda x.e)(...).$$

Assume that a type for e, say τ', can be deduced from the assumption on x, $(x, \forall t_1..t_n.\tau)$. If different instantiations (on non unifiable types) of the type variables of x are required in such a deduction, the ML type-deduction algorithm would fail to assign a type to $\lambda x.e$. (For instance in ML to assign a type to $\langle f(n), f(true)\rangle$ in $let\ f = ...\ in\ \langle f(n), f(true)\rangle$ the type of f has to be instantiated to $Int \rightarrow \tau$ and $Bool \rightarrow \tau'$ for some τ and τ'.) However, if we allow ourselves the use of a less restrictive set of types, the type of $\lambda x.e$ can be expressed by $(\forall t_1...t_n.\tau) \rightarrow (\forall t'_1...t'_m.\tau')$,

where the t'_i's are the type variables introduced by instantiations of the type of x. Moreover, the type of $\lambda x.e$ does not depend on any of the type variables free in the type of x and not free in the type of any other assumption. We can express this polymorphism by introducing an outermost quantification (on all such variables). The resulting type for $\lambda x.e$ would then be

$$\forall t''_1 ... t''_i .((\forall t_1 ... t_n .\tau) \rightarrow (\forall t'_1 ... t'_m .\tau')).$$

Now, as expressions of such types may be specified as parameters of functions, the complexity of the types required grows. In particular τ and τ' may be quantified types, as well. This leads to consider the full hierarchy of quantified types Δ, where $\delta \in \Delta$ is defined as follows.

$$\delta ::= t \mid \kappa \mid \delta \rightarrow \delta' \mid \forall t.\delta \tag{20}$$

How useful such type structure would be is still an open problem. It is clear, from the *let* example that we *do need* some sort of quantification over types to be able to use the polymorphism intrinsic in certain expressions. However, it is not clear what we get from the use of the full hierarchy of quantified types. In the following we present a type deduction system (see [26] and [22]) for the previous type structure.

The treatment of user defined types (abstract types) is another debatable point of the type system of ML. The elaboration of abstract data type definition:

$$\textbf{abstype } (t_1, ..., t_n)name = \tau \textbf{ with } f_1 = e_1, ..., f_n = e_n$$

creates let bindings for the functions defined $(f_1, ..., f_n)$ and introduces polymorphic type operators (in this case *name*). No type is ascribed to the compound definition. Consider the example of the data structure tree:

$$\textbf{abstype } (t)tree = t + (t)tree \times (t)tree \textbf{ with...}$$

The main problem is that the representation of trees and the implementation of the abstract data type functions cannot be separated from their specifications. That precludes the possibility of using, in the same program, more than one representation and implementation of the functions for the generic data structure tree. (We remark that separation would preclude inference of types.) This sort of polymorphism is quite different from the polymorphism analyzed in this paper. Here we want to be able to write functions that have, as explicit parameters, the functions associated with a given data type. We will see how a sort of representation independence of the kind previously stated can be achieved in languages in which types are part of the expression language and can be passed as parameters in a controlled way.

2.2. Polymorphic Lambda-Calculus

From the examples given in the previous section, the main problem of the parametric approach to polymorphism seems to be how to limit the scope of free type variables of expressions.

A uniform approach to such problem is given by Reynolds' polymorphic lambda-calculus. Here the idea is that polymorphic expressions are functions over types and explicit type application is needed to instantiate their types. Type abstraction and application are made explicit in the expression language, E_R, that is defined by the following clauses.

$$e ::= x \mid \lambda x : \delta.e \mid e(e') \mid c_\delta \mid \Lambda t.e \mid e[\delta]$$

where types $\delta \in \Delta$ are defined in (20). Family of constants, like the selectors and constructors for products, can be specified by

$$\begin{array}{l} fst_{\forall t.\forall t'.((t\times t')\to t)} \\ snd_{\forall t.\forall t'.((t\times t')\to t')} \\ \langle\cdot,\cdot\rangle_{\forall t.\forall t'.(t\to t'\to(t\times t'))} \end{array} .$$

Definition 13: Let A be a context on Δ, $e \in E_R$ and $\delta \in \Delta$. The relation $A \supset e : \delta$ (e *has type* δ *in* A) is the smallest relation satisfying :

1. $A \supset c_\delta : \delta$ for all c_δ;
2. if $A(x) = \delta$, then $A \supset x : \delta$;
3. if $A_x \cup \{(x, \delta)\} \supset e : \delta'$, then $A \supset \lambda x : \delta.e : \delta \to \delta'$;
4. if $A \supset e : \delta \to \delta'$, $A \supset e' : \delta''$, and δ'' is equal to δ up to renaming of quantified variables, then $A \supset e(e') : \delta'$;
5. if $A \supset e : \delta$, and t is not free in the type of any variable free in e, then $A \supset \Lambda t.e : \forall t.\delta$;
6. if $A \supset e : \forall t.\delta$, and δ' is free for t in δ, then $A \supset e[\delta'] : \delta[\delta'/t]$.

We say that e is *well typed* w.r.t. A if $A \supset e : \delta$ for some δ. □

The restriction on t in 13.5 is meant to rule out expressions like

$$\lambda x : t.(\Lambda t.(\lambda y : t \to t.y(x))). \tag{21}$$

Such an expression is equivalent to $\lambda x : t.(\Lambda t'.(\lambda y : t' \to t'.y(x)))$ which is not well typed. However, the subexpression $y(x)$ of (21) is well typed w.r.t. $A = \{(y, t \to t), (x, t)\}$. Therefore, without the restriction in 13.5, (21) would be well typed. Similarly, consider the expression

$$(\Lambda t'.(\lambda x : (\forall t.t \to t').x)[t](\Lambda t.\lambda x : t.x) \tag{22}$$

which is equivalent to

$$(\Lambda t'.(\lambda x : (\forall t''.t'' \to t').x)[t](\Lambda t.\lambda x : t.x). \tag{23}$$

Without the restriction 13.6, (22) is well typed, whereas (23) is not.

The result of the explicit introduction of types in the expression language is that (polymorphic) expressions are uniquely typed. As an example we can express our staple polymorphic identity function by $\Lambda t.\lambda x : t.x$. Here, explicit application to a type δ, is needed to obtain the identity function of type $\delta \to \delta$. Self application, among other ways, can be expressed by $\lambda x : (\forall t'.t').((x[t \to t])(x[t]))$. We can see that the instantiation of the type variable is made explicit in the expression by the use of type application.

As shown in [13] the expressions of this language are strongly normalizable. However, the result does not depend on not allowing any form of self application as it does for the parametric case.

The type-deduction problem for this system is obtained by erasing types and Λ's from expressions and defining rules for assigning types and inserting type abstraction and application, if necessary, in the expression. (Consider for example $\lambda x.xx$). The problem is a notational variant of the one for the quantificational discipline considered below[3].

2.3. Type Quantification

The idea of expressing polymorphism through type quantification (implicit in the *let* construct) is generalized to arbitrary types and extended to lambda-binding in the quantificational approach to polymorphism, see [26].

The set of types the language is again Δ defined in (20). The expression of the language $e \in E_Q$ are defined by

$$e ::= \lambda x : \delta.e \mid e(e') \mid x \mid c_\delta$$

No explicit construct is introduced in the expression language to delimit the scope of type variables. A definition of the well typed expressions can be obtained by 13 replacing condition 5 and 6 by

$5'$. if $A \supset e : \delta$, and t is not free in A, then $A \supset e : \forall t.\delta$;

$6'$: if $A \supset e : \forall t.\delta$, and δ' is free for t in δ, then $A \supset e : \delta[\delta'/t]$.

In this language, uniqueness of the type of an expression in a context is lost. Type quantification can be introduced on any type variable of the type of an expression which is not free in the context. An expression e having a type δ has implicitly all the types that are an instance of $\forall t_1 ... t_n.\delta$ where the t_i's are type variables free in the context of e. In particular the type of a closed expression can be quantified over all its free type variables. The *let* construct is now superfluous as its semantic equivalent application (9) is equivalent also from the typing point of view.

To introduce type deduction we erase types from lambda-bound variables obtaining, as in the previous section, an untyped lambda-calculus with constants. (For constant we specify a principal type). The type constraints are expressed by the inference system $\vdash_\forall$. The rules of $\vdash_\forall$ are: $TAUT$,

[3] As shown in [13] all the expression of this language are strongly normalizable.

$CONST$, $\rightarrow IN$ and $\rightarrow EL$ of 1 (in which δ replaces τ) and the polymorphic type derivation and instantiation:

$$\forall IN: \quad \frac{A \vdash_\forall e : \delta}{A \vdash_\forall e : \forall t.\delta} \qquad (t \notin V(A))$$

$$\forall EL: \quad \frac{A \vdash_\forall e : \forall t.\delta}{A \vdash_\forall e : \delta[\delta'/t]} \quad (\delta' \text{ free for } t \text{ in } \delta)$$

The difficulty in this deduction problem is the almost complete absence of a syntax driven strategy. Compare this system with $\vdash_L$ for ML. There the only case in which type quantification is actually used is in the LET rule From the limitation to simple types, type quantification can be simulated, as the proof of completeness of W_L shows, by substituting new type variables for the quantified ones and using standard first-order unification. However, if we assume that instantiation can be on any type $\delta \in \Delta$, then first-order unification cannot be directly used. On the other hand, considering the type structure as natural, a second-order language (see [20]) does not help as second-order unification is known to be an undecidable problem.

The deduction system $\vdash_\forall$ can be trivially modified to deduce polymorphic lambda-calculus expressions. (Just replace e by $\Lambda t.e$ and $e[\delta']$ in the consequence of $\forall IN$ and $\forall EL$ rules respectively). Therefore, as for polymorphic lambda-calculus, all the expressions that can be assigned a type are strongly normalizable. The characterization of the set of expressions for which we can deduce a type and the decidability of the system are still unsolved problems. Attempts to solve different versions of the problem have been done in [2], [22], and [25]. None of the previous, however shows enough evidence of the result.

Let ω be the expression $\lambda x.xx$. From the strong normalization property of typable expressions we can derive that no type can be deduced for $\omega\omega$ (which is not normalizable). A direct proof of the statement gives some light on the kind of statement derivable in the system. Observe that, to derive a type for xx in some context $A = \{(x, \bar{\delta})\}$, $\bar{\delta} \in \Delta$ must be a non vacuously quantified type. Otherwise we could not derive types for the two occurrences of x that match the premises of $\rightarrow EL$. Let $\bar{\delta} = \forall t_1 ... t_n.\delta''$, for some t_1,..., t_n and δ'' such that δ'' is not externally quantified. Assume for the moment that $\delta'' = \delta \rightarrow \delta'$ for some δ and δ', i.e., δ'' is not a variable. By the matching involved in the application xx it must be that, for some types δ_i's, δ_i''s and variables t_i''s not free in δ''

$$\delta[\delta_1/t_1, ..., \delta_n/t_n] \cong \forall t_1' ... t_m'.(\delta \rightarrow \delta')[\delta_1'/t_1, ..., \delta_n'/t_n] \tag{24}$$

where $\cong$ denotes equal up to renaming of bound variables. It is not difficult to show that (24) implies that the innermost leftmost subterm of δ has to be an occurrence of one of the t_i's. Therefore the type derived for ω is the following

$$\forall t_1'' ... t_q''.((\forall t_1 ... t_n.\delta \rightarrow \delta') \rightarrow \bar{\delta}') \tag{25}$$

for some $\bar{\delta}'$ and t_i'''s. Consider now deriving a type for $\omega\omega$. The types of both occurrences of ω must satisfies similar restrictions. Let (25) be the type of rightf occurrence of ω and for some σ, σ' and $\bar{\sigma}' \in \Delta$ and $\bar{t}_i$'s let

$$(\forall \bar{t}_1 ... \bar{t}_q . \sigma \rightarrow \sigma') \rightarrow \bar{\sigma}'$$

be the type of the left occurrence of ω. By (24) the leftmost innermost subterm of σ has to be one of the $\bar{t}_i$'s. (Note that any application of $\forall IN$ or $\forall EL$ after a $\rightarrow IN$ and before $\rightarrow EL$ can be eliminated, hence no quantification has to be added to such a type.) Now (25) has to match $\forall \bar{t}_1 ... \bar{t}_q . \sigma \rightarrow \sigma'$ and then $\forall t_1 ... t_n . \delta \rightarrow \delta'$ has to match σ. However, the leftmost innermost subterm of $\delta \rightarrow \delta'$ (that coincides with the one of δ) has to be one of the t_i's, whereas the one of σ must be (after renaming the variables) one of the t_i'''s. This is a contradiction and thus $\omega\omega$ cannot be assigned any type. (Even though quite tedious the proof is as far as I know the only direct proof of non-existence of a typing for an expression in the quantificational type discipline.)

2.4. RUSSELL

The language RUSSELL ([10] and [11]) combines the type abstraction and application constructs of polymorphic lambda-calculus with the algebraic approach to abstract data types. However type abstraction and application means in this context data type abstraction and application. We remark the difference between polymorphic lambda-calculus types and RUSSELL data types. The first can be interpreted as some set of values (may be complex, due to type abstraction). The second are sets of operations that describe the possible transformations that can be performed on objects of the given data type. (Data types are nothing more than sets of expressions of the language.) This is the sense in which the algebraic approach to abstract data types comes into play. However, operations are not characterized by equations, but simply by their type information. The limitation comes on one hand from the requirement of static type checking, on the other from the computational nature of the language. It would be unclear what is meant by an expression returning or getting as an input a set of equations. The purely algebraic approach, indeed, is mainly found in specification languages where description, rather than computation, is the issue. RUSSELL's view of data types is shared by other languages. The most notable are ALPHARD and CLU.

In the following we give an overview of an *applicative subset* of RUSSELL. We introduce the definition of *signature* and *expression* and briefly describe the type-checking algorithm: *signature calculus*. *Signatures* have a role similar to types of polymorphic lambda-calculus. The relation between them will be analyzed åfter giving the details of the language.

Signatures. The basic signatures of expressions of the language are:

$$S ::= \mathsf{fun}[I_1 : S_1, ..., I_n : S_n] : S' \mid \mathsf{type}\ I[I_1 : S_1, ..., I_n : S_n] \mid e$$

fun is the signature of functions. It specifies the signature of the parameters and that of the result. For $i < j$, I_i is bound to S_i in S_j and S'. type is the signature of a data type; it specifies the signature of the operations of the data type. Here the S's have to be either data type or function signatures. The identifier I is bound in all the S's so that recursive data types can be defined. Finally e is the signature of a value of a given data type, where e is any expression that has a type signature. In the examples the identifiers I's when not significant will be omitted.

Examples. A signature for the polymorphic identity function could be:

$$\mathsf{fun}[t : \mathsf{type} \ ...] : \mathsf{fun}[x : t] : t$$

As we can see, type abstraction and normal abstraction have the same signature (fun). It is not so for the polymorphic lambda-calculus where the type of the identity would be $\forall t.t \rightarrow t$. However, while in polymorphic lambda-calculus $t \rightarrow t$ is a type, $\mathsf{fun}[x : t] : t$ is not a type in RUSSELL. Consider expressing self application $g = \lambda x.xx$. First the signature of x must be $\mathsf{fun}[I : S] : S'$ where S has to be a fun signature as well in which I cannot occur. It is thus impossible to obtain a signature for g. We will see in the following how the introduction of a signature transformation operator makes possible the introduction of self application.

Expressions. We introduce a subset of the expressions of RUSSELL. The constructs chosen capture the underlying philosophy of the language. The expression e of R are defined by the following syntax:

$$\begin{array}{rcl} e & ::= & \mathit{fun}\ f(I_1 : S_1, ...I_n : S_n) : S\{e\} \mid e(e_1, ..., e_n) \mid \\ & & \mathit{Int} \mid \mathit{Bool} \mid \mathit{record}\ I(I_1 : S_1, ..., I_n : S_n) \mid e\$I \mid \\ & & \mathit{let}\ x : S = e\ \mathit{in}\ e' \mid x \end{array}$$

The first line corresponds to function definition and application. The syntax is the standard one. The signature of the defined function is: $\mathsf{fun}[I_1 : S_1, ..., I_n : S_n] : S$. For an application e has to have a signature $\mathsf{fun}[I_1 : S_1, ...I_n : S_n] : S$ for some I's and S's and if S'_1,...,S'_n are the signatures of the arguments, the signature of the result is obtained by substituting S'_1,...,S'_n for I_1,...,I_n in S. The *type-checking problem* is concerned with defining when such application is well typed. We will describe the rules later.

Data types can be defined from the basic data types *Bool* and *Int*. The signature of such types specifies their basic operations defined. For instance for *Bool* we would have the constants *true* and *false* of signature $\mathsf{fun}[] : \mathit{Bool}$ and all the logical operators. Similarly for *Int*. So an example of an expression of signature *Bool* is:

$$\mathit{and}(\mathit{true}(), \mathit{false}()).$$

Nullary functions are the standard way of introducing constants. The construct *record* defines the cartesian product of the corresponding data types. The signature of the expressions describe the

operations provided by the resulting data type. In this case they are the selectors π_i with signature $\mathsf{fun}[x : I] : I_i$ and the constructor *mkrec* with signature $\mathsf{fun}[x_1 : t_1, ..., x_n : t_n] : I$[4]. The construct $e\$I$, where e is an expression of signature type $I'(..., I : S, ...)$, selects the operation named I from the set of operations provided by I', its signature is obtained by substituting e for I' in S. In this way different data types can have operation with the same name (in ML that is not possible).

Finally

$$let\ x : S = e\ in\ e'$$

binds x to the expression e of signature S in e'. The expression is equivalent to:

$$(fun\ f(x : S) : S'\{e'\})(e)$$

where S' is the signature of e'. Here no problem arises as far as typing is concerned since the expressions are explicitly typed.

Examples. As seen in the previous example self application is not expressible with the defined signatures. However, RUSSELL provides the construct *image* with signature

$$\mathsf{fun}[I : S] : \mathsf{type}[in : \mathsf{fun}[S] : I, out : \mathsf{fun}[I] : S].$$

The application of *image* to a signature S returns a data type whose operations, *in*, and *out* convert back and forth the signature in the data type. For example, the data type declaration (could be in a *let* construct):

$$t : ... = image(\mathsf{fun}[t] : t)$$

defines the recursive data type t with the *in* and *out* functions previously described. The identity function, $fun\ id(x : t) : t\{x\}$, which signature is $\mathsf{fun}[t] : t$, can be coerced to a value of type t by

$$t\$in(fun\ id(x : t) : t\{x\}). \tag{26}$$

Let's define f to be the expressions (26). The signature of f is t (f is a value of type t). Therefore

$$(t\$out(f))(f)$$

is a correctly typed self application. The conversion of f from a function to a value of a certain data type to a function is explicitly done via $t\$out$.

Type checking. As said before type checking is specified by giving the rules for matching the signatures of formal and actual parameters in an application. The rules make clear the syntactic nature of signatures. Let $\mathsf{fun}[I_1 : S_1, ...I_n : S_n] : S$ be the signature of e in $e(e_1, ..., e_n)$ and S'_1,...,S'_n

[4]RUSSELL has a *union* constructor, as well.

be the signatures of e_1,...,e_n respectively. Consider the signature resulting from the substitution of the argument signatures for the I identifiers in S_i:

$$S_i'' = S_i[S_1'/I_1, ..., S_n'/I_n]$$

The signature of the defined function and the one of the arguments *matches* if for all i, S_i'' and S_i' are both either type or e (expressions) or fun and:

- For fun signatures S_i'' and S_i' are equal up to renaming of their identifiers.
- For type signatures some of the components of S_i'' may be eliminated and the remaining ones have to be equal up to permutation and renaming of their identifiers and to the components of S_i.
- For expression signatures e, S_i'' and S_i' have to be equal. This is necessary for static type checking. Remember that e could be a function application with result signature type. Admitting a less restrictive matching, for instance conditions on the value of the expression would cause the impossibility of static checking.

Comparing RUSSELL type checking with type deduction/checking in ML we can see that RUSSELL treatment of data types (similar to ML abstract data types) satisfy the requirements of separation of implementation and specification. However, in RUSSELL no type deduction (for functions and data types) is possible.

RUSSELL and its type structure has been analyzed in [16] The language is translated into (and then given a semantics in) a polymorphic lambda-calculus enriched with sequence types needed to interpret the set of operations of data types. This shows the expressive power of polymorphic lambda-calculus as well as the versatility of its type structure as a basis for an uniform interpretation of a variety of polymorphic constructs.

Appendix: Unification Algorithms

Let K be a set of constants, with an associated arity function $\rho : K \to \mathbf{N}$, and let V be a set of variables. The first-order term language on K and V, $T(K,V)$, is the smallest set containing V such that:

$$\text{if } \kappa \in K \text{ and } \tau_1, ..., \tau_{\rho(\kappa)} \in T(K,V), \quad \text{then } \kappa(\tau_1, ..., \tau_{\rho(\kappa)}) \in T(K,V).$$

The set of types T introduced in the first section is one such language. The underlying set of constants of T is $K \cup \{\to, \times\}$, where K, there, is the set of constants of arity zero (constant types).

The metavariables t, κ, and τ, with subscripts and superscripts if necessary, range in V, K, and $T(K,V))$ respectively.

The *size* of a term τ, $\sigma(\tau)$ is defined as follows:

$$\sigma(\tau) = \begin{cases} 0 & \tau \in V \\ 1 + \sum_{1 \le i \le \rho(\kappa)} \sigma(\tau_i) & \tau = \kappa(\tau_1, ..., \tau_{\rho(\kappa)}). \end{cases}$$

$V(\tau)$ is the set of variables of τ.

Substitutions, instances, and the partial order $\le$ are a trivial generalization of the corresponding notions defined in the first section. For a substitution S, $D(S) = \{t \mid St \neq t\}$ and $R(S) = \bigcup_{t \in D(S)} V(St)$. We recall that a *unifier* U of the terms τ and τ' is a substitution such that $U\tau = U\tau'$. A *most general unifier* U is a unifier with the property that for all unifiers U' of τ and τ', there is a substitution S such that $U' = SU$. The first-order languages have the following property:

Theorem 14: Let τ, $\tau' \in T(K,V)$, if τ and τ' are unifiable then they have a most general unifier U such that $R(U) \subseteq V(\tau) \cup V(\tau')$. □

The definition of unifier can be generalized to sets of pairs of terms $P = \{\langle \tau_i, \tau_i' \rangle \mid 1 \le i \le n\}$ by defining U to be a unifier of P if $U\tau_i = U\tau_i'$ for all $1 \le i \le n$. $SP = \{\langle S\tau_i, S\tau_i' \rangle \mid 1 \le i \le n\}$ and $V(P) = \bigcup_{0 \le i \le n} V(\tau_i) \cup V(\tau_i')$. The algorithms given in the following sections solve the following generalized unification problem.

Generalized Unification Problem: given a set of pairs of terms P, find (if any) a most general unifier U of P such that $R(U) \subseteq V(P)$.

2.5. Robinson's Unification Algorithm

The first algorithm we present is Huet's version of Robinson's original algorithm, [20] and [39]. The algorithm consists of the iteration of a simplification step, $Simpl(P)$, which reduces the the complexity of the set of pairs P preserving the set of unifiers of P. The description of *Simpl* is as follows.

Simpl(P)

1. $P_0 \leftarrow P$ and $j \leftarrow 0$.
2. If $P_j = P' \cup \{\langle \tau, \tau' \rangle \mid \sigma(\tau), \sigma(\tau') > 0\}$, i.e., there is a pair neither of whose components is a variable,
   - then let $\tau = \kappa(\tau_1, ..., \tau_{\rho(\kappa)})$ and $\tau' = \kappa'(\tau'_1, ..., \tau'_{\rho(\kappa')})$. If $\kappa \neq \kappa'$ then exit with *failure*, otherwise $P_{j+1} \leftarrow P' \cup \{\langle \tau_i, \tau'_i \rangle \mid 1 \leq i \leq \rho(\kappa)\}$, $j \leftarrow j + 1$ and go to step 2.
   - otherwise go to step 3.
3. In P_j replace $\langle \tau, t \rangle$ by $\langle t, \tau \rangle$ and delete the pairs $\langle t, t \rangle$. If there is a pair $\langle t, \tau \rangle$ such that $t \in V(\tau)$ then exit with *FAIL* otherwise return the modified P_j.

Let $\sigma(P) = \sum_{\langle \tau, \tau' \rangle \in P} \sigma(\tau) + \sigma(\tau')$ be the size of the set of pairs P. Termination of *Simpl* can be proved by observing that $\sigma(P_i) > \sigma(P_{i+1})$. *Simpl* has the following properties:

- U is a unifier of P if and only if U is a unifier of *Simpl*(P),
- $V($*Simpl*$(P)) \subseteq V(P)$, and
- if $\langle \tau, \tau' \rangle \in$ *Simpl*(P) then $\tau \in V$.

This can be established by induction on the number of iterations of step 2. It is interesting to note, for step 3, that if there is a pair $\langle t, \tau \rangle$ such that $t \in V(\tau)$ then there is no unifier for P since no substitution can unify t and τ. Such a test is known as *occurrency check*.

The unification algorithm *Unify* takes as input a set of pairs of terms P and terminates with either *failure* or *success*. The unifier of P produced by the algorithm will be defined below before the theorem stating the correctness of *Unify*.

Unify(P)

1. $P_0 \leftarrow$ *Simpl*(P) and $j \leftarrow 0$.
2. If $P_j = \emptyset$
   - then exit with *success*.
   - otherwise, let $S_{j+1} = I[t \mapsto \tau]$ for $P_j = P'_j \cup \{\langle t, \tau \rangle\}$. ($\langle t, \tau \rangle$ is one of the pairs in P_j and S_{j+1} is the substitution that is the identity on all the variables except t, which is mapped to τ.)
     $P_{j+1} \leftarrow$ *Simpl*$(S_{j+1}P_j)$, $j \leftarrow j + 1$ and go to step 2.

If any call of *Simpl* fails then *Unify* fails. Note that, in step 2, since the pair $\langle S_{j+1}t, S_{j+1}\tau \rangle$ is eliminated by *Simpl*, *Simpl*$(S_{j+1}P_j) =$ *Simpl*$(S_{j+1}P'_j)$. Termination of *Unify* can be proved by observing that, since $t \notin V(\tau)$, $t \notin V(S_{j+1}P_j)$. Therefore, as *Simpl* does not introduce new variables, $\#V(P_j) > \#V(P_{j+1})$.

If the algorithm terminates in n steps ($n-1$ iterations of step 2), define the following sequence of substitutions:

- $U_n = I$, where I is the identity substitution. ($D(I) = R(I) = \emptyset$.)
- $U_{j-1} = U_j S_j$, i.e., the substitution that coincides with S_j for all the variables but t (the one chosen at the j-th step) and on t is equal to $U_j\tau$.

The correctness of the algorithm can now be stated by the following theorem.

Theorem 15: Let P be a set of pairs of terms. If there is a unifier U for P then the algorithm exits after n steps, $U_0 = S_n S_{n-1} \cdots S_1$ is a unifier of P and for some substitution S, $U = SU_0$.

Proof. (Sketch) We first prove that for all $0 \leq j \leq n$, U_j is a unifier for P_j. For $j = n$, P_j is empty and then the identity substitution is an unifier for P. For $0 < j \leq n$, assume U_j is a unifier for $P_j = \mathit{Simpl}(S_j P'_{j-1})$; then $U_{j-1} = U_j S_j$ is a unifier for P'_{j-1} and so of $P_{j-1} = \{\langle t, \tau\rangle\} \cup P'_{j-1}$. Hence U_0 is a unifier for P_0 and, by the first property of *Simpl*, of P.

Let U be a unifier for P, the existence of S is proved by showing that for all $1 \leq j \leq n+1$ there is an S'_j such that:

$$U = S'_j S_{j-1} \cdots S_1.$$

The proof is by induction on j (starting for $j = 1$ with $S'_1 = U$). □

One of the sources of inefficiency of the previous algorithm is the redundant copying of terms done in *Simpl*. For instance for the terms

$$\begin{aligned} \tau_1 &= \kappa(t, t, ..., t) \\ \tau_2 &= \kappa(\tau, t', ..., t'). \end{aligned}$$

the first application of *Simpl* in $\mathit{Unify}(\{\langle \tau_1, \tau_2\rangle\})$ makes n copies of τ (that could be a big term) and performs as many comparisons. On the other hand, even adopting a clever representation for the set of pairs that avoids repeating the same term several times, the previous algorithm is, in the worst case, exponential. The problem comes from the application of the substitution done in step 2 of *Unify*. This substitution increases the size of the pairs in P. Consider the following terms:

$$\begin{aligned} \tau_3 &= \kappa(t_2, t_3, ..., t_n) \\ \tau_4 &= \kappa(\kappa'(t_1, t_1), \kappa'(t_2, t_2), ..., \kappa'(t_{n-1}, t_{n-1})) \end{aligned}$$

$\mathit{Unify}(\{\langle \tau_3, \tau_4\rangle\})$ produces, after the first application of *Simpl*, the set of pairs $P_0 = \{\langle t_{i+1}, \kappa'(t_i, t_i)\rangle \mid 1 \leq i \leq n\}$. If the pairs are chosen in sequence the number of occurrences of t_1 in P_j grows exponentially (is equal to 2^j).

2.6. Linear Unification

The algorithm we present in this section is due to Paterson and Wegman, [35].

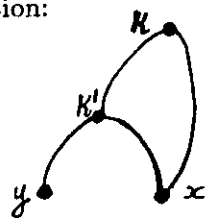
From the previous observations it is clear that the choice of the representation for both substitutions and terms is crucial for the efficiency of the algorithm. For example assume the representation for a substitution S to be a set of pairs $\{\langle t, \tau\rangle \mid St = \tau \neq t\}$. In the previous example any most general unifier of the terms τ_3 and τ_4 takes exponential space. The problem can be solved by keeping the substitution in sequential form, i.e., as a sequence of pairs that have to be applied to get the complete result. For instance a most general unifier for τ_3 and τ_4 could be represented by the sequence

$$\langle t_n, \kappa'(t_{n-1}, t_{n-1})\rangle, \langle t_{n-1}, \kappa'(t_{n-2}, t_{n-2})\rangle, ..., \langle t_2, \kappa'(t_1, t_1)\rangle$$

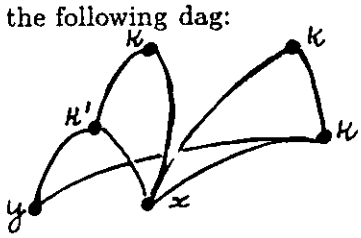
Terms are represented by directed acyclic graphs, *dags*. (The symbols u and v are used to refer to nodes of dags.) This representation is common in most term algorithms and has the advantage of sharing common subexpressions. The dag corresponding to a term has size linear in the size of the term and can be built in linear time. For example the term

$$\tau = \kappa(\kappa'(y, x), x)$$

has the following representation:



For pairs of terms, e.g., $\langle \tau, \kappa(x, \kappa(y, x))\rangle$, subexpressions common to the two terms are shared as well producing the following dag:



Nodes of the dag will be referred to as either *function* or *variable* nodes. For the description of the algorithm we need some properties of equivalence relations between nodes of a dag.

Definition 16: We say that an equivalence relation $\cong$ between nodes of a dag is *valid*, if the following three conditions are satisfied.

1. All the function nodes in an equivalence class are labelled by the same function symbol. ($\cong$ is *coherent.*)
2. If two function nodes are equivalent then their corresponding sons are equivalent. ($\cong$ is *simplifiable.*)

3. The equivalence classes can be partially ordered by the partial order on the dag . That is: the relation $<$ defined by $[u]_{\cong} < [u']_{\cong}$ if there is an edge from u to u', is a strict partial order. ($\cong$ is *acyclic.*) $\square$

The importance of the previous definition is shown by the following theorem.

Theorem 17: The terms represented by two nodes of a dag, u and u', are unifiable if and only if there is a valid equivalence relation $\cong$ with $u \cong u'$. What's more in that case there is a unique minimal such relation (that corresponds to the most general unifier). $\square$

Let P be a coherent equivalence relation on nodes of a dag. $S(P) = \{\langle v, v' \rangle\}$, where v and v' are the k-th sons of a pair of function nodes $\langle u, u' \rangle \in P$ for some k. If $P \cup S(P) = P$ then P is simplifiable.

Given a set of pairs of terms $\{\langle \tau_i, \tau_i' \rangle \mid 1 \leq i \leq n\}$ which we want to unify, we first build the dag representation of the terms τ_i, τ_i'. Let $P = \{\langle u_i, u_i' \rangle \mid 1 \leq i \leq n\}$ be the set of pairs of nodes representing the corresponding terms τ_i and τ_i'. The abstract description of the algorithm suggested by the previous theorem is as follows:

1. Let $\cong$ be the smallest equivalence relation on the nodes of the dag containing the pairs P. $\cong_0 \leftarrow \cong$ and $j \leftarrow 0$.
2. If $\cong_j$ is coherent and acyclic
   - then $\cong_{j+1} \leftarrow (\cong_j \cup S(\cong_{j+1}))$ and $j \leftarrow j + 1$.
   - otherwise exit with *failure.*
3. If $\cong_{j+1} = \cong_j$
   - then exit with *success.*
   - otherwise go to step 2.

The termination of the algorithm is obvious as there are only a finite number of nodes in the dag and $\cong_j \subseteq \cong_{j+1}$. If the algorithm terminates with *success* in n steps, then $\cong_n$ is coherent, acyclic and simplifiable ($(\cong_n \cup S(\cong_n)) = \cong_n$) Moreover $\cong_n$ is the smallest simplifiable, coherent and acyclic relation containing P. The unifier of the two terms is represented by the equivalence classes of $\cong_n$ containing the variables of the two terms.

The equivalence relation is represented by undirected edges between equivalent nodes of the dag. However, representing all the pairs entails a number of edges quadratic in the number of nodes. A node of the class is then chosen as the representative of the class and edges are drawn only from that node to the others.

The minimal equivalence classes , w.r.t. $<$, of a valid equivalence relation, are called *root classes.* Such classes indeed contain only roots of the dag. Any non-empty valid relation clearly has at least one root class.

The following algorithm $Unify_L$ implements the previous abstract algorithm. $Unify_L$ uses *Propagate* whose inputs are: a dag d, and a set of nodes N of d. *Propagate* checks if the nodes in N are pairwise coherent and if so modifies the dag d adding undirected edges representing the equivalence relation $S(\{\langle u, u' \rangle \mid u, u' \in N\})$. *Propagate* also outputs pairs of the substitution if some of the nodes are variable nodes.

Propagate(N, d)
Let $N = \{u_1, ..., u_n\}$. If one of the nodes is a function node, assume it is u_1.

1. If some u_j is a function node with an associated symbol different from the one of u_1 then exit with *failure*.
2. $i \leftarrow 2$.
3. If $i > n$ then exit with d.
4. If u_i is a function node
   - then for k from 1 to the outdegree of u_1
     draw an undirected edge (in d) from v to v', where v is the k-th son of u_i and v' is the k-th son of u_1.
   - otherwise let t be the variable labelling u_i and τ the term corresponding to u_1; output the substitution pair $\langle t, \tau \rangle$.
5. $i \leftarrow i + 1$ and go to step 3.

The inputs to the unification algorithm are a dag d and a set of pairs P of nodes of d that have to be unified.

$Unify_L(P, d)$

1. Draw an undirected edge from u to u' (in d) for all $\langle u, u' \rangle \in P$.
2. If d is empty then exit with *success*.
3. If the relation represented by the undirected edges in d has a root class
   $R = \{u_1, ..., u_n\}$
   - then
     (a) $d \leftarrow$ *Propagate*(R, d), (this call may output, as a side effect, some substitution pair)
     (b) delete from d all the nodes in R and their outgoing nodes,
     (c) go to step 2.
   - otherwise exit with *failure*.

The test in step 3 supplies a check for acyclicity of the equivalence relation represented by the undirected edges of the dag d. If there is a cycle one of the equivalence classes is such that some

of its nodes are not roots of the dag. Therefore, when no other root class can be selected, the algorithm fails. Moreover, as we know that all the nodes in a root class are roots of the dag, once the relation is propagated to the sons of such a class no further node can be added to that class and its nodes can be correctly deleted.

The following facts provide a rough analysis of the algorithm that shows its linearity with the size of the input.

1. The total time spent in the propagation of the equivalence relation (all calls of *Propagate*) is of the order of the number of directed edges plus nodes of the initial dag. That is, the total number of undirected edges created during the execution of $Unify_L$ is, by the representation of the chosen equivalence relation, bounded by the number of directed edges of the graph.
2. Assuming that each node maintains a list of pointers to its immediate predecessors, the selection of a root class can be done automatically. Starting from any node of the dag we climb up to a root. We choose that node as the representative of the equivalence class and from that propagate the equivalence in its class iterating the climbing if some of the nodes are not roots. If a node is visited more than once then the relation is cyclic, otherwise a root node is found and then after propagation its nodes are deleted.

A more detailed and concrete version of the algorithm is presented in the original paper by Paterson and Wegman, [35].

Other almost linear unification algorithms have been described. In particular Huet in [20] gives a very similar algorithm whose running time is $O(nG(n))$ where G is the inverse of the Ackermann function. Huet does not embed the test for cyclicity in the choice of the equivalence class to be propagated. The selection is simply sequential. The test is performed at the end of the cycle when the complete relation has been generated. (Note that the correctness of the previous linear algorithm depends on the acyclicity of the relation.) Finally Martelli and Montanari, [27], present an algorithm with the same running time as Huet's which differs basically by performing the propagation not only to the immediate successors of equivalent nodes but even deeper. Unlike the linear algorithm, in this case the selection of the class to propagate does not rule out the possibility of adding new elements to the class, later on. The same authors have given a linear algorithm that is very similar to the one presented here, but more complex in the data structures used in the implementation.

Acknowledgements

This paper was written, as part of an Area Qualifier, under the guidance of Daniel Leivant. I thank him for the encouragement and the helpful insights. Steve Brookes, also, gave suggestions that led to improvements in the presentation of the material. Finally, thanks to Roberto Minio for the help with typesetting.

REFERENCES

[1] H. Barendregt, M. Coppo, and M. Dezani. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 1983.

[2] H.J. Boehm. Personal comunications. Courtesy of the ARPA net.

[3] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. Hope: An Experimental Applicative Language. In *Symposium on LISP and Functional Programming*. ACM, 1980.

[4] L. Cardelli. Basic Polymorphic Type Checking. *Polymorphism. The ML/ LCF/ Hope Newsletter*, January 1985.

[5] M. Coppo. An Extended Polymorphic Type System for Applicative Languages. In P.Dembinsky (editor), *Mathematical Foundations of Computer Science*. Springer-Verlag, 1980.

[6] M. Coppo, M. Dezani, and B.Venneri. Functional Characters of Solvable Terms. *Zeitschrift fur Math. Logik und Grund. der Matematik*, 1981.

[7] M. Coppo, and E. Giovannetti. Completeness Results for a Polymorphic Type System. In G. Ausiello, and M. Protassi (editors), *CAAP'83, Trees in Algebra and Programming*. Springer-Verlag, 1983.

[8] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*. Volume 2. Noth-Holland, 1972.

[9] L. Damas, and R. Milner. Principal Type Schemes for Functional Programs. In *Symposium on Principles of Programming Languages*. ACM, 1982.

[10] D. Demers, and J. Donahue. *Report on the Programming Language Russel*. Tecnical Report TR 79-371, Computer Science Dept., Cornell University, 1979.

[11] A.J. Demers, and J.E. Donahue. Data Types, Parameters and Type Checking. In *Symposium on Principles of Programming Languages*. ACM, 1980.

[12] J. Fairbarn. Ponder and its Type System. *Polymorphism. The ML/ LCF/ Hope Newsletter*, April 1983.

[13] S. Fortune, D. Leivant, and M. O'Donnell. The Expressiveness of Simple and Second Order Type Structures. *Journal of ACM*, to appear.

[14] M. M. Fokkinga. On the Notion of Strong Typing. In J. de Bakker, and van Vliet (editors), *Algorithmic Languages*, 1981.

[15] M.J. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF*. Springer-Verlag, 1979.

[16] C.T. Haynes. A Theory of Data Type Representation Independence. In G. Kahan, D.B. MacQueen, and G. Plotkin (editors), *Semantics of Data Types*. Springer-Verlag, 1984.

[17] R. Hindley. The Principal Type Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Society* ,1969, pp. 29-60.

[18] R. Hindley. The Completeness Theorem for Typing λ-Terms. *Theoretical Computer Science* ,1983, pp. 1-17.

[19] M. Hook. Understanding Russel- A First Attempt. In G. Kahan, D.B. MacQueen, and G. Plotkin (editors), *Semantics of Data Types*. Springer-Verlag, 1984.

[20] G. P. Huet. *Résolution d' Équations dans des Languages d' Ordre 1,2,...,ω*. Thèse de Doctorat d' État, Université Paris VII,1976.

[21] G. P. Huet, and J. M. Hullot. *Proof by Induction in Equational Theories with Constructors*

[22] D. Leivant. Polymorphic Type Inference. In *Symposium on Principles of Programming Languages*. ACM, 1983.

[23] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *Communication of ACM*. 1977.

[24] N. J. McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure*. PhD Thesis, Syracuse University, 1979.

[25] N. J. McCracken. The Type-Checking of Programs with Implicit Type Structure. In G. Kahan, D.B. MacQueen, and G. Plotkin (editors), *Semantics of Data Types*. Springer-Verlag, 1984.

[26] D.B. McQueen, and R.Sethi. A Semantic Model of Types for Applicative Languages. In *Symposium on LISP and Functional Programming*. ACM, 1982.

[27] A. Martelli, and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 1982.

[28] D.C.J. Matthews. Poly report. *Polymorphism. The ML/ LCF/ Hope Newsletter*, April 1983.

[29] D. McQueen, G. Plotkin, and R. Sethi. An Ideal Model for Recursive Polymorphic Types. In *Symposium on Principles of Programming Languages*. ACM, 1984.

[30] R.Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer Systems Sciences*, 1978.

[31] R.Milner. *A Proposal for Standard ML*. University of Edinburgh Internal Report. 1983.

[32] R.Milner. How ML Evolved. *Polymorphism. The ML/ LCF/ Hope Newsletter*, April 1983.

[33] J. C. Mitchell. Coercion and Type Inference (Summary). In *Symposium on Principles of Programming Languages*. ACM, 1984.

[34] J. C. Mitchell. Type Inference and Type Containment. In G. Kahan, D.B. MacQueen, and G. Plotkin (editors), *Semantics of Data Types*. Springer-Verlag, 1984.

[35] M.S. Paterson, and M.N. Wegman. Linear Unification. In *Proceedings of the 8th ACM Symposium on Theory of Computing.* Hershey, 1976.

[36] D. Prawitz. Ideas, and Results in Proof Theory. In Fenstand (editor), *Proceedings of the II Scandinavian Logic Symposium.* North-Holland, 1971, pp.235-307.

[37] J. C. Reynolds. Towards a Theory of Type Structures. In *Programming Symposium (Colloque sur la Programmation)* Springer-Verlag, 1974.

[38] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In *IFIP Congress '83*, 1983.

[39] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of ACM*, 1965.

[40] M. Shaw (editor). *Alphard: Form and Content.* Springer-Verlag, 1981.

[41] M. Wand. A Types-as-Set Semantics for Milner-Style Polymorphism. In *Symposium on Principles of Programming Languages.* ACM, 1984.

[42] B. Wegbreit. The Treatment of Data Types in EL1. *Communication of ACM.* 1974.