

SUBTYPE INFERENCE IN FUNCTIONAL LANGUAGES



by

AH Dekker BSc (Hons)

in the Department of Electrical Engineering and Computer Science

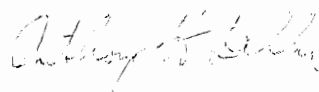
Submitted in fulfilment of the requirements
for the degree of

Doctor of Philosophy

University of Tasmania

July 1989

This thesis contains no material which has been accepted for the award of any other higher degree or graduate diploma in any tertiary institution. To the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference has been made in the text of the thesis.

A handwritten signature in cursive script, appearing to read 'Anthony H. Dekker'.

AH Dekker

ABSTRACT

This thesis describes techniques for efficiently performing polymorphic type inference for functional programming languages in the presence of subtype relationships. We permit subtype relationships which are based on implicit coercion functions between primitive types or type constructors, such as between integers and reals or between lists of arbitrary type and sets of the same type. Coercions between different kinds of functions are also permitted. In particular, finite database functions can be coerced to general functions. This makes functional languages useful as database query and update languages.

In the first chapter we describe basic notation, and define a *subtype ordering* to be a partial order on type variables and type constructors, satisfying certain conditions which ensure that the inequalities in a subtype ordering always have a solution. We define a number of operations for manipulating subtype orderings, which exploit the representation of subtype orderings as transitively reduced directed acyclic graphs. Generic operations such as addition are given a *type scheme* containing a subtype ordering which constrains the bound variables. *Colouring* of type variables imposes a further constraint, which allows a type scheme to be given to the equality function.

The second chapter defines a simple functional language and gives inference rules which type-annotate expressions. A number of additional rules are derived, including a rule for applying a substitution to a typing assertion. We provide a rewrite rule semantics for typed expressions, and show that this preserves typing information and is Church-Rosser. Since coercions produce a unique result and commute with primitive operations, including equality, we conjecture that type-annotating an expression is semantically unambiguous.

In the third chapter we give a sound and complete, but naive, type inference algorithm which is inefficient, since intermediate subtype orderings rapidly become large. Standardisations and simplifications, which reduce the size of subtype orderings without loss of generality, lead to a more efficient type inference algorithm, which is quartic-time for expressions, but which we expect to be linear for typical user programs.

The fourth chapter applies this work to database query and update, using the Functional Data Model. A case study involving a hospital database shows the utility of subtype inference.

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Phil Collier and Clem Baker-Finch for their advice and for discussions on parts of this work. Ben Lian, David Wright, Andrew Martin, and especially Ed Kazmierczak also offered many helpful suggestions. Thanks also to the many friends who provided moral support, and to Toni Hickey for infinite patience in typing a difficult manuscript.

Finally I thank the Australian Computer Research Board and the Australian Telecommunications and Electronics Research Board for their financial support.

TABLE OF CONTENTS

INTRODUCTION

Why Type Inference?	1
Why Subtypes?	1
Subtypes and Databases	4
Subtype Inference : Existing Work	5
Extensions and Improvements	7
Thesis Structure	9

1	PRELIMINARY NOTATION	11
1.1	Type constructors	12
1.2	Introducing the Subtype Ordering	13
1.3	Constant Identifiers	15
1.4	The Type Structure	16
1.5	Type Substitutions	21
1.6	Subtype Orderings	23
1.7	Matching Types	27
1.8	Union of Subtype Orderings	32
1.9	Splittings and Graphs	35
1.10	Operations on Subtype Orderings	38
1.11	Type Schemes and Assumption Sets	45

2	A FUNCTIONAL LANGUAGE AND TYPE SYSTEM	49
2.1	The Functional Language	49
2.2	Replacements and Replications	53
2.3	The Type Inference System	55
2.4	Derived Inference Rules	60
2.5	A Rewrite — Rule Semantics	63
2.6	A Church-Rosser Result	68
2.7	Weak Head Normal Form and Böhm Trees	70
2.8	Properties of Böhm Trees	74

3	ALGORITHMS	77
3.1	Type Inference Subroutines	78
3.2	A Type Inference Algorithm	80
3.3	Standardisations	85
3.4	Satisfying Subtype Orderings	91
3.5	Simplifications	93
3.6	An Improved Type Inference Algorithm	101
3.7	Efficiency Considerations	105

4	DATABASE APPLICATIONS	107
4.1	A Subtype Hierarchy for Database Use	109
4.2	Inverses of Database Functions	110
4.3	Composition of Database Functions	111
4.4	Entities and Multiple Inheritance	112
4.5	Case Study — A Hospital Database	114
4.6	Query in Our Hospital Database	115
4.7	Update in our Hospital Database	115
4.8	General Remarks	117
5	RELATED WORK	119
5.1	A Comprehensive Theory of Coercions	120
5.2	Inclusion Polymorphism with Intersection and Union	121
5.3	Subtypes and Generality	125
5.4	Subtypes and Object Oriented Languages	126
5.5	The Mitchell Approach	127
5.6	Other Approaches	131
5.7	Generic Functions, Type Classes and Coloured Types	132
6	CONCLUSIONS AND FURTHER WORK	134
7	REFERENCES	138
	Appendix — Table 1 : Rewrite Rules for Programs	144

INTRODUCTION

Why Type Inference?

The benefits of static strong typing of computer programs have been known since the advent of Algol 60. These include the encouragement of disciplined programming, and an increase in security due to the elimination of a large class of erroneous programs. Many recent programming languages, such as Pascal and Modula 2, also use static typing, because its advantages are considered to outweigh the restrictions it imposes on the programmer. In the field of functional programming languages, the benefits of static typing were made even more convincing by the development of a *type inference algorithm* by Milner (1978), independently of earlier work by Hindley (1969).

Milner's type inference algorithm (which he called *W*) deduces all required type information from the text of a program, so that the programmer need not give explicit type information. The security of static typing is thus retained, with the added convenience of more concise programs. Another major consequence of the algorithm is that the programmer can write *polymorphic* functions, ie functions which operate on arguments of many different types. For example, the function which calculates the length of a list operates on lists of arbitrary type, and thus has an infinite number of types of the form $list\ \tau \rightarrow nat$ (where *nat* is the type of the natural numbers). This is represented by giving the length function the type scheme $\forall \alpha. list\ \alpha \rightarrow nat$, and all possible types for the function can then be obtained by substituting types for the type parameter α . Damas and Milner (1982) provide proofs that the type schemes inferred by Milner's algorithm are the most general ones. This lays a solid theoretical foundation for the algorithm. The practical importance of Milner's algorithm can be seen in the way that it has been used to great advantage in functional languages such as ML and MirandaTM.

Why Subtypes?

In spite of the obvious advantages of Milner's type inference algorithm, it still has a number of disadvantages. One of these is the inability to use, for example, expressions of integer type in a context where values of real type are expected. Such an ability, involving the implicit insertion of a type coercion function, was one of the great advantages that Algol 60 and its descendents had

over Fortran, allowing the programmer to write $pi + 3$ instead of $pi + \text{float}(3)$. Of course, mixing integer and real values in arithmetic expressions can also be permitted by combining all numeric types into a single type *number*, and using run-time tests to see what sort of value is present. However, this alternative is not only less efficient, but decreases security, in that some type errors may not be detected, even if a program is run many times. We consider essential the ability to distinguish integer and real types, and still mix integer and real values freely in arithmetic expressions.

Since the publication of Milner's algorithm, many researchers have explored ways of loosening the restrictions that it imposes. We will examine some of these later in the thesis. Surveys of some of this work are given by Leviant (1983), Reynolds (1985) and Cardelli and Wegner (1985). Although the type systems that have been proposed are not necessarily compatible, they all involve some extension of the notion of polymorphism.

A useful classification of various kinds of extended polymorphism is given by Cardelli and Wegner (1985), who distinguish *parametric* polymorphism, *inclusion* polymorphism, *coercion* and *overloading*. However, these authors do not answer the question : which of these forms of polymorphism can be combined with type inference?

Parametric polymorphism includes the notion of polymorphism inherent in Milner's algorithm, where polymorphic functions are introduced by a **let** construct. Because the algorithm operates by substituting types for the type parameters in type schemes, it suffers from the limitation that polymorphic functions cannot have polymorphic functions as arguments. This limitation can be removed by permitting type schemes to be substituted for type parameters. However, this leads to the second order typed lambda calculus (Reynolds 1974), for which no type inference algorithm has yet been found.

Inclusion polymorphism is based on containment relations between types, which leads to a notion of *subtype*. One type is said to be a subtype of another if all the values of the first type also have the second type. For example, the type *nat* is a subtype of *int*, since the natural numbers are properly contained within the integers. Inclusion polymorphism can be combined with parametric polymorphism by qualifying the parameters of a type scheme with inequalities. Cardelli and Wegner (1985) refer to such a qualification of type schemes as *bounded quantification*. The intention is that the types which are substituted for the type parameters must satisfy the inequalities. For example, we can

express the fact that addition is defined on integers, but that two natural numbers added together always give a natural number, by giving the addition operation the type scheme :

$$\forall \alpha : nat \leq \alpha \leq int. \alpha \rightarrow \alpha \rightarrow \alpha$$

Unfortunately, inclusion polymorphism is not powerful enough to describe the use of integers in contexts where real numbers are expected. In most computers, the set of integer representations is not a subset of the set of real number representations. Even in mathematics, formalisations of the real numbers, by such techniques as Dedekind cuts, represent real numbers as *sets* of rational numbers (Cohn 1981). In both cases, the best that we can say is that there is an implicit *coercion* function from integers to real numbers. This leads us to a notion of subtype which generalises inclusion polymorphism : one type is a subtype of another if there is a designated coercion function between the two types. One of the objectives of our thesis is to indicate that polymorphic type inference can be efficiently combined with this notion of subtype.

Implicit coercions are a powerful and useful feature. Like other powerful and useful features, they can be abused. For example, the language Algol 68 (Lindsey and van der Meulen, 1977) allows an astonishingly large range of implicit coercions between types, resulting in a baroque and unwieldy system. The solution to this problem is provided by Reynolds (1980), who provides a detailed mathematical treatment of coercions, and suggests that an implicit coercion between two types is acceptable exactly when it *commutes* with all functions defined on both the types. For example, the integer to real coercion commutes with addition in the sense that performing an integer addition on two numbers and coercing the result to a real gives the same result as coercing the two numbers to reals and performing a real addition. It is essentially this condition that permits mathematicians to treat the integers as if they were a subset of the real numbers, and to write expressions such as $1 + 2 + \pi i$ without ambiguity. We will restrict our attention to implicit coercions which satisfy this commutativity criterion.

The fourth, and most general, form of extended polymorphism is overloading, which refers to any other system where an expression may have several types. Overloading in its full generality cannot be combined with type inference, for theoretical reasons which we will discuss later in this thesis. However, restricted kinds of overloading are useful. In particular, the use of implicit

coercions requires a particular kind of overloading, which Reynolds (1980) refers to as *generic functions*. Such functions are expressed using a parameterised type, where the parameter ranges over a fixed set of primitive types. For example, addition is parameterised on the numeric types. Our type inference algorithm will permit those generic functions whose types can be expressed using the notation of bounded quantification. This includes many useful functions. For example, the function which adds together the elements of a list, which can be written in Miranda as :

$$\begin{aligned} \text{sum } [] &= 0 \\ \text{sum } (x : y) &= x + \text{sum } y \end{aligned}$$

can be assigned the type scheme $\forall \alpha : \text{nat} \leq \alpha \leq \text{real}. \text{list } \alpha \rightarrow \alpha$. Where coercions are not involved, we will also permit overloading of functions over a class of ‘coloured’ types. This permits us to give a type to the equality function, which often poses problems.

Subtypes and Databases

There is a second motivation for incorporating a notion of subtype into functional languages, namely to produce an integrated functional programming and database manipulation language, thereby applying the benefits of functional programming to the database arena. Within database systems we have a notion of subtype generated by containment relations between classes of real-world entities. Cardelli and Wegner (1985) study this notion of subtype within the framework of *inheritance* in object-oriented languages (Cardelli 1984, Albano et al 1985). In such languages, real-world entities are represented as tuples of attributes. Subtypes have more attributes than their supertypes, since they *inherit* attributes of their supertypes, and may have other attributes of their own.

Our interest lies in functional rather than object-oriented languages, and we feel that the Functional Data Model (Shipman 1981, Atkinson and Kulkarni 1984, Buneman et al 1982) is a more natural model for database systems than the object-oriented model, for reasons which we will discuss in chapter four of this thesis. The Functional Data Model also combines neatly with functional programming languages. It views a database as a set of entity types, which may have containment relations between them, and a set of finite functions, which are essentially special cases of binary relations. We can use inclusion

polymorphism to describe the containment relations between entity types, but coercions are required in order to use finite functions in contexts where general functions are expected. A novel feature of this thesis is to introduce such coercions between different kinds of functions.

Subtype Inference : Existing Work

Having identified the subtype relationships we are interested in, and having claimed that type inference in the presence of such subtype relationships is possible, we now turn to the question of a type inference algorithm. The first steps towards a suitable algorithm were taken by Mitchell (1984), who partially describes an algorithm in the style of algorithm *V* of Leivant (1983). Given an expression, Mitchell's algorithm will infer:

- A type-annotated expression, where each binding of a type variable, and each subexpression, is annotated with a type.
- An assumption set which assigns types to program variables, and in which program variables occur at most once.
- A *coercion set* of inequalities between atomic types, which must hold if the expression is to be well-typed.

Type-annotated expressions in which a program variable is used with a type other than that with which it was bound, or in which an argument to a function has a type other than that which the function is expecting, generate type inequalities and mark places where implicit coercions would be inserted. For example, if $int \leq real$, then the two type-annotated expressions $\lambda x_{int}. f_{real} \rightarrow bool\ x_{real}$ and $g_{real} \rightarrow bool\ e_{int}$ would require coercions from *int* to *real*.

An advantage of Mitchell's (and Leivant's) style of algorithm is that substitutions need not be passed around, but can be applied directly to objects of interest, thus improving efficiency. Also, types for the two parts of an application can be inferred independently, permitting parallel implementations of the algorithm, which are of value if functional languages are to be implemented on parallel processors (Peyton Jones 1987).

Mitchell proves his algorithm is *sound* (correct) and *complete* (most general) with respect to a set of type inference rules. These rules are much like those of Damas and Milner (1982) which apply to Milner's algorithm, except that type schemes and the polymorphic *let* construct are not dealt with, and one new rule

is introduced. The new rule expresses the fact that if $\tau \leq \tau'$ is a consequence of the inequalities in a coercion set, and the expression e has type τ given that coercion set, then the expression also has type τ' . Put more simply, an expression can be coerced upwards to any supertype. While a significant step forward, Mitchell's algorithm suffers from a number of limitations, which we shall briefly consider :

The polymorphic **let** construct is not handled by Mitchell, and there are therefore no type schemes, and no bounded polymorphism. This makes the algorithm unsuitable for an actual programming language, where polymorphic definitions are essential (Peyton Jones 1987, Bird and Wadler 1988). There is also no description of how types are assigned to constants such as the numerals. This relates to the absence of type schemes, since some constants in functional programming languages, eg the null list, are polymorphic.

Mitchell restricts himself to coercions which 'can be viewed as set containment'. This suggests that coercions are assumed to be injective. Now, from the fact that *int* is a subtype of *real*, we can conclude that the functional type *real* \rightarrow *int* is a subtype of *int* \rightarrow *int*. In other words, functions such as *round* and *truncate* that convert reals to integers can be applied to integers alone. However, both of these functions (and a number of others) act as the identity function when applied to integers alone. Thus the coercion from *real* \rightarrow *int* to *int* \rightarrow *int* is not injective, assuming the natural notion of extensional equality is used. This seems to suggest that a more detailed examination of the semantics of coercions is required. In most of Mitchell's paper, coercions are not explicitly mentioned, and some parts seem to deal with inclusion polymorphism only. However, the type inference algorithm itself is very general, and will deal with non-injective coercions.

Mitchell also restricts himself to coercions which are logical consequences of coercions between atomic types, and his coercion sets are accordingly sets of inequalities between atomic types. Here again, his type inference algorithm is sufficiently general to deal with other forms of coercion, and we can extend coercion sets to other structures that induce partial orders (or preorders) on types. For example, Stansifer (1988) gives a similar algorithm, in the style of Milner's algorithm *W*, that deals with coercion functions which delete attributes from a tuple. Such coercion functions can describe inheritance in object-oriented languages. Thatte (1988) gives a semi-algorithm, also in the style of Milner's algorithm *W*, that deduces partial type information, in order to deal with heterogeneous data structures. This algorithm allows any type

whatsoever to be coerced to an atomic type Ω , which includes all values, and thus conveys the least possible amount of type information. The general form of Mitchell's algorithm is given explicitly by Fuh and Mishra (1988).

Another limitation of Mitchell's algorithm is that there is no notion of *consistency* of coercion sets. For example, we may apply the type inference algorithm to an expression, and deduce that it has type *int*, subject to a coercion set containing the inequality $real \leq int$. In other words, a function accepting only integers is applied to a real number at some point. Intuitively, one would wish this to be considered as a type error. We would therefore want a type inference algorithm which tested coercion sets to ensure that they were 'consistent' in some sense.

The final issue is one of efficiency. The coercion set inferred by Mitchell's algorithm grows very rapidly with the length of the input expression, since inequalities are added to it for each application and each use of a program variable. Now, checking the consistency of a coercion set seems to require a transitive closure, which is a cubic-time operation. Calculating minimal coercion sets requires a transitive reduct which is also cubic-time. This makes Mitchell's algorithm impractical for use in an actual language implementation. A technique is required that will reduce the size of a coercion set while retaining generality.

Extensions and Improvements

In this thesis we extend the work of Mitchell by addressing the limitations we have discussed. We introduce a limited **let** construct. As Milner (1978) indicates, the interaction between lambda abstractions and **let** constructs is the most difficult part of type inference. This is especially so in the presence of bounded polymorphism. For example, the type scheme $\forall \alpha : \alpha \leq \beta. \beta \rightarrow \alpha$ contains a free type variable, and we can apply substitutions to this type scheme. However, for substitutions such as $S\beta = \gamma \rightarrow \eta$, the result is dubious. We therefore follow the language MirandaTM by only allowing polymorphic **let** constructs at the 'top level'. We do not permit polymorphic **let** constructs inside lambda abstractions, and therefore we can ensure that all type schemes are *closed*. This avoids the problem of substitution, since substitutions do not affect closed type schemes. This decision allows us to combine Mitchell's algorithm with Milner's algorithm *W*. In particular, our type inference algorithm will accept as input an assumption set containing only closed type

schemes, relating to enclosing **let** constructs, and will produce as output an assumption set containing only types, relating to free program variables. This combined approach retains the advantages of Leivant's algorithm *V*, while providing a simple treatment of polymorphism. We will treat constants such as *nil* as being variables defined in an enclosing **let**. They will thus be assigned a type scheme. Constants such as the numerals which have atomic types are assumed to have a least type, which is the type that we will infer for them.

We note that coercions should commute with polymorphic functions, in order to avoid ambiguity. In particular, coercions should commute with polymorphic equality, so that coercions must be injective on types where equality is defined. However, we permit non-injective coercions on functions, since equality is not defined on functions. In fact, we use an extension of the concept of 'eqtype variable' in ML to ensure that applying equality to functions is a type error. This extension is expressed using our notion of 'coloured' types.

A major extension that we introduce is type inequalities not only on atomic types, but on *type constructors*. For example, the inequality $list \leq set$ allows a coercion from $list\ \tau$ to $set\ \tau$ for any type τ . Such a coercion, which would remove duplicates and 'forget' ordering, is obviously non-injective. Its introduction would therefore mean that the polymorphic equality operator could not apply to lists and sets, and we would need two functions *eqlist* and *eqset*. However, for some applications this may be a worthwhile trade-off. Coercions of this nature are a special case of *implementations* in algebraic data types (Ehrig et al 1982).

We formalise coercions on type constructors by assigning a *rank* to type constructors based on the number of arguments. Atomic types are assigned rank 0, *list* and *set* are assigned rank 1, the product constructor is assigned rank 2, and the functional type constructor \rightarrow is given the special rank *. We extend coercion sets to families of sets of inequalities indexed on rank. Inequalities of rank * permit coercions between different sorts of functions. In particular, we can coerce functional types that have a finite-function constructor to general functions, thus satisfying the requirement for integrating functional programming languages with the Functional Data Model. Inequalities of rank * also offer the possibility of integrating strictness analysis (Burn et al 1986) with type inference by introducing two subtypes of general functions, namely strict and non-strict functions. This issue is examined further in Wright and Dekker (1988).

We restrict the predefined hierarchy of subtypes to a ‘forest’ of semilattices. This does not appear to eliminate any useful examples of type hierarchies, and permits us to define an easily checkable set of conditions necessary and sufficient for a coercion set to be consistent. Our type inference algorithm will check these conditions, and only return consistent coercion sets. We refer to such consistent coercion sets as *subtype orderings*.

We introduce two kinds of operations on subtype orderings which simplify the subtype ordering without loss of generality. We call these operations *standardisations* and *simplifications*. By applying these operations during type inference we can ensure that inferred subtype orderings grow very slowly in size. This solves the efficiency problem of Mitchell’s algorithm and makes subtype inference feasible for actual language implementations.

Thesis Structure

The first chapter of this thesis defines basic notation and gives an example type hierarchy. We define a subtype ordering to be a partial order on type constructors and type variables satisfying certain conditions. These conditions ensure that a subtype ordering is always *satisfiable*, in the sense that we can find a mapping from type variables to type constructors such that the inequalities remain true. This formalises the notion of a ‘consistent’ coercion set. We define a matching relation on types, similar to that of Mitchell, where two types match if they have the same ‘shape’. This allows us to generalise unification to an algorithm which calculates a minimal substitution that causes two types to match while remaining consistent with the inequalities in a coercion set. We also define a number of other operations that manipulate subtype orderings, exploiting the obvious representation of subtype orderings as directed acyclic graphs. Finally we discuss type schemes, assumption sets, and generic instances of type schemes.

Our intention is for our subtype inference algorithm to apply to a Miranda-like language. However, for simplicity we work with a simpler functional language involving application, lambda abstraction, a polymorphic *let*, and predefined functions which perform case analysis. The techniques outlined in chapters 4 to 7 of Peyton Jones (1987) indicate how a Miranda-like language can be translated to such a simpler form. In our second chapter, we define the syntax of our simple language and give type inference rules which type-annotate expressions. We give some example derivations using these rules, and derive

some useful new rules. We follow this with a rewrite-rule semantics, and show that it is Church-Rosser, and preserves types. A discussion of head normal form and Böhm trees allows us to prove that coercions produce a unique result, are injective on equality types, and commute with polymorphic functions. These are the required conditions for implicit coercions to be unambiguous.

Chapter three defines two type inference algorithms for our subtyping system. The first algorithm is not efficient, but is proved to be sound and complete. We then introduce standardisations and simplifications, which reduce the size of a subtype ordering without loss of generality. This leads us to an algorithm which finds a solution to the inequalities in a subtype ordering. This acts as a kind of abstract compilation. It also permits us to define an efficient type inference algorithm, which is still sound and complete. Several examples indicate that intermediate subtype orderings are small, and final subtype orderings usually contain at most one type variable.

The fourth chapter examines database applications. We use type constructors of rank $*$ to distinguish finite (database) functions and injective finite functions. This allows us to write type schemes for database operations such as *inverse*. We show how to describe inheritance in the Functional Data Model, and sketch a database query and transaction-based update language, by giving a substantial case study. The case study involves a hospital database model, and shows that our subtyping system allows a concise, clean and powerful functional programming and database language.

Our fifth chapter examines related work on subtype inference in more detail, and explores the connections with our approach. Much of this work has been developed concurrently with the work presented here. In particular, Fuh and Mishra (1988, 1989) also describe techniques for subtype inference based on extending Mitchell's work, developed independently from our approach. However, there are a number of close correspondences which suggest avenues for further improvements, by combining the two approaches.

We close this thesis with a general conclusion. Due to the high technical content of much of this thesis, the reader interested in a quick overview should read the introductions to chapters 1 and 2, sections 2.3 and 2.4, the introduction to chapter 3, and sections 3.2 and 3.6.

1 PRELIMINARY NOTATION

In this chapter we define basic notation relating to types, subtype orderings and type schemes. Section 1.1 introduces *type constructors* and gives a simple example of a type hierarchy which we will use throughout the thesis. Section 1.2 introduces the concept of a *subtype ordering*, and gives the restrictions which subtype orderings must satisfy. Our example type hierarchy satisfies these conditions. Section 1.3 introduces *constant identifiers*, such as numerals, which may have several types. We give conditions which ensure that constant identifiers have least types.

In section 1.4 we define *types*, *coloured types*, and *occurrences* of type variables in types. We show how coloured types allow generic functions to be defined. Section 1.5 introduces *type substitutions* and defines an object X to be *minimal* satisfying certain conditions if every other object satisfying those conditions has the form SX for some substitution S . This concept of minimality is used throughout. Section 1.6 introduces subtype orderings containing type variables, and gives necessary and sufficient conditions for a subtype ordering to be *satisfiable*. Section 1.7 introduces the *matching* relation between types and defines a substitution to be *shape-consistent* with respect to a subtype ordering if it preserves matching between subtype and supertype. This allows us to define an algorithm MATCH which finds a minimal substitution matching two types and MATCH2 which finds a minimal substitution matching two types which is shape-consistent with respect to a given subtype ordering. The algorithm MATCH2 is closely related to the algorithm MATCH proposed by Mitchell (1984).

Section 1.8 defines an operation UNION which combines a number of subtype orderings, returning a substitution and a new subtype ordering. The substitution combines cycles into a single element, so that the subtype ordering produced is a partial order. We define minimality on subtype ordering/substitution pairs, and show that UNION produces a minimal result. Section 1.9 examines the implementation of subtype orderings as transitively reduced acyclic graphs. In section 1.10 we define four operations on subtype orderings, namely ORDER, APPLYSUBST, ENRICH and SUBEN. The operation ORDER is defined in terms of an auxiliary operation ORDERSET, and creates a minimal subtype ordering/substitution pair which implies an inequality between two matching types. The APPLYSUBST operation calculates the minimal pair resulting from applying an arbitrary substitution to a subtype ordering (the substitution has to be extended to preserve

shape-consistency), and the ENRICH operation calculates the minimal pair resulting from enriching a subtype ordering by a set of inequalities between arbitrary types. We show that ORDER is a special case of ENRICH, and that any sequence of ENRICH and APPLYSUBST operations can be expressed by a single APPLYSUBST followed by a single ENRICH. This motivates the SUBEN operation which combines APPLYSUBST and ENRICH.

Section 1.11 introduces *type schemes* and *assumption sets*. Unlike Milner (1978), we ensure that types and type schemes are completely disjoint, by allowing $\forall.\tau$ as a type scheme with no bound variables. This allows us to distinguish program variables bound by **let** and program variables bound by lambda abstraction. We define *generic instances* of type schemes to be subtype ordering/type pairs which result from instantiating bound variables. We also define the operation *gen* which produces a type scheme from a type and a subtype ordering, provided type variables do not occur in a given assumption set.

1.1 Type Constructors

Our type structure, as discussed above, is based on a family of disjoint sets of *type constructors* Δ_r , where the *rank* r is a non-negative integer or $*$. The type constructors of rank 2 which we will consider are *product* (\times) and *sum* ($+$). We will have the two type constructors *list* and *optional* of rank 1. For each type τ , the values of type *optional* τ can be either the constant *absent*, or of the form *present* (v), where v is a value of type τ . The type *optional* τ is useful for explicitly describing partially defined functions, such as head-of-list, to which we would assign the types *list* $\tau \rightarrow \text{optional } \tau$, for each τ . We consider *optional* τ to be a subtype of *list* τ , with *absent* coercing to the empty list, and *present* (v) coercing to the one-element list $[v]$. This provides an example of a coercion between type constructors.

The primitive types *nat*, *int*, *bool*, *atom* and *void* provide examples of type constructors of rank 0. We consider *nat* to be a subtype of *int*, and *nat*, *int* and *bool* to be subtypes of *atom*. We provide two type constructors of rank $*$, namely the usual function type constructor (\rightarrow) and a type constructor for constant functions (\supset). For example, the term $\lambda x.e$ where e has type τ and x does not occur in e , can be given both the types $\tau' \rightarrow \tau$ and $\tau' \supset \tau$, where τ' is any type. We consider the type $\tau' \supset \tau$ of constant functions to be a subtype of the type $\tau' \rightarrow \tau$ of all functions, thus providing an example of a coercion between type constructors of rank $*$.

We define the *closed types* to be of the form :

$$\bullet \quad \delta_r \bar{\tau}$$

where δ_r is a type constructor of rank r , and $\bar{\tau}$ is a vector of closed types of corresponding length, ie of length r if r is a non-negative integer, and of length 2 if $r = *$.

We will write the type constructors \times , $+$, \rightarrow and \supset as infix operators, with \times and $+$ having precedence over \rightarrow and \supset , and with all four constructors associating to the right. We also assume that *list* and *optional* bind more tightly than other type constructors. Some example closed types are :

- int
- $void + bool$
- $list (nat \rightarrow bool \times int)$

To each closed type τ there corresponds a set of *values* $V(\tau)$, to be discussed further in section 2.5. For example, $V(nat)$ contains the natural numbers, $V(list\ bool)$ contains lists of booleans, and $V(int \times bool)$ contains pairs whose first component is an integer, and whose second is a boolean.

1.2 Introducing the Subtype Ordering

Definition 1.2.1

For each r , the set Δ_r of type constructors, is assumed to have a partial order relation \ll_r , representing the subtype relationship. We omit the subscript when the rank r is obvious from context. We denote by \ll -relatedness the family of equivalence relations generated by the reflexive, transitive, symmetric closure of the \ll_r .

We enforce two restrictions on the partial orders \ll_r :

- We require that if two elements of Δ_r have a lower bound, then they have a greatest lower bound (*glb*), and if they have an upper bound, then they have a least upper bound (*lub*).

- We require that each «-related equivalence class has either lub or glb defined on all pairs of its elements. In other words, it is a semilattice, and the subtype ordering is a ‘forest’ of semilattices.

These two restrictions eliminate many pathological cases of subtype hierarchies. The first restriction, which follows Reynolds (1985), ensures that all expressions have *least* types. The second restriction ensures that subtype orderings containing type variables (definition 1.6.1) will always be *satisfiable*.

The family of partial orders « for our example type constructors is given in Figure 1. For example, we will write $\text{nat} \ll \text{int}$. Note that we have six «-related equivalence classes, namely $\{\text{nat}, \text{int}, \text{bool}, \text{atom}\}$, $\{\text{void}\}$, $\{\text{optional}, \text{list}\}$, $\{\times\}$, $\{+\}$ and $\{\supset, \rightarrow\}$. The first of these has lub defined on all pairs of its elements, while the others have both lub and glb defined on all pairs of their elements.

Definition 1.2.2

We extend « to the least partial order on closed types satisfying :

- $\delta_r \bar{\tau} \ll \delta_r' \bar{\tau}'$ if and only if $r \neq *$,
 $\delta_r \ll_r \delta_r'$, and $\tau_i \ll \tau'_i$ for $1 \leq i \leq r$,
 where $\bar{\tau}$ and $\bar{\tau}'$ are vectors of length r .
- $\delta^* \tau_1 \tau_2 \ll \delta'^* \tau'_1 \tau'_2$ if and only if
 $\delta^* \ll \delta'^*$, $\tau'_1 \ll \tau_1$ and $\tau_2 \ll \tau'_2$

Note that the type constructors of rank $*$ are *monotonic* in their second argument, and *antimonotonic* in their first argument, while other type constructors are monotonic in all arguments. This is because functions can be coerced either by coercing the result (giving a ‘larger’ result), or by coercing the argument (accepting a ‘smaller’ argument). The following examples illustrate this :

- $\text{void} \times \text{nat} \ll \text{void} \times \text{int}$
- $\text{bool} \supset \text{int} \ll \text{bool} \rightarrow \text{int}$
- $\text{int} \rightarrow \text{nat} \ll \text{nat} \rightarrow \text{nat} \ll \text{nat} \rightarrow \text{int}$
- $\text{optional nat} \ll \text{list nat} \ll \text{list int}$

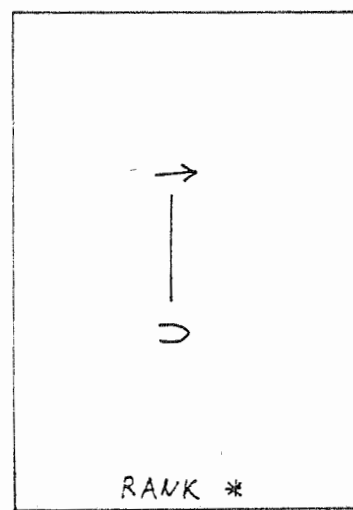
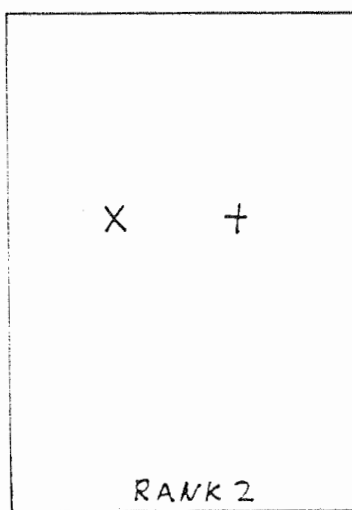
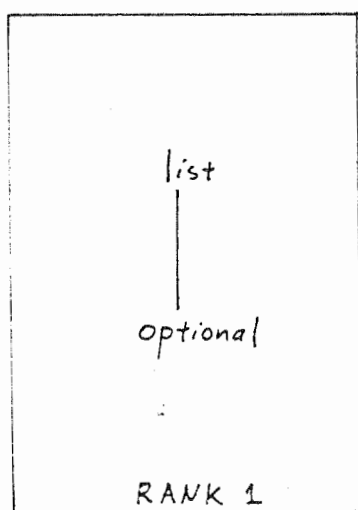
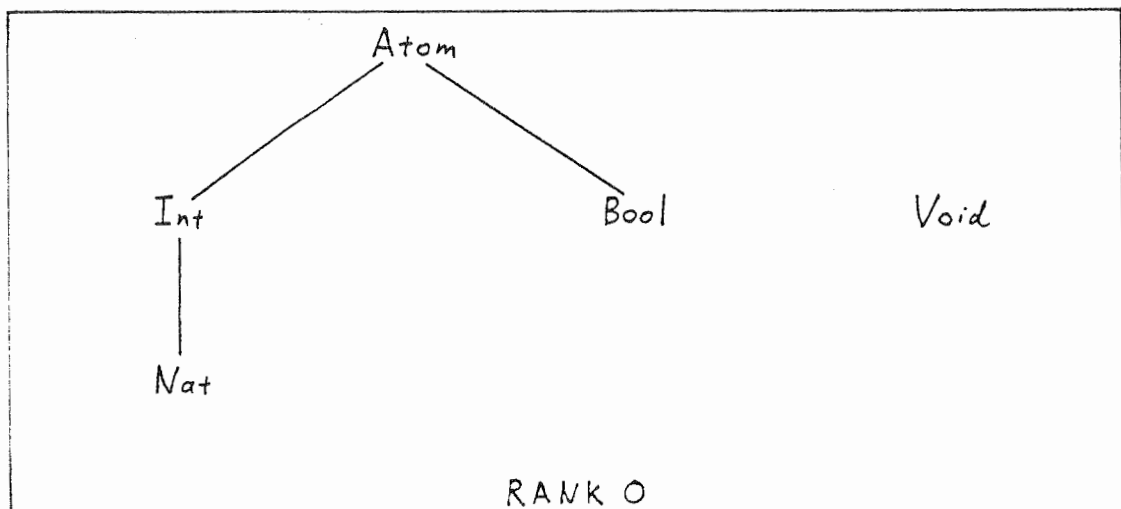


Figure 1 - The partial order \ll on type constructors

Our intention is that if $\tau \ll \tau'$ then there is a unique coercion function from $V(\tau)$ to $V(\tau')$. We require these coercion functions to commute with polymorphic functions, in the sense of Reynolds (1980). This will be discussed further in Section 2.7.

1.3 Constant Identifiers

Definition 1.3.1

The type constructors of rank 0 are also referred to as *primitive types*. Corresponding to each primitive type π , there is a set $K(\pi)$ of *constant identifiers*, which form part of our functional language syntax. (The remainder of our language syntax is defined in Section 2.1).

We require two conditions :

- if two primitive types ρ and π have no glb under \ll , then $K(\rho)$ and $K(\pi)$ are disjoint.
- for each primitive type π , there is an injective function P_π from $K(\pi)$ to $V(\pi)$.

The first condition ensures that pairs of possible types for a constant identifier have glbs, and thus by proposition 1.3.2, all constant identifiers have a least type. The second condition says that each constant identifier denotes a value distinct from that of other constant identifiers.

If we make the further restriction that $K(\rho)$ is contained in $K(\pi)$ wherever $\rho \ll \pi$, then we can construct simple term models for our functional language, as in section 2.5. However, this further restriction is not necessary in the general case.

As examples, consider :

- $K(nat) = \{0, 1, 2, \dots\}$
- $K(bool) = \{\text{true}, \text{false}\}$
- $K(void) = \{\text{unit}\}$
- $K(int) = \{\dots, -2, -1, 0, 1, 2, \dots\}$
- $K(atom) = \{\text{true}, \text{false}, \dots, -2, -1, 0, 1, 2, \dots\}$

Note that 0, 1, etc ambiguously denote either natural numbers, integers or atoms.

Proposition 1.3.2

Let c be a constant identifier such that $c \in K(\pi_i)$ for $1 \leq i \leq n$. Then there is a least π_j such that $c \in K(\pi_j)$, ie there is a j , $1 \leq j \leq n$, such that $\pi_j \ll \pi_i$ for all $1 \leq i \leq n$.

Proof: From the first condition in definition 1.3.1 and the restrictions in definition 1.2.1. \square

1.4 The Type Structure

Definition 1.4.1

We assume a family of disjoint sets of *type variables* TV_r for each rank r . We can then define *types* to be of the form :

$$\bullet \quad \delta_r \bar{\tau} \text{ or } \alpha_r \bar{\tau}$$

where δ_r is a type constructor of rank r , α_r is a type variable of rank r , and $\bar{\tau}$ is a vector of types of corresponding length.

Note that the types which do not contain type variables are exactly the closed types defined above. Effectively the type variables of rank 0 denote unknown types, while the type variables of other ranks denote unknown type constructors. For example, if α and β are type variables of rank 0, θ is a type variable of rank 1, and ϕ is a type variable of rank 2, then the following are types :

- α
- $nat \times \beta$
- $\theta (bool)$
- $\phi (\alpha, \beta + int)$

We define the length $|\tau|$ of a type τ by :

- $|\delta_r \tau_1 \dots \tau_n| = 1 + |\tau_1| + \dots + |\tau_n|$
- $|\alpha_r \tau_1 \dots \tau_n| = 1 + |\tau_1| + \dots + |\tau_n|$

For example, the lengths of the four types above are 1, 3, 2 and 5 respectively.

Definition 1.4.2

We also assume a set of *colours*, at present including only *red* and *blue*, such that each type variable of rank 0 may be coloured with zero or more distinct colours. These colours are used to define several useful subsets of types, referred to as types of a given colour. We make the restriction that if the type τ has a given set of colours, and $\tau \ll \tau'$, then τ' has precisely the same set of colours. In particular, each \ll -related equivalence class of rank 0 contains type constructors with exactly the same colours.

We also require that type variables can only be coloured with combinations of colours for which ground types exist. In our case, ground types coloured both red and blue will exist, so type variables coloured both red and blue are permitted.

These restrictions ensures that colours are ‘orthogonal’ to subtyping. After a suitable definition of *substitution* (definition 1.5.1), we will not have to discuss colours explicitly for most sections of this work.

A type variable with no colours is referred to as an *uncoloured* type variable.

Definition 1.4.3

We define the *basic types*, or types coloured red, to be of the form :

$$\bullet \quad \alpha \quad \text{or} \quad \delta_r \, \bar{\tau} \quad \text{or} \quad \beta_q \, \bar{\tau}'$$

where α is a type variable of rank 0 which is coloured red, δ_r is a type constructor of rank $r \neq *$, β_q is a type variable of rank $q \notin \{0, *\}$, and $\bar{\tau}$, $\bar{\tau}'$ are vectors of basic types of length r, q . In other words, the basic types are exactly those types containing no type constructors or type variables of rank $*$. In Standard ML, these are referred to as ‘eq types’, and basic type variables are called ‘eq type variables’.

The *tuple types*, or types coloured blue, are the types of the form :

$$\bullet \quad \alpha \quad \text{or} \quad \text{void} \quad \text{or} \quad \tau \times \tau'$$

where α is a type variable of rank 0 which is coloured blue, τ is any type, and τ' is a tuple type. Note that by the restriction on coloured types in definition 1.4.2, we must have the type constructor \times incomparable to each other type

constructor of rank 2 under \ll , and *void* incomparable to each other primitive type under \ll . Note that some types are both basic and tuple types.

When we discuss type substitutions, we will require a coloured type variable to be instantiated to a type with the same, but possibly additional, colours. That is, basic type variables (coloured red) can only be instantiated to basic types, tuple type variables (coloured blue) can only be instantiated to tuple types, and basic tuple type variables (coloured red and blue) can only be instantiated to types which are both basic and tuple types.

The use of coloured typed variables allows us to assign types to functions which previously gave some difficulty. For example, the polymorphic equality function has the type :

- $\tau \rightarrow \tau \rightarrow \text{bool}$

for each basic type τ , and can be assigned the *type scheme* :

- $\forall \alpha. \alpha \rightarrow \alpha \rightarrow \text{bool}$

where α is a basic type variable, ie α is coloured red. This ensures that an attempt to test two functions for equality results in a type error. Also, a functional language implementation would only print the value of expressions that have basic types, since functions do not usually have a printable value.

Assuming an n -element tuple type is represented by :

- $\tau_1 \times \dots \tau_n \times \text{void}$

then we can assign to the tuple selector $\#m$, as defined in FP (Backus 1978, Dekker 1983) the type scheme :

- $\forall \beta_1 \dots \beta_m \gamma. \beta_1 \times \dots \times \beta_m \times \gamma \rightarrow \beta_m$

where β_1, \dots, β_m are not coloured, and γ is a tuple type variable, ie γ is coloured blue. The handling of tuple types is similar to the technique used by Collier (1987), and permits the selector $\#m$ to be used with any n -element tuple, provided $m \leq n$. The reader familiar with Prolog will notice the analogy with lists in that language.

Remark 1.4.4

Coloured types allow other groupings of types to which a function is applicable. For example, in our example type hierarchy, the inequality $\alpha \leq \text{int}$ specifies numeric types. We could however, add unrelated types such as *time* and *money* which would have addition defined on them. By specifying a colour (say orange) and colouring the types *nat*, *int*, *time* and *money* (and appropriate type variables) with that colour, we could make addition applicable to only those types. However, by our requirements in definition 1.4.2, this would not be possible in the presence of a coercion from *int* to *atom*. We therefore have *generic functions* in the sense of Reynolds (1980), which are functions with types parameterised on a fixed set of ‘key’ types. However, we do not permit coercions between ‘key’ types and other types. On the other hand, we extend the generic functions of Reynolds by allowing ‘key’ type constructors. For example, we can make a function applicable to both sums and products by assigning a colour (say green) to types of the form $\tau \times \tau'$ or $\tau + \tau'$ (and of course, to appropriate type variables).

We have made the restriction that a type variable can only be coloured with a combination of colours for which ground types exist. We also require that the types for each such legal combination of colours have the form :

$$\bullet \quad \alpha \text{ or } a \text{ or } b_r \overline{\tau_r}$$

where α is a type variable of rank 0 with the given combination of colours, a ranges over a possibly empty set of primitive types, and for each rank $r \neq 0$, b_r is either a specific type constructor (such as \times), or b_r ranges over all type variables and type constructors of rank r , or b_r ranges over an empty set. Furthermore, for each rank r , the $\tau_{r,i}$ range over types of a specified colour. Comparing definition 1.4.3 shows that basic types, tuple types and basic tuple types all satisfy this restriction. In fact, it is clear that the restriction holds for the individual colours, it will also hold for all combinations of colours. Making this restriction allows us to easily check that substitutions, which we

will define in the next section, are legal, and to compose substitutions. It also makes possible an extension of unification, which we will define in section 1.7.

We thus have a general technique for defining generic functions, which is orthogonal to our subtype system. Effectively, colouring is a way of

constraining the bound variables in a type scheme that complements bounded quantification. In chapter 4 we introduce a third technique for constraining type variables. Chapter 5 will touch briefly on other techniques for generic functions, and compare them to our colouring scheme.

Definition 1.4.5

Following Huet (1986), we can identify *occurrences* of type variables and type constructors within types with sequences of positive integers. We define these sequences as follows, where a and b can be type variables or type constructors:

- the empty sequence ' ' is an occurrence of a in $a \ \bar{\tau}$
- if ' $k_1 \dots k_m$ ' is an occurrence of a in τ_i , then ' $ik_1 \dots k_m$ ' is an occurrence of a in $b \ \bar{\tau}$.

We say that a and b have *corresponding occurrences* in τ and τ' respectively, if there are identical sequences of positive integers representing occurrences of a and b . For example, in the following pairs of types, α and β have corresponding occurrences :

- $\alpha \times int, \beta \times bool$
- $void \rightarrow \alpha + int, atom \rightarrow \beta + nat$
- $\alpha \rightarrow bool \times nat, \beta \rightarrow int$

Definition 1.4.6

We define *left* and *right occurrences* of type variables and type constructors recursively as follows :

- a occurs right in $a \ \bar{\tau}$
- if a occurs right (left) in some τ_i and b does not have rank *, then a occurs right (left) in $b \ \bar{\tau}$
- if a occurs right (left) in τ_2 and b has rank *, then a occurs right (left) in $b(\tau_1, \tau_2)$
- if a occurs right (left) in τ_1 and b has rank *, then a occurs left (right) in $b(\tau_1, \tau_2)$

Note that it is possible for a to occur both left and right in some type.

Proposition 1.6.5 indicates that left occurrences are exactly the 'antimonotonic positions' in a type. For example, α only occurs left, and β only occurs right,

in the following types :

- β
- $(\alpha \rightarrow \beta) \times \beta$
- $(\beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta$

1.5 Type Substitutions

Definition 1.5.1

We write $S = [t_1/\alpha_1, \dots, t_n/\alpha_n]$, or more concisely $S = [\bar{t}/\bar{\alpha}]$ for the simultaneous *substitution* of type variables α_i by t_i , where :

- if α_i has rank $r \neq 0$, then t_i is either a type variable or a type constructor of rank r .
- if α_i has rank 0, then t_i is a type with at least the same colours as α_i .

We define the application of such a substitution S to type variables, type constructors, and types by :

- $S\alpha_i = t_i$
- $S\delta = \delta$
- $S\beta = \beta$ where β is not one of the α_i
- $S(a\tau_1 \dots \tau_n) = (Sa) (S\tau_1) \dots (S\tau_n)$

The two restrictions in definition 1.5.1 ensure that for each type τ , $S\tau$ is a legal type, and that restrictions on instantiating coloured type variables discussed in section 1.4 are satisfied.

Given two substitutions R and S , we write RS for the composition of R and S , and since :

- $(RS) \tau = R(S \tau)$

we omit the parentheses.

A number of type substitutions are of special interest. The *identity substitution* ID is defined by $ID = []$, and for every type τ we have :

- $ID \tau = \tau$

A *ground substitution* is a substitution $[\bar{t}/\bar{\alpha}]$ such that the t_i do not contain type variables. An *atomic substitution* is a substitution $[\bar{t}/\bar{\alpha}]$ such that the t_i are either type constructors or type variables. An *instantiation* of type variables $\bar{\alpha}$ is a substitution $[\bar{t}/\bar{\alpha}]$ such that no α_i occurs in \bar{t} . A *renaming* of type variables $\bar{\alpha}$ is an atomic instantiation $[\bar{\beta}/\bar{\alpha}]$ such that the β_i are previously unused type variables of precisely the same rank and colouring as the corresponding α_i . Note that ground atomic substitutions, ground instantiations and ground atomic instantiations are also possible.

Definition 1.5.2

We say that a substitution S is a *minimal* substitution satisfying certain conditions, if for every substitution S' satisfying those conditions, there is a substitution R such that $S' = RS$.

Similarly, we say that a type τ is minimal satisfying certain conditions, if for every type τ' satisfying those conditions, there is a substitution R such that $\tau' = R\tau$. For example, any basic type variable is a minimal basic type.

Definition 1.5.3

We say that two types or substitutions, X and Y , are *equivalent up to renaming* if there are atomic substitutions S and R such that $X = SY$ and $Y = RX$, in which case we write $X \approx Y$.

It is clear from definition 1.5.2 that if X and Y are both minimal satisfying certain conditions, then X and Y will be equivalent up to renaming. More specifically :

Proposition 1.5.4

Let X be a minimal type or substitution satisfying certain conditions. Then Y is minimal satisfying those conditions if and only if $X \approx Y$.

Proof: For the if part, let Z satisfy the conditions and $Z = SX$. If $X \approx Y$ then $X = RY$, so $Z = SRY$, and hence Y is minimal satisfying the conditions. The only-if part is trivial, since if $X = SY$ and $Y = RX$, then S and R must be atomic. \square

Definition 1.5.5

If $S = [t_1/\alpha_1, \dots, t_n/\alpha_n]$ is any substitution, we define :

- $Dom(S) = \{ \alpha_1, \dots, \alpha_n \}$
- $New(S) = FV(t_1) \cup \dots \cup FV(t_n)$

where $FV(t_i)$ is the set of type variables occurring in t_i .

1.6 Subtype Orderings

Definition 1.6.1

Given a family of disjoint sets of type variables W_r (ie W_r is a subset of the set of type variables TV_r) and a family of partial order relations \leq_r on $\Delta_r \cup W_r$, we call the family of partially ordered sets $\Delta_r \cup W_r$, a *subtype ordering* if \leq_r preserves \ll_r in the following sense :

- for each $\delta_r, \delta'_r \in \Delta_r$, $\delta_r \leq_r \delta'_r$ if and only if $\delta_r \ll_r \delta'_r$
- δ_r and δ'_r have a lower (upper) bound under \leq_r only when they have a glb (lub) under \ll_r
- no two \ll -unrelated elements are \leq -related (where \leq -relatedness refers to the reflexive, transitive, symmetric closure of the \leq_r)
- if α is \leq -related to δ of rank 0, then α only has colours possessed by δ , ie $[\delta/\alpha]$ is legal substitution.
- if α is \leq -related to β of rank 0, then there exist type variables with the colours of both α and β (By definition 1.4.2 this means that there exist ground types with these colours).

If the \leq_r are preorders preserving \ll_r , but not partial orders, then we call the family of pre-ordered sets a *semi-subtype ordering*.

The five restrictions ensure that the subtype ordering is *satisfiable* in the sense that there is a ground substitution S such that :

- for $\alpha \in W_r$, $S\alpha \in \Delta_r$
- $a \leq_r b$ if and only if $Sa \ll_r Sb$

In Section 3.4 we show how such a substitution S can be calculated. Figure 2a shows a subtype ordering, while Figure 2b shows a partial order which contravenes the first three restrictions.

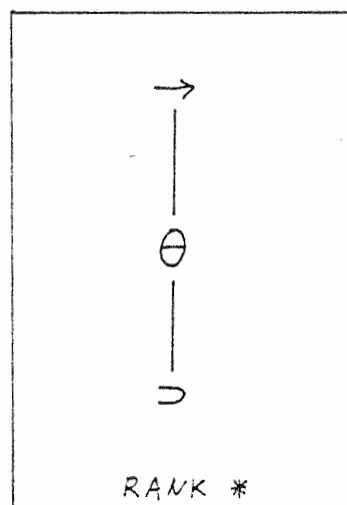
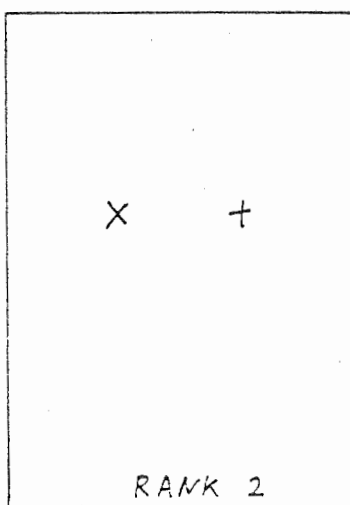
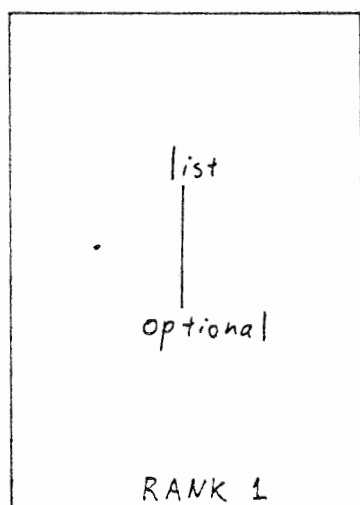
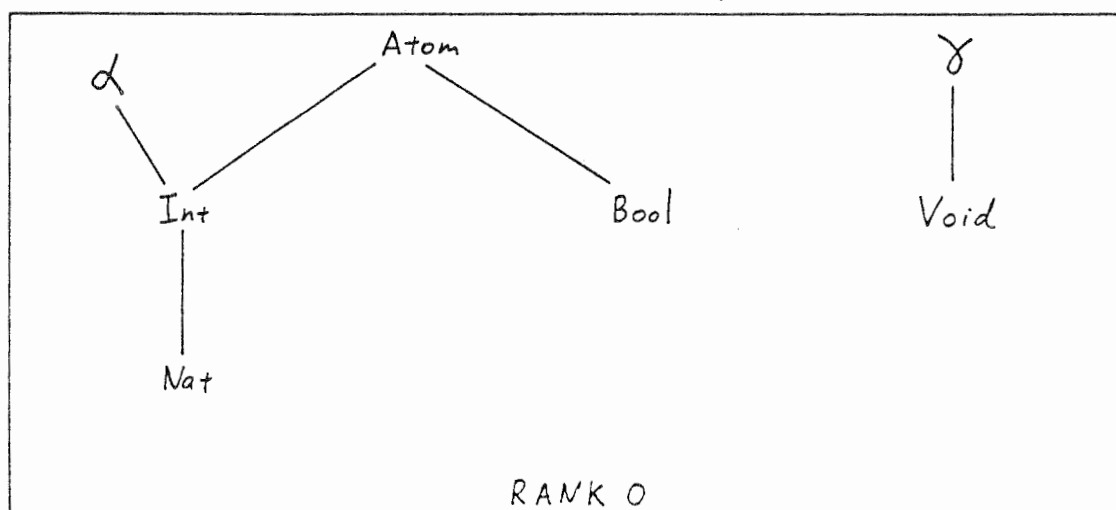


Figure 2a - An example subtype ordering

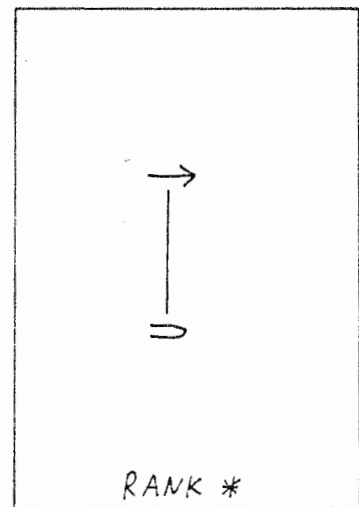
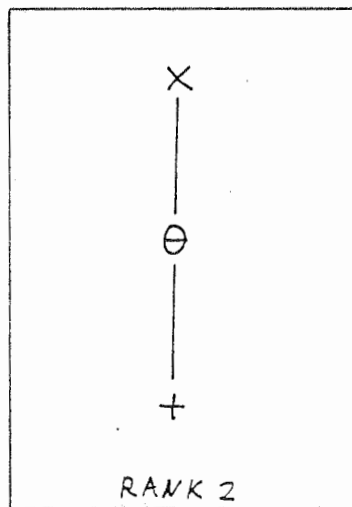
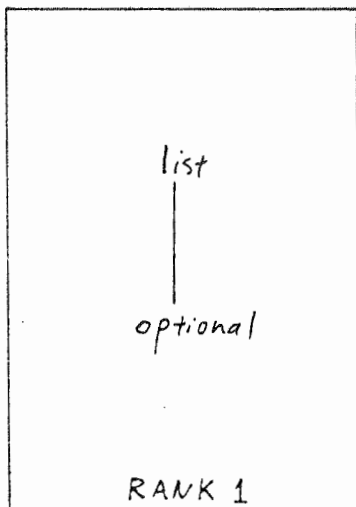
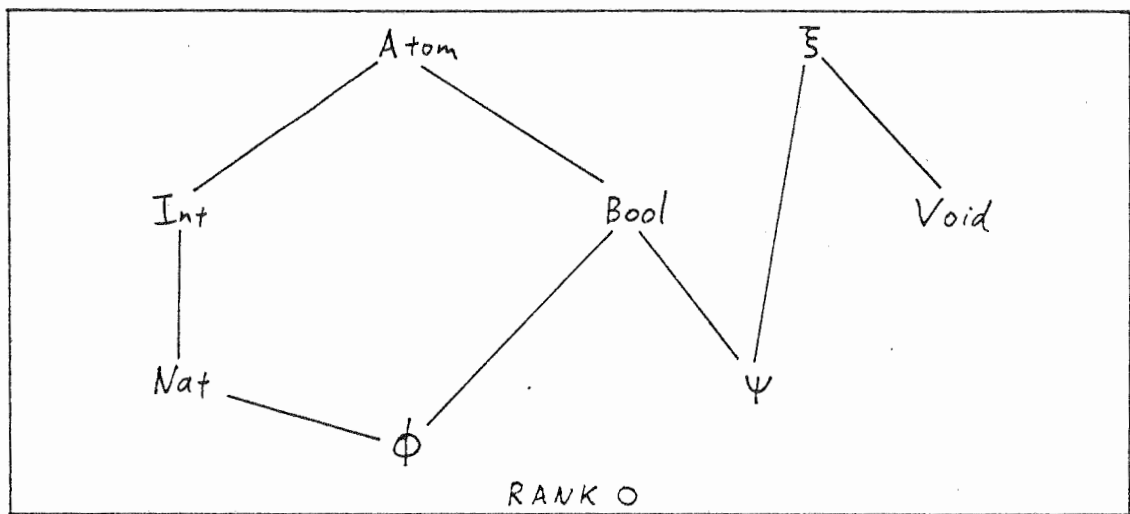


Figure 26- A partial order which is not a subtype ordering

Section 3.4 effectively shows that our five restrictions are sufficient for the substitution S to exist. Figure 2b shows that the first three conditions are necessary, in that no ground types can be assigned to ξ , θ , ψ or ϕ such that the give inequalities hold. It is easy to show that the last two restrictions are also necessary.

As before, we will omit the subscript r where the rank is obvious from context. We will also use \leq to refer to the family of partial orders \leq_r , and to the family of partially ordered sets $\Delta_r \cup W_r$ if the family W of sets of type variables is obvious from context.

We will use lower case letters a, b, \dots for elements of $\Delta_r \cup W_r$, which can be either type constructors or type variables, and write Sa, Sb, \dots for the results of applying the substitution S . By definition 1.5.1, if a is type constructor we will have $Sa = a$.

We will use the symbol Π for the primitive subtype ordering which has each $W_r = \{ \}$ and $\leq_r = \ll_r$, and we introduce the symbol \diamond for the *undefined* subtype ordering. We will use upper case letters X, Y, \dots for possibly undefined subtype orderings.

Definition 1.6.2

We define the application of an atomic substitution S to a subtype ordering as follows. Given an atomic substitution S , we define the family of sets of type variables SW by :

- $SW = \text{New}(S) \cup (W - \text{Dom}(S))$

If \leq is a subtype ordering with family W of sets of type variables, and S is an atomic substitution, then let \leq' be the smallest family of preorders such that :

- $Sa \leq' Sb$ whenever $a \leq b$

If each \leq'_r preserves \ll_r , then we define $S(\leq)$ to be the semi-subtype ordering \leq' on the family SW of sets of type variables. If some \leq'_r does not preserve \ll_r , then we define $S(\leq) = \diamond$. We also define $S(\diamond) = \diamond$. Note that if each \leq'_r is a partial order, then \leq' is a subtype ordering. In Section 1.10 we will define a more useful and general form of substitution on subtype orderings.

Proposition 1.6.3

Let R, S be atomic substitutions. Then $RS(\leq) = R(S(\leq))$ for each subtype ordering \leq .

Proof: By definition 1.6.2, \leq', \leq'', \leq''' are smallest such that $a \leq b$ implies $Sa \leq' Sb$ implies $RSa \leq'' RSb$, and $a \leq b$ implies $RSa \leq''' RSb$.

Thus $\leq'' = \leq'''$. \square

Definition 1.6.4

If \leq is a subtype ordering (semi-subtype ordering), we extend it to a partial order (preorder) on types in the same way that we extended \ll to a partial order on closed types :

- $a \bar{\tau} \leq b \bar{\tau}'$ where a and b have rank $r \neq *$, $a \leq b$, and $\tau_i \leq \tau'_i$ for $1 \leq i \leq r$, where $\bar{\tau}$ and $\bar{\tau}'$ are vectors of length r .
- $a\tau_1\tau_2 \leq b\tau'_1\tau'_2$, where a and b have rank $*$, if and only if $a \leq b$, $\tau_1 \leq \tau'_1$, and $\tau_2 \leq \tau'_2$.

For example, if $\alpha \leq \beta$ and $\text{nat} \leq \gamma$, we have :

- $\text{nat} \times \alpha \leq \text{int} \times \beta$
- $\beta \supset \text{nat} \leq \alpha \rightarrow \gamma$

Proposition 1.6.5

For any types τ and τ' , $\tau \leq \tau'$ if and only if the following two conditions hold :

- $a \leq b$ for all a and b with corresponding right occurrences in τ and τ' respectively.
- $b \leq a$ for all a and b with corresponding left occurrences in τ and τ' respectively.

Proof: By induction on the structure of τ . \square

Corollary 1.6.6

Let S be an atomic substitution. Then $S(\leq) = \leq'$ if and only if $\tau \leq \tau'$ implies $S\tau \leq' S\tau'$ for all τ, τ' .

Proof: By proposition 1.6.5 and definition 1.6.2. \square

Definition 1.6.7

We say that a subtype ordering \leq is *least* satisfying certain conditions if for every subtype ordering \leq' satisfying those conditions, $a \leq b$ implies $a \leq' b$. If no subtype ordering satisfies the conditions, then we will say that \diamond is least satisfying the conditions.

Clearly the least subtype ordering satisfying given conditions is unique. For example, Π is the least subtype ordering such that $nat \leq int$, while \diamond is the least subtype ordering such that $bool \leq int$.

Definition 1.6.8

As in definition 1.5.3, two subtype orderings \leq and \leq' are *equivalent up to renaming*, written $\leq \approx \leq'$, if there are atomic substitutions S and R such that $S(\leq')$ and $\leq = R(\leq)$.

Definition 1.6.9

We define the *simple* subtype orderings as follows :

- The primitive subtype ordering Π is a simple subtype ordering.
- Let \leq be a subtype ordering which contains exactly one type variable α of rank r , ie $W_r = \{\alpha\}$ for exactly one rank r , such that $\alpha \leq \delta_r$ for some type constructor δ_r , and whenever $\alpha \leq \delta'_r$ then $\delta_r \leq \delta'_r$, and there is no δ''_r such that $\delta''_r \leq \alpha$. Then \leq is a simple subtype ordering, written $[\alpha \leq \delta_r]$.
- The subtype ordering defined as above but with the sign \leq reversed is a simple subtype ordering, written $[\delta_r \leq \alpha]$.
- Let \leq be a subtype ordering which contains exactly two type variables α and β of rank r , ie $W_r = \{\alpha, \beta\}$ for exactly one rank r , such that $\alpha \leq \beta$, and α, β are incomparable to all type constructors of rank r . Then \leq is a simple subtype ordering, written $[\alpha \leq \beta]$.
- The subtype ordering where each type variable is incomparable to all other type variables and type constructors is a simple subtype ordering, written $[W]$, where W is the associated family of sets of type variables. We will refer to such type variables as *isolated* type variables.

We will use upper case letter B, C, \dots for simple subtype orderings. We note that, like all subtype orderings, the simple subtype orderings all contain Π .

Proposition 1.6.10

The subtype orderings $[\alpha \leq \delta]$, $[\delta \leq \alpha]$, $[\alpha \leq \beta]$ are the least subtype orderings such that the indicated inequality holds.

Proof: By definitions 1.6.7 and 1.6.9. \square

Proposition 1.6.10 indicates that the three cases of simple subtype orderings involving inequalities on type variables can be viewed as ‘atomic’ objects. In section 1.9 we will show that they correspond to *edges* in a directed graph representing a subtype ordering. We will also show how a subtype ordering \leq can be *split* into the constituent simple subtype orderings which form the corresponding graph. This graph has the property that there is a path from a to b if and only if $a \leq b$, and it is the smallest such graph.

1.7 Matching Types

Definition 1.7.1

We define an equivalence relation of *matching* on types, extending that of Mitchell (1984). Two types match if they have the same ‘shape’.

More formally :

- $a \ \overline{\tau}$ matches $b \ \overline{\tau'}$ if a and b are type variables or type constructors of rank r , and $\overline{\tau}, \overline{\tau'}$ are vectors of types of corresponding length n , with τ_i matching τ'_i for $1 \leq i \leq n$.

For example $list(\alpha \rightarrow int)$ matches $\beta(bool \rightarrow nat)$ where α has rank 0, and β has rank 1. Some properties of matching are given by the following proposition :

Proposition 1.7.2

- i If $\tau \leq \tau'$ for some subtype ordering \leq , then τ matches τ' .
- ii If τ matches τ' , then $|\tau| = |\tau'|$

Proof: By definitions 1.4.1, 1.6.4 and 1.7.1. \square

Definition 1.7.3

Let \leq be a subtype ordering. Then we say that a substitution S is *shape-consistent with respect to \leq* if Sa matches Sb whenever $a \leq b$. Mitchell (1984) uses the terminology ' S respects \leq ' for the same condition.

Proposition 1.7.4

S is shape-consistent with respect to \leq if and only if $S\tau$ matches $S\tau'$ whenever $\tau \leq \tau'$.

Proof: By definition 1.7.1 and proposition 1.6.5. \square

Definition 1.7.5

We call a substitution S an *expansion* if S has the form :

$$\bullet \quad [a_1 \ \bar{\beta}_1/\gamma_1] \dots [a_n \ \bar{\beta}_n/\gamma_n]$$

where $n \geq 0$, the γ_i are type variables of rank 0, the a_i are type constructors or previously unused type variables of rank $r \neq 0$, and the $\bar{\beta}_i$ are vectors of distinct previously unused type variables of rank 0, such that β_{ij} do not have more colours than required for the substitution to be legal, and the a_i are only type constructors if the colouring of γ_i demands this.

For example, if α, β and γ are previously unused uncoloured type variables of rank 0, θ is a previously unused type variable of rank 1, ϕ is a previously unused type variable of rank 2, and ψ, ξ have rank 0 and are uncoloured, then :

$$\bullet \quad [\theta(\alpha) / \psi, \phi(\beta, \gamma) / \xi]$$

is an expansion. We will sometimes say that this substitution *expands* ψ and ξ .

Note that if ξ had been coloured red, then by definition 1.4.3, β and γ would also have to be coloured red. If ξ had been coloured blue, ϕ would have to be replaced by \times , and γ would have to be coloured blue. Note also that $\theta(\alpha)$ and $\phi(\beta, \gamma)$ are minimal types matching, say, *list int* and *bool \times nat* respectively. More generally :

Proposition 1.7.6

Let $S = [\overline{\tau} / \overline{\alpha}]$ be an expansion such that each $S\alpha_i$ matches τ'_i . Then S is a minimal substitution satisfying that condition, and the τ_i are minimal types matching τ'_i .

Proof: Let τ'_i match τ'_i . Then by induction on the structure of τ'_i , there exists an R such that $R\tau_i = \tau'_i$. Also if S' satisfies the required condition then $\text{Dom}(S) \subseteq \text{Dom}(S')$, and using the minimality of τ_i , S is minimal. \square

Corollary 1.7.7

Every type τ can be expressed in the form $RS\alpha$, where R is atomic and S is an expansion.

Proof: Construct S such that $S\alpha$ is minimal matching τ . Then R exists by minimality, and must be atomic. \square

Corollary 1.7.8

Let S be any substitution. Then there is an atomic substitution S' and an expansion S'' such that $S = S'S''$. We refer to $S'S''$ as a *factorisation* of S .

Proposition 1.7.9

If S is an expansion, then $S\tau = S\tau'$ if and only if $\tau = \tau'$.

Proof: By definition 1.7.5, since S introduces distinct new type variables. \square

By proposition 1.7.9, expansions are in some sense *injective*, and the factorisation result in corollary 1.7.8 recalls epi-mono factorisations in category theory (Herrlich and Strecker 1979), although a formal categorical treatment of types and substitutions is beyond the scope of this work. We do, however, use the injectivity of expansions to make the following definition :

Definition 1.7.10

We define the application of an expansion S to a subtype ordering similarly to definition 1.6.2. We define :

- $SW = \text{New}(S) \cup (W - \text{Dom}(s))$

If \leq is a subtype ordering with family W of sets of type variables, and the expansion S is shape-consistent with respect to \leq , then let \leq' be the smallest family of partial orders such that :

- $Sa \leq' Sb$ whenever $a \leq b$

Since S only introduces distinct new type variables, \leq' is uniquely defined, and preserves \ll . Hence we define $S(\leq)$ to be the subtype ordering \leq' on the family SW of sets of type variables.

Theorem 1.7.11 - MATCH algorithm

Let τ and τ' be arbitrary types. Then there is an algorithm $MATCH(\tau, \tau')$ which computes a minimal expansion S such that $S\tau$ matches $S\tau'$, if such an expansion exists, and fails otherwise.

Proof : Based on the unification algorithm (Robinson 1965, Martelli and Montanari 1982). The required algorithm does unification with matching as the 'equality' relation. The restrictions we have placed on coloured types in Remark 1.4.4 and on expansions in Definition 1.7.5 ensure that the required expansions can be constructed. \square

Note that there are many minimal expansions S such that $S\tau$ matches $S\tau'$, but by proposition 1.5.4, they are all equivalent up to renaming. For example, if $\alpha, \beta, \gamma, \xi$ are new uncoloured type variables of rank 0, and θ, ϕ are new type variables of rank 2, then two minimal expansions such that ψ matches $int \times bool$ are :

- $[\theta(\alpha, \beta) / \psi]$
- $[\phi(\gamma, \xi) / \psi]$

Definition 1.7.12 - MATCH2 algorithm

Let \leq be a subtype ordering. For arbitrary types τ and τ' we then define :

- $MATCH2(\leq, \tau, \tau') = MATCH(a_1 \times \dots \times a_n \times \tau, b_1 \times \dots \times b_n \times \tau')$
- $MATCH2(\diamond, \tau, \tau')$ fails

where a_1, \dots, a_n and b_1, \dots, b_n are suitably chosen type variables or type constructors of rank 0, such that $a_i \leq b_i$ for each i , $1 \leq i \leq n$, and the smallest partial order generated by the $a_i \leq b_i$ together with the rank 0 portion of Π , is the rank 0 portion of \leq .

At the end of section 1.6 we indicated that a subtype ordering could be viewed as a directed graph. The condition we require in the above definition is that the (a_i, b_i) are precisely the rank 0 edges in the directed graph corresponding to \leq that are additional to those in the directed graph corresponding to Π . This condition ensures that whenever a type variable is expanded, all \leq -related type variables are also expanded. For example, let $\leq = [\alpha \leq \beta]$, and consider :

$$\text{MATCH2}(\leq, \alpha \rightarrow \xi, \gamma \times \eta \rightarrow \beta)$$

Clearly α must be expanded to match $\gamma \times \eta$, but since β is \leq -related to α , we must also expand β , and hence ξ , so that one possible result is :

$$[\alpha_1 \times \alpha_2 / \alpha, \beta_1 \times \beta_2 / \beta, \xi_1 \times \xi_2 / \xi]$$

Section 1.9 discusses the directed graph corresponding to a subtype ordering in more detail.

Theorem 1.7.13

$\text{MATCH2}(\leq, \tau, \tau')$ computes a minimal expansion S , shape-consistent with respect to \leq , such that $S\tau$ matches $S\tau'$, if such an expansion exists.

Furthermore the expansion is independent of the particular choice of a_i and b_i .

Proof

- Sa_i matches Sb_i by theorem 1.7.11. Since if $a \leq b$ this is implied by the $a_i \leq b_i$, shape-consistency follows.
- $S\tau$ matches $S\tau'$ by theorem 1.7.11.
- Minimality of S follows from theorem 1.7.11., and the fact that for shape-consistency it is necessary that Sa_i match Sb_i .
- Independence of the choice of a_i and b_i follows from the fact that any choice of a_i and b_i satisfying the required conditions will provide a minimal expansion S such that S is shape-consistent with respect to \leq , and $S\tau$ matches $S\tau'$.
- If no expansion S satisfying the conditions exists, then the call to MATCH2 will fail. \square

Note that, like MATCH , MATCH2 is a non-deterministic algorithm, which produces one of many minimal expansions which are all equivalent up to renaming.

1.8 Union of Subtype Orderings

Definition 1.8.1

Let \leq_1, \dots, \leq_n be arbitrary subtype orderings or semi-subtype orderings, with families of sets of type variables W_1, \dots, W_n . Let $W = W_1 \cup \dots \cup W_n$, and let \leq be the smallest family of preorders on $\Delta \cup W$ generated by reflexivity, transitivity, and the rule :

- $a \leq b$ if $a \leq_i b$ for some i , $1 \leq i \leq n$.

Let \sim be the family of equivalence relations on $\Delta \cup W$ generated by symmetry on the family of preorders \leq , ie

- $a \sim b$ if and only if $a \leq b$ and $b \leq a$.

Let E_1, \dots, E_m be the equivalence classes of \sim . Then we define atomic substitutions S_1, \dots, S_m , which map these equivalence classes to suitably chosen representatives as follows :

- If E_i contains two or more type constructors then S_i is undefined.
- If E_i contains one type constructor and $k \geq 0$ type variables, ie $E_i = \{\delta, \alpha_1, \dots, \alpha_k\}$, then we define $S_i = [\delta/\alpha_1, \dots, \delta/\alpha_k]$, provided this is a legal substitution (ie if the elements of E_i have rank 0 then no α_j has more colours than δ , viewed as a type).
- If E_i contains only type variables of rank other than 0, ie $E_i = \{\alpha_1, \dots, \alpha_k\}$, then we define $S_i = [\alpha_j/\alpha_1, \dots, \alpha_j/\alpha_k]$ for some arbitrary $\alpha_j \in E_i$.
- If E_i contains only type variables of rank 0, ie $E_i = \{\alpha_1, \dots, \alpha_k\}$ and some α_j has all colours which occur in any of $\alpha_1, \dots, \alpha_k$, then we define $S_i = [\alpha_j/\alpha_1, \dots, \alpha_j/\alpha_k]$.
- If E_i contains only type variables of rank 0, ie $E_i = \{\alpha_1, \dots, \alpha_k\}$, but the above condition is not satisfied, then let β be a previously unused type variable with precisely all the colours which occur in E_i , and we define $S_i = [\beta/\alpha_1, \dots, \beta/\alpha_k]$.

If some S_i is undefined, then we say that \leq_1, \dots, \leq_n are *incompatible*, and define :

- $\text{UNION}(\leq_1, \dots, \leq_n) = \diamond, ID$

Otherwise we let $S = S_1 \dots S_m$, and note that $Sa \leq Sb \leq Sa$ implies $Sa = Sb$.

We can therefore define the partial order \leq' on $\Delta \cup SW$ by :

- $Sa \leq' Sb$ if and only if $a \leq b$.

If \leq' preserves \ll as in definition 1.6.1, then \leq' is a subtype ordering, and we define :

- $\text{UNION}(\leq_1, \dots, \leq_n) = \leq', S$

Otherwise we again say that \leq_1, \dots, \leq_n are incompatible, and define $\text{UNION}(\leq_1, \dots, \leq_n) = \diamond, ID$.

We also extend UNION to possibly undefined subtype orderings by defining :

- $\text{UNION}(X_1, \dots, X_n) = \diamond, ID$ if some $X_i = \diamond$

We note that the equivalence classes E_i in this definition are precisely the strongly connected components of the directed graph corresponding to the preorder \leq , as will be discussed further in section 1.9. In order to prove the required properties of the UNION operation, we require the following definitions of minimality and equivalence up to renaming of subtype ordering/substitution pairs.

Definition 1.8.2

We say that the pair \leq, S is *minimal* satisfying certain conditions if for every \leq', R satisfying those conditions, there is a substitution S' such that :

- $R = S'S$
- $\tau \leq \tau'$ implies $S'\tau \leq' S'\tau'$ for all τ, τ'

By proposition 1.6.5, to verify the second requirement it is sufficient to show that $a \leq b$ implies $S'a \leq' S'b$ for all a, b .

Definition 1.8.3

We say that the pairs \leq, S and \leq', R are *equivalent up to renaming*, and write $\leq, S \approx \leq', R$ if there are atomic substitutions S' and R' such that :

- $R = R'S, \leq' = R'(\leq)$
- $S = S'R, \leq = S'(\leq')$

and such that whenever $S = R = ID$, then $\leq = \leq'$.

It is clear from definitions 1.5.3 and 1.6.8 that if $\leq, S \approx \leq', R$ then $S \approx R$ and $\leq \approx \leq'$. The following propositions extends the result of proposition 1.5.4.

Proposition 1.8.4

Let \leq, S be minimal satisfying certain conditions. Then \leq', R is minimal satisfying those conditions if and only if $\leq, S \approx \leq', R$.

Proof: If \leq, S and \leq', R are both minimal then $R = S'S$ and $S = R'R$, and S', R' must be atomic. Equivalence up to renaming follows from definition 1.8.2 and corollary 1.6.6. Conversely, if \leq, S is minimal, $\leq, S \approx \leq', R$, and \leq'', R'' satisfies the conditions, then there exists S' such that $R'' = S'S$ and $a \leq b$ implies $S'a \leq'' S'b$. Also there exists S'' such that $S = S''R$ and $\leq = S''(\leq')$. Hence $S'S''$ is the required substitution, and \leq', R is minimal, by definition 1.8.2 and corollary 1.6.6. \square

We can now prove the required properties of UNION by the following theorem and corollaries :

Theorem 1.8.5

Let $\text{UNION}(\leq_1, \dots, \leq_n) = \leq', S$. Then \leq', S is minimal such that $\tau \leq_i \tau'$ implies $S\tau \leq' S\tau'$ for all τ, τ' .

Proof: Let \leq'', R be such that $a \leq_i b$ implies $Ra \leq'' Rb$. Let \leq be the smallest family of preorders generated by the \leq_i . Then $a \leq b$ implies $Ra \leq'' Rb$, and hence $a \leq b \leq a$ implies $Ra = Rb$. Since in definition 1.8.1, S is constructed to be minimal, we must have $R = S'S$ for some S' . Furthermore, $Sa \leq' Sb$ implies $a \leq b$ which implies $Ra = S'Sa \leq'' S'Sb = Rb$. The theorem follows by corollary 1.6.5. \square

Corollary 1.8.6

Let Y_1, \dots, Y_n be a permutation of X_1, \dots, X_n . Then $\text{UNION}(X_1, \dots, X_n) \approx \text{UNION}(Y_1, \dots, Y_n)$.

Corollary 1.8.7

Let $\text{UNION}(X_1, \dots, X_n) = X, S$ and $\text{UNION}(X, SY_1, \dots, SY_m) = Y, R$.

The $\text{UNION}(X_1, \dots, X_n, Y_1, \dots, Y_m) \approx Y, RS$.

Corollary 1.8.8

Let $\text{UNION}(\leq_1, \dots, \leq_n) = \leq, S$ and

$\text{UNION}(\leq_1, \dots, \leq_n, \leq'_1, \dots, \leq'_m) = \leq', R$. Then $R = S'S$ for some atomic S' and $S\tau \leq S\tau'$ implies $R\tau \leq' R\tau'$ for all τ, τ' .

Proof: By theorem 1.8.5, since \leq', R satisfies the required conditions for the first UNION operation. Since R is atomic, $R = S'S$ for some atomic S' , and $Sa \leq Sb$ implies $S'Sa = Ra \leq' Rb = S'Sb$. \square

1.9 Splittings and Graphs

Definition 1.9.1

Recalling definition 1.6.9, we define a *splitting* of a subtype ordering \leq as a set of simple orderings $\{\leq_1, \dots, \leq_n\}$ such that :

- For some i , $1 \leq i \leq n$, $\leq_i = \Pi$ or $\leq_i = [W]$
- $\text{UNION}(\leq_1, \dots, \leq_n) = \leq, ID$

We refer to the smallest such set as the *least* splitting of \leq .

It is clear that if \leq has isolated type variables (ie type variables which are not \leq -related to any other type variables and type constructors), then the least splitting of \leq has the form :

- $\{[W'], [a_1 \leq b_1], \dots, [a_n \leq b_n]\}$

where W' is the set of isolated type variables. If W' is empty, then $[W'] = \Pi$, and the least splitting of \leq has the form :

- $\{\Pi, [a_1 \leq b_1], \dots, [a_n \leq b_n]\}$

At the end of section 1.6 we indicated that a subtype ordering could be viewed as a directed graph, with simple subtype orderings as edges. Clearly a splitting of a subtype ordering can be viewed as a directed graph in this sense, with Π or $[W]$ representing the edges of the form $\delta \ll \delta'$. The least splitting then represents the smallest such graph, as shown by the following theorem.

Theorem 1.9.2

Let \leq be a subtype ordering on $\Delta \cup W$. Let G be the set of directed acyclic graphs with vertices $\Delta \cup W$ such that there is a path from a to b if and only if $a \leq b$. Then the graphs G form a lattice under containment of sets of edges, with greatest element the transitive closure of the graphs G , and least element the transitive reduct of the graphs G .

Let G_{red} be the transitive reduct of the graphs G , and let the edges of G_{red} not of the form (δ, δ') be $(a_1, b_1), \dots, (a_n, b_n)$, and let W' be the (possibly empty) family of isolated type variables in \leq . The least splitting of \leq is then the set L defined by :

- $\{[W'], [a_1 \leq b_1], \dots, [a_n \leq b_n]\}$

Proof

- For the first part of the theorem we refer the reader to Aho et al (1972) and Mehlhorn (1984).
- The set L clearly contains all the type variables and type constructors $\Delta \cup W$. Furthermore if (c, d) is an edge in G_{red} , then $c \leq d$ in some element of the set.
- The family of preorders generated by reflexivity, transitivity, and the rule $a_i \leq b_i$ is the original subtype ordering \leq , since $a \leq b$ if and only if there is a path from a to b , and hence $\text{UNION}(L) = \leq, ID$, ie the set L is a splitting.
- The splitting is least by virtue of the leastness of the transitive reduct. \square

The theorem indicates that we can represent subtype orderings uniquely by their least splittings, or more concretely, by the corresponding transitively

reduced directed acyclic graph. We will exploit this in future by writing \leq both for a subtype ordering, and for the corresponding graph. We will also use the notation for splittings in definition 1.9.1 to denote the corresponding subtype ordering or graph. The following corollary indicates that from the graph (or least splitting) we can obtain the best choice of a_i and b_i required to define the MATCH2 operation in definition 1.7.12.

Corollary 1.9.3

Let \leq be a subtype ordering with least splitting

$$\{[W'], [a_1 \leq b_1], \dots, [a_n \leq b_n], [a'_1 \leq b'_1], \dots, [a'_m \leq b'_m]\}$$

where the a_i, b_i have rank 0, and the a'_j, b'_j have rank $\neq 0$. Then the smallest partial order generated by the $a_i \leq b_i$ together with the rank 0 portion of Π , is the rank 0 portion of \leq .

Remark 1.9.4

We note that semi-subtype orderings can also be represented by directed, possibly cyclic, graphs. The UNION operation (definition 1.8.1) can then be viewed as an operation on graphs. A graph representing a family of preorders can then be formed by set union of the graphs representing the arguments. A set of equivalence classes can then be formed by a *strongly-connected components* algorithm (Aho et al 1974), and representatives of each equivalence class are chosen as in definition 1.8.1. We then perform a *transitive reduction* (Aho et al 1972), and check that the graph is indeed a subtype ordering (definition 1.6.1).

To check that a graph satisfies definition 1.6.1, we require the *transitive closure*, which can be performed together with the transitive reduct. The results in Aho et al (1972) show that transitive closure and transitive reduct are computationally equivalent operations, so that this does not add significant extra cost. We also require the *connected components*, viewing the edges in the graph as bidirectional. These correspond to \leq -relatedness equivalence classes. We must check that :

- the transitive closure of the graph, restricted to type constructors only, is identical to the transitive closure of the graph for Π
- for each δ, δ' with no glb (lub) in Π , there is no vertex with paths to (from) both δ, δ'

- the connected components of the graph, restricted to type constructors only, are identical to the connected components of the graph for Π .
- no type variable in a connected component has colours not possessed by all the type constructors in the component. (Note that by definition 1.4.2, all type constructors in the component will have the same colours).
- if all colours in a connected component are combined, a valid combination of colours results, ie ground types exist with that combination of colours.

Figure 3 shows an example of UNION of graphs. The further operations we will define in section 1.10 can also be regarded as operations on graphs in a similar way. They will all contain a UNION operation which will perform the transitive reduction and validity check.

1.10 Operations on Subtype Orderings

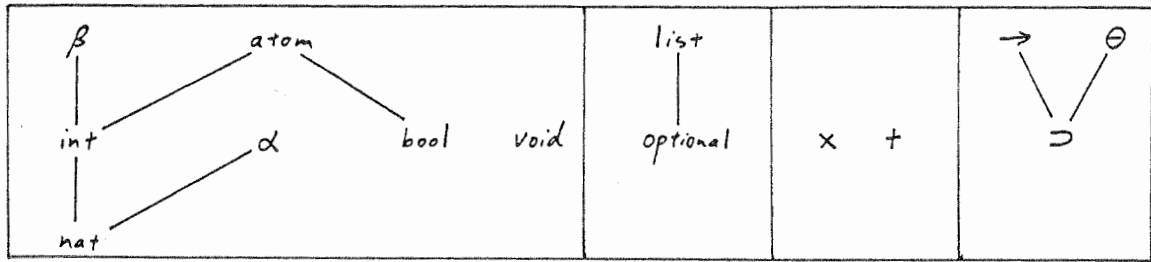
Using the UNION operation, we can now define five operations which we will use in the type inference algorithm, namely ORDERSET, ORDER, APPLYSUBST, ENRICH and SUBEN. We use these operations to create subtype orderings, apply substitutions to them, and enrich them with new inequalities.

Definition 1.10.1

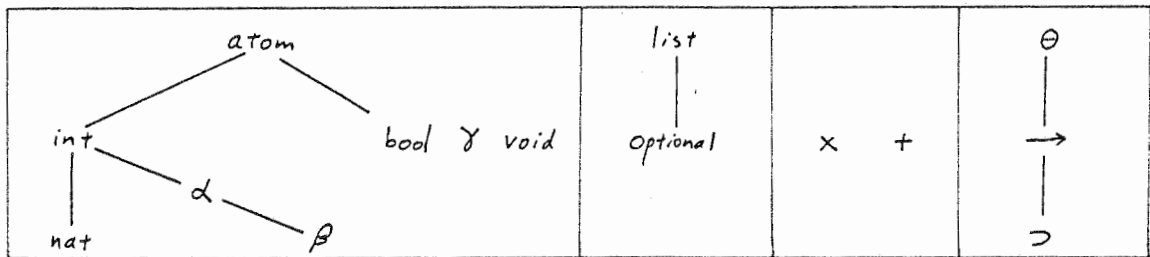
Let τ and τ' be matching types. Then we define the set of simple subtype orderings corresponding to the assertion $\tau \leq \tau'$ as follows :

- $\text{ORDERSET}(\alpha \leq \alpha) = \{[\alpha]\}$
- $\text{ORDERSET}(\delta \leq \delta') = \{\Pi\}$ if $\delta \ll \delta'$
- $\text{ORDERSET}(a \leq b) = \{[a \leq b]\}$ if one or both of a, b are type variables, and if only one is a type variable, all of its colours are possessed by the other
- $\text{ORDERSET}(a \leq b) = \{\diamond\}$ otherwise

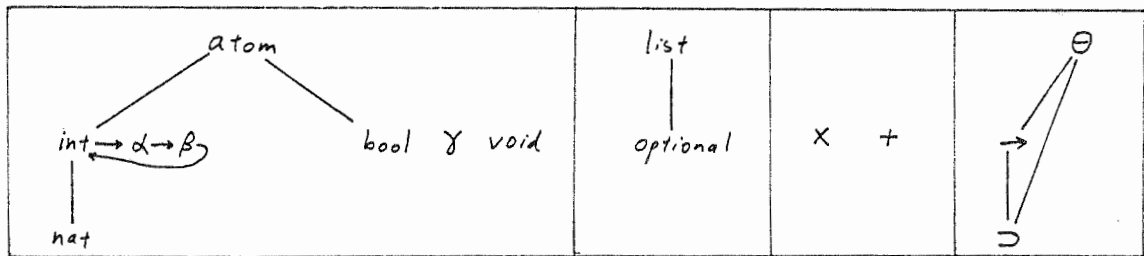
$X = \{\Pi, [\text{nat} \leq \alpha], [\text{int} \leq \beta], [\supseteq \leq \theta]\}$ has graph:



$Y = \{[\gamma], [\beta \leq \alpha], [\alpha \leq \text{int}], [\rightarrow \leq \theta]\}$ has graph:



To form $\text{UNION}(X, Y)$ construct set union of graphs :



Choose int as representative for cycle.

$\text{UNION}(X, Y) = \{[\gamma], [\rightarrow \leq \theta]\}, [\text{int}/\alpha, \text{int}/\beta]$ with graph:

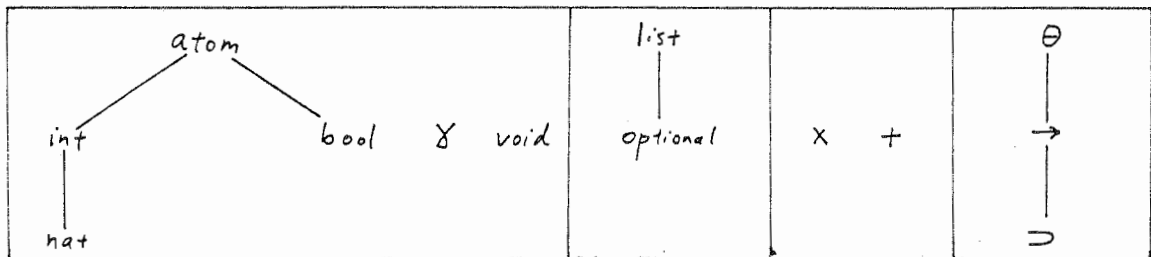


Figure 3 - Calculation of UNION by graph operations

- $\text{ORDERSET}(a \bar{\tau} \leq b \bar{\tau}) =$
 $\text{ORDERSET}(a \leq b) \cup \text{ORDERSET}(\tau_1 \leq \tau_1)$
 $\dots \cup \text{ORDERSET}(\tau_r \leq \tau_r)$
 if a, b have rank $r \notin \{0, *\}$
- $\text{ORDERSET}(a\tau_1\tau_2 \leq b\tau'_1\tau'_2) = \text{ORDERSET}(a \leq b)$
 $\cup \text{ORDERSET}(\tau'_1 \leq \tau_1)$
 $\cup \text{ORDERSET}(\tau_2 \leq \tau'_2)$
 if a, b have rank $*$

We then define the operation ORDER which performs the UNION of this set to obtain a subtype ordering and an atomic substitution :

- $\text{ORDER}(\tau \leq \tau') = \text{UNION}(\text{ORDERSET}(\tau \leq \tau'))$

The ORDERSET operation can be viewed as generating a directed graph, not necessarily acyclic. Applying a UNION operation then involves finding strongly connected components, choosing representatives for these, performing a transitive reduction, and carrying out the check described in remark 1.9.4. The major property of the ORDER operation is given by the following proposition :

Proposition 1.10.2

If $\text{ORDER}(\tau \leq \tau') = \leq', S$ then $S\tau \leq' S\tau'$ and \leq', S is minimal such that this condition holds.

Proof : The first part follows from definition 1.6.4 and theorem 1.8..5

Minimality follows from theorem 1.8.5. \square

Corollary 1.10.3

If $\text{ORDERSET}(\tau \leq \tau') = \{B_1, \dots, B_n\}$ and

$\text{UNION}(B_1, \dots, B_n, \leq_1, \dots, \leq_m) = \leq', S$ then $S\tau \leq' S\tau'$.

Proof : Using corollary 1.8.8. \square

Definition 1.10.4

We now define an operation APPLYSUBST which applies an arbitrary substitution S to a subtype ordering, returning an extended substitution RS and

a modified subtype ordering. The extended substitution may be necessary to ensure shape-consistency. We first define :

- $\text{APPLYSUBST}(S, \diamond) = \diamond, S$
- $\text{APPLYSUBST}(ID, \leq) = \leq, ID$

For the case $\text{APPLYSUBST}(S, \leq)$ with $S \neq ID$, we let $S'S''$ be a factorisation of S , ie S' is atomic and S'' is an expansion. We let $\{B_1, \dots, B_n, B_{n+1}, \dots, B_m\}$ be the least splitting of \leq , such that the B_i for $1 \leq i \leq n$ are exactly the $\llbracket a_i \leq b_i \rrbracket$ involving type variables of rank 0. If $n > 0$, we define :

- $\tau = Sa_1 \times \dots \times Sa_n$
- $\tau' = Sb_1 \times \dots \times Sb_n$
- $R' = \text{MATCH}(\tau, \tau')$

If $n = 0$, we define $\tau = \tau' = \text{void}$ and $R' = ID$.

We then let $\text{ORDERSET}(R'\tau \leq R'\tau') = \{C_1, \dots, C_k\}$ and $\text{UNION}(C_1, \dots, C_k, S'B_{n+1}, \dots, S'B_m) = X, R''$. The final case for APPLYSUBST is then defined by :

- $\text{APPLYSUBST}(S, \leq) = X, R''R'S$

The major properties of APPLYSUBST are given by the following theorem and corollaries :

Theorem 1.10.5

If $\text{APPLYSUBST}(S, \leq) = \leq', RS$ then $\tau_1 \leq \tau_2$ implies $RS\tau_1 \leq' RS\tau_2$ for all τ_1, τ_2 , and \leq', R is minimal such that this condition holds.

Proof : We need only show $a \leq b$ implies $RSa \leq' Rsb$, and minimality.

Let $S = S'S''$ and $R = R''R'$ as in definition 1.10.4.

- If $a \leq b$ of rank $\neq 0$ and $\text{UNION}(S'B_{n+1}, \dots, S'B_m) = \leq_1, R_1$, then by theorem 1.8.5 and corollary 1.8.8, $R'' = R_2R_1$ for some atomic R_2 , and $a \leq b$ implies $R_1S'a \leq_1 R_1S'b$ implies $R''S'a \leq' R''S'b$. Hence $RSa \leq' Rsb$ since expansions have no effect on type variables of rank $\neq 0$.

- If $a \leq b$ of rank 0, then ORDERSET ($R'Sa \leq R'Sb$) is a subset of $\{C_1, \dots, C_k\}$ and hence by corollary 1.10.3, $RSa \leq' RSb$.
- For minimality, let \leq_2, R_2 be such that $\tau_1 \leq \tau_2$ implies $R_1 S \tau_1 \leq_2 R_2 S \tau_2$. Then $R_2 Sa$ matches $R_2 Sb$ whenever $a \leq b$ of rank 0, so $R_2 \tau$ matches $R_2 \tau'$ where τ, τ' are as in definition 1.10.4. Thus by theorem 1.7.11, $R_2 = S_1 R'$ for some S_1 . Also $S_1 R' \tau \leq_2 S_1 R' \tau'$, and $a \leq b$ of rank $\neq 0$ implies $S_1 R' Sa \leq_2 S_1 R' Sb$, so by theorem 1.8.5, $S_1 = S_2 R''$ for some S_2 and hence $R_2 = S_2 R'' R' = S_2 R$, and $RS \tau_1 \leq' RS \tau_2$ implies $S_2 RS \tau_1 \leq_2 S_2 RS \tau_2$. \square

Corollary 1.10.6

If S is shape-consistent with respect to \leq , then APPLYSUBST (S, \leq) = X, RS where R is atomic.

Proof: By minimality, since no expansion is needed. \square

Corollary 1.10.7

If S is an atomic substitution, then APPLYSUBST (S, \leq) = X, RS where R is atomic and $RS (\leq) = X$.

Proof: The first part follows from corollary 1.10.6 and the fact that any atomic substitution is shape-consistent with respect to \leq . The second part follows from definition 1.6.2, since if $RS (\leq) = \leq'$, \leq' is least such that $a \leq b$ implies $RSa \leq' RSb$. \square

Corollary 1.10.8

If S is an expansion, then APPLYSUBST (S, \leq) = X, RS where R is an expansion and $RS (\leq) = X$.

Proof: Since S is an expansion, by proposition 1.7.9, all equivalence classes formed during the UNION operation in definition 1.10.4 are singletons, and hence :

- $\text{UNION } (C_1, \dots, C_k, S'B_{n+1}, \dots, S'B_m) = X, ID$

The second part follows from definition 1.10.7. \square

Corollary 1.10.9

If $S = \text{MATCH2 } (\leq, \tau, \tau')$, then APPLYSUBST (S, \leq) = X, S and $S (\leq) = X$.

Proof: By corollaries 1.10.6 and 1.10.8, since ID is the only atomic expansion. \square

Corollary 1.10.10

If S is a renaming, then $\text{APPLYSUBST}(S, \leq) = X, S$ and $S(\leq) = X$.

Proof: By corollaries 1.10.6 and 1.10.7, and since equivalence classes formed during the UNION operation in definition 1.10.4 are singletons. \square

Corollary 1.10.11

If $\text{APPLYSUBST}(S, \leq) = X, RS$ then $\text{APPLYSUBST}(RS, \leq) = X, RS$.

Corollary 1.10.12

Let $\text{APPLYSUBST}(S_1, \leq) = \leq_1, R_1S_1$ and $\text{APPLYSUBST}(S_2, \leq_1) = \leq_2, R_2S_2$. Then $\text{APPLYSUBST}(S_2R_1S_1, \leq) \approx \leq_2, R_2S_2R_1S_1$.

As the above theorem and corollaries indicate, APPLYSUBST generalises the substitution operations given in definitions 1.6.2 and 1.7.10, and allows us to apply any substitution to a subtype ordering. The minimality condition indicates that the result is, in some sense, the ‘natural’ result.

Our next operation extends a subtype ordering by additional inequalities. This is our major operation for constructing subtype orderings, since it generalises ORDER and APPLYSUBST .

Definition 1.10.13

Let L be a sequence of assertions of the form

- $L = \tau_1 \leq \tau'_1 ; \dots ; \tau_n \leq \tau'_n$

or

- $L = \text{empty}$

We define the *enrichment* of a subtype ordering by the sequence L as follows :

- $\text{ENRICH}(\diamond, L) = \diamond, ID$
- $\text{ENRICH}(X, \text{empty}) = X, ID$

- $\text{ENRICH}(X, \tau_1 \leq \tau'_1; \dots; \tau_n \leq \tau'_n) = \diamond, ID$
if $\text{MATCH2}(X, \tau_1 \times \dots \times \tau_n, \tau'_1 \times \dots \times \tau'_n)$ fails
- $\text{ENRICH}(X, \tau_1 \leq \tau'_1; \dots; \tau_n \leq \tau'_n) = Z, RS$
where $\text{MATCH2}(X, \tau_1 \times \dots \times \tau_n, \tau'_1 \times \dots \times \tau'_n) = S$
and $\text{APPLYSUBST}(S, X) = Y, S$
and $\text{ORDERSET}(S\tau_1 \times \dots \times S\tau_n \leq S\tau'_1 \times \dots \times S\tau'_n) = \{B_1, \dots, B_m\}$
and $\text{UNION}(Y, B_1, \dots, B_m) = Z, R$

Note that in the above definition R is atomic and S is an expansion. Note also that $\text{APPLYSUBST}(S, X) = Y, S$ by corollary 1.10.9. The following theorem and corollaries give the properties of ENRICH :

Theorem 1.10.14

If $\text{ENRICH}(\leq, \tau_1 \leq \tau'_1; \dots; \tau_n \leq \tau'_n) = \leq', RS$, then $RS\tau_i \leq' RS\tau'_i$ for $1 \leq i \leq n$, and $\tau \leq \tau'$ implies $RS\tau \leq' RS\tau'$ for all τ, τ' . Furthermore, \leq', RS is minimal such that these conditions hold.

Proof : Let R be atomic and S an expansion, as in definition 1.10.13 :

- The first part follows by corollary 1.10.3 and theorems 1.8.5, 1.10.5.
- For minimality, let \leq_2, R_2 satisfy the conditions. Then $R_2\tau_i$ matches $R_2\tau'_i$, and R_2 is shape-consistent with respect to \leq , so by theorem 1.7.13, $R_2 = S_1S$ for some S_1 . Also by theorem 1.8.5, since \leq', R is minimal, $S_1S\tau \leq_2 S_1S\tau'$ and $S_1S\tau_i \leq_2 S_1S\tau'_i$, we have $S_1 = S_2R$ for some S_2 , and hence $R_2 = S_2RS$. Also $RS\tau \leq' RS\tau'$ implies $S_2RS\tau \leq_2 S_2RS\tau'$. \square

Corollary 1.10.15

Let τ, τ' be matching types. Then $\text{ENRICH}(\Pi, \tau \leq \tau') \approx \text{ORDER}(\tau \leq \tau')$.

Corollary 1.10.16

Let $\text{UNION}(\leq_1, \dots, \leq_n) = \leq', R$. Then there is a sequence L such that $\text{ENRICH}(\leq_1, L) = \leq', R$.

Proof : Construct L by concatenating the least splittings of \leq_2, \dots, \leq_n . \square

Corollary 1.10.17

Let S be any substitution, and \leq a subtype ordering. Then there is a sequence L such that $\text{ENRICH}(\leq, L) = \text{APPLYSUBST}(S, \leq)$.

Proof: Without loss of generality, $S = [\tau_1 / \alpha_1, \dots, \tau_n / \alpha_n, a_1 / \beta_1, \dots, a_m / \beta_m]$, where the α_i have rank 0, and the a_j, β_j have rank $\neq 0$. Form τ'_j, τ''_j from a_j, β_j respectively by providing a sufficient number of replications of *void* as arguments.

Then $L = \tau_1 \leq \alpha_1; \alpha_1 \leq \tau_1; \dots; \tau_n \leq \alpha_n; \alpha_n \leq \tau_n;$
 $\tau'_1 \leq \tau''_1; \tau''_1 \leq \tau'_1; \dots; \tau'_m \leq \tau''_m; \tau''_m \leq \tau'_m \quad \square$

Corollary 1.10.18

Let $\text{ENRICH}(\leq, L) = \leq', RS$. Then $\text{APPLYSUBST}(RS, \leq) = \leq'', RS$ and $\text{ENRICH}(\leq', L') \approx \text{ENRICH}(\leq'', RSL; L')$. In particular if $L' = \text{empty}$, $\text{ENRICH}(\leq'', RSL) = \leq', ID$.

Corollary 1.10.19

Let $\text{ENRICH}(\leq_1, L_1) = \leq_2, S_1 \dots \text{ENRICH}(\leq_n, L_n) = \leq_{n+1}, S_n$.
 Then there exists \leq'_1, L such that $\text{APPLYSUBST}(R, \leq_1) = \leq'_1, R$, for $R = S_n \dots S_1$, and $\text{ENRICH}(\leq'_1, L) = \leq_{n+1}, ID$.

Proof: By induction using corollary 1.10.18 for the basis case, and corollaries 1.10.18 and 1.10.12 for the induction case. \square

Corollary 1.10.20

Corollary 1.10.19 holds if we replace any of the initial ENRICH operations by an equivalent APPLYSUBST.

Proof: Using corollary 1.10.11. \square

The last two corollaries motivate the following definition of the operation SUBEN. Any sequence of APPLYSUBST and ENRICH operations can be replaced by one use of SUBEN, and we will therefore find SUBEN to be the most useful operation on subtype orderings.

Definition 1.10.21

If $\text{APPLYSUBST}(R, \leq) = \leq', R$ and $\text{ENRICH}(\leq', L) = \leq'', ID$ then we write $\text{SUBEN}(R, \leq, L) = \leq''$.

Having now defined all our operations for manipulating subtype orderings, we turn our attention to the notions of *type scheme* and *assumption set*, which we will require for type inference.

1.11 Type Schemes and Assumption Sets

Definition 1.11.1

We obtain *polymorphism* by means of *type schemes* as in Damas and Milner (1982). Our type schemes will contain subtype orderings, with the intention that they be instantiated in such a way that the inequalities in the subtype orderings are satisfied. Type schemes will have the form :

$$\bullet \quad \forall \bar{\alpha} : \leq . \tau$$

where $\bar{\alpha}$ is a vector of type variables, \leq is a subtype ordering on type variables $\bar{\alpha}$, and τ is a type, and $\bar{\alpha}$ contains all the type variables in τ . In other words, we insist that type schemes are *closed*.

If \leq has only isolated type variables we write simply : $\forall \bar{\alpha} . \tau$, and if in addition $\bar{\alpha}$ is empty (in which case τ must be a closed type) we write $\forall . \tau$. Hence types and type schemes are completely disjoint.

Remark 1.11.2

The reason we insist that type schemes be closed is to disallow expressions of the form :

$\lambda x. \text{let } y = \dots x \dots \text{ in } \dots$

Such an expression, involving an interaction between bound and free type variables would be very difficult to type-check with our scheme. This is because applying a substitution to an open type scheme involves applying an arbitrary substitution to a subtype ordering. This is an operation we have defined (APPLYSUBST), but it returns an extended substitution. Thus, when applying a substitution to a set of type schemes, each application could extend the substitution, which would in turn affect the result of applying the substitution to the other type schemes. Therefore applying a substitution to an *assumption set*, which can contain several type schemes, is a difficult operation to define. We avoid this problem by insisting that all type schemes be closed. It is worth noting that in the functional language Miranda such an

expression cannot be constructed, because the true polymorphic **let** occurs only at the top level of function definition. This supports our decision, and indicates that we are not seriously compromising usability.

The type inference algorithm that we will outline in chapter 3 integrates inference of types for non-polymorphic variables (bound by λ) with direct use of known type schemes for polymorphic variables (bound by **let**). This is facilitated by ensuring types and type schemes are disjoint.

Definition 1.11.3

Since type schemes are closed, we define $S\sigma = \sigma$ for all substitutions S and type schemes σ .

Definition 1.11.4

An *assumption set* is a set of assumptions of the form $x : \tau$ or $x : \sigma$ where x is a variable in our functional language, τ is a type and σ is a type scheme. We use A and B to denote assumption sets in the remainder of this thesis. We write A_x for the result of removing any assumption $x : \tau$ or $x : \sigma$ from A , and define the free type variables of an assumption set by :

$$\begin{aligned} & \bullet \quad FV(\{x_1 : \sigma_1, \dots, x_n : \sigma_n, y_1 : \tau_1, \dots, y_m : \tau_m\}) \\ & \quad = FV(\tau_1) \cup \dots \cup FV(\tau_m) \end{aligned}$$

We define the *domain* of an assumption set by :

$$\begin{aligned} & \bullet \quad Dom(\{x_1 : \sigma_1, \dots, x_n : \sigma_n, y_1 : \tau_1, \dots, y_m : \tau_m\}) \\ & \quad = \{x_1, \dots, x_n, y_1, \dots, y_m\} \end{aligned}$$

We can then write $x \in A$, $x \notin A$ if $x \in Dom(A)$, $x \notin Dom(A)$ respectively. We will say that two assumption sets are *disjoint* if their domains are disjoint. We define SA for substitutions S in the obvious way, applying S to the types and type schemes in A .

If $x : \tau \in A$ we will also write $A(x) = \tau$.

If $Dom(A') \subseteq Dom(A)$, and A contains only types, and for some subtype ordering \leq , $A(x) \leq A'(x)$ for each $x \in Dom(A')$, then we will write $A \leq A'$.

Definition 1.11.5

We will use the following initial assumption set in our examples, to give the type schemes for predefined primitive functions :

- $A_0 = \{fst : \forall \alpha \beta. \alpha \times \beta \rightarrow \alpha,$
 $snd : \forall \alpha \beta. \alpha \times \beta \rightarrow \beta,$
 $pair : \forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \alpha \times \beta,$
 $inl : \forall \alpha \beta. \alpha \rightarrow \alpha + \beta,$
 $inr : \forall \alpha \beta. \beta \rightarrow \alpha + \beta,$
 $case : \forall \alpha \beta \gamma. (\alpha \rightarrow \gamma) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha + \beta) \rightarrow \gamma,$
 $cons : \forall \alpha. \alpha \rightarrow list \alpha \rightarrow list \alpha,$
 $nil : \forall \alpha. list \alpha$
 $lchoose : \forall \alpha \beta. \beta \rightarrow (\alpha \rightarrow list \alpha \rightarrow \beta) \rightarrow list \alpha \rightarrow \beta$
 $absent : \forall \alpha. optional \alpha$
 $present : \forall \alpha. \alpha \rightarrow optional \alpha$
 $ochoose : \forall \alpha \beta. \beta \rightarrow (\alpha \rightarrow \beta) \rightarrow optional \alpha \rightarrow \beta$
 $fix : \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow \alpha,$
 $plus : \forall \alpha. [\alpha \leq int] \alpha \rightarrow \alpha \rightarrow \alpha,$
 $neg : \forall. int \rightarrow int$
 $and : \forall. bool \rightarrow bool \rightarrow bool$
 $cond : \forall \alpha. bool \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$
 $eq : \forall \zeta. \zeta \rightarrow \zeta \rightarrow bool\}$

where α, β, γ are uncoloured type variables of rank 0, and ζ is a basic type variable of rank 0. The behaviour of the predefined primitive functions will be given in section 2.5.

Definition 1.11.6

We say that the pair $RS\tau, \leq'$ is a *generic instance* of a type scheme $\forall \alpha: \leq. \tau$ if S is an instantiation of the type variables α and $APPLYSUBST(S, \leq) = \leq', RS$.

For example, the type scheme A_0 (*plus*) has generic instances :

- $nat \rightarrow nat \rightarrow nat, \Pi$
- $int \rightarrow int \rightarrow int, \Pi$
- $\xi \rightarrow \xi \rightarrow \xi, [\xi \leq int]$

Note that as well as producing a type, as in the usual definition of generic instantiation (Clement et al 1986), our definition also produces an altered subtype ordering.

Definition 1.11.7

If A is an assumption set, \leq is a subtype ordering with type variables W , and τ is a type, we define the corresponding type scheme $gen(A, \leq, \tau)$ by :

$$\bullet \quad gen(A, \leq, \tau) = \forall \alpha_1 \dots \alpha_n : \leq'. \tau$$

where $W \cup FV(\tau) = \{\alpha_1, \dots, \alpha_n\}$, \leq' is the family of partial orders \leq on $\{\alpha_1, \dots, \alpha_n\}$, and $n \geq 0$, provided that $\alpha_i \notin FV(A)$ for $1 \leq i \leq n$. If the proviso fails, gen is undefined. However, if $A = \{\}$, gen is always defined.

This check will ensure that non-polymorphic variables are not used inside the **let**-part of **let**-expressions, as discussed in remark 1.11.2.

We now have all the machinery that we require for our type inference system. The next sections will define a functional language, and use this machinery to define its type inference algorithm.

2 A FUNCTIONAL LANGUAGE AND TYPE SYSTEM

In this chapter we use the above notation to define a simple functional language with implicit typing. We give a set of *type inference rules* for the language, which allow us to infer from an expression a *typed expression*, which is the original expression annotated with type information. This is similar to the approach given by Mitchell (1984), and in contrast to the usual approach (Damas and Milner 1982) where only a type is inferred. We do, however, extend Mitchell's approach by including *type polymorphism*.

Section 2.1 defines *expressions* and *typed expressions* for our simple language, and distinguishes polymorphic and nonpolymorphic occurrences of program variables in typed expressions. Section 2.2 defines *replacements* of nonpolymorphic variables, which allow beta-reduction to be defined, and *replications*, which allow let-reduction to be defined. Section 2.3 defines the type inference rules for our system, and gives two example derivations of typed expressions using these rules. In section 2.4 we present some derived rules which will be useful in our inference algorithm. In particular, the application of APPLYSUBST, ENRICH and SUBEN operations to a derivation is valid.

Section 2.5 presents a rewrite-rule semantics for our language using the example type hierarchy, and proves the rewrite rules preserve well-typing. The Church-Rosser property for these rules is proved in section 2.6. Section 2.7 defines Böhm trees modulo extensional equality as the natural *value* of a typed expression. Section 2.8 shows that coercions in our semantics have the required properties of producing a unique result, being injective on equality types and commuting with the polymorphic primitive functions. We conjecture that this means that different type-annotations of an expression have the same value, thus ensuring that implicit coercions introduce no ambiguity.

2.1 The Functional Language

Definition 2.1.1

We assume a set *PV* of *program variables*. The *expressions* of our functional language are then defined as follows, using a notion of *limited expression* :

- if x is a program variable, then x is a limited expression.
- if c is a constant identifier, ie $c \in K(\pi)$ for some primitive type π , then c is a limited expression.
- if x is a program variable and e is a limited expression, then $\lambda x.e$ is a limited expression.
- if e and e' are limited expressions then $(e\ e')$ is a limited expression.

We refer to $(e\ e')$ as an *application*, and omit the parentheses where no ambiguity results, given that application associates to the left. We can now define expressions, which may involve the polymorphic **let** construct :

- a limited expression is an expression.
- if x is a program variable, e is a limited expression, and e' is an expression, then **let** $x = e$ **in** e' is an expression.

In general, expressions are of the form :

let $x_1 = e_1$ **in** ... **let** $x_n = e_n$ **in** e

where e_1, \dots, e_n, e are limited expressions, and $n \geq 0$. This corresponds to functional languages such as Miranda™, where a list of polymorphic function definitions is followed by an expression to be evaluated, and uses of **let** and **where** within that expression are not polymorphic, but are merely shorthand for an application of a lambda abstraction. This restriction of polymorphism to the top level greatly simplifies type inference, as we have already discussed, without seriously compromising usability.

We may allow certain program variables to be declared as mixfix operators with a particular arity n , so that there will be a special syntax for $x e_1 \dots e_n$. The variables *pair*, *case* and *cons* in A_0 might be so declared. We do not discuss this issue further, but refer the reader to Holyer (1986).

Definition 2.1.2

We say that a program variable x *occurs free* in an expression e , written $x \in e$, if x occurs in e , and x is not bound by an enclosing λ or **let** in e .

Unlike most typing systems, we do not attach any meaning to expressions *before* typing, but only to *typed expressions* which are described in the following definition. Section 2.3 describes how many such typed expressions can be inferred from an expression, but section 2.7 will show that these typed expressions all, in some sense, represent the same value. We can therefore still speak about *the* meaning of an expression.

Definition 2.1.3

The *typed expressions* of our language result from attaching type information to all subexpressions of an expression, and are defined as follows :

- if x is a program variable and τ is a type, then $x : \tau$ is a limited typed expression.
- if x is a program variable, and t_1, \dots, t_n are types, type constructors, or type variables (referred to as the *type parameters* of x), and τ is a type, then $x [t_1, \dots, t_n] : \tau$ is a limited typed expression.
- if c is a constant identifier and $c \in K(\pi)$, then $c : \pi$ is a limited typed expression.
- if x is a program variable, $e : \tau$ is a limited typed expression, and τ' is a type, then $(\lambda x : \tau'. e : \tau) : \tau' \rightarrow \tau$ and $(\lambda x : \tau'. e : \tau) : \tau' \supset \tau$ are limited typed expressions.
- if $e : a\tau'\tau$ and $e' : \tau'$ are limited typed expressions, where a is a type variable or type constructor of rank $*$, then $(e : a\tau'\tau) (e' : \tau') : \tau$ is a limited typed expression.
- if $e : \tau$ is a limited typed expression and τ' is a type, then $(e : \tau) : \tau'$ is a limited typed expression.
- a limited typed expression is a typed expression.
- if x is a program variable, $e : \tau$ is a limited typed expression, $e' : \tau'$ is a typed expression, and $\alpha_1, \dots, \alpha_n$ are type variables, then $(\text{let } x = [\alpha_1, \dots, \alpha_n] e : \tau \text{ in } e' : \tau') : \tau'$ is a typed expression.

We intend $x : \tau$ to represent a program variable bound by an enclosing λ , and $x [\bar{t}] : \tau$ to represent a variable bound by an enclosing polymorphic **let**. The t_i are the substituted bound variables of the polymorphic type scheme after generic instantiation, and τ is the substituted type. For example :

$$\begin{aligned} & \text{let } x = [\alpha] (\lambda y : \alpha. y : \alpha) : \alpha \rightarrow \alpha \\ & \text{in } (x [\text{nat}] : \text{nat} \rightarrow \text{nat}) (\mathbf{3} : \text{nat}) : \text{nat} \end{aligned}$$

where x has the type scheme $\forall \alpha. \alpha \rightarrow \alpha$ with generic instance $\text{nat} \rightarrow \text{nat}$, Π . This example corresponds to :

$$(\Lambda \alpha. \lambda y : \alpha. y) [\text{nat}] \mathbf{3}$$

in the second-order lambda calculus (Reynolds 1985). The variable x has a polymorphic value which takes a type parameter (in this case nat) which is substituted for α to obtain a λ -abstraction which expects a further argument (in this case **3**).

We intend $(e : \tau) : \tau'$ to represent the coercion of the expression e of type τ to the corresponding expression of type τ' . We will therefore require $\tau \leq \tau'$ for some subtype ordering \leq . The typed expression $c : \pi$ represents a constant identifier of type π , while the two forms of λ -abstraction are used to construct functions of the usual kind (\rightarrow) as well as constant functions (\supset). For application, we permit both kinds of functions, as well as functional types formed with type variables of rank $*$, but the argument type of the function must be equal to the type of the expression it is applied to. For example :

$$((\lambda x : bool. \mathbf{3} : nat) : bool \supset nat) (\mathbf{true} : bool) : nat$$

Definition 2.1.4

If $e : \tau$ is a typed expression, we write $untype (e : \tau)$ for the expression that results by removing all type information in the obvious way. For example :

$$\begin{aligned} untype (((\lambda x : bool. \mathbf{3} : nat) : bool \supset nat) (\mathbf{true} : bool) : nat) \\ = (\lambda x. \mathbf{3}) \mathbf{true} \end{aligned}$$

Definition 2.1.5

We say that a program variable x occurs *polymorphically free* in $e : \tau$ if there is an occurrence of the form $x[\overline{t}] : \tau'$ which is not bound by an enclosing λ or **let**, and that x occurs *nonpolymorphically free* if there is an occurrence of the form $x : \tau'$ which is not bound by an enclosing λ or **let**.

It is clear that if x occurs free (polymorphically or nonpolymorphically) in $e : \tau$, then $x \in untype (e : \tau)$.

This distinction between polymorphic and nonpolymorphic occurrences, mentioned previously in remark 1.11.2, will assist us in developing the type inference algorithm.

Restriction 2.1.6

We now make some restrictions to typed expressions that ensure that program variables are used in a way that corresponds to their binding :

- in $(\lambda x : \tau'. e : \tau) : \tau' \supset \tau$, x does not occur free in $e : \tau$
- in $(\lambda x : \tau'. e : \tau) : \tau' \rightarrow \tau$, x does not occur polymorphically free in $e : \tau$, and all nonpolymorphic free occurrences are of the form $x : \tau'$.
- in $(\text{let } x = [\bar{\alpha}] e : \tau \text{ in } e' : \tau') : \tau'$, no program variable occurs nonpolymorphically free in $e : \tau$ or $e' : \tau'$, and all polymorphic free occurrences of x in $e' : \tau'$ are of the form $x[\bar{t}_i] : \tau_i$, where $[\bar{t}_i / \bar{\alpha}] \tau' = \tau_i$

We assume that in future all typed expressions satisfy these restrictions. This ensures that beta-reduction and **let**-reduction will be possible.

Definition 2.1.7

If S is a substitution, then for each typed expression $e : \tau$ we define $S(e : \tau)$ to be the application of the substitution S to all the types in $e : \tau$ with the exception that :

$$\begin{aligned} & S((\text{let } x = [\bar{\alpha}] e : \tau \text{ in } e' : \tau') : \tau') \\ &= (\text{let } x = [\bar{\alpha}] e : \tau \text{ in } S(e' : \tau')) : S\tau' \end{aligned}$$

Proposition 2.1.8

If S is a substitution, and $e : \tau$ is a (limited) typed expression, then $S(e : \tau)$ is a (limited) typed expression.

Proof : By induction on the structure of e . \square

2.2 Replacements and Replications

To permit beta-reduction and **let**-reduction on typed expressions, we define the notions of *replacement* and *replication*. We will use these in section 2.5, when we define a rewrite-rule semantics.

Definition 2.2.1

We write a *replacement* as $Q = [e_1 : \tau_1/x_1 : \tau_1, \dots, e_n : \tau_n/x_n : \tau_n]$ where each x_i is a program variable and each $e_i : \tau_i$ is a limited typed expression. We define $Q(e : \tau)$ only if all free occurrences of the x_i in $e : \tau$ are nonpolymorphic and of the form $x_i : \tau_i$, in which case $Q(e : \tau)$ is the result of replacing all such occurrences by $e_i : \tau_i$.

Definition 2.2.2

If S is a substitution, we define :

$$\begin{aligned} S [e_1 : \tau_1/x_1 : \tau_1, \dots, e_n : \tau_n/x_n : \tau_n] \\ = [S(e_1 : \tau_1)/x_1 : S\tau_1, \dots, S(e_n : \tau_n)/x_n : S\tau_n] \end{aligned}$$

Proposition 2.2.3

If S is a substitution and Q is a replacement, then if $S(Q(e : \tau))$ is defined, then so is $SQ(S(e : \tau))$, and when they are both defined they have the same value.

Proof : By induction. \square

Definition 2.2.4

For the special case that $Q = [(x_1 : \tau'_1) : \tau_1/x_1 : \tau_1, \dots, (x_n : \tau'_n) : \tau_n/x_n : \tau_n]$ and $A = \{x_1 : \tau_1, \dots, x_m : \tau_m\} \cup A'$ is an assumption set, for $m \leq n$, we define :

$$QA = \{x_1 : \tau'_1, \dots, x_m : \tau'_m\} \cup A'$$

Clearly if $\tau'_i \leq \tau_i$ for each i , and A contains only types, then $QA \leq A$. This operation on assumption sets will become useful in section 3.1 when we attempt to ‘unify’ assumption sets.

The following definition allows us to define **let**-reduction by replacing polymorphic program variables :

Definition 2.2.5

We write a *replication* as $Z = [[\bar{\alpha}] e' : \tau // x]$ where x is a program variable and $e' : \tau$ is a limited type expression. We define $Z(e : \tau)$ only if all free occurrences of x in $e : \tau$ are polymorphic and of the form $x[\bar{t}_i] : \tau_i$ for some \bar{t}_i , where $[\bar{t}_i / \bar{\alpha}] \tau = \tau_i$. In this case $Z(e : \tau)$ is the result of replacing each such occurrence by $[\bar{t}_i / \bar{\alpha}] (e' : \tau)$.

2.3 The Type Inference System

Definition 2.3.1

We can now present the inference rules which define the *legal* typed expressions. An assertion of the form :

$$\bullet \quad A, \leq, e \vdash e' : \tau$$

indicates that $e' : \tau$ is a legal typed expression obtained by type-annotating the expression e , given the assumption set A and the subtype ordering \leq . The inference rules are as follows :

$$\begin{array}{l} \text{PVAR} \quad x : \forall \bar{\alpha} : \leq. \tau \in A \\ \quad \text{SUBEN } (S, \leq, L) = \leq' \text{ for some } L \\ \hline SA, \leq', x \vdash x [S \bar{\alpha}] : S\tau \end{array}$$

$$\begin{array}{l} \text{VAR} \quad x : \tau \in A \\ \hline A, \leq, x \vdash x : \tau \end{array}$$

$$\begin{array}{l} \text{CONST} \quad c \in K(\pi) \\ \hline A, \leq, c \vdash c : \pi \end{array}$$

$$\begin{array}{l} \text{AP} \quad A, \leq, e_1 \vdash e'_1 : a\tau'\tau, \text{ where } a \leq \rightarrow \\ \quad A, \leq, e_2 \vdash e'_2 : \tau' \\ \hline A, \leq, e_1 e_2 \vdash (e'_1 : a\tau'\tau) (e'_2 : \tau') : \tau \end{array}$$

$$\begin{array}{l} \text{LESS} \quad A, \leq, e \vdash e' : \tau \\ \quad \tau \leq \tau' \\ \hline A, \leq, e \vdash (e' : \tau) : \tau' \end{array}$$

ABS	$\frac{A_x \cup \{x : \tau'\}, \leq, e \vdash e' : \tau}{A, \leq, \lambda x. e \vdash (\lambda x : \tau'. e' : \tau) : \tau' \rightarrow \tau}$
ABS2	$\frac{A_x, \leq, e \vdash e' : \tau}{A, \leq, \lambda x. e \vdash (\lambda x : \tau'. e' : \tau) : \tau \supset \tau}$
LET	$\frac{\begin{array}{l} A, \leq, e_1 \vdash e'_1 : \tau \quad \text{where } A \text{ contains only type schemes} \\ \text{gen}(A, \leq, \tau) = \sigma = \forall \bar{\alpha} : \leq'. \tau \\ A_x \cup \{x : \sigma\}, \leq'', e_2 \vdash e'_2 : \tau \end{array}}{A, \leq'', \text{let } x = e_1 \text{ in } e_2 \vdash (\text{let } x = [\bar{\alpha}] e'_1 : \tau \text{ in } e'_2 : \tau) : \tau}$

We require that in the AP, LESS, ABS and ABS2 rules the derivations of the antecedents involve limited typed expressions only, ie they cannot be derived using the LET rule. We can then prove the following proposition :

Proposition 2.3.2

If $A, \leq, e \vdash e' : \tau$, then $e' : \tau$ is a typed expression satisfying restriction 2.1.6, such that $\text{untype}(e' : \tau) = e$. Furthermore, for each program variable x , if $A(x) = \tau'$ then all free occurrences of x in $e' : \tau$ are nonpolymorphic and of the form $x : \tau'$; if $A(x) = \forall \bar{\alpha} : \leq'. \tau'$ then all free occurrences of x in $e' : \tau$ are polymorphic and of the form $x[\bar{t}_i] : \tau_i$ where $[\bar{t}_i / \bar{\alpha}] \tau' = \tau_i$; and if $x \notin A$ then x has no free occurrences in $e' : \tau$.

Proof : By induction on derivations. \square

Example 2.3.3

Consider the following expression, representing the increment function :

- $\lambda x. \text{plus } x \ 1$

with the initial assumption set A_0 given in section 1.11.5, and the primitive subtype ordering Π . We derive a corresponding typed expression below, where we first define A_1 and \leq_1 :

- 1 $A_1 = A_0 \cup \{x : \beta\}$
- 2 $\leq_1 = \{\Pi, [nat \leq \alpha], [\alpha \leq int], [\beta \leq \alpha]\}$

Note that we have :

- 3 $SUBEN (ID, [\alpha \leq int], nat \leq \alpha; \beta \leq \alpha) = \leq_1$

The derivation then proceeds as follows, where we indicate the rules and steps used at the right :

- 4 $A_1, \leq_1, plus \vdash plus [\alpha] : \alpha \rightarrow \alpha \rightarrow \alpha$ (PVAR,3)
- 5 $A_1, \leq_1, x \vdash x : \beta$ (VAR, 1)
- 6 $A_1, \leq_1, x \vdash (x : \beta) : \alpha$ (LESS, 2, 5)
- 7 $A_1, \leq_1, plus x \vdash (plus [\alpha] : \alpha \rightarrow \alpha \rightarrow \alpha)$
 $(x : \beta) : \alpha$
 $: \alpha \rightarrow \alpha$ (AP, 4, 6)
- 8 $A_1, \leq_1, 1 \vdash 1 : nat$ (CONST)
- 9 $A_1, \leq_1, 1 \vdash (1 : nat) : \alpha$ (LESS, 2, 8)
- 10 $A_1, \leq_1, plus x 1 \vdash ((plus [\alpha] : \alpha \rightarrow \alpha \rightarrow \alpha)$
 $(x : \beta) : \alpha$
 $: \alpha \rightarrow \alpha)$
 $((1 : nat) : \alpha)$
 $: \alpha$ (AP, 7, 9)
- 11 $A_0, \leq_1, \lambda x. plus x 1 \vdash$
 $(\lambda x : \beta. ((plus [\alpha] : \alpha \rightarrow \alpha \rightarrow \alpha)$
 $(x : \beta) : \alpha$
 $: \alpha \rightarrow \alpha)$
 $((1 : nat) : \alpha)$
 $: \alpha$
 $): \beta \rightarrow \alpha$ (ABS, 1, 10)

Thus we have derived the type $\beta \rightarrow \alpha$ for the given expression, given the constraints $nat \leq \alpha$, $\alpha \leq int$ and $\beta \leq \alpha$. We could also derive the types $nat \rightarrow int$, $nat \rightarrow \alpha$, $int \rightarrow atom$, etc. The type we have derived is, however, the ‘most general’ (or *principle*) type in the sense that if :

$$A_0, \leq_2, \lambda x. plus x 1 \vdash e : \tau$$

then $SUBEN (S, \leq_1, L) = \leq_2$ for some S, L (1)

and $S (\beta \rightarrow \alpha) \leq_2 \tau$ (2)

The algorithm we will outline in section 3.2 will infer this ‘most general’ type from the expression $\lambda x. \text{plus } x \ 1$.

We can derive a better type than this ‘most general’ type, however. In section 2.5 we will only give a semantics for typed expressions derived with the primitive subtype ordering Π , ie with types τ such that equation (1) above is $S(\leq_1) = \Pi$ and equation (2) is $S(\beta \rightarrow \alpha) \ll \tau$. The only such types are $\text{nat} \rightarrow \text{nat}$, $\text{nat} \rightarrow \text{int}$, $\text{nat} \rightarrow \text{atom}$, $\text{int} \rightarrow \text{int}$ and $\text{int} \rightarrow \text{atom}$, and of these $\text{nat} \rightarrow \text{nat}$ and $\text{int} \rightarrow \text{int}$ are most general. We therefore assert that the ‘best’ type for the given expression is $\alpha \rightarrow \alpha$ with the constraints $\text{nat} \leq \alpha$ and $\alpha \leq \text{int}$. From this type we can derive $\text{nat} \rightarrow \text{nat}$ and $\text{int} \rightarrow \text{int}$ by instantiating α , and hence the other three types listed. We thus have the property that if

$$A_0, \Pi, \lambda x. \text{plus } x \ 1 \vdash e : \tau$$

then $S\alpha \in \{\text{nat}, \text{int}\}$ and $S(\alpha \rightarrow \alpha) \ll \tau$ for some S

In sections 3.3 to 3.6 we will show how such a ‘best’ type can be derived.

Example 2.3.4

As a slightly more complex example, consider the expression below, representing an infinite list of 1’s :

$$\bullet \quad \text{fix } (\lambda x. \text{cons } 1 \ x)$$

We define A_1 in this example by :

$$1 \quad A_1 = A_0 \cup \{x : \text{list } \alpha\}$$

We also define \leq_1 by :

$$2 \quad \leq_1 = \{\Pi, [\text{nat} \leq \alpha]\}$$

The derivation then proceeds as follows :

$$3 \quad A_1, \leq_1, \text{cons} \vdash \text{cons } [\alpha] : \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha \quad (\text{PVAR}, 2)$$

$$4 \quad A_1, \leq_1, 1 \vdash (1 : \text{nat}) : \alpha \quad (\text{CONST}, \text{LESS}, 2)$$

$$5 \quad A_1, \leq_1, \text{cons } 1 \vdash (\text{cons } [\alpha] : \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha) \\ ((1 : \text{nat}) : \alpha)$$

$$: \text{list } \alpha \rightarrow \text{list } \alpha \quad (\text{AP}, 3, 4)$$

$$6 \quad A_1, \leq_1, x \vdash x : \text{list } \alpha \quad (\text{VAR})$$

- 7 $A_0, \leq_1, \lambda x. \text{cons } 1 \ x \vdash$
 $(\lambda x : \text{list } \alpha. ((\text{cons } [\alpha] : \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha)$
 $((1 : \text{nat}) : \alpha)$
 $: \text{list } \alpha \rightarrow \text{list } \alpha)$
 $(x : \text{list } \alpha)$
 $: \text{list } \alpha$
 $) : \text{list } \alpha \rightarrow \text{list } \alpha \quad (\text{AP}, 5, 6, \text{ABS}, 1)$
- 8 $A_0, \leq_1, \text{fix} \vdash \text{fix } [\text{list } \alpha] : (\text{list } \alpha \rightarrow \text{list } \alpha) \rightarrow \text{list } \alpha \quad (\text{PVAR})$
- 9 $A_0, \leq_1, \text{fix } (\lambda x. \text{cons } 1 \ x) \vdash$
 $(\text{fix } [\text{list } \alpha] : (\text{list } \alpha \rightarrow \text{list } \alpha) \rightarrow \text{list } \alpha)$
 $((\lambda x : \text{list } \alpha. ((\text{cons } [\alpha] : \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha)$
 $((1 : \text{nat}) : \alpha)$
 $: \text{list } \alpha \rightarrow \text{list } \alpha)$
 $(x : \text{list } \alpha)$
 $: \text{list } \alpha$
 $) : \text{list } \alpha \rightarrow \text{list } \alpha)$
 $: \text{list } \alpha \quad (\text{AP}, 7, 8)$

Thus we have derived the type $\text{list } \alpha$ for the given expression, with the constraint $\text{nat} \leq \alpha$. This is the most general type. We could also derive the types $\text{list } \text{nat}$, $\text{list } \text{int}$ and $\text{list } \text{atom}$ for this example, and it is clear that the ‘best’ type is $\text{list } \text{nat}$. We will return to this example in examples 3.2.8 and 3.6.7 to demonstrate the action of our type inference algorithms.

Remark 2.3.5

We have used *fix* as the only way of obtaining recursion. A *letrec* construct would be a more natural way of expressing recursive definitions, but has some problems associated with it (Mycroft 1984). The section by Hancock in Peyton-Jones (1987) describes a type discipline using *letrec* that avoids these problems, which is equivalent to the use of *fix*.

2.4 Derived Inference Rules

We now extend the inference system of the previous section with some derived rules which will be useful in our inference algorithm. In previous work (Dekker 1987) some of these rules appeared as part of the inference system itself.

Proposition 2.4.1

The following rule is valid :

$$\begin{array}{l}
 \text{SUBA} \quad A' \leq, e \vdash e' : \tau \\
 \text{where } A' \subseteq A \text{ contains only type schemes and } S \text{ is any substitution} \\
 \hline
 SA, \leq, e \vdash e' : \tau
 \end{array}$$

Proof: By induction, since $SA' = A'$. \square

Proposition 2.4.2

The following rule is valid :

$$\begin{array}{l}
 \text{SUBEN} \quad A \leq, e \vdash e' : \tau \\
 \text{SUBEN } (S, \leq, L) = \leq' \\
 \hline
 SA, \leq', e \vdash S(e' : \tau)
 \end{array}$$

Proof: By induction. For the PVAR case we use corollaries 1.10.19 and 1.10.20 with definition 1.10.21. For the LET case we use SUBA for the first antecedent and the induction hypothesis for the second, the result following by definition 2.1.7. The other cases are trivial. \square

We note that by corollaries 1.10.19 and 1.10.20 any sequence of APPLYSUBST and ENRICH operations can be combined into one SUBEN operation. We thus have the following corollaries :

Corollary 2.4.3

The following rule is valid as a special case of SUBEN :

$$\begin{array}{l}
\text{ENRICH} \quad A, \leq, e \vdash e' : \tau \\
\text{ENRICH } (\leq, L) = \leq', S \\
\hline
SA, \leq', e \vdash S(e' : \tau)
\end{array}$$

Corollary 2.4.4

The following rule is valid as a special case of SUBEN :

$$\begin{array}{l}
\text{SUBST} \quad A, \leq, e \vdash e' : \tau \\
\text{APPLYSUBST } (S, \leq) = \leq', RS \\
\hline
RSA, \leq', e \vdash RS(e' : \tau)
\end{array}$$

We now turn our attention to adding program variables to assumption sets.

Proposition 2.4.5

The following rule is valid :

$$\begin{array}{l}
\text{ASSUMPN} \quad A, \leq, e \vdash e' : \tau \\
A' = \{x_1 : \alpha_1, \dots, x_n : \alpha_n\} \\
\text{where } A' \text{ is disjoint from } A \text{ and the } \alpha_i \text{ are} \\
\text{not used in the above derivation} \\
\hline
A \cup A', \leq, e \vdash e' : \tau
\end{array}$$

Proof : By induction. \square

Corollary 2.4.6

The following rule is valid :

$$\begin{array}{l}
\text{ASSUMP} \quad A, \leq, e \vdash e' : \tau \\
A' \text{ is disjoint from } A \text{ and contains only types} \\
\hline
A \cup A', \leq, e \vdash e' : \tau
\end{array}$$

Proof : By ASSUMPN and SUBST. \square

We note that in this rule, if A contains only types, then trivially $A \cup A' \leq A$. The following proposition gives a corresponding result for adding type schemes :

Proposition 2.4.7

The following rule is valid :

$$\begin{array}{c}
 \text{ASSUMPS} \quad A, \leq, e \vdash e' : \tau \\
 A' \text{ is disjoint from } A \text{ and contains only type schemes} \\
 \hline
 A \cup A', \leq, e \vdash e' : \tau
 \end{array}$$

Proof: By induction. \square

Proposition 2.4.8

The following rule is valid, and the given replacement is defined :

$$\begin{array}{c}
 \text{REPLACE} \quad A, \leq, e \vdash e' : \tau \\
 \text{for } 1 \leq i \leq n, \tau'_i \leq \tau_i \text{ and } x_i \notin A \text{ or } A(x_i) = \tau_i \\
 Q = [x_1 : \tau'_1 : \tau_1 / x_1 : \tau_1, \dots, x_n : \tau'_n : \tau_n / x_n : \tau_n] \\
 \hline
 QA, \leq, e \vdash e'' : \tau
 \end{array}$$

Proof: By induction, using definitions 2.2.1 and 2.2.4, and proposition 2.3.2. Essentially uses of VAR for the x_i are replaced by VAR and LESS. \square

We note that if A contains only types then $QA \leq A$. This property will be useful in section 3.2. The final derived rule is the most complex, but is also useful in section 3.2.

Proposition 2.4.9

The following rule is valid, showing that we can always perform a derivation using a more general type scheme :

$$\begin{array}{c}
 \text{REPLACES} \quad A_x \cup \{x : \sigma_1\}, \leq, e \vdash e' : \tau \\
 \sigma_1 = \forall \bar{\alpha} : \leq_1. \tau_1 \\
 \sigma_2 = \forall \bar{\beta} : \leq_2. \tau_2 \\
 \text{where } \bar{\beta} \text{ is a vector of previously unused type variables,} \\
 \text{SUBEN } (S, \leq_2, L) = \leq_1, \text{Dom } (S) \subseteq \bar{\beta}, \\
 \text{and } S\tau_2 \leq_1 \tau_1 \\
 \hline
 A_x \cup \{x : \sigma_2\}, \leq, e \vdash e'' : \tau
 \end{array}$$

Proof: By induction, using corollaries 1.10.19 and 1.10.20, definition 1.10.21, the LESS rule, and the fact that $SA_x = A_x$ for the PVAR case. The other cases are trivial. \square

This last, fairly complex, rule will be used in the proof of completeness for our inference algorithm.

2.5 A Rewrite – Rule Semantics

We are now in a position to give an operational semantics for a subset of legal typed expressions. In particular, we give a semantics for typed expressions $e' : \tau$ such that :

$$A_0, \Pi, e \vdash e' : \tau$$

We provide a set of rewrite rules that is non-ambiguous and left-linear in the sense of Klop (1980), and hence has the Church-Rosser and Unique Normal Form properties. However, we first make a further restriction to typed expressions :

Restriction 2.5.1

For the remainder of section 2, we assume that no program variable in A_0 (ie *pair*, *fst*, etc) is bound by a λ or *let*, or occurs nonpolymorphically free. In other words, these program variables only occur polymorphically free in typed expressions.

The restriction simplifies the rewrite rules, since we can treat program variables in A_0 as constants. If necessary, program variables can be renamed to ensure that the restriction is satisfied. In the interests of readability, we make the following notational convention :

Convention 2.5.2

When writing typed expressions, we will omit some type information where the context makes this unambiguous. For example, instead of :

$$((\text{plus } [nat] : nat \rightarrow nat \rightarrow nat) (n : nat) : nat \rightarrow nat (m : nat) : nat$$

we will write :

$$\text{plus } (n : nat) (m : nat) : nat$$

Since we do not permit *plus* to occur nonpolymorphically and since the types of the arguments and of the result uniquely determine the omitted type information, no ambiguity can result.

Definition 2.5.3

We define a *program* to be a typed expression $e' : \tau$ such that :

$$A_0, \Pi, e \vdash e' : \tau$$

As stated above, these are the typed expressions for which we give a semantics. We have the following property of programs :

Proposition 2.5.4

If $e : \tau$ is a program then for each subexpression $e' : \tau'$ of $e : \tau$,

$$A, \leq, e'' \vdash e' : \tau'$$

where each program variable in A not in A_0 is bound by an enclosing λ or **let**, and each type variable in \leq is bound by an enclosing $[\bar{\alpha}]$ in a **let** expression. In particular if $e' : \tau'$ does not occur inside the first part of an enclosing **let**, then :

$$A, \Pi, e'' \vdash e' : \tau'$$

Proof : by induction on derivations, using proposition 2.3.2. \square

Corollary 2.5.5

If $x[\bar{t}] : \tau$ is a polymorphic occurrence of a program variable in a program, not occurring inside the first part of an enclosing **let**, then \bar{t} is ground.

Proof : By the type inference rule PVAR and the fact that the subtype ordering in a type scheme contains all bound variables (definition 1.11.1). \square

Definition 2.5.6

We define a reduction relation \Rightarrow on programs by the rewrite rules in Table 1, which appears as an appendix. All rules have the form $e : \tau \Rightarrow e' : \tau$. We say that a typed expression is a *redex* if it is an instance of the left hand side of a rule and we say that a program is in *normal form* if it contains no redexes.

Remark 2.5.7

We make the following comments on the rules in Table 1 :

- 1,2: Note that there are separate rewrite rules for $plus[nat]$ and $plus[int]$, but the difference in types ensures that these rules do not interfere. Furthermore, $plus$ commutes with the coercion from nat to int , specified by rule 20, ie :

$$plus(n : nat : int) (m : nat : int) : int \Rightarrow n + m : int$$

$$plus(n : nat) (m : nat) : nat : int \Rightarrow n + m : int$$

In the general case, with types such as real and complex, the rules for $plus$ would differ significantly, and commutativity would be more difficult to prove. In fact, in many actual machines, commutativity of $plus$ over int and $real$ does not hold. The former Burroughs B6000/7000 series (Organick, 1973) is an exception in this respect.

- 16: The rule for fix allows recursion, and results in the possibility of infinite reduction sequences.

- 17,18: As for $plus$, there are two rewrite rules for function application, the first for constant functions, and the second the usual beta-substitution rule using *replacements* (definition 2.2.1). Note that by proposition 2.3.2 the replacement is defined, and replaces all free occurrences of x in e_1 by e_2 . Application commutes with the coercion from constant functions to general functions, specified by rule 31, ie :

$$((\lambda x : \tau. e_1 : \tau) : \tau \supset \tau' : \tau \rightarrow \tau') (e_2 : \tau) : \tau' \Rightarrow e_1 : \tau'$$

$$((\lambda x : \tau. e_1 : \tau) : \tau \supset \tau') (e_2 : \tau) : \tau' \Rightarrow e_1 : \tau'$$

since if the type inference rule ABS2 is applicable in the first example, giving the type $\tau \supset \tau'$, x does not occur free in e_1 .

If we included other functional type constructors, there would be other, more complex, rules for application. For example, finite database functions would use table lookup rather than beta-substitution. Strict functions would use beta-substitution, but in a parallel implementation (Peyton Jones, 1987), applications of the form :

$$(e_1 : \delta\tau\tau') (e_2 : \tau) : \tau'$$

where δ is the strict-function type constructor would be flagged to reduce e_1 and e_2 in parallel. For both finite database functions and strict functions, application would commute with coercion as for constant functions above.

- 19: The reduction rule for **let** is defined using *replications*, (definition 2.2.5). Note that by proposition 2.3.2 the replication is defined, and replaces each free occurrence $x[\bar{t}_i]$ in e_2 by $[\bar{t}/\bar{\alpha}]e_1$.
- 20-33: These rules define coercion rewrites on programs of the form $e : \tau : \tau'$ where $\tau \ll \tau'$ and $e : \tau$ is not a redex. Rule 20 is justified by the restriction from definition 1.3.1 that $K(\rho)$ is contained in $K(\pi)$ whenever $\rho \ll \pi$. We can thus coerce $n : nat : int$ to $n : int$. In the general case we would have more complex coercions such as *int* to *real* and *real* to *complex*.
- Rules 22 and 24 implement our example of a coercion between type constructors of rank 1, ie *optional* to *list*, while rule 31 implements our example of a coercion between type constructors of rank *, ie \supset to \rightarrow . The other rules allow coercions between structured types. Rule 32 is justified by the antimonotonicity of type constructors of rank *, and uses a replacement to apply a coercion to the argument of a function. By proposition 2.3.2 the replacement is defined and replaces all free occurrences of x in e by $x : \tau_1$. The family of rules 33 creates abstractions to provide the same effect.
- 34-48: These rules define the equality function *eq*. Note that this is defined only on basic ground types, but by corollary 2.5.5 and the type scheme for *eq*, all occurrences of *eq* in a program will be of the form $eq[\tau]$, where τ is a basic ground type.

We will now show that our reduction rules map programs to programs, and that type information is preserved. This gives us a semantic soundness result. First we prove a number of lemmas :

Lemma 2.5.8

If $A_x \cup \{x : \tau\}, \leq, e \vdash e' : \tau$
 and $A, \leq, e_1 \vdash e'_1 : \tau$
 then $A, \leq, [e_1/x]e \vdash [e'_1 : \tau/x : \tau] (e' : \tau)$

Proof : By induction on the first derivation, replacing appropriate uses of the VAR rule by the second derivation. \square

Lemma 2.5.9

If $A, \leq', \text{let } x = e_1 \text{ in } e_2 \vdash (\text{let } x = [\bar{\alpha}] e'_1 : \tau \text{ in } e'_2 : \tau) : \tau$
 then $A, \leq', [e_1/x] e_2 \vdash [[\bar{\alpha}] e'_1 : \tau // x] (e'_2 : \tau)$

Proof: By considering how the first derivation is obtained using the LET rule from derivations for e_1 and e_2 . We use induction on the derivation of e_2 , replacing appropriate uses of the PVAR rule by the derived inference rule SUBEN applied to the derivation for e_1 . \square

Lemma 2.5.10

If $A, \leq, (\lambda x. e) \vdash (\lambda x : \tau_1. e' : \tau_2) : \tau_1 \rightarrow \tau_2 : \tau'_1 \rightarrow \tau'_2$
 then $A, \leq, (\lambda x. e) \vdash (\lambda x : \tau_1. [x : \tau'_1 : \tau_1 / x : \tau_1] e : \tau_2 : \tau'_2) : \tau'_1 \rightarrow \tau'_2$

Proof: The first derivation uses ABS followed by LESS. This can be replaced by the derived inference rule REPLACE, followed by LESS and ABS. \square

We can now prove our semantic soundness theorem :

Theorem 2.5.11

For each reduction rule $e : \tau \Rightarrow e' : \tau$ in Table 1,

if $A, \leq, \text{untype}(e : \tau) \vdash e : \tau$ (1)

then $A, \leq, \text{untype}(e' : \tau) \vdash e' : \tau$ (2)

and hence programs rewrite to programs.

Proof: The first part is by cases, using lemmas 2.5.8, 2.5.9, and 2.5.10 for the most difficult cases. For the second part, we note that a redex inside a program has a derivation of the form (1), and so after applying a single rewrite rule, we can replace the derivation (1) by a derivation of the form (2). We can then use induction on the number of applications of rewrite rules. \square

2.6 A Church-Rosser Result

To show that our reduction relation indeed defines a semantics for programs, we must show that the rules in Table 1 are Church-Rosser. A program will then have a unique value, which is the possibly infinite *Böhm tree* obtained by reducing all redexes, and replacing by the symbol \perp terms which do not have a *weak head normal form* (Peyton Jones 1987, Chapter 11). We will first simplify rule 19, which involves replication :

Proposition 2.6.1

If rule 19 in Table 1 is replaced by the pair of rules :

$$19A \quad (\text{let } x = [\bar{\alpha}] e_1 : \tau \text{ in } e_2 : \tau) : \tau \Rightarrow [[\bar{\alpha}] e_1 : \tau // x] (e_2 : \tau)$$

$$19B \quad [\bar{\alpha}] (e : \tau) [\bar{t}] : \tau \Rightarrow ([\bar{t} / \bar{\alpha}] e) : \tau$$

where rule 19A uses a substitution of terms for program variables, and typed expressions are extended to include the syntax on the left-hand side of rule 19B, then if $e : \tau \Rightarrow e' : \tau$ using rules 1-48, then $e : \tau \Rightarrow e' : \tau$ using the modified rules. Furthermore, the original set of rules is Church-Rosser if the modified set of rules is Church-Rosser.

Proof : The first part is by definition 2.2.5 and proposition 2.3.2. For the second part, note that 19B-redexes are only created by rule 19A, and only rewritten by rule 19B. Then (using M, N, P, Q, R for typed expressions), if $M \Rightarrow N$ and $M \Rightarrow P$ by rules 1-48, then $M \Rightarrow N \Rightarrow Q \Rightarrow R$ and $M \Rightarrow P \Rightarrow Q \Rightarrow R$ where R contains no 19B-redexes. Also there are reductions $N \Rightarrow R$ and $P \Rightarrow R$ in which each use of rule 19A is followed by sufficient uses of rule 19B to remove all 19B-redexes, and these correspond to reductions using rule 19. \square

We prove the Church-Rosser property for the modified (and hence for the original) rules by defining a Combinatory Reduction System in the sense of Klop (1980), which is isomorphic to the reduction system defined by the modified rules.

Definition 2.6.2

We define the function Tr on typed expressions as follows, where $Coerce$, Let and $Poly$ are new constants, and $[x]$ represents variable binding :

$$\begin{aligned}
Tr(x : \tau) &= x \\
Tr(x [\bar{t}] : \tau) &= x \ \bar{t} \\
Tr(c : \pi) &= c \ \pi \\
Tr((\lambda x : \tau. e : \tau) a \ \tau) &= a \ \tau ([x] Tr(e : \tau)) \\
Tr((e : a \ \tau) (e' : \tau) : \tau) &= (Tr(e : a \ \tau)) (Tr(e' : \tau)) \\
Tr(e : \tau : \tau) &= Coerce \ \tau \ \tau (Tr(e : \tau)) \\
Tr((let x = [\bar{\alpha}] e : \tau in e' : \tau) : \tau) &= Let ([\bar{\alpha}] Tr(e : \tau)) ([x] Tr(e' : \tau)) \\
Tr([\bar{\alpha}] (e : \tau) [\bar{t}] : \tau) &= Poly ([\bar{\alpha}] Tr(e : \tau)) \ \bar{t} \\
Tr([\bar{\alpha}] (e : \tau)) &= Poly ([\bar{\alpha}] Tr(e : \tau))
\end{aligned}$$

Applying Tr to both sides of the modified rewrite rules gives us a new set of rewrite rules. We express the substitutions in rules 18, 19A, 19B and 32 by using curly brackets, as in Klop (1980). Some of the translated rules are :

$$\begin{aligned}
14 \quad &fst(\tau \ \tau') (pair(\tau \ \tau') e_1 e_2) \Rightarrow e_1 \\
18 \quad &\rightarrow \tau ([x] e_1 \{x\}) e_2 \Rightarrow e_1 \{e_2\} \\
19A \quad &Let([\bar{\alpha}] e_1 \{ \bar{\alpha} \}) ([x] e_2 \{x\}) \Rightarrow e_2 \{Poly([\bar{\alpha}] e_1 \{ \bar{\alpha} \})\} \\
19B \quad &Poly([\bar{\alpha}] e \{ \bar{\alpha} \}) \ \bar{t} \Rightarrow e \{ \bar{t} \} \\
32 \quad &Coerce(\tau_1 \rightarrow \tau_2) (\tau'_1 \rightarrow \tau'_2) (\rightarrow \tau_1 [x] e \{x\}) \\
&\Rightarrow \rightarrow \tau'_1 [x] Coerce \ \tau_2 \ \tau'_2 e \{Coerce \ \tau'_1 \ \tau_1 x\}
\end{aligned}$$

Note that the translated rules 1, 2, 3, 20, 34, 35 specify *families* of rewrite rules.

Proposition 2.6.3

For each translated rewrite rule $E \Rightarrow E'$:

- E begins with a constant (where program variables in A_0 , such as fst , are treated as constants).
- all metavariables e, e_i, τ, τ_i in E' occur already in E .
- if a metavariable e has arguments in curly brackets then it has the same number of arguments in E and E' , and the arguments in E are distinct variables (type variables or program variables).
- all variables (type variables or program variables) in E and E' are bound.
- no metavariable e in E occurs as a subterm eE'' , $e\{x\}E''$ or $e\{ \bar{\alpha} \}E''$.

- the only metavariables that occur twice in E and E' are metavariables corresponding to types, and these can be renamed to be distinct without altering the class of terms that can be rewritten by the rule.
- no left-hand side of a rule can be unified with a subterm of E , apart from the trivial cases of E unifying with itself, or any left-hand side unifying with a metavariable.

Hence definition 2.6.2 specifies a *regular* Combinatory Reduction System in the sense of Klop (1980, definition II.1.11, II.1.14 and II.1.16).

Proof: By inspection. The sixth case uses the fact that we are dealing only with terms that result from applying Tr to subexpressions of programs. \square

Corollary 2.6.4

The reduction system in definition 2.6.2 is Church-Rosser.

Proof: By theorem II.3.11 in Klop (1980). \square

Proposition 2.6.5

There is a function Tr^{-1} such that for each program $e : \tau$, $Tr^{-1}(Tr(e : \tau)) = e : \tau$, and $e : \tau \Rightarrow e' : \tau$ using the modified rules or proposition 2.6.1 if and only if $Tr(e : \tau) \Rightarrow Tr(e' : \tau)$ using the rules of definition 2.6.2.

Proof: The function Tr^{-1} can be easily constructed using definition 2.6.2 and the type inference rules. \square

Corollary 2.6.6

The modified reduction system of proposition 2.6.1 is Church-Rosser, and hence so is the reduction system of Table 1.

2.7 Weak Head Normal Form and Böhm Trees

We now return to the concept of Böhm tree introduced previously. The concept is discussed briefly in Klop (1980, IntermezzoIII.2), and in detail in Barendregt (1984, Chapter 10) for the case of the pure Lambda Calculus.

Definition 2.7.1

We say that a typed expression $M = e : \tau$ has *weak head normal forms* (whnfs) if there is a typed expression $N = P\{Q_1, \dots, Q_n\}$ such that $M \Rightarrow N$, and $N \Rightarrow N'$ implies that $N' = P\{Q'_1, \dots, Q'_n\}$, and for all R_1, \dots, R_n in normal form, $P\{R_1, \dots, R_n\}$ is also in normal form. We also say that N and N' are *weak head normal forms* (whnfs) of M , and that N and N' are *in whnf*.

Example 2.7.2

- All normal forms are in whnf.
- $\Omega_\tau = (\text{fix } ((\lambda x: \tau.x : \tau) : \tau \rightarrow \tau)) : \tau$ has no whnf, since $\Omega_\tau \Rightarrow (\lambda x: \tau.x : \tau) \Omega_\tau \Rightarrow \Omega_\tau$
- $(\text{pair } \Omega_\tau \ \Omega_\tau) : \tau \times \tau$ is in whnf, since $\text{pair } M \ N$ is never a redex.
- $(\text{add } \Omega_{\text{nat}} \ \Omega_{\text{nat}}) : \text{nat}$ has no whnf, since, for example, $(\text{add } (3 : \text{nat}) \ (3 : \text{nat})) : \text{nat}$ is not in normal form
- $(\Omega_{\text{nat}} : \text{int})$ has no whnf, since $(3 : \text{nat} : \text{int})$ is not in normal form

Definition 2.7.3

If $M = e : \tau$ is a typed expression, then we define the Böhm tree of M by :

- $BT(M) = \perp : \tau$ if M does not have whnfs
- $BT(M) = P\{BT(Q_1), \dots, BT(Q_n)\}$
if $P\{Q_1, \dots, Q_n\}$ is a whnf of M

It is clear, by the Church-Rosser property proved in the previous section, that the process of producing a Böhm tree gives a unique result. The Böhm tree may be finite or infinite. The following proposition characterises Böhm trees of programs :

Proposition 2.7.4

If $M = e : \tau$ is a program, then $BT(M)$ has one of the following forms :

$\perp : \tau$

$c : \pi$

$(\lambda x : \tau'. Q) : \tau' \rightarrow \tau''$

$(\lambda x : \tau'. Q) : \tau' \supset \tau''$

$y \ Q_1 \dots Q_n : \tau$

where $n = 0, y \in \{\text{absent}, \text{nil}, \text{fst}, \text{snd}, \text{fix}, \text{neg}\}$

or $n \leq 1, y \in \{\text{inl}, \text{inr}, \text{present}, \text{plus}, \text{and}, \text{eq}\}$

or $n \leq 2, y \in \{\text{pair}, \text{cons}, \text{case}, \text{lchoose}, \text{ochoose}, \text{cond}\}$

Proof : By induction on the derivation of programs, each whnf must be of the indicated form. \square

Some examples of Böhm trees follow :

Example 2.7.5

- $BT(\Omega_\tau) = \perp : \tau$
- $BT(M)$, if M is in normal form
- $BT(\text{fix}((\lambda x : \text{list nat. cons } (1 : \text{nat}) (x : \text{list nat}) : \text{list nat}) : \text{list nat} \rightarrow \text{list nat}) : \text{list nat})$
 $= \text{cons } (1 : \text{nat}) (\text{cons } (1 : \text{nat}) (... : \text{list nat}) : \text{list nat})$

The Böhm tree $BT(M)$ can be viewed as the meaning or *value* of the program M . However, more natural for our purposes is to take the value of a program to be an equivalence class of Böhm trees modulo *extensional equality*. This ensures that functions with the same effect will be viewed as having the same value. Before we can define such an equality, however, we must define application on Böhm trees.

Definition 2.7.6

We define the typed expression $E(M)$ corresponding to a finite Böhm tree M by replacing each subtree of the form $\perp : \tau$ by Ω_τ . For infinite Böhm trees M we define the n^{th} approximation M^n by replacing all subtrees at depth n by \perp . We write $M \subseteq N$ if M is the tree resulting by replacing zero or more subtrees $M_i : \tau_i$ of N by $\perp : \tau_i$.

Proposition 2.7.7

For all Böhm trees M , $M = \cup_{n \geq 0} M^n$, where the union is the obvious one with $M \cup N = N$ if $M \subseteq N$. Furthermore, for finite Böhm trees, $BT(E(M)) = M$.

Proof: By definitions 2.7.3 and 2.7.6. \square

Definition 2.7.8

We define application of Böhm trees M and N by $MN = \cup_{n \geq 0} BT(E(M^n) E(N^n))$, ie as the limit of application of approximations.

Proposition 2.7.9

The limit in definition 2.7.8 exists, and furthermore, application is continuous.

Proof: As in section 10.2, 14.3 and 18.3 of Barendregt (1984). This material generalizes readily. For example, in Dekker (1988) we apply it to combinatory logic. In particular, we can distinguish the Böhm trees $\perp : \tau \rightarrow \tau$ and $(\lambda x : \tau. \perp : \tau) : \tau \rightarrow \tau$ without problems, and indeed in this respect we resemble combinatory logic, where \perp and $K\perp$ are distinct.

An alternative proof would be by translating programs (or more easily, the translated programs of proposition 2.6.2) to the pure lambda calculus, as in Kazmierczak (1989). To perform this translation, we note that let-redexes can be reduced as part of the translation process, and that there are functions which perform the coercions defined by reduction rules 20-33, and the equalities defined by reduction rules 34-48. Continuity of Böhm tree application would then follow by fixpoint induction. \square

Definition 2.7.10

We define *extensional equality* on Böhm trees as follows. For each type τ we define an equivalence relation $=_{ext}$ by the transitive, symmetric, reflexive closure of the following equalities :

- if $M_i =_{ext} N_i$ then $P\{M_1 \dots M_n\} =_{ext} P\{N_1 \dots N_n\}$
- if $MQ =_{ext} NQ$ for all closed Böhm trees Q then $M =_{ext} N$

Remark 2.7.11

For each type τ , the set of values $V(\tau)$ referred to in section 1.1 is the set of Böhm trees of type τ , modulo $=_{ext}$. For example :

$$V(nat) = \{[\perp : nat], [0 : nat], [1 : nat], \dots\}$$

consists only of one-element equivalence classes, while for functional types we have more complex equivalence classes :

$$\begin{aligned} V(nat \rightarrow nat) = \{ & [\perp : nat \rightarrow nat, (\lambda x : nat. \perp : nat) : nat \rightarrow nat, \dots], \\ & [(\lambda x : nat. 0 : nat) : nat \rightarrow nat, \dots], \\ & \dots\} \end{aligned}$$

To prove properties of Böhm trees, we will use noetherian induction on finite Böhm trees, and use fixpoint induction to extend the result to infinite Böhm trees. The latter technique is valid because application of Böhm trees is continuous. The former technique uses the following inequality relation :

Definition 2.7.12

We say that a Böhm tree $e : \tau$ is *simpler than* a Böhm tree $e' : \tau'$ if :

- $|\tau| < |\tau'|$ or
- $|\tau| \leq |\tau'|$ and $e : \tau$ is a subtree of $e' : \tau'$
where $|\tau|$ is the length of τ (definition 1.4.1).

Proposition 2.7.13

For finite Böhm trees, the relation *simpler than* is a partial order with no infinite descending chains.

Proof: Antisymmetry, transitivity and reflexivity are trivial, and all descending chains clearly terminate with Böhm trees of constant or variable types. \square

2.8 Properties of Böhm Trees

We now use the proof technique outlined in the previous section to prove some properties of Böhm trees of programs. Our first proposition and corollary show that coercions produce a unique result :

Proposition 2.8.1

For programs $e : \tau$ that have finite Böhm trees,

- $BT(e : \tau) =_{ext} BT(e : \tau : \tau)$
- $BT(e : \tau : \tau') =_{ext} BT(e : \tau : \tau' : \tau')$

Proof: By noetherian induction (on the relation *simpler than*) on $BT(e : \tau)$, using proposition 2.7.4. The most difficult case is

$M = (\lambda x : \tau_1. e : \tau_2) : \tau_1 \rightarrow \tau_2$. We have $N = (\lambda x : \tau_1 [x : \tau_1 : \tau_1 / x : \tau_1] e : \tau_2 : \tau_2) : \tau_1 \rightarrow \tau_2$. Let $P = e' : \tau_1$, and by induction hypothesis (since $|\tau_1| < |\tau_1 \rightarrow \tau_2|$), $BT(e' : \tau_1 : \tau_1) = BT(e' : \tau_1)$, similarly $MP =_{ext} NP$, so $M =_{ext} N$. The other equality is proved similarly. \square

Corollary 2.8.2

The equalities in the above proposition also hold for infinite Böhm trees.

Proof: By fixpoint induction. \square

The following proposition and corollary show that coercion is injective on basic types, and hence that the equality operator commutes with coercions :

Proposition 2.8.3

For programs $e : \tau$, $e' : \tau$ that have finite Böhm trees, with τ a basic ground type,

- $BT(e : \tau) = BT(e' : \tau)$ if
 $BT(e : \tau : \tau') = BT(e' : \tau : \tau')$

Proof: By noetherian induction (on the relation *simpler than*) on $BT(e : \tau)$, using proposition 2.7.4. \square

Corollary 2.8.4

The above proposition also holds for infinite Böhm trees.

Proof: By fixpoint induction. \square

The next proposition shows that application commutes with coercions on functions.

Proposition 2.8.5

For all programs $e : a\tau\tau'$, with $a \ll b$, $a, b \in \{\supset, \rightarrow\}$, and $\tau' \ll \tau''$

- $BT((e : a\tau\tau' : b\tau\tau') (e' : \tau) : \tau')$
 $=_{ext} BT((e : a\tau\tau') (e' : \tau) : \tau' : \tau')$

Proof: By cases on $BT(e : a\tau\tau')$, using corollary 2.8.2. \square

We are now in a position to prove the last result required in section 1.2, namely that coercions commute with polymorphic operations.

Theorem 2.8.6

If $(y : \forall \bar{\alpha} : \leq. \tau) \in A_0$, and $\tau = \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau_{n+1}$ and $S\tau = \tau'_1 \rightarrow \dots \rightarrow \tau'_n \rightarrow \tau'$, $R\tau = \tau''_1 \rightarrow \dots \rightarrow \tau''_n \rightarrow \tau'$ and $\tau'_i \ll \tau''_i$, $\tau' \ll \tau''$, $S(\leq) = R(\leq) = \Pi$, then :

- $BT(y(e_1 : \tau'_1) \dots (e_n : \tau'_n) : \tau' : \tau'')$
 $=_{ext} BT(y(e_1 : \tau'_1 : \tau''_1) \dots (e_n : \tau'_n : \tau''_n) : \tau'')$

Proof: By cases for each $y \in A_0$ and $n \leq 0$, using corollary 2.8.4 for the case of *eq*, and proposition 2.8.5 for *ochoose*, *lchoose* and *case*. For *plus* we use simple case analysis on the arguments, which must have finite Böhm trees, and for *absent*, *present*, *nil*, *cons*, *inl*, *inr* and *pair* we use rewrite rules 21-29. The cases *cond*, *fst* and *snd* are trivial, while *neg*, *and* and *fix* must have $S\tau = R\tau$, and follow by corollary 2.8.2. \square

Remark 2.8.7

Theorem 2.8.6, together with corollary 2.8.2, justifies the fact that our type inference rules can introduce coercions in different places when typing an expression. If, for example $A_0, \Pi, e \vdash e' : \tau$ and $A_0, \Pi, e \vdash e'' : \tau$, then $e' : \tau$ and $e'' : \tau$ will differ only in type information and coercions introduced, and we conjecture that, in this case, $BT(e' : \tau) =_{ext} BT(e'' : \tau)$.

In the next section we describe an algorithm TYPE, such that $TYPE(A_0, e) = e' : \tau, \leq, \{ \}$ and $A_0, \leq, e \vdash e' : \tau$ for all e from which a program can be derived using the type inference rules. The typed expression $e' : \tau$ will be the ‘best’ typed expression that can be derived from e , but since we will have in general $\leq \neq \Pi$, we have no semantics for $e : \tau$. However, we can ‘compile’ this typed expression by finding a substitution S which satisfies \leq , ie $S(\leq) = \Pi$. The typed expression $S(e' : \tau)$ is then a program, and has a semantics. For example, consider :

$$(\lambda x : \alpha. plus(x : \alpha)(1 : nat : \alpha) : \alpha)(3 : nat : \alpha) : \alpha : int$$

with $nat \leq \alpha \leq int$. By choosing either $\alpha = nat$ or $\alpha = int$, we choose a rewrite rule for *plus*, and we choose particular rewrites for the coercions. Because of the commutativity of *plus* with the coercion from *nat* to *int*, we obtain the same result $(4 : int)$ in either case. Thus, although ‘compilation’ is not deterministic, depending on the choice of a substitution satisfying the given subtype ordering, the final result is independent of that choice.

3 ALGORITHMS

We now present two type inference algorithms for the type inference system outlined in section 2.3. The algorithms operate by passing inwards a set of type schemes for polymorphic program variables as in Damas and Milner (1982), but returning outwards a set of types for nonpolymorphic program variables as in Mitchell (1984). The combination of the two approaches, together with the restriction from section 1.11 that type schemes be closed, allows us to combine subtypes and polymorphism.

Section 3.1 describes two subroutines, AUNIFY and ARROW, which are used in the type inference algorithms. The first of these takes a subtype ordering \leq , and two assumption sets A_1 and A_2 , and ‘unifies’ them by returning a minimal pair \leq', S and an assumption set A such that $A \leq' SA_1$ and $A \leq' SA_2$. Two replacements are also returned which coerce program variables from the types in A to the types in SA_1 or SA_2 . The subroutine ARROW takes two types and a subtype ordering, and returns a minimal pair \leq', S such that applying the substitution to the first type gives a functional type, and applying the substitution to the second type gives a type coercible to the argument type of the function. The functional type is also returned. In section 3.2 we give a naive type inference algorithm using these subroutines, prove that it is sound and complete, and give examples of its behaviour.

In section 3.3 we define a *standardisation* of a subtype ordering \leq to be a pair \leq', S which is generated from \leq by APPLYSUBST and ENRICH, such that all atomic substitutions R which *satisfy* \leq (by consistently mapping type variables to type constructors) have the form $R'S$ where R' satisfies \leq' . We provide standardisation *rules*, and give an algorithm STANDARDISE which computes the unique maximal application of standardisation rules to a subtype ordering. Section 3.4 uses standardisations to obtain an algorithm which computes a substitution satisfying any subtype ordering, and shows how this can be viewed as a kind of abstract compilation.

Section 3.5 defines *simplifications* which extend standardisations. The requirement that any R satisfying \leq has the form $R'S$ is relaxed, and instead we require that $R'S\tau$ coerces to $R\tau$ for some fixed type τ . We give four simplification rules (EQU, EQDOWN, MOVEUP and MOVEDOWN), and discuss the benefits and disadvantages of the last two of these. We then give an algorithm SIMPLIFY which computes the unique (up to renaming) maximal application of EQU and EQDOWN to a subtype ordering.

An improved type inference algorithm is given in section 3.6, which uses STANDARDISE and SIMPLIFY to reduce the size of intermediate subtype orderings. We show this improved algorithm is sound and complete. A number of examples demonstrate how the size of intermediate subtype orderings is kept manageable, and how the final subtype ordering typically contains at most one type variable. Section 3.7 discusses efficiency of the algorithm in more detail.

3.1 Type Inference Subroutines

Our first subroutine is for ‘unifying’ assumption sets. During type inference, we may infer a typed expression $e_1 : \tau_1$ together with the assumption $\{x : \tau_1\}$, and infer $e_2 : \tau_2$ with the assumption $\{x : \tau_2\}$. To handle an expression which has $e_1 : \tau_1$ and $e_2 : \tau_2$ as subexpressions, we require a single assumption $\{x : \tau\}$ with $\tau \leq \tau_1$, $\tau \leq \tau_2$ for some \leq , and two replacements $[x : \tau : \tau_1 / x : \tau_1]$, $[x : \tau : \tau_2 / x : \tau_2]$ which we can apply to $e_1 : \tau_1$ and $e_2 : \tau_2$ to ensure that all occurrences of x have the form $x : \tau$. The subroutine AUNIFY takes two assumption sets A_1 and A_2 , and splits each of these into three parts. We then obtain one pair of identical assumption sets, one pair of assumption sets with the same domain, and one disjoint pair. For the second pair we then construct ‘lower bounds’ for the two types assigned to each program variable x_i .

Definition 3.1.1

Let \leq be a subtype ordering, and A_1, A_2 be two assumption sets containing only types. Then we define :

$$\text{AUNIFY}(A_1, A_2, \leq) = X, S, A, Q_1, Q_2$$

$$\begin{aligned} \text{where } A_1 &= A' \cup A'_1 \cup A''_1 \\ A_2 &= A' \cup A'_2 \cup A''_2 \\ A'_1 &= \{x_1 : \tau_1, \dots, x_n : \tau_n\} \\ A'_2 &= \{x_1 : \tau'_1, \dots, x_n : \tau'_n\} \\ A', A'_1, A''_1, A'_2 &\text{ are pairwise disjoint} \end{aligned}$$

$$\begin{aligned} \text{and } \beta_1, \dots, \beta_n &\text{ are previously unused type variables} \\ L &= \beta_1 \leq \tau_1; \beta_1 \leq \tau'_1; \dots; \beta_n \leq \tau_n; \beta_n \leq \tau'_n \\ X, S &= \text{ENRICH}(\leq, L) \\ A &= SA' \cup \{x_1 : S\beta_1, \dots, x_n : S\beta_n\} \cup SA''_1 \cup SA''_2 \\ Q_1 &= [x_1 : S\beta_1 : S\tau_1 / x_1 : S\tau_1, \dots, x_n : S\beta_n : S\tau_n / x_n : S\tau_n] \\ Q_2 &= [x_1 : S\beta_1 : S\tau'_1 / x_1 : S\tau'_1, \dots, x_n : S\beta_n : S\tau'_n / x_n : S\tau'_n] \end{aligned}$$

Note that $A = Q_1 SA_1 \cup SA''_2 = Q_2 SA_2 \cup SA''_1$ and by definition 1.11.4,

$X = \leq', A \leq' SA_1$ and $A \leq' SA_2$.

Proposition 3.1.2

If $\text{AUNIFY}(A_1, A_2, \leq) = \leq', S, A, Q_1, Q_2$ then S is an expansion, and \leq', S is minimal such that $S\beta_i \leq' S\tau_i$ and $S\beta_i \leq' S\tau'_i$ for each i , and for $\tau, \tau', \tau \leq \tau'$ implies $S\tau \leq' S\tau'$. Furthermore, if $\text{SUBEN}(R, \leq, L') = \leq''$ and $A' \leq'' RA_1$, $A' \leq'' RA_2$, then $R = S'S$, $\text{SUBEN}(S', \leq', L'') = \leq''$ for some L' , and $A' \leq'' S'A$.

Proof: From definition 1.10.13 $\text{ENRICH}(\leq, L)$ is constructed by MATCH2 , APPLYSUBST , ORDERSET and UNION , and the UNION operation adds to \leq only pairs of inequalities of the form $\alpha \leq a$, $\alpha \leq b$ or $a \leq \alpha$, $b \leq \alpha$ where α is an otherwise unused type variable. Thus no cycles result, and S is an expansion. The second part follows from theorem 1.10.14. The third part follows from minimality. \square

Proposition 3.1.3

If $B \cup A_1, \leq, e_1 \vdash e'_1 : \tau_1$
 $B \cup A_2, \leq, e_2 \vdash e'_2 : \tau_2$
 B contains only type schemes, A_1, A_2 only types
 and $\text{AUNIFY}(A_1, A_2, \leq) = \leq', S, A, Q_1, Q_2$
 then $B \cup A, \leq', e_1 \vdash Q_1S(e'_1 : \tau_1)$
 and $B \cup A, \leq', e_2 \vdash Q_2S(e'_2 : \tau_2)$

Proof: Using the derived inference rules ENRICH , REPLACE , and ASSUMP . \square

Our second subroutine deals with constructing applications, where we require that an appropriate coercion be introduced, and that the expression being applied is an appropriate function.

Definition 3.1.4

Let \leq be a subtype ordering, and let τ_1, τ_2 be types. Then we define :

$\text{ARROW}(\tau_1, \tau_2, \leq) = \leq'', RS, R(a\tau\tau')$
 where $\text{MATCH2}(\leq, \tau_1, \tau_2 \rightarrow \alpha) = S'$
 α is an otherwise unused type variable
 $S = [S' \ \bar{\beta} / \bar{\beta}]$ for $\bar{\beta} = \text{Dom}(S') - \{\alpha\}$
 $\text{APPLYSUBST}(S, \leq) = \leq', S$
 $S\tau_1 = a\tau\tau'$
 $\text{ENRICH}(\leq', S\tau_2 \leq \tau, a \leq \rightarrow) = \leq'', R$

Note that $\text{MATCH2}(\leq, \tau_1, \tau_2 \rightarrow S'\alpha) = S$, and hence by corollary 1.10.9, $\text{APPLYSUBST}(S, \leq) = \leq', S$. Note also that, since $S\tau_2$ matches τ , R is atomic. The properties of **ARROW** are given by the following propositions :

Proposition 3.1.5

If $\text{ARROW}(\tau_1, \tau_2, \leq) = \leq', RS, R(a\tau\tau')$ then \leq', RS is minimal such that $RS\tau_1 = b\tau'_1\tau'_2$ for some $b, \tau'_1, \tau'_2, RS\tau_2 \leq' \tau'_1, b \leq' \rightarrow$ and for all $\tau'', \tau''', \tau' \leq \tau''$ implies $RS\tau' \leq' RS\tau''$

Proof: By theorems 1.7.13, 1.10.5 and 1.10.14. \square

Proposition 3.1.6

If $A, \leq, e_1 \vdash e'_1 : \tau_1$
 $A, \leq, e_2 \vdash e'_2 : \tau_2$
 and $\text{ARROW}(\tau_1, \tau_2, \leq) = \leq', RS, R(a\tau\tau')$
 then $RSA, \leq', e_1 e_2 \vdash (RS(e'_1 : \tau_1))(RS(e'_2 : \tau_2) : R\tau) : R\tau$

Proof: Using the derived inference rules **SUBST** and **ENRICH**, and the inference rules **LESS** and **AP**. \square

3.2 A Type Inference Algorithm

We now describe our first type inference algorithm and prove that it is sound and complete.

Definition 3.2.1

Let B be an assumption set containing only type schemes. Then we define $\text{TYPE}(B, e) = e' : \tau, \leq, A$ for each expression e as follows :

$\text{TYPE}(B, c) = c : \pi, \Pi, \{ \}$	where π is the least type such that $c \in k(\pi)$, which exists by proposition 1.3.2
$\text{TYPE}(B, x) = x : \alpha, \Pi, \{x : \alpha\}$	where $x \notin B$, and α is a previously unused type variable
$\text{TYPE}(B, y) = y[S \bar{\alpha}] : S\tau, S(\leq), \{ \}$	where $y : \forall \bar{\alpha} : \leq. \tau \in B$ and S is a renaming of $\bar{\alpha}$ (definition 1.5.1)

$$\begin{aligned} \text{TYPE}(B, \lambda x. e) &= (\lambda x : \tau. e' : \tau) : \tau \rightarrow \tau, \leq, A_x \\ &\quad \text{if } \text{TYPE}(B_x, e) = e' : \tau, \leq, A \\ &\quad \text{and } x : \tau \in A \end{aligned}$$

$$\begin{aligned} \text{TYPE}(B, \lambda x. e) &= (\lambda x : \alpha. e' : \tau) : \alpha \supset \tau, \leq, A \\ &\quad \text{if } \text{TYPE}(B_x, e) = e' : \tau, \leq, A \\ &\quad \text{and } x \notin A \\ &\quad \text{and } \alpha \text{ is a previously unused type variable} \end{aligned}$$

$$\begin{aligned} \text{TYPE}(B, e_1 e_2) &= (SQ_1 R(e'_1 : \tau_1)) (SQ_2 R(e'_2 : \tau_2) : \tau) : \tau', \leq_5, SA \\ &\quad \text{if } \text{TYPE}(B, e_i) = e'_i : \tau_i, \leq_i, A_i \\ &\quad \text{and } \text{UNION}(\leq_1, \leq_2) = \leq_3, ID \\ &\quad \text{and } \text{AUNIFY}(A_1, A_2, \leq_3) = \leq_4, R, A, Q_1, Q_2 \\ &\quad \text{and } \text{ARROW}(R\tau_1, R\tau_2, \leq_4) = \leq_5, S, a\tau\tau' \end{aligned}$$

$$\begin{aligned} \text{TYPE}(B, \text{let } x = e_1 \text{ in } e_2) &= (\text{let } x = [\bar{\alpha}] e'_1 : \tau_1 \text{ in } e'_2 : \tau_2) : \tau_2, \leq_2, \{\} \\ &\quad \text{if } \text{TYPE}(B, e_1) = e'_1 : \tau_1, \leq_1, \{\} \\ &\quad \text{and } \text{gen}(\{\}, \leq_1, \tau_1) = \sigma = \forall \bar{\alpha} : \leq_1. \tau_1 \\ &\quad \text{by definition 1.11.7} \\ &\quad \text{and } \text{TYPE}(B_x \cup \{x : \sigma\}, e_2) = e'_2 : \tau_2, \leq_2, \{\} \end{aligned}$$

If any of the restrictions do not hold, then $\text{TYPE}(B, e)$ is undefined. Note that if $\text{TYPE}(B, e) = e' : \tau, \leq, A$, then A and B are disjoint, and A will contain only types. Also, since only previously unused type variables are introduced into subtype orderings by the cases $x \in B$ and $y \notin B$, the UNION in the case $e_1 e_2$ will always return the indicated result. The following theorem shows that this algorithm is sound :

Theorem 3.2.2 : Soundness

If $\text{TYPE}(B, e) = e' : \tau, \leq, A$, where B contains only type schemes, then :

$$A \cup B, \leq, e \vdash e' : \tau$$

Proof : By induction on the structure of e , as follows :

- c : by the inference rule CONST
- x : by VAR
- y : by PVAR, using corollary 1.10.10 and the fact that $SB = B$
- $\lambda x. e$: by ABS, ABS2
- $e_1 e_2$: by the derived inference rule ENRICH (using corollary 1.10.16) and propositions 3.1.3 and 3.1.6.
- $\text{let } x = e_1 \text{ in } e_2$: by the derived inference rule ASSUMP. \square

Completeness of the algorithm is shown by the next theorem, which requires a complex proof. We first prove the following proposition.

Proposition 3.2.3

If $\text{TYPE}(B, e) = e' : \tau, \leq, A$, then all type variables occurring free in $e' : \tau$ occur in τ, \leq , or free in A .

Proof: By induction on e , noting that if $\text{ARROW}(\tau_1, \tau_2, \leq) = \leq', S, a\tau\tau'$ then all type variables in $S\tau_1, S\tau_2$ occur in \leq' or in τ' . \square

Corollary 3.2.4

If $\text{TYPE}(B, e) = e' : \tau, \leq, \{\}$ and $\text{gen}(\{\}, \leq, \tau) = \sigma$, then the bound variables of σ include all free type variables in $e' : \tau$.

Theorem 3.2.5 : Completeness (Limited)

If e is a limited expression and :

$$(1) \quad A \cup B, \leq, e \vdash e' : \tau$$

where A and B are disjoint, A contains only types, and B contains only type schemes, then :

$$\text{TYPE}(B, e) = e'' : \tau, \leq', A'$$

and from :

$$(2) \quad A' \cup B, \leq', e \vdash e'' : \tau$$

we can derive :

$$(3) \quad A \cup B, \leq, e \vdash QS(e'' : \tau) : \tau$$

by using the derived inference rules SUBEN (with $\text{SUBEN}(S, \leq', L) = \leq$), REPLACE (with replacement Q), and ASSUMP (with $A = QSA' \cup A''$) and the inference rule LESS.

Proof: By induction on the derivation (1), as follows :

CONST : we have $\tau' = \pi \ll \rho = \tau, S = ID, Q = [], L$ by corollary 1.10.16

VAR : we have $\tau' = \alpha, S = [\tau/\alpha], Q = [], L$ by corollary 1.10.16

PVAR : we have S', L from the use of SUBEN in the PVAR rule, and $S' = SR$, where R is the renaming introduced by TYPE

AP : we have $\text{TYPE}(B, e_1, e_2) = e'' : \tau, \leq, A'$
 where $e'' : \tau = (R'Q'_1 R(e''_1 : \tau_1))(R'Q'_2 R(e''_2 : \tau_2) : \tau') : \tau$
 and by induction hypothesis we have S_i, Q_i, L_i, A''_i such that :
 $\text{SUBEN}(S_i, \leq_i, L_i) = \leq$
 $A = Q_i S_i A'_i \cup A''_i$
 $S_1 \tau_1 = b \tau''_1 \tau'_2 \leq a \tau_2 \tau = \tau_1$
 $S_2 \tau_2 \leq \tau_2 \leq \tau'_1$

By propositions 2.2.3, 3.1.2, 3.1.3, 3.1.5, 3.1.6 and the fact that \leq_1, \leq_2 will share no type variables, we can find S, Q, L, A'' such that :

$\text{SUBEN}(S, \leq, L) = \leq$
 $SR'R\tau_1 = S_1 \tau_1 = S(a'\tau'\tau) = b\tau''_1 \tau'_2 \leq a\tau_2 \tau$
 $SR'R\tau_2 = S_1 \tau_2 = \leq \tau'_1 = S\tau'$
 and $A = QSA' \cup A''$

and $A \cup B, \leq e_1 e_2 \vdash QS(e'' : \tau) : \tau$ since $S\tau \leq \tau$

ABS : we have $\text{TYPE}(B, \lambda x.e) = e'' : \tau, \leq, A'$
 where $e'' : \tau = (\lambda x. : \tau_2. e''_1 : \tau_1) : a\tau_2 \tau_1$
 and $(x \notin A', \tau_2 = \alpha, a = \supset)$ or $(A'(x) = \tau_2, a = \rightarrow)$
 By induction hypothesis we have S_1, Q_1, L_1, A''_1 .
 If $x \notin A'$ we let $S = [\tau_2/\alpha]S_1$ and if $x \in A'$ we let $S = S_1$
 so that in either case $S(a\tau_2 \tau_1) \leq \tau_2 \rightarrow \tau_1 = \tau$.
 We let Q be as for Q_1 , but without affecting x .
 If $x \in A$ we let $A'' = A''_1 \cup \{x : A(x)\}$
 and if $x \notin A$ we let $A'' = A''_1$.

ABS2 : Similarly.

LESS : By induction hypothesis. \square

Corollary 3.2.6 Completeness

If e is an expression, and :

(1) $B, \leq, e \vdash e' : \tau$

where B contains only type schemes, then :

$\text{TYPE}(B, e) = e'' : \tau, \leq, \{\}$

and from :

(2) $B, \leq, e \vdash e'' : \tau$

we can derive :

(3) $B, \leq, e \vdash S(e'' : \tau) : \tau$

by using the derived inference rule SUBEN (with $\text{SUBEN}(S, \leq, L) = \leq$) and the inference rule LESS.

Proof : By induction, using theorem 3.2.5 if e is a limited expression, and the derived inference rule REPLACES for **let**-expressions. \square

Remark 3.2.7

If we consider the case that $B = A_0, \leq = \Pi$, then for each program $e' : \tau$ we can derive a program $S(e' : \tau) : \tau$ from the output of the type inference algorithm. By the conjecture made in remark 2.8.7, the two programs then have the same meaning (ie their Böhm trees are extensionally equal).

Example 3.2.8

We now return to example 2.3.4 :

$\text{fix } (\lambda x. \text{cons } 1 \ x)$

and justify our assertion that the derivation in that example was the most general one. Following definition 3.2.1, we have :

$\text{TYPE } (A_0, \text{cons}) = \text{cons } [\alpha] : \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha, \Pi, \{ \}$

$\text{TYPE } (A_0, 1) = 1 : \text{nat}, \Pi, \{ \}$

$\text{TYPE } (A_0, \text{cons } 1) = \text{cons } [\alpha] (1 : \text{nat} : \alpha) : \text{list } \alpha \rightarrow \text{list } \alpha, \\ \{ \Pi, [\text{nat} \leq \alpha] \}, \{ \}$

$\text{TYPE } (A_0, x) = x : \beta, \Pi, \{ x : \beta \}$

$\text{TYPE } (A_0, \text{cons } 1 \ x) = \text{cons } [\alpha] (1 : \text{nat} : \alpha) (x : \text{list } \gamma : \text{list } \alpha) : \text{list } \alpha, \\ \{ \Pi, [\text{nat} \leq \alpha], [\gamma \leq \alpha] \}, \{ x : \text{list } \gamma \} \\ = e : \text{list } \alpha, \leq, \{ x : \text{list } \gamma \}$

$\text{TYPE } (A_0, \lambda x. \text{cons } 1 \ x) = (\lambda x : \text{list } \gamma. e : \text{list } \alpha) : \text{list } \gamma \rightarrow \text{list } \alpha, \leq, \{ \}$

$\text{TYPE } (A_0, \text{fix}) = \text{fix } [\theta] : (\theta \rightarrow \theta) \rightarrow \theta$

$\text{TYPE } (A_0, \text{fix } (\lambda x. \text{cons } 1 \ x)) = \text{fix } [\text{list } \alpha]$

$(((\lambda x : \text{list } \alpha. \text{cons } [\alpha] \\ (1 : \text{nat} : \alpha) \\ (x : \text{list } \alpha : \text{list } \alpha) : \text{list } \alpha) \\ : \text{list } \alpha \rightarrow \text{list } \alpha) : \text{list } \alpha \rightarrow \text{list } \alpha) \\ : \text{list } \alpha, \{ \Pi, [\text{nat} \leq \alpha] \}, \{ \}$

By theorem 3.2.5, this is the most general derivation. Note that the typed expression we have produced is not identical to that in example 2.3.4, since it has two redundant coercions of the form $e : \tau : \tau$. A realistic implementation would remove these during code generation. We also note that, as discussed in example 2.3.4, the ‘best’ type for this expression would be list nat . We examine this notion of ‘best’ type further in the following sections, but first we give a more complex example.

Example 3.2.9

Consider the factorial function :

$$\text{fix } (\lambda f. \lambda x. \text{cond } (eq \ x \ 0) \ 1 \ (\text{mult } x(f \ (\text{dec } x))))$$

where *mult* and *dec* have the same type schemes as *plus* and *neg* respectively.

Figure 4 shows the result of TYPE, with the subtype orderings at each stage shown graphically. This shows the typical behaviour of the algorithm, which is to introduce coercions for each application, and by AUNIFY for multiple occurrences of program variables. The subtype ordering grows rapidly in size, until it is substantially simplified by the cycles introduced by *fix*. Since the algorithms we employ at each stage are $O(n^3)$ in the size of the subtype ordering, this makes the algorithm extremely inefficient. In section 3.6 we present an improved algorithm which ensures that the intermediate subtype orderings remain small throughout the type inference procedure.

3.3 Standardisations

Recalling the discussion in Remark 2.8.7, we make the following definition :

Definition 3.3.1

We say that a ground atomic substitution S satisfies a subtype ordering \leq if $S(\leq) = \Pi$. In other words :

- for each α in \leq , $S\alpha \in \Delta$
- for all a, b such that $a \leq b$ we have $Sa \ll Sb$.

In the discussion after example 2.3.3, we indicated that, since we only give a semantics for *programs* derived with the subtype ordering Π , we are only interested in the types obtained by *satisfying* the ‘most general’ subtype ordering produced by TYPE. This leads us to the following concept of a *standardisation*, which is a modification of a subtype ordering without loss of generality in satisfiability :

Definition 3.3.2

A *standardisation* \leq' , S of a subtype ordering \leq is a pair such that S is atomic, $S(\leq)$ is a subtype ordering, and :

- $\text{ENRICH } (S(\leq), L) = \leq', ID$ for some L , or equivalently, $\text{SUBEN } (S, \leq, L) = \leq'$.
- for all atomic R satisfying \leq , there is an atomic R' satisfying \leq' such that $R = R'S$.

Input Expression	Typed Expression (replacing $e:\tau:\tau$ by $e:\tau$ for clarity)	Subtype Ordering	Assumption Set
$eq\ x\ 0$	$eq[\beta](x:d:\beta)$ $(0:nat:\beta):bool$	β $\swarrow \searrow$ $nat\ d$	$\{x:d\}$
$cond(eq\ x\ 0)\ 1$	$cond[\gamma](eq[\beta](x:d:\beta)$ $(0:nat:\beta):bool)$ $(1:nat:\gamma):\gamma \rightarrow \gamma$	γ $\swarrow \searrow$ $nat\ \beta$ $\swarrow \searrow$ $nat\ d$	$\{x:d\}$
$mult\ x$	$mult[\psi](x:\nu:\psi):\psi \rightarrow \psi$	int \downarrow ψ \downarrow ν	$\{x:\nu\}$
$f(dec\ x)$	$(f:\theta(\Sigma\mu))(dec(x:\eta:int):int:\Sigma):\mu$	Σ \downarrow int \downarrow η \rightarrow θ	$\{x:\eta,$ $f:\theta(\Sigma\mu)\}$
$mult\ x(f(dec\ x))$	$mult[\psi](x:\phi:\nu:\psi)$ $((f:\theta(\Sigma\mu))$ $(dec(x:\phi:\eta:int):int:\Sigma):\mu:\psi):\psi$	Σ \downarrow int \downarrow η \downarrow ψ \downarrow ν \downarrow ϕ \rightarrow θ	$\{x:\phi,$ $f:\theta(\Sigma\mu)\}$
$cond(eq\ x\ 0)\ 1$ $mult\ x(f(dec\ x))$	$cond[\gamma](eq[\beta](x:\Sigma:d:\beta)$ $(0:nat:\beta):bool)$ $(1:nat:\gamma)$ $(mult[\psi](x:\Sigma:\phi:\nu:\psi)$ $((f:\theta(\Sigma\mu))$ $(dec(x:\Sigma:\phi:\eta:int):int:\Sigma)$ $:\mu:\psi):\psi:\gamma):\gamma$	Σ \downarrow int \downarrow η \downarrow ψ \downarrow ν \downarrow ϕ \rightarrow θ	$\{x:\Sigma,$ $f:\theta(\Sigma\mu)\}$
$fix(\lambda f. \lambda x.$ $cond(eq\ x\ 0)\ 1$ $(mult\ x$ $(f(dec\ x))))$	$fix[int](\lambda f:int \rightarrow int. (\lambda x:int. cond[int]$ $(eq[\beta](x:int:d:\beta)(0:nat:\beta):bool)$ $(1:nat:int)$ $(mult[int](x:int)((f:int \rightarrow int)(dec(x:int):int):int)$ $:int):int):int \rightarrow int): (int \rightarrow int) \rightarrow int \rightarrow int$ $:int \rightarrow int$	β \downarrow d \downarrow int	$\{\}$

Figure 4 - The application of TYPE to the factorial function

In section 3.4 we will use the concept of standardisation to provide an algorithm for finding substitutions to satisfy subtype orderings. This means that if :

$$\text{TYPE } (A_0, e) = e' : \tau, \leq, \{ \}$$

then, by finding an S satisfying \leq , we obtain a program $S(e' : \tau)$ which we can then reduce.

The following proposition shows that we can *compose* standardisations.

Proposition 3.3.3

Let \leq' , S be a standardisation of \leq and \leq'' , S' be a standardisation of \leq' . Then the *composition* of these standardisations, defined to be \leq'' , $S'S$ is a standardisation of \leq , and such composition is associative.

Proof : By corollary 1.10.20, $S'S (\leq)$ is a subtype ordering, and there is an L'' such that $\text{SUBEN } (S'S, \leq, L'') = \leq''$. Also if R satisfies \leq , then there is an R' satisfying \leq' such that $R = R'S$, and hence there is an R'' satisfying \leq'' such that $R' = R''S'$, so $R = R''S'S$. Associativity is trivial. \square

We now recall definitions 1.2.1 and 1.6.1, in which we required that each «-related equivalence class in Π have either lub or glb defined on all pairs of its elements, and no two «-unrelated elements are \leq -related by any subtype ordering \leq . This leads us to make the following definition :

Definition 3.3.4

If \leq is a subtype ordering, then we say that the type variable or type constructor a has *top* (*bottom*) δ in \leq if the \leq -related equivalence class of a contains a «-related equivalence class that has lub (glb) δ .

Clearly if α is \leq -related to some δ' , then α has either a top or a bottom in \leq , and possibly both. For example, in the subtype ordering shown in Figure 5a, $\alpha, \beta, \gamma, \zeta$ and η have top *atom*, ψ has top and bottom *void*, θ has top *list* and bottom *optional*, ϕ has top \rightarrow and bottom \supset , and ξ has neither top or bottom.

The following theorem characterises the standardisations which we will use in our second type inference algorithm.

Theorem 3.3.5

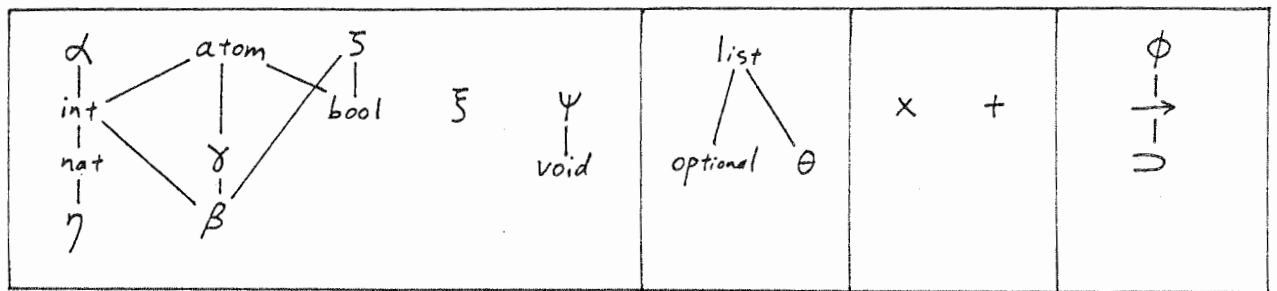
The following pairs \leq', S are standardisations of a subtype ordering \leq :

- TOP: $\leq', S = \text{ENRICH}(\leq, \alpha \leq \delta)$ where α has top δ in \leq
- BOTTOM : $\leq', S = \text{ENRICH}(\leq, \delta \leq \alpha)$ where α has bottom δ in \leq
- CTOP: $S(\leq), S$ where $S = [\beta/\alpha]$, α of rank 0 has top δ in \leq , and β is a new type variable with all the colours of δ , provided δ has more colours than α .
- CBOTTOM: $S(\leq), S$ where $S = [\beta/\alpha]$, α of rank 0 has bottom δ in \leq , and β is a new type variable with all the colours of δ , provided δ has more colours than α .
- GLB: $\leq', S = \text{ENRICH}(\leq, \alpha \leq \delta)$ if $\alpha \leq \delta_i$ for $1 \leq i \leq n$, and δ is the glb of the δ_i .
- LUB: $\leq', S = \text{ENRICH}(\leq, \delta \leq \alpha)$ if $\delta_i \leq \alpha$ for $1 \leq i \leq n$, and δ is the lub of the δ_i .
- MAX: $\leq', S = \text{ENRICH}(\leq, \alpha \leq \delta)$ if $\delta' \leq \alpha$ and δ is the greatest element of Π above δ' .
- MIN: $\leq', S = \text{ENRICH}(\leq, \delta \leq \alpha)$ if $\alpha \leq \delta'$ and δ is the least element of Π below δ' .
- COLOURU: $S(\leq), S$ where $S = [\beta/\alpha]$, $\alpha \leq \gamma$ of rank 0, and β is a new type variable with all the colours of α and γ , provided β has more colours than α .
- COLOURD: $S(\leq), S$ where $S = [\beta/\alpha]$, $\gamma \leq \alpha$ of rank 0, and β is a new type variable with all the colours of α and γ , provided β has more colours than α .

Proof: Clearly an atomic substitution results in each case, and by corollary 1.10.18, if $\text{ENRICH}(\leq, L) = \leq', S$, then $\text{SUBEN}(S, \leq, SL) = \leq'$. By definition 1.6.1, we can easily verify that \leq' is a subtype ordering in each case, and that all cases define standardisations, since for the cases involving $\text{ENRICH}(\leq, a \leq b)$, we must have $Ra \ll Rb$ for each R satisfying \leq . \square

Note that if α has top δ , then the MAX rule, if applicable, produces the same result as the TOP rule. However, if α does not have a top, then the MAX rule may still be applicable. For MIN and BOTTOM, the situation is similar. We note the following notational conventions.

5a. An example subtype ordering



5b. Its regular standardisation

$$S = [\text{nat}/\gamma, \text{atom}/\delta, \text{void}/\psi, \rightarrow/\phi]$$

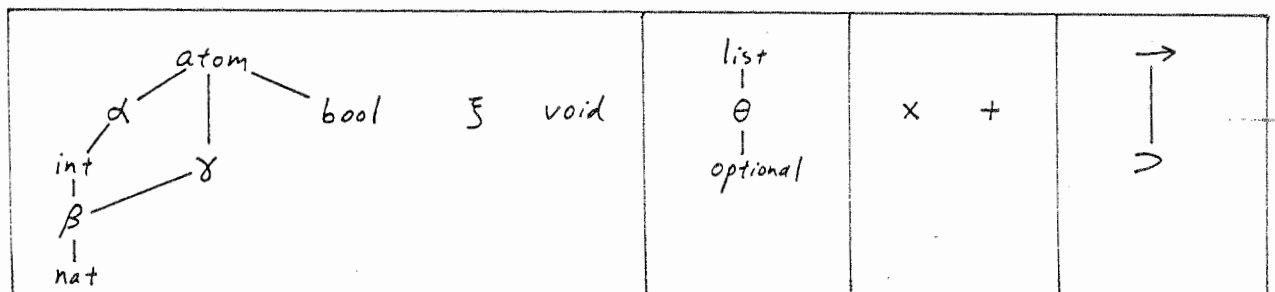


Figure 5 - An example subtype ordering and its regular standardisation

5c. Action of STANDARDISE - the first ENRICH

$$S = [\text{void} / \psi, \rightarrow / \phi]$$

	$\begin{array}{c} \text{list} \\ \\ \emptyset \\ \\ \text{optional} \end{array}$	$\times \quad +$	$\begin{array}{c} \rightarrow \\ \\ \supset \end{array}$
--	--	------------------	--

5d. Action of STANDARDISE - the main loop

Type variable selected	δ_i above variable	consequences	δ_j' below variable	consequences
η	nat, int, atom	$\text{nat} \leq \eta \leq \text{nat}$	nat	—
β	int, atom	$\text{nat} \leq \beta$	nat	—
α	atom	—	nat, int	—
γ	atom	—	nat	—
ζ	atom	—	nat, bool	$\text{atom} \leq \zeta \leq \text{atom}$
θ	list	—	optional	—

Figure 5 (continued) - The action of STANDARDISE

Definition 3.3.6

We say that a *standardisation rule* (TOP, BOTTOM, etc) is *applicable* to a type variable α in a subtype ordering \leq if \leq' , S is the result of a single application of that rule, $\leq' \neq \leq$, and α is the type variable referred to in the rule. We also say that the rule is *applicable to* \leq , and call \leq' , S an *instance* of the rule. We say that \leq' , R is a *regular standardisation* of \leq if \leq' , $R = \leq$, ID or if \leq' , R results from composing one or more instances of standardisation rules, as in proposition 3.3.3.

Figure 5b shows a regular standardisation of the subtype ordering of Figure 5a. The following algorithm calculates the (unique) *maximal* regular standardisation of a subtype ordering, ie the regular standardisation to which no standardisation rule is applicable. The required property is proved in Theorem 3.3.12.

Definition 3.3.7

Let \leq be a subtype ordering. Then we define the operation STANDARDISE as follows :

- For each α_i in \leq , let a_i (b_i) be the top (bottom) of α_i respectively, if it is defined, and α_i otherwise.
- Let \leq' , $S'' = \text{ENRICH}(\leq, \alpha_1 \leq a_1 ; b_1 \leq \alpha_1; \dots; \alpha_n \leq a_n; b_n \leq \alpha_n)$
- Let S' be a substitution mapping type variables of rank 0 in \leq' -related equivalence classes with top (bottom) to new type variables with all the colours of their top (bottom), provided the variables have fewer colours than their top (bottom); and mapping type variables of rank 0 in \leq' -related equivalence classes with no top or bottom to new type variables with all the colours of the equivalence class, provided the new type variables have more colours than the old ones.
- Let $\leq'' = S'(\leq')$
- Let G be the least splitting of \leq'' , viewed as a directed graph, and let H be the transitive closure of G .
- Let $R = ID$, initially.
- Let T be the family of \leq'' -related equivalence classes of type variables in \leq'' which have a top (bottom), topologically sorted in ascending (descending) order.
- If $a = \alpha$, $b = \delta$ or $a = \delta$, $b = \alpha$, let the subroutine ADDEDGE (a , b) perform the following action :
 - Let the β_i be such that $[b \leq \beta_i]$, $[\beta_i \leq a]$ are in H , and let $S = [\delta/\beta_1, \dots, \delta/\beta_m]$
 - Apply S to H , and delete the β_i from T .

- Add the edge $[Sa \leq Sb]$ to G , and update H to be the transitive closure of the result.
- Topologically sort the equivalence class which contained α again.
- Return the substitution S .
- For each α in T , chosen in order from each sorted equivalence class with top, do the following :
 - Let δ be the glb of the δ_i above α in H .
 - If $[\alpha \leq \delta]$ is not in H then let $S_1 = \text{ADDEDGE}(\alpha, \delta)$, otherwise let $S_1 = ID$.
 - If δ' is the least element of Π below δ , $S_1\alpha = \alpha$, and $[\delta' \leq \alpha]$ is not in H , then let $S_2 = \text{ADDEDGE}(\delta', \alpha)$, otherwise let $S_2 = ID$.
 - If $S_2S_1\alpha = \alpha$, δ'' is the lub of the δ'_j below α in H , and $[\delta'' \leq \alpha]$ is not in H , then let $S_3 = \text{ADDEDGE}(\delta'', \alpha)$, otherwise let $S_3 = ID$.
 - Let $R := S_3S_2S_1R$ and delete α from its equivalence class.
- For each α in T , chosen in order from each sorted equivalence class with bottom, but not top, do the dual action.
- Let $\text{UNION}(H) = \leq''', ID$
- Define $\text{STANDARDISE}(\leq) = \leq''', RS'S''$

Figures 5c and 5d show the action of the STANDARDISE algorithm applied to the subtype ordering of Figure 5a. It can be seen that the result is that shown in Figure 5b. The following lemmas lead to Theorem 3.3.12 which proves the required property of the algorithm.

Lemma 3.3.8

During each iteration of the loop in definition 3.3.7, standardisation rules are applied to the subtype ordering represented by H . Furthermore H is acyclic at each stage, and hence the final UNION indeed returns the substitution ID .

Proof : Each edge added involves a use of the GLB, MIN or LUB rules (or dually LUB, MAX or GLB rules). By theorem 3.3.5, any cycles produced will contain exactly one type constructor δ , and the procedure for adding edges equates all type variables in the cycle to δ . \square

Lemma 3.3.9

For each iteration of the above algorithm, no standardisation rule is applicable to any previously examined type variable.

Proof: The TOP, BOTTOM, CTOP, COLOURU, COLOURD and CBOTTOM rules are all applied initially, and clearly can only be applied once. Each iteration applies GLB, MIN, LUB (or LUB, MAX, GLB). Now MAX (MIN), which is the omitted rule, would not be applicable, since TOP (BOTTOM) has already been applied. Furthermore, MIN (MAX) is applied in its most general form, since if $\alpha \leq \delta_i$, δ is the glb of the δ_i and the least element of Π below δ is δ' , then if there is a least element of Π below any δ_i , it is δ' . Now since MIN (MAX) is applied before LUB(GLB), no rule will be applicable to the most recently examined type variable. Since type variables are chosen in topologically sorted order, only the use of the GLB(LUB) rule could affect a previously examined type variable other than the most recently examined, and it does not do so, since if δ is the glb of the set D , then $D \cup D'$ and $D \cup D' \cup \{\delta\}$ have the same glb. \square

Lemma 3.3.10

If \leq_1, S_1 and \leq_2, S_2 are instances of standardisation rules applied to \leq , then we can find \leq', S' such that for $1 \leq i \leq 2$, either $\leq', S' = \leq_i, S_i$ or $S' = R_i S_i$ and \leq', R_i is an instance of a standardisation rule applied to \leq_i .

Proof: By cases. \square

Corollary 3.3.11

Each subtype ordering has a unique maximal regular standardisation.

Proof: Since the process of applying standardisation rules must terminate. \square

Theorem 3.3.12

STANDARDISE (\leq) computes the unique maximal regular standardisation of \leq .

Proof: By lemmas 3.3.8 and 3.3.9 and corollary 3.3.11. \square

This motivates the following definition :

Definition 3.3.13

If STANDARDISE (\leq) = \leq', S , we will speak of \leq' as a *standardised* subtype ordering, or as the *standardised form* of \leq .

In the next section we will show how to calculate, from a standardised subtype ordering \leq' , a substitution R which satisfies \leq' . If $\text{STANDARDISE}(\leq) = \leq'$, S , then RS will satisfy \leq .

Figure 6 shows the action of the STANDARDISE algorithm on the intermediate subtype ordering of Figure 4. Although standardisation does not reduce the size of this subtype ordering, it produces a more ‘regular’ structure. This is exploited in section 3.5, where *simplifications* become possible, which reduce the size of the subtype ordering, and hence make possible a more efficient type inference algorithm.

Remark 3.3.14

The STANDARDISE algorithm will execute in time $O(n^3)$, where n is the number of type constructors in the arguments. This follows since :

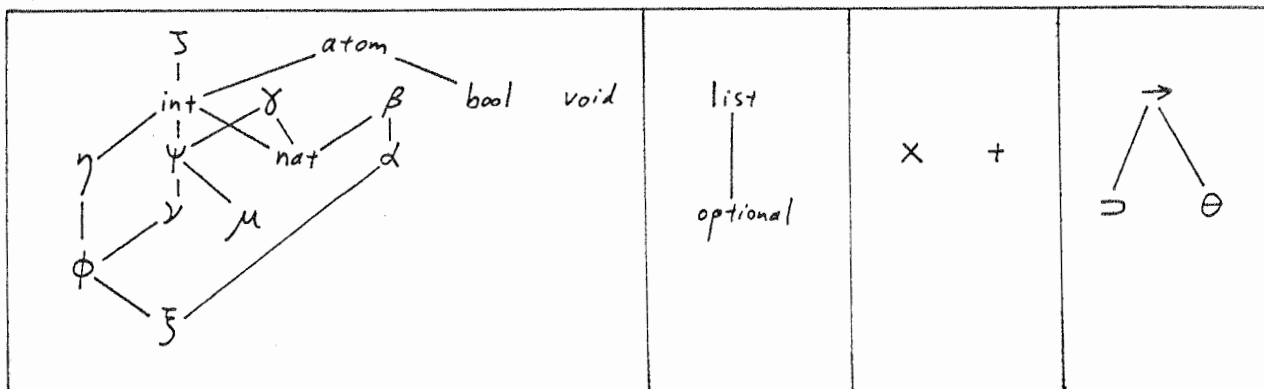
- the ADDEGE subroutine is $O(n^2)$ and can be applied three times for each type variable
- a transitive closure is performed

Note that forming H as the transitive closure of G can be done as part of the first ENRICH operation in definition 3.3.7. Note also that the ENRICH and UNION operations need not perform any of the checks outlined in remark 1.9.4, since the result must be a legal subtype ordering. Indeed, the final UNION need only consist of a transitive reduction, since its resultant substitution is ID . The division into \leq' -related equivalence classes can be done by a connected components algorithm as described in remark 1.9.4

3.4 Satisfying Subtype Orderings

We now show how to satisfy any standardised subtype ordering, and hence how to satisfy *any* subtype ordering, by first standardising it. Recall our Remark 2.8.7, that this is a kind of ‘compilation’, since it maps typed expressions to programs, which can be reduced. It does this by ‘fixing’ all coercions to be particular coercions on ground types; fixing all polymorphic variables to have ground type parameters, thus choosing particular rewrite rules for *plus* and *eq*; and fixing all applications to have functional constructor \supset or \rightarrow , thus choosing particular rewrite rules for application. We first give the following property of standardised subtype orderings, which we will exploit :

6a. Subtype Ordering from Figure 4



6b. Result of STANDARDISE

S = ID

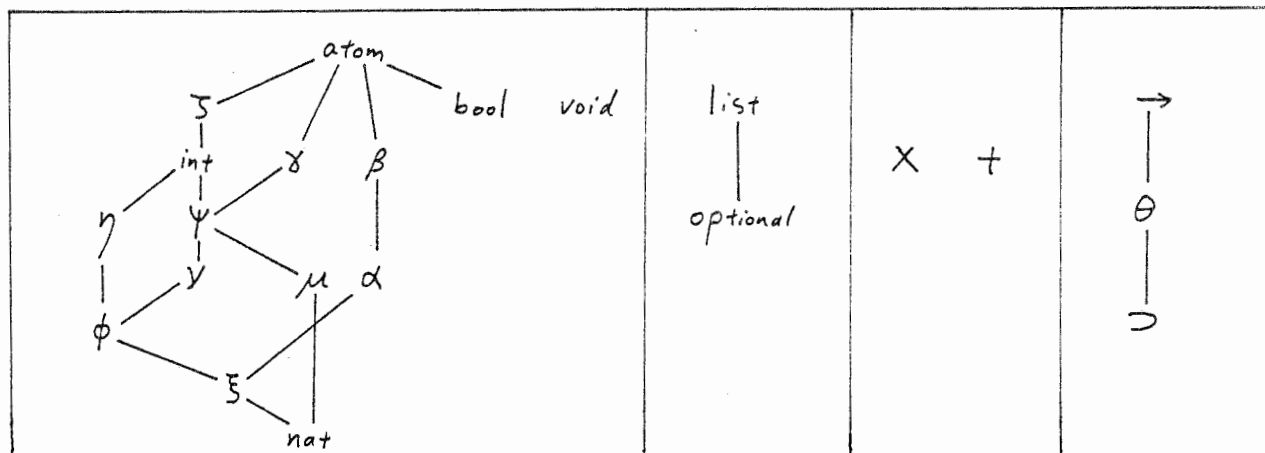


Figure 6 - The action of STANDARDISE: a complex example

Proposition 3.4.1

Let α have a top in \leq . Then there is a type constructor δ such that $\alpha \leq \delta$, and $\alpha \leq \delta'$ implies $\delta \ll \delta'$. Dually, let α have a bottom in \leq . Then there is a type constructor δ such that $\delta \leq \alpha$, and $\delta' \leq \alpha$ implies $\delta' \ll \delta$.

Proof: By the TOP (BOTTOM) rule the required type constructor exists, and the second property follows from the GLB (LUB) rule. \square

We can now define an algorithm to find a substitution which satisfies a standardised subtype ordering.

Definition 3.4.2

Let \leq be a standardised subtype ordering. Then we define the operation SATISFY as follows :

Choose an arbitrary δ_r for each rank r . For example :

$$\delta_0 = \text{void}$$

$$\delta_1 = \text{list}$$

$$\delta_2 = \times$$

$$\delta_* = \rightarrow$$

Let $\alpha_1, \dots, \alpha_n$ be all the type variables in \leq , and for each α_i define S_i as follows :

- If α_i has a bottom in \leq , let δ_i be the type constructor existing by proposition 3.4.1, and let $S_i = [\delta_i/\alpha_i]$.
- If α_i has a top, but not a bottom, in \leq , let δ_i be the type constructor existing by proposition 3.4.1, and let $S_i = [\delta_i/\alpha_i]$.
- If α_i has neither a top or a bottom in \leq , and α_i has rank r , let $S_i = [\delta_r/\alpha_i]$.

Define $\text{SATISFY}(\leq) = S_1 \dots S_n$.

Proposition 3.4.3

Let $\text{SATISFY}(\leq) = S$. Then S satisfies \leq .

Proof: Clearly each type variable in \leq is mapped to a type constructor, so we need only show that $a \leq b$ implies $Sa \ll Sb$. If a, b are from an equivalence class with no top or bottom, then they are both type variables, and $Sa = Sb$. If a, b are from an equivalence class with bottom, then we have four cases :

- $\alpha \leq \beta$, and $S\alpha, S\beta$ are the type constructors existing by proposition 3.4.1. Then since $S\alpha \leq \beta$, we have $S\alpha \ll S\beta$.
- $\alpha \leq \delta$, trivially, since $S\alpha \ll \alpha$.
- $\delta \leq \alpha$, by proposition 3.4.1
- $\delta \leq \delta'$, trivially.

If a, b are from an equivalence class with top, but no bottom, the proof is similar. \square

Figure 7 shows the result of applying the SATISFY algorithm to the standardised subtype ordering of Figure 5b.

Remark 3.4.4

In section 3.6 we will give an improved type inference algorithm TYPE2, such that :

$$\text{TYPE2}(B, e) = e' : \tau, \leq, A$$

and \leq is a standardised subtype ordering. In particular, for expressions e such that :

$$\text{TYPE2}(A_0, e) = e' : \tau, \leq, \{ \}$$

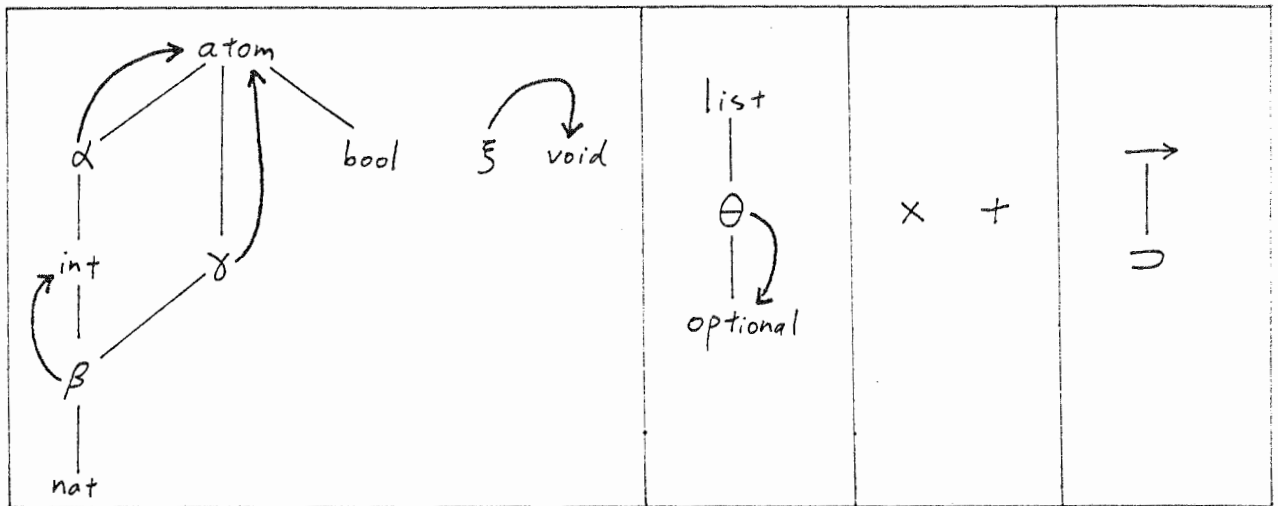
$$\text{SATISFY}(\leq) = S$$

we obtain a program $S(e' : \tau)$ which we can reduce. Indeed, if we wish to give a semantics for the expression e , the Böhm tree of the program $S(e' : \tau)$ is a suitable value.

3.5 Simplifications

Before we describe our improved type inference algorithm, we require the concept of a *simplification* of a standardised subtype ordering. The intention is that simplification produces a 'simpler' subtype ordering which is still standardised. Simplifications are closely related to standardisations, as the following definition makes clear. However, we relax the requirement of being able to satisfy the result in exactly the same ways that we could satisfy the argument. Instead we require that the ground types that could be obtained by applying the satisfying substitution to a particular type can be obtained by applying the new satisfying substitution and coercing. The concept of simplification formalizes the concept of 'best type' introduced in section 2.

7a. Choices made by SATISFY on standardised subtype ordering from Figure 5b.



7b. Result of SATISFY

[atom/ α , int/ β , atom/ γ , void/ ξ , optional/ θ]

Figure 7 - The action of SATISFY

Definition 3.5.1

A *simplification* \leq' , S of a standardised subtype ordering \leq , with respect to a type τ , is a pair such that S is atomic, $S(\leq)$ is a subtype ordering, and :

- $\text{ENRICH}(S(\leq), L) = <'$, ID for some L , or equivalently,
 $\text{SUBEN}(S, \leq, L) = \leq'$
- \leq' is standardised
- for all atomic R satisfying \leq , there is an atomic R' satisfying \leq' such that $R'S\tau \ll R\tau$

Example 3.5.2

Consider example 2.3.3, where for the expression $\lambda x. \text{plus } x \ 1$, we obtained the type $\beta \rightarrow \alpha$ and the subtype ordering $\{\Pi, [\text{nat} \leq \alpha], [\alpha \leq \text{int}], [\beta \leq \alpha]\}$.

Satisfying this subtype ordering and applying the satisfying substitution, we obtain the ground types $\text{nat} \rightarrow \text{nat}$, $\text{nat} \rightarrow \text{int}$, $\text{nat} \rightarrow \text{atom}$, $\text{int} \rightarrow \text{int}$ and $\text{int} \rightarrow \text{atom}$. The standardised form of this subtype ordering is

$\{\Pi, [\text{nat} \leq \beta], [\beta \leq \alpha], [\alpha \leq \text{int}]\}$, which clearly can be satisfied in exactly the same ways. The pair $\{\Pi, [\text{nat} \leq \alpha], [\alpha \leq \text{int}]\}$, $[\alpha/\beta]$ is a simplification of this standardised subtype ordering, since by satisfying we obtain the ground types $\text{nat} \rightarrow \text{nat}$ and $\text{int} \rightarrow \text{int}$, and these can be coerced to give all the ground types obtained above. This is precisely what we meant by saying that $\alpha \rightarrow \alpha$, with the simplified subtype ordering, was the 'best' type for $\lambda x. \text{plus } x \ 1$.

We compose simplifications in the same way as standardisation :

Proposition 3.5.3

Let \leq' , S be a simplification of \leq , with respect to τ , and \leq'' , S' be a simplification of \leq' with respect to $S\tau$. Then the *composition* of these simplifications, defined to be \leq'' , $S'S$ is a simplification of \leq with respect to τ , and such composition is associative.

Proof: As for proposition 3.3.3, noting that if R satisfies \leq , there is an R' satisfying \leq' such that $R'S\tau \ll R\tau$, and hence there is an R'' satisfying \leq'' such that $R''S'S\tau \ll R'S\tau \ll R\tau$. \square

The following theorem characteristises some useful simplifications, where we use the concept of *left* and *right occurrences* in a type, given in definition 1.4.6. The intuition is that we can safely *decrease* the result type of an expression (and indeed, all type variables occurring only right), and safely *increase* the argument types of an expression (and indeed, all type variables

occurring only left). These are both ways of decreasing the overall type of the expression, which must lead to a 'better' type, in the sense that $list\ nat$ is a better type for the infinite list of 1's than $list\ \alpha$ with $nat \leq \alpha$ (example 2.3.4).

Theorem 3.5.4

The following pairs \leq', S are simplifications of a standardised subtype ordering \leq with respect to a type τ :

EQUUP : $S(\leq), S$

where $S = [a/\alpha]$, provided that $\alpha \leq a$, and whenever $\alpha \leq b$, then $a \leq b$; and α does not occur right in τ

EQDOWN : $S(\leq), S$

where $S = [a/\alpha]$, provided that $a \leq \alpha$, and whenever $b \leq \alpha$, then $b \leq a$; and α does not occur left in τ

MOVEUP : $\leq', ID = ENRICH(\leq, a \leq \alpha)$

provided that there is at least one $b \neq \alpha$ such that $\alpha \leq b$, and for all such b , $a \leq b$, and α does not occur right in τ , and we do not already have $\alpha \leq a$.

MOVEDOWN : $\leq', ID = ENRICH(\leq, a \leq \alpha)$

provided that there is at least one $b \neq \alpha$ such that $b \leq \alpha$, and for all such b , $a \leq b$, and α does not occur left in τ , and we do not already have $a \leq \alpha$.

Proof: An atomic substitution clearly results in each case. Now EQUUP and EQDOWN can be viewed as adding $a \leq \alpha$ and $\alpha \leq a$ respectively. Thus all four rules add an inequality $c \leq d$, but this has no further implications, since if $e \leq c$, $d \leq f$, we already have $e \leq f$ in each case. In particular, the substitution returned for MOVEUP and MOVEDOWN is indeed ID , and the resultant subtype orderings in each case are standardised. We show the remaining condition is true for EQUUP and MOVEUP, since the proof for the other two cases is dual:

EQUUP : Let R satisfy \leq . Then $R\alpha \ll Ra$, and we can choose R' to have the same effect as R on all type variables other than α . Hence by proposition 1.6.5, $R'S\tau \leq R\tau$. Also, R' clearly satisfies $S(\leq)$.

MOVEUP : Let R satisfy \leq , and let the b_i be above α , and the Rb_i have glb δ . Then $R\alpha \ll \delta$ and $Ra \ll \delta$, and we can choose R' to have the same effect as R on all type variables other than α , and such that $R'\alpha = \delta$. By proposition 1.6.5, $R'\tau \ll R\tau$. Also, R' satisfies \leq' , since if the c_j are below α , $Rc_j \ll R\alpha \ll \delta \ll Rb_i$. \square

Example 3.5.5

Figure 8 shows how by applying the simplification rules EQUIP and EQDOWN to the standardised subtype ordering of Figure 6, with the type $\theta(\zeta, \mu) \rightarrow \xi \rightarrow \gamma$, we can obtain a much smaller subtype ordering containing only one type variable. If we relax the requirement that simplifications operate on standardised subtype orderings, and apply the simplification rules to the unstandardised subtype ordering of Figure 6, we obtain the same result. However, in that case we must also use the MOVEDOWN rule, as shown in Figure 9. It is thus evident that standardisation assists the simplification process.

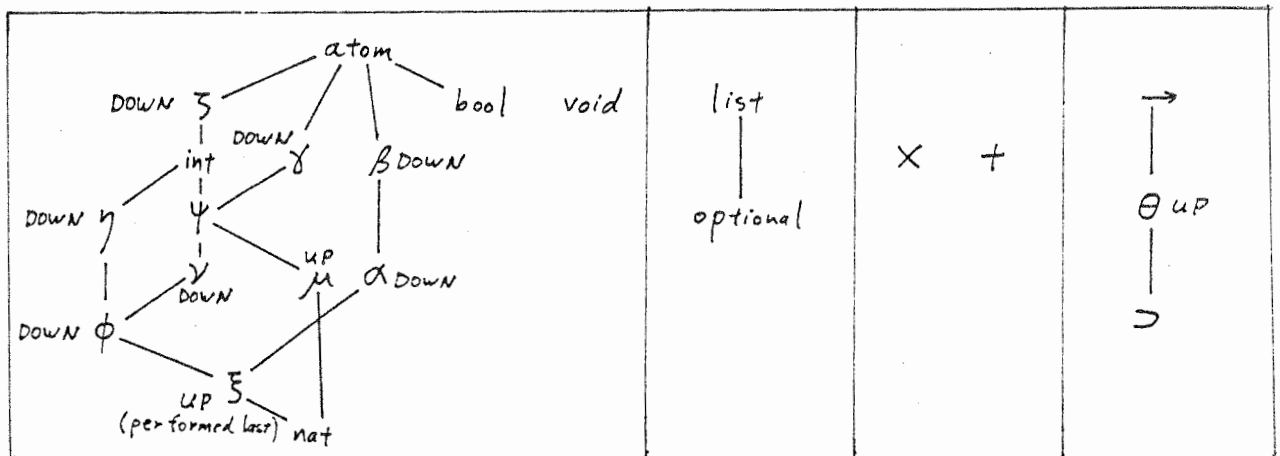
Remark 3.5.6

Having noted that the rules MOVEUP and MOVEDOWN were not needed in the above example, we turn our attention to the benefits of these two rules. Unlike the rules EQUIP and EQDOWN, they do not decrease the number of type variables in a subtype ordering. However, Figure 10 shows a standardised subtype ordering where EQUIP and EQDOWN are not applicable until after MOVEUP is applied. Thus MOVEUP and MOVEDOWN are potentially of benefit in reducing the size of subtype orderings. On the other hand, MOVEUP and MOVEDOWN suffer from a number of disadvantages. First, simplification using all four rules does not lead to unique results. This is in contrast to standardisations, since corollary 3.3.11 showed that standardisations are 'Church-Rosser'. Figure 11 shows two cases where different choices of simplification rules lead to different results. On the other hand, using EQUIP and EQDOWN alone allows us to obtain simplified subtype orderings unique up to renaming.

The second disadvantage of MOVEUP and MOVEDOWN is related to the fact that, just as MOVEUP and MOVEDOWN may make EQUIP and EQDOWN applicable (as in Figure 10), EQUIP and EQDOWN may make MOVEUP and MOVEDOWN applicable (see, for example, Figure 12). Thus, in calculating a maximal regular simplification, we will need to repeatedly re-examine type

8a. Standardised Subtype Ordering from Figure 6 showing applicability of EQUP and EQDOWN

$$\tau = \theta(\zeta, \mu) \rightarrow \zeta \rightarrow \gamma$$



8b. Final Result

$$\tau = (\text{int} \rightarrow \psi) \rightarrow \psi \rightarrow \psi$$

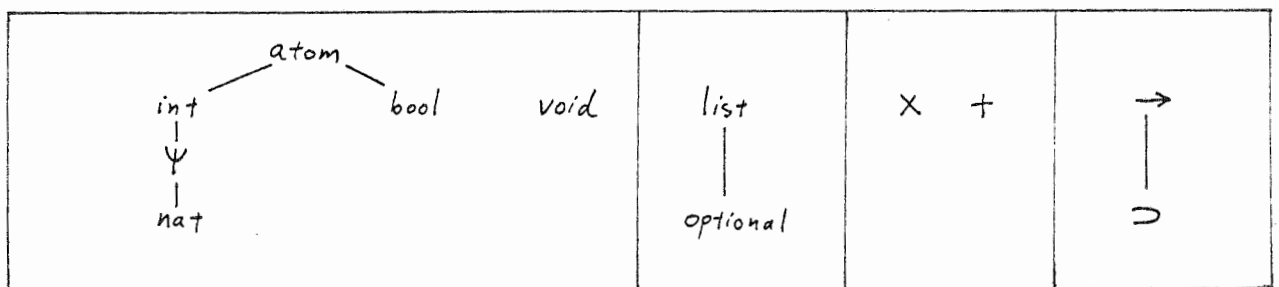
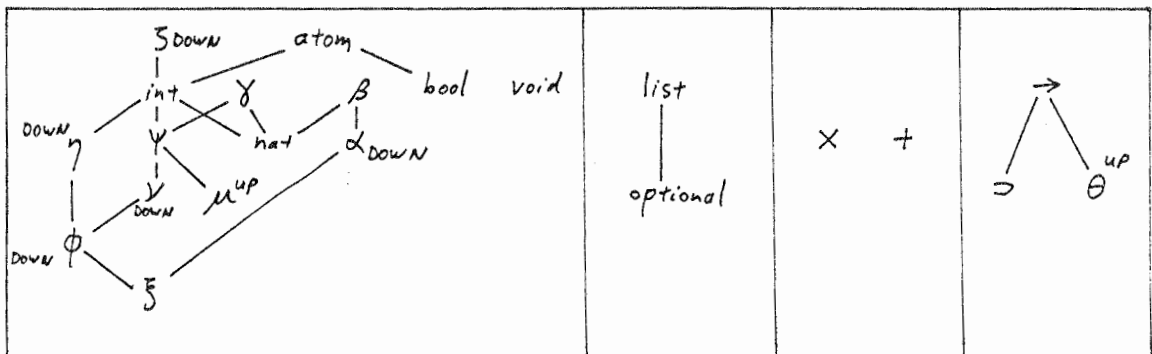


Figure 8 - Simplification of a standardised subtype ordering using EQUP and EQDOWN

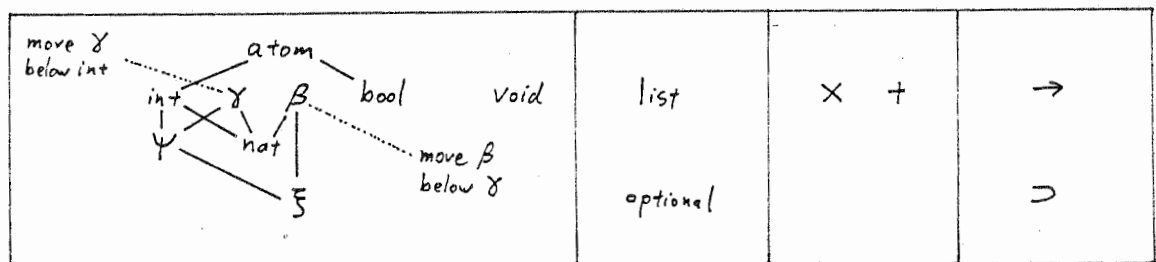
9a. Unstandardised Subtype Ordering from Figure 6 showing applicability of EQUP and EQDOWN

$$\tau = \theta(\mathcal{I}, \mu) \rightarrow \mathcal{F} \rightarrow \mathcal{X}$$



9b. Result, showing applicability of MOVEUP and MOVEDOWN

$$\tau = (\text{int} \rightarrow \psi) \rightarrow \mathcal{F} \rightarrow \mathcal{X}$$



9c. Result, showing applicability of EQUP

$$\tau = (\text{int} \rightarrow \psi) \rightarrow \mathcal{F} \rightarrow \mathcal{X}$$

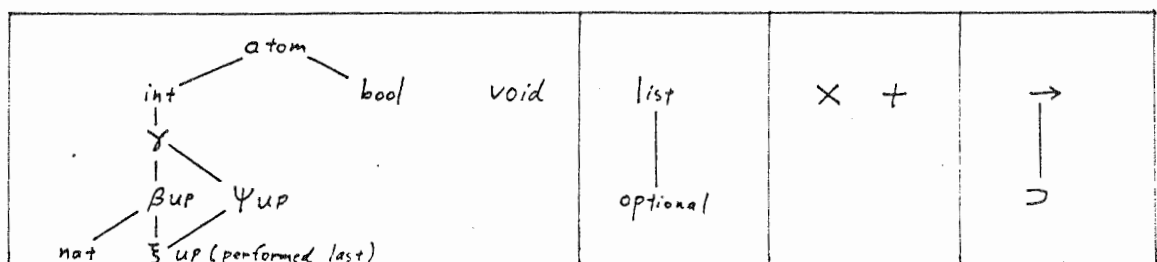
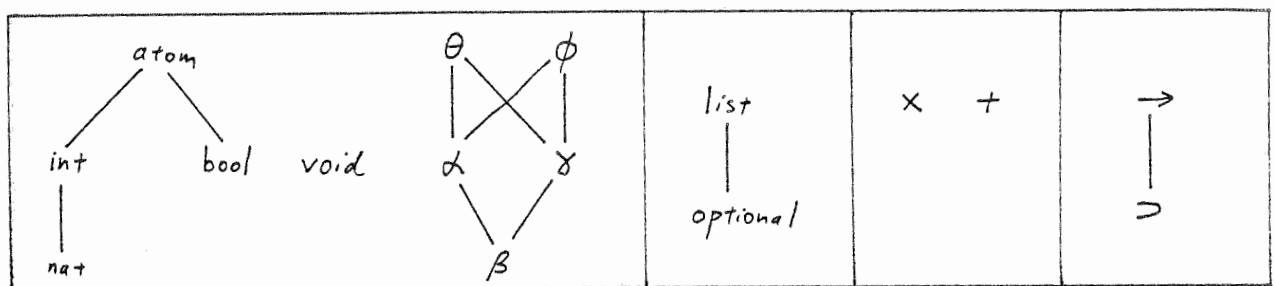


Figure 9 - Simplifying an unstandardised subtype ordering: the need for MOVEDOWN

10a. A Subtype Ordering where EQUP, EQDOWN are inapplicable

$$\tau = \alpha \times \gamma \rightarrow \text{int}$$



10b. After MOVEUP, showing applicability of EQUP and EQDOWN

$$\tau = \alpha \times \gamma \rightarrow \text{int}$$

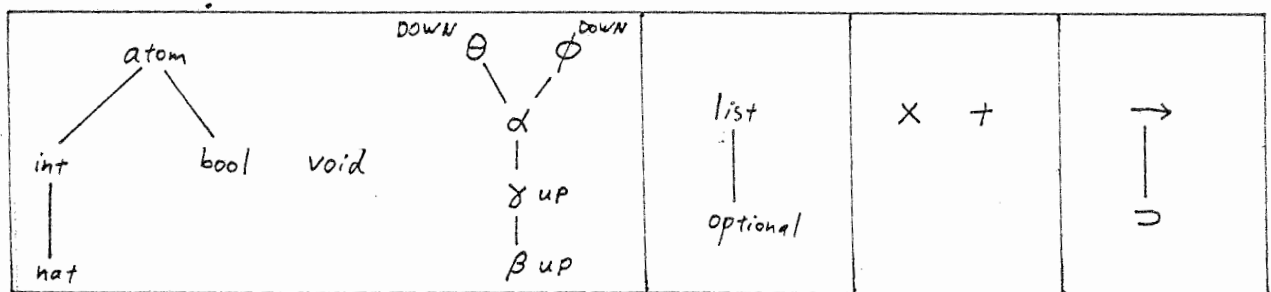
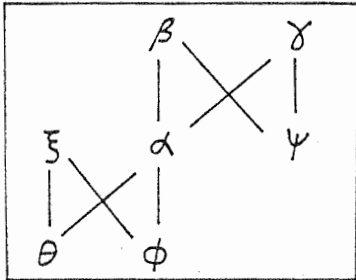


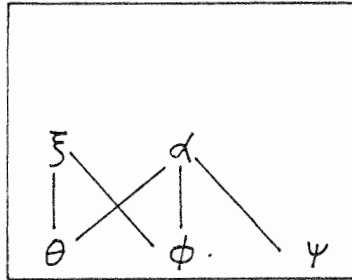
Figure 10 - How MOVEUP aids EQUP and EQDOWN

11a. First example with $\tau = \xi \rightarrow \psi$

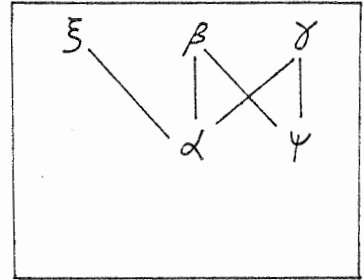
Initial Subtype Ordering
(part of rank 0 section)



After applying MOVEUP
(α above ψ) and EQDOWN

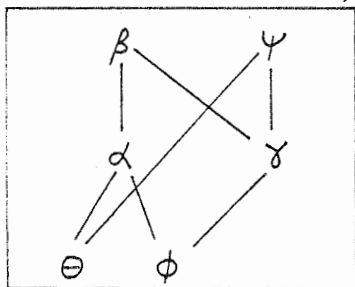


After applying MOVEDOWN
(α below ξ) and EQUP

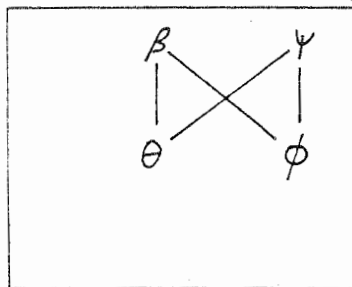


11b. Second example with $\tau = \beta \times \psi \rightarrow \theta \times \phi$

Initial Subtype Ordering
(part of rank 0 section)



After applying EQUP
(to α) and EQDOWN (to β)



After applying MOVEDOWN,
MOVEUP and EQDOWN

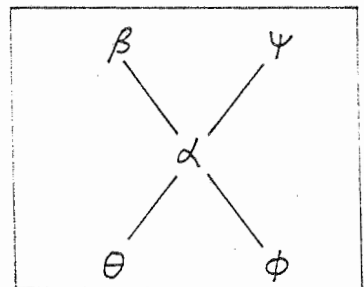
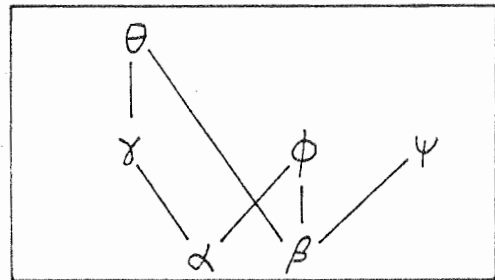
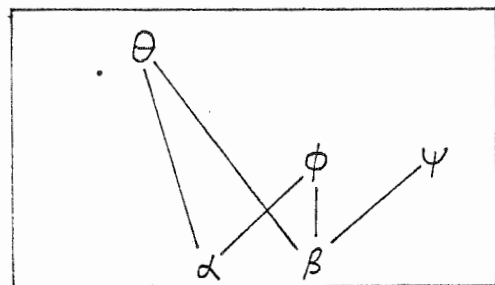


Figure 11 - Church-Rosser counter-examples

- 12a. A subtype ordering
(part of rank 0 section)
with MOVEUP, MOVEDOWN
not applicable for
 $\tau = \alpha \rightarrow \gamma \rightarrow \psi \rightarrow \phi \rightarrow \beta$



- 12b. After applying EQUP,
MOVEUP becomes
applicable



- 12c. Final Result

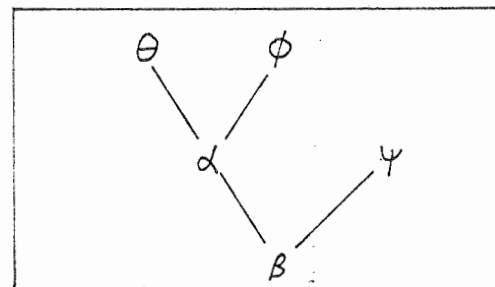


Figure 12- How EQUP can make MOVEUP applicable

variables to see if a rule is applicable. Now applicability of EQUIP and EQDOWN can be tested in constant time, by examining a transitively reduced graph. However, testing applicability of MOVEUP and MOVEDOWN requires a test for containment of sets of vertices, ie

$$\{b \mid \alpha \leq b\} \subseteq \{b \mid a \leq b\}$$

This test must be done for several possible a , so that testing applicability of MOVEUP and MOVEDOWN to α requires time $O(n^2)$. Since each type variable can be examined $O(n)$ times, an algorithm to calculate maximal regular simplifications would require time $O(n^4)$. Thus MOVEUP and MOVEDOWN carry with them a (small) efficiency penalty.

Together with these two disadvantages of MOVEUP and MOVEDOWN, we note that their benefit is reduced by working with standardised subtype orderings, as shown by example 3.5.5. Indeed, we have found no examples where after standardising and using EQUIP and EQDOWN, the simplification rules MOVEUP and MOVEDOWN are applicable. We therefore do not consider MOVEUP and MOVEDOWN in the remainder of this chapter, and make the following definition :

Definition 3.5.7

We say that \leq' , S is a *regular simplification* of \leq with respect to τ if \leq' , $S = \leq$, ID or \leq' , S results from composing one or more instances of the simplification rules EQUIP or EQDOWN only.

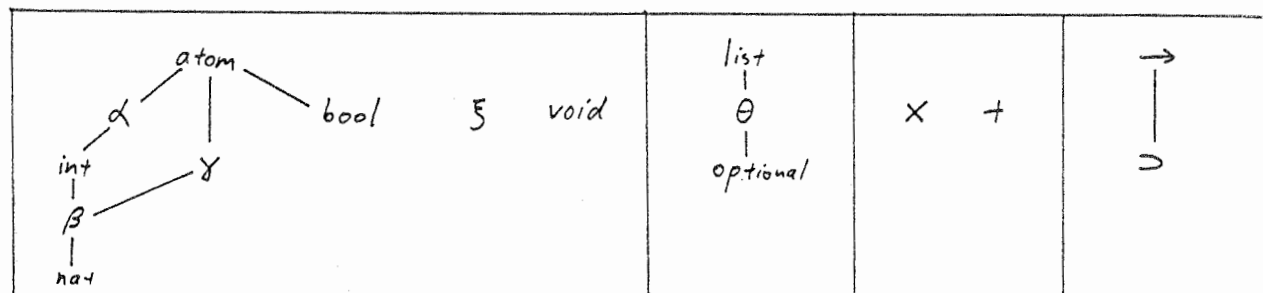
It is clear from proposition 3.5.3 that all regular simplifications will have the form $S(\leq)$, S where $S = [\bar{a}/\bar{\alpha}]$. Figure 13 shows two maximal regular simplifications of the standardised subtype ordering of Figure 5b with respect to the type $\tau = \theta(\beta) \rightarrow \gamma$. Note that they have the form \leq , R and \leq , S where the subtype orderings are the same, but $R \neq S$. However $R\tau = S\tau$. In general, the subtype ordering in maximal regular simplifications is unique up to renaming, and so is the result of applying the substitution to the given type. The substitution itself, however, does not have this property. The following algorithm calculates maximal regular simplifications. Theorem 3.5.15 proves the required properties.

Definition 3.5.8

Let \leq be a standardised subtype ordering, and τ be any type. Then we define the operation SIMPLIFY as follows :

13a. Standardised Subtype Ordering from Figure 5b

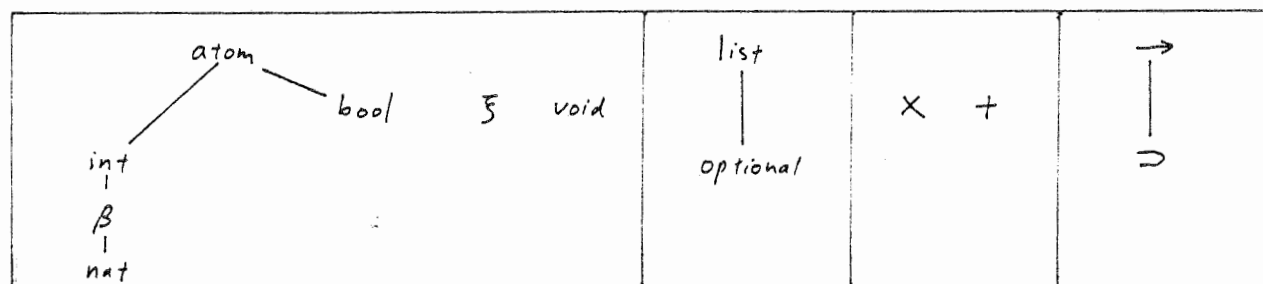
$$\tau = \theta(\beta) \rightarrow \gamma$$



13b. A maximal regular simplification

$$S = [\text{atom}/\alpha, \beta/\gamma, \text{list}/\theta]$$

$$S\tau = \text{list } \beta \rightarrow \beta$$



13c. An alternative substitution

$$R = [\text{int}/\alpha, \beta/\gamma, \text{list}/\theta]$$

$$R\tau = S\tau$$

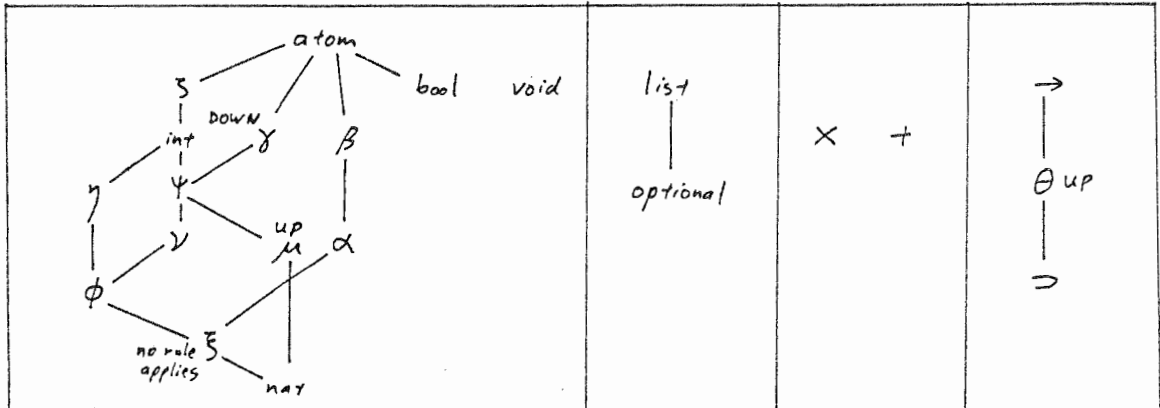
Figure 13 - Two maximal regular simplifications of the standardised subtype ordering from Figure 5b

- Let G be the least splitting of \leq , viewed as a directed graph, and let H be the transitive closure of G .
- Let U, U' initially contain all the type variables in \leq , which do not occur right in τ .
- Let D, D' initially contain all the type variables in \leq , which do not occur left in τ .
- Let $R = ID$, initially.
- Let the subroutine EQUATEUP (α, a) perform the following action :
 - Let $S = [a/\alpha]$
 - Delete $[\alpha \leq a]$ from G
 - For all b such that $[b \leq \alpha]$ is in G , and $[b \leq c]$ is in G for $c \neq \alpha$, and $[c \leq a]$ is in H : delete $[b \leq \alpha]$ from G , and if b is in $U - U'$ then add b to U'
 - Alter all remaining $[b \leq \alpha]$ in G to $[b \leq a]$
 - If a is in U , and α is not in U , then delete a from U, U'
 - If a is in D , and α is not in D , then delete a from D, D'
 - If a is in $D - D'$ then add a to D'
 - Delete α from U, U', D, D', G and H , and delete all edges containing α from H
 - Return the substitution S
- Let the subroutine EQUATEDOWN (α, a) perform the dual action, reversing inequalities and replacing U, U', D, D' by D, D', U, U' respectively.
- While $U' \cup D'$ is not empty, choose some α from $U' \cup U'$, and :
 - if $\alpha \in D'$ then
 - if $[a \leq \alpha]$ is the only edge in G terminating at α , let $S = \text{EQUATEDOWN}(\alpha, a)$ and $R := SR$
 - otherwise delete α from D'
 - if $\alpha \in U'$ then
 - if $[\alpha \leq a]$ is the only edge in G beginning at α , let $S = \text{EQUATEUP}(\alpha, a)$ and $R := SR$
 - otherwise delete α from U'
- Let $\text{UNION}(G) = \leq', ID$
- Define $\text{SIMPLIFY}(\leq, \tau) = \leq', R$.

Figure 14 shows the action of the SIMPLIFY algorithm on example 3.5.5 (presented briefly in Figure 8). The algorithm gives priority to EQDOWN over EQUUP. The following lemmas and theorem give the properties of the algorithm.

14a. Action of SIMPLIFY on example from Figure 8 showing first four type variables examined

$$\tau = \theta(\zeta, \mu) \rightarrow \xi \rightarrow \gamma$$



14b. State after first four type variables examined

$$R\tau = (\zeta \rightarrow \psi) \rightarrow \xi \rightarrow \gamma$$

$$U = \{\alpha, \beta, \gamma, \nu, \phi, \xi\}$$

$$U' = \{\alpha, \beta, \gamma, \nu, \phi\}$$

$$D = D' = \{\alpha, \beta, \gamma, \nu, \phi, \xi\}$$

14c. Graph showing next six type variables examined

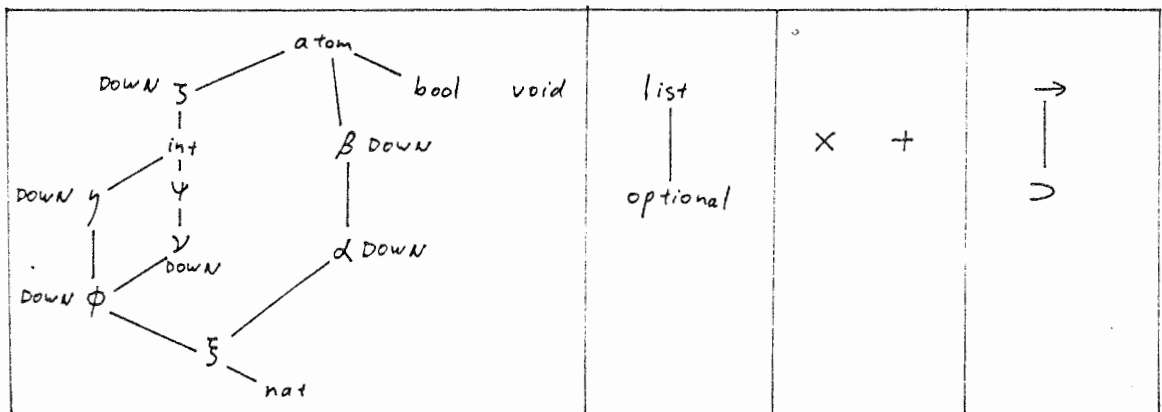


Figure 14 - The action of SIMPLIFY

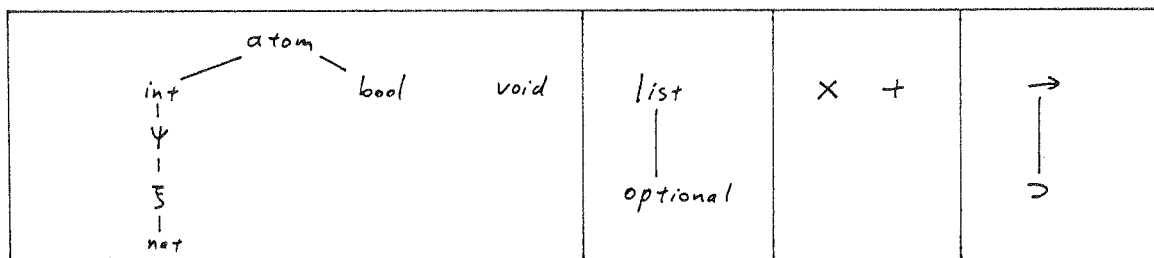
14d. State after first ten type variables examined

$$RT = (int \rightarrow \Psi) \rightarrow \S \rightarrow \Psi$$

$$U = U' = \{\S\}$$

$$D = D' = \{\}$$

14e. Corresponding graph



14f. Final result after re-examining \S

$$RT = (int \rightarrow \Psi) \rightarrow \Psi \rightarrow \Psi$$

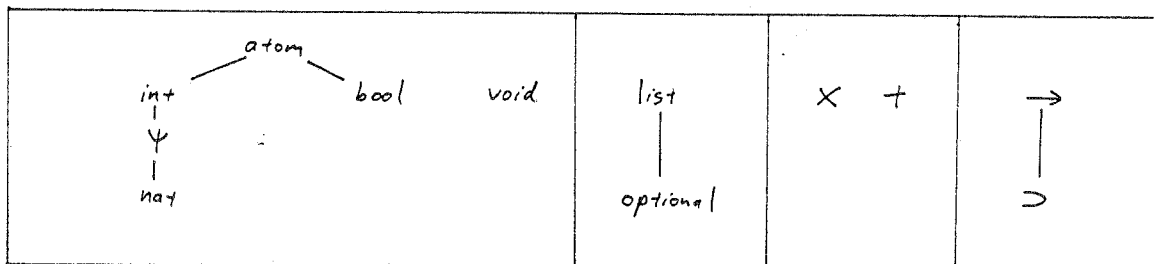


Figure 14 (continued) - Final result of SIMPLIFY

Lemma 3.5.9

During each iteration of the loop in the SIMPLIFY algorithm, an instance of the EQUP or EQDOWN rules is applied, or a type variable to which neither rule applies is deleted from D' and U' . Furthermore, at each iteration :

- H is the transitive closure of G
- G is acyclic and transitively reduced, and $\text{UNION}(G) = \leq, ID$ for some \leq
- $U(D)$ contains exactly the type variables in G which do not occur right (left) in $R\tau$
- U', D' are subsets of U, D respectively

Proof: By comparing theorem 3.5.4 and definition 3.5.8. The final four properties can be shown trivially from definition 3.5.8. \square

A consequence of this lemma is that the final UNION operation is unnecessary if subtype orderings are represented by transitively reduced acyclic graphs, for then the result of the union will be G, ID .

Lemma 3.5.10

During each iteration of the above algorithm, the type variables to which EQUP (EQDOWN) is applicable are all contained in $U'(D')$.

Proof: By the above lemma, the type variables are clearly contained in U and D . Type variables are deleted from U' and D' when EQUP(EQDOWN) does not apply, and are restored whenever deletion of other edges can make the rule applicable. \square

Lemma 3.5.11

The algorithm terminates in time $O(n^3)$, where n is the number of type variables and type constructors in \leq .

Proof: Since EQUP and EQDOWN can be applied at most $O(n)$ times, each type variable is tested for applicability at most $O(n)$ times. Also, the subroutines EQUATEUP and EQUATEDOWN take time $O(n^2)$, and are applied $O(n)$ times. \square

Lemma 3.5.12

If $S_1 (\leq), S_1$ and $S_2 (\leq), S_2$ are two instances of EQUP or EQDOWN applied to \leq with respect to τ , then for $1 \leq i \leq 2$ we can find R_i such that $R_i = ID$ or $R_i S_i (\leq), R_i S_i$ is an instance of EQUP or EQDOWN applied to $S_i (\leq)$ with respect to $S_i \tau$, and such that $R_1 S_1 (\leq) \approx R_2 S_2 (\leq)$ and $R_1 S_1 \tau \approx R_2 S_2 \tau$.

Proof: There are two cases where interference may occur, ie where we cannot choose $R_1 = S_2, R_2 = S_1$.

- $b \leq \alpha \leq a$ where $S_1 = [a/\alpha]$ and $S_2 = [b/\alpha]$.

Clearly $S_1(\leq) = S_2(\leq)$, and since α cannot occur in τ , $S_1\tau = S_2\tau$.

- $\beta \leq \alpha$ where $S_1 = [\beta/\alpha]$ and $S_2 = [\alpha/\beta]$.

Clearly $S_1(\leq) \approx S_2(\leq)$ and $S_1\tau \approx S_2\tau$. \square

Lemma 3.5.13

If $\leq \approx \leq'$ such that $R'(\leq) = \leq'$ and $R(\leq') = \leq$, and $S(\leq)$, S is a regular simplification of \leq with respect to τ , where $S = [\bar{a}/\bar{\alpha}]$, and if $S' = [R' \bar{a}/R' \bar{\alpha}]$, then $S'(\leq')$, S' is a regular simplification of \leq' with respect to $R'\tau$.

Furthermore, $S(\leq) \approx S'(\leq')$ and $S\tau \approx S'R'\tau$.

Proof: By induction on the number of instances of EQUP or EQDOWN composed to form $S(\leq) = S$. \square

Corollary 3.5.14

If $S(\leq)$, S and $R(\leq)$, R are maximal regular standardisations of \leq with respect to τ , then $S(\leq) \approx R(\leq)$ and $S\tau \approx R\tau$.

Proof: By lemmas 3.5.12 and 3.5.13. \square

Theorem 3.5.15

SIMPLIFY (\leq, τ) computes a maximal regular standardisation $S(\leq)$, S of \leq with respect to τ , and $S(\leq)$ and $S\tau$ are unique up to renaming. Furthermore, the algorithm takes time $O(n^3)$, where n is the number of type variables and type constructors in \leq .

Proof: By lemmas 3.5.9 to 3.5.14. \square

Definition 3.5.16

If $\text{SIMPLIFY}(\leq, \tau) = S(\leq)$, S , we will speak of $S(\leq)$ as a *simplified* subtype ordering with respect to τ , or as a *simplified form* of \leq with respect to τ .

We can now present our improved type inference algorithm, which uses STANDARDISE and SIMPLIFY to reduce the size of subtype orderings. Since most of the algorithms applied to subtype orderings are $O(n^3)$, the improved algorithm will be much more efficient, since n will be smaller.

3.6 An Improved Type Inference Algorithm

We now modify the type inference algorithm of section 3.2 to use STANDARDISE and SIMPLIFY to reduce the size of subtype orderings. Recall from definition 3.5.1 that simplifications decrease a given type, where this is safe. Our intuition for the ‘best’ type of an expression is that we must decrease the result type and increase the argument types. We thus make the following useful definition :

Definition 3.6.1

If $A = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$ and τ is a type, then we write $A \rightarrow \tau$ for τ if $n = 0$, and for $\tau_1 \times \dots \times \tau_n \rightarrow \tau$ if $n > 0$.

At each point in our algorithm we will attempt to decrease $A \rightarrow \tau$. Since we wish to work with *simplified* subtype orderings, we make the following definitions :

Definition 3.6.2

We say that a type scheme $\sigma = \forall \bar{\alpha} : \leq. \tau$ is *simplified* if \leq is standardised and simplified with respect to τ .

Definition 3.6.3

Let the assumption set B_0 be as for A_0 in definition 1.11.5, but with :

$$B_0(\text{plus}) = \forall \bar{\alpha} : \{\Pi, [\text{nat} \leq \alpha], [\alpha \leq \text{int}]\}. \alpha \rightarrow \alpha \rightarrow \alpha$$

where α is a basic type variable. It is clear that B_0 will then only contain simplified type schemes.

Definition 3.6.4

Let B be an assumption set containing only simplified type schemes. Then we define $\text{TYPE2}(B, e) = e' : \tau, \leq, A$ as in definition 3.2.1, but with the case $e_1 e_2$ as follows :

$$\begin{aligned} \text{TYPE2}(B, e_1, e_2) &= (SQ_1 R(e'_1 : \tau_1)) (SQ_2 R(e'_2 : \tau_2) : \tau') : \tau'', \leq_7, SA \\ &\quad \text{if } \text{TYPE2}(B, e_i) = e'_i : \tau_i, \leq_i, A_i \\ &\quad \text{and } \text{UNION}(\leq_1, \leq_2) = \leq_3, ID \\ &\quad \text{and } \text{AUNIFY}(A_1, A_2, \leq_3) = \leq_4, R, A, Q_1, Q_2 \\ &\quad \text{and } \text{ARROW}(R\tau_1, R\tau_2, \leq_4) = \leq_5, S', a\tau\tau' \\ &\quad \text{and } \text{STANDARDISE}(\leq_5) = \leq_6, S'' \\ &\quad \text{and } \text{SIMPLIFY}(\leq_6, S''S'A \rightarrow S''\tau) = \leq_7, S''' \\ &\quad \text{and } S = S'''S''S' \end{aligned}$$

$$\begin{aligned} \text{and } \tau'' &= S'''S''\tau \\ \text{and } \tau''' &= S'''S''\tau' \end{aligned}$$

Example 3.6.5

Returning to the infinite list of 1's (Example 2.3.4 and 3.2.7), consider :

$$\text{fix } (\lambda x. \text{cons } 1 \ x)$$

Type inference is as for TYPE, with the first difference being a standardisation for the case *cons* 1. At the next step we can apply a simplification so :

$$\begin{aligned} \text{TYPE2 } (B_0, \lambda x. \text{cons } 1 \ x) &= (\lambda x : \text{list } \alpha. \text{cons} [\alpha] (1 : \text{nat} : \alpha) \\ &\quad (x : \text{list } \alpha : \text{list } \alpha) \\ &\quad : \text{list } \alpha) : \text{list } \alpha \rightarrow \text{list } \alpha, \\ &\quad \{\Pi, [\text{nat} \leq \alpha], [\alpha \leq \text{atom}]\}, \{\} \end{aligned}$$

where α is a basic type variable. This is intuitively the 'best' type for this subexpression. After applying *fix*, we obtain the type *list nat*, which is the 'best' type for the infinite list of 1's :

$$\begin{aligned} \text{TYPE2 } (B_0, \text{fix } (\lambda x. \text{cons } 1 \ x)) &= \text{fix } [\text{list nat}] \\ &\quad (((\lambda x : \text{list nat}. \text{cons } [\text{nat}] \\ &\quad \quad (1 : \text{nat} : \text{nat}) \\ &\quad \quad (x : \text{list nat} : \text{list nat}) \\ &\quad : \text{list nat}) : \text{list nat} \rightarrow \text{list nat}) \\ &\quad : \text{list nat} \rightarrow \text{list nat}) : \text{list nat}, \\ &\quad \Pi, \{\} \end{aligned}$$

Note that we obtain three redundant coercions in the result, which could be removed by post-processing.

Example 3.6.6

Figure 15 shows the application of TYPE2 to the factorial function of Figure 4, using the assumption set B_0 , with *mult* and *dec* having the same type schemes as *plus* and *neg* respectively. At each stage we have a subtype ordering containing at most one type variable, whereas the algorithm TYPE produced intermediate subtype orderings with up to eleven type variables. The benefits of simplifications are thus obvious. After the following propositions, we will give some more complex examples.

Figure 15 - The application of TYPE2 to the factorial function

Proposition 3.6.7

If B contains only simplified type schemes, and $\text{TYPE2}(B, e) = e' : \tau, \leq, A$, then \leq is simplified with respect to $A \rightarrow \tau$, and hence if $A = \{\}, \text{gen}(\{\}, \leq, \tau)$ is a simplified type scheme.

Proof: By induction on the structure of e . \square

This proposition justifies the extension of the **let** case in **TYPE** to **TYPE2**. The next proposition shows soundness.

Proposition 3.6.8 : Soundness

If B contains only simplified type schemes, and $\text{TYPE2}(B, e) = e' : \tau, \leq, A$ then :

$$A \cup B, \leq, e \vdash e' : \tau$$

Proof: Since by definitions 3.3.2 and 3.5.1, **STANDARDISE** and **SIMPLIFY** correspond to uses of the derived inference rule **SUBEN**. \square

Proposition 3.6.9

If B contains only simplified type schemes, and $\text{TYPE}(B, e) = e' : \tau, \leq, A$, then $\text{TYPE2}(B, e) = e'' : S'\tau, \le'', S'SA$ where \le', S is a standardisation of \leq and \le'', S' is a simplification of \le' with respect to $SA \rightarrow S\tau$.

Proof: By induction on the structure of e , using propositions 3.1.2 and 3.1.5 for the case e_1e_2 . \square

Remark 3.6.10

We note that the standardisations and simplifications referred to in proposition 3.6.9 are not necessarily regular, since applying regular standardisations and simplifications at each step may produce a small subtype ordering \le'' in cases where \leq is too complex for our rules to apply directly.

Corollary 3.6.11 : Completeness

If B contains only simplified type schemes, and $B, \Pi, e \vdash e' : \tau$ then :

- $\text{TYPE2}(B, e) = e'' : \tau', \leq, \{\}$
- $\text{SUBEN}(S, \leq, L) = \Pi$ for some S, L
- $S\tau' \ll \tau$

and hence $B, \Pi, e \vdash S(e'' : \tau) : \tau$. In particular, if $B = B_0$ then $S(e'' : \tau) : \tau$ is a program, and if the conjecture made in remark 2.8.7 holds, it has the same meaning (or value) as $e' : \tau$.

Proof: Using Corollary 3.2.6. \square

Remark 3.6.12

The above corollary formalizes our notion of ‘best’ type introduced in section 2.3, and shows that TYPE2 indeed produces the ‘best’ type for an expression. The algorithm TYPE2 also has the advantage that intermediate subtype orderings are kept small. Figure 16 shows the behaviour of TYPE2 on some other examples. Since the expressions involved are complex, we also show the equivalent MIRANDA definitions, some of which are taken from Bird and Wadler (1988). The following is our most complex example :

Example 3.6.13

Figure 17a shows a complex numeric type hierarchy containing integers, reals and complex numbers, with non zero and non negative types distinguished. This allows us to give a type for division which excludes division by zero. The types for division, addition, multiplication and increment are shown in Figure 17b. We then consider the MIRANDA function :

$$\begin{aligned}
 e\ x &= f\ x\ 2\ x\ 1 \\
 \text{where } f\ x\ n\ t\ s &= \text{if } t + s = s \text{ then } s \\
 &\quad \text{else } f\ x\ (\text{inc } n)(x \times t)/n)(t + s)
 \end{aligned}$$

which calculates (approximations to) e^x . For the corresponding function in our language, shown in Figure 17c, we derive the type :

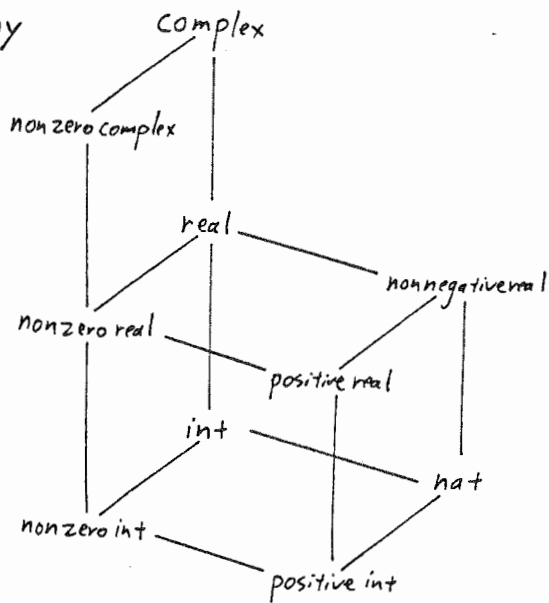
$$\alpha \rightarrow \alpha \text{ where } \text{non negative real} \leq \alpha \leq \text{complex}$$

In other words, the obvious definition extends automatically to the complex case.

The largest simplified type ordering is (as usual) just before the application of *fix*, and contains 6 type variables. The largest subtype ordering before simplification contains 9 type variables.

With the given type for e^x , we can deduce that e^0 and e^π have type *non negative real*, e^{-1} has type *real*, and $e^{i\pi}$ has type *complex*. This is the best that we can reasonably expect from a type inference algorithm.

17a. A numeric type hierarchy



17b. Type schemes for arithmetic operators

$$\text{div: } \forall \alpha \beta : \left[\begin{array}{c} \text{non zero complex} \xrightarrow{\quad} \text{complex} \\ \alpha \xrightarrow{\quad} \beta \\ \text{positive real} \xrightarrow{\quad} \text{non negative real} \end{array} \right] . \beta \rightarrow \alpha \rightarrow \beta$$

$$\text{plus: } \forall \alpha : \text{nat} \leq \alpha \leq \text{complex} . \alpha \rightarrow \alpha \rightarrow \alpha$$

$$\text{mult: } \forall \alpha : \text{positive int} \leq \alpha \leq \text{complex} . \alpha \rightarrow \alpha \rightarrow \alpha$$

$$\text{inc: } \forall . \text{nat} \rightarrow \text{positive int}$$

17c. The exponential function

$$\lambda x. \left(\left(\lambda g. g \times 2 \times 1 \right) \right. \\ \left. \left(\text{fix } \left(\lambda f. \lambda x. \lambda n. \lambda t. \lambda s. \text{cond } (\text{eq } (\text{plus } t \ s) \ s) \ s \right. \right. \right. \\ \left. \left. \left(f \times (\text{inc } n) (\text{div } (\text{mult } x \ t) \ n) (\text{plus } t \ s) \right) \right) \right) \right)$$

Figure 17- A complex example: the exponential function

Remark 3.6.14

In the absence of an actual implementation of our type inference algorithm, we have only type-checked a number of example programs. Furthermore, we have looked at realistic functions, rather than pathological examples. However, we note the following facts about the examples we have tested :

- The final type is intuitively the ‘best’ type.
- For functions without generic arithmetic operators, our final type agrees with that produced by the usual Hindley-Milner algorithm, and our final subtype ordering is Π .
- In all cases tested, the final subtype ordering contains at most *one* (non-isolated) type variable, which is the type parameter for the generic arithmetic operators used.
- The simplification rules MOVEUP and MOVEDOWN are not useful at any stage, supporting the comments we made in Remark 3.5.6.
- The largest intermediate simplified subtype ordering is small (at most 7 type variables for our examples), but approximately proportional to the size of the function.
- The largest subtype ordering before simplification contains approximately twice as many type variables as the largest simplified subtype ordering. The exceptions (21 type variables for *select* and 10 for *sort*) occur when applying two expressions which share three or more variables. Such an application usually occurs only once in a function.

In the next section we examine the efficiency of our algorithm in more detail.

3.7 Efficiency Considerations

In our analysis, we assume that in a ‘sensible’ function definition or expression, types are bounded in size. This is supported by the fact that, although doubly exponential in theory, functional language type inference is efficient in practice (Kanellakis and Mitchell, 1989). Examining the definitions of the algorithms TYPE and TYPE2, we see that the subtype ordering must then grow by a bounded amount at each application and at second and subsequently uses of a program variable. The largest intermediate subtype ordering will then be linear in the length of the expression. For the algorithm TYPE2, this is supported by the experiments reported in section 3.6. The difference between the two algorithms appears to be a constant factor, although a large one.

We have already shown that our operations on subtype orderings (UNION, APPLYSUBST, ENRICH, SUBEN, AUNIFY, ARROW, STANDARDISE and SIMPLIFY) are $O(n^3)$ in the size of subtype orderings. Since these operations are applied a linear number of times, we conclude that the algorithms TYPE and TYPE2 are $O(n^4)$ in the size of expressions. This bound will hold for TYPE2 even in the presence of improved transitive closure algorithms such as that of Simon (1986), which have better than $O(n^3)$ time in the average case, since STANDARDISE and SIMPLIFY are also limited by other factors.

We must now ask whether an $O(n^4)$ type inference algorithm is indeed feasible. We assert that it is, since 'sensible' functional language programs consist of many function definitions of bounded size. Specifically, a typical program in our notation will have the form :

$$\text{let } x_1 = e_1 \text{ in } \dots \text{let } x_m = e_m \text{ in } e$$

where e_1, \dots, e_n, e have bounded size. For such a program, type inference can be done in $O(m)$ time.

In a language such as Miranda, it is possible that a user program will have the form :

$$\begin{array}{l} f \quad \bar{x} = e \\ \text{where } g_1 \quad \bar{y}_1 = e_1 \\ \dots \quad g_n \quad \bar{y}_n = e_n \end{array}$$

Such a program would normally translate to a large expression not involving **let** (since **where** in Miranda is not polymorphic). However, most of the g_i will not refer to each other, and we can extract them as function definitions of the form :

$$g_i f \quad \bar{x} \quad \bar{y}_i = e_i$$

and replace uses of g_i in e by $g_i f \quad \bar{x}$. This process, which is essentially a special case of *lambda-lifting* (Peyton Jones, 1987, Chapter 13), will translate Miranda programs into a series of nested **let** clauses. In fact, for most cases we need not make f a parameter to g_i . Usually one or two program variables from \bar{x} will suffice.

In conclusion then, our improved type inference algorithm TYPE2, although better than TYPE by a large constant factor, is still $O(n^4)$. However, we expect it to run in linear time for typical user programs after appropriate translation.

4 DATABASE APPLICATIONS

We have already seen, in Chapter 3, how our subtype inference scheme has benefits in defining polymorphic functions on a hierarchy of numeric types, such as the exponential function. We now turn to the second major application of our scheme. Recall from the Introduction that our motivation for introducing a hierarchy of type constructors of rank $*$ was to permit different kinds of functions. In particular, we suggested differentiating between finite database functions, which have inverses, and other functions which do not. In this chapter we show that we can give a type scheme for the *inverse* operation, which correctly handles the inverse of both many-to-one and one-to-one database functions. This allows use to do type inference on database queries which involve inverses of both primitive and derived database functions.

We believe that the most natural model for database systems is the Functional Data Model (Shipman 1981, Atkinson and Kulkarni 1984, Buneman et al 1982). This model views a database as a set of entity types, some of which may be subtypes of others, and a set of functions between them. This is simple and conceptually clean, and is closely related to the concept of an algebraic data type (Goguen et al 1978) which consists of a set of *sorts*, and a family of *operations* on those sorts. There are also close links with the related notion of a category (Herrlich and Strecker, 1979). Categories in turn are closely related to typed lambda-calculi and functional programming languages (Lambek and Scott 1986, Kazmierczak 1989). This provides a theoretical bridge between the Functional Data Model and functional programming languages, which is supported by practical work such as that of Buneman et al (1982). However, these authors, while describing the benefits of functional programming and lazy evaluation, fall well short of using the full power of a lazy functional language. They also restrict themselves to the user-unfriendly syntax of FP (Backus 1978).

In this chapter we combine a MirandaTM-like language (which translates to the simpler language of chapters 1-13 as described in the Introduction) with the Functional Data Model. Our subtyping scheme permits us to make the following extensions to the model :

- Database functions can return arbitrary types, including other functions, as results.
- Inverses are defined on derived as well as primitive database functions.

- Higher-order functions can be defined for manipulating database functions, rather than relying on a fixed set of predefined higher-order functions as in Buneman et al (1982).
- Implicit subtype inference is possible for all expressions. In particular, subtype relationships allow entity types to 'inherit' the functions defined on all their subtypes.

The result is a more powerful model, which is also cleaner in that the sharp distinction between data and programs is completely removed.

A comparison between the Functional Data Model and other modelling techniques is given in Hull and King (1987). One model worth special consideration is the object-oriented model (Albano et al 1985), since much recent work on inheritance, such as Stansifer (1988) has been done within that framework. However, we feel that the object-oriented model is too ad-hoc, for two reasons. First, the representation of an object as a tuple of attributes is not as nice conceptually as treating an object as an abstract entity. It also leads to the problem that objects are treated as equal whenever their attributes are the same. This leads to counter-intuitive situations where two objects are considered different as members of one entity type, but are considered identical as members of some supertype. Second, the object-oriented model inherently relies on assignment to fields of the tuples which represent objects as the fundamental update mechanism. In contrast, update in the Functional Data Model can be considered as replacing one function by another function, ie as a redefinition. This is neater, and means that the Functional Data Model shares the benefits of avoiding assignment discussed extensively elsewhere (Turner 1979, Henson 1987).

The first part of this chapter briefly discusses a subtype hierarchy suitable for database use, and gives type schemes for the *inverse* and *composition* operations. We then examine *inheritance* of functions from supertypes to subtypes. The remainder of the chapter is concerned with a substantial case study, namely a medical records database for a hospital. We develop a powerful but readable database manipulation language, and describe it by giving examples taken from the case study.

4.1 A Subtype Hierarchy for Database Use

For use in functional databases, we introduce some new types, as follows :

- *debit credit* : a signed number with two decimal places, used for accounting applications. The type coerces to *string* in the usual way, prefixing a dollar sign, and suffixing minus sign for negative amounts.
- *money* : a subtype of *debit credit* restricted to non-negative values.
- *datetime* : specifies the date and time of an event, coercing to a string of the form '*dd/mm/yyyy hh : mm*'.
- *date* : specifies a date only. There is a function *dateof* which extracts the date from a *datetime*. A date coerces to a string of the form '*dd/mm/yyyy*'.
- *time* : specifies a time only. There is a function *timeof* of type $\text{datetime} \rightarrow \text{time}$, and a function *event* : $\text{date} \rightarrow \text{time} \rightarrow \text{datetime}$ which specifies an event by combining a date and a time. A time coerces to a string of the form '*hh : mm*'.
- *checks* : used for checking conditions and returning error messages. The possible values are *ok* and *error s*, where *s* is a string. These values coerce to string as 'OK' and 'ERROR' & *s* respectively, where & is a binary operator which concatenates two strings and inserts a space between them. The type *bool* also coerces to *checks*, with *true* and *false* coercing to *ok* and *error ''*, respectively.

We also introduce a new type constructor of rank 1, which we call *opterror*. The values are *present x*, for *x* a value of the parameter type, and *missing s*, for *s* a string. This type constructor extends *optional* by supplying a message which indicates why a value is missing, and we thus coerce *absent* to *missing ''*.

Finally, we introduce two type constructors of rank *, as follows :

- database functions (\Rightarrow) : these form the types of database functions which are, or can be, represented by a table. Viewed another way, they are one-to-one or many-to-one database relations.
- injective database functions ($\approx\Rightarrow$) : these form subtypes of database functions which are injective, ie one-to-one.

These two kinds of functions are subtypes of functions in general. We thus obtain the subtype ordering shown in Figure 18. (Note that the type constructor \supset of rank * has been omitted). Figure 19 gives some useful function definitions using this subtype ordering.

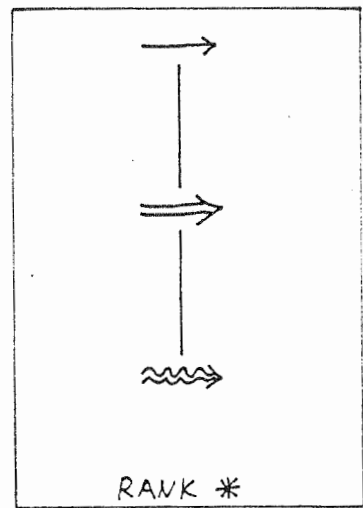
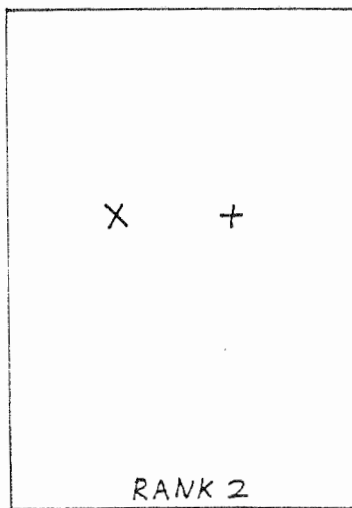
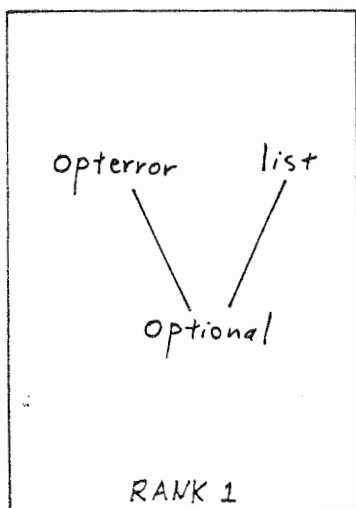
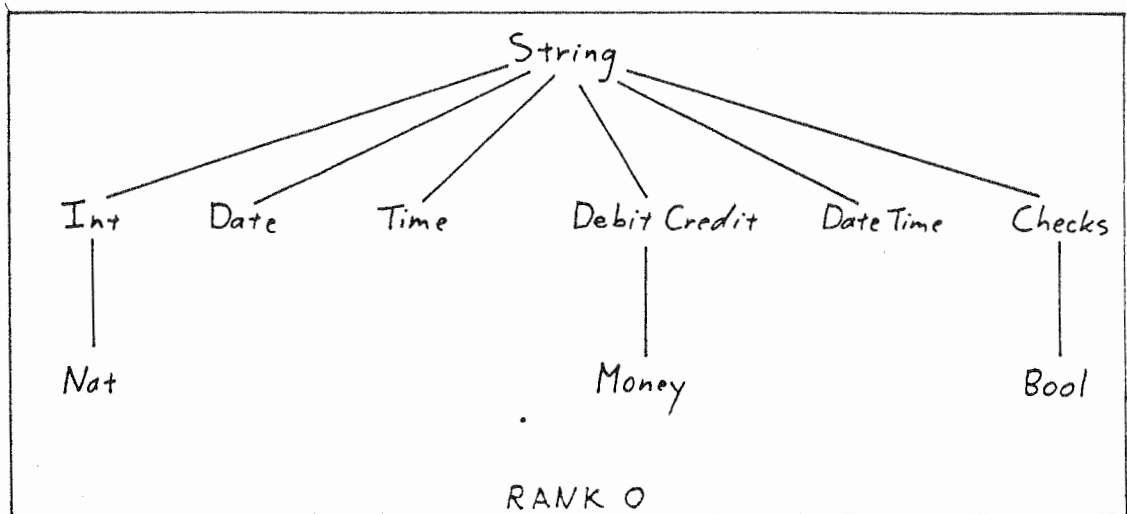


Figure 18 - A subtype ordering for database use

Library USEFUL

Encrypt: String \rightarrow String
DateOf: DateTime \rightarrow Date
TimeOf: DateTime \rightarrow Time
Days Between: Date \rightarrow Date \rightarrow Int
Years Between: Date \rightarrow Date \rightarrow Int
Less DateTime: DateTime \rightarrow DateTime \rightarrow Bool

} unspecified

Next After: $d \rightarrow \text{list } d \rightarrow \text{optional } d$

Next After $x \ [] = \text{absent}$

Next After $x \ [y] = \text{absent}$

Next After $x \ [y:z:zs] = \text{if } x = y \text{ then present } z$
else Next After $x \ [z:zs]$

SortWith: $(d \rightarrow d \rightarrow \text{Bool}) \rightarrow \text{list } d \rightarrow \text{list } d$

SortWith $ls \ [] = []$

SortWith $ls \ [x:xs] = \text{SortWith } ls \ [y \mid y \leftarrow xs; ls \ x \ y]$
 $++ [x]$
 $++ \text{SortWith } ls \ [y \mid y \leftarrow xs; \text{not}(ls \ x \ y)]$

ifmissing: $\text{opterror } d \rightarrow \text{string} \rightarrow \text{opterror } d$

infix $x \ \text{ifmissing } y = \text{case } x \ \text{of}$

present x : present x

missing s : missing y

Figure 19- A Library of useful functions

We must also note that, since database functions are those implementable by means of table lookup, the domain type must have an equality operator defined on it. Therefore in $\tau \Rightarrow \tau'$ or $\tau \approx \tau'$, τ must be a basic type. In order for the injectivity condition to make sense, τ' must also be a basic type, in the second case. However, in the first case we can return functions as results, and the function f such that $f \text{ true} = \text{neg}$ and $f \text{ false} = \text{square}$ could be represented as a table of type $\text{bool} \Rightarrow (\text{int} \rightarrow \text{int})$.

4.2 Inverses of Database Functions

We are finally in a position to give a type scheme for an inverse operation, which inverts only database functions. The inverse of an ordinary database function is multi-valued, and we indicate this using lists :

$$\text{inverse} : (\tau \Rightarrow \tau') \rightarrow (\tau' \Rightarrow \text{list } \tau)$$

A more purist approach would use sets, where *set* was a supertype of *list*, and the coercion from *list* to *set* removed duplicate elements. Such an approach could be handled using our subtype scheme. However, due to the non-injectivity of the *list* to *set* coercion, we could not use the polymorphic equality operator on lists. Instead, we would need additional equality operators *listeq* and *seteq*. We would also need an explicit conversion function from sets to lists, so that the results of *inverse* could be processed with the usual list-handling operations. In the interests of simplicity, we choose to have *inverse* return a list, where the order of elements in the list is dependent on the order in which items were added to the database.

The inverse of an injective database function is optional-valued (by definition of injectivity), so that :

$$\text{inverse} : (\tau \approx \tau') \rightarrow (\tau' \Rightarrow \text{optional } \tau)$$

We also observe that, since the domain of a database function must be basic, τ and τ' must be basic types in both cases of *inverse*. Before we can give a type scheme for *inverse* which generalises the two types we have given, we need the following relation between functional type variables and type variables of rank 1 :

Definition 4.2.1

We say that a type variable θ of rank 1, and a type variable ϕ of rank * are *analogous*, written $\theta \approx \phi$, if :

- for all subtype orderings \leq , $optional \leq \theta \leq list$ and $\approx \leq \phi \leq \Rightarrow$.
- for all substitutions S , $S\theta = optional$ if and only if $S\phi = \approx$, $S\theta = list$ if and only if $S\phi = \Rightarrow$, and $S\theta = \theta'$ if and only if $S\phi = \phi'$ and $\theta' \approx \phi'$.

This relationship can be implemented by merging the rank 1 and rank * subtype hierarchies, using context to determine whether any type variable is to be treated as having rank 1 or rank *. The relationship $\theta \approx \phi$ then means simply that θ and ϕ represent a single type variable ψ , which satisfies the constraint $optional \leq \psi \leq list$. Figure 20 shows the merged subtype hierarchy.

Definition 4.2.2

We give the operation *inverse* the following type scheme :

$$\forall \alpha \beta \theta \phi : \theta \approx \phi . \phi(\alpha, \beta) \rightarrow (\beta \Rightarrow \theta(\alpha))$$

where α and β are basic type variables. We easily verify that ground instances of this type scheme are of the two forms given above.

4.3 Composition of Database Functions

Composition of functions is a little more difficult. We have the usual type for composition :

$$(\tau' \rightarrow \tau'') \rightarrow (\tau \rightarrow \tau') \rightarrow (\tau \rightarrow \tau'')$$

Also, since we can apply a function to the second column of the table representing a database function, we have the type :

$$(\tau' \rightarrow \tau'') \rightarrow (\tau \Rightarrow \tau') \rightarrow (\tau \Rightarrow \tau'')$$

However, to produce an injective database function as a result, we need to compose two injective functions, ie

$$(\tau \approx \tau') \rightarrow (\tau \approx \tau') \rightarrow (\tau \approx \tau')$$

We cannot find a type scheme which has instances only of the three given forms, so we are forced to define two composition operators as follows :

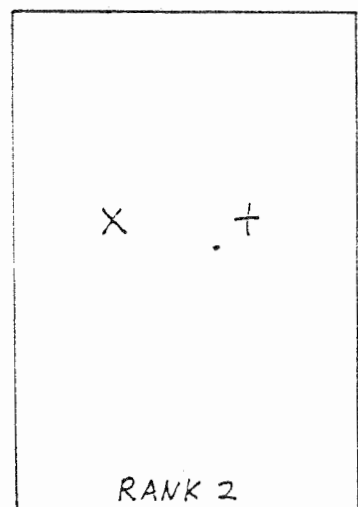
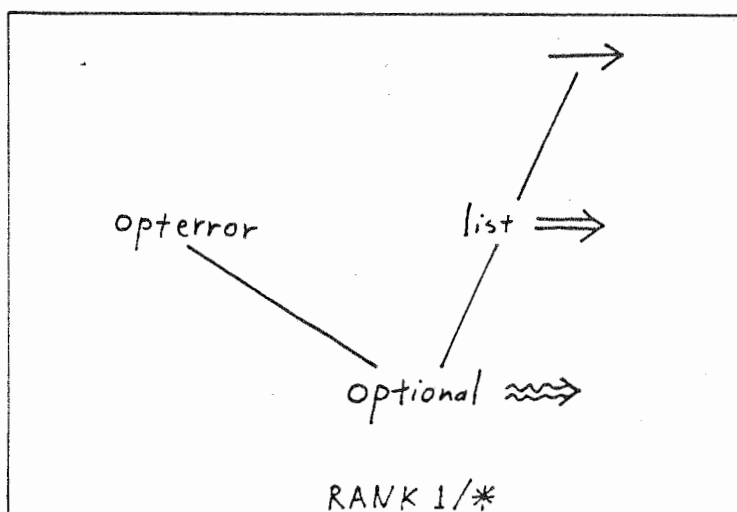
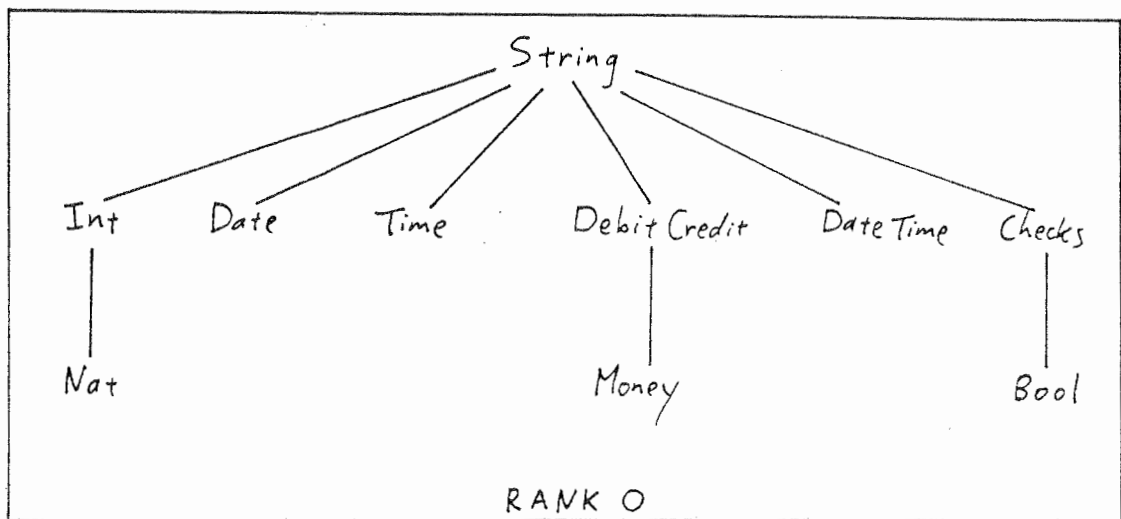


Figure 20 - The merged subtype ordering

Definition 4.3.1

We give the *composition* operator, denoted by the usual symbol, the following type scheme :

$$\forall \alpha \beta \gamma \phi : \Rightarrow \leq \phi \leq \rightarrow. (\beta \rightarrow \gamma) \rightarrow \phi(\alpha, \beta) \rightarrow \phi(\alpha, \gamma)$$

We give the injective composition operator, written *incomp*, the type scheme :

$$\forall \alpha \beta \gamma. (\beta \approx \gamma) \rightarrow (\alpha \approx \beta) \rightarrow (\alpha \approx \gamma)$$

where α, β and γ are basic type variables.

We can easily verify that together these composition operators describe the three possibilities listed above. The first type scheme does not specify that if ϕ is \Rightarrow then α must be basic but it need not do so, since all primitive functions of type $\tau \Rightarrow \tau'$ will have τ basic, and by the type scheme for *inverse*, we cannot construct any expressions of type $\tau \Rightarrow \tau'$ for non-basic τ .

The division of composition into two operators is a notational inconvenience, but we note that the *incomp* operator is, in fact, rarely used. The *incomp* operator is only applicable when there are two injective primitive database functions, where the argument type of one function is the same as the result type of the other. For example, in the substantial case study which we shall describe later in this chapter, the *incomp* operator is never applicable.

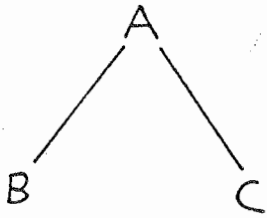
One further problem is that constructing derived database functions by composition alone is too restrictive. We solve this by noting that any function on an entity type can be turned into a database function by enumerating all the elements in the entity and applying the function to each in turn. We therefore assume that for each entity type E , there is an operator *for each* E with type scheme :

$$\forall \alpha. (E \rightarrow \alpha) \rightarrow (E \Rightarrow \alpha)$$

This permits us to construct inverses of any function defined on an entity type (although such inverses would probably not be extremely efficient).

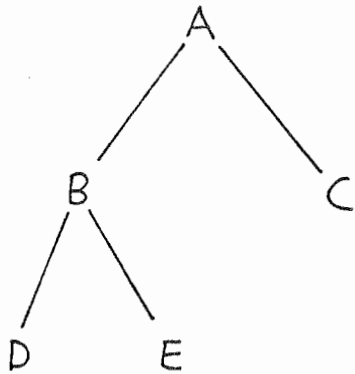
4.4 Entities and Multiple Inheritance

In the Functional Data Model, we assume that there are a number of *entity* types, which may have subtype relationships between them. We assume that hierarchies of entity types are semilattices with a top element (and also satisfying the other conditions required of a subtype ordering). This is not a



$$A = B + C$$

B, C inherit operations on A

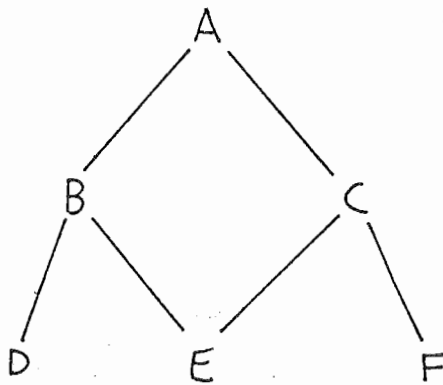


$$A = B + C = D + E + C$$

$$B = D + E$$

B, C inherit operations on A

D, E inherit operations on A, B



$$A = B + F = D + C = D + E + F$$

$$B = D + E$$

$$C = E + F$$

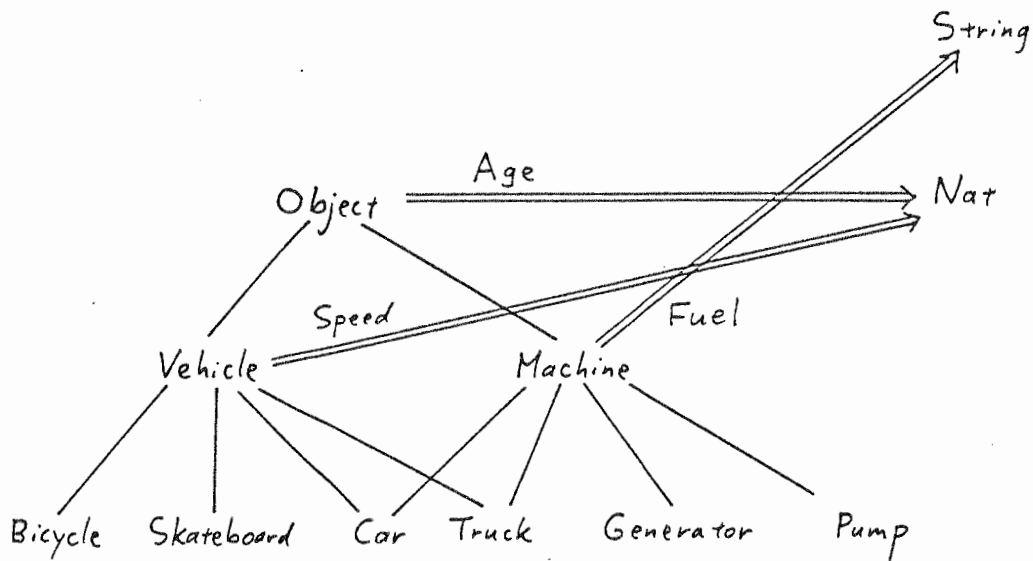
B, C inherit operations on A

D inherits operations on A, B

E inherits operations on A, B, C

F inherits operations on A, C

Figure 21- Some hierarchies of entity types



$\text{mycar} \in \text{Car}$

$\text{Fuel}(\text{mycar}) = \text{"gasoline"}$

$\text{Speed}(\text{mycar}) = 100$

$\text{Age}(\text{mycar}) = 23$

$\text{yourtruck} \in \text{Truck}$

$\text{Fuel}(\text{yourtruck}) = \text{"diesel"}$

$\text{Speed}(\text{yourtruck}) = 90$

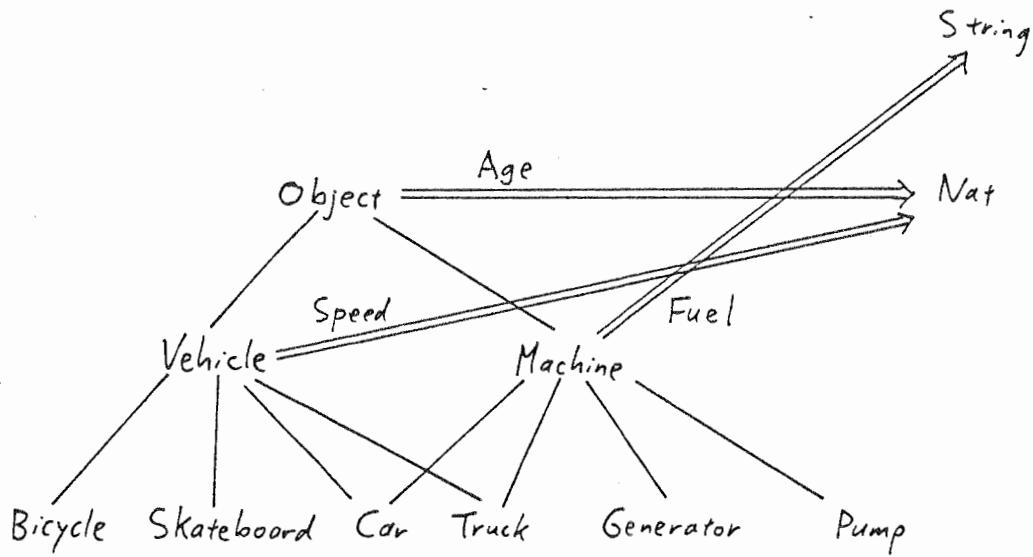
$\text{Age}(\text{yourtruck}) = 4$

$\text{hisbike} \in \text{Bicycle}$

$\text{Speed}(\text{hisbike}) = 60$

$\text{Age}(\text{hisbike}) = 1$

Figure 22 - An example of multiple inheritance



Driving Force: $\text{Object} \rightarrow \text{String}$

Driving Force $x = \text{case } x \text{ of}$

Bicycle b : "sweat"

Skateboard s : "gravity"

Machine m : Fuel m

Description: $\text{Object} \rightarrow \text{String}$

Description $x = \text{case } x \text{ of}$

Bicycle b : (if Age $b > 5$ then "old" else "new") & "bicycle"

Skateboard s : "skateboard"

Car c : (if Speed $c > 100$ then "fast" else "slow") & "car"

Truck t : (Fuel t) & "truck"

Generator g : "generator"

Pump p : "pump"

Figure 23 - Case analysis on entity types

restrictive condition, as we can always adjoin a top element to a hierarchy. Note that the hierarchy need not be a tree. We assume that an entity type is the union of its subtypes, and contains no other elements. This again is not restrictive, since if we desire A to be a subtype of B , we can add C as a subtype of B , where C consists of those members of B not in A . Lastly we assume that two entity types are disjoint unless they have a common subtype, which can be adjoined if necessary.

Figure 21 shows some simple example hierarchies of entity types. We note that, because of subtype coercions, entities *inherit* the operations on all their supertypes. Since the entity hierarchy is not necessarily a tree, this is what is known as *multiple inheritance* in the world of object-oriented languages. However, unlike object-oriented systems, we assume that the coercions are *injective*. For example, in Figure 22, we have two distinct items *mycar* and *yourtruck* which have the same attribute *age*. Considered as members of the entity type *object*, they cannot be distinguished by their attributes, but we are justified in saying $mycar \neq yourtruck$ (if you will walk outside onto the street, you will see that they are distinctly different objects). The same argument applies to distinct cars. On the other hand, since objects are not represented by tuples of attributes, non-injective coercions are not needed with the Functional Data Model. We also note that if there is an injective function from the top of an entity hierarchy to a primitive type such as *nat* or *string*, then we can use it as an implicit coercion. This is useful when there is only one sensible way of identifying an element of an entity type by a printable value. Our case study will show some examples of an entity type being coerced to *string* for printout.

Finally we turn to case analysis on entity types. We assume that the *case* expression extends to entity types, by providing a list of disjoint subtypes. Figure 23 shows some examples, using the hierarchy of Figure 22. We also assume there is a predicate (eg *iscar*) to test membership of each subtype, and an *optional*-valued function (eg *forcecar* : *object* \rightarrow *optional car*) which returns an object viewed as being a member of a particular subtype, if that is possible.

4.5 Case Study — A Hospital Database

As a case study, we consider a medical records database for a hospital. This is a complex example, even though we have simplified it by omitting several database functions. The complexity of the example makes it difficult to specify using ‘fourth generation’ techniques (Burroughs, 1984).

A hospital is physically divided into a number of places in which computer terminals may or may not be located. Some of these places are *wards* which house patients. Organisationally, a hospital is divided into a number of *units* (which include outpatient clinics) which take responsibility for patients during their visit to hospital. People in the hospital fall into three overlapping groups : *patients*, *staff* and *babies*. Some staff are *doctors* who attend patients, and some staff have passwords which allow access to important database update transactions. Babies may be patients, or may be with a mother in hospital.

Registered patients makes zero or more *visits* to hospital where they fall under the responsibility of a particular doctor and unit. Some visits are clinic or outpatient visits, while others involve an actual *stay* in hospital. For each stay we record a list of comments (dated and initialled by a staff member), and either the ward the patient is currently located in, or the date and time they were discharged from hospital. Comments may be made on a stay after the patients leaves hospital, as paperwork is processed. The last visit for a patient may be either an in-hospital stay or a planned visit with a future admission date. All preceeding visits must be clinic vists or discharged stays.

Places, units, and terminals are uniquely identified by a name which is a string, and these entities can be coerced to *string* for output purposes. However, persons have two kinds of unique key — a *unit record number* for patients, and a string of initials for staff. There may also be a *medicare number* which is not necessarily unique.

Visits are uniquely identified by a patient and a date/time. We also note that only some unit/doctor and unit/ward combinations are possible. To facilitate multiple operations on a patient, we would also like to (optionally) associate a patient with a logged-on terminal. Figure 24 shows the entity subtype hierarchy for our hospital database, and Figure 25 adds the primitive database functions. In Figure 26 we give some simple queries in our functional languages. The types are given for documentation purposes but note that all

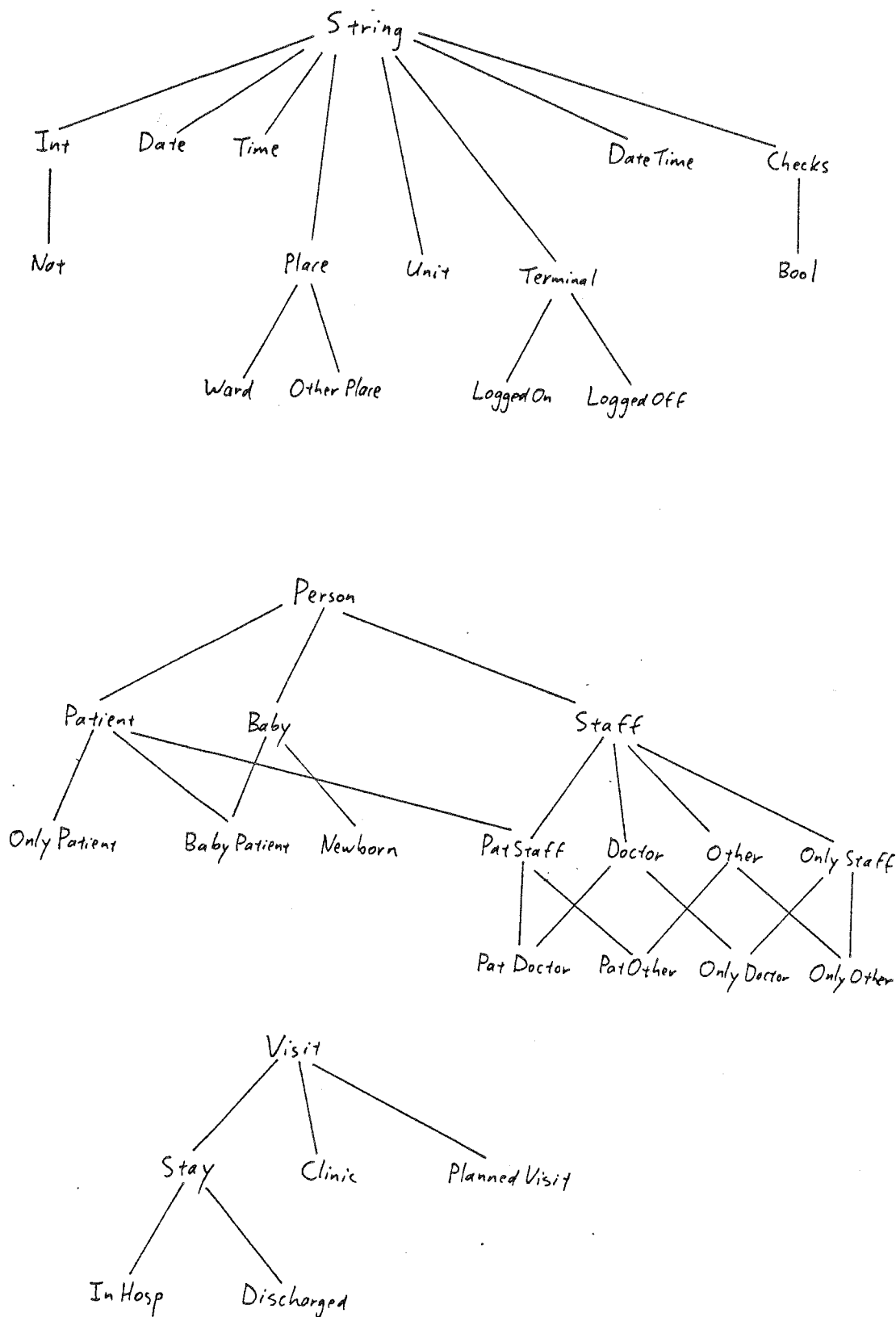


Figure 24 - A subtype hierarchy for a hospital

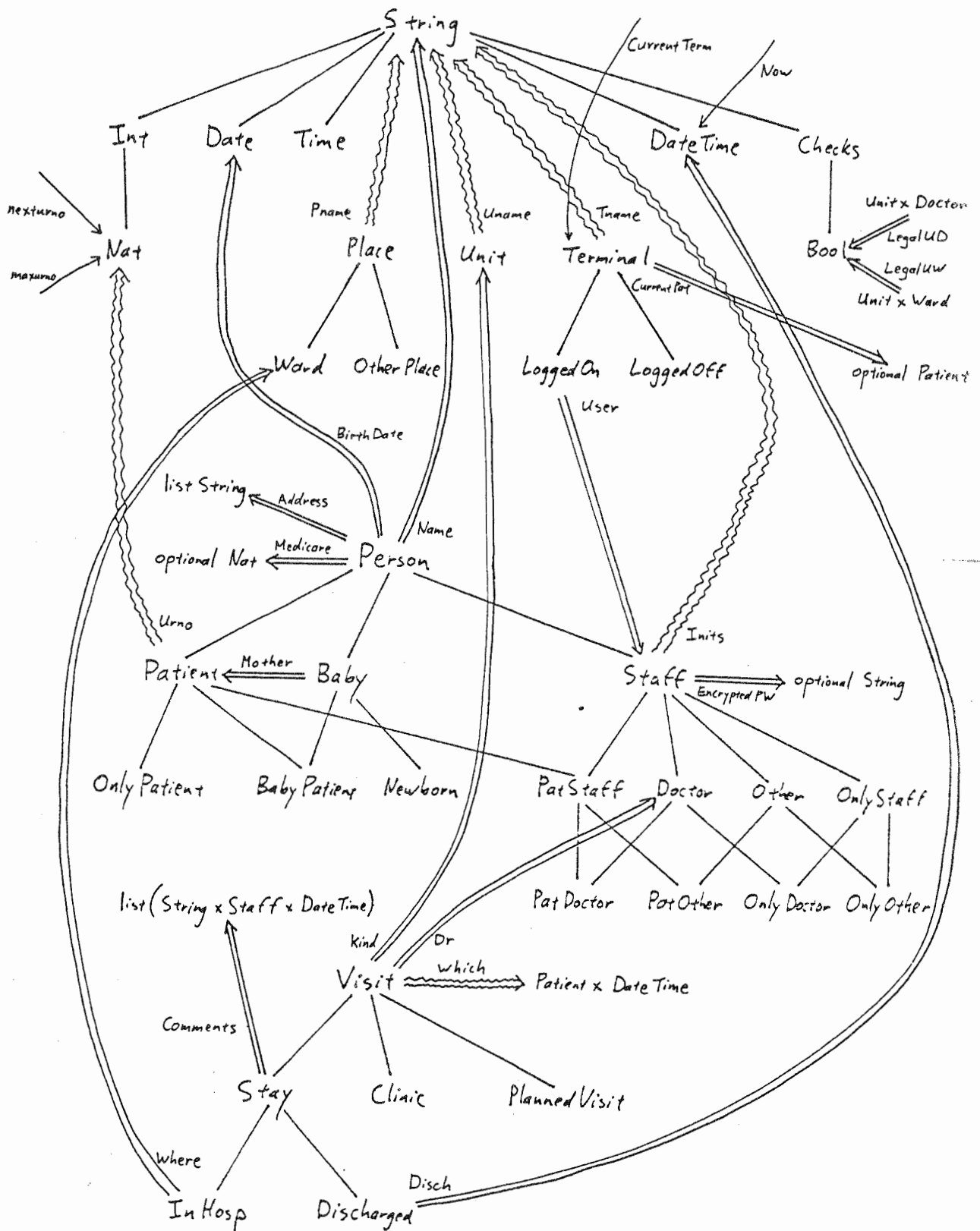


Figure 25 - Primitive database functions on the hierarchy

Library QUERIES 1

Admit: Visit \Rightarrow Date Time

Admit = snd \circ Which

Who: Visit \Rightarrow Patient

Who = fst \circ Which

Visits By: Patient \Rightarrow list Visit

Visits By = inverse Who

Age: Person \rightarrow Int

Age p = Years Between (DateOf Now) (Birth Date p)

After: Visit \rightarrow Visit \rightarrow Bool

After x y = Less Date Time (Admit y) (Admit x)

Desc Date Sort: List Visit \rightarrow List Visit

Desc Date Sort = SortWith After

Last Visit: Patient \Rightarrow optional Visit

Last Visit = head \circ Desc Date Sort \circ Visits By

Visit Before: Visit \rightarrow Optional Visit

Visit Before v = Next After v (Desc Date Sort (Visits By (Who v)))

Who Is In: Word \rightarrow list Patient

Who Is In w = [Who s | s \leftarrow inverse Where w]

Pats With Name: String \rightarrow list Patient

Pats With Name s = concat [for Patient p | p \leftarrow inverse Name s]

Num Babies: Patient \Rightarrow Nat

Num Babies = len \circ inverse Mother

Figure 26 - Some simple queries

these types can be inferred using our type inference algorithm, after translating to the simpler language of sections 1-3.

The types of the primitive database functions do not impose all the restrictions we require, and so we impose a number of *constraints* which are checked on each update. These are also expressed using our functional language as *checks*-valued functions which return either *ok* or an informative error message. Note that because of the subtype relationship, *boolean*-valued functions are also acceptable. Figure 27 shows the constraints for our hospital example.

4.6 Query in Our Hospital Database

We assume that most user queries would take place using *forms*. A user would select a form by hitting an appropriate key on the terminal, and optionally giving a numeric or string parameter. Figures 28B and 28C show two query form definitions, using some general purpose definitions in Figure 28A. The first query lists patients in a particular ward (by default the one in which the terminal originating the query is located), and the second lists all patients in hospital being attended by a given doctor. Each form definition consists of a form layout, a check for the existence of the specified ward or doctor (using an *opterror*-valued expression), possibly some definitions, and expressions to evaluate. The results of the query are 'filled in' in boxes on the form, which may be scrolling windows. We assume that fields in the result may be selected (by mouse or cursor movements) to use as parameters to other query forms. Figure 28D shows an example dialogue with the system.

4.7 Update in Our Hospital Database

To update our database, we extend our form definitions with update operations. The intention is that after an optional query is performed, the user fills in some boxes on the form. If a given sequence of checks is satisfied, and the updates do not violate the database constraints, then the old database is replaced by the updated database, and a confirmation message is written to the terminal. The following update operations are possible :

addnew <entity type name> with $f_i = e_i$

Constraints

forall Person p : if Age $p \leq 150$ then ok else error "Maximum age 150 years"

forall Person p : if len (Address p) ≤ 4 then ok else error "Max 4 lines of address"

forall Person p : case Medicare p of

absent : ok

present n : if checkdigit [9,7,3,1,9,7,3,1] ($n \text{ div } 10$) = $n \text{ mod } 10$
then ok else error "illegal medicare number"

where checkdigit [] $n = n$

checkdigit [$x:xs$] $n = (\text{checkdigit } xs \text{ (} n \text{ div } 10) + x * n) \text{ mod } 10$

ensure if nexturno \leq maxurno then ok else error "urno allocation overflow"

forall Visit v : if LegalUD $< \text{kind } v, \text{Dr } v >$ then ok

else error "illegal unit/doctor combination"

forall Visit v : case Visit Before v of

absent : ok

present b : if nonoverlap $b \vee$ then ok

else error "overlapping visits"

where nonoverlap $x \ y = \text{case } x \text{ of}$

InHosp s : false

Discharged d : LessDateTime (Disch d) (Admit y)

Clinic c : LessDateTime (Admit c) (Admit y)

PlannedVisit p : false

forall Stay s : case s of

InHosp h : if LegalUW $< \text{kind } h, \text{where } h >$ then ok

else error "illegal unit/ward combination"

Discharged d : if LessDateTime (Admit d) (Disch d) then ok

else error "Discharge must be after Admit"

forall Stay s : if DescFrom (Admits) (Comments s) then ok

else error "comments must be in descending order after Admit"

where DescFrom d [] = true

DescFrom d [$< c, s, d2 > : \text{rest}$] = if LessDateTime $d \ d2$ then false

else . DescFrom $d2 \text{ rest}$

Figure 27 - Database constraints

Library USEFUL1

GetWord : String \rightarrow opterror Word

GetWord s = case inverse Pname s of
absent : missing "no place"&s
present p : force Word p ifmissing "no word"&s

GetDoctor : String \rightarrow opterror Doctor

GetDoctor s = case inverse Inits s of
absent : missing "unknown initials"&s
present st : force Doctor st ifmissing s&"is not a doctor"

StayInfo : InHosp \rightarrow String

StayInfo s = Urno(Whos) & Name(Whos) & Kind s &
(case Comments s of [] : ""
[<<c,s,d>>:rest] : c)

Figure 28 A - Useful functions for query

Form `WARDQUERY(WRD:String)` -- list patients in a ward

Patient List for Ward

WOUT: String Out, MAIN: Window

Query exists $w =$ if $wRD = ""$
 then for word (Location CurrentTerm) if missing "no default word"
 else GetWord wRD

return $w_{OUT} = w$

return MAIN = [stay into s | s ← inverse where w]

Figure 28B - An example query form

Form DOCTOR QUERY (INITS : String) -- list patients for a doctor

Patient List for <input type="text"/>

NME : String Out , MAIN : Window

Query exists d = Get Doctor INITS

return NME = Name d

def combine [] = []

combine [<w, [] > : rest] = combine rest

combine [<w, [x : y] > : rest] =

["patients in word" & w : x : y] ++ combine rest

def stays w = [stayinfo s | s ← inverse where w ; Or s = d]

return MAIN = combine [<w, stays w > | w ← all Word]

Figure 28C - A second query form

After user types AJS

Patient List for <input type="text" value="Dr. Andrew J. Smith"/>				
patients in ward 1A				
192347	Jane Smith	PAED	Under Observation	
209412	Alex Jones	PAED	Recovering after appendisectomy	
patients in ward 7A				
134927	John Smith	GSURG	Operation scheduled for 16:30	
181233	Mary Doe	GMED	Medication as per chart	

After user types 7A or in ward 7A simply

Patient List for Ward <input type="text" value="7A"/>			
105979	Fred Bloggs	GSURG	Due for discharge tomorrow
134927	John Smith	GSURG	Operation scheduled for 16:30
153271	Peter Taylor	GMED	Admitted to ward 7A
181233	Mary Doe	GMED	Medication as per chart
194271	Frederick Smith-Jones	GSURG	Condition Stable
230047	Blanche Dubois	GMED	Medication as per chart
234906	Carol Scott	GSURG	Condition Stable
▽ 12 more entries			

User can now use cursor to scroll through entries, or to select patient Mary Doe as indicated, supplying 181233 as parameter to next form

Figure 28D - An example query dialogue

This operation adds a new element to the given entity type. The f_i must be a complete list of primitive database functions defined on that entity type, and the e_i must be appropriate values for these functions. The update fails if the constraints are violated. If the update succeeds, derived functions (such as *who* and *admit*) are also updated, if the implementation makes this necessary.

update <object> with $f_i = e_i$

This operation updates function values for an existing element of an entity type. Otherwise it behaves like *addnew*, except that any previous primitive database functions not occurring among the f_i are retained.

delete <object>

This deletes an element of an entity type from the database. Unlike the delete operation of Atkinson and Kulkarni (1984), this does not cause a cascade of other deletions. Instead the operation fails if there are references to the object other than by primitive database functions defined on it. This can be detected using reference-counting memory management techniques. If the operation succeeds, the database functions defined on the object are adjusted by deleting the object from the appropriate tables.

reclassify <object> as <entity type name> with $f_i = e_i$

This moves an element of an entity type to another entity type in the same component of the subtype hierarchy. The f_i must be a complete list of functions defined on the new entity type but not on the old one. Any functions defined on both entity types are retained.

We note that the database is only updated if all the updates specified in a given form can succeed. If any update would fail, the appropriate error message is given, and the database is not updated. To make this possible, we assume that all update transactions generated by users filling out forms are *serialised* and processed in a fixed order, rather than in parallel.

Figure 29 shows some simple general-purpose definitions, and a number of form definitions using them. The form definitions specify a query and an update to be performed after the user has filled in the form. Default values for a form are handled by 'filling-in' values with the query, which the user can

Library USEFUL2

Get Staff: String \rightarrow opterror Staff

Get Staff s = inverse Inits s ifmissing "unknown initials"&s

Get Unit: String \rightarrow opterror Unit

Get Unit s = inverse Uname s ifmissing "no unit"&s

Get Patient: Nat \rightarrow opterror Patient

Get Patient n = inverse Urno n ifmissing "no patient with urno"&n

Get Planned Visit: Patient \rightarrow opterror Planned Visit

Get Planned Visit p = case Last Visit p of
absent: missing "no planned visit"
present lv: force Planned Visit lv ifmissing "no planned visit"

Current Stay: Patient \rightarrow opterror In Hosp

Current Stay p = case Last Visit p of
absent: missing "no visits made"
present lv: force In Hosp lv ifmissing "no current stay"

User Logged On: opterror Staff

User Logged On = case Current Term of
Logged On x: present (User x)
Logged Off y: missing "terminal is not logged on"

Legal Pass Wd: Staff \rightarrow String \rightarrow checks

Legal Pass Wd s pw = if Encrypted PW s = present (Encrypt pw) then ok
else error "incorrect password"

Figure 29A - Useful functions for update

Form LOGON (INITS: String) -- gain access to sensitive transactions

Log On to System

Initials	<input type="text"/>	Name	<input type="text"/>
Terminal	<input type="text"/>	Location	<input type="text"/>

Enter Password to Log On

INITD: String Out, NME: String Out, TERM: String Out,
LOC: String Out, PWD: String Secret In

Query exists NewUser = GetStaff INITS
return INITD, NME, TERM, LOC = < INITS, Name NewUser,
 Current Term, Location CurrentTerm >

Update check Legal Pass Wd NewUser PWD
 case Current Term of

Logged On x: update x with User = NewUser,
Current Pat = absent

Logged Off y: reclassify y as Logged On
with User = NewUser, Current Pat = absent

confirm NME & "logged on at" & Now

Figure 29B - An example update form

Form PATINFO (N: Nat) -- patient information query/update

Patient Information Query/Update					
URNO	<input style="width: 90%;" type="text"/>	Name	<input style="width: 90%;" type="text"/>		
Ward	<input style="width: 90%;" type="text"/>	Days in Hospital	<input style="width: 90%;" type="text"/>	Unit	<input style="width: 90%;" type="text"/>
		Doctor	<input style="width: 90%;" type="text"/>		
Comments	<input style="width: 90%;" type="text"/>				

UR: Nat Out, NME: String Out, WRD: String InOut,
 DAYS: Nat Out, UNIT: String Out, DOC: String Out,
 COM: String In, MAIN: Window

Query exists p = if N=0 then CurrentPat CurrentTerm if missing "no default urno"
else GetPatient N

exists s = CurrentStay p

return UR, NME, WRD, UNIT, DOC = <Urno p, Name p, Where s, Kind s, Dr s>

return DAYS = Days Between (DateOf Now) (DateOf (Admit s))

return MAIN = [c&d& Inits St | <c, St, d> ← Comments s]

Update exists User = User Logged On

exists w = GetWard WRD

if w ≠ Where s then update s with Where = w else SKIP

update s with Comments = <COM, User, Now> : Comments s

update Current Term with CurrentPat = present p

confirm "information updated for" & NME

Figure 29C - A second update form

Form ADMIT ($N: \text{Nat}$) -- admit patient with U.R. No. N

Patient Admission				
URNO	<input style="width: 90%;" type="text"/>	Name	<input style="width: 95%;" type="text"/>	
		Address	<input style="width: 95%;" type="text"/>	
			<input style="width: 95%;" type="text"/>	
			<input style="width: 95%;" type="text"/>	
			<input style="width: 95%;" type="text"/>	
Date/Time	<input style="width: 100%;" type="text"/>	Unit	<input style="width: 100%;" type="text"/>	Doctor
			<input style="width: 100%;" type="text"/>	Ward
			<input style="width: 100%;" type="text"/>	<input style="width: 100%;" type="text"/>
Comments	<input style="width: 95%;" type="text"/>			

UR: Nat Out, NME: String Out, ADDR: String[4] InOut,
 ADM: DateTime InOut, UNIT: String InOut, DOC: String InOut,
 WRD: String InOut, COM: String In

Query exists term = force LoggedOn CurrentTerm ifmissing "not logged on"
exists p = if $N=0$ then CurrentPat term ifmissing "no default urno"
else GetPatient N
return UR, NME, ADDR, ADM = $\langle \text{Urno } p, \text{ Name } p, \text{ Address } p, \text{ Now} \rangle$
return WRD = case Location term of Ward $w: w$
 OtherPlace $p: ""$
return UNIT, DOC = case GetPlannedVisit p of missing $s: \langle "", "" \rangle$
 present $v: \langle \text{Unit } v, \text{ Inits(Dr } v) \rangle$

Update exists $u = \text{GetUnit UNIT}$
exists $d = \text{GetDoctor DOC}$
exists $w = \text{GetWard WRD}$
def $c = \text{if COM} = "" \text{ then "admitted to ward" \& } w \text{ else COM}$
update p with Address = ADDR
case GetPlannedVisit p of missing $s: \text{skip}$
 present $pv: \text{delete } pv$
addnew InHosp with which = $\langle p, \text{ADM} \rangle$, kind = u , Dr = d ,
 where = w , comments = $[\langle c, \text{User term}, \text{ADM} \rangle]$
update term with CurrentPat = present p
confirm NME & "admitted to ward" & WRD

Figure 29D - A complex update form

Log On to System

Initials Name

Terminal Location

Enter Password to Log On

Sr. Patricia M. Jones logged on at 20/3/89 11:30

Patient Admission

URNO Name

Address

Date/Time Unit Doctor Ward

Comments

Mary Doe admitted to ward 7A

Patient Information Query/Update

URNO Name

Ward Days in Hospital Unit Doctor

Comments

suspected food poisoning 20/3/89 11:37 PMJ

information updated for Mary Doe

Figure 29E - An example update dialogue

over-write. An example dialog with the system is also shown. Note that the admission form in Figure 29D is by far the most complex transaction encountered in the hospital database, involving default values (eg for a patient that has just been found on a query and has therefore been selected as the current patient), a possible change of patient address, a choice of actions depending on whether the visit was previously planned, and a check for the terminal being logged on. In spite of this the transaction is specified readably and concisely.

4.8 General Remarks

Although we have covered the database applications of our subtype inference scheme only briefly, we can draw a number of conclusions. Firstly, the hierarchy of type constructors of rank $*$ allows us to mix database functions with general functions, while being able to infer which kind of function is denoted by a particular expression. This permits us to treat primitive and derived database functions in the same way. In particular, we can take the inverse of derived database functions. We thus obtain a clean merger of a lazy functional programming language with the Functional Data Model.

We can use the full power of a lazy functional programming language to define higher-order functions, queries, constraint checks and updates. We can use recursion or list comprehensions instead of looping to iterate over entities. This is theoretically neater, and contributes to conciseness and readability. The ability to do type inference means that the programmer need not specify redundant type information. Should this type information be needed for documentation purposes, it can be produced by a compiler. Our case study has shown the usefulness of the language we propose.

The hierarchy on type constructors of rank other than $*$ is also used throughout our case study. In particular, the *optional* to *opterror* coercion is useful. The coercions to *string* are essential for writing readable queries, as they allow us to write expressions of entity types, and obtain printable results. Finally, the coercions between entity types provide multiple inheritance, by making functions on supertypes available to subtypes. We do this within the Functional Data Model, rather than the object-oriented model. However, the Functional Data Model has conceptual advantages over other database models, which result in its clean fit with functional programming languages.

We have not examined the important issues of implementation and program optimization. However, we have shown that our subtyping scheme can lead to a powerful and useful database manipulation language.

5 RELATED WORK

We will now examine the relationship of our work to similar work on subtypes and especially to type inference algorithms for various notions of subtype. We have already referred to the overviews of Leivant (1983), Reynolds (1985) and Cardelli and Wegner (1985) in our Introduction. Reynolds discusses Milner's type inference algorithm, the second order typed lambda calculus, and the theory of subtypes and generic functions. His notion of subtype includes deletion of attributes from a tuple, which models inheritance in object-oriented languages. Reynolds then poses several questions on the relationships between these three approaches to type structure.

In the past five years, there has been a flood of new work in the area of typing which helps to answer the questions which Reynolds raised. For example, Cardelli and Wegner (1985) follow their survey with a combination of subtyping (without generic functions) and the second order typed lambda calculus, resulting in bounded polymorphism. However, they do not have a type inference algorithm. The work of Mitchell (1984), of which Reynolds does not seem to have been aware, combines a limited form of subtype with type inference. Our thesis also aims to answer the questions raised by Reynolds, by extending Mitchell's work. It is therefore appropriate to examine how our work fits into this new area of typing. Because of the recent nature of much work in the area, some work may not be listed here. For example, we have not been able to obtain a copy of Jategaonkar and Mitchell (1988), which appears to be relevant.

In the first section of this chapter we will examine a comprehensive theory of subtypes and coercions by Henson (1985). This provides a framework for describing various approaches to subtype inference. Section 5.2 discusses approaches based on *intersection* and *union* of types. An intersection operator on types leads to a form of inclusion polymorphism, where $\tau \cap \tau'$ is a subtype of both τ and τ' . Similarly, τ and τ' are both subtypes of the union $\tau \cup \tau'$.

In section 5.3 we discuss the second order typed lambda calculus, where we have a notion of subtype based on *generality*. Section 5.4 discusses the notion of subtype favoured by Cardelli and Wegner (1985) which allows deletion of attributes from tuples, while section 5.5 re-examines Mitchell (1984) and

further work derived from it which is very similar to our work. In section 5.6 we discuss other approaches, and in section 5.7 we examine the relationship between our coloured types and other notions of generic functions.

5.1 A Comprehensive Theory of Coercions

The work of Henson (1985) provides a good framework for relating together various approaches to subtype inference. Henson permits types constructed from primitive types, type variables, and an extensive list of operators on types. These include *union*, which he calls *sum*, and *intersection*, which he calls *overloading*. This is motivated by the fact that any expression of type $\tau \cap \tau'$ can be viewed as having both types τ and τ' (using our notation for intersection). The *product* (\times) and *function* (\rightarrow) type constructors which we described in chapter 1 are also included, as is the *universal quantifier* (\forall). The *existential quantifier* (\exists) is also listed. This quantifier is suggested by the close relationship of types and logic (Coquand 1985, Huet 1986). The existential quantifier is also discussed by Cardelli and Wegner (1985). It has debatable utility in modelling type abstraction in the sense of modules or CLU clusters (Mitchell and Plotkin 1985, Liskov et al 1977). For example it deals awkwardly with the definition of a type constructor, such as *stack*, and cannot deal with the simultaneous definition of two types that have an inclusion relation between them. MacQueen (1986) makes some other criticisms of the use of the existential quantifier. Henson also lists a *fixpoint operator* (μ) for constructing recursive types, and the *type abstractions* and *type applications* of the second order typed lambda calculus.

A curious omission in Henson's list of operators is the disjoint union ($+$) operator. However it, like the product operator, can be modelled using type abstraction (Reynolds 1985). MacQueen et al (1984, 1986) present a list of operators similar to that of Henson, but excluding type abstractions and type applications. There is some justification for this, since the discipline of universal type quantification is essentially a notational variant of type abstraction (Leivant 1983). MacQueen et al present a model for their type system as a metric space of *ideals* (downward-closed sets closed under least upper bounds of directed subsets), which makes their work suitable as a semantics for languages such as ML and MirandaTM. They do not deal with the notion of subtype, however.

Henson provides a lattice model for his type system which is abstract in the sense that it applies to any choice of primitive types. The model satisfies a very extensive list of subtype axioms. These include the inclusion relationships generated by intersection and union; subtype relations on primitive types, products, and functions as in our chapter 1; currying and uncurrying of functions; folding and unfolding of recursion; taking an instance of a universally quantified type; forming an existentially quantified type from an instance; and alpha-, beta-, and eta-reduction on type abstractions and applications. The subtype relationship is a preorder, ie it is reflexive and transitive, but not antisymmetric. In other words, there may be distinct inter-convertible types. However, there is an obvious partial order on equivalence classes of types. One unfortunate feature is that the least element of the lattice is not the same as the intersection of all types (ie $\forall \alpha. \alpha$), and the greatest element is not the same as the union of all types (ie $\exists \alpha. \alpha$). As a result, the intersection of all types is not a subtype of all types, which is a somewhat counter-intuitive result.

The axiomatic treatment of subtype relationships by Henson describes most forms of subtyping, possibly with some minor modifications. One notable exception is the attribute-deletion coercion used to describe inheritance in object-oriented languages. If Henson's scheme is extended by a set of type constructors which behave like the product operator in that they are monotonic, and a set of type constructors of rank $*$ which share the monotonicity/antimonotonicity properties of functions; then we can embed our notion of subtype in that of Henson, by extending the primitive coercions to type constructors as well as primitive types. Henson's model should therefore provide a semantics for our typed expressions.

We will now turn to forms of inclusion polymorphism which are generated by intersection and union operators on types.

5.2 Inclusion Polymorphism with Intersection and Union

One extensively studied case of Inclusion Polymorphism is the Intersection Type Discipline for the pure lambda calculus, most recently formulated in Ronchi Della Rocca (1988). This involves a universal type, written ω , and the function and intersection operators on types. It has axioms for intersections and functions equivalent to those of Henson. In particular, intersection is a

greatest lower bound operator, and $(\tau \rightarrow \tau') \cap (\tau \rightarrow \tau'')$ and $\tau \rightarrow (\tau' \cap \tau'')$ are inter-convertible. However, it adds axioms making ω and $\omega \rightarrow \omega$ inter-convertible, and making every type a subtype of ω . Since ω corresponds to the type $\exists \alpha. \alpha$ in Henson's system (ie the union of all types), the behaviour of ω in the Intersection Type Discipline goes beyond Henson's framework. Indeed, the equivalence between ω and $\omega \rightarrow \omega$ is only justified by the fact that in the pure lambda calculus, every term is a function.

A significant feature of Ronchi Della Rocca's work is the definition of an *expansion* operation (unrelated to what we have called expansions in chapter 1) and a *lifting* operation. These allow a notion of principal (most general) type to be defined. She gives a semi-algorithm for unification which leads to a type inference semi-algorithm. Since the inference algorithm can be viewed as a reduction machine, using an innermost reduction strategy, the algorithm only terminates for strongly normalising terms. However, if the algorithm terminates, it returns a principal type. This behaviour is the best that can be expected, since the problem is semi-decidable. A semi-algorithm is also given for type inference in the Intersection Type Discipline without ω (which falls within Henson's framework). This system is of great theoretical interest, since precisely the strongly normalising terms have types. However, due to the problem of semi-decidability, it is of limited practical value.

Although type inference in the full Intersection Type Discipline is semi-decidable, there are restrictions of the system where type inference is decidable. Leivant (1983) shows that for the Intersection Type Discipline without ω , where intersection symbols cannot occur to the left of more than one arrow, type inference is decidable. Leivant provides a type inference algorithm for this system, based on his algorithm V. A corollary to his result is that the type inference algorithm applies to the system formed by allowing unrestricted intersection, but only first-order functions.

An interesting case of inclusion polymorphism is presented by Mishra and Reddy (1985). This uses the intersection, union, and fixpoint operators of Henson, with products extended to arbitrary tuples, and functions restricted to be first-order. The restriction on functions avoids the semi-decidability problem of the Intersection Type Discipline. Mishra and Reddy also permit an arbitrary set of *constructors*. If e is a term of type τ , and a is a constructor, then $a(e)$ has type $a(\tau)$. This allows us to use algebraic data types without type declarations for the constructors. For example, the type *list* α corresponds to

$\mu\beta. nil \cup cons [\alpha, \beta]$. This system allows us to define many interesting subtypes of lists. These include lists of length zero (ie only *nil*), lists of length at most one, lists of length precisely two, lists of even length, and (if Mishra and Reddy's semantics is replaced with a lazy one along the lines of MacQueen et al (1984, 1986)), non-finite lists. In contrast, the only subtypes of lists involving inclusion polymorphism that can be defined in our system are the type consisting only of *nil*, and the type of non-finite lists. The other cases of subtyping can only be handled in our system by declaring new constructors and specifying a coercion. For example, our *optional* type corresponds to lists of length at most one.

In Mishra and Reddy's system, functions are defined by case analysis, and if cases are omitted, the system can infer to which subtype the function is applicable. For example, if we have a lazy semantics, and write :

$$double (cons [a, x]) = cons [a, cons [a, double (x)]]$$

we can infer that *double* is defined as non-finite lists only. Another benefit is that constructors are overloaded on an arbitrary number of types, and that arbitrary overloading of functions is possible using the intersection operator. This solves the problem of assigning types to arithmetic operators. However there are two disadvantages. The first is that type declarations are still necessary to apply stronger type restrictions. For example, with the usual definition of *append* on lists, we have $append(cons [1, nil], 2) = cons [1, 2]$. Most programmers would not wish the *append* function to apply to non-lists. The second disadvantage is that, due to the properties of the Intersection Type Discipline, Mishra and Reddy's system does not extend to higher-order functional languages.

Mishra and Reddy provide a type inference algorithm for their system, although it may not be complete. The algorithm operates by generating inequalities on types from function definitions, and computing upper and lower bounds for type variables. Essentially argument types are maximised, and result types are minimised, as in our simplifications. However, there appears to be little relation between our inference algorithm and that of Mishra and Reddy.

A type system involving arbitrarily overloaded constructors generalising that of Mishra and Reddy is described in Holyer (1988). Holyer represents (possibly recursive) types defined by constructors by (possibly cyclic) directed graphs, and provides algorithms to calculate unions, intersections, and differences of types. One limitation is that the last two algorithms do not apply to types containing infinite values (eg lists with a lazy semantics). Another limitation is that the types of parameters in function definitions cannot be reliably inferred, and hence type declarations are necessary for parameters. However, Holyer's work suggests that graph representations of types could usefully be employed in Mishra and Reddy's inference algorithm.

We have remarked that Mishra and Reddy's type system is unsatisfactory for functional languages, since it is limited to first order functions. However, its properties make it ideal for type inference in the logic programming language Prolog (Clocksin and Mellish 1984). This issue is examined in Mishra (1984). The viewpoint taken is that Prolog has a 'totally defined semantics', ie there are not type errors in the traditional sense. Instead, the type of a predicate describes the terms for which it may succeed. The notion of 'ill-typed' is replaced by 'definitely failing'.

There are other approaches to type systems for Prolog. For example, Mycroft and O'Keefe (1984) provide a type system for the subset of Prolog described by first-order logic. Their system is closely based on Milner's work, but they do not have a type inference algorithm. Dietrich and Hagl (1988) extend this work by allowing inclusion polymorphism. They provide a type inference algorithm, which is subject to some restrictions, and which requires *mode* information for predicates. The algorithm involves calculating upper and lower bounds for type variables, and checking that these are consistent. The authors are engaged in further research on decidability and efficient algorithms for this process. Two recent systems similar to that of Mishra (1984) are given by Kluzniak (1987) and Zobel (1987). The first of these works with a heavily restricted language called Ground Prolog (which is almost a functional programming language). The second of these is a more straightforward extension of Mishra (1984), giving an inference algorithm and a proof that ill-typed programs definitely fail.

In conclusion we may say that inclusion polymorphism obtained using type intersections and unions is too powerful a notion of type to permit type inference in higher-order functional languages. Since we consider

higher-order capabilities essential, this makes this form of inclusion polymorphism inappropriate for the languages we are interested in. However, this form of inclusion polymorphism is of theoretical interest, and is applicable to logic programming languages.

5.3 Subtypes and Generality

Within the second order typed lambda calculus, we can consider a universally quantified type to be a subtype of all its instances, and indeed this is one of the cases considered by Henson. However, we can go beyond this, and consider τ a subtype of $\forall \alpha. \tau$ if α does not occur in τ . This makes the subtype relationship a preorder. Mitchell (1984a) considers this notion of subtype in a study of type inference rules for the second order typed lambda calculus. As we have remarked earlier, no type inference algorithm is known for this calculus, although a number of algorithms that can infer some type information are known. Two such limited algorithms are given by McCracken (1984), although these do not make the notion of subtype explicit.

One limited inference algorithm for an extension of the second order typed lambda calculus is given by Fairbairn (1986). Fairbairn calls his language *Ponder*. It's main additional feature is the ability to define recursive types, equivalent to adding a fixpoint operator on types. This adds sufficient power to define Turing's fixpoint combinator, and hence all partial recursive functions (Fairbairn 1985). Fairbairn provides an extensive list of subtype relationships, including folding and unfolding of recursive types. He gives an interesting, although complex, inference algorithm, which requires types to be explicitly given for all bound variables. There are however, no proofs of either soundness or completeness.

In summary, a notion of subtype is implicit in any attempt at type inference for the second order typed lambda calculus. This calculus removes the sometimes annoying restriction of languages with Milner's type inference algorithm, that polymorphic functions cannot have polymorphic arguments. However, to date this has been at the expense of being able to perform full type inference, a penalty we feel is not worthwhile. Thus this form of subtyping is also inappropriate for the languages we are interested in, and the algorithms in this area are of no assistance in solving the subtype inference problems we have examined.

5.4 Subtypes and Object Oriented Languages

We now turn to the form of subtyping which occurs in Object Oriented Languages. This is motivated by the need to model inheritance of attributes. Our subtyping system has a similar motivation, but we prefer the Functional Data Model, for reasons we have already discussed. However, it is appropriate to examine inference algorithms for the Object Oriented approach at this point.

In the Object Oriented approach, entities are described by tuples such as $\langle \text{fuel} : \text{'diesel'}, \text{speed} : 90 \rangle$. Attribute selectors and attribute updates are applicable to all tuples that contain the appropriate attributes. Wand (1987) provides a type inference algorithm for this system, but without using a notion of subtype. This algorithm is incomplete, and a proposed modification solves the problem at the cost of being exponential in time (Wand 1988).

Stansifer (1988) provides a type inference algorithm which uses the notion of subtype provided by Cardelli and Wegner (1985). This is that a tuple type is a subtype of all tuple types with fewer attributes, so that we essentially have a coercion that deletes attributes. Stansifer's algorithm also deals with a dual notion of subtypes for variants. The algorithm is sound and complete, but since it does not check the consistency of sets of inequalities, it will return a type for expressions which intuitively are ill-typed. Another limitation is that the polymorphic attribute update function which Wand would write as $(\lambda x. x \text{ with fuel} := \text{'gasoline'})$ is not handled. The type inference algorithm is closely related to that of Mitchell (1984), but is simpler. The reason for the simplicity is that if one type is a subtype of another then either the two types are identical, or they are both tuple types, or they are both variant types. Stansifer therefore has inequalities on *row expressions* (which represent the contents of tuple brackets) rather than on types, and can mark inequalities as referring to either tuples or variants. This avoids the need for some of the machinery that we define.

Rémy (1989) also uses a notion of subtype to deal with the Object Oriented approach. However, rather than deleting attributes, he assumes that all attributes are (potentially) present and have an applicability marker. He thus has a coercion which marks attributes as inapplicable. This approach leads to a very simple type inference algorithm, which is sound and complete, and

handles the polymorphic attribute update function. However, the algorithm generates large sets of inequalities, and no algorithms are provided to simplify them, or to check consistency.

One interesting feature of Rémy's work is that it shows how the Object Oriented notion of subtype can be considered as part of our type system. Assume there is a finite set of attribute labels a_1, \dots, a_n . Then tuples can be considered as type constructors of rank n , forming a lattice with least element $\langle a_1 : _ , \dots, a_n : _ \rangle$ and greatest element the empty tuple $\langle \rangle$. We can then give type schemes to attribute selectors and attribute deletion functions, although not to attribute update functions. For example, in the case of two attributes, we have four possible tuples, namely $\langle a : _ , b : _ \rangle$, $\langle a : _ \rangle$, $\langle b : _ \rangle$ and $\langle \rangle$. Selection of the a attribute has the type scheme $\forall \alpha. \langle a : \alpha \rangle \rightarrow \alpha$, while deletion of the a attribute then has type scheme :

$$\forall \alpha \beta \theta \phi : \{ \theta \leq \phi, \langle b : _ \rangle \leq \phi \}. \theta(\alpha, \beta) \rightarrow \phi(\alpha, \beta)$$

The type scheme $\forall \alpha \beta \theta : \{ \theta \leq \langle a : _ \rangle \}. \theta(\alpha, \beta) \rightarrow \alpha \rightarrow \theta(\alpha, \beta)$ could be assigned to the function which updates the attribute a , but this will apply only to tuples where the attribute a is already present, and does not have Wand's intended meaning of adding the attribute if it was absent.

In conclusion, we have found that Stansifer's work is in theory a special case of ours, although the large type hierarchy required would probably make our algorithm very inefficient, unless tuples were considered as a special case. More promising is that our notions of standardisation and simplification may also be applicable to the algorithms of Stansifer and Rémy. In summary, although our motivation was to exclude the Object Oriented approach, our notion of subtype is powerful enough to encompass this form of subtyping also.

5.5 The Mitchell Approach

We have introduced our work as an extension of that of Mitchell (1984). Although our basic operations on subtype orderings differ somewhat from his, we have essentially followed his approach. We have permitted a polymorphic **let** construct, limited to top-level polymorphism, as in the functional language MirandaTM. We have allowed coercions to be non-injective, as long as they commute with polymorphic functions. We have extended type coercions to

type constructors, which permits us to distinguish different kinds of functions, and to model inheritance in both the Functional Data Model and the Object Oriented approach. Finally, we have ensured that subtype orderings are always consistent.

At this point, it is appropriate to consider whether we have succeeded in making our type inference algorithm more efficient than that of Mitchell. Comparison of Mitchell's algorithm with ours indicates that our naive algorithm TYPE is less efficient than that of Mitchell, since it produces subtype orderings which may be double in size. However, our use of simplifications ensures that the improved algorithm TYPE2 produces subtype orderings which are never larger than those produced by Mitchell's algorithm. All three algorithms are quartic-time, assuming types are bounded in size. However, our algorithm TYPE2 produces much smaller subtype orderings in practice, with the final subtype ordering containing only one type variable. For a typical user program consisting of many small polymorphic function definitions, our algorithm TYPE2 can process each definition independently, thus running in linear time.

We have not been alone in considering Mitchell's work the most promising basis for work on subtype inference. The work of Fuh and Mishra (1988, 1989), developed independently from our work, also takes this direction. We have taken the approach of defining a subtype ordering to be a partial order on type variables and type constructors, satisfying a number of conditions which guarantee consistency. We have then defined basic operations on subtype orderings, and built our way upwards to an efficient type inference algorithm. In contrast, Fuh and Mishra (1988) begin by expressing Mitchell's type inference algorithm in its most general form, such that any preorder on types is permitted. Recall that we remarked in our Introduction that such an extension was possible. Fuh and Mishra express their general algorithm using the general *functional letters* of Leivant (1983). This has the advantage of requiring only one type inference rule in place of rules for lambda abstraction, application, if-then-else, etc. Interestingly, their algorithm is in the style of Milner's algorithm *W*, rather than the parallel style of Leivant's algorithm *V*, for which the functional-letter notation was designed. Because of its generality, their algorithm encompasses not only our type system, but all the notions of subtype inference discussed in this chapter. However, this generality is at the cost of efficiency. Furthermore, the large set of inequalities produced may not be consistent.

Fuh and Mishra follow their general type inference algorithm by considering the case of subtyping examined by Mitchell, namely subtype relations generated by coercions on atomic types. They do not make our extension to coercions on type constructors, although making this extension to their work would not be difficult. On the other hand, they permit the subtype relationship to be a preorder, ie there may be distinct types which are inter-convertible. This may be useful, although it is difficult to find examples where it is essential. One example that could be suggested is polar and rectangular versions of complex numbers, but this would be of dubious utility.

Since Fuh and Mishra permit any preorder of primitive coercions, not just our forest of semilattices, sets of inequalities are not necessarily consistent. Their notion of consistency is essentially the same as our notion of satisfiability, and they provide algorithms for checking the consistency of a set of inequalities. Unfortunately, correctness of a key algorithm (CONSISTENT) is not proved. Checking the consistency of a set of inequalities can be done in quadratic time, assuming that set intersection can be done in constant time. This assumption is dubious in the general case, but like us, Fuh and Mishra (1989) provide algorithms for reducing the size of sets of inequalities. Some realistic upper bound can thus be placed on the size of sets, and constant-time bit-string set implementations can be used. One disadvantage of Fuh and Mishra's algorithm is that consistency checking is performed once only, at the end of type inference. Errors are thus detected, but not localised. However, there is no reason why consistency checking should not be done repeatedly, together with the algorithms for reducing the size of sets of inequalities.

The first work of Fuh and Mishra (1988) permits sets of inequalities on arbitrary, not just atomic types. This has the advantage that polymorphic type inference with an unrestricted **let** construct is easy. This is because, given a substitution S such that $S\beta = \gamma \rightarrow \eta$, we can apply the substitution to a type scheme $\forall \alpha : \{\alpha \leq \beta\}. \beta \rightarrow \alpha$ without problems. The result will be $\forall \alpha : \{\alpha \leq \gamma \rightarrow \eta\}. (\gamma \rightarrow \eta) \rightarrow \alpha$. However, in their later work (Fuh and Mishra 1989) they lose this advantage by restricting themselves to sets of inequalities on atomic types only. We will follow Mitchell (1984) in referring to such sets of inequalities on atomic types as *coercion sets*. They do not discuss polymorphism in this restricted system, but we would suggest that their algorithm would require the same restrictions on **let** that we have proposed.

The work of Fuh and Mishra (1989) converges closely to our work, although with a more concise terminology. A notion of ‘lazy’ instance on type assertions is given, which corresponds to uses of our derived inference rules SUBEN, REPLACE, ASSUMP and LESS. As a result, Fuh and Mishra’s completeness theorem is simpler to state and prove than our Theorem 3.2.5. An interesting additional result is a notion of ‘minimal’ typing and a very inefficient algorithm for computing such minimal typings. However, the ‘minimal’ type is not necessarily the same as our ‘best’ type. For example, consider the following assertion :

$$\{x : \alpha\}, [\alpha \leq \beta], \text{pair } x \text{ } x \vdash \text{pair } (x : \alpha) (x : \alpha : \beta) : \alpha \times \beta$$

This appears to be ‘minimal’ in Fuh and Mishra’s terminology, but clearly the ‘best’ type would involve equating α and β . (We remark as an aside that Fuh and Mishra’s type inference rules only infer types, not typed expressions, but the extension to typed expressions would be trivial).

Fuh and Mishra’s algorithm for calculating ‘minimal’ types is not intended for practical use, although they prove it gives a result unique up to renaming. They do, however, provide two practical techniques for reducing the size of coercion sets, which they call G -subsumption and S -subsumption. The technique of G -subsumption applies to type variables which do not occur in an assumption set or type, and is exactly equivalent to using our simplification rule MOVEUP together with the rule EQDOWN (or MOVEDOWN with EQUP). The efficiency criticism we make for MOVEUP and MOVEDOWN is countered by observing that bit-string set operations can be used, making an algorithm for performing G -subsumptions cubic-time, rather than quartic-time. The technique of S -subsumption is exactly equivalent to using our simplification rules EQUP and EQDOWN. Because G -subsumptions use MOVEUP and MOVEDOWN in a very restricted way, combining G -subsumptions and S -subsumption permits a unique result. Furthermore, this result can be obtained by performing all the G -subsumptions first, and all the S -subsumptions afterwards. We note that if bit-string set operations are used, our algorithm SIMPLIFY can be considerably simplified, since it would not need to maintain a transitively reduced graph.

Fuh and Mishra do not have our concept of *standardisation*, and this may reduce the effectiveness of S -subsumption in complex examples. By a converse argument to our Remark 3.5.6, we could argue that the presence of

G -subsumptions reduces the need for standardisations. However, if we consider our exponential function (Example 3.16.13), there are no opportunities for G -subsumption, and a lack of standardisations would increase the size of the largest intermediate (simplified) coercion set from 9(6) to 12(9). Clearly what is required is an approach combining standardisations with simplifications which include both G -subsumptions and S -subsumptions.

In summary, by unifying our approach with that of Fuh and Mishra, we can construct a better extension of Mitchell's work. We would choose Fuh and Mishra's elegant notation for instances of typing assertions, their use of bit-string set operations, their algorithm for checking consistency of a coercion set (provided a correctness proof can be found), and their notion of G -subsumption. We would choose our extension of subtypes to type constructors, our treatment of polymorphism, our use of coloured types, our notions of standardisation and simplification, and our overall style of inference algorithm, which permits parallelism. If a correctness proof cannot be found for Fuh and Mishra's consistency-checking algorithm, we would retain our restrictions on primitive coercion sets, which make consistency-checking trivial. We believe that such a unified approach would combine the best features of both approaches.

5.6 Other Approaches

One interesting case of subtyping we have briefly referred to in our Introduction is that of Thatte (1988). This work uses the universal type of the Intersection Type Discipline (written in uppercase), and subtype relationships generated by the inequality $\Omega \leq \tau$. The motivation for this is to deal with heterogeneous data structures, such as the list $[1, 2, \text{true}]$ which has type *list* Ω . This is later combined with a notion of statically generated dynamic type checks which allow operations to be performed on the elements of heterogeneous data structures. Thatte gives an inference algorithm, which is an instance of the general algorithm given by Fuh and Mishra (1988), together with a semi-algorithm for checking the consistency of sets of inequalities on types. Thus it seems that even this restriction of the Intersection Type Discipline suffers from the problem of semi-decidability.

On a different note, the issue of subtype inference in a first order functional language is treated by Collier (1987). Collier suggests that his work can be

viewed as a form of overloading, which makes it a special case both of our work and that of Mishra and Reddy (1985). The language used is FP (Backus 1978, Dekker 1983), and the example subtype hierarchy is a special case of ours, with *int* and *bool* being subtypes of *atom*. The algorithm relies on flagging type variables with the atomic types to which they can be instantiated. The algorithm reduces the size of the set of inequalities produced, although the technique for doing so is complex and inefficient. There are no proofs of soundness, completeness, or that the algorithm is polynomial-time. However, earlier versions of this work helped motivate our search for an efficient subtype inference algorithm.

5.7 Generic Functions, Type Classes and Coloured Types

Although bounded polymorphism can assign type schemes to many functions, there are some functions to which a single type scheme cannot be assigned. One such example is the addition of types *date* and *time* to our example type hierarchy, with the intention that addition is defined on *nat*, *int*, *date* and *time*. This problem can be solved in our system by the use of coloured types, defining a new colour, say orange, which includes the types *nat*, *int*, *date* and *time*, as well as orange type variables. All types coloured orange will also be basic types (ie coloured red). However, this solution is at the cost of abandoning the coercion between *int* and *atom*. A second problem is the exponentiation function in a type hierarchy involving *nat*, *int* and *real*. This has the type scheme $\forall \alpha. \alpha \leq \text{real}. \alpha \rightarrow \text{nat} \rightarrow \alpha$ and also the type $\text{real} \rightarrow \text{real} \rightarrow \text{real}$. A similar problem was experienced with the composition function in section 4.3, and the only solution in our system is to define two functions with different names. A third case which cannot be solved with bounded polymorphism is the type of the equality function, which we have solved using coloured (basic) types. In this section we will examine two other approaches to assigning types to these three cases of functions.

In his treatment of implicit coercions and generic functions, Reynolds (1980) describes ‘category-sorted algebras’. These consist of a category of types and a family of function symbols, where the types of the function symbols are given by a functor from tuples of types to types. The commutativity of functions with coercions is expressed by the condition that the interpretation of function symbols as functions be a natural transformation. Although this approach is first-order, the very general framework suggests that an extension to

higher-order functions is possible, using cartesian closed categories (Lambek and Scott 1986). However, the extension to hierarchies of type constructors may be more difficult. There is also no type inference algorithm. On the positive side, the framework appears to be sufficiently general to deal with all the functions that can be typed using bounded polymorphism (at least on primitive types), as well as the first two problems we raised in the previous paragraph.

Another approach to assigning types to difficult functions is the *type classes* of Wadler and Blott (1989). This solves the first and third of our three problems, although it is not clear if a type can be given to exponentiation. Type classes are a way of grouping types, and appear to be very similar to our colours. Wadler and Blott present a very elegant type inference algorithm and translation method for resolving the overloading inherent in type classes. This permits a value to be assigned to a function such as :

$$\text{double } x = \text{mult } 2 \ x$$

In contrast, in our scheme the ‘code’ for *double* will depend on how it is later used, since the appropriate typed expression contains a polymorphic use of *mult*, and a coercion :

$$(\lambda x : \alpha. \text{mult } [\alpha] (2 : \text{nat} : \alpha) (x : \alpha) : \alpha) : \alpha \rightarrow \alpha$$

The natural notion of ‘code’ for this function would be to take the actual value of *mult* and the actual value of the coercion as parameters to be supplied where *double* is used. The technique of Wadler and Blott may provide a more efficient alternative, if it can be adapted to our system.

Wadler and Blott also permit definitions of new type classes by the programmer, which is an issue that we have not addressed. However, the cost of their approach is that all type coercions must be explicit, so that we must write, for example *pi + (float 3)*. This return to the notational inconvenience of Fortran is too high a price, in our opinion. We have shown that implicit coercions and grouping of types can be combined, using our notion of coloured types, with only minor restrictions on allowable coercions. Such a combined approach seems preferable to that of Wadler and Blott.

6 CONCLUSIONS AND FURTHER WORK

In this thesis, our main objectives have been to extend and improve the subtype inference algorithm of Mitchell (1984), in order to create a more efficient and powerful system, and to apply this to an integrated functional programming and database manipulation language. We will now examine to what extent our objectives have been attained, and to look at areas which would benefit from further extension and improvement.

Chapter one introduced our basic notation and operations. We extended the coercion sets of Mitchell (1984) to include inequalities on type constructors, but restricted primitive (variable-free) coercion sets to be a forest of semilattices. This restriction allowed easily checkable conditions for consistency of a set of inequalities, which we formalised by *satisfiability*. It also made possible *standardisations*, which can reduce the size of a subtype ordering, and create a more regular structure which assists the *simplification* procedure in further reducing size. The SATISFY algorithm, which provides a kind of abstract compilation by finding a substitution satisfying a subtype ordering, also relies on the restriction. However, the restriction is sufficiently weak that most useful subtype hierarchies do not seem to be excluded. We introduced the term *subtype ordering* for the partial order generated by a consistent coercion set.

We then defined a series of operations for manipulating subtype orderings, and gave correctness proofs for them which exploited the concept of *minimality*. The concept of *splitting* provided a link between subtype orderings and their representation as transitively reduced acyclic graphs, and our operations on subtype orderings can thus be implemented by graph operations. The alternative approach of Fuh and Mishra (1988, 1989) may have led to somewhat simpler operations. This approach uses constant-time set operations to calculate transitive closures in linear time, to avoid the need for using the transitive reduct, and to check consistency in quadratic time (by an algorithm unfortunately not proved correct). Constant-time set operations are possible if some upper bound can be set on the size of subtype orderings, and the use of standardisations and simplifications, together with a programming style using many small function definitions, appears to make this possible. A combination of our work with that of Fuh and Mishra (1988, 1989) is one promising avenue for further research.

By using quantification, we permitted a large class of useful generic functions. The notion of *coloured* types allowed other generic and overloaded functions to be given a type scheme. The equality operator was one such function. By ensuring that all type schemes be closed, we avoided the problem of applying a substitution to a type scheme. This was at the cost of restricting polymorphism to the top level, but experience with Miranda TM has shown this suffices for most applications. Apart from being essential for programming in functional languages, polymorphism has the added benefit that it prevents the subtype orderings inferred for functions from being merged. This encapsulation helps reduce the size of subtype orderings.

In chapter two, we defined an example functional language, and gave a set of type inference rules with type-annotate expressions. The VAR, AP, ABS and LESS rules were essentially those of Mitchell (1984), but we added PVAR and LESS to handle polymorphism, and ABS2 to introduce a second kind of function. We then derived some useful additional inference rules. In particular, combining the derived inference rules SUBEN, REPLACE, and ASSUMP with the inference rule LESS was equivalent to the notion of ‘lazy instance’ in Fuh and Mishra (1989).

We gave a semantics for typed expressions, rather than the usual approach of giving a semantics for untyped expressions. This was because type inference in the presence of implicit coercions can best be viewed as reconstructing the terms of a typed language, by inserting omitted type information and calls to coercion functions, rather than as selecting a subset of terms from an untyped language which do not ‘go wrong’. Thus rather than proving semantic soundness of type inference rules in the usual way, we showed that our semantics was Church-Rosser and preserved well-typing. We also showed that coercions were injective on equality types, gave a unique result, and commuted with primitive functions. We conjectured that this was sufficient to guarantee that different type-annotations of an expression were equivalent, if they had the same over-all type. Since expressions have principal types (defined by our type inference algorithm), this conjecture would make it possible to speak of *the* meaning of an untyped expression.

In chapter three we gave a naive type inference algorithm, and proved it sound and complete. We then defined standardisations and simplifications. We gave four simplification rules (EQUP, EQDOWN, MOVEUP and MOVEDOWN) of which we only used the first two, in order to ensure a unique result. The G -subsumptions of Fuh and Mishra (1989), which are essentially combinations of EQUP and MOVEDOWN, could also have been included as simplification rules, while retaining a unique result. The use of standardisations and simplifications allowed the definition of a more efficient type inference algorithm which was still sound and complete. The algorithm (like that of Fuh and Mishra (1989)) was quartic time, assuming types of bounded size. This assumption is justified by the good performance of Milner's type inference algorithm in practice. For typical user programs involving many small polymorphic function definitions, the algorithm would be linear. If necessary, local function definitions could be 'lifted' to the top level to achieve this. The algorithm produces a final type which is intuitively the 'best' type, and a final subtype ordering which usually contains at most one type variable.

Chapter four applied our work to the database arena, using subtypes to describe inheritance in the Functional Data Model. We defined a hierarchy of three kinds of functions, namely injective database functions, general database functions, and general functions. The notion of analogous type variables made possible a type scheme for the inverse operation, which was essential in writing database queries. No unique type scheme could be given for composition, and so two composition operators were defined. However, the second of these was never used in our case study. This case study involved a hospital database, and our language proved to be sufficiently powerful for writing a selection of queries and update transactions.

In summary, we have attained our main objectives. We combined subtype inference, including subtype relationships on type constructors, with a limited but sufficient form of polymorphism. This allowed us to mix different kinds of numbers in arithmetic expressions, and to define a powerful and useful database manipulation language. The introduction of standardisations and simplifications to reduce the size of subtype orderings ensured a relatively efficient type inference algorithm, making subtype inference feasible for practical use.

Further development of this work would require actual implementations of the algorithms, and a study of many substantial examples. Such implementation would benefit from a combined approach incorporating features of the work of Fuh and Mishra (1988, 1989). Compilation of user programs also requires attention. While the SATISFY algorithm provides a kind of abstract compilation, in practice we would wish to generate code for function definitions by extracting coercion functions and generic arithmetic or equality operations as additional parameters. The possibility of user-defined types, which may be subtypes of primitive types, is also a promising area for further work. In the database area, there are a host of questions on implementation and optimisation which deserve attention. On the theoretical front, our conjecture on the equivalence of different type-annotations of an expression requires proof. The relationship between this work and the approach to strictness analysis by type inference in Wright and Dekker (1988) is also an interesting avenue for further study, since it too involves subtype relationships on different kinds of functions.

7 REFERENCES

- Aho AV, Garey MR and Ullman JD (1972) The Transitive Reduction of A Directed Graph, *SIAM J. Comput.*, Vol 1 No 2 pp 131-137
- Aho AV, Hopcroft JE and Ullman JD (1974) *The Design and Analysis of Computer Algorithms*, Addison Wesley
- Albano A, Cardelli L and Orsini R (1985) Galileo : A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, Vol 10 No 2 pp 230-260
- Atkinson MP and Kulkarni KG (1984) Experimenting with the Functional Data Model. *Databases — Role and Structure*, Stocker PM et al eds, Cambridge University Press pp 311-338
- Backus J (1978) Can Programming Be Liberated from the von Neumann Style? A Function Style and its Algebra of Programs. *Comm ACM*, Vol 21 No 8 pp 613-641
- Barendregt HP (1984) *The Lambda Calculus : Its Syntax and Semantics (Revised Edition)* North-Holland, Amsterdam
- Bird R and Wadler P (1988) *Introduction to Functional Programming* Prentice-Hall International
- Buneman P and Nikhil R (1981) A Practical Programming System for Databases. *Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture* pp 195-201
- Buneman P, Frankel RE and Nikhil R (1982) An Implementation Technique for Database Query Languages. *ACM Transactions on Database Systems* Vol 7 No 2 pp 164-186
- Burn GL, Hankin CL and Abramsky S (1986) Strictness Analysis for Higher-Order Functions. *Science of Computer Programming* Vol 7 No 3 pp 249-278
- Burroughs (1984) *Medilinc Programming Manual*. Item PACO-166, Burrough Ltd, Australia
- Cardelli L (1983) *ML under Unix*. Bell Laboratories Technical Report. Murray Hill, NJ
- Cardelli L (1984) A Semantics of Multiple Inheritance. *Semantics of Data Types*, Springer-Verlag LNCS 173, Berlin, pp 51-67
- Cardelli L and Wegner P (1985) On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys* Vol 17 No 8 pp 471-522
- Clement D, Despeyroux J, Despeyroux T and Kahn G (1986) A Simple Applicative Language : Mini – ML, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* pp 13-27

- Clocksin WF and Mellish CS (1984) *Programming in Prolog (Second Edition)*. Springer-Verlag, Berlin
- Cohn PM (1981) *Universal Algebra*. D Reidel, Dordrecht, Holland
- Collier PA (1987) Type Inference in the Presence of a Basic Type Hierarchy. *Actas de la VII Conferencia de la Sociedad Chilena de Ciencia de la Computacion*, Santiago, pp 87-96
- Coquand T (1985) Sur l'Analogie entre les Propositions et les Types. *Combinators and Functional Programming Languages*, Springer-Verlag LNCS 242, Berlin, pp 71-84
- Damas L and Milner R (1982) Principal Type-schemes for Functional Programs. *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp 207-212
- Dekker AH (1983) *Implementation of a Backus FP System*. Honours Thesis. Information Science Department, University of Tasmania
- Dekker AH (1987) *A Type Inference System for Subtype Hierarchies*. Technical Report R87-8, Department of Information Science, University of Tasmania
- Dekker AH (1988) *Output as Reduction in Functional Programming Languages*. Technical Report R88-11, Department of Electrical Engineering and Computer Science, University of Tasmania
- Dietrich R and Hagl F (1988) A Polymorphic Type System with Subtypes for Prolog. *ESOP '88 : 2nd European Symposium on Programming*, Springer-Verlag LNCS 300, Berlin, pp 79-93
- Ehrig H, Kreowski H-J, Mahr B and Padawitz P (1982) Algebraic Implementation of Abstract Data Types. *Theoretical Computer Science*, Vol 20 pp 209-263
- Fairbairn J (1985) *Design and Implementation of a Simple Typed Language Based on the Lambda-Calculus*. Technical Report no 75, Computer Laboratory, University of Cambridge
- Fairbairn J (1986) A New Type-Checker for a Functional Language. *Science of Computer Programming*, Vol 6 pp 273-290
- Fuh Y-C and Mishra P (1988) Type Inference with Subtypes. *ESOP '88 : 2nd European Symposium on Programming*, Springer-Verlag LNCS 300, Berlin, pp 94-114
- Fuh Y-C and Mishra P (1989) Polymorphic Type Inference : Closing the Theory-Practice Gap. *TAPSOFT '89 : Volume 2*, Springer-Verlag LNCS 352, Berlin, pp 167-183

- Goguen JA, Thatcher JW and Wagner EG (1978) *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*. *Current Trends in Programming Methodology IV*, Yeh R ed, Prentice Hall
- Henson MC (1985) *A Comprehensive Theory of Types and Coercions*. Internal Report, Department of Computer Science, University of Essex
- Henson MC (1987) *Elements of Functional Languages*. Blackwell Scientific Publications, Oxford
- Herrlich H and Strecker GE (1979) *Category Theory : An Introduction*, Heldermann Verlag, Berlin
- Hindley R (1969) The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, Vol 146 pp 29-60
- Holyer I (1988) *Practical Programming in a Functional Language*. Technical Report R86-6, Information Science Department, University of Tasmania
- Holyer I (1988) *Types and Sets in Functional Languages*. Technical Report CS-88-11, Computer Science Department, University of Bristol
- Huet G (1986) Deduction and Computation. *Fundamentals of Artificial Intelligence*, Springer-Verlag LNCS 232, Berlin pp 39-74
- Hull R and King R (1987) *Semantic Database Modelling : Survey, Applications, and Research Issues*. ACM Computing Surveys, Vol 19 No 3 pp 201-260
- Jategaonkar LA and Mitchell JC (1988) ML with Extended pattern Matching and Subtypes. *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*
- Kanellakis PC and Mitchell JC (1989) Polymorphic unification and ML typing. *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pp 105-115
- Kazmierczak EA (1989) *Algebraic Reasoning in Lambda Calculi*. Draft PhD thesis, University of Tasmania
- Klop JW (1980) *Combinatory Reduction Systems*. Mathematical Centre Tracts 127, Mathematisch Centrum, Amsterdam
- Kluzniak F (1987) Type Synthesis for Ground Prolog. *Logic Programming : Proceedings of the Fourth International Conference*, Volume 2 pp 788-816

- Mitchell JC and Plotkin GD (1985) Abstract types have existential type. *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp 37-51
- Mycroft A (1984) Polymorphic Type Schemes and Recursive Definitions. *International Symposium on Programming, 6th Colloquium*. Springer-Verlag LNCS 167, Berlin, pp 217-288
- Mycroft A and O'Keefe RA (1984) A Polymorphic Type System for Prolog, *Artificial Intelligence*, Vol 23 pp 295-307
- Organick EI (1973) *Computer System Organization : The B5700/B6700 Series*. Academic Press, New York
- Peyton Jones SL (1987) *The Implementation of Functional Programming Languages*. Prentice-Hall International.
- Rémy D (1989) Type checking records and variants in a natural extension of ML. *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pp 77-87
- Reynolds JC (1980) Using Category Theory to Design Implicit Conversions and Generic Operators. *Semantics-Directed Compiler Generation*. Springer-Verlag LNCS 94, Berlin, pp 211-258
- Reynolds JC (1984) Towards a Theory of Type Structure. *Programming Symposium : Proceedings, Colloque sur la Programmation*, Springer-Verlag LNCS 19, Berlin pp 408-425
- Reynolds JC (1985) Three Approaches to Type Structure. *Mathematical Foundations of Software Development*. Springer-Verlag LNCS 185, Berlin, pp 97-138
- Robinson JA (1965) A machine-oriented logic based on the resolution principle. *Journal of the ACM*, Vol 12 No 1 pp 23-41
- Ronchi Della Rocca S (1988) Principal Type Scheme and Unification for Intersection Type Discipline. *Theoretical Computer Science*, Vol 59 pp 181-209
- Shipman DW (1981) The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, Vol 6 No 1 pp 140-173
- Stansifer R (1988) Type Inference with Subtypes. *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*.
- Thatte S (1988) Type Inference with Partial Types. *Automata, Languages and Programming : 15th Colloquium*, Springer-Verlag LNCS 317, Berlin, pp 615-629
- Turner DA (1979) *Aspects of the Implementation of Programming Languages*. D Phil thesis, University of Kent

- Wadler P and Blott S (1989) How to make *ad-hoc* polymorphism less *ad hoc*. *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pp 60-76
- Wand M (1987) Complete Type Inference for Simple Objects. *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, pp 37-44
- Wand M (1988) Corrigendum : Complete Type Inference for Simple Objects. *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, p132
- Wright DA and Dekker AH (1988) *Strictness Analysis and Type Inference for the λ -calculus (Summary)*. Technical Report R88-9, Department of Electrical Engineering and Computer Science, University of Tasmania
- Zobel J (1987) Derivation of Polymorphic Types for Prolog Programs. *Logic Programming : Proceedings of the Fourth International Conference*, Volume 2 pp 817-838

Table 1 : Rewrite Rules for Programs

1	$plus (n : nat) (m : nat) : nat \Rightarrow n + m : nat$
2	$plus (n : int) (m : int) : int \Rightarrow n + m : int$
3	$neg (n : int) : int \Rightarrow -n : int$
4	$cond (\mathbf{true} : bool) (e_1 : \tau) (e_2 : \tau) : \tau \Rightarrow e_1 : \tau$
5	$cond (\mathbf{false} : bool) (e_1 : \tau) (e_2 : \tau) : \tau \Rightarrow e_2 : \tau$
6	$and (\mathbf{true} : bool) (e : bool) : bool \Rightarrow e : bool$
7	$and (\mathbf{false} : bool) (e : bool) : bool \Rightarrow \mathbf{false} : bool$
8	$ochoose (e_1 : \tau) (e_2 : \tau \rightarrow \tau') (absent : optional \tau) : \tau' \Rightarrow e_1 : \tau'$
9	$ochoose (e_1 : \tau) (e_2 : \tau \rightarrow \tau') (present (e_3 : \tau) : optional \tau) : \tau' \Rightarrow$ $(e_2 : \tau \rightarrow \tau') (e_3 : \tau) : \tau'$
10	$lchoose (e_1 : \tau') (e_2 : \tau \rightarrow list \tau \rightarrow \tau') (nil : list \tau) : \tau' \Rightarrow e_1 : \tau'$
11	$lchoose (e_1 : \tau') (e_2 : \tau \rightarrow list \tau \rightarrow \tau')$ $(cons (e_3 : \tau) (e_4 : list \tau) : list \tau) : \tau' \Rightarrow$ $(e_2 : \tau \rightarrow list \tau \rightarrow \tau') (e_3 : \tau) (e_4 : list \tau) : \tau'$
12	$case (e_1 : \tau \rightarrow \tau') (e_2 : \tau' \rightarrow \tau') (inl (e_3 : \tau) : \tau + \tau') : \tau' \Rightarrow$ $(e_1 : \tau \rightarrow \tau') (e_3 : \tau) : \tau'$
13	$case (e_1 : \tau \rightarrow \tau') (e_2 : \tau' \rightarrow \tau') (inr (e_3 : \tau) : \tau + \tau') : \tau' \Rightarrow$ $(e_2 : \tau' \rightarrow \tau') (e_3 : \tau) : \tau'$
14	$fst (pair (e_1 : \tau) (e_2 : \tau') : \tau \times \tau') : \tau \Rightarrow e_1 : \tau$
15	$snd (pair (e_1 : \tau) (e_2 : \tau') : \tau \times \tau') : \tau' \Rightarrow e_2 : \tau'$
16	$fix (e : \tau \rightarrow \tau) : \tau \Rightarrow (e : \tau \rightarrow \tau) (fix (e : \tau \rightarrow \tau) : \tau) : \tau$
17	$((\lambda x : \tau'. e_1 : \tau) : \tau' \supset \tau) (e_2 : \tau') : \tau \Rightarrow e_1 : \tau$
18	$((\lambda x : \tau'. e_1 : \tau) : \tau' \rightarrow \tau) (e_2 : \tau') : \tau \Rightarrow [e_2 : \tau'/x : \tau'] (e_1 : \tau)$
19	$(let x = [\alpha_1, \dots, \alpha_n] (e_1 : \tau) in e_2 : \tau') : \tau' \Rightarrow$ $[[\alpha_1, \dots, \alpha_n] (e_1 : \tau) // x] (e_2 : \tau')$

- 20 $c : \rho : \pi \Rightarrow c : \pi, \quad c \in K(\rho)$
- 21 $(\text{absent} : \text{optional } \tau) : \text{optional } \tau' \Rightarrow \text{absent} : \text{optional } \tau'$
- 22 $(\text{absent} : \text{optional } \tau) : \text{list } \tau' \Rightarrow \text{nil} : \text{list } \tau'$
- 23 $(\text{present } (e_3 : \tau) : \text{optional } \tau) : \text{optional } \tau' \Rightarrow$
 $\text{present } ((e_3 : \tau) : \tau') : \text{optional } \tau'$
- 24 $(\text{present } (e_3 : \tau) : \text{optional } \tau) : \text{list } \tau' \Rightarrow$
 $\text{cons } ((e_3 : \tau) : \tau') (\text{nil} : \text{list } \tau') : \text{list } \tau'$
- 25 $(\text{nil} : \text{list } \tau) : \text{list } \tau' \Rightarrow \text{nil} : \text{list } \tau'$
- 26 $(\text{cons } (e_3 : \tau) (e_4 : \text{list } \tau) : \text{list } \tau) : \text{list } \tau' \Rightarrow$
 $\text{cons } ((e_3 : \tau) : \tau') ((e_4 : \text{list } \tau) : \text{list } \tau') : \text{list } \tau'$
- 27 $(\text{inl } (e_3 : \tau) : \tau + \tau') : \tau' + \tau'' \Rightarrow \text{inl } ((e_3 : \tau) : \tau') : \tau' + \tau''$
- 28 $(\text{inr } (e_3 : \tau) : \tau + \tau') : \tau' + \tau'' \Rightarrow \text{inr } ((e_3 : \tau) : \tau') : \tau' + \tau''$
- 29 $(\text{pair } (e_1 : \tau) (e_2 : \tau') : \tau \times \tau') : \tau'' \times \tau''' \Rightarrow$
 $\text{pair } ((e_1 : \tau) : \tau'') ((e_2 : \tau') : \tau''') : \tau'' \times \tau'''$
- 30 $((\lambda x : \tau. e_1 : \tau) : \tau' \supset \tau) : \tau'' \supset \tau' \Rightarrow$
 $(\lambda x : \tau''. (e_1 : \tau) : \tau') : \tau'' \supset \tau'$
- 31 $((\lambda x : \tau. e_1 : \tau) : \tau' \supset \tau) : \tau'' \rightarrow \tau' \Rightarrow$
 $(\lambda x : \tau''. (e_1 : \tau) : \tau') : \tau'' \rightarrow \tau'$
- 32 $((\lambda x : \tau. e_1 : \tau) : \tau' \rightarrow \tau) : \tau'' \rightarrow \tau' \Rightarrow$
 $(\lambda x : \tau''. [(x : \tau'') : \tau/x : \tau] (e_1 : \tau) : \tau') : \tau'' \rightarrow \tau'$
- 33 $(\gamma (e_1 : \tau_1) \dots (e_n : \tau_n) : \tau' \rightarrow \tau) : \tau'' \rightarrow \tau' \Rightarrow$
 $(\lambda x : \tau''. (\gamma (e_1 : \tau_1) \dots (e_n : \tau_n) : \tau' \rightarrow \tau))$
 $(x : \tau'' : \tau') : \tau : \tau' : \tau'' \rightarrow \tau'$
- where $n = 0, \gamma \in \{\text{fst}, \text{snd}, \text{inl}, \text{inr}, \text{present}, \text{fix}, \text{neg}\}$
or $n \leq 1, \gamma \in \{\text{pair}, \text{cons}, \text{plus}, \text{and}, \text{eq}\}$
or $n \leq 2, \gamma \in \{\text{case}, \text{lchoose}, \text{ochoose}, \text{cond}\}$
- 34 $\text{eq } (c : \pi) (c : \pi) : \text{bool} \Rightarrow \text{true} : \text{bool}, \quad c \in K(\pi)$
- 35 $\text{eq } (c : \pi) (c' : \pi) : \text{bool} \Rightarrow \text{false} : \text{bool}, \quad c, c' \in K(\pi), c \neq c'$

- 36 $eq (absent : optional \tau) (absent : optional \tau) : bool \Rightarrow \mathbf{true} : bool$
 37 $eq (absent : optional \tau) (present (e_3 : \tau) : optional \tau) : bool \Rightarrow$
 $\mathbf{false} : bool$
- 38 $eq (present (e_1 : \tau) : optional \tau) (absent : optional \tau) : bool \Rightarrow$
 $\mathbf{false} : bool$
- 39 $eq (present (e_1 : \tau) : optional \tau) (present (e_3 : \tau) : optional \tau) : bool \Rightarrow$
 $eq (e_1 : \tau) (e_3 : \tau) : bool$
- 40 $eq (nil : list \tau) (nil : list \tau) : bool \Rightarrow \mathbf{true} : bool$
 41 $eq (nil : list \tau) (cons (e_3 : \tau) (e_4 : list \tau) : list \tau) : bool \Rightarrow \mathbf{false} : bool$
 42 $eq (cons (e_1 : \tau) (e_2 : list \tau) : list \tau) (nil : list \tau) : bool \Rightarrow \mathbf{false} : bool$
 43 $eq (cons (e_1 : \tau) (e_2 : list \tau) : list \tau)$
 $(cons (e_3 : \tau) (e_4 : list \tau) : list \tau) : bool \Rightarrow$
 $and (eq (e_1 : \tau) (e_3 : \tau) : bool)$
 $(eq (e_2 : list \tau) (e_4 : list \tau) : bool) : bool$
- 44 $eq (inl (e_1 : \tau) : \tau + \tau') (inl (e_3 : \tau) : \tau + \tau') : bool \Rightarrow$
 $eq (e_1 : \tau) (e_3 : \tau) : bool$
- 45 $eq (inl (e_1 : \tau) : \tau + \tau') (inr (e_3 : \tau') : \tau + \tau') : bool \Rightarrow \mathbf{false} : bool$
 46 $eq (inr (e_1 : \tau') : \tau + \tau') (inl (e_3 : \tau) : \tau + \tau') : bool \Rightarrow \mathbf{false} : bool$
 47 $eq (inr (e_1 : \tau') : \tau + \tau') (inr (e_3 : \tau') : \tau + \tau') : bool \Rightarrow$
 $eq (e_1 : \tau') (e_3 : \tau') : bool$
- 48 $eq (pair (e_1 : \tau) (e_2 : \tau') : \tau \times \tau') (pair (e_3 : \tau) (e_4 : \tau') : \tau \times \tau') : bool$
 $\Rightarrow and (eq (e_1 : \tau) (e_3 : \tau) : bool)$
 $(eq (e_2 : \tau') (e_4 : \tau') : bool) : bool$