TYPE-CHECKING SYSTEMS

WITH

PARTICULAR APPLICATIONS TO FUNCTIONAL LANGUAGES

by

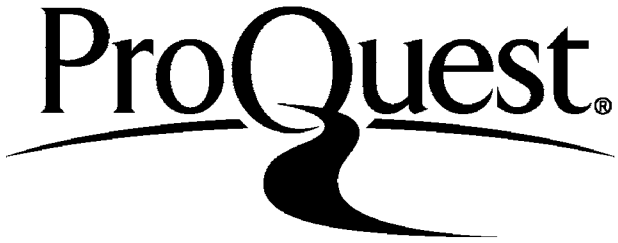
HOCK KUEN FRANCIS YEUNG

Royal Holloway College, University of London

Submitted to the University Of London

in September, 1976

for the Degree of Doctor of Philosophy.



Abstract

J.Morris in his thesis discovered that conventional type-checking systems inhibit users of typed languages and he left two problems for future solution--parametric polymorphism and circular types.

Any typed language L_i is related to a type-checking system T_j by a function Φ_{ij}. Consequently type-checking systems may be studied independently of particular languages. Therefore a logic to illustrate how such systems are intended to work must preserve language and machine independencies, and it must not be inhibited by Morris' two problems. We have therefore chosen the λ-K Calculus.

Fundamental concepts of types and type-checking are discussed and these include theorems of functionality, a set-theoretical approach to types, and intersection-types. After preliminary examination of previous type-checking systems, we propose two systems of our own. The first one we have implemented is System-F. In attempting to generalize it beyond the work of conventional type checkers we discovered that it is necessary to abandon the distinction between so-called statically- and dynamically-typed systems.

In this way we alight on our most fundamental problem. This is how to design type-checking systems that permit declaration of arbitrary functions and functionals whose type declarations are incomplete or missing. We solve this by introducing a class of type expressions we call type abstractions. We have also introduced a way to describe type-checking processes by certain sets of equations, and shown how to solve them. These thoughts are implemented in our second system, the System-Y. Later, we explored further the nature of circular types in the light of lattice theory. Both our systems are adequate to handle Morris' problems.

ACKNOWLEDGEMENTS

My grateful thanks go to my thesis supervisor, Mr. R.P. Edwards, for his guidance, constructive criticism, and patience throughout the investigation and preparation of this thesis.

Most importantly, I wish to thank my parents in Hong Kong for giving me their love, understanding, encouragement and confidence throughout my studies in Britain.

I would also like to thank Miss A.C.L. Man for typing the first draft of this thesis.

Finally, I wish to thank the University of London for its Postgraduate Studentship.

TABLE OF CONTENTS

PART TWO

PART THREE

INTRODUCTION

Types are checked in most implementations of high level languages (Algol68, Fortran, Pascal, etc.). However, it is fair to say that type checking is simply regarded as an adhoc process, and unlike other parts of language implementations (say, code generation), it is seldom mentioned. There are a few reasons why we have to carry out this project. Data types and structures have become increasingly important in recent language designs (e.g. Algol68) and we believe that this trend will continue in the foreseeable future. Hence it is reasonable to suggest that the naive type-checking systems as we have now would be unable to cope with future language developments. There are problems that have been known for a long time but still awaiting solutions, for example, parametric polymorphism and circular types. We hope that in attempting to solve these problems from first principles, we might be able to suggest some ways that we could discuss type checking systematically. On the other hand if type checking could be studied independently of particular languages, we might be able to shed some light on what future languages would be.

Of course, it is impossible to achieve all our aims in one project, so we shall concentrate on solving problems in parametric polymorphism and circular types here. We find that these problems are well exhibited in functions and

functionals. Hence, in choosing a model to represent these problems, there are two criteria to be satisfied. Firstly, it must preserve language and machine independences. Secondly, functions and functionals can be expressed in it conveniently. We have therefore chosen λ-K Calculus as the most suitable model.

We find it necessary in Part One of this thesis to review the known principles of the λ-K Calculus so as to exhibit those logical properties important to our work, and then to state the notation-- Reckon, in which we propose to abbreviate description of the larger functions needed in subsequent parts, although (it will bear repeating) notations are incidental to this thesis. We implemented Reckon on the computer during our undertaking.

After preliminary examination of previous type-checking system proposals in the light of our two problems, we propose two systems of our own. In Chapter 2 of Part Two, we discuss some of the fundamental concepts which includes theorems of functionality, a set-theoretical approach to types including unions and intersections of types. While avoiding throwing a load of mathematical symbols at readers, we formulate our proof of some theorems on logical deductions. We also demonstrate that these theorems are isomorphic to theorems in propositional Calculus.

The first type-checking system we have implemented is System-F. In attempting to genaralize it beyond the work of conventional type checkers we discover that it is necessary to abandon the distinction between so-called statically- and dynamically- type-checking systems.

The aim of this project is to identify the underlying logical properties of our problems and uncover their solutions progressively. Thus we find ourselves embarking on the logic of System-Y at the outset of Part Three. System-Y permits declarations of arbitrary functions and functionals even where type declarations of parameters are incomplete or missing. This is possible because we have introduced a class of type expressions which we call type abstractions. The analogy is with the progression to λ-abstraction in the λ-Calculi, or with the progression from propositional to first order expressions in the predicate Calculi. We have also introduced a way to describe type-checking processes by certain sets of equations, and shown how to solve them. Hence type checking may be studied independently of machines or even, it transpires, of particular interpretations of a system of types.

Both System-F and System-Y are implemented on CDC 64/6600 computers. So as to carry conviction to the reader, we have included the semantics of the implementing programs and some programming examples in the appendices immediately after theoretical discussions of the respective systems. The semantics were written in Reckon, so the reader is recommended to familiarize himself with the language from

the examples presented for this purpose in the appendix at the end of Part One.

In Chapter 2 of Part Three, we explored further the nature of circular types in the light of lattice theory, and showed how the problem can be handled by System-Y. What remains is for us to claim that both our systems are capable of handling parametric polymorphism and circular types.

PART ONE

CHAPTER ONE

THE λ-CALCULUS

Our prime interest in this project is to study type behaviour of computer programs with particular emphasis on functional languages, but without confining ourselves to any specific language.

Freedom from the various tedious constructs of a specific language allow us to see type-checking problems more clearly and deeply. We can also avoid the risk of being misled by features of a particular language into unfruitful investigations of problems that are of limited interest.

With regards to the above discussions, we find that λ-K Calculus is the most suitable model for our investigation (we shall call it simply λ-Calculus hereafter). The λ-Calculus is simple in its syntax. It has been claimed that all computable functions on natural numbers are λ-definable [Turing,1937]. Landin [Landin,1965] has also shown that the class of all Algol-60 programs is a subset of the class of λ-expressions; while [MacCarthy,1963] exhibited the class of linear lists over finite alphabets as only a 'conservative extension' of natural numbers, so that λ-K Calculus can describe data structures too. In a sentence,λ-Calculus is adequate for our demands on it.

A type system was proposed by Ledgard [Ledgard,1972] in handling type-checking of Algol-60 programs (and thus of only a subset of λ-expressions). We shall see in later chapters that his system shares shortcomings with other systems, especially in checking the types of functions. Those problems, such as parametric polymorphism and circular types, are well amplified in λ-Calculus. Thus our choice of λ-Calculus for investigation is well justified.

Another advantage that we have experienced in using λ-Calculus is that once a normal order of evaluation is prescribed so that each step of evaluation is defined uniquely, the complete process of evaluation can be studied stepwise. Intermediate results at any stage can be obtained which proved to be invaluable in verifying properties that are under investigation. Yet the λ-Calculus is machine independent.

1.1 Applicative Expressions and Their Conversions

In this section, we shall give a brief account of the syntax of λ-expressions and the laws of conversion will be discussed towards the end of the section.

1.1.0 Functions

Work on λ-Calculus was initially motivated by studies of functions. The notion of function is by no means a novel idea, be it be regarded as rule of correspondence, mapping or relation between two sets of objects. Suppose f is a singulary function (function of one argument) defined on a set of objects R_1 and yielding results in set R_2, then R_1 and R_2 are called domain and range of f respectively. It is possible that either or both domain and range of a function are sets of functions and perhaps the function itself as well. At least we do not wish to exclude this possibility.

Example

Identity function IDENT can be defined as
IDENT(x)=x regardless of what x is, thus
IDENT(IDENT)=IDENT

1.1.1 Applicative Expressions

Functions of two arguments can be re-expressed as corresponding functions of one argument, whose values are functions of one argument. We shall denote functional application by (f x) where f is a function and x is the argument of f.

Example

f(x,y)=((f' x)y)=f' x y

where (f' x) is the value obtained by applying an f' to x, which is a function of one argument.

In general, functions of n arguments (n≥2) can be re-expressed as functions of one argument, whose values are functions of (n-1) arguments and so on.

Some brackets can be eliminated by adopting the convention that expressions are left associative, for example, f x y=((f x)y). Only where is any ambiguity, we insist that brackets should be written.

1.1.2 λ-expressions

Suppose function f is defined as (f x)=x*x, let us consider the two statements below

(1) (f x) is greater than 100

(2) (f x) is a square function.

In deciding the truth of statement (1), we have to ask what is the value of x and consequently the value of (f x). So in case (1), we are concerned with values. On the other hand, in (2), value of x is irrelevant, and neither is (f x). The truth of the statement depends on how f is defined, or simply the process that is associated with it. Clearly the two occurrences of (f x) serve for two distinct purposes. The former stands for value of the function,

while, the latter the function itself.

Such ambiguities can be avoided if extra symbols are used to defferentiate the two cases. The symbol 'λ' is used for this purpose. Statement (2) can then be re-written as

"(λx:(f x)) is a square function" ...(a)

The first occurrence of x in (a) is called binding variable (in Algol, this is called the formal parameter of a function). We call (a) "λ-abstraction" of expression (f x), and (f x) is called "body" of (a).

1.1.3 The class of λ-expressions

Binding variables are separated from bodies by ":" in λ-expressions. Assuming that we have an infinite set of variables a,...,z,a_1,...,z_1,a_2,... and also the following symbols "λ", "(", ")", and ":", then arbitrary strings can be constructed from them. These strings are called well-formed-formulae only if they satisfy any of the following conditions :

(1) a variable is a well-formed-formula

(2) if F, A are well-formed-formulae, so is (F A). They may be called combination with F as operator and A as operand.

(3) if M is a well-formed formula, so is (λx:M), where x is the binding variable which may or may not occur in M.

Examples

(1) $(\lambda x:(x\ x))$

(2) $(\lambda x:(\lambda y:(x\ y)))$

(3) $(\lambda x:(\lambda y:y))$

Brackets can be omitted with the understanding that

(1) λ-expressions are left associative

(2) an opening bracket "(" is placed immediately after ":" and the matching closing bracket ")" is inserted as far right as possible in the subexpression.

Examples

(1) $(\lambda f:\lambda x:f(f\ x)) \equiv (\lambda f:(\lambda x:(f(f\ x))))$

(2) $(\lambda M:\lambda N:\lambda a:\lambda b:(M\ a)((N\ a)b)) \equiv$

$(\lambda M:(\lambda N:(\lambda a:(\lambda b:((M\ a)((N\ a)b))))))$

1.1.4 Free and Bound variables

It is well known in mathematics that if we replace all occurrences of x in the integral "$\int x\ dx$" by "y", the resulting integral "$\int y\ dy$" has the same meaning as the original one. However this is not true for "$\int x\ y\ dx$" The phenomeon is formalized in λ-Calculus. A variable is free or bound in a formula depending on the following conditions stated recursively,

(1) a variable is free in itself

(2) A variable x is free in (F A) if it is free in both F and A. Conversely, if it is bound in either F or A, then it is bound in (F A).

(3) All occurrences of x will be bound in (λx:M) (this statement is superfluous if x does not occur in M) where M is a well-formed formula. A variable y which is not the same as x, will be free or bound in (λx:M) depends on whether it is free or bound in M.

Example

(1) In (λx:(x y)), x is bound and y is free

(2) In (λx:λy:(x y)), both x and y are bound

1.1.5 Substitution

We use the notation "[N/x]M" to stand for the formula obtained by substituting all occurrences of x in M by N. Notice that if N is not a simple variable and x is bound in M, then the resulting formula would not be well formed.

Example

(1) [y/x](λx:(x x))=(λy:(y y))

(2) [(λw:w)/x](λx:(x x))=(λ(λw:w):((λw:w)(λw:w)))

1.1.6 Conversion

We proceed to the operations in λ-Calculus -- transformation or conversion of well-formed formulae. Notationally, we use "A→B" to stand for conversion from A to B. One may regard conversions as processes for replacing a part of a well-formed formula by another well-formed formula according to the following rules :

(1) α-conversion

A part M of a well formed formula can be replaced by [y/x]M provided that variable x is not free in M and y does not occur in M.

Examples

for the expression $(\lambda a{:}\lambda b{:}a\ b)$, we consider 3 different cases :

(a) We can replace $(\lambda b{:}a\ b)$ by $[c/b](\lambda b{:}a\ b)$, thus $(\lambda a{:}\lambda b{:}a\ b)\rightarrow(\lambda a{:}\lambda c{:}a\ c)$

(b) we cannot replace $(a\ b)$ by $[c/b](a\ b)$

(c) we cannot replace $(\lambda b{:}a\ b)$ by $[a/b](\lambda b{:}a\ b)$

(2) β-reduction

A part $((\lambda x{:}M)N)$ of a well-formed formula can be replaced by [N/x]M, provided that the bound variables of M are distinct from both x and the free variable of N.

Examples

(a) We can replace $((\lambda x{:}\lambda y{:}x\ y)z)$ by $[z/x](\lambda y{:}x\ y)$, thus $((\lambda x{:}\lambda y{:}x\ y)z)\rightarrow(\lambda y{:}z\ y)$

(b) we cannot replace $((\lambda x{:}\lambda y{:}x\ y)y)$ by $[y/x](\lambda y{:}x\ y)$

(c) we cannot replace $(\lambda x{:}((\lambda x{:}x\ y)x))$ by $[(\lambda z{:}z)/x]((\lambda x{:}x\ y)x)$

(3) β-expansion

A part [N/x]M can be replaced by ((λx:M)N), provided that the bound variables of M are distinct both from x and from the free variables of N.

(4) η-conversion

A part (λx:(M x)) can be replaced by M if x is not free in M.

Example

(λx:(λy:y y)x) can be replaced by (λy:y y),

but, this is not true for (λx:(λy:x y)x).

(5) δ-conversion

The Calculus we have been discussing so far is known strictly as the λ-K-αβη-Calculus. We shall later informally add some rules of correspondence to permit primitive functions to be applied to primitive arguments, by what we called δ-rules, to give the fuller λ-K-αβδη-Calculus. Since these δ-rules are conveniences for practical computing only, we need answer only for the interpretation of our αβη-Calculus, and consider δ-rules as a speed up trick for a special branch in our interpreter (see Chapter 2).

1.2 λ-Definability of Computable Functions

Applicative expressions which have so far been considered only syntactically may be given interpretations as denoting computable objects.

In section 1.1.2 we said that (f x) in "(f x) is a square function" should be replaced by (λx:f x) which describes the corresponding function. This is generalized by the law of η-conversion which permits (λx:M x) to be interpreted as function for any x, M. For, if x is not free in M, (λx:M x)=M and this is true for anything bound to x. In other words, for any object x', whatever the result is by applying M to x', the same result will be obtained by applying (λx:M x) to x'. Thus, η-conversion is also called the law of Extensionality. Hence, λ-expressions describe functions and this interpretation is complemented by regarding β-reduction as application of function to arguments; while β-expansion may be interpreted as abstraction or extraction of common subexpressions to create a function. There is no useful interpretation for α-conversion in this respect as no new object is created by the process.

If (λx:λy:F x y) denotes the function which gives the sum of any two numbers, then ((λx:λy:F x y)x') can be interpreted as the function which adds a fixed increment x' to any given number. We say the latter function is obtained from the former by "partial application", the concept which we shall use very often later.

1.2.1 λ-Definability

An object is said to be λ-definable if there exists a well-formed formula which denotes the object and in that case we say the formula λ-defines the object. For example, we might define numeral 0 by (λa:λb:b), 1 by (λa:λb:a b), 2 by (λa:λb:a(a b)) and so on. It is more convenient to use symbols to stand for complicated formulae, for example, 2 for (λa:λb:a(a b)). In general, we write sym←M if "M" is a well-formed formula to be designated by the symbol "sym".

Example

$\underline{3} \leftarrow (\lambda a{:}\lambda b{:}(a(a(a\ b))))$

and 3 is λ-defined by $\underline{3}$

If a function f is said to be λ-definable, then there should exist a well-formed formula F such that F M→N if f(m)=n, where m, n is λ-defined by M and N respectively.

Before we state the result that all computable functions are λ-definable, we shall give below a list of definitions which are required for subsequent discussions.

Definition (1)

The successor function is λ-defined by S, where

$S \leftarrow (\lambda n{:}\lambda f{:}\lambda x{:}f(n\ f\ x))$

Example

$S\ \underline{1} \rightarrow (\lambda f{:}\lambda x{:}f(\underline{1}\ f\ x))$

$\rightarrow (\lambda f{:}\lambda x{:}f((\lambda a{:}\lambda b{:}a\ b)f\ x)) \rightarrow (\lambda f{:}\lambda x{:}f(f\ x))$

Definition (2)

The addition function is λ-defined by A, where

A←(λm:λn:λf:λx:m f(n f x))

Example

A $\underline{1}$ $\underline{2}$→(λf:λx:$\underline{1}$ f((λa:λb:a(a b))f x))

→(λf:λx:$\underline{1}$ f(f(f x)))

→(λf:λx:(λa:λb:a b)f(f(f x)))

→(λf:λx:f(f(f x)))

Definition (3)

The multiplication function is λ-defined by M, where

M←(λm:λn:λf:λx:m(n f)x)

Example

M $\underline{2}$ $\underline{2}$→(λf:λx:$\underline{2}$($\underline{2}$ f)x)

→(λf:λx:($\underline{2}$ f)($\underline{2}$ f x))

→(λf:λx:($\underline{2}$ f)(f(f x)))

→(λf:λx:f(f(f(f x))))

Definition (4)

The constructor function which forms an ordered pair from two arbitrary objects is λ- defined by PAIR, where

PAIR←(λx:λy:λf:f x y)

Definition (5)

The two functions for selecting the first and second component of an ordered pair are λ-defined by K_1 and K_2 respectively, where,

$K_1 \leftarrow (\lambda x:\lambda y:x)$

$K_2 \leftarrow (\lambda x:\lambda y:y)$

Examples

PAIR a b $K_1 \rightarrow K_1$ a b$\rightarrow$a

PAIR a b $K_2 \rightarrow K_2$ a b$\rightarrow$b

Definition-schema (6)

. The constructor function which forms ordered n-tuples from n objects ($n \geq 2$) is λ-defined by NTUPLE, where for given n

$NTUPLE \leftarrow (\lambda x_1:\lambda x_2:\ldots:\lambda x_n:\lambda f:f\ x_1\ x_2\ \ldots x_n)$

Definition-schema (7)

The function which selects the i' th object out of ordered n-tuples ($i \leq n$) is λ-defined by U_n^i, where

$U_n^i \leftarrow (\lambda x_1:\lambda x_2:\ldots:\lambda x_n:x_i)$

Definition (8)

Let $G \leftarrow (\lambda n:PAIR(n\ \underline{0})(S(n\ \underline{0})))$.

The predecessor function is λ-defined by P, where,

$P \leftarrow (\lambda n:n\ G(PAIR\ \underline{0}\ \underline{0})K_1)$

Example

$P\ \underline{2} \rightarrow \underline{2}\ G(PAIR\ \underline{0}\ \underline{0})K_1$

$\rightarrow G(G(PAIR\ \underline{0}\ \underline{0}))K_1$

$\rightarrow G(PAIR((PAIR\ \underline{0}\ \underline{0})\underline{0})(S(PAIR\ \underline{0}\ \underline{0}\ \underline{0})))K_1$

$\rightarrow G(PAIR(\underline{0}\ \underline{0}\ \underline{0})(S(\underline{0}\ \underline{0}\ \underline{0})))K_1$

$\rightarrow G(PAIR\ \underline{0}\ \underline{1})K_1$

$\rightarrow$PAIR(PAIR $\underline{0}$ $\underline{1}$ $\underline{0}$)(S(PAIR $\underline{0}$ $\underline{1}$ $\underline{0}$))K_1

$\rightarrow$PAIR $\underline{1}$ $\underline{2}$ K_1

$\rightarrow K_1 \underline{1}$ $\underline{2}$

$\rightarrow \underline{1}$

Example

P $\underline{0} \rightarrow \underline{0}$ G(PAIR $\underline{0}$ $\underline{0}$)K_1

$\rightarrow$PAIR $\underline{0}$ $\underline{0}$ K_1

$\rightarrow K_1$ $\underline{0}$ $\underline{0}$

$\rightarrow \underline{0}$

Definition (9)

If the infixed operation $\underline{\cdot}$ is defined as follows

$x \mathbin{\underline{\cdot}} y = 0$ if $x \leq y$ otherwise $x-y$, then $\underline{\cdot}$ can be λ-defined by S_0, where

$S_0 \leftarrow (\lambda x:\lambda y:y\ P\ x)$

Example

S_0 $\underline{3}$ $\underline{2} \rightarrow \underline{2}$ P $\underline{3} \rightarrow$P P $\underline{3} \rightarrow \underline{1}$

Definition (10)

The function for finding the minimum of 2 numbers can be λ-defined by MIN, where

MIN$\leftarrow (\lambda x:\lambda y:S_0\ y\ (S_0\ y\ x))$

Example

MIN $\underline{3}$ $\underline{2} \rightarrow S_0$ $\underline{2}(S_0$ $\underline{2}$ $\underline{3})$

$\rightarrow S_0$ $\underline{2}$ $\underline{0}$

$\rightarrow \underline{2}$

Definition (11)

If the function LESS is defined as follows

(a) LESS x y = $\underline{1}$ if $x<y$

(b) LESS x y = $\underline{0}$ if $x\geq y$

then LESS can be λ-defined by L where

$L\leftarrow(\lambda x:\lambda y:\text{MIN } \underline{1}(S_0\ y\ x))$

Example

$L\ \underline{2}\ \underline{3}\rightarrow\text{MIN } \underline{1}(S_0\ \underline{3}\ \underline{2})$

$\rightarrow\text{MIN } \underline{1}\ \underline{1}$

$\rightarrow\underline{1}$

$L\ \underline{3}\ \underline{2}\rightarrow\text{MIN } \underline{1}(S_0\ \underline{2}\ \underline{3})$

$\rightarrow\text{MIN } \underline{1}\ \underline{0}$

$\rightarrow\underline{0}$

Definition (12)

The duplication function can be λ-defined by W, where

$W\leftarrow(\lambda f:\lambda x:f\ x\ x)$

Example

$W\ M\ \underline{2}\rightarrow M\ \underline{2}\ \underline{2}\rightarrow\underline{4}$ (by definition 3)

So (W M) λ-defines the SQUARE function.

Definition (13)

The composition function can be λ-defined by B, where

$B\leftarrow(\lambda f:\lambda g:\lambda x:f(g\ x))$

Example

$B(W\ M)(W\ M)\underline{2}\rightarrow W\ M(W\ M\ \underline{2})\rightarrow W\ M\ \underline{4}\rightarrow\underline{16}$

Definition (14)

Let $C^* \leftarrow (\lambda f:\lambda x:\lambda y:f\ y\ x)$

$C \leftarrow (\lambda a:a\ C^*\ \underline{0}\ K_2\ K_1)$

Example

$C\ \underline{0} \rightarrow \underline{0}\ C^*\ \underline{0}\ K_2\ K_1$

$\rightarrow \underline{0}\ K_2\ K_1$

$\rightarrow K_1$

Example

$C\ \underline{1} \rightarrow \underline{1}\ C^*\ \underline{0}\ K_2\ K_1$

$\rightarrow C^*\ \underline{0}\ K_2\ K_1$

$\rightarrow \underline{0}\ K_1\ K_2$

$\rightarrow K_2$

Definition (15)

Let $H \leftarrow (\lambda a:\lambda f:\lambda g:\lambda n:((PAIR(K_1\ a)(f(P\ n))(C(MIN\ n\ \underline{1})))$
$(g(P\ n))))$

Examples

(1) $H\ a\ f\ g\ \underline{0} \rightarrow (PAIR(K_1\ a)(f\ \underline{0})(C\ \underline{0}))(g\ \underline{0})$

$\rightarrow (K_1(K_1\ a)(f\ \underline{0}))(g\ \underline{0})$

$\rightarrow (K_1\ a)(g\ \underline{0})$

$\rightarrow a$

(2) $H\ a\ f\ g\ (S\ n) \rightarrow (PAIR(K_1\ a)(f\ P(S\ n))$
$(C(MIN(S\ n)\underline{1})))(g(P(S\ n)))$

$\rightarrow (PAIR(K_1\ a)(f\ n)(C\ \underline{1}))(g\ n)$

$\rightarrow (K_2(K_1\ a)(f\ n))(g\ n)$

$\rightarrow f\ n\ (g\ n)$

Definition (16)

Let $Y \leftarrow (\lambda f:(W(B\ f))(W(B\ f)))$

Example

$Y\ f \rightarrow (W(B\ f))(W(B\ f))$

$\rightarrow (B\ f)(W(B\ f))(W(B\ f))$

$\rightarrow f((W(B\ f))(W(B\ f)))$

$\rightarrow f(Y\ f)$

This provides the schema for defining recursive functions in the following way.

Definition (17)

Let $REC \leftarrow (\lambda x:\lambda f:Y(H\ x\ f))$, therefore

$REC\ x\ f\ \underline{0} \rightarrow Y(H\ x\ f)\underline{0}$

$\rightarrow H\ x\ f\ (Y(H\ x\ f))\underline{0}$

$\rightarrow x$ (by definition 14)

$REC\ x\ f\ (S\ n) \rightarrow H\ x\ f\ (Y(H\ x\ f))(S\ n)$

$\rightarrow f\ n\ (Y(H\ x\ f)n)$

$\rightarrow f\ n\ (REC\ x\ f\ n)$

Example

We can add the first two numbers by writing

$REC\ \underline{0}\ A(S\ \underline{2}) \rightarrow A\ \underline{2}(REC\ \underline{0}\ A\ \underline{2})$

$\rightarrow A\ \underline{2}(A\ \underline{1}(REC\ \underline{0}\ A\ \underline{1}))$

$\rightarrow A\ \underline{2}(A\ \underline{1}\ \underline{0})$

$\rightarrow A\ \underline{2}\ \underline{1}$

$\rightarrow \underline{3}$

1.2.2 Primitive Recursive Functions

Primitive recursive functions can be defined recursively as follows,

(1) S, $\underline{0}$ and U_n^i are primitive recursive functions.

(2) If a function f of n arguments is defined by "composition" in terms of functions g(of m arguments) and h_1, h_2..., h_m (each of n arguments) as follows,

$$f(x_1, \ldots, x_n)=g(h_1(x_1, \ldots, x_n),\ldots h_m(x_1,\ldots x_n))$$

and if g, h_1, h_2, ... h_m are primitive recursive, then f is primitive recursive.

(3) If a function f of n+1 arguments is defined by "primitive recursion" in terms of functions g_1 and g_2 as follows

$$f(x_1, \ldots, x_n\ 0)=g_1(x_1, \ldots, x_n)$$

$$f(x_1, \ldots, x_n, (y+1))=g_2(x_1, \ldots, x_n,y,f(x_1,\ldots, x_n, y))$$

and if g_1 and g_2 are primitive recursive, then f is also primitive recursive.

It has already been shown that $\underline{0}$, S and U_n^i are λ-definable. If G, H_1, H_2, ..., H_m λ-defines g, h_1, ..., h_m as mentioned in (2), then F_1 λ-defines f where

$$F_1 \leftarrow (\lambda x_1 : \lambda x_2 : \ldots : \lambda x_n : G(H_1 x_1 x_2 \ldots x_n)(H_2 x_1 x_2 \ldots)\ldots (H_m\ x_1\ x_2\ \ldots\ x_n))$$

If G_1 and G_2 λ-define g_1 and g_2 as mentioned in (3), then F_2 λ-defines f, where

$$F_2 \leftarrow (\lambda x_1 : \lambda x_2 : \ldots \lambda x_n : \lambda y : \mathrm{REC}(G_1\ x_1 \ldots\ x_n)(G_2\ x_1 \ldots\ x_n)y)$$

hence, for-loops can be λ-defined by F_2.

The class of primitive recursive functions is known to include substantially all the ordinarily used numerical functions [Church,1941], and as shown above, all primitive recursive functions are λ-definable.

1.2.3 General Recursive Functions

If R of n+1 arguments is a proposition function, then the notation $\mu^*(\lambda y : R\ x_1\ x_2 \ldots\ x_n\ y)$ stands for the least value of y (if it exists) that $(R\ x_1 \ldots\ x_n\ y)$ is true.

let $M \leftarrow (\lambda x : \lambda y : \lambda z : (C(y\ z))(x\ y\ (S\ z))z)$

then μ^* can be λ-defined by μ, where

$\mu \leftarrow (Y\ M)$

Example

This example is for finding the least value y that is greater than 1.

$$\mu(\lambda y{:}L\ \underline{1}\ y)\underline{0} \rightarrow Y\ M(\lambda y{:}L\ \underline{1}\ y)\underline{0}$$
$$\rightarrow M(Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{0}$$
$$\rightarrow (C\ \underline{0})((Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{1})\underline{0}$$
$$\rightarrow K_1((Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{1})\underline{0}$$
$$\rightarrow Y\ M(\lambda y{:}L\ \underline{1}\ y)\underline{1}$$
$$\rightarrow M(Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{1}$$
$$\rightarrow (C((\lambda y{:}L\ \underline{1}\ y)\underline{1}))((Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{2})\underline{1}$$
$$\rightarrow (C\ \underline{0})((Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{2})\underline{1}$$
$$\rightarrow Y\ M(\lambda y{:}L\ \underline{1}\ y)\underline{2}$$
$$\rightarrow (C((\lambda y{:}L\ \underline{1}\ y)\underline{2}))((Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{3})\underline{2}$$
$$\rightarrow (C\ \underline{1})((Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{3})\underline{2}$$
$$\rightarrow K_2((Y\ M)(\lambda y{:}L\ \underline{1}\ y)\underline{3})\underline{2}$$
$$\rightarrow \underline{2}$$

Any general recursive functions can be rewritten in terms of μ^*, g and h where g and h are primitive recursive functions related as follows,

$$f(x_1,x_2,\ldots,x_n)=h(x_1,x_2,\ldots,x_n,(\mu^*(\lambda y{:}g\ x_1\ x_2\ \ldots x_n\ y\ )))$$

If H and G λ-defines g and h respectively, then F λ-defines f as follows,

$$F\leftarrow(\lambda x_1{:}\lambda x_2{:}\ldots{:}\lambda x_n{:}H(x_1\ldots x_n(\mu(\lambda y{:}G\ x_1\ldots x_n\ y))))$$

If we identify f as a while loop, then h is a for-loop and g is the function for testing the exit conditions, while μ^* is for finding the number of times the loop has to be executed before the conditions are satisfied.

1.2.4 Church's Thesis

It was stated that every general recursive function is λ-definable [Church,1941], [Kleene,1936]. It was also shown that every computable function is general recursive [Minsky,1967], [Turing,1937]. So it can be claimed that every computable function is λ-definable. This result is important to us because it partly justifies our decision in choosing the λ-Calculus for our investigation.

As part of the illustration of λ-definability, we have outlined the proof of the first part (i.e. general recursiveness is included in λ-definability).

CHAPTER TWO

THE RECKON LANGUAGE

In last chapter, it has been shown how λ-Calculus can be used in studying computable functions and processes of computing. However, it cannot be denied that it is quite painful to read long strings of symbols whose meaning is by no means obvious. It thus fails in one very important aspect. For, a computer program is not only an interface between human beings and machines but it also serves as a link of communication among human beings. A reasonable approach to this problem is that a language L can be used so that programs written in L can be translated mechanically into equivalent λ-expressions. Any legal λ-expression must be expressible in L. In finding such a language L, it can be seen that it is easier to satisfy the first requirement but not necessary the second one. For example, Algol-60 programs can be translated into λ-expressions, but it is not possible to have partial evaluation of a function in Algol-60.

Example

$(\lambda(x\ y\ z):...x...)u$ is reduced initially to

$(\lambda(y\ z):...u...)$

In the remainder of this thesis we shall have to express programs that would call for enormous λ-expressions yet for which the abbreviating conventions of Algol-60 or Algol-68 do not suffice, as we have just suggested. We

need a set of conventions of our own for abbreviating large λ-expressions. Reckon [Edwards,1974a] is the language that we claim that will satisfy all these requirements. It is not difficult to justify this statement because λ-expressions are legal expressions in Reckon so undoubtedly it will satisfy the second requirement. As all legal Reckon programs are computable so it must be expressible in λ- calculus [Edwards,1975], thus satisfying the first requirement. In the following sections, we give a brief account of this language.

2.1 Basic Objects

As a first step in simplifying λ-expressions, we may suggest that we shall consider those objects which occur throughout as constants, (these may include numbers, boolean values and others) and primitive functions over them. The rules of correspondence mapping the operator/ operand combinations into these objects may be given by the δ-rules mentioned in last chapter and used in the practical interpreter, or they may be αβη-defined as we do here.

2.1.1 Primitive values

(1) Integers are denoted by "1", "2", ... and are λ-defined by $\underline{1}$, $\underline{2}$, ... respectively

(2) Boolean values are denoted by TRUE and FALSE and as mentioned before, they are λ-defined by $\underline{1}$ and $\underline{0}$.

2.1.2 Primitive Functions

The use of fixed constants is not limited to "values" but can be extended to functions (operators) as well.

(1) Arithemetical operations : "+", "-" and "*" stand for addition, subtraction and multiplication respectively and are λ-defined in last chapter.

(2) Logical operations : "NOT", "AND" and "OR", and are λ-defined as follows,

NOT←(λx:(C x)$\underline{1}$ $\underline{0}$)

AND←(λx:λy:(C x)$\underline{0}$ y)

OR←(λx:λy:(C x)y $\underline{1}$)

Examples

(1) NOT(TRUE)=FALSE

$(\lambda x:(C\ x)\underline{1}\ \underline{0})\underline{1}\rightarrow(C\ \underline{1})\underline{1}\ \underline{0}\rightarrow K_2\ \underline{1}\ \underline{0}\rightarrow\underline{0}$

(2) AND TRUE FALSE=FALSE

$(\lambda x:\lambda y:(C\ x)\underline{0}\ y)\underline{1}\ \underline{0}\rightarrow(C\ \underline{1})\underline{0}\ \underline{0}\rightarrow K_2\ \underline{0}\ \underline{0}\rightarrow\underline{0}$

(3) AND FALSE TRUE=FALSE

$(\lambda x:\lambda y:(C\ x)\underline{0}\ y)\underline{0}\ \underline{1}\rightarrow K_1\ \underline{0}\ \underline{1}\rightarrow\underline{0}$

(4) OR TRUE FALSE=TRUE

$(\lambda x:\lambda y:(C\ x)y\ \underline{1})\underline{1}\ \underline{0}\rightarrow(C\ \underline{1})\underline{0}\ \underline{1}\rightarrow K_2\ \underline{0}\ \underline{1}\rightarrow\underline{1}$

(5) OR FALSE FALSE=FALSE

$(\lambda x:\lambda y:(C\ x)y\ \underline{1})\underline{0}\ \underline{0}\rightarrow(C\ \underline{0})\underline{0}\ \underline{1}\rightarrow K_1\ \underline{0}\ \underline{1}\rightarrow\underline{0}$

(3) Relational operations : "<", ">" and "=", and they are λ-defined by "L", "GR" and "EQ" respectively as follows,

$L\leftarrow(\lambda x:\lambda y:MIN\ \underline{1}(S_0\ y\ x))$

$GR\leftarrow(\lambda x:\lambda y:MIN\ \underline{1}(S_0\ x\ y))$

$EQ\leftarrow(\lambda x:\lambda y:\underline{1}\ \underline{\cdot}(A(GR\ x\ y)(GR\ y\ x)))$

Examples

(1) 2>3=FALSE

$GR\ \underline{2}\ \underline{3}\rightarrow MIN\ \underline{1}(S_0\ \underline{2}\ \underline{3})\rightarrow MIN\ \underline{1}\ \underline{0}\rightarrow\underline{0}$

(2) 2=2 = TRUE

$EQ\ \underline{2}\ \underline{2}\rightarrow\underline{1}\ \underline{\cdot}(A(GR\ \underline{2}\ \underline{2})(GR\ \underline{2}\ \underline{2}))$

$\rightarrow\underline{1}\ \underline{\cdot}(A\ \underline{0}\ \underline{0})\rightarrow\underline{1}$

2.2 Lists

It has been stated that in λ-Calculus a function of n arguments can be expressed as a function of one argument whose value is a function of (n-1) arguments and so on. This tedious treatment can be avoided by using "Lists". Lists can consists of nothing (i.e. NULL-lists) or can be constructed by pairing heads that are anything and tails that are lists. We shall use the symbol "," to separate the two parts of a list and "NIL" is used to stand for NULL-list.

Example

(2,(TRUE,NIL)) is a list

We can assume that "," is right associative so that (a,(b,c)) is equivalent to (a,b,c). The notation can be simplified further by omitting all NIL appearing as the last element of a list. Beware that this is only a convention so that "," does not denote the pairing operation.

Example

(2,(TRUE,NIL)) can now stated as

(2,TRUE)

In future in order to avoid any confusion, all NIL's at the end of a list have to be removed according to the above rules. So that (2,(3)) is constructed by the following process.

```
(construct-list 2 (construct-list(construct-list 3 NIL)NIL))
```

assuming that construct-list is the function for constructing a pair.

It remains to replace any function of n arguments by a function of one argument which is a list of n elements.

Example

(λ(x,y,z): ...) is a function of 3 arguments.

Lists can be nested within each other so in Reckon we allow very complicated structures in binding variables.

Example

$$(\lambda(x_1,(x_2,(x_3,x_4),x_5),x_6): \ldots)$$

If a list L is constructed from objects that are λ-defined by A and B (B is the λ-definition of some other lists), then L can be λ-defined by "(λf:f A B)".

Lists not only occur as binding variables but also occur in other contexts which are logically correct. Some of the list-manipulation functions are discussed below,

(1) "NULL"

This is the function for testing NULL-list. In other words it tests whether the object under consideration is "NIL" or not. Suppose "NIL" is λ-defined by $(\lambda x:\underline{1})$ then NULL will be λ-defined by $(\lambda f:f(\lambda a:\lambda b:\underline{0}))$.

Examples

$$\text{NULL NIL} \rightarrow (\lambda f:f(\lambda a:\lambda b:\underline{0}))(\lambda x:\underline{1})$$
$$\rightarrow (\lambda x:\underline{1})(\lambda a:\lambda b:\underline{0})$$
$$\rightarrow \underline{1}$$

$$\text{NULL}(\lambda f:f\ a\ b) \rightarrow (\lambda f:f(\lambda a:\lambda b:\underline{0}))(\lambda f:f\ a\ b)$$
$$\rightarrow (\lambda f:f\ a\ b)(\lambda a:\lambda b:\underline{0})$$
$$\rightarrow (\lambda a:\lambda b:\underline{0})\ a\ b$$
$$\rightarrow \underline{0}$$

Obviously, if "NIL" is λ-defined by other well-formed formulae, then the λ-definition of "NULL" has to be changed accordingly.

(2) "TH","TL"

These are list selectors such that, if $i \leq m$

$$i\ TH\ (n_1,n_2,\ldots,n_m)=n_i$$

$$i\ TL\ (n_1,n_2,\ldots,n_m)=(n_{i+1},\ldots,n_m)$$

and they can be λ-defined by "th" and "tl" respectively as follows,

$tl \leftarrow (\lambda n:\lambda x:n(\lambda z:z\ K_2)x)$

$th \leftarrow (\lambda n:\lambda x:(tl(P\ n)x)K_1)$

Example

$th\ \underline{2}\ (\lambda f:f\ A(\lambda g:g\ B\ C))$

$\rightarrow tl\ \underline{1}\ (\lambda f:f\ A(\lambda g:g\ B\ C))K_1$

$\rightarrow \underline{1}(\lambda z:z\ K_2)(\lambda f:f\ A(\lambda g:g\ B\ C))K_1$

$\rightarrow (\lambda z:z\ K_2)(\lambda f:f\ A(\lambda g:g\ B\ C))K_1$

$\rightarrow (\lambda f:f\ A(\lambda g:g\ B\ C))K_2\ K_1$

$\rightarrow K_2\ A(\lambda g:g\ B\ C)K_1$

$\rightarrow (\lambda g:g\ B\ C)K_1$

$\rightarrow K_1\ B\ C$

$\rightarrow B$

As in contrast to LISP [McCarthy,1962], the selector functions here do not test whether their argument is NULL-list or not. We shall see later, as example of conditional branchings, how they can be modified to include such tests.

(3) "..", "::"

Both are list constructors such that

$$(a_1,a_2)..(a_3,a_4)=((a_1,a_2),a_3,a_4)$$

$$(a_1,a_2)::(a_3,a_4)=(a_1,a_2,a_3,a_4)$$

and they can be λ-defined by "CONS" and "CONCAT" respectively as follows,

CONS←(λa:λb:λc:c a b)

CONCAT←(λx:λy:G_2 x y (μ(λz:NULL(z TL x))0))

where,

G_2←(λx:λy:λn:REC y (G_1 n x) n), and

G_1←(λn:λx:λy:λz:CONS((n-y) TH x) z)

Example

assuming that "d" is a list,

CONCAT (λf:f a(λg:g b NIL)) d

→G_2 (λf:...) d 2

→REC d (G_1 2 (λf:...)) 2

→(G_1 2 (λf:...)) 1 (REC d (G_1 2 (λf:...)) 1)

→(λz:CONS((2-1)TH(λf:...))z)(REC d (G_1 2 (λf:...)) 1)

→CONS(1 TH (λf:...))(REC d (G_1 2 (λf:...))1)

→CONS a (CONS b d)

→(λc:c a (λc:c b d))

2.2.1 Strings

Strings are included in Reckon as one of the system defined types but the potential uses of this data type have not been fully explored and in the present CDC computer implementation of Reckon, manipulations are limited to selections, tests of nullness, and concatenations only. Bearing this in mind and also to simplify our discussions hereafter, we shall consider strings as special cases of lists. This decision is well justified for our purposes with regarding to the logical similarities of their structures and the operations they share with each other. Notationally, we shall enclose a string by "<" and ">", such as "<strings>".

2.3 Assignments

At first sight, we may λ-define "...; x:=x'; ..." by "...; (λx:...)x' ". The effect of which is to enforce a new entry of x in the current environment. Hence, the original entry of x will be left unchanged. Consequently it will not be possible to have the so-called "side effects" of procedures. In Reckon, we regard assignments as modifications on execution environments, so it can be λ-defined as follows--(λ(e,x,y):(update(search x e)y)), where "search" is the routine for finding x in e and "update" changes the value associated with x to y. Notice that it is not necessary for x to be a simple variable, for example, it can be "i TH L", where L is a list.

2.4 Sequential expressions

Let SEQUENT $S_1\ S_2 = S_1;S_2$, then SEQUENT can be λ-defined by $(\lambda a:\lambda b:K_2\ a\ b)$.

Example

$$S_1;S_2;S_3 = (\text{SEQUENT}(\text{SEQUENT}\ S_1\ S_2)S_3)$$
$$(\lambda a:\lambda b:K_2\ a\ b)((\lambda a:\lambda b:K_2\ a\ b)S_1\ S_2)S_3$$

Assuming that the order of execution is from left to right, then when the expression is β-reduced S_1, S_2 and S_3 will be executed in that order. Moreover, the result of execution is that of S_3.

2.5 Call by Name and Call by Value

Generally, actual parameters or actuals can be handled in two ways,

(1) The actuals are evaluated and values obtained are then assigned to the formals. This process is known as "call by value".

(2) The actual parameters are not evaluated so it is unevaluated expressions that are assigned to the formals and this does not exclude the case that the actual is a variable too.

In Reckon, parameters are called by value. However the effect of call by name can be achieved by suitable λ-

abstraction of the actual parameters. Suppose v is the actual that we wish to be called by name, then by suitable λ-abstraction we mean that we have to abstract v with an arbitrary binding variable that does not occur free in v. Conventionally we use "()" for that arbitrary binding variable.

Example

.	(λn: ...n...)v	is called by value
	(λn: ...n()...)(λ():v)	is called by name

The trick here is to take advantage of the ordering rule by which (λ():v) will not be evaluated until it appears in the operator-part of a combination. So the final effect is that n is assigned the "value" (λ():v)-- a process for evaluating v. Inside the λ-body, we have to write "n()" instead of "n" to enforce execution of (λ():v). We can see that the combination "n()" is quite arbitrary. In fact, we can replace () by anything and can still achieve the same effect.

Let us design an abbreviating convention. Recalling that we have value specification in Algol-60, and name specification in Reckon as just described, it appears open to us to choose our notation either way. We may wish to declare in a λ-expression that its formal calls by name so that subsequently programmers may discarded all () that are used for the purpose of "call by name" and rely on a syntax analyser to insert all omitted (), thus leading to

a much simplified notation .

Had parameter passing in Algol-60 been handled by "call by value" (as in Reckon), then presumably Algol-60 would have had name declaration as follows,

```
PROCEDURE f(n);
NAME n; BEGIN ...n... END;
... f(v) ...
```

In λ-notation, this will be

```
(λf:...(f v)...)(λNAME n:...n...)
```

after β-reduction, we have

```
...(λNAME n:...n...)v...
```

We assume that NAME occurring immediately after λ indicates a NAME declaration. By having the name declaration mechanism, the following two statements are equivalent.

(1) (λn:...n()...)(λ():v)

(2) (λNAME n:...n...)v

But this idea has not been implemented in any versions of Reckon so that some work would still have to be done before

the idea could be materialized. Currently Reckon relies on explicit "()" notations.

2.6 Conditional Branchings

For simple programs, it is not difficult to predict in advance each step of evaluation and arrange them in sequential order. However in more complicated programs, it is desirable to change course of evaluation upon occurrences of certain events in them. For example, not to divide a number if the divisor is zero. We shall write "conditional expressions" whose alternative paths are called "conditional branches". It is worth mentioning that the alternative paths in conditional expressions will not be executed until after the "condition" has been evaluated. In Reckon, conditional expressions can be expressed as

$$\text{IF } exp_b \text{ THEN } exp_1 \text{ ELSE } exp_2 \text{ FI}$$

where exp_b is some expression which will yield Boolean values when computed, while exp_1 and exp_2 are arbitrary expressions. "IF", "THEN", "ELSE" and "FI" are reserved words in Reckon. One may replace "IF" and "FI" by a pair of brackets.

Let COND $exp_b(\lambda():exp_1)(\lambda():exp_2)=$

IF exp_b THEN exp_1 ELSE exp_2 FI

then COND can be λ-defined as $(\lambda b{:}\lambda x{:}\lambda y{:}\ C\ b\ y\ x\ ()\ )$.

Example

th and tl can now be re-defined so as to test whether their argument is the NULL-list or not.

th'←(λn:λx:COND(NULL x)(λ():ERROR)(λ():th n x))

tl'←(λn:λx:COND(NULL x)(λ():ERROR)(λ():tl n x))

where ERROR is a system- defined function for handling error conditions.

It is not excluded that exp_b, exp_1 and exp_2 may be sequential expressions or conditional branchings, provided exp_b yields Boolean results.

Example

IF print x; x=3 THEN $s_1;s_2$ ELSE $s_3;s_4$ FI

2.7 Choice Expressions

Another way to express a nested conditional expression is by "Choice Expression", whose general format is

CASE exp_n IN exp_1, exp_2, ..., exp_q OUT exp_0 ESAC

which can be λ-defined as

(λn:COND(0≥n>q)(λ():exp_0)(n TH((λ():exp_1),...,(λ():exp_q))))

"CASE", "IN", "OUT" and "ESAC" are reserved words in the language. One may replace "CASE" and "ESAC" by a pair of brackets. Any exp_n must be an expression which yields a numerical value when executed. Suppose i is the result and if $1 \leq i \leq q$, then exp_i will be executed otherwise exp_0. The discussions on the different possibilities of exp_1, exp_2 and exp_b above are applicable here for exp_1, ..., exp_q, exp_0 and exp_n respectively with some trivial adjustments. The relationship between conditional expressions and choice expressions can be illustrated by the example below,

Example

IF a_1 THEN b_1 ELSE IF a_2 THEN b_2 ELSE b_3 FI FI

is equivalent to

CASE n IN b_1,b_2 OUT b_3 ESAC

assuming that n is 1 or 2 depending whether a_1 or a_2 is true, and in cases both a_1 and a_2 are false then n is neither 1 or 2.

The user can omit ELSE branches or OUT choices completely. So the following expressions are also correct.

(1) IF exp_b THEN exp_1 FI

(2) CASE exp_n IN exp_1,exp_2 ESAC

2.8 LET and WHERE expressions

In writing "(λx:x+x)3", what may be meant possibly is "let x be 3; now evaluate the expression (x+x)". It is felt that we may as well make this meaning more explicit by having notations such that the above example can be re-written as "LET x≡3; x+x".

Example

(λsq:sq 2)(λn:n*n) can be rewritten as

LET sq n≡n*n; sq 2

One may regard the statement enclosed by "LET" and ";" as the definition part of a function and in this respect we may find let-notation more natural to us than λ-notation. Syntactically, there are some differences between the following two statements,

(1) LET f a b c≡a b c; ...

(2) LET f(a,b,c)≡a b c; ...

The differences can be seen quite easily if we rewrite them in λ-notation.

(1) (λf:...)(λa:λb:λc:a b c)

(2) (λf:...)(λ(a,b,c):a b c)

It is true that the two expressions will give the same result when applied to appropriate arguments. Differences arise only where partial application is contemplated.

There are a few variations in writing let-expressions. They are listed below with the corresponding λ-expressions to which they are mapped by Reckon implementations.

(1) LET $x_1,x_2,\ldots,x_n \equiv a_1,a_2,\ldots a_n$; ... is equivalent to
$(\lambda(x_1,x_2,\ldots,x_n):\ldots)(a_1,a_2,\ldots a_n)$

(2) LET $x_1 \equiv a_1$ AND $x_2 \equiv a_2$ AND ... AND $x_n \equiv a_n$; ...
is equivalent to (1)

(3) Any combinations of the different variations are allowed.

(4) Recursive function definitions will be considered later however.

In adopting let-notations, incidentally, we require that all functions and variables have to be defined prior to their uses. This feature is shared by ALGOL-like languages in which we have to define all procedures before they are used. On the other hand in FORTRAN, definitions of all subroutines are placed after the main program, so they are defined after the uses. In Reckon, we also allow this form of programming. Post-definition mechanism can be incorporated into the language by having where-notation.

Examples

(1) x+x WHERE x≡3;

(2) sq 2 WHERE sq n≡n*n;

It is not uncommon to wish to express ideas in a number of different ways in human languages. It is hoped that in providing certain flexibilities in Reckon, we may bring it closer to users' wishes. Program writing is by no means just an exercise of grammatical rules, after all. It is an art of expressing one's ideas. In common with other arts, it is no good just playing with rules, we have to explore the nature of it and develop our skill in it. Thus it is of utmost importance that a language should have enough depth for such developments. There is no doubt that strict rules and unnecessary restrictions would destroy art completely. Conversely, variations in a language may stimulate experiments from users.

2.9 Recursion

If a function f is described so that the same f description is needed as part of the description then f is said to be defined by recursion.

Example

Factorial function might be defined as

```
LET factorial n≡IF n=0 THEN 1
                    ELSE n*factorial(n-1) FI;
```

and in λ-notation, this can be

```
(λfactorial:...)(λn:IF n=0 THEN 1 ELSE
                        n*factorial(n-1) FI)
```

It may be noted that "factorial" is free in "(λn:IF n=0 THEN...FI)" and this may cause errors in execution. In order to avoid it, we have to earmark a definition if it is recursive. We may do this by the markers "REC" and "LABEL".

Examples

```
LET REC factorial n≡
    IF n=0 THEN 1 ELSE n*factorial(n-1) FI;
```

which is equivalent to

```
(LABEL factorial:λn:
    IF n=0 THEN 1 ELSE n*factorial(n-1) FI)
```

"LABEL ..." is expanded by syntax analyzer to "Y ..." and "Y" was defined to have the property as described in chapter 1.

Examples

for-loops can be defined by

```
LET REC for i j k s≡
    IF i≤k THEN s(); for(i+j) j k s FI;
```

so that "for 1 1 10 (λ():print<string>)" will print the string of characters ten times.

Similarly, while-loops can be defined by

```
LET REC while p s≡
    IF p() THEN s(); while p s FI;
```

so that "while (λ():i≤10)(λ():print<string>; i:=1+i)" will produce the same effect as the for-loop, assuming that i is bound in outer expression and ":=" stands for assignment.

2.10 Polymorphic Operations

There are arguments for supposing that distinct symbols should be used for operation on operands of different types irrespective of logical similarities. For example, different symbols should be used for integer-add, real-add and so on for the other arithemetical operations [Laski,1968]. As far as "computer men" are concerned, this is a splendid idea because there would be less work in type-checking and related problems. However most people using machines are problem-solvers and, undoubtedly, will make less mistakes if they can "talk" in their "own language". For example, mathematicians like to use the same symbol not only for integer or real addition but even for matrix-, vector-, array-, complex number- addition, and even operations over determinants, characters, strings, booleans and etc. We can imagine the confusions that might arise in a mathematical book if one symbol were used for adding two matrices and another for adding its elements depending on what type of element they are. The problem here is not that mathematicians do not

know what they are doing but the tedious treatments that would otherwise be involved. In a certain sense this is a moral question of who is in charge, language implementors or users?

In Reckon, we do not regard such "polymorphic" operators as sinful. In fact, we have gone even further to the extent that where there is no ambiguity, some operators can be dropped completely [Edwards,1974b].

Examples

```
2 <S T R I N G>=2 TH <S T R I N G>
2 (S,T,R,I,N,G)=2 TH (S,T,R,I,N,G)
2 3=2*3
<S T R><I N G>=<S T R>::<I N G>
(1,2,3)(4,5,6)=(1,2,3)::(4,5,6)
x'=x-1   if x is an integer
x'=1 TL x   if x is a list or string
```

Some of these simplifications can be regarded as observations on our writing habits and instinctive reactions towards certain notations. For example, in placing <S T R><I N G> together as it is shown, naturally we have a feeling of grouping them together (if readers do not agree with this, then they may have more disagreement with ALGOL-68 in which identifier "feel bad" is regarded as "feelbad"). And in Arithmetic, it is conventional to write 2(3+4) for 2*(3+4).

Of course, one has to be careful in allowing such

shorthand notation in a language. There are still numerous questions to be answered. Are we abetting bad programming habits which are more evil than polymorphic operators? How far can we go in this direction? Further discussion is limited by the scope of this project, so we shall let these questions remain as questions. We mention the matter only by way of introduction to the next chapter.

2.11 Implementation

There are a few versions of type-free Reckon implemented on CDC 66/6400 at ULCC. Earlier versions of Reckon are written in LISP, later versions are written in Pascal. One of the Pascal versions formed the basis of the typed-Reckon systems whose logic we shall now discuss.

Appendix A

Programming Examples of Reckon

We shall use Reckon very often in later chapters, so readers are recommended to familiarize themselves with the language from the examples present here for this purpose.

The followings are reprints of the computer outputs from CDC 6400. Unfortunately not all the characters we use in this thesis are available on the CDC machine. We list below the differences that will affect us in this appendix (as well as in appendices C and E)

character used in this thesis	CDC 6400
λ	$
∪	!
∩	&
≤	@
≥	\
≡	#

Comments are enclosed by "COMMENT" and "COMMENTEND" in the listings. A new instruction "print" is used in the examples, which is just ordinary I/O instruction and the information printed will be in the format

<<<OUTPUT IS : ...>>>

EXAMPLE 1

```
START
BEGINNING

COMMENT
DEFINE "DIVIDE" AND "DIVISIBLE"
COMMENTEND

LET REC DIV M N A B#
  IF M\N THEN DIV(M-N) N (A+1) B ELSE (A,M) FI;
LET DIVIDE M N#DIV M N 0 0;
LET DIVISIBLE M N#IF(2TH(DIVIDE M N))=0
  THEN PRINT<DIVISIBLE> ELSE PRINT<NOT DIVISIBLE> FI;
PRINT(DIVIDE 18 5);
(DIVISIBLE 24 6)
ENDING
FINISH

<<<OUTPUT IS : [3,3] >>>
<<<OUTPUT IS : <DIVISIBLE> >>>

RESULT OF PROGRAM IS : <DIVISIBLE>
QED.
```

EXAMPLE 2

```
START
BEGINNING

COMMENT
DEFINING A DO-LOOP
COMMENTEND

LET DOLOOP V N S#
  (V:=1;
  LET REC G#($():IF V\N THEN () ELSE
    S(); V:=V+1; G() FI);
  G() );
LET X#0;
DOLOOP 0 6 ($():X:=X+1);
PRINT X
ENDING
FINISH

<<<OUTPUT IS : 5 >>>

RESULT OF PROGRAM IS : 5
QED.
```

EXAMPLE 3

```
START
BEGINNING

COMMENT
MAP F(X1,X2,...,XN)=(F X1,F X2,...,F XN)
COMMENTEND

LET REC MAP F X#IF NULL X THEN ()
  ELSE F(X.1)..MAP F X' FI;
MAP($X:X*X)(1,2,3,4,5)
ENDING
FINISH

RESULT OF PROGRAM IS : [1,4,9,16,25]
QED.
```

EXAMPLE 4

```
START
BEGINNING

COMMENT
THE ARGUMENTS OF GENLIST CAN BE INTERPRETED AS FOLLOWS:
"X" IS A LIST OR STRING
"F" IS THE FUNCTION TO BE APPLIED TO EACH ELEMENT OF "X"
"A" IS THE VALUE PRODUCED WHEN X IS NULL
"G" IS USED TO COMBINE THE INDIVIDUAL RESULTS
COMMENTEND

LET REC GENLIST A G F X#IF NULL X THEN A ELSE
  G(F(X.1))(GENLIST A G F X') FI;
LET COMBINE,IDENT#($X:$Y:X::Y),($X:X);
PRINT(GENLIST < > COMBINE IDENT (<C>,<O>,<M>,<P>,
  <U>,<T>,<E>,<R>));
COMMENT GENERATE STRING FROM LIST COMMENTEND
PRINT(GENLIST () ($X:$Y:X..Y) IDENT <MACHINE>);
COMMENT GENERATE LIST FROM STRING COMMENTEND
PRINT(GENLIST 0 ($X:$Y:X+Y) ($X:X*X) (1,2,3,4))
COMMENT SUM OF SQUARE COMMENTEND
ENDING
FINISH

<<<OUTPUT IS : <COMPUTER > >>>
<<<OUTPUT IS : [<M>,<A>,<C>,<H>,<I>,<N>,<E>] >>>
<<<OUTPUT IS : 30 >>>

RESULT OF PROGRAM IS : 30
QED.
```

PART TWO

INTRODUCTION

Various reasons have been offered for including types in algorithmic languages, including the following :

(1) Programs written in typed languages are easier to debug.

(2) Errors can be detected as early as possible if type information is available so that time would not be wasted in executing erroneous programs.

(3) Type information can be used to produce better code, for example, polymorphic operations can be replaced by the apposite routine and coercion operators can be inserted whenever necessary.

(4) Objects can be represented more economically in machines so that storage space can be minimized.

(5) Programmers need not have to worry about machine representation of their data.

In our project, we are more concerned with (2). The process of detecting errors in programs based on type information is known as type-checking. Type-checking performed at compile time is known as static type-checking, while that performed at run time is known as dynamic type-checking.

It will be more fruitful if type-checking process is studied independent from particular languages. This can be achieved by having a type-checking system so that each computing element can be projected into a corresponding type element by a mapping function. The type-checking system is then independent from computing languages. The relation between a language L_i and a type-checking system T_j may be defined by a mapping function Φ_{ij}.

A type-checking system is characterized by its elements and the operations that are defined on these elements.

In next chapter, we shall examine three type-checking systems proposed by Ledgard, Hext and Morris respectively, which will be followed by discussions on the two problems that cannot be solved by them.

From our studies on these problems, we found that some fundamental concepts have been overlooked by these systems, and the concepts will be discussed in chapter 2.

We propose our system in chapter 3. Of course it is not designed specifically for those problems, but we think we are obliged to suggest some solutions to them from our system. A more general system will not be proposed until we reach Part Three of this thesis.

CHAPTER ONE

THREE SIMPLE SYSTEMS

This chapter is intended to give a brief literature survey on some recent work in type-checking. The problems that are experienced by these systems will be presented in the last section of this chapter.

1.1 System L

It is a well known techique to prove certain properties of computer programs by mapping them into some isomorphic mathematical system. For example, we may use predicate Calculus as a mathematical model to prove correctness and equivalence of programs [Manna,1970]. However, it is not our aim here to find out which existing model is adequate to representing a type-checking system. We are only interested in the properties and characteristics of type-checking systems which are suitable for functional languages. Consequently, we shall not be embarrased if a known model may already exist which satisfies our requirements. But until we have established the necessary attributes, there shall be no way that we can identify or construct a suitable model.

The system proposed in [Ledgard,1972] is modelled on a subset of the set of expressions in λ-Calculus. Type-constants such as [INTEGER], [INTEGER→INTEGER] and others are treated as constants in the Calculus. There is

a special constant called [ERROR] which whenever present in any part of a type expression indicates that there is a mistake in the type combinations and consequently a mistake in the program.

A mapping function Φ is used to map each computing expression into a corresponding type expression.

Examples

(1) Φ(3)=[INTEGER]

(2) Φ(TRUE)=[BOOLEAN]

Primitive operators (functions) are mapped into type-constants which are pre-defined.

Examples

(1) Φ(NOT)=[BOOLEAN→BOOLEAN]

(2) Φ(OR)=[[BOOLEAN,BOOLEAN]→BOOLEAN]

(3) Φ(+)=[[INTEGER,INTEGER]→INTEGER]∪
[[REAL,REAL]→REAL]∪
[[INTEGER,REAL]→REAL]∪
[[REAL,INTEGER]→REAL]

(which we shall abbreviate to t_+)

Three type constructors are introduced in the examples above. They are "$\rightarrow$", "$\cup$" and "," and the types constructed are known as functional, union and tuple types respectively.

Computing variables are mapped into "pseudo type-variables". Since the type of all computing variables has to be declared prior to use, so, effectively, the value of each "pseudo type-variable" is fixed by the mapping. To comply with the rules of λ-Calculus, declaration, such as [INTEGER]x, is mapped into the following construct

$$(\lambda x: \ldots)[\text{INTEGER}]$$

there should be no confusion if we use the same notation for computing variables and "pseudo type-variables".

Example

$$\Phi([\text{INTEGER}]x;x) = (\lambda x:x)[\text{INTEGER}]$$
$$= [\text{INTEGER}] \quad (\text{by } \beta\text{-reduction})$$

Type expressions must be well-formed formulae in λ-Calculus and they are defined recursively as

(1) all type-constants are type-expressions

(2) all "pseudo type-variables" are type-expressions

(3) if t_i, t_j are type expressions, then the combination $(t_i\ t_j)$ is also a type-expression

(4) if t_i is a type expression and x is a "pseudo type-variable", then $(\lambda x{:}t_i)$ is a type-expression.

Type expressions can be reduced either by β-reduction or, in the case that they are combinations of type-constants, by the following reduction rules (all t, with or without subscripts, are any type expressions unless stated otherwise)

(RL1) $[t_i \to t_j]t_k$ is reduced to t_k if $t_i = t_k$ otherwise [ERROR]. Note that if $t_i=(t_{i1},t_{i2},\ldots,t_{in})$ and $t_j=(t_{j1},t_{j2},\ldots,t_{jm})$ then $t_i=t_j$ iff n=m and for every k, $1\le k\le n$, $t_{ik}=t_{jk}$.

(RL2) $[[t_{11}\to t_{12}]\cup[t_{21}\to t_{22}]\cup \ldots \cup[t_{n1}\to t_{n2}]]t_k$ is reduced to t_{j2} if there exist j ($1\le j\le n$) such that $t_{j1}=t_k$ otherwise [ERROR].

(RL3) [ERROR]t is reduced to [ERROR].

(RL4) t[ERROR] is reduced to [ERROR].

It may be assumed that all infixed operations, such as (x op y), are transformed into prefixed notation, say op(x,y), before the mapping function is applied to them.

Example

$$\Phi([\text{INTEGER}]x;x+3)=(\lambda x{:}t_+(x,[\text{INTEGER}]))[\text{INTEGER}]$$
$$=t_+([\text{INTEGER}],[\text{INTEGER}])$$
$$=[\text{INTEGER}] \quad \text{by rule (RL2)}$$

It could be much easier to understand the reduction rules if we λ-defined the functional types instead of regarding them as constants.

Examples

(1) $[t_i \to t_j]$ is λ-defined by

$(\lambda x$:IF $x=t_i$ THEN t_j ELSE [ERROR] FI)

thus, $[t_i \to t_j]t_k =$

$(\lambda x$:IF $x=t_i$ THEN t_j ELSE [ERROR] FI)t_k,

which , when β-reduced, yields either t_j or [ERROR]. We may therefore say that (1) formalize rule (RL1).

(2) $(t_1,t_2,\ldots,t_n)\to t_{n+1}$ is λ-defined by

$(\lambda(x_1,x_2,\ldots,x_n)$:

IF$(x_1=t_1)$AND$(x_2=t_2)$AND...AND$(x_n=t_n)$ THEN

t_{n+1} ELSE [ERROR] FI)

(3) $[t_{11}\to t_{12}]\cup[t_{21}\to t_{22}]\cup\ldots\cup[t_{n1}\to t_{n2}]$is λdefined

$(\lambda x$:IF $x=t_{11}$ THEN t_{12} ELSE IF $x=t_{21}$ THEN t_{22}

ELSE IF ... ELSE IF $x=t_{n1}$ THEN t_{n2}

ELSE [ERROR] FI...FI FI)

Hence, (2) and (3) formalize the corresponding rules.

Summarizing now, we notice that the types of all variables have to be declared. We have discussed various constructed types and how type expressions are formed. Type expressions are reduced according to reduction rules. We have suggested the semantics of these rules by λ-defining the functional types. As System-L is a subpart of λ-Calculus, so type-checking here is a process which reduces λ-expressions to their normal form, and in this case, the type-constants. Since type expressions must be well-formed formulae too, so a mapping function is required to map computing expressions into corresponding well-formed formulae.

We can see from the simplicity of system-L how it is an advantage to use an established model as the basis for a type-checking system. Nevertheless, it is admitted by Ledgard that his system cannot handle parametric polymorphism (to be explained later) and other problems. Accordingly, we shall come back to these problems in later sections.

1.2 System-H

Hext's system[Hext,1966] was not defined exactly in the format we require. This is not surprising because his paper described a practical approach to the problem. We shall therefore read his work with an eye to his contribution to general theory.

His System-H is defined by four functions, TYM, TYPE, UPTYPE and SETTYPE. TYM is his main routine while the others are its subroutines. TYM is defined recursively. SETTYPE maps primitives (i.e. variables and constants) in the program to elements in the type-checking system. "TYPE(E)" finds the type of expression E; for example, if "E" is "2+3", then "TYPE(2+3)" is [INTEGER]. "UPTYPE(E,t)" proves whether or not expression E can have type t (maybe after coercion). This is to say UPTYPE(E,t) can be rewritten as "EQUAL(t,TYPE(E))", assuming that coercion will be applied whenever necessary.

The reader may relate TYPE and UPTYPE to the reduction mechanism described in System-L.

Higher ordered types are available to describe types of functions (or procedures), the general format of which is [D→R], where D is the type of domain which can be primitive or higher ordered types in turn. "R" is the type of range which must be primitive. The restriction means that a function cannot produce function as result. This is a restriction that we aim to remove.

Structured types are also allowed. The general format of these is $(t_1,t_2,\ldots,t_n)$ where t_i are types. For example lists can be described by structured types. This data type is not particularly relevant to our discussion here, so we shall not pursue it any further.

Two special type-constants, "GENERAL" and "UNKNOWN" are included. One can consider "GENERAL" as the union of all types. But the semantic of "UNKNOWN" is not mentioned in Hext's paper. It seems to us that it is quite logical to interpret "UNKNOWN" as an intersection of all types (most probably it will be empty). From a set-theoretical point of view, one may regard "GENERAL" and "UNKNOWN" as the universal set and empty set respectively. However, any definition of universal set as "the set of all sets" offers us little help in understanding its property, nor is the name "empty set" particularly meaningful to us. When viewed as special elements of a lattice (with partial ordering $\geq$), as is the case in System-H, "GENERAL" and "UNKNOWN" are regarded as the top-most and bottom-most elements of the lattice in the sense that for any element e in the lattice, "GENERAL"$\geq$e$\geq$"UNKNOWN" is always true.

Now, we try to construct a type-checking system for Hext, following the guidelines we have set down in the opening chapter of this part.

(1) The elements of the system include all primitive types in the language and functional types $[t_1 \rightarrow t_2]$ for

elements t_1, t_2 already in the system.

(2) Some type-constants constructed in (1) will be assigned to primitive operators and functions (for example,+).

(3) To complete the set of type-constants, we have to include "GENERAL" and "UNKNOWN".

(4) The reduction mechanisms will be those defined by routines TYPE and UPTYPE.

So far we regard System-H as similar to System-L except for elements "GENERAL" and "UNKNOWN". The following are a list of remarkable differences,

(1) The primitive types in System-H are partially ordered. This means that the ordering does not necessarily hold for any two arbitrary types. Generally, when $t_i \geq t_j$, we say that t_i is more defined than t_j. This shall be interpreted to mean that objects of type t_j can be represented by objects of type t_i. Therefore we completely agree that such a relation should not hold between any two arbitrary types, otherwise the following expression will be regarded as legal--"3+FALSE"--which may not be very desirable. In his subsequent treatment it proved convenient to extend his partial ordering to a lattice.

However, we are disappointed that partial ordering is not defined for functional types (i.e. types constructed by "→") in system-H and in fact in most type-checking systems. In the next chapter, we shall prove partial ordering does exist among functional types (in the sense of inclusion). Incidentally, the absence of partial ordering among functional types in most type-checking systems indicates that the properties of functional types have not been properly studied. So it is not surprising that these systems are not suitable for functional languages.

(2) Types of conditional expressions :

The type of the expression "IF b THEN x ELSE y FI" is $t_x \cup t_y$ where t_x, t_y are types of x and y respectively. The type expression $t_x \cup t_y$ is read as "the union of types t_x and t_y", and b stands for the Boolean expression which yields wither TRUE or FALSE.

In general, $t_1 \cup t_2 = t_1$ if $t_1 \geq t_2$, otherwise $t_1 \cup t_2$="the least upper bound of t_1 and t_2" which might be "GENERAL" in a lattice. It seems to us that in having "GENERAL" in a type-checking system, in some cases, run-time type-checking is inevitable unless we have some method to suppress it.

(3) Treatment of Recursive definitions :

In the example "REC f [INTEGER]n≡IF n=0 THEN 1 ELSE n*f(n-1) FI", one may readily assign or deduce types

of every object in the right side of the definition (let it be e) except f. In order to perform type-checking, Hext's system will assign a first approximation type to f, which in the absence of any knowledge can only be [UNKNOWN]; call it t_0. Let t_i be the type obtained for f in the i' th iteration of type deduction and TYPE be the process for determining the type of "e", then t_{i+1} is obtained as follows:

$$t_{i+1}=t_i \cup TYPE(e)$$

The series of iteration will terminate if there exists n such that $t_n=t_{n-1}$. The termination can only be guaranteed if there exist an upper bound for every path in the lattice, which is why Hext was forced to adopt a lattice model of types. There are thorough discussions on such type deductions and the conditions for their termination in [Tenenbaum,1974].

(4) Coercion among functional types

Coercion is outside the scope of our discussion. We mention it here because what is described in Hext's paper is quite interesting. Suppose f is a formal parameter of type $t_1 \rightarrow t_2$ and (f x) is typewise correct. If g is the corresponding actual parameter of type $t_3 \rightarrow t_4$, then

(a) x has to be coerced from t_1 to t_3 before g is applied to it during execution.

(b) the result of (g x) has to be coerced from t_4 to t_2.

Therefore, we require coercion from $t_3 \rightarrow t_4$ to $t_1 \rightarrow t_2$ to be performed in two steps. It is admitted by Hext that this idea has not been implemented.

1.2.1 Summary

We have used System-H to illustrate two points:

(1) partial ordering among elements of a type-checking system.

(2) the use of "GENERAL" and "UNKNOWN" in type-checking.

Our disappointment with system-H is that there is not enough work on functional types, so it is not an unexpected result that system-H shares shortcomings with System-L. In fact, in one way or the other, type-checking of functional types is emerging as our central unsolved problem. So let us turn now to Morris' treatment of this.

1.3 System-M

System-M [Morris,1968] is claimed to apply entirely to functional types. Elements of this system are:

(1) type-constants which can be divided into two sub-classes,

(a) primitives $t_1, t_2, \ldots$ (e.g.[INTEGER],[REAL]), though in his paper, only the type "NUMBER" is used in the examples.

(b) higher-ordered types: functional types are the only class of higher ordered type mentioned in his paper, the general form of which is $[t_i \rightarrow t_j]$ where t_i, t_j are types, (which can be primitive or functional).

(2) type-variables, V_1, V_2, ..., type values as described in (1) can be assigned to type-variables during type deduction.

Primitive operators (functions) will be mapped into type-expressions as before. The system is used to check functionality of expressions in λ-Calculus. Functionality tells us what type deductions are permissible from syntactic considerations alone.

For the purpose of illustration, expressions are assumed already parsed into tree forms. For example, the expression $(\lambda x{:}x)3$ can be represented as in diagram (1.3.1).

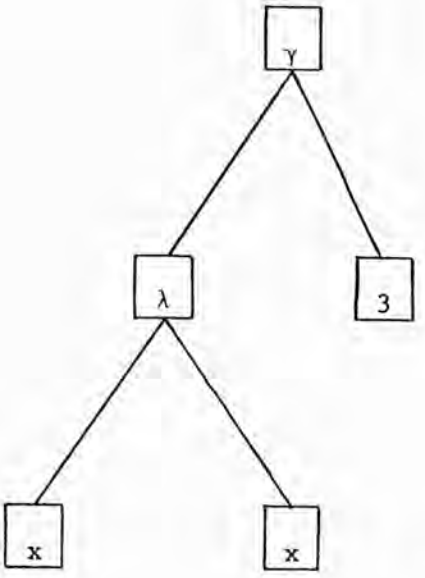


Diagram (1.3.1)

There are four rules governing the type relations among various constructs of the language.

(1) All occurrences of the same bound variable must have the same type. For example, the two occurrences of x in diagram(1.3.1) must have the same type.

(2) Type of λ-node=[type of the left descendant]→[type of the right descendant]. This is to say that λ-abstractions can be interpreted as functions.

(3) If $t_1 \rightarrow t_2$ is the type of the left descendant of a γ-node, then the right descendant must be of the type t_1, and the type of that γ-node will be t_2.

(4) A constant is always mapped into a pre-defined type-constant. For example, 3 will always have the type

[NUMBER] (which can be abbreviated to [N], the reader being aware that it is the only primitive type considered here).

Type-checking

Type checking will proceed as follows,

(1) Every node and leaf is mapped into a distinct type-variable V_i as shown in diagram(1.3.2) for the example given in diagram(1.3.1). The V_i's are related to each other according to the rules stated in the last paragraph. These relationships can be represented by a set of simultaneous equations awaiting solution. For example,

$V_4=V_5$ rule 1

$V_2=V_4 \rightarrow V_5$ rule 2

$V_2=V_3 \rightarrow V_1$ rule 3

$V_3=[N]$ rule 4

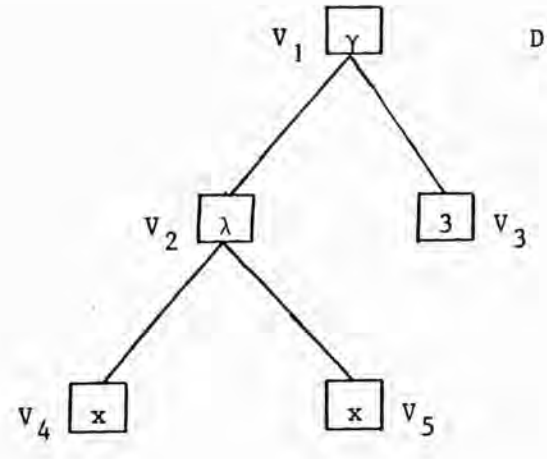


Diagram (1.3.2)

(2) All redundent variables appearing on the left of an equation will be removed. For example, all occurrences of V_4 can be replaced by V_5. Similarly, all V_3 can be replaced by [N]. So we shall be left with the following equations,

$$V_2=V_5\rightarrow V_5$$
$$V_2=[N]\rightarrow V_1$$
$$V_3=[N]$$

If there is any contradiction in the remaining equations then the expression under consideration is regarded as typewise incorrect.

Example of contradictions

(1) $t_1=t_2$ where t_1 and t_2 are two distinct type constants

(2) $t_1=V_i\rightarrow V_j$ where t_1 is a primitive type-constant, and V_i, V_j are any type-variables.

(3) If there exists two equations whose lefts are the same, for example, $A=B_1\rightarrow C_1$ and $A=B_2\rightarrow C_2$, then we can replace them by two additional equations, $B_1=B_2$ and $C_1=C_2$. At the completion of this step, we shall return to step(2) again. Thus steps(2) and (3) will form a loop which will be exited whenever step(3) is no longer applicable.

Example

the two equations of V_2 can be replaced by

$V_5=[N]$ and

$V_5=V_1$

(4) If there is no circularity among the type expressions, then the program will be regarded as typewise correct.

Example of circularity

$V_i=V_j\rightarrow V_k$ while

$V_j=V_i\rightarrow V_m$

if we replace V_j in the first equation by the right side of the second equation, we shall have

$V_i=[V_i\rightarrow V_m]\rightarrow V_k$

Occurrence of the same type variable on both sides of the equation, as V_i, means it describes a circular type. There cannot exist type-constants that have such properties. Consequently, any programs whose type equations resulted in such a situation will not be accepted by System-M.

Example of step (4)

At this stage, we shall have the following equations:

$V_5=N$ $V_3=N$ $V_1=N$

Neither of these equations is circular, so "$(\lambda x{:}x)3$" is typewise correct.

Let us examine the two occurrences of V_2 in step(1) again. They are "$V_2=V_4\rightarrow V_5$" and "$V_2=V_3\rightarrow V_1$". The former describes the type of "$(\lambda x:x)$", while the latter occurs when "$(\lambda x:x)$" is applied to an argument. Therefore V_2 is used to link up the defining instance and the applying instance of the λ-expression. Since V_3 is the type of the argument to be acted upon by the function "$(\lambda x:x)$" of type $V_4\rightarrow V_5$, so it is necessary that $V_3=V_4$, and the same conclusion can be applied to V_1 and V_5. So step (3) in this type-checking process is analogous to the type reduction mechanism in previous systems (although it seems "expansion" would be a more appropriate word here). Unlike the previous systems, type errors may not be determined immediately they occur. Determination has to wait until eventually some contradiction appears among the equations. In system-H or system-L, type-checking proceeds more or less in the same order as the program will be executed. But in system-M, there is no predefined order within each step, as the result will be the same if we change the order of the equations.

In conclusion, we find that Morris' introduction of type-variables into type algebra is a great advance. For example, in the expression "$(\lambda x:x)3$", there is no declaration for the type of x. Nor do we know the type of "$(\lambda x:x)3$". Nevertheless, as we have shown above, we are able to prove that whether expressions are typewise correct or not.

1.4 Problems in type-checking

There are two main problems which cannot be solved by any of the systems described so far. They are parametric polymorphism and circular types. We shall only define these problems here. Solutions to them will be proposed in later chapters after we have discussed our own systems.

1.4.1 Parametric Polymorphism

The problem is better illustrated by example,

Example

```
(λtwice:(twice numericfunction)(numeral)...
        (twice stringfunction)(string)...)
(λf:λx:f(f x))
```

Assuming that the types of "numeral" and "string" are t_n and t_s respectively, the types of numericfunction and stringfunction will be $[t_n \rightarrow t_n]$ and $[t_s \rightarrow t_s]$ respectively.

Our problem is "what is the type of twice?". We notice that as far as "twice" is concerned, it will accept arguments of any type. Whether or not these arguments are "happy" with each other is a matter that cannot be determined at compile time, unless we can express some relationship among arguments in the type expression of a function. This is not possible in ordinary type-checking systems.

This example reveals that there are some objects in type-free models that are unacceptable by most type-checking systems because there is no adequate type expression to be assigned to the objects.

1.4.2 Circular Types

Again, we illustrate this problem by example,

$(\lambda g{:}g(g,2))(\lambda(f,n){:}$IF n=1 THEN 1 ELSE n*f(f,n-1) FI)

there is no recursive definition in this example, and it is a correct program which yields 2 as result.

In order to simplify this problem, let us assume that the function takes only one argument, say f(f). Suppose t is the type of f, since f is a function, so t must be a functional type, say $t_1 \rightarrow t_2$, thus we can write down the following equation :

$$t = t_1 \rightarrow t_2 \qquad \ldots(1)$$

where t_1 is the type of the argument expected by f as we have said before. But if f can be applied to itself, then t_1 must be the same as t. Therefore (1) can be rewritten as:

$$t = t \rightarrow t_2 \qquad \ldots(2)$$

"(2)" is impossible because it is not possible to have a type expression to be a subpart of itself, as can be seen from the way that functional types are constructed. Consequently, it will be impossible to assign a proper type to f.

Another example, we saw in Part One, chapter 1, how the classes of recursive functions, for-loops and while-loops are founded on the Y-combinator, whose expression includes a subexpression of the form (f f). Yet we have just seen that the logical theories of type-checking developed so far all reject expressions of this form.

1.4.3 The effect of type problems on language design

The easiest way to solve the problems posed above is to abandon altogether these "unsafe" structures in the design of a language. This could be done by insisting all types of variables must be fully declared as in Algol-68.

The cost that we have to pay is to give up the expressive power and flexibility that we have enjoyed in type-free programming. Let us illustrate this by an example which we have generalized from [Burge,1972] ,

Example

$$\text{LET REC } f(x_1,x_2,x_3,x_4,x_5,x_6) \equiv$$
$$\text{IF } x_1 = x_2 \text{ THEN } x_3 \text{ ELSE}$$
$$x_4(x_5\ x_1)(f((x_6\ x_1),x_2,x_3,x_4,x_5,x_6))$$

Suppose we have also defined the following functions:

```
LET times x y≡x*y
AND pred n≡n-1
AND ident x≡x;
```

then the factorial of any positive number n can be obtained by simply applying f to the list of arguments (n,0,1,times,ident,pred).

In fact, with suitable choice of arguments, we can describe a few list-processing functions in terms of "f" and the corresponding list of arguments. Lest it be thought that this kind of thing is not serious programming, let it be pointed out quite firmly that programs of this generality are called operating systems or simply computers.

However, in spite of the usefulness of the function "f" in our example, it is impossible to define it in Algol-68 because we are unable to declare its type due to absence of parametric polymorphism from this language.

The foregoing paragraphs describe the situation that we are facing at the moment. The problems stated here account for the initial motives of our investigations which are to be described in following chapters.

CHAPTER TWO

TYPE CALCULUS

In this chapter we shall examine properties of some type constructors. These will include the constructors of functional, union and intersection types. The inter-relations among them are also discussed. The concepts established in this chapter will be used in constructing two type-checking systems in later chapters.

2.1 Functional types

Definition (2.0)

If t_i, t_j are types, then $t_i \rightarrow t_j$ is also a type which describes a class of functions so that each function when applied to objects of type t_i, yields an object of type t_j.

Theorem (2.1) (theorem of functionality:strict)

If $t_i \rightarrow t_j$ is the type of f and t_i is the type of x, then t_j is the type of (f x)

The proof of Theorem(2.1) comes directly from definition(2.0), so we shall omit it here.

2.2 Types are Sets

Consider types as sets, for example, [INTEGER] is the name of the set {1,2,3,...} and [BOOLEAN] is the name of the set {TRUE,FALSE}. Similarly, [INTEGER→INTEGER] is the name of the set $\{f_1,f_2,...\}$ so that for every i, f_i is in the set [INTEGER→INTEGER] and for every x in [INTEGER], $(f_i\ x)$ is in the set [INTEGER].

Most of the theorems below may be novel to types, but they are quite well known among sets, so in those cases where proof is omitted, readers may consult any book on set theory or propositional calculus.

Definition (2.2)

If A and B are types, then A includes B ($A \supseteq B$) means that for every x in B, x is in A.

Example

let EVEN be the set {2,4,6,...}, then it is trivial that [INTEGER]$\supseteq$[EVEN]

Now, let us modify Theorem(2.1) as follows:

Theorem(2.3) (theorem of functionality: extended)

If $t_i \rightarrow t_j$ is the type of f and t_k is the type of x, then t_j is the type of (f x) if $t_i \supseteq t_k$

Proof

If x is in t_k and $t_i \supseteq t_k$, then by Definition(2.2), x must be in t_i. Since x is in t_i, then by Theorem(2.1) (f x) must be in t_j.

It is clear that Theorem(2.3) is more powerful than Theorem(2.1) because the latter requires "$t_k = t_i$" which is a stronger requirement than "$t_i \supseteq t_k$", ($t_k = t_i$ implies $t_i \supseteq t_k$ as well as $t_k \supseteq t_i$). It is worth noticing that type-checking systems for most languages are based on the concept of Theorem(2.1) while all our systems (System-Y and System-F) are based on Theorem(2.3).

In considering types are sets, there is no indication that [REAL]$\supseteq$[INTEGER] or [INTEGER]$\supseteq$[REAL]. This is particularly true for languages where coercion does not exist so that different symbols have to be used for integer and real arithemetical operations as it they were distinct types.

2.3 Union types

We shall seldom come across union types if we are only interested in arithemetical manipulations. However, union types can occur quite naturally if we are dealing with more complicated data structures. For example, we may have data types called [BOYS] and [GIRLS], then the data type [CHILDREN] can be regarded as the union of [BOYS] and [GIRLS]. Symbolically, we write

$$[CHILDREN]=[BOYS]\cup[GIRLS]$$

It is obvious that $[CHILDREN]\supseteq[BOYS]$ and $[CHILDREN]\supseteq[GIRLS]$. In general, we have the following theorem.

Theorem(2.4) (proof omitted)

If $t=t_j \cup t_k$, then $t\supseteq t_j$ and $t\supseteq t_k$.

Suppose ageofchildren is the function for finding the age of children and let us assume its type is [CHILDREN→INTEGER]. If [BOYS] is the type of x, then according to Theorem(2.3), [INTEGER] will be the type of (ageofchildren x). We notice that the same expression is undefined under Theorem(2.1), (but Algol-68 makes it legal by having coercions). On the other hand, if [BOYS→INTEGER] and [CHILDREN] are types of ageofboys and y respectively, then the type of (ageofboys y) will not be defined in either theorems.

Union types can be united from any other types, including union types.

Theorem(2.5) (proof omitted)

For any types A, B, and C,

$$[A\cup B]\cup C=A\cup[B\cup C]=[A\cup B\cup C]$$

2.4 Type puzzle

Suppose function f is defined as follows:

$$f(g)=(g\ 1)\boxplus(g\ 1.0)$$

where g is the formal parameter of f and $\boxplus$ is some operation that we shall leave undefined and "1", "1.0" are in [INTEGER] and [REAL] respectively. Let "t" be the type of the result of the expression "(g 1)$\boxplus$(g 1.0)". Assuming that coercion is not allowed, our puzzle is

"what are the types of f and g?"

2.5 Solution to the puzzle: first attempt

According to the definition of f above, g is a function which is defined for both integers and reals. So as our first attempt, we let

$$\text{type of "g"}=[\text{INTEGER}\rightarrow t_1]\cup[\text{REAL}\rightarrow t_2]$$

which means that when g is applied to an integer, it yields a result of type t_1 or if applied to a real, the result is of type t_2 or otherwise undefined. Ledgard and others also used the union notation to define the type of polymorphic operations such as "+". Correspondingly, the type of f is:

$$[[[\text{INTEGER}\rightarrow t_1]\cup[\text{REAL}\rightarrow t_2]]\rightarrow t]$$

which follows the theorem of functionality (Theorem 2.3 or 2.1) and t is defined in last section.

Suppose "h" is a function of type $[\text{INTEGER}\rightarrow t_1]$, then according to Theorem(2.3), the type of (f h) is defined and is [t] because $[[\text{INTEGER}\rightarrow t_1]\cup[\text{REAL}\rightarrow t_2]]\geq[\text{INTEGER}\rightarrow t_1]$ according to Theorem(2.4). Thus as far as the type is concerned, f can be applied to h.

From the computational point of view, when f is applied to h, the actual parameter h will substitute for the formal parameter g throughout the definition of f. So we should end up with

$$(h\ 1)\boxplus(h\ 1\cdot 0)$$

As we have said above that the type of h is $[\text{INTEGER}\rightarrow t_1]$ which means h is applicable to integers only and will be undefined for any other objects. Consequently, (h 1·0) would be undefined. This contradicts our statements above that (f h) is typewise correct.

So it is not desirable for the type of f to be $[[\text{INTEGER}\rightarrow t_1]\cup[\text{REAL}\rightarrow t_2]\rightarrow t]$ nor for the type of g to be $[[\text{INTEGER}\rightarrow t_1]\cup[\text{REAL}\rightarrow t_2]]$.

We have to introduce the idea of intersection types before we can propose our second attempt at solution of this puzzle.

2.6 Intersection types

Definition(2.6)

If A, B are types, then for any object x, x in type [A∩B] implies that x in A and x in B.

At first sight, it seems that the intersection of two types must always be empty (that is, the two types have no elements in common).

Examples

(1) BOYS∩GIRLS=EMPTY

(2) INTEGER∩REAL=EMPTY

However, a function f applicable to BOYS may also be applicable to GIRLS, (for example, f may be the function for finding their height, weight, etc.). Thus we establish the following facts:

(1) f is in [BOYS→t]

(2) f is in [GIRLS→t]

where t is some appropriate type.

Since it is the same "f" in (1) and (2) above, it means that f is the element of both sets. So the intersection of [BOYS→t] and [GIRLS→t] cannot be empty and f is in [BOYS→t]∩[GIRLS→t]. Therefore we reject the proviso in most algorithmic languages that types should be distinct.

Theorem(2.7) (proofs omitted)

If t_1 and t_2 are types, then

(a) $t_1 \cap t_2 \subseteq t_1$

(b) $t_1 \cap t_2 \subseteq t_2$

(c) $t_1 \cap t_2 \subseteq t_1 \cup t_2$

(d) $[t_1 \cap t_2] \cap t_3 = t_1 \cap [t_2 \cap t_3] = [t_1 \cap t_2 \cap t_3]$

2.7 Solution to the puzzle: second attempt

In this second attempt, we let

type of g=[INTEGER→t_1]∩[REAL→t_2], so that

type of f=[[[INTEGER→t_1]∩[REAL→t_2]]→t]

when f is applied to h, depending on the type of h, the following are some of the possible cases:

(1) type of h is $[INTEGER \rightarrow t_1]$:
type of (f h) is undefined because $[[INTEGER \rightarrow t_1] \cap [REAL \rightarrow t_2]] \subseteq [INTEGER \rightarrow t_1]$ according to Theorem(2.7a)

(2) type of h is $[REAL \rightarrow t_2]$:
type of (f h) is undefined for the same reason as (1)

(3) type of h is $[[INTEGER \rightarrow t_1] \cup [REAL \rightarrow t_2]]$:
type of (f h) is also undefined according to Theorems (2.7c and 2.3)

(4) type of h is $[INTEGER \rightarrow t_1] \cap [REAL \rightarrow t_2] \cap t$ (where t can be any type):
type of (f h) is defined because $[INTEGER \rightarrow t_1] \cap [REAL \rightarrow t_2] \supseteq [INTEGER \rightarrow t_1] \cap [REAL \rightarrow t_2] \cap t$.

Readers may be interested to find that in cases undefined above, the computation will also be undefined. And for defined cases, they will be defined during computation.

Thus in this attempt, we have obtained the "correct" types for both f and g. But if g is in the set "$[[INTEGER \rightarrow t_1] \cap [REAL \rightarrow t_2]]$" then g must be in the set "$[[INTEGER \rightarrow t_1] \cup [REAL \rightarrow t_2]]$" because the latter$\supseteq$the former according to Theorem(2.7c). In other words, our first attempt on the type of g should also be correct. In fact, there are infinitely many solutions because if "t" is the solution, then for any type t_i, "$t \cup t_i$" is also a solution. But as we iterate the process further and further, our knowledge

of the type of the object diverges from our second attempt and that is the reason why solutions in the first attempt are less well defined than that in the second attempt.

Theorem(2.8)

If t_1, t_2 are types and $t_1 \geq t_2$, then for any object f, if both "f in t_1" and "f in t_2" are correct, then we say "f in t_2" is a more exact description than "f in t_1"

Suppose we have a universal type [U] such that for any arbitrary type t, $[U] \geq t$ is always true. Then it is obvious that we can always assign [U] to any object and so in our program, every object will be of the same type. This is the situation in type-free languages (logically it is the same whether we say every object has the same type or every object has no type). So it is not enough in just having an arbitrary correct type assignment (e.g.[U]), we always have to look for a more exact one.

Thus in the above case, if both "g in $[[\text{INTEGER} \to t_1] \cap [\text{REAL} \to t_2]]$" and "g in $[[\text{INTEGER} \to t_1] \cup [\text{REAL} \to t_2]]$" are correct, the former should be preferable to the latter. In general we should ascribe intersection types to polymorphic functions rather than union types.

Corollary

The type puzzle can be generalized as

$$f(g)=(g\ a_1)\boxplus(g\ a_2)\boxplus \ldots \boxplus(g\ a_n)$$

where for every i, $1\le i\le n$, a_i is of type t_i and $(g\ a_i)$ is of type t_i'.

So "g" is of type $[t_1\to t_1' \cap t_2\to t_2' \cap \ldots \cap t_n\to t_n']$, and by Theorem(2.7), this is a smaller set than any $[t_i\to t_i']$. In a more general sense, g may be of type $[t_1\to t_1' \cap \ldots \cap t_i\to t_i' \cap \ldots]$ for all i, $1\le i\le\infty$, as in the example $f(g,a)=g(a)$ where a can be of any type t_i $(1\le i\le\infty)$. We can imagine that the set will become smaller and smaller as i become bigger and bigger. Thus in practice, we may regard $[t_1\to t_1' \cap \ldots \cap t_i\to t_i' \cap \ldots]$ as an empty set as i approach infinity. In other words, we could not then declare f, g (as defined above) in typed languages. But f then defines a Turing Machine or Operating System. Hence Turing Machines or Operating Systems are proved impossible to declare in typed languages.

Theorem(2.9)

For any types t_1, t_2 and t,

$$[t_1 \cup t_2]\to t=[t_1\to t]\cap[t_2\to t]$$

Proof

let (L1) t_1 be the set $\{a_1,a_2,\ldots\}$

(L2) t_2 be the set $\{b_1,b_2,\ldots\}$

(L3) $[t_1 \cup t_2]\rightarrow t$ be the set $\{f_1,f_2,\ldots\}$

(L4) $[t_1\rightarrow t]$ be the set $\{g_1,g_2,\ldots\}$

(L5) $[t_1\rightarrow t]\cap[t_2\rightarrow t]$ be the set $\{c_1,c_2,\ldots\}$

(L6) $[t_2\rightarrow t]$ be the set $\{h_1,h_2,\ldots\}$

Deductions

(D1) for every i, g_i is defined for the set $\{a_1,a_2,\ldots\}$ and produces result in the set t. (by Theorem 2.1)

(D2) for every j, h_j is defined for the set $\{b_1,b_2,\ldots\}$ and produces result in the set t. (by Theorem 2.1)

(D3) for every k, there must exist i and j such that

$$c_k=g_i=h_j \qquad \text{(by definition 2.6)}$$

Thus, c_k is defined for the set $\{a_1,a_2,\ldots\}$ as well as the set $\{b_1,b_2,\ldots\}$ and produces results in the set t in both cases. In other words, c_k is defined for the set $\{a_1,a_2,\ldots,b_1,b_2,\ldots\}$ and produces results in the set t.

(D4) for every v, f_v is defined for the set $\{a_1,a_2,\ldots,b_1,b_2,\ldots\}$ and produces results in the set t.

(D5) from (D3) and (D4), $\{f_1,f_2,\ldots\}$ must be the same set as $\{c_1,c_2,\ldots\}$, therefore

$$[t_1 \cup t_2]\rightarrow t=[t_1\rightarrow t]\cap[t_2\rightarrow t]$$

qed.

In fact, if we reinterpret "→" as implication in propositional calculus [Edwards,1975][Curry,1958], then Theorem(2.9) can be proved quite easily by truth-tables which one can find in any textbook on propositional calculus.

2.8 Summary of the three constructors

(1) We expect that our relaxation on the definition of type functionality provides a basis for a more powerful type-checking system.

(2) Some programs which we hold to be logically correct are rejected by most systems due to the inflexibilities of Theorem(2.1)

Example

(λ[BOYS→INTEGER]f:...)([CHILDREN→INTEGER]g)

is, we believe, logically correct because "g" can be applied to a larger domain than that required by "f".
In this example, we extend our notation by allowing a type to precede an object that is not a binding variable. For example, ([CHILDREN→INTEGER]g) means the type of g is that preceding it.

(3) We encountered a problem in using Theorem(2.3) when the union type is used in an unrestricted manner. We resolved the problem by introducing intersection types and the concept of "more exact" type description.

(4) Theorem(2.1) has sufficed for computing systems for so long partly because need for partial orderings of types (coercion) has only recently been felt and partly because functional types have been playing a very minor part in type checking until our discovery that Algol-68 does not suffice for declaration of the useful class of polymorphic procedures.

Similarly, the argument can be applied to intersection of two or more types. These are not necessary if our concern has been restricted to primitive types only (such as integer or real). But it will be a vital area of study when functional types are studied seriously as in our systems.

(5) In Algol-68 and etc., polymorphic functions are restricted to only a limited number of operators and each operator has to be redeclared for each desired type. But, if a user can define his arbitrary intersection and union types, we shall expect polymorphic functions to occur more generally.

Example

Assuming MALE and FEMALE are types, then the function sexof-child will have the type

$$[BOYS \rightarrow MALE] \cap [GIRLS \rightarrow FEMALE]$$

As far as we know, there is no language that currently supports such facilities. It is our intention to examine the nature of such type-checking systems so as to have an insight into future languages.

(6) Since we already have union and intersection of types, so implicitly we have negation of types as well. The idea of negation type is indeed seductive.

Example

> for any types t_1 and t_2, suppose function f is defined for all types except t_1 and produces results in t_2, then the type of f would be:

$$[[\sim t_1] \rightarrow t_2]$$

We believe that negation type is a very important area for future research.

2.9 Rule of Inclusion: functional types

The rules of Inclusion for union and intersection types have been stated in Theorems(2.4) and (2.7) respectively. We have to pay special attention to the rule of Inclusion for functional types because (1) the result is less familiar to us and (2) it is the backbone of all type-checking systems handling functional languages.

Theorem(2.10)

For all types, t_1, t_2, t_3 and t_4,

$$t_1 \to t_2 \supseteq t_3 \to t_4 \quad \text{if } t_1 \subseteq t_3 \text{ and } t_2 \supseteq t_4$$

(Obviously, the "only if" part is not necessarily true. However, in type checking, the "only if" part is not required.)

Proof

Given: $(t_1 \subseteq t_3)$ and $(t_2 \supseteq t_4)$

to prove: $[t_1 \to t_2] \supseteq [t_3 \to t_4]$

Let (L1) t_1 be the set $\{a_1, a_2, \ldots\}$

(L2) t_2 be the set $\{b_1, b_2, \ldots\}$

(L3) t_3 be the set $\{c_1, c_2, \ldots\}$

(L4) t_4 be the set $\{d_1, d_2, \ldots\}$

(L5) $[t_1 \to t_2]$ be the set $\{f_1, f_2, \ldots\}$

(L6) $[t_3 \to t_4]$ be the set $\{g_1, g_2, \ldots\}$

Deductions

(D1) Any function which is defined for all elements of the set $\{a_1,a_2,\ldots\}$ and produces results in the set $\{b_1,b_2,\ldots\}$ must be an element of the set $\{f_1,f_2,\ldots\}$ (by Definition 2.1)

(D2) For every i, g_i is defined for all elements in the set $\{c_1,c_2,\ldots\}$, since $\{c_1,c_2,\ldots\}\supseteq\{a_1,a_2,\ldots\}$ (given), so g_i is defined for all elements in the set $\{a_1,a_2,\ldots\}$

(D3) For every i, for every x and x is an element of $\{a_1,a_2,\ldots\}$, $g_i(x)$ is defined, (from D2). Let $y=g_i(x)$, according to Theorems 2.1 and 2.3, y must be an element of the set $\{d_1,d_2,\ldots\}$. Since $\{b_1,b_2,\ldots\}\supseteq\{d_1,d_2,\ldots\}$, therefore y is an element of $\{d_1,d_2,\ldots\}$ implies that y is an element of $\{b_1,b_2,\ldots\}$

(D4) From (D2) and (D3), we come to the conclusion that, for every i, g_i is defined for all elements in the set $\{a_1,a_2,\ldots\}$ and produces results in the set $\{b_1,b_2,\ldots\}$.

(D5) From (D4) and (D1), we obtain the result that g_i must be an element of the set $\{f_1,f_2,\ldots\}$

(D6) Since (D5) is true for all i, therefore

$$\{g_1,g_2,\ldots\}\subseteq\{f_1,f_2,\ldots\} \text{ that is } [t_3\rightarrow t_4]\subseteq[t_1\rightarrow t_2]$$

qed.

Examples

(1) [INTEGER→REAL]≥[INTEGER→REAL]

(2) [INTEGER→REAL]≥[[INTEGER∪BOOLEAN]→REAL]

(3) [[[INTEGER∪BOOLEAN]→REAL]→t]≥[[INTEGER→REAL]→t]
where t is any type.

(4) From theorems 2.10 and 2.3, the expression
(λ[BOYS→INTEGER]f:...)([CHILDREN→INTEGER]g)
is type-wise correct.

2.10 Closing Remark

The main purpose of the discussion here is to explore the less familiar properties of functional types. These properties have been largely ignored in other systems. We have found that the discussion is fruitful because

(1) We shall be able to construct practical systems upon theoretical considerations, although the theoretical results obtained so far are quite informal.

(2) We can have the same treatment on functional types as of primitive types, for example, the inclusion rules.

(3) We shall have more information and knowledge to re-examine the problems we have encountered so far in type

checking. We have noticed already that most of these problems concern functional types.

We now present two systems in which we shall see how the ideas in this chapter are put into practice.

CHAPTER THREE

SYSTEM-F

We feel that it is very undesirable that a program has to be written in a type-free language simply because a subpart of it would be rejected by static type-checking systems. Nor, on the other hand, is it desirable to have to forgo the flexibility of type-free languages in order to have static type-checking. To test out our feelings we decided to design System-F to operate on programs with the following goals in mind.

(1) Wherever programs are fully typed (i.e. types of all variables are declared), the system should perform type-checking such as one normally expects from a static type-checking system.

(2) It should accept type-free expressions wherever they are subparts of a typed program. A subpart of a program can be as small as a variable, a subexpression, expression or any other well-formed structure, or as large as a complete program.

(3) Type-checking processes should nevertheless be expressible in terms of simple reduction rules.

3.1 Basic types of System-F

There are six basic types in the system, namely INTEGER, REAL, BOOLEAN, STRING, ANY and ERROR. We shall use the abbreviation I, R, B, S, A and E for these types respectively. Certainly the set of basic types can be extended or amended , if necessary, if the system is applied to a language with other primitive types. Type expressions might be enclosed by square brackets, "[" and "]", in order to distinguish them from the typefree parts of the text. One may consider [A] as union of all types so that it may be regarded as containing utterly inexact information.

3.2 Constructed types

Types can be constructed from the basic types or other constructed types. There are four constructors in System-F,

(1) "$\rightarrow$", types constructed by it are called functional-types

(2) "$\cup$", types constructed by it are called union-types

(3) "$\cap$", types constructed by it are called intersection-types

(4) "&", types constructed by it are called ordered-types (to be discussed in §3.7)

3.3 Type Expressions

Type expressions can be defined recursively as

(1) if t is a basic type, then t is a type expression

(2) if t is a constructed type, then t is a type expression

(3) if t_1, t_2 are type expressions, then $(t_1\ t_2)$ is a type expression

(4) the only type expressions are those defined by (a)-(c)

Examples

(1) [I]

(2) [I→I]∪[R→R]

(3) ([I→I][I])

3.4 Rules of reduction

Given a type expression $(t_1\ t_2)$, it may be possible to reduce it to a simpler expression according to the rules of reduction. Before we state these rules, we shall explain the conventions first,

(1) The symbol "≃" means "reduces to", see (3) below

(2) All t's with or without subscripts stand for any type expression unless specified otherwise.

(3) $(t_i\ t_j) \simeq t_k$ means the expression $(t_i\ t_j)$ is reduced to t_k

(4) $(t_i\ t_j) \simeq t_k{}^*$ means the same as $(t_i\ t_j) \simeq t_k$ except that the reduction has to be confirmed at a later stage (usually this means at run-time). Hence we place on record our decision that type checking is an iterative process in which early approximations may be refined to subsequent better approximations. The symbol "*" is not part of the type expression. In other words, it will not affect the value of the type expression it is attached to. It might be helpful if we imagine that there exists a special register that would be set in those cases where "*" appears in the reduction rules. The parser (or other routines) on discovering the register set would construct a modified parse tree and unset the register. Hence, the "*" need not exist at all. It is included here to remind the reader of what might happen.

(5) $[t_i{}^* \cup t_j{}^*]$ can be expressed as $[t_i \cup t_j]^*$, similarly, $t_i{}^* \cap t_j{}^* = [t_i \cap t_j]^*$

(6) $t_i \supseteq t_j$ means t_i includes t_j according to the rules of inclusion.

The following are the rules of reduction:

(R0) $t_i\ t_j \simeq [E]$ if no rule is applicable

(R1) $[A]t_j \simeq [A]^*$

(R2) $[t_{i1} \cup t_{i2} \cup \ldots \cup t_{ik}]t_j \simeq [(t_{i1}\ t_j) \cup (t_{i2}\ t_j) \cup \ldots \cup (t_{ik}\ t_j)]^*$

(R3) $[t_{i1} \cap t_{i2} \cap \ldots \cap t_{ik}]t_j \simeq [(t_{i1}\ t_j) \cup (t_{i2}\ t_j) \cup \ldots \cup (t_{ik}\ t_j)]$

(R4) $[t_i \rightarrow t_j][A] \simeq t_j^*$

(R4a) $[t_i \rightarrow t_j][t_{k1} \cup t_{k2} \cup \ldots \cup t_{kn}] \simeq$
$[([t_i \rightarrow t_j]t_{k1}) \cup \ldots \cup ([t_i \rightarrow t_j]t_{kn})]^*$

(R5) $[t_i \rightarrow t_j]t_k \simeq t_j$ if $t_i \geq t_k$

(R6) (a) $t \cup [E] \simeq t$

(b) $t \cup t \simeq t$

(c) $t \cup [A] \simeq [A]$

Examples

(1) $[A][I] \simeq [A]^*$

(2) $[A][I \rightarrow I] \simeq [A]^*$

(3) $[[I \rightarrow I] \cup [R \rightarrow R]][I] \simeq [I \cup E]^* \simeq [I]^*$

(4) $[[I \rightarrow I] \cap [R \rightarrow R]][I] \simeq [I \cup E] \simeq [I]$

(5) $[[I \rightarrow I] \cap [R \rightarrow R]][I \cup R] \simeq [[I \rightarrow I][I \cup R]] \cup [[R \rightarrow R][I \cup R]]$
$\simeq [I^* \cup R^*] = [I \cup R]^*$

alternatively, we may replace $[I \to I] \cap [R \to R]$ by $[I \cup R] \to [I \cup R]$,

(5') $[[I \cup R] \to [I \cup R]][I \cup R] \simeq [I \cup R]$

and the advantage is that the marker "*" is not required in (5'). The implementor may ignore this possibility and we can see that in both cases the expression is reduced to $[I \cup R]$ since the marker is not part of the expression.

(6) $[I \to I][I \cup R] \simeq I^*$

(7) $[[I \to I] \to I][[I \to I] \cap [R \to R]] \simeq I$

in practice, we may try (R5) before (R4a), so that $([I \cup R] \to I][I \cup R])$ is reduced to $[I]$ instead of $[I]^*$.

(8) $[I][R] \simeq [E]$

3.5 Coercion

In most languages, some form of coercions are allowed. This is to say the types are arranged into a partially ordered set, the set of types being augmented by a set of paths between them. Coercion is an operation by the computer that corresponds to a move along an allowable path in the partially ordered set.

It is not necessary to incorporate coercion into our basic system. All we need is to regard coercion as an independent system which can be called by the type-checking system. In the example (f a), if the type of f and a are [R→t] and [I] respectively, the type of (f a) would be ([R→t][I]), which is [E] when reduced. At this point, the coercion system will be invoked to see whether there exist or not a coercion function C_I^R which coerces an integer to a real. If the function exist, the code (f a) will be amended to (f(C_I^R a)). The control will then be returned to the type-checking system which will deduce the type of the amended expression. If the coercion function does not exist, the erroneous state is confirmed.

3.6 Definition of the mapping function Φ

In this section, we shall describe a simple version of Φ. The domain of Φ is expressions in λ-Calculus. It is assumed that a λ-expression has only one binding variable, Φ is then defined recursively as below:

(0) All constants are mapped into the respective basic types.

(1) All primitive operators (functions) are mapped into some fixed type expressions.

Example

$\Phi(\sim)=[B\rightarrow B]$ where "~" stands for negation

(2) If x is a variable, then $\Phi(x)$ is the type expression assigned to x either by default or by declaration. The default type is [A].

(3) If (f x) is a combination, then $\Phi(f\ x)=(\Phi\ f)(\Phi\ x)$

(4) $\Phi(\lambda x{:}M)=(\Phi\ x)\rightarrow(\Phi\ M)$ where M is a well-formed expression.

Examples

(1) $\Phi(\langle STRING\rangle)=[S]$

(2) $\Phi(3)=[I]$

(3) $\Phi((\lambda x{:}x)3)=([A\rightarrow A][I])\simeq[A]$
no run-time checking is required.

(4) $\Phi((\lambda f{:}(f\ 3)\ldots\ f\ \langle STRING\rangle)(\lambda x{:}x))$
$=[A\rightarrow(([A][I])\ldots([A][S]))][A\rightarrow A]\simeq[A^*\ldots A^*]$

"..." represents some part of the expression

which can be ignored in our discussion.
We see that run-time checking is required
when (λx:x) is applied to 3 as well as when
it is applied to <STRING>.

(5) Φ((λ[A→A]f:(f 3)...(f <STRING>))(λx:x))
=[[A→A]→(([A→A][I])...([A→A][S]))][A→A]
≃[A...A]

this example is similar to (4) with the exception
that f is declared to be [A→A] and as the result
of this declaration, no run-time checking is
required. It is very important to realize the
differences of these two examples.

(6) Φ((λf:f 3)(λx:x))=[A→([A][I])][A→A]≃[A]*

(7) Φ((λ[A→A]f:(f 3))(λ[I]x:x))
=[[A→A]→([A→A][I])][I→I]
≃[[A→A]→A*][I→I]
≃[E] because (R5) is not satisfied.

3.7 Extension: ordered types

If we want to consider λ-expressions with lists of binding variables, we need type expressions which represent not only the type of each binding variable, but also the relative positions among them. Ordered types are designed for this purpose and are constructed by the operator "&".

Example

[I&R] is an ordered type of two elements and it is different from [R&I]

We add the following rules for manipulating ordered types:

(1) ordered types are right associative, that is

$$[t_1\&t_2\&t_3]=[t_1\&[t_2\&t_3]]$$

(2) extension of the rules of Inclusion

$[t_{i1}\&t_{i2}\&\ldots\&t_{in}]\geq[t_{j1}\&t_{j2}\&\ldots\&t_{jm}]$

if (a) n=m and (b) for every k, $1\leq k\leq n$, $t_{ik}\geq t_{jk}$

(3) extension of the mapping function

$$\Phi(x_1,x_2,x_3,\ldots,x_n)=((\Phi\ x_1)\&(\Phi\ x_2)\&\ldots\&(\Phi\ x_n))$$

Example

$$\Phi(1,2,3)=[I\&I\&I]$$

If the list of binding variables is (x_1,x_2,x_3) and the user wishes to declare the type of them to be t_1, t_2 and t_3 respectively, then he can simply write

$$(\lambda[t_1]x_1,[t_2]x_2,[t_3]x_3: \ldots)$$

in other words, the system would convert $([t_1]x_1,\ldots[t_n]x_n)$ into $[t_1\&t_2\&\ldots\&t_n](x_1,x_2,\ldots,x_n)$.

Example

$$\Phi((\lambda([I]x,[I]y):x+y)(2,3))$$
$$=[[I\&I]\rightarrow I][I\&I]\simeq[I]$$

In the beginning of this section, we said that ordered types are needed to account for lists of binding variables. But lists of objects are required as actual parameters for lists of binding variables and since both the formal and actual parameters must be of the same type, we conclude that ordered types have to be the type of all lists.

It has been suggested that lists can be represented by functions. To illustrate the concept of functional data structures, Reynolds [Reynolds,1970] defined the functions CONS, CAR and CDR as (presented here with minor changes in notation): (λx:λy:λz:IF z= 1 THEN x ELSE y FI), (λx:x 1) and (λx:x 2) respectively. Church [Church,1941] has also shown us how to λ-define diads, triads and the corresponding selecting functions. Edwards [Edwards,1975] however has found that it is not possible to find a general type expression to describe the type of lists from their functional representations.

In the light of these discussions, let us re-examine the problem again.

We recall that numbers 1, 2, 3, etc. are λ-defined by (λf:λx:(f x)), (λf:λx:f(f x)), (λf:λx:f(f(f x))), and so on. If we let the type of f and x be β and α respectively, then β must be [α→α] for the type expression to be meaningful. Thus the type of these expressions must be [α→α]→[α→α] for any type α. On the other hand, it is generally assumed by us that the type of these objects is [I]. In other words, [I] is used to abbreviate the class of type expressions [α→α]→[α→α] for arbitrary type α. Therefore, at the outset, we have to announce that "there is a subset of λ-expressions (of the general format (λf: x:f^n x), where n is a positive integer denoting the number of f's in the expression) whose type is always [I] and need not be deduced from underlying λ-expressions". We are not saying that the analysis of these formulae is not important, but for our purpose, that is not necessary. On the other hand, the discussion here demonstrates that the abbreviating of a subset of λ-expressions by a specific type name is an important techique in simplifying type notations and provides us a foundation to investigate other type properties.

Similarly we may apply the same argument to lists. We state that there is a subset of λ-expressions of the form (λf:f a b) whose type is [LIST] (abbreviated to [L]). Thus we do not have to worry how to derive the general type from the formulae because they are all abbreviated [L].

In table(1), we illustrate the type of some list processing functions. We find that type [L] is adequate to describe all of them except CAR because whenever CAR is applied to a list, the resulting type is always [A]. Thus we need to know more about [L]. Recalling that at the beginning of this section, we proposed ordered types for lists, so it would be logical to assume [L] abbreviates the class of all ordered types. Furthermore, we adopt the convention that if [L] is superscripted, such as $[L^i]$, then during type reduction, $[L^i]$ can be replaced by a specific element in [L] according to the following rule:

$$[L^i \rightarrow t_1]t_2 \text{ can be replaced by } [t_2 \rightarrow t_1']t_2$$

where t_2 is an element of [L] and t_1' is obtained by replacing all $[L^i]$ in t_1 by t_2. Similarly, we may apply the same treatment on [A]. Finally, if $[L^i]$ and $[L^j]$ denotes the expressions $[t_{11}\&t_{12}\&\ldots\&t_{1n}]$ and $[t_{21}\&t_{22}\&\ldots\&t_{2m}]$ respectively, where t_{ij} is any type expression, then

$$\uparrow[L^i]=t_{11}$$

$$\downarrow[L^i]=[t_{12}\&t_{13}\&\ldots\&t_{1n}]$$

$$t\&[L^i]=[t\&t_{11}\&t_{12}\&\ldots\&t_{1n}]$$

$$[L^j]\,|\,[L^i]=[t_{21}\&t_{22}\&\ldots\&t_{2m}\&t_{11}\&t_{12}\&\ldots\&t_{1n}]$$

Table(2) shows the revised type expressions for the functions shown in table(1), (except NULL, which is unchanged).

It is worth noticing that each type equation above resembles a procedure declaration in the sense that $[A^i]$ and $[L^i]$ behave as formal parameters to operations "↑", "↓", "|" and "&" whose definitions constitute the procedure bodies. The difference between $[A^i]$ and $[L^i]$ is that the actual parameter for $[A^i]$ can be any type expression, while that for $[L^i]$ must be ordered types. This suggest that we may even associate "type" to these formals. In our opinion the inclusion of type abstraction such as these in a type-checking system should be a very important project. We therefore return to type abstractions in a later chapter. For the present let us note the tabulation:

Table (1)

FUNCTION	TYPE
CONS	$[A \rightarrow [L \rightarrow L]]$
CONCAT	$[L \rightarrow [L \rightarrow L]]$
CDR	$[L \rightarrow L]$
CAR	$[L \rightarrow A]$
NULL	$[L \rightarrow B]$

Table (2)

FUNCTION	TYPE
CONS	$[A^i] \rightarrow [L^i \rightarrow [A^i \& L^i]]$
CONCAT	$[L^i \rightarrow [L^j \rightarrow [L^i \mid L^j]]]$
CDR	$[L^i \rightarrow [\downarrow L^i]]$
CAR	$[L^i \rightarrow [\uparrow L^i]]$

3.8 Application of System-F to Reckon

An additional basic type [NILS] (abbreviated to [N]) is added to the system so that Φ(NIL) or Φ(())=[N]. This is necessary because in Reckon some functions do not need any formal parameter (or NIL), but there is no need to introduce any additional rule for it. The type of each system function and operation is predefined at implementation and is denoted here as t_{op}, for example, t_{+}, t_{cond}, $t_{=}$ and etc.

Examples

(1) $\Phi(2+3)=[t_{+}\ I\ I]\simeq[I]$

(2) $\Phi((2\cdot0)+(3\cdot0))=[t_{+}\ R\ R]\simeq[R]$

(3) Φ(IF b THEN e_1 ELSE e_2 FI)

$=[t_{cond}\ t_b\ t_{e1}\ t_{e2}]$

$\simeq[t_{e1}\cup t_{e2}]$ if $t_b\supseteq[B]$ otherwise [E]

where t_b, t_{e1}, t_{e2} are the types of b, e_1 and e_2 respectively

In typed Reckon, one can declare the type of binding variables.

Examples

(1) (λ[I]n:n+n)3

(2) LET [I]n$\equiv$3;n+n;

However, type declarations are optional. In other words, all programs written in type-free Reckon are legal programs in typed Reckon. If the type of a variable has not been declared, then it will be given the default type [A] by typed Reckon.

Example

The following two programs are equivalent,

(1) (λx:x+x)3

(2) (λ[A]x:x+x)3

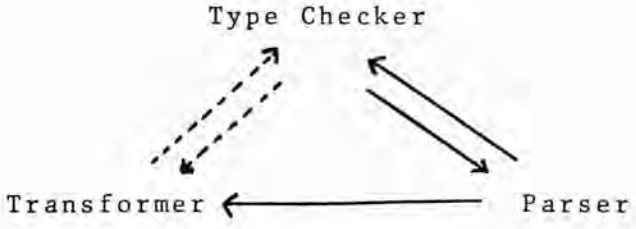
It is possible to construct an ordered type for any list from the types of its elements but since the process is time consuming, ordered types are used only for binding variable-lists (and corresponding actual parameter-lists), and in general, the crude approximation [L] is assumed or $[L_t]$ if all the elements are of type [t] (the subscript being used here for a different purpose than in the last section).

Example

$$\Phi(\lambda[L_I]p:2\ \text{TH}\ p)(2,3,4)$$
$$=[L_I \to t_{th}[I]L_I][L_I]$$
$$\simeq[L_I \to I][L_I]$$
$$\simeq[I]$$

In our implementation, type checking is performed at the same time as the source program is parsed. The advantage of this is that information gathered during type checking can be immediately available to the parser. The parser acts as a main program which will call for the type checker (or part of it) whenever it thinks it needs to. The type checker will then return control to the parser. It has already been mentioned in the previous chapter that the monadic operator "." is polymorphic in the sense that it stands for functional application, multiplication as well as string and list concatenation. One of the tasks of the parser is to replace this polymorphic operator by an appropriate typed operator based on the information provided by the checker. The semantics of the parser and the type checker is described in Appendix(B).

The routine that executes the parse tree constructed by the parser is the transformer. Further type checking is required for parts of the parse tree at places where the marker "*" was appended to type expressions during reductions, so it has been arranged that the transformer can also call the type checker. The relationship of the three components is shown in the following diagram.



Typed Reckon is implemented on the CDC 64/6600 and the program is written in Pascal. Program examples of typed Reckon are included in Appendix(C).

3.9 Parametric Polymorphism: solution

This is one of the problems that we have mentioned before. We state the example again here and see how it can be solved in System-F.

(λtwice:...(twice numericfunction)(numeral)...
(twice stringfunction)(string)...)
(λf:λx:f(f x))

as abbreviation, we let e be the expression, and $t_n \to t_n$, $t_s \to t_s$, t_n, t_s be the type of numericfunction, stringfunction, numeral and string respectively. The type of twice is then declared to be $[A \to [A \to A]]$ (abbreviated to t_w) and the type of f and x is [A] by default. The type expression is reduced as below:

$$\Phi(e) = [t_w \to [\ldots((t_w[t_n \to t_n])t_n)\ldots((t_w[t_s \to t_s])t_s)\ldots]]$$
$$[A \to [A \to (A(A\ A))]]$$

$$\simeq [t_w \to [\ldots A \ldots A \ldots]][A \to [A \to (A\ A^*)]]$$

$$\simeq [t_w \to [\ldots A \ldots A \ldots]][A \to [A \to A^*]]$$

$$\simeq [t_w \to [\ldots A \ldots A \ldots]]t_w$$

$$\simeq [\ldots A \ldots A \ldots]$$

It can be seen that twice will accept arguments of any type and this fact is well reflected in its type expression. Type errors will be detected at run-time in places where "*" occurs. This corresponds to situations where f is applied to x (or, in fact, the corresponding actual parameters) and where f is applied to the results of (f x).

3.10 Circular types: solution

The example we have shown before is,

$(\lambda g{:}g(g,2))(\lambda(f,n){:}\text{IF } n=1 \text{ THEN } 1 \text{ ELSE } n*f(f,n-1) \text{ FI})$

let e be the expression and the type of g, f, n be $[A\&A]\rightarrow[A]$, $[A]$, and $[A]$ respectively. Type checking is now just a simple exercise on the reduction rules as we have seen in the case of parametric polymorphism.

$$\Phi(e)=[[[A\&A]\rightarrow A]\rightarrow([[A\&A]\rightarrow A][[[A\&A]\rightarrow A]\&\ I])]$$
$$[[A\&A]\rightarrow(t_{cond}(t_{=}\ A\ I)[I](t_{*}\ A([A][A\&(t_{-}\ A\ I)])))]$$
$$\simeq[[[A\&A]\rightarrow A]\rightarrow A][[A\&A]\rightarrow(t_{cond}\ B*[I](t_{*}\ A\ A*))]$$
$$\simeq[[[\ A\&A]\rightarrow A]\rightarrow A][[A\&A]\rightarrow(t_{cond}\ B*[I]A*)]$$
$$\simeq [[[A\&A]\rightarrow A]\rightarrow A][[A\&A]\rightarrow A*]$$
$$\simeq [A]$$

3.11 Summary and Remarks

At the beginning of this chapter, a type-checking system was introduced. It was stated what elements are included in the basic system. The corresponding rules of reduction were explained. Knowledge of the rules of Inclusion is assumed when these rules of reduction were laid down. For illustration, the system was applied to

(1) λ-expressions with one argument

(2) λ-expression with list of arguments

(3) the Reckon language

In the discussions on application, the following points were stressed

(1) how to define the mapping function

(2) how to extend the basic system

Basically the system is a static one. By use of the "*" device, it is able to issue further checking requests in the same way as coercion operators are inserted in parse trees. Treating further checking and coercions insertion as coroutines, the system obtained is simple and easy to handle. We have seen how it is impossible to carry out all type checkings at compile time in general programming

systems. Consequently, we have had to reject the idea of purely static or purely dynamic systems. Some of the problems are solved partly due to the use of inclusion principles in type checking and partly due to a better understanding of functional types. Note how extensive use of the notional type [A] plays a very important role in System-F.

Appendix B

This appendix is intended to let readers have a better insight into our implementation of System-F (applied to Reckon). Since it would be extremely tedious to define the semantics of all routines, we shall concentrate on those routines that best represent the concepts of System-F. So we omit definitions of routines that are repetitions of those already defined. The routines we shall define are parser,monadictypecheck(MTC), poly and include. Since the transformer is just a standard SECD machine which calls the type checker in the same way as the parser, its definition is omitted here. The routine poly is the part of the type checker responsible for monadic operations, (f x), and we use it to illustrate how the system handles polymorphic functions. The routine MTC is the interface between parser and poly (correspondingly there is an interface between transformer and poly). The routine include defines the rules of inclusion stated in chapter 2.

For this document we adopt the following abbreviations of λ-expressions in addition to those which have been incorporated into the 66/6400 implementation of the parser described here.

We write "f.g x" for "f(g(x))".
We enclose comments in "{" and "}".
We write ϕ for "()"

We write "1st, 2nd, 3rd, ..." for "1 TH, 2 TH, 3 TH, ...". We write "x', x'', x''', ..." for "1 TL x, 2 TL x, 3 TL x, ...".

We write "CASE x IN $x_1 \to f_1$ x OR $x_2 \to f_2$ x... OUT f_0 x ESAC" for "IF $x=x_1$ THEN f_1 x ELIF $x=x_2$ THEN f_2 x ELIF ... ELSE f_0 x FI", and we write ELIF for ELSE IF. Notice that it is possible for x_i to be a set of alternatives, for example, x_i may be $(x_{i1}, x_{i2}, \ldots)$, and in that case "$x=x_i$" should be interpreted as "$x=x_{i1}$ or $x=x_{i2}$ or ...", (in other words x is a member of $(x_{i1}, x_{i2}, \ldots)$).

We write "LET $x_1,\ldots,x_n \equiv a_1,\ldots,a_n$;

WHILE $p(x_1,\ldots,x_n)$ DO $x_1 := f_1(x_1,\ldots,x_n)$;

...;

$x_n := f_n(x_1,\ldots,x_n)$

OD;

$g(x_1,\ldots,x_n)$"

to abbreviate general recursions

"LET REC $f(x_1,\ldots,x_n) \equiv$ IF $p(x_1,\ldots,x_n)$ THEN

$f(f_1(x_1,\ldots,x_n),\ldots,f_n(x_1,\ldots,x_n))$

ELSE $g(x_1,\ldots,x_n)$

FI;

$f(a_1,\ldots,a_n)$"

Machine representations of types are variants of records in Pascal, "domain", "range" and "unionelements" are some of the field selectors we use in the definitions.

```
LET output,input,stack≡ϕ,readinputϕ,ϕ;
  WHILE NOT(endϕ)
  DO
    LET lp,rp≡leftprecedence.lst stack,rightprecedence.lst input;
      IF lp=rp THEN input:=input'; lst stack:=listcounts(lst stack)
      ELIF lp<rp THEN
        LET w≡lst input;
          stack:=CASE w
                 IN(variables,numbers,strings)→w..stack
                 OR keywords→IF closebracket w THEN stack'
                             ELSE w..stack FI
                 ESAC;
          input:=input'
      ELSE
        LET w≡lst stack;
          CASE w
          IN(variables,numbers,strings)→
            output:=w..output; stack:=stack'
          OR keywords→
            LET op≡lookup(w,parseenvironment);
              output,stack:=op(output,stack)
          ESAC
      FI
  OD;
transform(output{which has now been parsed})


WHERE parseenvironment≡(("LET",consblock),(":",consλexp),("+",dyadcheck),
                        (".",monadictypecheck{see below}),...)
```

```
WHERE monadictypecheck(o,s)≡
 (LET functy,argty≡type.2nd o,type.1st o;
   IF functional functy THEN
     LET w1,w2≡include(domain functy,argty,0);
       IF w1=1∨w1=2 THEN
         (assigntype((range functy),IF w1=2 THEN insertcodetochecktype-
                                    atruntime ELSE identityfunction FI.
                                    consmonadcode IF w2=2 THEN insert-
                                    coercioncode o ELSE o FI
                     ),s')
       ELSE conserrorcode(o,s)
       FI
   ELSE
     LET w1,w2≡poly(functy,argty) {to sort our polymorphic functions};
       CASE w1
       IN 1→(o,changetomultiplyop s)
       OR(2,11)→(IF w1=11 THEN swap o ELSE o FI,
                 changetostringselectionop s)
       OR(3,21)→(IF w1=21 THEN swap o ELSE o FI,
                 changetolistselectionop s)
       OR(4,5,14,15,24,25,30,40)→
        (assigntype(w2,insertcode-to-checktypeatruntime.consmonadcode o)
         ,s')
       OR 23→(o,changetoconcatlistop s)
       OR 12→(o,changetoconcatstringop s)
       OUT conserrorcode(o,s)
       ESAC
   FI );


WHERE REC poly(p,q)≡
  CASE p
  IN [ANY]→(40,[A])
  OR numbers→
    CASE q
    IN [ANY]→(4,[A])
    OR numbers→(1,type2(p,q))
    OR strings→(2,[S])
    OR lists→
```

```
    (LET n≡length q;
      LET q1,...qn≡q;
        LET t1,...,tn≡type q1,...,type qn;
          IF t1=t2=...=tn THEN (3,t1) ELSE (3,t1∪t2 ...∪tn) FI)
  OR unions→
    (LET q,r≡unionelements q,ϕ;
      WHILE NOT(NULL q)
      DO
        LET w1,w2≡poly(p,1st q);
          q:=q'; r:=consunion(w2,r)
      OD;
      IF NULL r THEN (0,ϕ) ELSE (5,r) FI)
  OUT (0,ϕ)
  ESAC
OR strings→
  CASE q
  IN [ANY]→(14,[S])
  OR numbers→(11,[S])
  OR strings→(12,[S])
  OR unions→
    LET q,r≡unionelements q,ϕ;
      WHILE NOT(NULL q)
      DO
        LET w1,w2≡poly(p,1st q);
          q:=q'; r:=consunion(w2,r)
      OD;
      IF NULL r THEN (0,ϕ) ELSE (15,r) FI
  OUT (0,ϕ)
  ESAC
OR lists→
  CASE q
  IN [ANY]→(24,[A])
  OR numbers→
    (LET n≡length p;
      LET p1,...,pn≡p;
        LET t1,...,tn≡type p1,...,type pn;
          IF t1=...=tn THEN (21,t1) ELSE (21,t1∪t2 ...∪tn) FI)
  OR lists→(23,type2(p,q))
```

```
  OR unions→
    (LET q,r≡unionelements q,ϕ;
     WHILE NOT(NULL q)
     DO
       LET w1,w2≡poly(p,1st q);
         q:=q'; r:=consunion(w2,r)
     OD;
     IF NULL r THEN (0,ϕ) ELSE (25,r) FI)
  OUT (0,ϕ)
  ESAC
OR functionals→
  LET w1,w2≡include(domain p,q,1);
    (0,IF w1=0 THEN ϕ ELSE range p FI)
OR unions→
  LET p,r≡unionelements p,ϕ;
    WHILE NOT(NULL p)
    DO
      LET w1,w2≡poly(1st p,q);
        p:=p'; r:=consunion(w2,r)
    OD;
    IF NULL r THEN (0,ϕ) ELSE (30,r) FI
ESAC;


{end of definition: monadictypecheck}
```

```
WHERE REC include(f{formal parameter},a{actual parameter},c{coercion
                  state})≡
  CASE f
  IN primitives→
    (LET REC pin(f,x,c)≡
      CASE x
      IN primitives→
        IF x=[ANY] THEN (2,c) ELIF COERCIBLEPRIMITIVES(f,x,c)
        THEN (1,2) ELSE (0,c) FI
      OR unions→
        (LET x≡unionelements x;
          WHILE
            LET w1,w2≡pin(f,1st x,c); (NOT(NULL x))∧(w1=0)
          DO x:=x' OD; IF NULL x THEN (0,c) ELSE (2,c) FI)
      OUT (0,c)
      ESAC;
      pin(f,a,c))
  OR lists→
    CASE x
    IN primitives→IF x=[ANY] THEN (2,c) ELSE (0,c) FI
    OR unions→
      (LET x,c≡unionelements x,IF c=0 THEN 3 ELSE c FI;
        WHILE
          LET w1,w2≡2,0;
            IF NOT(NULL x) THEN w1,w2:=include(f,1st x,c) FI; w1=0
        DO x:=x' OD;
        IF NULL x THEN (0,c) ELSE (2,c) FI)
    OR lists→
      (LET l1,l2≡elementtype f,elementtype x;
        IF longlist f THEN
          IF longlist x THEN
            LET f,x,d≡l1,l2,1;
              WHILE
                LET w1,w2≡0,0;
                  IF NOT(NULL f) THEN w1,w2:=include(1st f,1st x,c);
                    IF w2=2 THEN 1st x:=insertcoercioncode.1st x FI;
                    w1≠0
                  FI
              DO f:=f'; x:=x'; d:=IF w1=2 THEN 2 ELSE d FI OD;
```

```
          IF NULL f THEN (d,c) ELSE (0,c) FI
        ELSE
          LET f,x,c≡11,12,1;
            WHILE
              LET w1,w2≡0,0;
                IF NOT(NULL f) THEN w1,w2:=include(1st f,x,c) FI;
                w1≠0
            DO f:=f' OD;
            IF NULL f THEN (2,c) ELSE (0,c) FI
        FI
      ELSE
        IF longlist x THEN
          LET f,x,d≡11,12,1;
            WHILE
              LET w1,w2≡0,0;
                IF NOT(NULL x) THEN w1,w2:=include(f,1st x,c);
                  IF w2=2 THEN 1st x:=insertcoercioncode.1st x FI;
                  w1≠0
                FI
            DO x:=x'; d:=IF w1=2 THEN 2 ELSE d FI OD;
            IF NULL x THEN (d,c) ELSE (0,c) FI
        ELSE include(11,12,1)
        FI
      FI)
  OUT (0,c)
  ESAC
OR functionals→
 (LET c≡1;
    CASE x
    IN [ANY]→(2,c)
    OR functionals→
     (LET w1,w2≡include(domain x,domain f,1);
        IF w1=1 THEN include(range f,range x,1) ELSE (0,c) FI)
    OR unions →
     (LET x≡unionelements x; WHILE LET w1,w2≡2,0;
        IF NOT(NULL x) THEN w1,w2:=include(f,1st x,c); w1=0 FI
      DO x:=x' OD ;
      IF NULL x THEN (0,c) ELSE (2,c) FI)
    OUT (0,c)
    ESAC )
```

```
  OR unions→
    (LET f,d≡unionelements f,0;
      WHILE
        LET w1,w2≡include(1st f,x,c);
          NOT((NULL f)∨(w1=1))
      DO f:=f'; d:=IF w1=2 THEN 2 ELSE d FI OD;
      IF NULL f THEN (d,c) ELSE (1,c) FI)
  OUT(0,c)
  ESAC


{End of definition: include}
```

Appendix C

Programming Examples of System-F

We reproduce here the computer outputs of three programming examples which were tested on CDC 6400. Two tests were conducted for each example, one with type declaration and the other without.

In order to assist readers to examine intermediate type-checking results, we have provided the function "printtype" such that "(printtype x)" will print the type of "x" at compile time (x can be any object allowed in the language except system functions). In our current implementation of System-F, no code is generated for "printtype", in other words, it does not exist at run time. The information printed by "printtype" will be in the format shown below:

Type of x is : t_x

where t_x is the type of x.

The symbol "←" in the computer outputs should be interpreted as "→", and readers should also refer to the table in Appendix A for other differences.

EXAMPLE 1

```
START
BEGINNING

COMMENT
A REPEAT LOOP IS APPLIED TO INTEGER, LIST AND STRING
RESPECTIVELY.    THERE IS NO TYPE DECLARATION IN THIS
EXAMPLE
COMMENTEND

LET REC REPEAT ACTION CONDITION#
  (ACTION(); (CONDITION() THEN () ELSE
   REPEAT ACTION CONDITION));
LET COUNT#0;
LET SUML, SUMS, SUMI #(1, 2, 3, 4), <RECKON>, 0;
LET B LIMIT TOTAL#IF COUNT=LIMIT THEN PRINT TOTAL;
  COUNT:=0; TRUE ELSE FALSE FI;
LET SQ()#(COUNT:=COUNT+1; SUMI:=SUMI+COUNT*COUNT);
LET SPACE()#(COUNT:=COUNT+1;
  SUMS:=(1 TL SUMS)::< >::(1 TH SUMS));
LET TREBLE()#(COUNT:=COUNT+1;
  (COUNT TH SUML):=(COUNT TH SUML)*3);
REPEAT(PRINTTYPE SQ)(PRINTTYPE $():B 10 SUMI);
REPEAT(PRINTTYPE TREBLE)(PRINTTYPE $(): B 4 SUML);
REPEAT(PRINTTYPE SPACE)(PRINTTYPE $():B 6 SUMS)
ENDING
FINISH

TYPE OF SQ IS : [[NILS]←[INTEGER]]

TYPE OF ($ () : ((B 10) SUMI)) IS : [[NILS]←[BOOLEAN]]

TYPE OF TREBLE IS : [[NILS]←[[INTEGER] UNION [REAL]]]

TYPE OF ($ () : ((B 4) SUML)) IS : [[NILS]←[BOOLEAN]]

TYPE OF SPACE IS : [[NILS]←[STRING]]

TYPE OF ($ () : ((B 6) SUMS)) IS : [[NILS]←[BOOLEAN]]

RESULT OF TYPE CHECKING IS : [ANY]

<<<OUTPUT IS : 385 >>>
<<<OUTPUT IS : (3, 6, 9, 12) >>>
<<<OUTPUT IS : < R E C K O N> >>>

RESULT OF PROGRAM IS : ()
QED.
```

EXAMPLE 2

```
START
BEGINNING

COMMENT
THIS EXAMPLE IS SIMILAR TO LAST ONE, BUT WITH TYPE
DECLARATIONS.
COMMENTEND

LET REC [[[NILS]←[ANY]]←[[[NILS]←[BOOLEAN]]←[NILS]]]REPEAT
  [[NIL]←[ANY]]ACTION [[NILS]←[BOOLEAN]]CONDITION#
  (ACTION(); (CONDITION() THEN () ELSE
   REPEAT ACTION CONDITION));
LET COUNT#0;
LET SUML, SUMS, SUMI#(1, 2, 3, 4), <RECKON>, 0;
LET B LIMIT TOTAL#IF COUNT=LIMIT THEN PRINT TOTAL;
  COUNT:=0; TRUE ELSE FALSE FI;
LET SQ()#(COUNT:=COUNT+1; SUMI:=SUMI+COUNT*COUNT);
LET SPACE()#(COUNT:=COUNT+1;
  SUMS:=(1 TL SUMS)::< >::(1 TH SUMS));
LET TREBLE()#(COUNT:=COUNT+1;
  (COUNT TH SUML):=(COUNT TH SUML)*3);
REPEAT(PRINTTYPE SQ)(PRINTTYPE $():B 10 SUMI);
REPEAT(PRINTTYPE TREBLE)(PRINTTYPE $(): B 4 SUML);
REPEAT(PRINTTYPE SPACE)(PRINTTYPE $():B 6 SUMS)
ENDING
FINISH

TYPE OF SQ IS : [[NILS]←[INTEGER]]

TYPE OF ($ () : ((B 10) SUMI)) IS : [[NILS]←[BOOLEAN]]

TYPE OF TREBLE IS : [[NILS]←[[INTEGER] UNION [REAL]]]

TYPE OF ($ () : ((B 4) SUML)) IS : [[NILS]←[BOOLEAN]]

TYPE OF SPACE IS : [[NILS]←[STRING]]

TYPE OF ($ () : ((B 6) SUMS)) IS : [[NILS]←[BOOLEAN]]

RESULT OF TYPE CHECKING IS : [NILS]

<<<OUTPUT IS : 385 >>>
<<<OUTPUT IS : (3, 6, 9, 12) >>>
<<<OUTPUT IS : < R E C K O N> >>>

RESULT OF PROGRAM IS : ()
QED.
```

EXAMPLE 3

```
START
BEGINNING

COMMENT
A WHILE-LOOP IS APPLIED TO REAL, LIST AND STRING
RESPECTIVELY.    THERE IS NO TYPE DECLARATION IN
THIS EXAMPLE.
COMMENTEND

LET REC WHILE ACTION CONDITION#
  (CONDITION() THEN ACTION(); WHILE ACTION CONDITION
  ELSE () );
LET INCREMENT, COUNT, SUM#1.0, 1, 0.0;
LET L, S#(1, 2, 3, 4, 5), <WHILELOOP>;
LET CONTROL()#COUNT:=COUNT+1;
LET LENGTH N P#
  (NULL P THEN PRINT N; FALSE ELSE
    (NULL(N TL P) THEN PRINT N; FALSE ELSE TRUE));
LET SERIES N P#
  (N\0.00005 THEN TRUE ELSE PRINT N; PRINT P; FALSE);
WHILE (PRINTTYPE CONTROL)(PRINTTYPE $():LENGTH COUNT L);
COUNT:=1;
WHILE CONTROL (PRINTTYPE $():LENGTH COUNT S);
WHILE (PRINTTYPE $():(INCREMENT:=INCREMENT*0.5;
                      SUM:=SUM+INCREMENT))
      (PRINTTYPE $():SERIES INCREMENT SUM)
ENDING
FINISH

TYPE OF CONTROL IS : [[NILS]→[INTEGER]]

TYPE OF ($ () : ((LENGTH COUNT) L)) IS : [[NILS]→[BOOLEAN]]

TYPE OF ($ () : ((LENGTH COUNT) S)) IS : [[NILS]→[BOOLEAN]]

TYPE OF ($ () : (INCREMENT := (INCREMENT * 0.5));
(SUM := (SUM + INCREMENT))) IS : [[NILS]→[REAL]]

TYPE OF ($ () : ((SERIES INCREMENT) SUM)) IS : [[NILS]→
[BOOLEAN]]


RESULT OF TYPE CHECKING IS : [ANY]

<<<OUTPUT IS : 5 >>>
<<<OUTPUT IS : 9 >>>
<<<OUTPUT IS : 0.0000305175 >>>
<<<OUTPUT IS : 0.9999694824 >>>

RESULT OF PROGRAM IS : ()
QED.
```

EXAMPLE 4

```
START
BEGINNING

COMMENT
THIS EXAMPLE IS SIMILAR TO LAST ONE, BUT WITH TYPE
DECLARATION.
COMMENTEND

LET REC [[[NILS]←[ANY]]←[[[NILS]←[BOOLEAN]]←[NILS]]]WHILE
[[NILS]←[ANY]]ACTION [[NILS]←[BOOLEAN]]CONDITION#
  (CONDITION() THEN ACTION(); WHILE ACTION CONDITION
  ELSE () );
LET INCREMENT, COUNT, SUM#1.0, 1, 0.0;
LET L, S#(1, 2, 3, 4, 5), <WHILELOOP>;
LET CONTROL()#COUNT:=COUNT+1;
LET LENGTH N P#
  (NULL P THEN PRINT N; FALSE ELSE
    (NULL(N TL P) THEN PRINT N; FALSE ELSE TRUE));
LET SERIES N P#
  (N\0.00005 THEN TRUE ELSE PRINT N; PRINT P; FALSE);
WHILE (PRINTTYPE CONTROL)(PRINTTYPE $():LENGTH COUNT L);
COUNT:=1;
WHILE CONTROL (PRINTTYPE $():LENGTH COUNT S);
WHILE (PRINTTYPE $():(INCREMENT:=INCREMENT*0.5;
                      SUM:=SUM+INCREMENT))
      (PRINTTYPE $():SERIES INCREMENT SUM)
ENDING
FINISH

TYPE OF CONTROL IS : [[NILS]←[INTEGER]]

TYPE OF ($ () : ((LENGTH COUNT) L)) IS : [[NILS]←[BOOLEAN]]

TYPE OF ($ () : ((LENGTH COUNT) S)) IS : [[NILS]←[BOOLEAN]]

TYPE OF ($ () : (INCREMENT := (INCREMENT * 0.5));
(SUM := (SUM + INCREMENT))) IS : [[NILS]←[REAL]]

TYPE OF ($ () : ((SERIES INCREMENT) SUM)) IS : [[NILS]←
[BOOLEAN]]


RESULT OF TYPE CHECKING IS : [NILS]

<<<OUTPUT IS : 5 >>>
<<<OUTPUT IS : 9 >>>
<<<OUTPUT IS : 0.0000305175 >>>
<<<OUTPUT IS : 0.9999694824 >>>

RESULT OF PROGRAM IS : ()
QED.
```

```
EXAMPLE 5


START
BEGINNING

COMMENT
PROCEDURE "PRODUCE" AND "CONSUME" ARE MUTUALLY RECURSIVE
TO EACH OTHER.    THERE IS NO TYPE DECLARATION IN THIS
EXAMPLE.
COMMENTEND

LET FACTORY B1 B2 A1 A2 D#
  (LET BUFFERP,BUFFERC#1,1;
    (LET REC(PRODUCE,CONSUME)#
     (($():(PRINTTYPE B1() THEN BUFFERP:=0 ELSE A1());
           (PRINTTYPE BUFFERC=1 THEN CONSUME() ELSE D())),
      ($():(PRINTTYPE B2() THEN BUFFERC:=0 ELSE A2());
        (PRINTTYPE BUFFERP=1 THEN PRODUCE() ELSE D()))));
    PRODUCE() ));
LET SUM,N#0,0;
  (PRINTTYPE FACTORY)($():N\100)($():SUM\1000)
                    ($():N:=N+1)($():SUM:=SUM+N*N)
                    ($():PRINT N; PRINT SUM)
ENDING
FINISH

TYPE OF COND (B1 ()) IN (BUFFERP :=0) OUT (A1 ())
DNOC IS : [ANY]

TYPE OF COND (BUFFERC = 1) IN (CONSUME ()) OUT (D ())
DNOC IS : [ANY]

TYPE OF COND (B2 ()) IN (BUFFERC := 0) OUT (A2 ())
DNOC IS : [ANY]

TYPE OF COND (BUFFERP = 1) IN (PRODUCE ()) OUT (D ())
DNOC IS : [ANY]

TYPE OF FACTORY IS : [[ANY]←[[ANY]←[[ANY]←[[ANY]←[[ANY]←
[ANY]]]]]]

RESULT OF TYPE CHECKING IS : [ANY]

<<<OUTPUT IS : 16 >>>
<<<OUTPUT IS : 1015 >>>

RESULT OF PROGRAM IS : 1015
QED.
```

EXAMPLE 6

```
START
BEGINNING

COMMENT
THIS EXAMPLE IS SIMILAR TO LAST ONE, BUT WITH TYPE
DECLARATIONS.
COMMENTEND

LET FACTORY [[NILS]←[BOOLEAN]]B1 [[NILS]←[BOOLEAN]]B2
    [[NILS]←[ANY]]A1 [[NILS]←[ANY]]A2 [[NILS]←ANY]]D#
  (LET BUFFERP, BUFFERC#1, 1;
    (LET REC([[NILS]←[ANY]]PRODUCE, [[NILS]←[ANY]]CONSUME)#
     (($():(PRINTTYPE B1() THEN BUFFERP:=0 ELSE A1());
           (PRINTTYPE BUFFERC=1 THEN CONSUME() ELSE D())),
      ($():(PRINTTYPE B2() THEN BUFFERC:=0 ELSE A2());
        (PRINTTYPE BUFFERP=1 THEN PRODUCE() ELSE D()))));
    PRODUCE() ));
LET SUM,N#0,0;
  (PRINTTYPE FACTORY)($():N\100)($():SUM\1000)
                    ($():N:=N+1)($():SUM:=SUM+N*N)
                    ($():PRINT N; PRINT SUM)
ENDING
FINISH

TYPE OF COND (B1 ()) IN (BUFFERP :=0) OUT (A1 ())
DNOC IS : [ANY]

TYPE OF COND (BUFFERC = 1) IN (CONSUME ()) OUT (D ())
DNOC IS : [ANY]

TYPE OF COND (B2 ()) IN (BUFFERC := 0) OUT (A2 ())
DNOC IS : [ANY]

TYPE OF COND (BUFFERP = 1) IN (PRODUCE ()) OUT (D ())
DNOC IS : [ANY]

TYPE OF FACTORY IS : [[[NILS]←[BOOLEAN]]←[[[NILS]←
[BOOLEAN]]←[[[NILS]←[ANY]]←[[[NILS]←[ANY]]←[[[NILS]←
[ANY]]←[ANY]]]]]]

RESULT OF TYPE CHECKING IS : [ANY]

<<<OUTPUT IS : 16 >>>
<<<OUTPUT IS : 1015 >>>

RESULT OF PROGRAM IS : 1015
QED.
```

PART THREE

CHAPTER ONE

SYSTEM-Y

Although [A] or [A→A] as proposed in System-F enable us to tackle (or to be more precise, "bypass") problems such as circular types and parametric polymorphism, they add little to our understanding of the nature of these types. In this chapter, we shall examine this important topic in detail. We shall discuss the requisite reduction mechanisms and illustrate them by examples.

1.1 Elements of System-Y

Elements of System-Y are grouped into three main categories--type constants, type variables and type abstractions. A type constant may be basic or functional as in the other systems. A computing variable will be mapped into a fresh type variable if the type of that computing variable is not known (in System-F, it was given default type [A]). Two or more computing variable of the same name and within the same scope (in other words, bound to the same binding variable) are mapped into the same type variable otherwise into different ones. Type variables will be assigned type values in the course of type reduction.

Type abstractions are used to describe the type of functions especially for those whose formal parameters' types are not declared. This chapter discusses their properties.

Notationally, we shall write t_i for type constants, V_j for type variables and γ_j for type abstractions. Usually the subscript "i" or "j" bears some useful information, for example t_+ is the type expression for the dyadic operator "+", V_x is probably the type variable corresponding to the computing variable "x" and γ_x is the type of a function with "x" as formal parameter (binding variable of λ-expression) assuming that we are considering single argument λ-expressions only.

1.2 Type abstractions

In System-F we said that the type of (λx:x) is [A→A]. In fact, it is only partly true, the function may be able to take arguments of any type, but the result it yields is not any type as it seems to be. The type of the result depends on the type of the actual parameter. So this suggests that a proper type expression for functions should be one that can express this context- sensitive relationship. For example, immediately we can exclude simple type variables as types for functions.

Type abstractions can be defined by equations. If Φ is the mapping function which maps a computing expression into a type expression, then

$$\Phi(\lambda x{:}M) = \gamma_x$$

where M is a well formed expression and γ_x is a type

abstraction, γ_x is then defined by the following equation:

$$\gamma_x(V_x)=V_x\rightarrow\phi(M)$$

where V_x is a type variable and $\phi(x)=V_x$.

Examples

(1) $\Phi(\lambda x:x)=\gamma_x$

where $\gamma_x(V_x)=V_x\rightarrow\phi(x)=V_x\rightarrow V_x$

(2) $\Phi(\lambda x:x+x)=\gamma_x$

where $\gamma_x(V_x)=V_x\rightarrow\phi(x+x)$

$=V_x\rightarrow(t_+\ V_x\ V_x)$

1.3 Type assignments

There are two ways that values can be assigned to type variables--by declaration and by application of type abstractions to types, which will be denoted by ":≡" and ":=" respectively.

1.3.1 Assignment by declaration

In the example $(\lambda[I]x:x)$, x is declared to be of type [I]. Suppose $\Phi(x)=V_x$, then we say that V_x and [I] are synonyms and so we write $V_x:\equiv[I]$, (readers may compare this with the identity declaration of Algol-68).

Let γ_x be the type of $(\lambda[I]x:x)$. This is defined to be in the type abstraction

$$\gamma_x(V_x)=V_x\rightarrow V_x.$$

However, by declaration, V_x cannot be of any value but [I], therefore γ_x can only have one value, which is [I→I]. So γ_x and [I→I] are also synonymous.

Notationally, we can write $\Phi(\lambda[t]x:M)=(V_x:\equiv t;\ t\rightarrow\phi(M))$, where t is any type and M is a well formed formula. We notice that the result of the mapping consists of two clauses separated by ";". The first clause is an imperative instruction to indicate the kind of assignment involved and the second clause is the result we normally have.

1.3.2 Assignment by application of type abstraction to types

If the square function is defined as sq(x)=x*x, we can assume (sq 3) is evaluated according to the following process,

(1) assign the value 3 to x

(2) replace every x in the equation by its value

(3) evaluate the right side of the definition

(4) finally obtain (sq 3)=9

Suppose $\gamma_i(V_i)=V_i\rightarrow\phi(M)$, $(\gamma_i\ V_i)$ can be analogously evaluated as follows:

(1) assign value t_i to V_i (i.e. $V_i := t_i$)

(2) replace every V_i in the equation by t_i

(3) evaluate (by reduction in this case) the right side of the equation

(4) finally obtain $(\gamma_i\ t_i) = t_i \rightarrow t_i'$, assuming that the reduction on (ΦM) yields t_i'.

In general, whenever a type abstraction γ_i is applied to an actual parameter t, such as $(\gamma_i\ t_i)$ above, the above process will be invoked. We shall see later how this can be avoid in certain cases by providing a memory for γ_i.

1.4 The mapping function Φ

The mapping function Φ is designed to transform an applicative expression into a corresponding type expression in System-Y. Φ can be defined recursively as follows for the various kinds of computing expression:

(1) Constants(c)

$(\Phi\ c) = t_i$ where t_i is a type constant

Examples

(1) $(\Phi\ 3) = [I]$

(2) $(\Phi\ +) = t_+$

(2) Variables(x)

$(\Phi\ x)=V_x$, for simplicity, we assume that there will be no name conflicts among the binding variables.

(3) Combinations (M N)

$(\Phi(M\ N))=((\Phi\ M)(\Phi\ N))$

(4) Infixed expressions (x op y)

Assuming that (x op y) can be rewritten as (op x y), then, $(\Phi(x\ op\ y))=(\Phi(op\ x\ y))=((\Phi\ op)(\Phi\ x)(\Phi\ y))$

(5) Conditional expressions (IF b THEN e_1 ELSE e_2 FI)

$(\Phi(IF\ b\ THEN\ e_1\ ELSE\ e_2\ FI))=(t_{cond}(\Phi\ b)(\Phi\ e_1)(\Phi\ e_2))$

where t_{cond} is the type constant for handling conditional expressions, the reduction routine associated with it may vary from one implementation to another.

(6) λ-expressions (λx:M)

$(\Phi(\lambda x:M))=\gamma_x$ where $(\gamma_x\ V_x)=[V_x \rightarrow (\Phi\ M)]$

This mapping function definition may be enlarged for more ambitious specific languages.

1.5 The Reduction Function

In the following discussion, we shall assume all "t", "V" and "γ" with or without subscripts are any type constants, type variables and type abstractions respectively except in those cases mentioned in section(1.1) where specific meaning is attached to some subscripts. These cases are either as we have met them before in previous examples or they should be clear from the context.

The reduction function θ is defined recursively as follows:

(RY1) $(\theta\ t_i)=t_i$

(RY2) $(\theta\ V_i)=(\text{lookup}\ V_i)$
where lookup is the function that returns the value that has been assigned by ":≡" or ":=" to V_i, if any.

(RY3) $(\theta\ \gamma_i)=\gamma_i$

(RY4) $(\theta(M\ N))=(\theta_2(\theta\ M)(\theta\ N))$
where M and N are any type expressions. The action of θ_2 depends on the nature of M and N. The following are some of the possible cases:

(a) both M and N are type constants
θ_2 reduces the combination according to an appropriate rule of reduction as described in System-F, for example,

$(\theta_2\ [I \to R\ ][I\ ]) = [R\ ]$

(b) M is a type abstraction:

let M be γ_m where $\gamma_m(V_m) = V_m \to M'$, and M' is a type expression, then

$(\theta_2\ M\ N) = (V_m := N;\ (\theta\ M'))$

for example,

$(\theta(\phi((\lambda x{:}x+x)\ 3)))$

$= (\theta(\gamma_x\ [I\ ]))$ where $\phi(x) = V_x$ and $\gamma_x(V_x) = V_x \to (t_+\ V_x\ V_x)$

$= (\theta_2\ \gamma_x\ [I\ ])$ by (RY4)

$= (V_x := [I\ ];\ (\theta(t_+\ V_x\ V_x)))$ by (RY4.b)

$= (\theta_3(\theta\ t_+)(\theta\ V_x)(\theta\ V_x))$ by (RY5--see below)

$= (\theta_3\ t_+\ [I\ ][I\ ])$ by (RY1,RY2)

$= [I\ ]$ by (RY5)

(c) M is a type constant and N is a type abstraction:

Suppose M is $t_1 \to t_2$, let N be γ_n and $(\gamma_n\ V_n) = V_n \to N'$, then

$(\theta_2\ M\ N) = (\theta_2'\ t_1\ (\theta\ N)\ t_2)$

where θ_2' is defined recursively as follows:

Given $(\theta_2'\ a\ b\ c)$,

(1) if b is a type abstraction and a is $[t_{11} \to t_{12}]$, let b be γ_b where $\gamma_b(V_b) = V_b \to b'$, then

$(\theta_2'\ a\ b\ c) = (V_b := t_{11};\ (\theta_2'\ t_{12}\ (\theta\ b')\ c))$

(2) otherwise if $a \supseteq b$ then c else [E]

for example,

$(\theta(\Phi(\lambda[I\to[I\to I]]f:((f\ 3)4))(\lambda a:\lambda b:a+b)))$

$=(\theta([[I\to[I\to I]]\to I]\ \gamma_a))$

where $\gamma_a(V_a)=V_a\to\gamma_b$

and $\gamma_b(V_b)=V_b\to(t_+\ V_a\ V_b)$

$=(\theta_2[[I\to[I\to I]]\to I]\ \gamma_a)$ by (RY4)

$=(\theta_2'\ [I\to[I\to I]]\ \gamma_a\ [I])$ by (RY4c)

$=(V_a:=[I];\ (\theta_2'\ [I\to I]\ \gamma_b\ [I]))$ by (RY4c1)

$=(V_b:=[I];\ (\theta_2'\ [I]\ (\theta(t_+\ V_a\ V_b))\ [I]))$ by (RY4c1)

$=(\theta_2'[I](\theta_3\ t_+[I][I])[I])$ by (RY5,RY1,RY2)

$=(\theta_2'[I][I][I])$ by (RY5)

$=[I]$ by (RY4c2)

(RY5) $(\theta(t_{dyadic}\ t_x\ t_y))=(\theta_3(\theta\ t_{dyadic})(\theta\ t_x)(\theta\ t_y))$

where θ_3 takes three arguments, and the first argument is an element of the set of predefined type-checking instructions (probably represented by type constants) so that the predefined routines will be applied to the second and third arguments, producing a result which is the type of the dyadic operations.

(RY6) $(\theta(t_{cond}\ t_b\ t_x\ t_y))=(\theta_4(\theta\ t_{cond})(\theta\ t_b)(\theta\ t_x)(\theta\ t_y))$

θ_4 is similar to θ_3 except that it takes 4 arguments. As said before the type checking

routine is denoted by t_{cond}.

Here are some examples,

(1)

$(\theta(\Phi(((\lambda a:\lambda b:a+b)3)4)))$

$=(\theta((\gamma_a[I])[I]))$ where $(\gamma_a\ V_a)=V_a\rightarrow\gamma_b$

and $(\gamma_b\ V_b)=V_b\rightarrow(t_+\ V_a\ V_b)$

$=(\theta_2(\theta(\gamma_a[I]))[I])$

$=(\theta_2(\theta_2\gamma_a[I])[I])$ by (RY4)

$=(V_a:=[I];\ (\theta_2(\theta\ \gamma_b)[I]))$ by (RY4b)

$=(\theta_2\ \gamma_b\ [I])$ by (RY3)

$=(V_b:=[I];\ (\theta(t_+\ V_a\ V_b)))$ by (RY4b)

$=(\theta_3\ t_+(\theta\ V_a)(\theta\ V_b))$ by (RY5)

$=(\theta_3\ t_+[I][I])$ by (RY2)

$=[I]$

(2) parametric polymorphism

As before, assume that the type of numericfunction, stringfunction, numeral and string are $[t_n\rightarrow t_n]$, $[t_s\rightarrow t_s]$, t_n and t_s respectively,

$(\theta(\Phi((\lambda twice:(...((twice\ numericfunction)numeral)...$

$((twice\ stringfunction)string)...))$

$(\lambda f:\lambda x:(f(f\ x))))))$

$=(\theta(\gamma_{tw}\ \gamma_f))$

$=(\theta_2\ \gamma_{tw}\ \gamma_f)$

$=(V_{tw}:=\gamma_f;\ (\theta(...((V_{tw}[t_n\rightarrow t_n])[t_n])...$

$((V_{tw}[t_s\rightarrow t_s])[t_s])...)))$

$=(\ldots(\theta((V_{tw}[t_n \to t_n])[t_n]))\ldots(\theta((V_{tw}[t_s \to t_s])[t_s]))\ldots)$

$=(\ldots(\theta_2(\theta(V_{tw}[t_n \to t_n]))[t_n])\ldots(\theta(\text{ ditto }))\ldots)$

$=(\ldots(\theta_2(\theta_2\ \gamma_f[t_n \to t_n])[t_n])\ldots(\theta(\text{ ditto }))\ldots)$

$=(V_f:=[t_n \to t_n];\ (\ldots(\theta_2(\theta\ \gamma_x)[t_n])\ldots(\theta(\text{ ditto }))\ldots))$

$=(\ldots(\theta_2\ \gamma_x\ [t_n])\ldots(\theta(\text{ ditto }))\ldots)$

$=(V_x:=[t_n];\ (\ldots(\theta(V_f(V_f\ V_x)))\ldots(\theta(\text{ ditto }))\ldots))$

$=(\ldots(\theta_2[t_n \to t_n](\theta_2[t_n \to t_n][t_n]))\ldots(\theta(\text{ ditto }))\ldots)$

$=(\ldots t_n\ldots(\theta((V_{tw}[t_s \to t_s])[t_s]))\ldots)$

$=(\ldots t_n\ldots(\theta_2(\theta_2(\theta\ V_{tw})[t_s \to t_s])[t_s])\ldots)$

$=(\ldots t_n\ldots(\theta_2(\theta_2\ \gamma_f[t_s \to t_s])[t_s])\ldots)$

$=(V_f:=[t_s \to t_s];\ (\ldots t_n\ldots(\theta_2\ \gamma_x[t_s])\ldots))$

$=(V_x:=[t_s];\ (\ldots t_n\ldots(\theta(V_f(V_f\ V_x)))\ldots))$

$=(\ldots t_n\ldots(\theta_2[t_s \to t_s](\theta_2[t_s \to t_s][t_s]))\ldots)$

$=(\ldots t_n\ldots t_s\ldots)$

There should be no confusion that V_f, V_x are assigned values in two instances corresponding to two distinct applications of γ_f and γ_x to their arguments. Alternatively, we may subscript the variables in order to differentiate the two instances. Indeed, for implementation, each type abstraction may carry an environment which includes all assignments to the type variables that are accessible to it.

1.6 Simplifications of the reduction function

Let us study the example $(\lambda f:(f\ 1)+(f\ 2))(\lambda n:n+n)$ as follows:

$(\theta(\Phi(\lambda f:(f\ 1)+(f\ 2))(\lambda n:n+n)))$

$=(\theta(\gamma_f\ \gamma_n))$ (1)

$=(\theta_2\ \gamma_f\ \gamma_n)$ (2)

$=(V_f:=\gamma_n;\ (\theta(t_+(V_f[I])(V_f[I]))))$ (3)

$=(\theta_3\ t_+(\theta(V_f[I]))(\theta(V_f[I])))$ (4)

{Now reduce on the first occurrence of $(\theta(V_f[I]))$}

$=(\theta_3\ t_+(\theta_2(\theta\ V_f)[I])(\theta(V_f[I])))$ (5)

$=(\theta_3\ t_+(\theta_2\ \gamma_n\ [I])(\theta(V_f[I])))$ (6)

$=(V_n:=[I];\ (\theta_3\ t_+(\theta(t_+\ V_n\ V_n))(\theta(V_f[I]))))$ (7)

$=(\theta_3\ t_+(\theta_3\ t_+[I][I])(\theta(V_f[I])))$ (8)

$=(\theta_3\ t_+[I](\theta(V_f[I])))$ (9)

{Reduce on the second occurrence of $(\theta(V_f[I]))$}

$=(\theta_3\ t_+[I](\theta_2(\theta\ V_f)[I]))$ (10)

$=(\theta_3\ t_+[I](\theta_2\ \gamma_n\ [I]))$ (11)

$=(V_n:=[I];\ (\theta_3\ t_+[I](\theta(t_+\ V_n\ V_n))))$ (12)

$=(\theta_3\ t_+[I](\theta_3\ t_+[I][I]))$ (13)

$=(\theta_3\ t_+[I][I])$ (14)

$=[I]$ (15)

We notice that steps 10-14 are repetitions of steps 5-9 and the purpose of both processes is to ensure that "f" is applied to apposite arguments. Thus it is reasonable to suggest that if we associate memories to type abstractions, it may be possible to avoid these repetitions. This would be more expeditious if our example were (λf:(f 1)+...+ (f 1000)). The idea of memory is simple, and it requires that the representation of type abstractions consist now of two parts, the definition and the memory.

The memory is used to record values of the type abstraction for some specific arguments. Initially, the memory is empty, and for each abstraction's application, say (γ t), if there is no value of γ for this particular value of t (correspondingly, there is no entry of t in γ's memory), then (γ t) will be reduced as usual by Rule (RY4b) and the result obtained is used to update the memory, otherwise, the result equivalent to application of several reduction rules can be simply extracted from the memory. For example, step(11) can be replaced by (θ_3 t_+[I][I]) so that steps (12-14) are no longer necessary.

There is no formal need to change the reduction rules to include explicitly references to memory because it is in all cases equivalent to the rules already given. It is a pragmatic device to enhance efficiency. We examine its efficiency now.

1.7 System-Y compared with other Static Systems

Type checking of System-Y is performed mainly in the parsing stage. It is reasonable to suggest that System-Y is a static system. The marked difference between it and other static systems is that type declaration is not compulsory in it, a feature not common to other static systems. We may ask whether more processes are required for type checking in System-Y than in static systems where type declaration is compulsory. Before we make any comments on this issue, we shall examine some examples,

Examples

(1) type checking by other systems

How many steps are required in type-checking (λ[I→I]f: (f 1)+(f 2))(λ[I]n:n+n)?

(A1) 3 steps are required in (λ[I→I]f:(f 1)+(f 2)), namely one each for the monadic operations (f 1) and (f 2) and one for the dyadic operation "+".

(A2) one step in (λn:n+n) for the dyadic operation "+"

(A3) one step for the combination (λ[I→I]f:...)(λ[I]n:...)

So 5 steps are required in all.

(2) type checking by System-Y

Consider the number of steps required in type checking our previous example $(\lambda f:(f\ 1)+(f\ 2))(\lambda n:n+n)$ (without type declaration this time)

(B1) Type reduction rule imposes no type checking on combination $(\lambda f:\ldots)(\lambda n:\ldots)$. γ_n will be assigned to V_f instead.

(B2) Similarly, there is no checking on (f 1) itself, but this combination initiates checking of $(\lambda n:n+n)$. As in (A2), one step is required.

(B3) Since the memory of γ_n will not be empty after step (B2), there will be checking of (f 2). In general the number of steps required depends on the number of entries in the memory. However, only one step is required here.

(B4) One step is required for the dyadic operation "+" as in (A1)

Therefore 3 steps are required in all.

This comparision between the performances of System-Y and other static systems was really quite naive. It would be optimistic to assume that System-Y is always more efficient than others. In fact, there are overheads involved in System-Y such as memory space and access time. Any comprehensive comparision of efficiencies might therefore be forced to equate system-Y with other systems.

The comparision of logical significance is that System-Y checks programs in a dynamic order (the order in which they are executed) whereas other systems check them in static order (the order in which they are written).

1.8 Intersection types

We have said before that it is possible to declare the type of any variable. Suppose we want x to be either [I] or [R] in example (λx:x+x), then we can write

$$(\lambda[I \cup R]x:x+x)$$

In system-Y, x is of union type, while the corresponding λ-expression is of intersection type $[I \to I] \cap [R \to R]$, thus the type of the combinations

$\Phi((\lambda[I \cup R]x:x+x)3)$ can be reduced to [I]

and $\Phi((\lambda[I \cup R]x:x+x)3 \cdot 0)$ can be reduced to [R]

Let us use the concept of type abstraction to derive these results. Suppose $(\Phi(\lambda[I \cup R]x:x+x))=\gamma_x$ and $(\Phi\ x)=V_x$, where $\gamma_x(V_x)=V_x \rightarrow (t_+\ V_x\ V_x)$. By declaration, V_x can only be either [I] or [R] and correspondingly γ_x is defined for these two values only,

$$\gamma_x([I])=[I]\rightarrow(t_+[I][I])$$
$$=[I\rightarrow I]$$

$$\gamma_x([R])=[R]\rightarrow(t_+[R][R])$$
$$=[R\rightarrow R]$$

The results obtained are the same as before, and we say that γ_x and $[I\rightarrow I]\cap[R\rightarrow R]$ are synonymous in this example.

1.9 The Anonymous Type

In λ-K Calculus, it is not necessary for binding variables to occur in their λ-bodies. The type of non-occurring binding variables is immaterial. In spite of the fact that they can be of any type, it could be undesirable to assign specific types to them. For example, the generality of combinator K_2 (i.e. "λx:λy:y") would be restricted if we declared the type of its first argument. We want to describe an object whose only property is that it occurs as a binding variable but does not appear in the λ-body. We invent the "anonymous" type (abbreviated [Z]). By allowing [Z] to include all other types (i.e. $[Z]\geq t$, for all types t), its properties can be described by the

following reduction rules (where t, t_1 can be any types):

(RY7) $(\theta_2[Z \rightarrow t_1]t) = t_1$

(RY8) $(\theta_2\ t\ Z) = [E]$

(RY9) $(\theta_2\ Z\ t) = [E]$

If [Z] in (RY8) and (RY9) describes a non-occurring variable there is no reason for it to occur independently in either part of the combination.

If the types of non-occurring variables are not declared, they should be assigned type [Z] by the system. It is possible to assign type [Z] to the empty binding variable "()", but care must be taken here because "()" may occur in λ-bodies. Hence reduction rule (RY7) must be applied before (RY8).

1.10 Implementation

System-Y has been implemented for checking types of a complete subset of Reckon, which includes applicative, binary, sequential and conditional expression (type checking of recursive functions needs further discussions, so we shall leave it until next chapter). These expressions in turn constitute the body of λ-expressions which must be ended with a special marker "endlambda". This marker enable the scanner, parser and type-checker to detect the end of a λ-expression. At present, λ-expressions can only take single argument, so the general format is

"(" "λ" single-binding-variable ":" body-expression
"endlambda" ")"

Meanwhile, let us note some of the features of this implementation.

(1) Scanning

During scanning, each syntactic token will be placed in a record. For example, after scanning, "2+3" will be represented as in diagram(1.1). λ-expressions' scans are stored as sublists, for example, (λn:n+n endlambda) will be represented as in diagram(1.2). This is necessary so that the λ-body will not be parsed if it is not required.

(2) Representations of type abstractions

These contain at least the following information--memory, and definition of γ. Diagram(1.3) shows the schema of representation so that diagram(1.4) illustrates the representation of a fully typed λ-expression.

(3) Parser

Two stacks are required for parsing, one for constructed parse trees and the other for source symbols. Whenever a type value is assigned to a type variable, the pair will be stored in the "environment". A copy of this environment will be made in the corresponding λ-expression when it is parsed. The parser will call the type checker (or part of it) whenever necessary. The relationship between the parser and the type checker is the same as in System-F. Generally, when a λ-expression is parsed, its body will be ignored. Eventually, at some appropriate stages, this λ-body will be considered. This is facilitated by the sublist representation of λ-expression we chose as explained above. All these processes are governed by the reduction rules (θ). During the parse of a λ-body, the parser may be required to work on another λ-body, so it is necessary to keep a list of the unfinished tasks. The list is called "dump". Dumps contain the parser's state at the moment it is interrupted.

If the binding variable of a λ-expression has been declared, the body of that particular λ-expression will be parsed immediately and the result will be stored in the

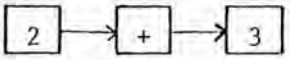


Diagram (1.1)

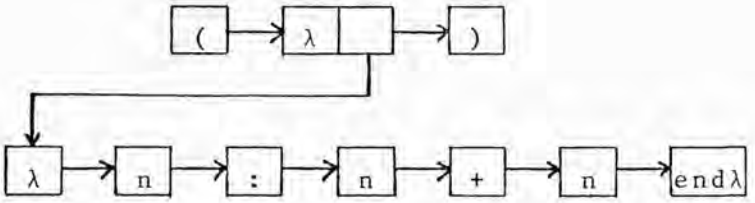


Diagram (1.2)

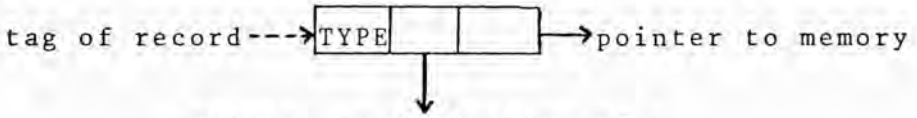


Diagram (1.3)

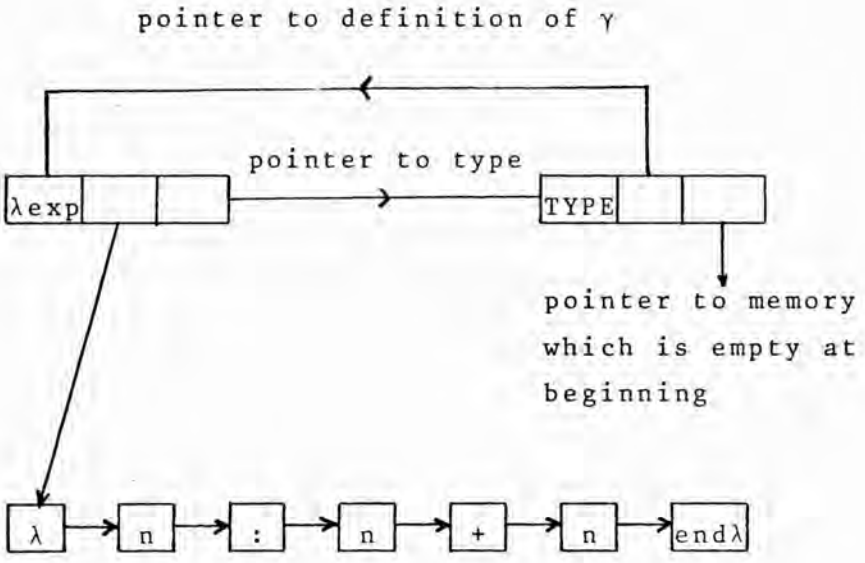


Diagram (1.4)

corresponding memory. In the example $(\lambda[t_1 \cup t_2 \cup \ldots \cup t_n]x{:}\ldots)$, the body of this λ-expression will be parsed n times, so that at the i' th time, x will be assigned type t_i. The results of each parse are stored in the memory accordingly.

During parsing or/and type checking, whenever new information is obtained about the type of a λ-expression, the memory of the corresponding λ-expression is updated with the new information.

(4) Type checker and Parser co-routines

Imprecisely, one may consider the type checker as a subroutine of the parser. More precisely, the relationship between them is that of co-routines, because the type checker is able to modify the actions of the parser as well as vice-versa. There exists some program control registers which are accessible by both type checker and parser. The parser fetches its instruction from the registers. Therefore the type checker simply has to modify the contents of these registers for the parser to follow the course desired by it. Alternatively, the checker can modify the "dump", thereby altering the subsequent tasks. Information can also be passed by the parser to the checker through the same channels.

(5) Machine State

The machine state is represented by the 5-tuple of current values of (the pointers to) the two stacks, input, environment and dump. The initial value of the dump is NIL. Prior to parsing or type checking on a new λ-body, the machine state will be recorded, the five-tuple thus created will form the most recent dump and will be linked to the previous dump formed in a similar way. Correspondingly, the pointers have to be re-set for a new task as follows:

(a) the environment pointer will be set to the environment associated with the new λ-expression

(b) the dump pointer will point to the most recent entry

(c) the input pointer is set to the beginning of the new λ-expression

(d) the two stack pointers will be adjusted accordingly.

1.11 Summary

In this chapter, we have claimed that the types of functions without formal parameter type declarations are best described by type abstractions. These permit a very general kind of type description indeed. The requisite mapping function and reduction rules were described formally. Type checking is done on paper by solving type equations.

Their machine implementation was enhanced by planting a type memory in the representation of type abstractions. We have shown thereby how the problem of polymorphic types is solved by System-Y. Resolution of circular types calls for another chapter.

CHAPTER TWO

SYSTEM-Y FOR CIRCULAR TYPES

System-Y so far fails to handle circular functions in two respects:

(1) no mechanism is provided for declaring the type of circular functions

(2) the type checker would enter an infinite loop if the expression it is reducing contains undeclared circular functions

This problem is mentioned in nearly every discussion, but so far there is no successful solution. In this chapter, we try to break through the theoretical deadlock by a practical proposal for a new basic type [C]. In order to stimulate further thoughts in this area, we relate our proposals to Scott's more theoretical studies of continuous lattices of types.

So we reopen the discussion of the previous chapter with a view to extending it. In this attempt, we expect to obtain a solution which is more informative than that suggested in System-F (using [A]). The discussion is divided into two parts. The first deals with declaration of circular functions and the second with that of undeclared ones.

In section (2.6), we shall briefly mention type checking of recursive functions. The treatment of this is parallel to that of undeclared circular functions. Owing to lack of time, only the treatment of recursive functions was implemented on the computer. However, we believe that this can be extended readily into circular types.

2.1 Circular types

We introduce into our system a new basic type denoted by [C]. The new type is used to construct type expressions for circular functions. Define these as functions applied to themselves. Let f be such a circular function, and let f'≡(f f). We show below how to find the type of f'. Suppose it is [t]. Then the first approximation to the type of f is [C→t], the second approximation is [[C→t]→t], the third is [[[C→t]→t]→t] and so on. For simplicity we shall represent [C→t], [[C→t]→t], ... by C_t^1, C_t^2, ..., and we also use C_t to stand for the whole family of types.

Example

> In the example (λg:(g g)4)(λf:λ[I]n:IF n=1 THEN 1 ELSE n*((f f)(n-1)) FI), if the process of f applying to itself will terminate at all, we would expect the result to be an integer. Hence ignoring the loopings in (λ[I]n:...), the type of this λ-expression must be [I→I] so that this is the value of "t" mentioned above.

2.2 Type Checking of Declared Circular Types

In order to check the type compatibility of (f g) in which f and g are circular functions (g may be the same as f), then ideally we would like to discover the "best approximation" of both types. If it is difficult to obtain the best approximation, we are prepared to accept a relaxation-- that is, the type of f and g must be of "comparable approximation" instead. That means if the type of g is at the j' th approximation, we require that the type of f must be at the (j+1)' th approximation before type checking can be conducted on them. This is necessary in order to preserve the stratification law which says that in expressions (f x) the type of f must be one level higher than that of x.

We assume that there exist system routines which will be invoked by the type checker to examine levels of approximation and to replace some of them by other apposite approximations in the family, where necessary. Such replacements will be reflected in the reduction equations by having changes in those places where adjustments are made on the circular types.

In the following examples, it is assumed that f, g and n are of the type [C→[I→I]], [[C→[I→I]]→[I→I]] and [I] respectively.

Example(1)

$(\theta(\Phi((f\ f)n)))$

$=(\theta((V_f\ V_f)V_n))$

$=(\theta_2(\theta(V_f\ V_f))[I])$

$=(\theta_2(\theta_2[C\rightarrow[I\rightarrow I]][C\rightarrow[I\rightarrow I]])[I])$

{adjustment on the level of approximation, in this case, change the first instance of $C^1_{I\rightarrow I}$ to $C^2_{I\rightarrow I}$}

$=(\theta_2(\theta_2[[C\rightarrow[I\rightarrow I]]\rightarrow[I\rightarrow I]][C\rightarrow[I\rightarrow I]])[I])$

$=(\theta_2[I\rightarrow I][I])$

$=[I]$

Example(2)

$(\theta(\Phi((g\ g)n)))$

$=(\theta_2(\theta(V_g\ V_g))[I])$

$=(\theta_2(\theta_2[[C\rightarrow[I\rightarrow I]]\rightarrow[I\rightarrow I]][[C\rightarrow[I\rightarrow I]]\rightarrow[I\rightarrow I]])[I])$

{adjustment on the level of approximation}

$=(\theta_2(\theta_2[[[C\rightarrow[I\rightarrow I]]\rightarrow[I\rightarrow I]]\rightarrow[I\rightarrow I]][[C\rightarrow[I\rightarrow I]]\rightarrow[I\rightarrow I]])$
$[I])$

$=(\theta_2[I\rightarrow I][I])$

$=[I]$

2.3 Undeclared Circular Types

For undeclared circular functions the process will be more complicated. Perhaps the reader has realized already that the type checker would enter an infinite loop if the expression contained variables of circular types which are not declared. Before we propose a solution to this problem we must therefore strengthen our present system.

A subset $\bar{S}$ of type variables is formed and its members are distinguished from others by having a bar on top of their names, for example, $\bar{V}$. The value of $\bar{V}$ is determined solely by the context of the program (thus indicating a certain degree of type deduction). For example, in the type expression "t_+ [I] $\bar{V}$", $\bar{V}$ would be assigned value [I]. The assignment is written "$\bar{V}$:-[I]". We use the new notation ":-" because this assignment is different from ":=" in the sense that $\bar{V}$ will be checked for any value that it may possess from previous assignments (in the sense of ":-") and any old value has to be consistent with the new one otherwise it will be in error. For example, assume that prior to reduction of the type expression "((t_v[B]$\bar{V}$)... (t_+[I]$\bar{V}$))", $\bar{V}$ has no value and t_v is the type constant of the boolean operator "OR". Then on the first occurrence of $\bar{V}$, it will be assigned value [B] as said above. However on the second occurrence, $\bar{V}$ cannot be assigned value [I], otherwise it will be inconsistent with its previous value. Other properties of $\bar{V}$ that we shall use in later examples are:

(1) $(\theta_3\ t_+\ \bar{V}_i\ \bar{V}_j)=\bar{V}_i$ if $i=j$

(2) $(\theta_3\ t_+\ \bar{V}_i\ \bar{V}_j)=(\bar{V}_j:-\bar{V}_i;\bar{V}_i)$ if $i\neq j$

according to this rule, any subsequent occurrences of $\bar{V}_j$ will be replaced by $\bar{V}_i$.

(3) $(\theta_2\ \bar{V}\ [t])=E$ for any type t

this is a very severe restriction on the uses of $\bar{V}$, but it is tolerable for our special purpose as can be seen from our examples.

(4) $(\theta_4\ t_{cond}\ [B]\ \bar{V}_i\ [t])=(\bar{V}_i:-[t];\ [t])$

(5) $(\theta_4\ t_{cond}\ [B]\ \bar{V}_i\ \bar{V}_j)=(\bar{V}_j:-\bar{V}_i;\ \bar{V}_i)$ if $i\neq j$

(6) $(\theta_4\ t_{cond}\ [B]\ \bar{V}_i\ \bar{V}_i)=\bar{V}_i$

Suppose $(\Phi(\lambda f:...))=\gamma_f$ and $(\Phi\ f)=V_f$ and in the course of type reduction we have to assign (in the sense of :=) value γ_f to V_f, then γ_f will be tagged by writing $V_f:=\gamma_f^c$. The symbol "c" indicates the possibility of circularity, and is only a marker which will never affect the value of the object that it is attached to.

2.4 Solution to the problem of Undeclared Circular Types

Our proposed solution can be formalized in the following reduction rule:

$$(RY10)\quad (\theta_2[t^c][t^c]) = (\theta_2[C\to\bar{V}][C\to\bar{V}])$$
$$= (\theta_2[[C\to\bar{V}]\to\bar{V}][C\to\bar{V}]) \quad \{\text{adjustment on approximation}\}$$
$$= \bar{V}$$

where t is a type expression that has been tagged and $\bar{V}$ is an element of $\bar{S}$. This solution will suffice for our examples, though we might extend it for more general cases to:

$$(\theta_2[t_i^c][t_j^c]) = (\bar{V}_j :- \bar{V}_i;\ (\theta_2[C\to\bar{V}_i][C\to\bar{V}_j])) = \bar{V}_i$$

Occurrences of the "c" marker on both of the arguments of θ_2 indicate that the circular function is applied to itself, so it is required to replace t^c by $[C\to\bar{V}]$ by assuming that the type of the final result is $\bar{V}$ (i.e. it is the "t" we mentioned in §2.1)

We have already emphasized that the use of the subset $\bar{S}$ has to be very restrictive. Furthermore, the smaller the object designated by $\bar{V}$, the quicker our solution will converge. Therefore we impose the following rule:

(RY11) If t^c is the type of $(\lambda f{:}\lambda a1{:}\lambda a2{:}\ldots{:}\lambda an{:}\ldots)$, it is required that t^c should be replaced by

$$[C\rightarrow[V_{a1}\rightarrow[V_{a2}\rightarrow\ldots[V_{an}\rightarrow\bar{V}]\ldots]]]$$

and not simply by $[C\rightarrow\bar{V}]$ when Rule (RY10) is applied. If t_{ai} is the value of V_{ai}, then (1) is further reduced to

$$[C\rightarrow[t_{a1}\rightarrow[t_{a2}\rightarrow\ldots[t_{an}\rightarrow\bar{V}]\ldots]]]$$

Reduction Example (1) {using the example at the outset of this chapter}

$(\theta(\Phi(\lambda g{:}(g\ g)4)(\lambda f{:}\lambda n{:}$IF n=1 THEN 1 ELSE n*((f f)(n-1)) FI$)))$

$=(\theta(\gamma_g\ \gamma_f))$

$=(\theta_2\ \gamma_g\ \gamma_f)$

$=(V_g{:=}\gamma_f;\ (\theta((V_g\ V_g)[I])))$

$=(\theta_2(\theta(V_g\ V_g))[I])$

$=(\theta_2(\theta_2(\theta\ V_g)(\theta\ V_g))[I])$

$=(\theta_2(\theta_2\ \gamma_f\ \gamma_f)[I])$

$=(V_f{:=}\gamma_f^c;\ (\theta_2(\theta\ \gamma_n)[I]))$ {γ_f is tagged}

$=(V_n{:=}[I];\ (\theta(t_{cond}(t_{=}\ V_n\ [I])[I](t_{*}\ [I]\ ((V_f\ V_f)(t_{-}\ V_n\ [I])$ $)))))$

$=(\theta_4\ t_{cond}\ [B][I]\ (\theta_3\ t_{*}\ [I]\ (\theta_2(\theta(V_f\ V_f))[I])))$

$=(\theta_4\ t_{cond}\ [B]\ [I]\ (\theta_3\ t_{*}\ [I]\ (\theta_2(\theta_2\ \gamma_f^c\ \gamma_f^c)[I])))$

$=(\theta_4\ t_{cond}\ [B]\ [I]\ (\theta_3\ t_{*}\ [I]\ (\theta_2(\theta_2\ [C\rightarrow[I\rightarrow\bar{V}]][C\rightarrow[I\rightarrow\bar{V}]])[I])))$

$=(\theta_4\ t_{cond}\ [B]\ [I]\ (\theta_3\ t_{*}\ [I]\ (\theta_2\ [I\rightarrow\bar{V}]\ [I])))$

$=(\theta_4\ t_{cond}\ [B]\ [I]\ (\theta_3\ t_{*}\ [I]\ \bar{V}))$

$=(\bar{V}:-[I];\ (\theta_4\ t_{cond}\ [B]\ [I]\ [I]))$

$=[I]$

Reduction Example (2)

$(\theta(\phi(\lambda f:(f\ f)3)(\lambda x:x)))$

$=(\theta\ \gamma_f\ \gamma_x)$

$=(V_f:=\gamma_x;\ (\theta((V_f\ V_f)[I])))$

$=(\theta_2(\theta(V_f\ V_f))[I])$

$=(\theta_2(\theta_2\ \gamma_x\ \gamma_x)[I])$

$=(V_x:=\gamma_x^c;\ (\theta_2(\theta\ V_x)[I]))$

$=(\theta_2\ \gamma_x^c\ [I])$

{onle one argument of θ_2 has the special marker, so neither RY10 or RY11 is applicable here}

$=(V_x:=[I];\ (\theta\ V_x))$

$=[I]$

In this example, f was applied to itself but produced no circular effect. It is important that the system handle this case properly too.

2.5 Lattice Representation of Types

We have not attempted any formal proof on our proposal. However, we notice that there are certain similarities between our approximation concept of circular types and Scott's [Scott,1972] lattices of data types, but the latter is a more theoretical approach than ours. So we shall give an account of lattice representations of types here and re-state our approximation concept in terms of them. We believe that it is worth to compare these results.

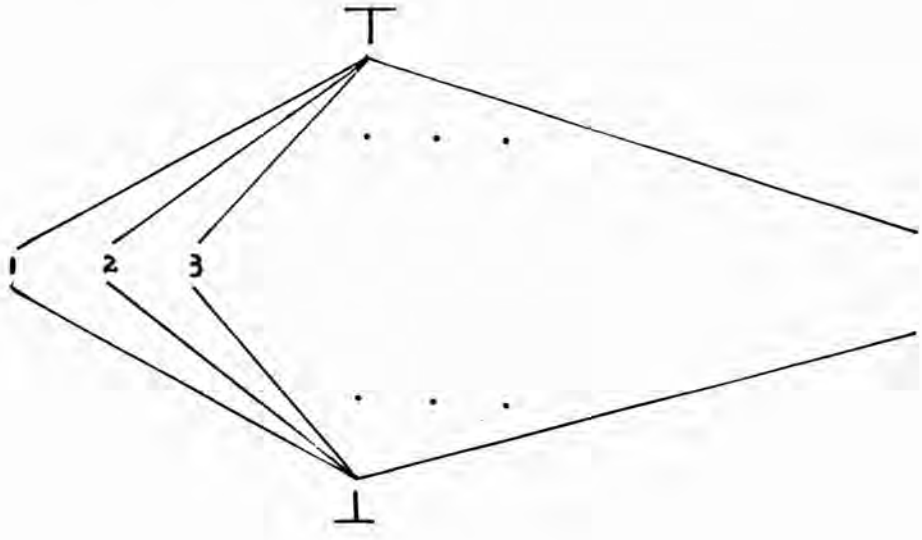
A lattice formed from a set S, with partial ordering $\geq$, consists of all elements of S and two special elements $\perp$ and $\top$ such that for all x in S, we have

$$\top \geq x \geq \perp$$

If $x \geq y$, we shall say x is a better approximation than y.

Example

The lattice of positive integers is:



The lattice for circular types is shown in diagram(2.1). Suppose t is $[I \to I]$, then each f_i^j can be interpreted as follows. For every i, the value of $(f_i^j\ n)$ will be $\perp$ if $n>j$, otherwise $(f_i^j\ n)=m$ for some integers n and m. Therefore for each i, f_i^{j+1} is a better approximation than f_i^j (i.e. $f_i^{j+1} \geq f_i^j$). Finally we let C_t^j denote the set $\{f_1^j, f_2^j, f_3^j, \ldots\}$.

We claim that C_t^{j+1} is a better approximation than C_t^j in the sense that each element of C_t^{j+1} is a better approximation than the corresponding element in C_t^j. In other words, for any j and t, C_t^{j+1} is a more exact description on the type of a circular function than C_t^j.

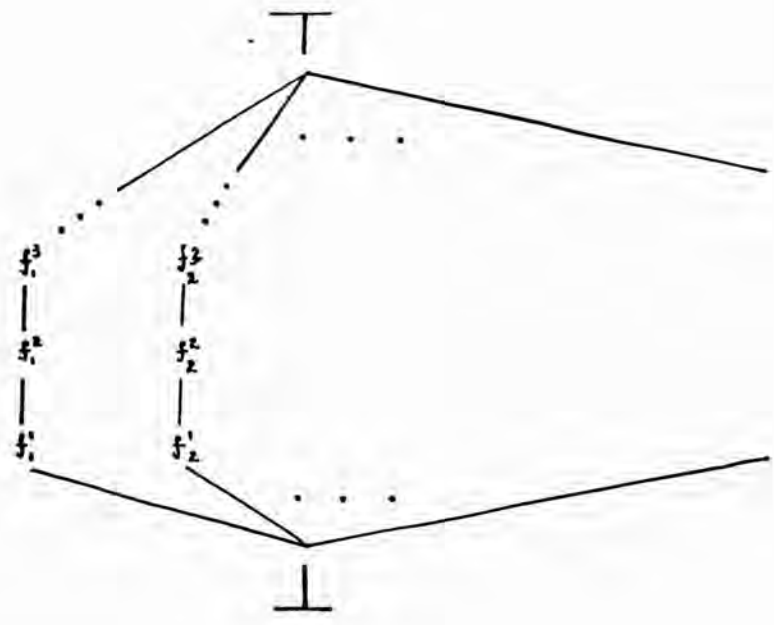


Diagram (2.1)

2.6 Recursive Functions

There is however no difficulty in declaring the type of recursive functions. For example, given (LABEL f:λn: IF n=0 THEN 1 ELSE n*f(n-1) FI), we can say the type of f is [I→I], or we can say [I→I] is the type of the minimal fix-point of this recursive function.

Where the types of the recursive functions are not declared, type deduction processes will be carried out with the aid of the special set $\bar{S}$ as in the case of circular functions.

In order for this to be done, the actions of the two functions, Φ and θ, should be extended as follows:

(1) $(\Phi(\text{LABEL f:}\lambda\text{n:...})) = (t_{rec}\ \gamma_f)$

where t_{rec} is a type constant.

(2) $(\theta(t_{rec}\ \gamma_f)) = (\theta_2''\ t_{rec}\ \gamma_f)$

where, if the type of the recursive function f has been declared, then define

(3) $(\theta_2''\ t_{rec}\ \gamma_f) = (\theta\ \gamma_n)$

otherwise, if it has not, define

(4) $(\theta_2''\ t_{rec}\ \gamma_f) = (V_f := [V_n \rightarrow \bar{V}];\ (\theta\ \gamma_n))$

Values will be assigned to V_n (by :=) and $\bar{V}$ (by :-) during the course of type reduction as before.

Example

$(\theta(\Phi((\text{LABEL f:}\lambda\text{n:IF n=0 THEN 1 ELSE n*f(n-1) FI})3)))$

$=(\theta((t_{rec}\ \gamma_f)[I]))$

$=(\theta_2(\theta(t_{rec}\ \gamma_f))[I])$

$=(\theta_2(\theta_2''\ t_{rec}\ \gamma_f)[I])$

$=(V_f:=V_n\rightarrow\bar{V};\ (\theta_2(\theta\ \gamma_n)[I]))$

$=(\theta_2\ \gamma_n\ [I])$

$=(V_n:=[I];\ (\theta(t_{cond}(t_=\ V_n\ [I])[I](t_*[I](V_f(t_-\ V_n\ [I])))))))$

$=(\theta_4\ t_{cond}\ [B]\ [I]\ (\theta_3\ t_*\ [I]\ (\theta(V_f\ [I]))))$

$=(\theta_4\ t_{cond}\ [B]\ [I]\ (\theta_3\ t_*\ [I]\ (\theta_2\ [I\rightarrow\bar{V}]\ [I])))$

$=(\theta_4\ t_{cond}\ [B]\ [I]\ (\theta_3\ t_*\ [I]\ \bar{V}))$

$=(\bar{V}:-[I];\ (\theta_4\ t_{cond}\ [B]\ [I]\ [I]))$

$=[I]$

2.6.1 Implementation

Our current implementation imposes the following restrictions on the definition of recursive functions:

(1) definition of recursive functions must be ended with a special marker "endlabel".

(2) parameters of recursive functions should be listed immediately after the name of the function without any "λ" or ":" in between.

Example

(LABEL f n:IF n=0 THEN ... FI endlabel)

which will be transformed by the system to

(LABEL f:λn:IF ... FI endlambda endlabel)

In this implementation, rule (3) in section (2.3) has been relaxed so that $(\theta_2\ \bar{V}\ [t])=\bar{V}$, for any type t. We are able to do this because type checking of recursive functions is performed in two steps,

(1) type checking proceeds according to the rules stated above until endlabel is reached.

(2) repeat the process with the type of the recursive function found in (1) and the process finishes when endlabel is reached again.

We have to be aware that(using the example in last section) "f" has to be added to the environment of (λn:...) before the type of this λ-expression is checked. In the example we noted that when [I] is assigned to V_n, the value of V_f is updated to $[I\rightarrow\bar{V}]$ too.

To conclude this section, we would like to mention that this version of System-Y implemented on CDC 64/6600 computers is just a minor extension of the one we proposed in last chapter.

Appendix D

In this appendix, we shall describe in Reckon (the Reckon notation uses the same abbreviations as in the previous appendix) an implemented version of System-Y. To bring out the main features of System-Y, it is only necessary to consider type checking of functional applications, and this is defined here by the routine "monadictypecheck". We list below some of the main routines defined in this appendix, and on the left are the corresponding theoretical discussions we have met in the foregoing chapters.

chapters 1/2, Part 3.	appendix D
RY4	monadictypecheck
RY4a	compatible
RY4b	C9A
θ_2' in RY4C	act9 and act10 in lambend
§1.6	updatememory
§1.8 and §1.10.3	fd1b
§2.6.1	labelend

The rest of the routines are mainly for improving the efficiency of type reductions. For example, in the combination (f x), assuming the type of f is $(t_{11}\rightarrow t_{12})\cap\ldots\cap(t_{n1}\rightarrow t_{n2})$ and the type of x is a type abstraction with empty memory, we would like to find whether there exists t_{i1} such that it would "fit" x and this process is defined in routine setup.

Readers may assume that the sets of typed λ-expressions in this implementation are defined recursively as

roots≡(functiondefinitions,bindingvariables,environments,
types)
WHERE functiondefinition≡λ-body
AND recursively types≡("basic",basictypes)∨("union",types,
types)∨("intersection",types,types)∨
(functional,domaintypes,rangetypes)∨
("typeabstraction",roots,typememories)
WHERE domaintypes,rangetypes,typememories≡
types,types,listoftypes

Machine representations of these sets are obviously variants of records in Pascal (or unions of structures in Algol-68) whose field selectors are
function, bv, environ, type over roots and
domain, range over functiontypes and
root, memory over type abstraction.
These selectors occur throughout the following routines.

Finally, the constants acti (where i is an integer) are used to represent activities awaiting completion, and for each specific value of i, acti denotes a specific routine in the following definitions.

```
LET setup(root,t,o,s,i,e,d)≡
  LET a,b,d1,e1≡IF isabstraction t THEN memory t ELSE t FI,bv root,
                function root,environ root;
    LET c,f≡IF isfunctional a THEN (domain a,act10)
            ELSE (domain.1st a,act9) FI;
      LET x,y≡newdump(o,f..a..root..s,i,e,d),newenvir(b,c,e1);
        (ϕ,ϕ,d1,y,x) {return to parser with new machine state}


AND C9A(o,s,i,e,d)≡
  LET p,argty≡root.2nd o,type.1st o;
    LET a,b,v≡function p,bv p,environ p;
      LET x,y≡newdump(o,act2..s',i,e,d),newenvir(b,argty,v);
        (ϕ,ϕ,a,y,x);


LET monadictypecheck(o,s,i,e,d)≡
  IF istypevariable.type.1st o THEN (consmonadcode o,s',i,e,d) ELSE
    CASE type.2nd o
    IN functional→
      LET functy,argty≡type.2nd o,type.1st o;
        IF undeclaredabstraction argty
        THEN
          LET p≡root.1st o;
            IF compatible(functy,type p)
            THEN updatememoryϕ; (consmonadcode o,s',i,e,d)
            ELSE
              LET q≡domain functy;
                setup(p,q,o,act4..functy..s',i,e,d)
            FI
        ELIF compatible(functy,argty)
        THEN (consmonadcode o,s',i,e,d) ELSE ERRORϕ
        FI
    OR undeclaredabstraction→
      LET functy,argty≡type.root.2nd o,type.1st o;
        IF emptymemory functy THEN C9A(o,s,i,e,d)
```

```
      ELIF undeclaredabstraction argty THEN
        LET arg≡root.lst o;
          IF compatible(functy,argty) THEN
            updatememoryϕ; (consmonadcode o,s',i,e,d) ELSE
            LET m≡memory functy;
              setup(arg,domain.lst m,o,act3..m..s',i,e,d)
          FI
      ELSE
        IF compatible(functy,argty) THEN
          updatememoryϕ; (consmonadcode o,s',i,e,d)
        ELSE C9A(o,s,i,e,d)
        FI
      FI
    OR declaredabstraction→
      LET argty,functy≡type.lst o,type.2nd o;
        IF undeclaredabstraction argty THEN
          LET arg≡root.lst o;
            IF compatible(functy,argty)THEN
              updatememoryϕ; (consmonadcode o,s',i,e,d)
            ELSE
              LET m≡memory functy;
                setup(arg,domain.lst m,o,act5..m..s',i,e,d)
            FI
        ELIF compatible(functy,argty) THEN (consmonadcode o,s',i,e,d)
        ELSE ERRORϕ
        FI
    OUT ERRORϕ
    ESAC
  FI;

{end of definition: monadictypecheck}
```

```
LET lambend(r,dump)≡
 (LET o,s,i,e,d≡1st dump;
    CASE 1st s
    IN act2 →
      IF iserror r THEN ERRORϕ  ELSE updatememoryϕ ;
        (consmonadcode o,s',i,e,d)
      FI
    OR act6→
      IF iserror r THEN ERRORϕ  ELSE updatememoryϕ ;
        IF NULL(2nd s)' THEN (o,s'',i,e,d) ELSE
          LET w≡(2nd s)';
            fd1b(1st w,o,act6..w..s'',i,e,d)
        FI
      FI
    OR act7→
      IF iserror r THEN ERRORϕ ELSE updatememoryϕ; (o,s'',i,e,d) FI
    OR act9→
      IF iserror r THEN L9A(o,s,i,e,d) ELSE
        LET w≡range.type r;
          IF undeclaredabstractionandemptymemory w THEN
            LET c≡1st.2nd s;
              LET p,q≡root w,range c;
                updatememoryϕ; setup(p,q,o,s,i,e,d)
          ELSE
            LET p,q≡type r,1st.2nd s;
              IF p⊆q THEN
                LET v,w≡3rd s,(2nd s)';
                  updatememoryϕ;
                  IF NULL w THEN L9C1A(o,s,i,e,d)
                  ELSE setup(v,w,o,s''',i,e,d)
                  FI
              ELSE L9A(o,s,i,e,d)
              FI
          FI
      FI
    OR act10→
      IF iserror r THEN L9A(o,s,i,e,d) ELSE
        LET w≡type r;
          IF undeclaredabstractionandemptymemory.range w THEN
```

```
          updatememoryϕ; setup(root.range w,range.2nd s,o,s''',i,e,d)
        ELSE
          IF 2nd s≥w THEN L9C1A ELSE L9A FI (o,s,i,e,d)
        FI
      FI
    ESAC
WHERE  REC L9A(o,s,i,e,d)≡
 (LET w≡s''';
    CASE 1st w
    IN(act9,act10)→L9A(o,w,i,e,d)
    OR(act3,act5)→
      LET u≡(2nd w)';
        IF NULL u THEN
          IF 1st w=act3 THEN C9A(o,w',i,e,d) ELSE ERRORϕ FI
        ELSE setup(root.1st o,domain.1st u,1st w..u..w'',i,e,d)
        FI
    OUT ERRORϕ
    ESAC)

AND REC L9C1A(o,s,i,e,d)≡
  (LET w≡s''';
    IF 1st w=act9 THEN
      LET v≡(2nd w)'
        IF NULL v THEN L9C1A(o,w,i,e,d)
        ELSE setup(3rd w,v,o,w''',i,e,d)
        FI
    ELSE updatememoryϕ; (consmonadcode o,w'',i,e,d)
    FI)
```

```
AND   formroot(o,s,i,e,d)≡
  (LET a≡formrootrecord.lst s;
    IF isdeclared.lst s THEN
      LET t≡type.bv.lst s;
        IF isunion t THEN fdlb(lst.unionelements t,a..o,act6..t..s',
                               i,e,d)
        ELSE fdlb(t,a..o,act7..t..s',i,e,d)
        FI
    ELSE (a..o,s',i,e,d)
    FI)

WHERE fdlb(t,o,s,i,e,d)≡
  (LET a,b,v≡function.lst o,bv. lst o,environ.lst o;
    LET x,y≡newdump(o,s,i,e,d),newenvir(b,t,v);
      (ϕ,ϕ,a,y,x)
  )


AND labelend(o,s,i,e,d)≡
  CASE lst.2nd.lst d  {the first item on the stack of last dump}
  IN(act2,act6,act7)→
    LET p≡function.root IF lst.2nd.lst d=act2 THEN 2nd.lst.lst d
                        ELSE lst.lst.lst d FI;
      LET w≡newdump(o,act13..s,i,e,d);
        (ϕ,ϕ,p,e,w)
  OUT
    LET r≡lst o;
      LET o1,s1,i1,e1,d1≡lst d;
        (r..o1,s1',i1',e1,d1)
  ESAC ;
```

```
LET output,stack,input,envir,dump≡ϕ,ϕ,readinputϕ,ϕ,ϕ;
  WHILE NOT(endϕ) DO
    LET lp,rp≡leftprecedence.1st stack,rightprecedence.1st input;
      IF rp>lp THEN
        LET w≡1st input;
        CASE w
        IN(numbers,variables,λ-exps,recexps)→
          stack:=w..stack; input:=input'
        OR keywords→
          IF closebracket w THEN stack:=stack'; input:=input'
          ELIF w=endlambda THEN
            output,stack,input,envir,dump:=lambend(1st output,dump)
          ELIF w=endlabel THEN
            output,stack,input,envir,dump:=labelend(output,stack,
              input,envir,dump)
          ELSE stack:=w..stack; input:=input'
          FI
        ESAC
      ELSE
        LET w≡1st stack;
        CASE w
        IN numbers→
          output:=w..output; stack:=stack'
        OR λ-exps→
          output,stack,input,envir,dump:=formroot(output,stack,
            input,envir,dump)
        OR variables→
          output:=(assigntypeto w)..output; stack:=stack'
        OR keywords→
          LET op≡lookup(w,parseenvironment);
            output,stack,input,envir,dump:=op(output,stack,input,
                                              envir,dump)
        OR recexps →
          stack:=recname.1st stack..recdefn.1st stack..rectag..stack'
        OR rectag→output:=consrec output
        ESAC
      FI OD;
transform({the parsed and typechecked}1st output){as in System-F}
```

Appendix E

Programming Examples of System-Y

Again, three examples are included here as we did for System-F in Appendix C. Since type checking of System-Y is not as straightforward as System-F, so the function "printtype" will be more important for the examples here because "printtype" will not only print the type information required, but from the order this information appears in the output, one can also obtain the order in which the types of the expressions are checked.

As an abbreviation, we write "$(b?e_1!e_2)$" for "IF b THEN e_1 ELSE e_2 FI".

Readers are recommended to compare the two results (with and without type declaration) of each example.

To simplify the notation, all "endlambda"s are discarded in the following listings.

EXAMPLE 1

```
START

COMMENT
"SQUARE" IS DEFINED IN TERMS OF "W" AND "MULTIPLY".
IT IS THEN APPLIED TO INTEGER AND REAL RESPECTIVELY.
THERE IS NO TYPE DECLARATION IN THIS EXAMPLE.
COMMENTEND

(SMULTIPLY:SW:(SSQUARE:(PRINTTYPE SQUARE 2);
  (PRINTTYPE SQUARE 2.0))
  (PRINTTYPE W MULTIPLY))
(SN:SM:(PRINTTYPE N*M))
(SF:SX:(PRINTTYPE(PRINTTYPE F X) X))
FINISH

TYPE OF (W . MULTIPLY) IS : [A-TYPE-ABSTRACTION-WITH-
EMPTY-MEMORY]

TYPE OF (F . X) IS : [A-TYPE-ABSTRACTION-WITH-
EMPTY-MEMORY]

TYPE OF (N * M) IS : [INTEGER]

TYPE OF ((F . X) . X) IS : [INTEGER]

TYPE OF (SQUARE . 2) IS : [INTEGER]

TYPE OF (F . X) IS : [A-TYPE-ABSTRACTION-WITH-
EMPTY-MEMORY]

TYPE OF (N * M) IS : [REAL]

TYPE OF ((F . X) . X) IS : [REAL]

TYPE OF (SQUARE . 2.0) IS : [REAL]

RESULT OF TYPE CHECKING IS : [REAL]
QED.
```

EXAMPLE 2

```
START

COMMENT
THIS EXAMPLE IS SIMILAR TO LAST ONE, BUT WITH TYPE
DECLARATIONS.
COMMENTEND

($[[[INTEGER]←[[INTEGER]←[INTEGER]]]&
  [[REAL]←[[REAL]←[REAL]]]]MULTIPLY:$W:
  ($SQUARE:(PRINTTYPE SQUARE 2);
    (PRINTTYPE SQUARE 2.0))
  (PRINTTYPE W MULTIPLY))
($N:$M:(PRINTTYPE N*M))
($F:$X:(PRINTTYPE(PRINTTYPE F X) X))
FINISH

TYPE OF (N * M) IS : [INTEGER]

TYPE OF (N * M) IS : [REAL]

TYPE OF (W . MULTIPLY) IS : [A-TYPE-ABSTRACTION-WITH
EMPTY-MEMORY]

TYPE OF (F . X) IS : [[INTEGER]←[INTEGER]]

TYPE OF ((F . X) . X) IS : [INTEGER]

TYPE OF (SQUARE . 2) IS : [INTEGER]

TYPE OF (F . X) IS : [[REAL]←[REAL]]

TYPE OF ((F . X) . X) IS : [REAL]

TYPE OF (SQUARE . 2.0) IS : [REAL]

RESULT OF TYPE CHECKING IS : [REAL]
QED.
```

EXAMPLE 3

```
START

COMMENT
THE TAGS ENABLE EVAL TO SELECT ROUTINES FROM THE PRELUDE.
THE SELECTED ROUTINE WILL THEN BE APPLIED TO THE FIRST
ARGUMENT OF EVAL.    THERE IS NO TYPE DECLARATION IN THIS
EXAMPLE.
COMMENTEND

($PRELUDE:$TAG1:$TAG2:
  ($EVAL:(EVAL 3 TAG1);(EVAL 3.0 TAG2))
  ($X:$SELECTOR:(PRINTTYPE(
    PRINTTYPE PRELUDE SELECTOR)X)))
($F:F($N:N+N)($M:M+M))($A:$B:A)($C:$D:D)
FINISH

TYPE OF (PRELUDE . SELECTOR) IS : [A-TYPE-ABSTRACTION-
WITH-EMPTY-MEMORY]

TYPE OF ((PRELUDE . SELECTOR) . X) IS : [INTEGER]

TYPE OF (PRELUDE . SELECTOR) IS : [A-TYPE-ABSTRACTION-
WITH-EMPTY-MEMORY]

TYPE OF ((PRELUDE . SELECTOR) . X) IS : [REAL]

RESULT OF TYPE CHECKING IS : [REAL]
QED.
```

EXAMPLE 4

```
START

COMMENT
THIS EXAMPLE IS SIMILAR TO LAST ONE, BUT WITH TYPE
DECLARATIONS.
COMMENTEND

($PRELUDE: $TAG1: $TAG2:
  ($EVAL: (EVAL 3 TAG1); (EVAL 3.0 TAG2))
  ($X: $SELECTOR: (PRINTTYPE(
    PRINTTYPE PRELUDE SELECTOR)X)))
($F: F($[INTEGER]N: N+N)($[REAL]M: M+M))
($A: $B: A)($C: $D: D)
FINISH

TYPE OF (PRELUDE . SELECTOR) IS : [[INTEGER]←[INTEGER]]

TYPE OF ((PRELUDE . SELECTOR) . X) IS : [INTEGER]

TYPE OF (PRELUDE . SELECTOR) IS : [[REAL]←[REAL]]

TYPE OF ((PRELUDE . SELECTOR) . X) IS : [REAL]

RESULT OF TYPE CHECKING IS : [REAL]
QED.
```

EXAMPLE 5


```
START

COMMENT
"GENERATE IS BOUND TO THE RECURSIVE FUNCTION "F".
"F" IS VERY SIMILAR TO OUR EXAMPLE IN 1.4.3, PART 2.
READERS ARE REMINDED THAT TYPE CHECKING OF RECURSIVE FUNCTIONS
HAS TO BE PERFORMED IN TWO PHASES.
NO TYPE DECLARATION IS PROVIDED IN THIS EXAMPLE
COMMENTEND

(SGENERATE:(SFINT:SFREAL:
            (PRINTTYPE (PRINTTYPE FINT)1 + (PRINTTYPE FINT)2);
            (PRINTTYPE (PRINTTYPE FREAL)1.0+(PRINTTYPE FREAL)2.0);
            (PRINTTYPE(PRINTTYPE FINT)3+(PRINTTYPE FINT)4);
            (PRINTTYPE(PRINTTYPE FREAL)3.0+(PRINTTYPE FREAL)4.0)
             )
    (GENERATE
     (SB:B=0 ) (S():1 )
     (SU:U-1 )
     (SV:SW:V*W  ))
    (GENERATE
     (SA:A=0.0 ) (S():1.0 )
     (SM:M-1.0 )
     (SX:SY:X*Y  )) )
(SPREDICATE:SEXIT:SMODIFY:SCOMBINE:
     ( LABEL F N :
              (PRINTTYPE(PREDICATE N)?EXIT()!
               ( PRINTTYPE COMBINE N
                           ((PRINTTYPE F)(MODIFY N)))
              )ENDLABEL)
   )

FINISH


  TYPE OF FINT IS : [A-TYPE-ABSTRACTION-WITH-EMPTY-MEMORY]

  TYPE OF F IS : [[INTEGER]→ A-TYPE-VARIABLE]

  TYPE OF ((COMBINE . N) . (F . (MODIFY . N))) IS :  A-TYPE-VARIABLE

  TYPE OF ((PREDICATE . N) ? (EXIT . ()) ! ((COMBINE . N) . (F . (MODI
FY . N)))) IS : [INTEGER]

  TYPE OF F IS : [[INTEGER]→[INTEGER]]

  TYPE OF ((COMBINE . N) . (F . (MODIFY . N))) IS : [INTEGER]

  TYPE OF ((PREDICATE . N) ? (EXIT . ()) ! ((COMBINE . N) . (F . (MODI
FY . N)))) IS : [INTEGER]

  TYPE OF FINT IS : [[[INTEGER]→[INTEGER]]]

  TYPE OF ((FINT . 1) + (FINT . 2)) IS : [INTEGER]

  TYPE OF FREAL IS : [A-TYPE-ABSTRACTION-WITH-EMPTY-MEMORY]

  TYPE OF F IS : [[REAL]→ A-TYPE-VARIABLE]

  TYPE OF ((COMBINE . N) . (F . (MODIFY . N))) IS :  A-TYPE-VARIABLE

  TYPE OF ((PREDICATE . N) ? (EXIT . ()) ! ((COMBINE . N) . (F . (MODI
FY . N)))) IS : [REAL]

  TYPE OF F IS : [[REAL]→[REAL]]

  TYPE OF ((COMBINE . N) . (F . (MODIFY . N))) IS : [REAL]

  TYPE OF ((PREDICATE . N) ? (EXIT . ()) ! ((COMBINE . N) . (F . (MODI
FY . N)))) IS : [REAL]

  TYPE OF FREAL IS : [[[REAL]→[REAL]]]

  TYPE OF ((FREAL . 1.0) + (FREAL . 2.0)) IS : [REAL]

  TYPE OF FINT IS : [[[INTEGER]→[INTEGER]]]

  TYPE OF FINT IS : [[[INTEGER]→[INTEGER]]]

  TYPE OF ((FINT . 3) + (FINT . 4)) IS : [INTEGER]

  TYPE OF FREAL IS : [[[REAL]→[REAL]]]

  TYPE OF FREAL IS : [[[REAL]→[REAL]]]

  TYPE OF ((FREAL . 3.0) + (FREAL . 4.0)) IS : [REAL]


 RESULT OF TYPE CHECKING IS : [REAL]


  QED
```

EXAMPLE 6

```
START

COMMENT
THIS IS SIMILAR TO LAST EXAMPLE, BUT WITH TYPE DECLARATIONS.
COMMENTEND

($GENERATE:($FINT:$FREAL:
              (PRINTTYPE (PRINTTYPE FINT)1 + (PRINTTYPE FINT)2);
              (PRINTTYPE (PRINTTYPE FREAL)1.0+(PRINTTYPE FREAL)2.0);
              (PRINTTYPE(PRINTTYPE FINT)3+(PRINTTYPE FINT)4);
              (PRINTTYPE(PRINTTYPE FREAL)3.0+(PRINTTYPE FREAL)4.0)
               )
      (GENERATE
       ($[INTEGER]B:B=0 ) ($():I )
       ($[INTEGER]U:U=I )
       ($[INTEGER]V:$[INTEGER]W:V*W  ))
      (GENERATE
       ($[REAL]A:A=0.0 ) ($():1.0 )
       ($[REAL]M:M=1.0 )
       ($[REAL]X:$[REAL]Y:X*Y  )) )
($PREDICATE:$EXIT:$MODIFY:$COMBINE:
       ( LABEL F N :
                (PRINTTYPE (PREDICATE N)?EXIT();
                  ( PRINTTYPE COMBINE N
                                ((PRINTTYPE F)(MODIFY N)))
                )ENDLABEL)
   )

FINISH
```

TYPE OF FINT IS : [A-TYPE-ABSTRACTION-WITH-EMPTY-MEMORY]

TYPE OF F IS : [[INTEGER]→ A-TYPE-VARIABLE]

TYPE OF ((COMBINE . N) . (F . (MODIFY . N))) IS : A-TYPE-VARIABLE

TYPE OF ((PREDICATE . N) ? (EXIT . ()) ! ((COMBINE . N) . (F . (MODIFY . N)))) IS : [INTEGER]

TYPE OF F IS : [[INTEGER]→[INTEGER]]

TYPE OF ((COMBINE . N) . (F . (MODIFY . N))) IS : [INTEGER]

TYPE OF ((PREDICATE . N) ? (EXIT . ()) ! ((COMBINE . N) . (F . (MODIFY . N)))) IS : [INTEGER]

TYPE OF FINT IS : [[[INTEGER]→[INTEGER]]]

TYPE OF ((FINT . 1) + (FINT . 2)) IS : [INTEGER]

TYPE OF FREAL IS : [A-TYPE-ABSTRACTION-WITH-EMPTY-MEMORY]

TYPE OF F IS : [[REAL]→ A-TYPE-VARIABLE]

TYPE OF ((COMBINE . N) . (F . (MODIFY . N))) IS : A-TYPE-VARIABLE

TYPE OF ((PREDICATE . N) ? (EXIT . ()) ! ((COMBINE . N) . (F . (MODIFY . N)))) IS : [REAL]

TYPE OF F IS : [[REAL]→[REAL]]

TYPE OF ((COMBINE . N) . (F . (MODIFY . N))) IS : [REAL]

TYPE OF ((PREDICATE . N) ? (EXIT . ()) ! ((COMBINE . N) . (F . (MODIFY . N)))) IS : [REAL]

TYPE OF FREAL IS : [[[REAL]→[REAL]]]

TYPE OF ((FREAL . 1.0) + (FREAL . 2.0)) IS : [REAL]

TYPE OF FINT IS : [[[INTEGER]→[INTEGER]]]

TYPE OF FINT IS : [[[INTEGER]→[INTEGER]]]

TYPE OF ((FINT . 3) + (FINT . 4)) IS : [INTEGER]

TYPE OF FREAL IS : [[[REAL]→[REAL]]]

TYPE OF FREAL IS : [[[REAL]→[REAL]]]

TYPE OF ((FREAL . 3.0) + (FREAL . 4.0)) IS : [REAL]

RESULT OF TYPE CHECKING IS : [REAL]

QED

CHAPTER THREE
FUTURE DEVELOPMENTS WITH RESPECTS TO USER PARTICIPATION AND DATA STRUCTURE TYPES

So far we have ignored programmers in our discussions of type-checking systems. This does not mean that their participation is not important. In fact, we think they may participate in three ways:

(1) they may declare the type of variables

(2) they may define new data types as in Algol-68 and Pascal

(3) they may present type expressions as arguments of certain functions.

Although user participation is optional in our proposal, we regard it as by no means insignificant. On the contrary, we think more work is needed. Discussion in this chapter represents our preliminary work in this area in the hope of generating more definite treatment of the issues raised. We have reasons for believing that work along the lines we propose might be more rewarding than striving for an ideal ω-order system.

3.1 Types as objects in a computing model

So far uses of types have been restricted to their declaration, checking and deduction. We propose now to allow types as arguments of certain functions. In order to do this, we have to construct a new set of objects whose elements are the elements of the type-checking system. In other words, the new set consists of types. Call this set [TYPE] and add it to the type-checking system as one of its types. We shall be careful to construct [TYPE] so that it is not included in itself. Meanwhile the set [TYPE] is available for use in programs much as other sets such as [INTEGER]. In particular we can declare an object to be of type [TYPE].

3.2 Another approach to parametric polymorphism

Consider anew the example,

(λtwice:twice [I] f' x')(λ[TYPE]t:λ[t→t]f:λ[t]x:f(f x))

with the type of f' and x', [I→I] and [I] respectively.

(1) t is declared to be of type [TYPE]. We can describe the type of f and x in terms of t. This suggests an alternative solution to the parametric polymorphism problem.
(Reynolds[Reynolds,1974] has also used variables, say t, to describe the type of other variables. In order to distinguish "t" from others, he introduced a new symbol

"∧" so that instead of writing (λ[TYPE]t:λ[t]x:...), he wrote (∧t:λ[t]x:...).)

(2) After the compilation of (twice [I]), t will be assigned the value [I], thus the type of f and x are finalized to [I→I] and [I] respectively.

(3) With the finalized types of f and x, type compatibilities of f' and x' will be checked as normal.

By allowing types to occur in the program as shown above, it is no longer a disadvantage that the type of twice is not known when it is defined as this information will be available immediately before it is used. On the other hand, there is provided a new opportunity for programmers to intervene actively in the process of computation. For example, in code generation, suppose the function ADD is applicable to both integer and real numbers, then there will exist two routines for executing the instruction. But, by writing ADD [I] n_1 n_2 or ADD [R] n_1 n_2, the programmer can guide the system's choice.

As a minor modification, we propose that (f [t] x) is the same as (f x) if the type of x is [t], thus allowing us to omit the type argument in some cases. Of course f must be a function expecting a type argument. In other words, if the type argument is absent for f, this information will be obtained from the following argument, but care must be

taken before we drop the type argument otherwise the operation may be meaningless (or even erroneous)-- as in the case of universal type predicates,for example.

For our second illustration, let us assume that it is possible to test the type of any object in a program. For each possible type t_i, suppose there exists a predicate is-t_i. Now, types can be constructed from other existing types, so closure of the type constructors is infinitely large and there is an infinite number of type predicates. The primitive predicates ought to correspond to the basic types while the others have to be defined. Furthermore, by allowing types as arguments of functions, one type predicate will be enough for our purposes. Let us call it ISTYPE so that (ISTYPE [t_i] x) is true if the type of x is [t_i] otherwise false. Since ISTYPE is defined by the system, it can share the routines that are available to the type checker in testing any arbitrary types.

3.3 Declaration of ordered types

Once a new ordered type has been defined, it should be treated by the type-checking system in the same way as existing types. In order to avoid repetitions, we omit union and intersection types here and concentrate on our ordered types (or cartesian products of types, as they are generally called in type and set theory or records or structures as they are called in programming languages).

A cartesian product of types is defined by applying the cartesian operator "&" to two existing types so that, for example, 3-dimensional integer vectors can be defined as:

```
DEFINE [IVEC]≡[I]&[I]&[I]
```

in this case, the cartesian operator will construct an object of 3 dimensions (i.e. of 3 components) from the domain [I] for any variables of type [IVEC].

There are three important classes of operations on ordered types:

(1) predicates: ISTYPE

As mentioned in last section, the universal type predicate is defined for any types, so it is applicable to ordered types too.

(2) constructors

By analogy with ISTYPE, we need only one universal constructor which we shall call "MAKE" and (MAKE [IVEC] i_1 i_2 i_3) is an object of [IVEC]. The function MAKE should check not only that the correct number of arguments are given, but also that these arguments are of correct types. MAKE obtains its type information from the first argument which must be of type [TYPE].

(3) Selectors

Ordered-type declaration should construct the following functions for selecting the 3 components of [IVEC] vectors.

```
LET [IVEC→I] xcomponent ≡(λ[IVEC]v: 1st v)
AND [IVEC→I] ycomponent ≡(λ[IVEC]v: 2nd v)
AND [IVEC→I] zcomponent ≡(λ[IVEC]v: 3rd v)
```

"nth" is the function defined by the system for selection. If v_1 is the object of type [IVEC], (ycomponent v_1) will be the second constituent of v_1. Alternatively, the selecting functions can be made implicit by including them in the definitions of ordered types.

```
DEFINE [IVEC] ≡[I]|xcomponent & [I]|ycomponent &
               [I]|zcomponent
```

where "|" is used to separate the selector from the type of the corresponding component.

Since there is no restriction on the name of the selecting functions, so the 3 dimensional real-vectors can be defined as

```
DEFINE [RVEC]≡[R]|zcomponent & [R]|xcomponent & [R]|ycomponent
```

This is to say (ycomponent v), for example, will be of of type [I] or [R] depending on whether v is of type [IVEC] or [RVEC], so that xcomponent, ycomponent and zcomponent are polymorphic functions.

By analogy, again, the universal selecting function can be defined in BNF as

<universal-selecting-function>::=<SELECT><cartesian-type>
<field-identifier><cartesian-ob>

where xcomponent, etc. are field identifiers. Intending programmers may find it helpful to compare the following two possible results.

SELECT [IVEC]xcomponent=(λ[IVEC]v:1st v)
SELECT [RVEC] xcomponent=(λ[RVEC]v:2nd v)

3.4 Contextual checking of types

Suppose the data type for personal record is defined as:

DEFINE [person]≡[S]|name & [S]|fathername & [I]|age & [I]|children

Let us consider the following two statements for constructing a new "person":

(1) (MAKE [person] <A.Smith> <B.Jones> 40 2)

(2) (MAKE [person] <A.Smith> <B.Smith> 2 2)

Both statements are permissible if the system judges type correctness only from syntactic combination. But, we know both statements are absured because in the first statement the name of the person is different from his father's and in the second statement the person is too young to have child.

The requisite checking may be called "context-sensitive type checking". Context-sensitive type checking takes note that components of ordered-types might be related to each other.

This example reveals the deficiency of ordinary type-checking systems in handling ordered (structured) types. As the ordered-type becomes more complicated, more contextuality is required. Without it, this responsibility falls on the shoulders of users. Various schemes have been discussed in [DAHL,1972], [MORRIS,1973] and [REYNOLDS,1975] who seem to agree that it would be safer if access to structured types were limited to certain functions only, so that in addition to the ordinary type checking on functions and their arguments, these functions could carry out some extra type checking processes on the arguments as well. Let us call these extra processes "screening", and we shall illustrate this point in the following example.

Example

Suppose a school-file is a list or array or structure of student-records, each containing information on students' names, ages, etc. Student-records might be arranged in alphabetical order or according to age or whatever criterion is suitable. As there would be no direct access to the data structures from outside functions, so the exact representations of the records should have no relevance in formulating algorithms in sovling problems. Permit the following operations on student-records,

(1) INSERT<name> BEGIN age:=...; year:=...; ... END

(2) DELETE <name>

(3) UPDATE <name> BEGIN age:=...; ... END

the kinds of screening that are necessary are suggested below:

(1) The instruction INSERT is a request to construct a new student-record. Before the new record is placed in the right slot, in order to avoid duplication, the screening routine ought to check that it is needed a new record and check information on age, year, etc.

(2) Before a student-record could be deleted, it is reasonable to ask the user to provide reason for the deletion via the

second parameter of this instruction (which could be some predefined strings or constants). Suppose the instruction is "DELETE <name> <graduate>", then the year of entry of that student should be checked to ensure that the deletion is legal.

(3) Most of the discussions above are also applicable to the instruction UPDATE.

3.5 Summary

Current type-checking systems fail to provide the kind of context-sensitive screening of ordered types that is required. Participation from programmers is particularly desired in this area. There are 3 ways that programmers can advise type-checking systems. We have studied two of them in detail, while the third (type declaration) has been elaborated in earlier parts of this thesis.

We would like to see types play a more active role in programming than the purely negative one of type checking. We have illustrated this idea by a few examples. We believe we have done enough to indicate what we have in mind for the future-- that algorithmic type theory will increasingly come to be seen as the branch of logic needed for design of memory protection systems.

References

BURGE,W.H., Combinatory Programming and Combinatorial Analysis, IBM Journal R&D, Vol.16, 1972.

BURSTALL, R.M., LANDIN, P.J., Programs and their proofs-an algebraic approach, Machine Intelligence, Vol.4, pp17-43, 1969.

CHURCH, A., The Calculi of Lambda Conversion, Annals of Mathematics Studies, No.6, Princeton, N.J., Princeton University Press, 1941.

CURRY, H.B., FEYS, R., Combinatory Logic, North Holland, Vol.1, 1958.

DAHL, O.J., HOARE, C.A.R., Hierarchical Program Structure, in Structure Programming, Academic Press, 1972.

EDWARDS, R.P., Reckon Manual, Royal Holloway College, 1974a.

EDWARDS, R.P., Implicit Operators, private communication, 1974b.

EDWARDS, R.P., A Functional View of Data Structures, book in preparation, 1975.

HEXT, J.B., Compile-time Type Checking, Computer Journal, Vol.9, pp 365-369, 1966.

KLEENE, S.C., λ-definability and recursiveness, Duke Mathematical Journal, Vol.2, pp 340-353, 1936.

LANDIN, P.J., A correspondence between Algol 60 and Church's Lambda Notation, CACM, Vol.8, pp 89-101 and 158-165, 1965.

LASKI, J.G., Sets and other types, Algol Bulletin, No.27 pp 41-48, 1968.

LEDGARD, H.F., A Model for Type Checking-with an application to Algol 60, CACM, Vol.15, pp 956-966, 1972.

MANNA, Z., PNUELI, A., Formalization of Properties of Functional Program, JACM, Vol17, 1970.

McCarthy,J., et al., LISP 1.5 Programmer's Manual, MIT Press, 1962.

Mc CARTHY, J., A Basis for a Mathematical Theory of Computation, Computer Programming and Formal Systems (ed. Braffort), 1963.

MINSKY, M.L., Computation:finite and infinite machine, Prentice-Hall, 1967.

MORRIS, J.H., Lambda-Calculus Models of Programming Languages, MAC-TR-57, Project MAC, MIT, 1968.

MORRIS, J.H., Types are not sets, Proc A.C.M Symposium on Principles of Programming Languages, Boston, pp 120-124, 1973.

REYNOLDS, J.C., GEDANKEN, CACM, Vol.13, pp 308-319, 1970.

REYNOLDS, J.C., Towards a Theory of Type Structures, Proceedings of Colloque sur la Programmation, Paris, 1974.

REYNOLDS, J.C., User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction, at the Conf. on New Directions in Algorithmic Languages sponsored by IFIP Working Group 2.1, Munich, 1975.

SCOTT, D., Lattice Theory, Data Types and Semantics, Formal Semantics of Programming Languages (ed. R.Rustin), 1972.

TENENBAUM, A.M., Type Determination For Very High Level Languages, Ph.D Thesis, New York University, 1974.

TURING, A.M., Computability and λ-definability, The Journal of Symbolic Logic, Vol. 2, pp 153-163, 1937.