ON DATA TYPES

by

Dr. Johannes J. Martin
Dept. of Computer Science
Virginia Tech
Blacksburg, Va 24061

June 1983

Technical Report #CS830004-R

O N D A T A T Y P E S

Johannes J. Martin

Virginia Tech

Blacksburg, Virginia 24060

June 1983

Abstract:

A semantic model of types is proposed. This model interprets types as elements in an augmented domain constructed with the Smyth powerdomain constructor. In this domain types approximate the values of which they are types. Within this model, a type of an application f(x) is found by applying a type of f to a type of x. This becomes the basis of type checking and type inference.

The model accomodates in a natural way type hierarchies, polymorphic functions, and recursive polymorphic types. A number of examples are worked out in some detail.

I. INTRODUCTION

This work pursues a simple semantic model for the treatment of types in strongly typed languages. One can view types as names for sets that are used as domains and codomains of functions (in a broad sense), and type checking as the process that ensures that functions are applied only to values of their own domains. This view seems to neglect the other function of type: being a vehicle of data abstraction.

But both views can be held simultaneously. Concerned mainly with data abstraction, we would not quarrel with the recommendation that our carrier sets ought to furnish the domains and codomains of all functions; similarly, recommending that the domains and codomains of all functions coincide with carrier sets of given types, we would not mind the requirement that there ought to be small sets of primitive operations on those sets with which all other functions are to be constructed. In this paper we shall deal with both concerns starting with the view that types name domains and codomains of functions.

The basic idea of our approach is that type checking and type inference take place in a domain T derived from the computational domain D by adding additional elements, namely the (potential) types. These types approximate the values of which they are types. Clearly, the claim "x is integer" gives less information than "x is odd" which in turn gives less than "x is 3". The basic notions of this theory are described in [Martin82]. Our method is akin to that of MacQueen [MacQueen82] in so far as elements of the

same type form an ideal. It differs by the fact that types are treated as ordinary objects in the (augmented) domain, and that they are subjected to the same rules of computation as all other objects. Consequently, we shall not need any new rules or concepts beyond the extension of the domain.

The domain T can also be viewed as a domain for a programming language with dynamic types. With static type checking the domain T is used for a rough calculation that is to ensure internal type consistency of all expression so that later calculations can be based on the simpler computational domain D and, hence, can be executed more efficiently.

II. THE AUGMENTED DOMAIN

For the purpose of our discussion we assume that the computational domain is given by the domain equation

$$D = A + (D \times D) + (D \to D)$$

where A is a simple domain of atomic objects such as the integers, boolean values, and so on. Viewed as an information system [Scott82]

$$D = (DD_D, \Delta_D, CON_D, \vdash_D),$$

the elements x of the domain |D| are consistent subsets of DD_D (unions of members of CON) that are downward closed by $\vdash$. The members of DD are interpreted as propositions, and a set of

propositions (a member of |D|) is the conjunction of its constituent propositions. The distinguished element Δ is the least informative proposition in DD.

The elements of the augmented domain are collections of sets of propositions interpreted as conjunctions of disjunctions of propositions. The conjunctive sets must again be consistent and downward closed, and the disjunctive sets can be viewed as actual data items of the augmented domain. A construction that gives us these types of elements is the Smyth powerdomain constructor [Smyth78]. Therefore, we replace in our original domain equation the base domain A by the powerdomain

$$P_{MS}(B) \quad \text{where} \quad B = A + B \times B,$$

and obtain

$$T = P_{MS}(B) + (T \times T) + (T \to T).$$

The subscript "MS" (modified Smyth) indicates that this constructor is modified in one important respect: instead of using the finite subsets of A as the elements of T, we use the recursively enumerable subsets of A. This ensures that the elements of DD_T are still finitely described and the members of |T| are defined by countable sets. As we shall see, this modification is needed because some of the elements gained are vital to the model.

Recall that the element $\{X_0,\ldots,X_{n-1}\}$ of a Smyth domain describes an object to which (at least) one of the propositions X_i applies.

For example, the element {true,false} asserts that the object described is either the value "true" or the value "false"; in other words, it asserts that the object is of type BOOL. In general, each recursively enumerable (RE) set S of total elements in B has a unique greatest lower bound t, that is to say, for each such S there is an element t in B with the property

for all x. $x \in S$ iff $t \subset x$.

Note: "$\subset$" means "included in" not "properly included in".

We view the element t as the type of the associated (carrier) set S. The function E from types to carrier sets is given by

for all x. $x \in E(t)$ iff $t \subset x$.

Because of the structure of B, our domain furnishes individual types for all RE sets composed of atoms and/or (nested) pairs of atoms. Furthermore,

with the types s and t for E(s) and E(t), respectively,

(s,t) is the type of $E(s) \times E(t)$.

The element $\{\{\Delta\}\}$ is the universal type.

Note that the domain

$$D = B + (D \times D) + (D \to D),$$

which we assume as our computational domain, is up to an isomorphism the "upper part" of T.

Recursive specification of elements in $P_{MS}(B)$

In $P_{MS}(B)$ (and also in T), we can solve some problems of the following sort:

Find an element x such that $x = G(x)$ and $E(x)$ is a minimal set. The smallest set $E(x)$, of course, belongs to the strongest set x that satisfies $x = G(x)$, that is, it belongs to the greatest fixed point of G. As an example consider

$$(1) \qquad G = \lambda x.\{\{nil\}\} \cap w \times x.$$

The greatest fixed point of G is

$$FIX(G) = \{\{nil\} \cap w \times \{nil\} \cap w \times (w \times \{nil\}) \cap \ldots\},$$

and $E(FIX(G))$ is the set of all lists over elements in $E(w)$.

There is no problem with the existence of a greatest fixed point because our domain is almost a complete lattice. We can complete the lattice by adding the strongest element τ. Hence, unless it happens to be τ itself, the greatest fixed point [Tarski55]

$$u = FIX(G) = \cup\{x \mid G(x) \supset x\}$$

is guaranteed to exist in the original domain. We obtain for the greatest fixed point operator

$$FIX(G) = \bigcap_{n}^{\infty} G^n(\tau).$$

With the requirement that G is downward continuous at the fixed point, the proof that FIX(G) is (a) a fixed point and (b) the

greatest fixed point follows the pattern of the analogous proof for the least fixed point operator found, e.g., in [Stoy77].

A function f is downward continuous if in downward directed sets S

$$f(\cap S) = \cap\{f(z) \mid z \in S\},$$

and a downward directed set contains for every pair of elements x and y an element t such that $t \subset x \cap y$.

The function G defined by (1) is clearly downward continuous (it is even downward additive since S does not need to be directed), because of the well known relationship

$$w \times (\cap S) = \cap\{w \times x \mid x \in S\}.$$

Functions

The elementary propositions that the system makes about functions are of the form

$$(x \mapsto y)$$

where "$\mapsto$" denotes the functional assignment and x and y are other propositions. This assertion is true for all functions (total elements) ψ where y is true of $\psi(\alpha)$ if x is true of α, that is

$$(x \mapsto y) \subset \psi \Rightarrow (x \subset \alpha \Rightarrow y \subset \psi(\alpha)).$$

Thus

$E(x \mapsto y)$ is a set of functions from $E(x)$ to $E(y)$.

In other words, if a function contains the element $(x \mapsto y)$, then we can be sure that it maps values from $E(x)$ to $E(y)$ since any assignment $(u \mapsto w)$ with $x \subset u$ for which w is not consistent with y is not consistent with $(x \mapsto y)$.

For the purpose of type checking and type inference, we go one step further and consider a function f to have the domain type x and the codomain type y only if f contains $(x \mapsto y)$ as an element.

Most assertions about functions will be (usually infinite) sets of elementary assertions. Let f be such a set. Then

$$f \subset \psi \;\&\; x \subset \alpha$$

$$\Rightarrow f(x) = \cup\{y \mid (z \mapsto y) \in f \text{ for some } z \subset x\} \subset \psi(\alpha).$$

This, of course, is the usual definition of application.

Now suppose we know only a type t of some value α, that is, $t \subset \alpha$, and we wish to find out what a function ψ defined by f computes if applied to α. From the previous discussion follows

if $\psi: E(r) \to E(s)$ then $(r \mapsto s) \in f$, and, hence,

if $r \subset t \subset \alpha$ then we know that $s \subset \psi(\alpha)$.

In words, by applying ψ's approximation f to a type t of α we obtain as a result (at least) the assertion that $\psi(\alpha)$ has the type of ψ's codomain provided that the type t is stonger than the domain type of ψ, that is, provided that $E(t) \subset E(r)$.

This looks like a reasonable basis for type checking. But what

happens if E(t) is not a subset of E(r)? In the worst case, we have to expect f(t) = {Δ} = ⊥, which tells us that type checking fails.

But here we seem to have a serious problem. The element Δ is the least informative element in D and as such it is part of every assertion. Thus, if we interpret ⊥ to mean "error", then the attribute "error" applies to every assertion, which is a rather useless convention!

On the other hand, categorically forbidding ⊥ to mean "error" seems to be equally unfounded. Imagine that a function f is defined exclusively by (r ↦ s). Now, given some type t of a prospective parameter for f, should we not be able to conclude "valid" if r ⊂ t and "wrong" otherwise? This conclusion hardly poses any computability problem!

Termination

Solving this problem amounts to recognizing that observed termination of a procedure that defines a function is additional information and, thus, makes for stronger assertions. Further, type checking involves computations that terminate with results that are usually considered partial, but, because of termination, must now be considered total! For example, with type calculations propositions that only claim "α is some element in E(x)" where E(x) may be quite a large set, are not always preliminary but may be final. Such propositions, then, are not partial results that,

with some luck, may become stronger, but final results that are presented as the complete information attainable.

But what is the relative position of these new values (for example, "final INT") in our domain? Is "final ⊥" weaker than "final INT"? The answer is no! If "final ⊥" were weaker than "final INT", then "final ⊥" could be strengthened to "final INT"; But, since "final ⊥" is a final result, no such improvement is possible; "final ⊥", as all other final values, is a total element.

As an immediate consequence we may conclude that "termination" is not the same kind of proposition as, for example, INT. Rather, for every partial element there is exactly one (stronger) total element that occurs upon computation of the partial element if the defining procedure stops. For example, there is final_⊥ ⊢ ⊥ and final_INT ⊢ INT, and so on.

A function computes a final element if two conditions are satisfied:

(i) the defining procedure terminates, and
(ii) all parameters are final elements.

The second condition is not always necessary since there may be parameters whose improvement would not alter the result; in this case the result may be declared final although all data may not yet be in.

Since final_⊥ is a total element, the comparison

if final_⊥ ⊂ x then ... {e.g.error}

is true exclusively for x = final_⊥. Hence a function that computes, for example, "wrong" for final_⊥ and "valid" for final_INT is monotonic as it should be!

III. THE BASIC RULE FOR TYPE INFERENCE AND TYPE CHECKING

From the above we obtain the following simple rule:

Because of monotonicity, we can compute a type for any expression by replacing the non-variable constituents of the expression by their known types. The only exception to this rule is the fixed point operator. The least fixed point of an expression weakened by the replacement of its parts by their types is usually ⊥. We shall find that in this case the greatest fixpoint gives a proper type.

Note that this rule is the direct consequence of the fact that "t is a type of x" and "t ⊂ x" mean the same thing. Hence the following lines are trivial observations!

Let f approximate some function and let t ⊂ x be a type of x. Then

(2.1) f(t) ⊂ f(x) is a type of f(x), and

with h ⊂ f, that is, with h being a type of f

(2.2) h(t) ⊂ h(x) ⊂ f(x),

(2.3) $\lambda x.h(x) \subset \lambda x.f(x)$, and

(2.4) $FIX(H) \subset FIX(H) \cup fix(F)$ for $H \subset F$.

While, by (2.1), f(t) is a type of f(x), the strongest type justified by f's type h is h(t). Since type mappings are usually considerably simpler than the corresponding value mappings, it is also much easier to evaluate h(t) than to evaluate f(t).

The remainder of this paper will demonstrate by concrete examples how this general rule allows type inference, type checking, handling of polymorphic functions and of user-defined polymorphic types.

IV. A LAMBDA NOTATION FOR THE DOMAIN ELEMENTS

To talk about specific elements in the domain T, we use lambda notation augmented by an expression for denoting functional assignments:

$\lambda x.\ a \in x \to b$, read "a in x gives b", denotes $a \mapsto b$.

The rest of our notation is conventional. Lower case letters denote elements in $|T|$ or in CON_T, and upper case letters as well as numerals and quoted letters denote members of DD_T or DD_A. Letters from the beginning of the alphabet denote constants, and letters from the end of the alphabet denote variables. As we did

above, we use identifiers, such as 'INT', 'CHAR', 'cond', or 'fac', with the traditional meaning to denote elements of $|T|$. All set braces are dropped. This notation, tailored after Scott's language LAMBDA, is informally defined as follows:

Definition:

(3.1) C denotes $\{\{C\}\}$, if $C \in DD_A$ (e.g. 3 denotes $\{\{3\}\}$)

(3.2) f(x) denotes $\{C \mid b \mapsto C \in f \text{ for some } b \subset x\}$;

(3.3) $\lambda x.E$ denotes $\{(b \mapsto C) \mid C \in E[b/x]\}$;

(3.4) $x \cup y$ denotes $\{Y \mid Y \in x \text{ or } Y \in y\}$

(3.5) $x \cap y$ denotes $\{Y \mid Y \in x \text{ and } Y \in y\}$

(3.6) $c \varepsilon x \to y$ denotes $\{Y \in y \mid c \subset x\}$

Notes: (i) $E[b/x]$ is the value of the expression E if x assumes the value of b (see [Scott76]).

(ii) We call the expression defined by (3.6) a function element because it permits us to specify a single functional assignment as a lambda expression by

$\lambda x.c \varepsilon x \to y$ denoting $\{(c \mapsto Y) \mid Y \in y\}$.

The value "c" in (3.6) must be an element of CON_T, and it must not be a variable since otherwise monotonicity is lost.

The following examples show how function elements are used to

define other functions. We define the conditional "if" by

$$\text{if} \equiv \lambda z \lambda x \lambda y.(\text{true} \ \varepsilon \ z \to x) \cup (\text{false} \ \varepsilon \ z \to y) \cup (\text{BOOL} \ \varepsilon \ z \to x \cap y),$$

where

$$\lambda z \lambda x \upsilon.\text{BOOL} \ \varepsilon \ z \to x \cap y$$

is a type of "if". and, with the assumption that "x + y" produces the integer sum if handed integer numbers x and y, we define (curried) addition by

$$\text{plus} \equiv \lambda x \lambda y.(\text{INT} \ \varepsilon \ x \to (\text{INT} \ \varepsilon \ y \to \text{INT}) \cup (N1 \ \varepsilon \ x \to (N2 \ \varepsilon \ y \to N1 + N2 \mid N1, N2 \text{ are integers})).$$

Below, we refer to other elementary operation (such as -, *, etc.) as it is convenient.

The following rules are useful for the transformation of expressions that contain function elements. These rules are derived in a straightforward manner from (3.1) - (3.6).

Lemmas:

(4.1) $(c \ \varepsilon \ x \to y) \subset y$;

(4.2) $(c_1 \to (c_2 \to y)) = (c_2 \to (c_1 \to y))$

{where the $c_i = (n_i \ \varepsilon \ x_i)$};

(4.3) $(z \ \varepsilon \ (c \to x) \to y)$

$= \text{if } (z = \perp) \text{ then } y \text{ else } (c \to (z \varepsilon x \to y));$

(4.4) $(c \to E(c \to y)) = (c \to E(y));$

(4.5) $a \varepsilon f(b)$ iff $(\lambda x.b \varepsilon x \to a) \varepsilon f$

{x not free in a or b};

(4.6) $(z \varepsilon (c \to x \mid P(c,x)) \to y \mid Q(z,y))$

$= (c \to (z \varepsilon x \to y) \mid P(c,x) \text{ and } Q(z,y));$

V. A SIMPLE EXAMPLE OF TYPE INFERENCE

A simple example may illustrate how our lambda notation can be used for the computation of types.

Let INT $\subset$ N and ODD $\subset$ 2N+1 where N is an integer number (a proposition). Hence INT $\subset$ ODD. Assuming that "times" is the integer multiplication similarly defined as "plus" above with the type

$\lambda x \lambda y.(\text{INT } \varepsilon\ x \to (\text{INT } \varepsilon\ y \to \text{INT})) \subset \text{times}$

we define the squaring function

$\text{sqr} \equiv \lambda x.\text{times}(x)(x)$

and claim that

$\text{tsq} \equiv \lambda x.\text{INT } \varepsilon\ x \to \text{INT}$

is a type of sqr. We verify this claim by applying (2.2), that

is,

if h is a type of f, then h(x) is a type of f(x).

Thus,

$$tsq = \lambda x.\langle \text{type of times} \rangle(x)(x)$$
$$= \lambda x.(INT \varepsilon x \to (INT \varepsilon x \to INT))$$
$$= \lambda x.(INT \varepsilon x \to INT) \quad \{\text{by (4.4)}\}. \ \square$$

Now suppose we wish to check if sqr may be applied to some number z of type "ODD". We find that

$$tsq(ODD) = (\lambda x.INT \varepsilon x \to INT)(ODD)$$
$$= INT \varepsilon ODD \to INT$$
$$= INT.$$

Notice that this type of natural coercion occurs simply as a result of monotonicity.

VI. TYPES OF RECURSIVELY DEFINED FUNCTIONS

Consider the definition of the factorial

$fac = \lambda x.$if $x = 0$ then 1 else $x * fac(x-1)$, or

$fac = fix(F)$, with

$F = \lambda f \lambda x.$if $x = 0$ then 1 else $x * f(x-1)$.

While the expected type $\lambda x.\mathrm{INT}\ \varepsilon\ x \rightarrow \mathrm{INT}$ is consistent with the recursive definition, it is not explicitely produced by the fixed point computation. To be sure, fix(F) computes the correct values if given any (proposition that denotes a) single number, but we would like to include explicitely the assertion that INT is mapped to INT. If we do not add this assertion, then type inference (or checking) across fac is impossible. In order to determine the type of fix(F) we analyse the type of the functional F. With the type for the conditional given above we obtain

$$F \supset \lambda f \lambda x.\ \mathrm{BOOL}\ \varepsilon\ (x=0) \rightarrow (1 \cap x * f(x-1))$$

With rules (4.1) - (4.6) and the types for "=", "*", and "-", this leads to

$$F \supset \lambda f \lambda x.\ ((\lambda u.\mathrm{INT}\ \varepsilon\ u \rightarrow \mathrm{INT})\ \varepsilon\ f \rightarrow (\mathrm{INT}\ \varepsilon\ x \rightarrow \mathrm{INT})).$$

We see that the type of F(f) is Δ unless the type of f is at least $\lambda u.\mathrm{INT}\ \varepsilon\ u \rightarrow \mathrm{INT}$. We want f to have the stronger type.

In general we want the strongest possible type suggested by F that is consistent with the least fixed point of F. The strongest possible type is the greatest fixed point of F's type. We now prove

(5) Lemma: let H be a type of F. Then FIX(H) is always consistent with the least fixed point of F.

Proof: Let $H \subset F$ be a type of F and $z = H(z)$ the strongest fixed point of H. Because of monotonicity $H(z) \subset F(z)$; thus

$\cup \{F^n(FIX(H)) \mid n = 0,1,\ldots\}$ is a fixed point of F,

more accurately, it is the least fixed point of F that is stronger than FIX(H). Thus FIX(H) and fix(F) are consistent. □

With this result the computation of the type of a recursively defined function becomes easier. Consider again

$$F \supset H = \lambda f \lambda x.\ BOOL\ \varepsilon\ (x=0) \rightarrow (1 \cap x * f(x-1)).$$

The greatest fixed point of H is

$$FIX(H) = \lambda x.\ BOOL\ \varepsilon\ (x=0) \rightarrow (1 \cap x * \tau)$$

$$= \lambda x.(INT\ \varepsilon\ x \rightarrow INT \cup 1\ \varepsilon\ x \rightarrow 1),$$

{note that both INT ε x and 1 ε x guarantee BOOL ε (x=0)}.

VII. TYPES AND POLYMORPHIC FUNCTIONS

The weakest (nontrivial) type of the approximation f of a function is always of the form

$$d \rightarrow r$$

where E(d) and E(r) are the domain and, respectively, the codomain of the function approximated. In our theory, polymorphic functions are no exception. Polymorphic function just have, loosely speaking, greater domains. Take as a simple example the identity

$$id \equiv \lambda x.x$$

with the type

$$tid \equiv \lambda x.\Delta \varepsilon x \rightarrow \Delta.$$

(Recall that Δ serves as the universal type).

In order to find out if restrictions of id to smaller domains yield a smaller codomain, simply apply "id" to the type of the domain in question, for example, to INT:

$$id(INT) = (\lambda x.x)(INT) = INT.$$

Many polymorphic functions have this very simple type behavior, that is, they map types and "real" data (e.g. numbers such 3 or 25) in the same way.

Notation: Since it is weaker than the complete function f, this mapping of types is itself a type of f. Below, if we talk about <u>the</u> type of a function, then we refer to the value t for which

$$E(t) = (\text{domain} \rightarrow \text{codomain}).$$

For distinction, we refer to the function's stronger type as f's <u>type mapping</u> t_f; this gives <u>the</u> type for the restriction of f to the E(s) as

$$\lambda x.\ s \varepsilon x \rightarrow t_f(s).$$

Some polymorphic functions treat types differently from ordinary values. As an example consider the operation

"cons" that takes an item of type t and a list of type List(s) and computes a list of type List(t ∩ s). While "cons" applied to ordinary values x and y computes a "pair" of its two parameters, applied to types it computes a single new type. As with all objects, also a function of this kind is formally, as a special case, its own type. But we exclude this choice when we talk about "a type" of a function; we rather refer to a mapping strictly weaker than the function itself.

As a second example consider functional composition defined by

$$comp \equiv \lambda f \lambda g \lambda x.f(g(x))$$

with the type

$$tcomp \equiv \lambda f \lambda g \lambda x.(\lambda u.\Delta \varepsilon u \to \Delta) \varepsilon f \to (\lambda w.\Delta \varepsilon w \to \Delta) \varepsilon g \to (\Delta \varepsilon x \to \Delta)$$

and suppose, for example, that

$$t_f \equiv \lambda y.INT\ \varepsilon\ y \to CHAR, \text{ and}$$
$$t_g \equiv \lambda z.REAL\ \varepsilon\ z \to ODD.$$

Then by (2.2) $comp(t_f)(t_g)$ is a type of comp(f)(g), and we obtain

$$comp(t_f) = \lambda g \lambda x.(\lambda y.INT\ \varepsilon\ y \to CHAR)(g(x)) = \lambda g \lambda x.INT\ \varepsilon\ g(x) \to CHAR,$$

and thus

$comp(t_f)(t_g)$

$= \lambda x.INT\ \varepsilon\ (\lambda z.REAL\ \varepsilon\ z \to ODD)(x) \to CHAR$

$= \lambda x.INT\ \varepsilon\ (REAL\ \varepsilon\ x \to ODD) \to CHAR$

$= \lambda x.REAL\ \varepsilon\ x \to (INT\ \varepsilon\ ODD \to CHAR)$ {by (4.3)}

$= \lambda x.REAL\ \varepsilon\ x \to CHAR$ {since INT ε ODD}

which is the desired result.

Now consider MacQueen's paradigm for his type inference rules:

$twice \equiv \lambda h \lambda z.h(h(z))$.

We can compute the type of twice in two ways; (i) we may use the type of "comp" interpreting "twice" as comp(f)(f), or (ii) we may use the type of the operation of application.
We shall proceed with the second method. With the universal type Δ, the type of application is

$tapp \equiv \lambda h \lambda z.(\lambda u.\Delta\ \varepsilon\ u \to \Delta)\ \varepsilon\ h \to (\Delta\ \varepsilon\ z \to \Delta)$

$tapp(f)(x) = (\lambda u.\Delta\ \varepsilon\ u \to \Delta)\ \varepsilon\ f \to (\Delta\ \varepsilon\ x \to \Delta)$

and

$tapp(f)(tapp(f)(x))$

$= (\lambda u.\Delta\ \varepsilon\ u \to \Delta)\ \varepsilon\ f \to (\Delta\ \varepsilon\ tapp(f)(x) \to \Delta)$

Since the term $(\lambda u.\Delta\ \varepsilon\ u \to \Delta)\ \varepsilon\ f$ occurs both at the beginning and within tapp(f)(x), we can remove its second occurence by (4.4) and obtain

$tapp(f)(tapp(f)(x))$

$= (\lambda u.\Delta\ \varepsilon\ u \rightarrow \Delta)\ \varepsilon\ f \rightarrow (\Delta\ \varepsilon\ (\Delta\ \varepsilon\ x \rightarrow \Delta) \rightarrow \Delta).$

The second part can be processed by rule (4.3) and gives

$\mathrm{tapp}(f)(\mathrm{tapp}(f)(x))$

$= (\lambda u.\Delta\ \varepsilon\ u \rightarrow \Delta)\ \varepsilon\ f \rightarrow (\Delta\ \varepsilon\ x \rightarrow (\Delta\ \varepsilon\ \Delta \rightarrow \Delta))$

$= (\lambda u.\Delta\ \varepsilon\ u \rightarrow \Delta)\ \varepsilon\ f \rightarrow (\Delta\ \varepsilon\ x \rightarrow \Delta),$

and the type of twice becomes

$\mathrm{ttwice} = \lambda h \lambda z.\mathrm{tapp}(h)(\mathrm{tapp}(h)(z))$

$= \lambda h \lambda z.(\lambda u.\Delta\ \varepsilon\ u \rightarrow \Delta)\ \varepsilon\ h \rightarrow (\Delta\ \varepsilon\ z \rightarrow \Delta).$

But what is the type mapping of twice?

Again, it is the function

$\lambda h \lambda z.h(h(z))$

itself! Suppose, for a function f we are given the type mapping

$\{t1 \leftrightarrow t2\ ,\ t2 \leftrightarrow t3\} \subset f.$

(The type of f is $(t1 \cap t2) \leftrightarrow (t2 \cap t3)$).

In our lambda notation the type mapping of f is

$\lambda z.t1\ \varepsilon\ z \rightarrow t2\ \cup\ t2\ \varepsilon\ z \rightarrow t3.$

We obtain for f(x)

$t1\ \varepsilon\ x \rightarrow t2\ \cup\ t2\ \varepsilon\ x \rightarrow t3,$

and for f(f(x))

$$t1 \varepsilon (t1 \varepsilon x \to t2 \cup t2 \varepsilon x \to t3) \to t2$$
$$\cup\ t2 \varepsilon (t1 \varepsilon x \to t2 \cup t2 \varepsilon x \to t3) \to t3$$

Assuming that the types t1, t2, and t3 are unrelated, we find by (4.1) that the first term computes $\perp$. For the same reason the second term yields

$$t2 \varepsilon (t1 \varepsilon x \to t2) \to t3,$$

which, by rule (4.3), gives

$$t1 \varepsilon x \to (t2 \varepsilon t2 \to t3) = t1 \varepsilon x \to t3.$$

This gives us the type of $\lambda x.f(f(x))$ as

$$\lambda x.t1 \varepsilon x \to t3. \quad \square$$

Now consider $\lambda f.f(f)$. It has been pointed out by MacQueen that the type mapping of this function is defined as a fixed point of

$$\lambda s \lambda t.s \varepsilon t \to a.$$

This is not a legal lambda expression in our notation since the variable "s" occurs in a position where only constants are permitted. Because of this defect, the expression is antimonotonic in s, and the least fixed can not be found by the usual methods.

Note: At the June 1983 workshop on types and polymorphism in Pittsburgh, MacQueen described a solution to this problem for functions that are contracting in a suitable metric.

But it turns out that the function again is its own type mapping. Suppose we wanted to find out whether a function with the type

$$\lambda t.(\lambda s.a \varepsilon s \to b) \varepsilon t \to c \cup a \varepsilon t \to b$$

is a proper parameter for $\lambda f.f(f)$ and, if so, which result type it brings about. We obtain

$$(\lambda f.f(f))(\lambda t.(\lambda s.a \varepsilon s \to b) \ldots)$$

$$= (\lambda t.(\lambda s.a \varepsilon s \to b) \varepsilon t \to c \cup a \varepsilon t \to b)(\lambda t.(\ldots)\ldots)$$

$$= (\lambda s.a \varepsilon s \to b) \varepsilon (\lambda t.(\ldots) \varepsilon t \to c \cup a \varepsilon t \to b) \to c \cup \ldots)$$

$$= c. \quad \square$$

VIII. POLYMORPHIC TYPES AND DATA ABSTRACTION

We discuss polymorphic types by working through an example. We choose the type of binary trees where the labels stored at the nodes can be of any type. First we define a functional whose greatest fixed point is a function from elements (types) to trees:

$$\text{TREE} \equiv \text{FIX } \lambda h \lambda x.(\text{nil} \cap (x \times (h(x) \times h(x)).$$

The expression TREE(t) has the value

$$\begin{aligned} &\text{nil} \\ \cap\ &(t \times (\text{nil} \times \text{nil})) \\ \cap\ &(t \times (\text{nil} \times (t \times (\text{nil} \times \text{nil})))) \end{aligned}$$

$\cap$ (t $\times$ ((t $\times$ (nil $\times$ nil)) $\times$ nil))

$\cap$ ...

It will prove useful to define the second, similar type

CTREE $\equiv$ λz.(z $\times$ (TREE(z) $\times$ TREE(z))).

Note that

nil $\cap$ CTREE(t) = TREE(t).

Since TREE is injective, there is the left inverse EERT with the properties

EERT(TREE(x)) = x, ERRT(CTREE(x)) = x, and EERT(nil) = $\top$.

The last equation is true because

EERT(nil) $\supset$ EERT(TREE(a)) for all a, and hence

EERT(nil) $\supset$ U {a | any a}.

Since the set of all a's is inconsistent, its upper bound is $\top$.

Now we wish to define the following tree primitives

NIL_T: $\rightarrow$ E(TREE(Δ)) {computes nil};

COMBO: E(Δ $\times$ (TREE(Δ) $\times$ TREE(Δ))) $\rightarrow$ E(CTREE(Δ));

LEFT: E(CTREE(Δ)) $\rightarrow$ E(TREE(Δ));

RGHT: E(CTREE(Δ)) $\rightarrow$ E(TREE(Δ));

INFO: E(CTREE(Δ)) $\rightarrow$ E(Δ);

IF_NIL: E(TREE(Δ)) → (E(nil) → E(Δ))

→ (E(CTREE(Δ)) → E(Δ)) → E(Δ).

The last operation, which could be read as

IF_NIL t then f(t) else g(t),

is a filter [Martin83] which splits the domain E(TREE) into E(nil) and E(CTREE) giving "nil" as a parameter to its then-part and values of type CTREE to its else-part. A filter permits the syntactic checking that the proper test for "nil" is present in the code. We introduce this device here because we need it for our last example below.

With the projections P(x × y) = x and Q(x × y) = y, we obtain the following lambda expressions:

NIL_T ≡ nil

COMBO ≡ λxλyλz. x × (y × z);

LEFT ≡ λx.P(Q(x));

RGHT ≡ λx.Q(Q(x));

INFO ≡ λx.P(x);

IF_NIL ≡ λtλfλg.NIL ε t → f(t)

∪ CTREE(Δ) ε t → g(t).

Now we wish that these functions have the following type behavior:

NIL_T $\supset$ nil;

COMBO $\supset$ $\lambda z \lambda l \lambda r.$ TREE(Δ) ε l $\rightarrow$ TREE(Δ) ε r
$\rightarrow$ CTREE(z $\cap$ EERT(l) $\cap$ EERT(r));

LEFT $\supset$ $\lambda t.$ CTREE(Δ) ε t $\rightarrow$ TREE(EERT(t));

RGHT $\supset$ $\lambda t.$ CTREE(Δ) ε t $\rightarrow$ TREE(EERT(t));

INFO $\supset$ $\lambda t.$ CTREE(Δ) ε t $\rightarrow$ EERT(t);

IF_NIL $\supset$ $\lambda t \lambda f \lambda g.$TREE($\Delta$) ε t $\rightarrow$ f(nil) $\cap$ g(CTREE(EERT(t)))

It is easy to verify that these expressions are proper types of the primitive operations. Consider, for example, LEFT. The type of LEFT applied to anything that is not at least as strong as CTREE(Δ) gives Δ; thus we need an analysis only for t = CTREE(z) with $\Delta \subset$ z. For this, the type computes TREE(EERT(CTREE(z))) = TREE(z).

Now the function LEFT applied to CTREE(z), that is, to

$$z \times (TREE(z) \times TREE(z)),$$

gives

$$P(Q(z \times (TREE(z) \times TREE(z)))) = TREE(z). \quad \square$$

Abstraction

The operations above are not only applicable to trees but also to all kinds of other values that happen to have the proper structure. Furthermore, objects of type TREE can be manipulated by any expression by means of the cartesian product and the projections P and Q. How can we make the tree primitives the exclusives operations for objects of type tree, and how can we restrict these operations to trees? We have to construct a new domain!

This new domain contains all elements of the old one. In addition, there is an atomic proposition A_T in the new domain for each cross product T denoting a tree object in the old domain. If TREE(a) or CTREE(a) is the greatest lower bound of a set of such cross products $\{T_1, \ldots\}$ in the old domain, then its counterpart in the new domain is the greatest lower bound of the set $\{A_{T_1}, \ldots\}$ of new atoms. The operations defined among the old tree objects have corresponding operations among the new tree objects. In short, there is a homomorphism H from old tree objects s (TREE(Δ) $\subset$ s) to the elements produced by downward closing the new atoms. H preserves the semantics of the tree primitives. In our examples with trees, this mapping underlying the homomorphism is a bijection; in general one can only claim that such a mapping is a surjection.

The information needed to specify the new domain from the old one consists of

the definitions of the carrier types in the old domain,

the semantics of the primitives in the old domain,

the type (mapping) of the primitives in the new domain.

The latter (the type behavior) is needed, because in the old domain there may be an ambiguity about the meaning of an object. For example, for the data type "tree" we might consider a primitive of the following form:

$$\text{DECOMP: } E(\text{CTREE}(\Delta)) \to E(\Delta \times (\text{TREE}(\Delta) \times \text{TREE}(\Delta)).$$

Its implementation in the old domain is simply the identity, but its type behavior is defined by

$$\lambda t.\ \text{CTREE}(\Delta)\ \varepsilon\ t \to \text{EERT}(t) \times (\text{TREE}(\text{EERT}(t)) \times \text{TREE}(\text{EERT}(t))).$$

We mentioned above that, in the general case, the mapping from the representations of objects in the old domain to the abstract objects in the new domain may not be bijective but merely surjective. This occurs if the representation of objects is not unique. Consider, for example, the representation of sets by lists. In this case, the mapping H carries each set of equivalent representations to one abstract object.

Restricted to total elements, the function H is Hoare's abstraction function [Hoare72] (retrieve function in [Jones80]). Hence, within this model, the algebraic theory of data abstraction has its natural place.

A final example: apply f to all nodes of a tree

This example is similar to the function MAP treated by Milner [Milner78]. Consider the function fix(F) where

```
F = λhλfλt.if_nil(t)
              then NIL_T
              else COMBO(f(INFO(t)), h(f,LEFT(t)), h(f,RGHT(t))).
```

The type of fix(F) is the greatest fixed point of a type of F. With the type of IF_NIL we obtain

$$
\begin{aligned}
& \lambda f\lambda t.\mathrm{nil} \cap \mathrm{COMBO}(f(\mathrm{INFO}(\mathrm{CTREE}(\mathrm{EERT}(t)))), \tau, \tau) \\
= \; & \lambda f\lambda t.\mathrm{nil} \cap \mathrm{COMBO}(f(\mathrm{EERT}(t)), \tau, \tau) \\
\supset \; & \lambda f\lambda t.\mathrm{nil} \cap \mathrm{TREE}(\Delta) \in \tau \to \mathrm{TREE}(\Delta) \in \tau \\
& \qquad \to \mathrm{CTREE}(f(\mathrm{EERT}(t)) \cap \mathrm{EERT}(\tau) \cap \mathrm{EERT}(\tau)) \\
= \; & \lambda f\lambda t.\mathrm{nil} \cap \mathrm{CTREE}(f(\mathrm{EERT}(t))) \\
= \; & \lambda f\lambda t.\mathrm{TREE}(f(\mathrm{EERT}(t))). \quad \square
\end{aligned}
$$

IX. CONCLUSION

The suggested model interprets types as elements in an augmented domain. These elements approximate the elements of which they are types. By monotonicity, we obtain a type for any expression E by evaluating the expression E' obtained from E by replacing all constituents by their known types. The strongest type attainable for the least fixed point of a functional F is the greatest fixed

point of a type of F.

The augmented domain also allows the definition of new recursive types in a most natural way. Functionals can be defined whose greatest fixed points define functions from existing types to new types. The structure of such a functional specifies directly the structure of the elements that make up the carrier set(s) of the new type(s).

Data abstraction is accomplished by creating a new domain which is the homomorphic image of an existing domain. Preserving the semantics of the corresponding primitive operations, this homomorphism takes certain compound objects of the old domain to new atomic objects in the new domain.

References:

[Hoare72] Hoare,C.A.R. "Proof of correctness of data representations," Acta Informatica, Vol 1 (1972), pp 271-281,

[Jones80] Jones,Cliff B. Software Development: A Rigorous Approach. Prentice-Hall International, 1980

[MacQueen82] MacQueen,D.B. "A semantic model of types for applicative languages," Conference Record, 1982 ACM Symposium on Lisp and Functional Programming.

[Martin82] Martin, Johannes J. "A semantic model of data types," Technical Report CS8201[illegible]-R VPI & SU, May 1982, also presented at a seminar at the Queen's University of Belfast in July 1982.

[Martin83] Martin, Johannes J. "Precise typing and filters," To appear in Information Processing Letters.

[Milner78] Milner, R. "A theory of type polymorphism in programming," JCSS 17(3) (Dec. 1978) pp. 348-357

[Scott82] Scott, Dana S. "Domains for denotational sematics," A corrected and expanded version of a paper prepared for ICALP '82, Aarhus, Denmark, July 1982.

[Smyth78] Smyth, M. "Power domains," Journal of Computer and System Science, (1978)

[Stoy77] Stoy, Joseph E. Denotational semantics: The Scott-Strachey approach to programming language theory, M.I.T. Press, 1977

[Tarski55] Tarski, Alfred "A lattice-theoretical fixpoint theorem and its applications," Pacific Journal of Mathematics, 5 (1955) pp. 285-309.