The University of Warwick

THEORY OF COMPUTATION

REPORT NO. 20

DATA TYPES AS OBJECTS

BY

ADI SHAMIR

WILLIAM W. WADGE

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

June 1977.

DATA TYPES AS OBJECTS

Adi Shamir
Mathematics Department
MIT
Cambridge, Mass. USA

William W. Wadge
Computer Science Department
University of Warwick
Coventry, UK

Abstract

In this paper we present a new approach to the semantics of data types, in which the types themselves are incorporated as elements of the domain of data objects. The approach allows types to have subtypes, allows genuinely polymorphic functions, and gives a precise semantics for recursive type definitions, including definitions with parameters. In addition, the approach yields simple and straightforward methods for proving type properties of recursive programs.

A slightly revised version of this paper appears in the proceedings of the Fourth International Conference on Automata, Languages and Programming (ICALP), held in Turku, Finland, 18-22 July, 1977. pp. [illegible]

0. Informal Introduction

Most current type systems (e.g. that of LCF, as in [1]) are based on A. Church's 1940 paper, "A simple theory of types" ([2]). In these systems there are a number of unrelated ground data types and an an arrow operation for forming function types of finite level.

The first problem with such systems is that each data object must have a unique type. Thus integers, for example, cannot be considered as a special case of real numbers, and in general one type cannot be a subtype of another.

A second problem, related to the first, is that polymorphism is not possible; each argument of a function must be of some specified single type. For example, we cannot have a single addition operation capable of adding both reals and integers; instead, we need four functions of various types for the various possible combinations. Especially serious is the lack of a general <u>if-then-else</u> conditional whose domains are left unspecified.

The third problem is that in a Church system there are in fact two 'meta'-types, namely types and data objects, which cannot be mixed together. For example, the intuitively true identity

$$\underline{\text{eveninteger}} + 1 = \underline{\text{oddinteger}}$$

relating the types <u>eveninteger</u> and <u>oddinteger</u> and the data object 1 makes no sense in a Church system (except, perhaps, informally). Even unmixed recursive type equations are problematic because the types in such a system do not constitute a domain.

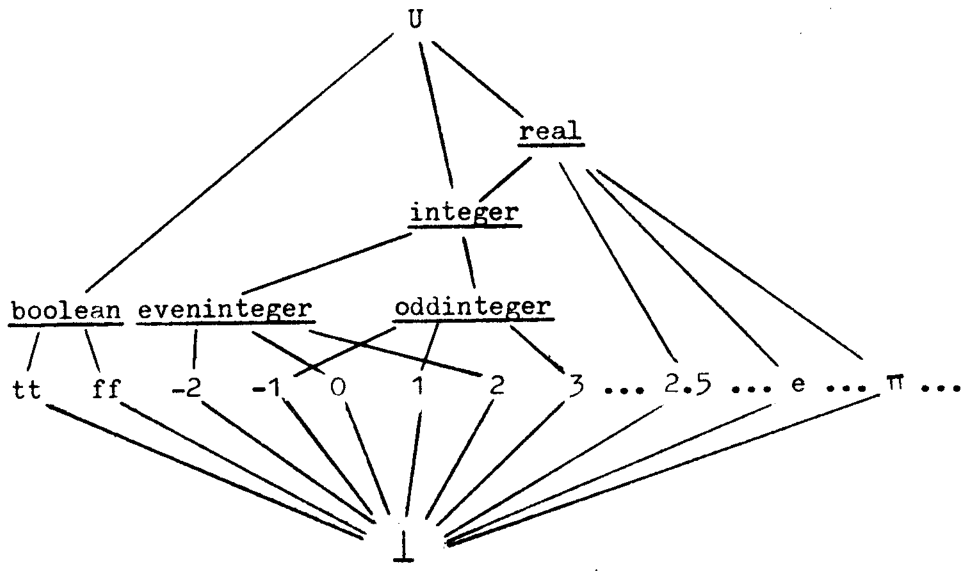
Our approach is to incorporate all the objects and data type together in a single unified domain. Any element of the resulting domain then has two roles:

(i) it is a <u>data object</u>, at which functions can be defined, including of course functions which are least fixed points of recursive definitions;

(ii) it is the <u>type</u> of all objects which approximate it; the assertions " $x \sqsubseteq y$ ", " x is of type y " and "any object of type x is of type y " are therefore all equivalent.

Given an initial domain D of data objects, and a collection of intuitive data types, we form a new domain $\hat{D}$ (which we will call a type extension of D) by adding the types as new data objects in the sense of (i). The extended $\sqsubseteq$ relation on $\hat{D}$ is determined by the equivalence in part (ii): a type object is placed <u>above</u> the data objects of the type in question, and above all its subtypes (so that $\sqsubseteq$ orders types by inclusion).

For example, if our original (untyped) domain contains the real numbers, the truth values, and $\perp$ (ordered as a flat domain), and if we are interested in the types <u>eveninteger</u>, <u>oddinteger</u>, <u>integer</u>, <u>real</u>, <u>boolean</u> .. and the universal type U, we get the following extended domain:



Our system, unlike most others, does not associate particular types with elements of D, and so a question like "what is the type of 2" is meaningless in our system. The relation $\sqsubseteq$ lets us answer only questions of the form "is x of type y?". For example, the assertions " 2 is of type <u>integer</u>", " 2 is of type <u>real</u> ", " <u>integer</u> is of type <u>real</u> " and " $\perp$ is of type 2 " are all true in $\hat{D}$. Note that the least element $\perp$ is of all types, and that everything is of the universal type U.

Monotonic functions over the original domain D can be extended to monotonic functions over $\hat{D}$ in many ways. Of particular interest are the tightly extended functions (i.e. the least monotonic extensions) whose definitions are usually clear. For example, the tight extension of addition (with $\hat{D}$ as above) yields

$$2 + 3 = 5$$

$$\underline{\text{integer}} + 3 = \underline{\text{integer}}$$

$$\underline{\text{integer}} + \underline{\text{real}} = \underline{\text{real}}$$

$$\underline{\text{integer}} + \pi = \underline{\text{real}}$$

$$\underline{\text{oddinteger}} + \underline{\text{oddinteger}} = \underline{\text{eveninteger}}$$

Certain functions in $[\hat{D}\to\hat{D}]$ (the space of monotonic functions from $\hat{D}$ to $\hat{D}$ with the standard ordering) play the same role as the type objects added to D, i.e. they embody intuitive types. Given x and y in $\hat{D}$ we define the arrow function $x\to y$ to be the following element of $[\hat{D}\to\hat{D}]$:

$$\lambda z\ \underline{\text{if}}\ z \sqsubseteq x\ \underline{\text{then}}\ y\ \underline{\text{else}}\ U\ .$$

This function maps objects of type x into the object y itself, and all other objects into the object U. It is easily verified that for any f in $[\hat{D}\to\hat{D}]$, the following are equivalent:

(i) $f \sqsubseteq x\to y$;

(ii) $f(x) \sqsubseteq y$;

(iii) $\forall z\ z \sqsubseteq x \Rightarrow f(z) \sqsubseteq y$.

Thus $x\to y$ represents the set of functions which, given an object of type x as the argument return an object of type y as the result.

For example, <u>integer→real</u> represents those functions which yield a a real number when given an integer. It lies (in $[\hat{D}\to\hat{D}]$) above <u>integer→ integer</u> which in turn lies above <u>real→integer</u>.

One of the more important properties of the proposed system is that type properties of least fixedpoints of recursive definitions can be deduced from the values of the least fixed points of appropiately extended definitions.

More precisely, let τ be a term, let b be a sequence of operations over D and let τ_b be the functional mapping $[D\to D]$ into itself defined by using b to interpret the base function symbols (such as "+") occurring in τ. Then the least fixed point f of τ_b can be considered as the 'meaning' of the recursive program $F \Leftarrow \tau[F]$. Now let $\hat{b}$ be formed by monotonically extending the operations of b to operations over $\hat{D}$ (e.g. as addition was extended in the above example) and let $\tau_{\hat{b}}$ be the corresponding functional mapping $[\hat{D}\to\hat{D}]$ into itself, with least fixed point $\hat{f}$.

Then under certain conditions it can be shown that $\hat{f}$ is a monotonic extension of f, and thus information about the behaviour of f on the possibly infinite number of objects of type x can be deduced from the single value $\hat{f}(x)$. For example, if $\hat{f}(\underline{\text{integer}}) = \underline{\text{real}}$, and if i is a genuine integer, then $i \sqsubseteq \underline{\text{integer}}$ and so $\hat{f}(i) \sqsubseteq \underline{\text{real}}$ by monotonicity. But $\hat{f}(i)$ is equal to f(i), and so the latter must be a genuine real number (or $\perp$).

In our system, all type properties of functions are expressed as inequalities (inclusions). In order to show that the least fixedpoint $\mu\tau_b$ of the functional τ_b has certain type properties we show that $\mu\tau_{\hat{b}} \sqsubseteq g$ for a suitable combination g of arrow functions. Such inclusions can can often be handled by Park's fixedpoint method, i.e. by showing that $\tau_{\hat{b}}(g) \sqsubseteq g$. This check can be performed by direct evaluation of $\tau_{\hat{b}}(g)$ and does not require an inductive proof.

Park's method is sufficient for many simple recursive definitions and properties, but it is too 'uniform' for most 'real life' cases. In order to analyse type properties of more complicated recursive definitions, one must proceed by case analysis. For example, in order to show that $\mu\tau_b$ maps integers to strings, it may be necessary to consider separately the values of $\mu\tau_{\hat{b}}$ over natural numbers and over negative integers. Our adaptation of Park's method cannot handle such proofs by case analysis, and we therefore develop a stronger variant of the method for use in such proofs.

An additional advantage of having a single unified domain is that recursive definitions of objects and types are handled in the same way. Some interesting possibilites can be realised by adding parameters to recursive definitions that mix objects and types. Consider, for example, the following type-generating recursive definition:

$$S(x,n) = \underline{\text{if}}\ n \leq 0\ \underline{\text{then}}\ \text{nil}\ \underline{\text{else}}\ x * S(x,n-1)$$

(where * is string concatenation). Then over an appropriate domain, S(<u>boolean</u>,3) represents the type consisting of strings of booleans of length 3, S(U,5) represents the type of strings of arbitrary elements of length 5, and S(0,<u>integer</u>) represents the type of strings of 0's of arbitrary length.

Another interesting possibility is the use of type objects to handle errors and exceptional situations. These new objects are essentially error messages, and they may have an internal structure giving various degrees of information about the nature of the error. For example, the general type error may be above the types divisionerror, overflow, subscripterror, and domainerror. The type subscripterror can itself be above outofrange and notinteger. the last would also be located below the general domainerror, which would be the result of meaninglees combinations such as 3 + tt .

We can also have unions of standard (error-free) types and error messages, which play the role of warning messages. The distinction between errors, warnings and error-free types is demonstrated by the identities:

$$\underline{\text{integer}}/\underline{\text{positive integer}} = \underline{\text{rational}}$$
$$\underline{\text{integer}}/0 = \underline{\text{divisionerror}}$$
$$\underline{\text{integer}}/\underline{\text{integer}} = \underline{\text{rational}} \sqcup \underline{\text{divisionerror}}$$

The last identity is a warning (which could be issued during type checking) that division of an integer by an integer may lead to an error.

We now proceed to the formal development of the system just outlined, together with more detailed examples. Because space is limited some proofs will be omitted and others only outlined.

1. The Construction of $\hat{D}$

By a domain we mean a partially ordered set D such that

(i) D has a least element;

(ii) any directed set of elements of D has a least upper bound.

We will use "$\sqsubseteq_D$", "$\perp_D$", "$\sqcup_D$" and "$\sqcap_D$" to denote the ordering on D, the least element of D, the lub and the glb operations over D, respectively. The subscript D will be omitted when no confusion is likely.

By a data type over D we mean a nonempty subset x of D such that

(i) x is closed downwards, i.e. if d_0 and d_1 are in D, and if $d_0 \sqsubseteq d_1$, then $d_1 \in x$ implies $d_0 \in x$;

(ii) x is closed under lub, i.e. is s is a subset of x and is a directed subset of D, then $\sqcup s \in x$.

Sets with these properties are also called ideals.

Classical examples of types are the sets of all integers, of all reals, of all strings, and so on. In addition, for purposes of type checking, we may consider unusual types such as "integers greater than or equal to 91" or "strings of 0's and 1's only".

Not every set is a type; our methods require (i) and (ii) as above. In particular, $\perp$ must be an element of every type and so the set of defined integers, for example, is not a type.

With every element d of D is associated the type $\tilde{d}$ defined as follows:

$$\tilde{d} = \{ d' : d' \sqsubseteq d \},$$

i.e. $\tilde{d}$ is the set of all elements which approximate d.

The expanded domain $\hat{D}$ is formed by adding (in effect) a collection of types to D. Not any collection will do, however. By a <u>type structure</u> over D we mean a collection T of types over D satisfying the following conditions:

(i) $\tilde{d} \in T$ whenever $d \in D$;

(ii) the universal type U, the set of all elements of D, is in T;

(iii) the set intersection of the types in any nonempty subcollection of T is again in T.

The domain $\hat{D}$ is the set T together with set inclusion as the order.

<u>THEOREM</u> I. For any domain D and any type structure T over D, if $\hat{D} = (T, \subseteq)$ then

(i) $\hat{D}$ is a complete lattice (and therefore a domain);

(ii) $\tilde{\perp}$ is the least element of $\hat{D}$;

(iii) for any d_0, d_1 in D, $d_0 \sqsubseteq_D d_1$ iff $\tilde{d}_0 \sqsubseteq_D \tilde{d}_1$;

(iv) if s is a $\sqsubseteq_D$-directed subset of D and $e = \bigsqcup_D s$, then $\{\tilde{d} : d \in s\}$ is a $\sqsubseteq_{\hat{D}}$-directed subset and $\tilde{e}$ is its $\hat{D}$-lub.

Thus $\hat{D}$ contains an isomorphic copy of D, and so can be considered an extension of D (and we will often treat it as such).

Property (iv) is particularly important when least fixedpoints are discussed because we do not want the structure of lub's in D to be changed in $\hat{D}$. Note, however, that if s is an arbitrary nondirected set with a lub x in D, then the lub in $\hat{D}$ of the corresponding subset of $\hat{D}$ may not be $\tilde{x}$. Note also that T may contain some sets whose sole purpose is to fill in the gaps in the intersection structure of the elements of D.

A given domain may be extended in many ways, One way is to let T be the collection of all possible data types over the domain in question. In most cases this turns out to be a highly undesirable extension, since its structure is so rich as to become unmanageable (for example, it may be extremely difficult to extend a given operation). In practice, smaller extensions are easier to handle than larger ones, and it seems best to restrict added types as far as possible to those actually needed.

2. Function Domains

The system developed in this paper is in a sense "data type free" but not "function type free" since we preserve the separation between function domains of different orders (e.g. between D and $[D\to D]$). For the sake of simplicity we consider only functions of a single argument; the extension to multiargument functions presents no difficulty.

If D and E are any two domains, we define $[D\to E]$ to be the collection of all monotonic functions from D to E, together with the usual (pointwise) ordering. That $[D\to E]$ is a domain is easily verified.

A function $\hat{f}$ in $[\hat{D}\to\hat{D}]$ is an <u>extension</u> of a function f in $[D\to D]$ iff $e = f(d)$ implies $\tilde{e} = f(\tilde{d})$ for any d and e in D. Not every function in $[\hat{D}\to\hat{D}]$ is an extension of one in $[D\to D]$; those that are, we call <u>conservative</u>.

A function g in $[D\to D]$ is said to be <u>tight</u> iff

$$g(x) = \bigsqcup_{d\in x} g(\tilde{d})$$

for any x in $\hat{D}$. In other words, a tight function is determined by its values over D. For example, if g is tight (and $\hat{D}$ is as in the introduction) then $g(\underline{\text{integer}})$ must be the lub of $g(0)$, $g(1)$, $g(-1)$, ... Given a function h in $[\hat{D}\to\hat{D}]$ we define the <u>tightening</u> $\bar{h}$ of h as follows:

$$\bar{h}(x) = \bigsqcup_{d\in x} h(\tilde{d})$$

for any x in D.

THEOREM II. For any h in $[\hat{D}\to\hat{D}]$:

(i) $\bar{h}$ is the least function in $[\hat{D}\to\hat{D}]$ which agrees with h on D;

(ii) $\bar{h}$ is a tight function;

(iii) h is tight iff $h = \bar{h}$.

Given any f in $[D\to D]$, we define the tight extension $\bar{f}$ of f in $[\hat{D}\to\hat{D}]$ as follows:

$$\bar{f}(x) = \bigsqcup_{d \in x} f(\bar{d})$$

for any x in $\hat{D}$. The tight extension of a function is that extension which, in a sense, adds no new possibilities not already inherent in the function itself. If, for example, $f(i) = 0$ for every integer i, then $\bar{f}$(<u>integer</u>) must also be 0.

THEOREM III. For any f in $[D\to D]$:

(i) $\bar{f}$ is a tight function;

(ii) $\bar{f}$ is the least extension of f;

(iii) $\bar{f} = \bar{\hat{f}}$ for any extension $\hat{f}$ (to $[\hat{D}\to\hat{D}]$) of f.

It might seem that nontight functions serve no purpose, and could be eliminated. There are three reasons why this is not the case:

(i) the composition of tight functions may not be tight;

(ii) the tight extension of a given function may be very complex, while at the same time simple and adequate nontight extensions exist;

(iii) the least fixedpoint of $\tau_{\bar{b}}$ may not be tight, even though $\bar{b}$ consists of the tight extensions of the functions of b.

Finally, we should mention why we define $[D\to E]$ to consist of all monotonic functions from D to E, and not just the continuous ones. The reason is that it is possible to find examples of a domain D, a type extension $\hat{D}$ of D and a continuous function f from D to D which can not be extended to a continuous function from $\hat{D}$ to $\hat{D}$. It is possible that restricting the notions of domain and type extension would eliminate this difficulty.

3. Arrow Functions

We now investigate more closely the properties of the arrow operator defined in the introduction. Our first result justifies the claim that $x \to y$ represents those functions which, given an argument of type x, return a result of type y.

THEOREM IV. For any x and y in $\hat{D}$ and any h in $[\hat{D} \to \hat{D}]$, the following are equivalent:

(i) $h \sqsubseteq x \to y$

(ii) $h(x) \sqsubseteq y$

(iii) $h(z) \sqsubseteq y$ whenever $z \sqsubseteq x$

The arrow operation is increasing in y but decreasing in x.

THEOREM V. For any x, x', y and y' in $\hat{D}$:

if $x' \sqsubseteq x$ and $y \sqsubseteq y'$ then $x \to y \sqsubseteq x' \to y'$.

This brings out the crucial difference between the arrow operation and the function domain constructor. The domain $[D \to E]$ is the collection of all monotonic functions from D to E, and the larger are D and E, the larger is $[D \to E]$. But the functions below $x \to y$ are functions which, given something in x, return something in y. Increasing x makes the condition $h \sqsubseteq x \to y$ more restrictive, and decreasing it makes it less restrictive, because $h \sqsubseteq x \to y$ says nothing about what h does to arguments not of type x. Our arrow operation is similar to Yeung's construct (see [3]), but Scott's arrow operation on retracts [4] is really a domain construction operation. Of course, it is only the fact that one type can be a subtype of another that brings about this distinction; in most systems, being of type $x \to y$ is the same as being an element of the domain $[x \to y]$.

Compound function types can be formed by $\sqcap$-ing together (taking glb's of) arrow functions, and these combinations obey certain intuitively plausible rules.

THEOREM VI. For any x, x', y, y' and z in $\hat{D}$,

(i) $x \to y \sqcap y \to z \sqsubseteq x \to z$

(ii) $x \to y \sqcap x' \to y' \sqsubseteq (x \sqcap x') \to (y \sqcap y')$.

Thus $\sqcap$ acts very much as would set intersection in a set based system, such as Yeung's (indeed his has much in common with ours). On the other hand, $\sqcup$, since it works pointwise, is very different from set union; in fact, unions of arrow types give nothing new, because

$$(x\to y) \sqcup (x'\to y') = (x \sqcap x') \to (y \sqcup y')$$

for <u>any</u> x, x', y, y'.

Of special interest are compound types which result from splitting an arrow type into cases. More precisely, a <u>case analysis</u> of the arrow function $x\to y$ is a function of the form

$$(x_0\to y) \sqcap (x_1\to y) \sqcap \dots \sqcap (x_{n-1}\to y)$$

such that the type x is the <u>set union</u> of the types $x_0, x_1, \dots, x_{n-1}$. For example, <u>eveninteger→real</u> ⊓ <u>oddinteger→real</u> is a case analysis of <u>integer→real</u>. Note that these last two functions are not equal; the first when applied to <u>integer</u> gives U, whereas the second gives <u>real</u>. If g is a case analysis of $x\to y$, it can be shown that $x\to y \sqsubseteq g$ and that $x\to y = \bar{g}$.

Our careful distinction between a function and its case analysis may seem pointless to those who think about types in terms of sets. But as it turns out, it is exactly the lack of such distinctions which inhibits proofs by case analysis in simple type checking systems. A special rule which handles the problem will be developed in section 6.

4. Recursion Over the Extended Domain

We now give more details about the relationship between the meanings of recursive programs interpreted over D and over $\hat{D}$. As in the introduction, we assume that τ is a term in some pure recursive programming language (we will not go into details on this point) and that we are interested in the recursive program

$$F = \tau[F]$$

For example, if τ is the term

$$\lambda n\ \underline{\text{if}}\ n<1\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ n\times F[n-1]$$

we are dealing with a program for the factorial function. In this example, the base function symbols are "<u>if</u>-<u>then</u>-<u>else</u>", "<", "×" and "-".

Our most important result is that the least fixed point of $\tau_{\hat{b}}$ (as defined in the introduction) is an extension of the least fixedpoint of τ_b. We prove first the following result to the effect that $\tau_{\hat{b}}$ is, in a sense, an extension of τ_b.

LEMMA I. For any domain D, any type extension $\hat{D}$ of D, any term τ, any sequences b and $\hat{b}$ of operations over D and $\hat{D}$ respectively, and any element g of $[D\to D]$ with extension $\hat{g}$ in $[\hat{D}\to\hat{D}]$:

$$\tau_{\hat{b}}(\hat{g}) \text{ is an extension of } \tau_b(g).$$

PROOF (sketch). Let d be in D. Then in the process of evaluating $\tau_{\hat{b}}(\hat{g})(\tilde{d})$, the extra elements added to D will never arise; thus the evaluation of $\tau_{\hat{b}}(\hat{g})(\tilde{d})$ will be completely analogous to that of $\tau_b(g)(d)$, so that if the result of the latter is e, the result of the former will be $\tilde{e}$.

COROLLARY I. Let τ, b, $\hat{b}$, g and $\hat{g}$ be as above, and let $\bar{b}$ be the sequence of tight extensions of the operations in b. Then

$$\overline{\tau_{\hat{b}}(\hat{g})} = \overline{\tau_b(g)} \sqsubseteq \tau_{\bar{b}}(\bar{g}) \sqsubseteq \tau_{\bar{b}}(\hat{g}) \sqsubseteq \tau_{\hat{b}}(\hat{g}).$$

Note that if f and h are in $[D\to D]$ and $[\hat{D}\to\hat{D}]$ respectively, then h is an extension of f iff $\bar{f} \sqsubseteq h$.

THEOREM VII. Let D, $\hat{D}$, t, b and $\hat{b}$ be as in the previous lemma. Then the least fixedpoint of $\tau_{\hat{b}}$ is an extension of the least fixedpoint of τ_b.

PROOF (sketch). Let $f_0 = \lambda x\, \bot_D$ and for any positive ordinal ξ let $f_\xi = \bigsqcup_{\nu<\xi} \tau_b(f_\nu)$; define the sequence $\hat{f}_\xi$ analogously. Then a simple induction on ξ shows that $\hat{f}_\xi$ is, for each ξ, an extension of f_ξ. Since the two least fixedpoints are the limits of the respective sequences, and since the D-lub and $\hat{D}$-lub structures over D are similar, our result follows. The only complication is that since the functions involved may not be continuous, we must consider infinite as well as finite ordinals.

The following corollary justifies our approach to type checking, as described in the next section.

COROLLARY I. Let D, $\hat{D}$, b and $\hat{b}$ be as before, let x and y be types in $\hat{D}$ and let f and $\hat{f}$ be the least fixed points of τ_b and $\tau_{\hat{b}}$ respectively. If either $\hat{f} \sqsubseteq x\to y$ or $\bar{f} \sqsubseteq x\to y$ then $f(d) \in y$ for any d in x.

There is also a refinement of theorem vii analogous to that of lemma i.

COROLLARY II. Let τ, b, $\hat{b}$ and $\bar{b}$ be as above. Then

$$\overline{\mu\tau_{\hat{b}}} = \overline{\mu\tau_b} \sqsubseteq \mu\tau_{\bar{b}} \sqsubseteq \mu\tau_{\hat{b}}$$

5. Type Checking

In our system, type checking the recursive program $F = \tau[F]$ is reduced to proving an inclusion of the form

$$\overline{\mu\tau_b} \sqsubseteq (x_0 \to y_0) \sqcap (x_1 \to y_1) \sqcap \ldots \sqcap (x_n \to y_n)$$

and, as we saw in the last section, it is sufficient to prove the weaker version in which $\mu\tau_{\hat{b}}$ replaces $\overline{\mu\tau_b}$. For example, suppose that our program is

$$F(n) = \underline{\text{if}}\ n{=}0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ 3{\times}F(n-1)\ ,$$

and that we wish to show that its least fixedpoint maps even integers to even integers and odd to odd. Then we must show

$$\mu\tau_{\hat{b}} \sqsubseteq (\underline{\text{eveninteger}} \to \underline{\text{eveninteger}}) \sqcap (\underline{\text{oddinteger}} \to \underline{\text{oddinteger}}).$$

The important point is that type checking is now just a case of the general and well studied problem of proving assertions about least fixedpoints, and so we have at our disposal several useful methods.

One of the simplest of these is direct evaluation: we evaluate each term $F(x_i)$ (using the standard substitution/simplification method) and try to obtain something below the corresponding y_i. Sometimes this results in a set of $n+1$ mutually dependant equations for $F(x_0)$, $F(x_1)$, ... , $F(x_n)$, which can be solved by computing successive approximations. For example, with the program given above, direct evaluation gives the equations

$$F(\underline{\text{eveninteger}}) = 0 \sqcup 3{\times}F(\underline{\text{oddinteger}})+1$$
$$F(\underline{\text{oddinteger}}) = 3{\times}F(\underline{\text{eveninteger}})+1$$

and the approximations settle down after five steps to the pair of values $F(\underline{\text{eveninteger}}) = \underline{\text{eveninteger}}$, $F(\underline{\text{oddinteger}}) = \underline{\text{oddinteger}}$.

Park's method is also (as was mentioned in the introduction) quite useful. For the program given above, we must show $\tau_{\hat{b}}(g) \sqsubseteq g$ where g is the function $\underline{\text{eveninteger}} \to \underline{\text{eveninteger}} \sqcap \underline{\text{oddinteger}} \to \underline{\text{oddinteger}}$. This is equivalent to showing the two inclusions

$$\tau_{\hat{b}}(\underline{\text{eveninteger}}) \sqsubseteq \underline{\text{eveninteger}}$$
$$\tau_{\hat{b}}(\underline{\text{oddinteger}}) \sqsubseteq \underline{\text{oddinteger}}.$$

These calculations are straightforward, e.g.

$$\begin{aligned}\tau_{\hat{b}}(g)(\underline{\text{eveninteger}}) &= \underline{\text{if}}\ \underline{\text{eveninteger}}{=}0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ 3{\times}g(\underline{\text{eveninteger}}-1)+1\\ &= \underline{\text{if}}\ \underline{\text{boolean}}\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ 3{\times}g(\underline{\text{eveninteger}}-1)+1\\ &= 0 \sqcup 3{\times}g(\underline{\text{eveninteger}}-1)+1\end{aligned}$$

$$= 0 \sqcup 3\times g(\underline{\text{oddinteger}})+1$$
$$= 0 \sqcup 3\times\underline{\text{oddinteger}}+1$$
$$= 0 \sqcup \underline{\text{eveninteger}}$$
$$= \underline{\text{eveninteger}}.$$

We might at this point comment briefly on a peculiar property of the ordering on $\hat{D}$: the (vaguely defined) notion of "amount of information" changes in two opposing directions. If $x \sqsubseteq y$ in $\hat{D}$, then y is a more defined data object than x, but a less precise (larger) type. In particular, the most defined object, U, gives the least type information - none at all.

It might seem, then, that it would be a good idea to change the definition of $\hat{D}$ by placing the types *below* the appropriate data objects (let us call this alternative domain $\check{D}$). This is possible, and the analogs of the theorems we have proved so far are also valid; in particular, the least fixedpoint $\check{f}$ of $\tau_{\check{D}}$ is a monotonic extension of f. But the problem is that in most cases $\check{f}$ gives us no type information at all, because $\check{f}$ applied to any type is simply $\perp$ (this is the case with the example given above). The reason we use $\hat{D}$ instead of $\check{D}$ is that for the purposes of type checking, we want the *least* fixedpoint to draw out the maximum type information from the program - and this is possible because $\sqsubseteq_{\hat{D}}$ orders types by set inclusion.

6. Case Analysis

The type checking methods just discussed fall down when some sort of analysis by cases is required. As a very simple example, consider the following program

$$F(n) = \underline{\text{if}}\ n=0\ \underline{\text{then}}\ n\ \underline{\text{else}}$$

with least fixedpoint f, and suppose that we are trying to show that f applied to any integer is 0. It is futile to try to show that $\hat{f}$ is of type $\underline{\text{integer}}\to 0$ because it is not true: $\hat{f}(\underline{\text{integer}})$ is $\underline{\text{integer}}$ (with D and $\hat{D}$ as in the introduction).

Now suppose that $\hat{D}$ has an extra type $\underline{\text{nonzerointeger}}$ (abbreviated $\underline{\text{nzi}}$) and that $\underline{\text{nzi}}=0$ is ff. Then simple calculation will show that

$$\hat{f} \sqsubseteq (0\to 0) \quad (\text{nzi}\to 0).$$

The right hand side is a case analysis of $\underline{\text{integer}}\to 0$, and so its tightening is $\underline{\text{integer}}\to 0$; thus

$$\bar{f} \sqsubseteq \underline{\text{integer}}\to 0$$

and from this we can conclude that f applied to any integer is 0.

This type of case analysis extends the power of the system, but it usually breaks down for more realistic, genuinely recursive programs. Consider the following program defining a function which flattens binary trees into strings:

$$F(u) = \underline{\text{if}}\ \underline{\text{isatom}}(u)\ \underline{\text{then}}\ u\ \underline{\text{else}}\ F(\underline{\text{left}}(u)) * F(\underline{\text{right}}(u)).$$

Assume that the domain D has binary trees (some of which are atoms) and strings of atoms, that b interprets "*" as string concatenation, and "<u>left</u>", "<u>right</u>" and the predicate "<u>isatom</u>" in the usual way.

We are trying to prove that the least fixedpoint f of this program takes trees into strings. If we construct $\hat{D}$ simply by adding the types <u>tree</u> and <u>string</u> (and define $\hat{b}$ appropriately) we will fail for reasons similar to those in the previous example. But even if we add two more types, <u>atom</u> and <u>nonatom</u>, both below <u>tree</u>, and let

$$g = (\underline{\text{atom}}\rightarrow\underline{\text{string}}) \sqcap (\underline{\text{nonatom}}\rightarrow\underline{\text{string}})$$

we will still not succeed. If we use Park's method, the evaluation of the expression $\tau_{\hat{b}}(g)(\underline{\text{nonatom}})$ gives us $g(\underline{\text{tree}})*g(\underline{\text{tree}})$ and since $g(\underline{\text{tree}}) = U$, the result is U.

Nevertheless, it is still true that $\bar{f} \sqsubseteq g$ and, as we have seen, this is really all we need. Fortunately, there is a variant of the Park fixedpoint induction rule with which we can prove inclusions of the form $\overline{\mu\tau_b} \sqsubseteq g$ without actually determining $\overline{\mu\tau_b}$.

THEOREM VIII. Let D, $\hat{D}$, τ, b and $\hat{b}$ be as in the previous theorems, and let g be any element of $[\hat{D}\rightarrow\hat{D}]$. Then

$$\tau_b(\bar{g}) \sqsubseteq g \text{ implies } \overline{\mu\tau_b} \sqsubseteq g$$

PROOF (sketch). For any ordinal ξ define f_ξ as in theorem vii. We prove by induction on ξ that $\bar{f}_\xi \sqsubseteq g$.

The base step is straight forward. Now let ξ be a positive ordinal and let $\nu<\xi$. By induction we have $\bar{f}_\nu \sqsubseteq g$. Thus $\bar{f}_\nu = \bar{\bar{f}}_\nu \sqsubseteq \bar{g}$ and so $\tau_{\hat{b}}(\bar{f}_\nu) \sqsubseteq \tau_{\hat{b}}(\bar{g}) \sqsubseteq g$. This implies $\overline{\tau_{\hat{b}}(\bar{f}_\nu)} \sqsubseteq g$ and $\tau_b(\bar{f}_\nu) = \overline{\tau_b(f_\nu)}$ by corollary ii of theorem vii. Thus $\bar{f}_{\nu+1} \sqsubseteq g$ and since this is true of every ν less than ξ, we can conclude that $\bar{f}_\xi \sqsubseteq g$. Our result follows easily.

We illustrate the method on our tree flattening example. Since the function g as defined above is a case analysis of $\underline{\text{tree}}\rightarrow\underline{\text{string}}$, we have $\bar{g} = \underline{\text{tree}}\rightarrow\underline{\text{string}}$. Thus we need only show that

$$\tau_{\hat{b}}(\underline{\text{tree}}\rightarrow\underline{\text{string}}) \sqsubseteq (\underline{\text{atom}}\rightarrow\underline{\text{string}}) \quad (\underline{\text{nonatom}}\rightarrow\underline{\text{string}}).$$

This is equivalent to showing that

$$\tau_{\hat{D}}(\underline{\text{tree}\to\text{string}})(\underline{\text{atom}}) \sqsubseteq \underline{\text{string}}$$
$$\tau_{\hat{D}}(\underline{\text{tree}\to\text{string}})(\underline{\text{nonatom}}) \sqsubseteq \underline{\text{string}}$$

and the calculations are straight forward. With this method, as well as with the ordinary Park method, its actual use involves no induction.

As another example, consider the program

$$F(n) = \underline{\text{if}}\ n>100\ \underline{\text{then}}\ n-10\ \underline{\text{else}}\ F(F(n+11))$$

for the well known 91-function. If $\hat{D}$ contains the type <u>integer</u> and also the types <u>ge91</u> (of integers greater than or equal to 91), <u>gr100</u> (of integers greater than 100) and <u>le100</u> (of integers less than or equal to 100) then our method shows that

$$\bar{f} \sqsubseteq \underline{\text{integer}\to\text{ge91}}$$

using the case analysis

$$(\underline{\text{gr100}\to\text{ge91}}) \sqcap (\underline{\text{le100}}\to\text{ge91}).$$

A natural question concerning this method is how to find the appropriate partition of a given type into complementary subtypes. Our experience indicates that the tests of the <u>if-then-else</u>'s are a good guide.

Acknowledgements

We would like to thank Robin Milner and Lockwood Morris for a very useful discussion which provided the original inspiration for this work. The research was financed in part by a grant from the Science Research Council of the United Kingdom.

References

1. R. Milner, L. Morris and M. Newey, "A logic for computable functions with reflexive and polymorphic types", proceedings of the Symposium on Proving and Improving Programs, Arc et Senans 1975, pp371-394.
2. A. Church, "A formulation of the simple theory of types", Journal of Symbolic Logic, v5 (1940), pp56-68.
3. H. K. F. Yeung, "Type checking systems with particular application to functional languages", PhD thesis, Royal Holloway College, University of London, 1976.
4. Dana Scott, "Data types as lattices", SIAM Journal on Computing, v5 (1976), pp522-587.