Towards A Theory Of Type Structure John C Reynolds Syracuse University Syracuse New York 13210 Usa Introduction The Type Structure Of Programming Languages Has Been The Subject Of An Active Development Characterized By Continued Controversy Ove

TOWARDS A THEORY OF TYPE STRUCTURE †

John C. Reynolds

Syracuse University

Syracuse, New York 13210, U.S.A.

Introduction

The type structure of programming languages has been the subject of an active development characterized by continued controversy over basic principles.[1-7] In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.[5] We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular representations used to implement its primitive types. For example, suppose that S and S' are two sets such that the members of S can be used to "represent" the members of S'. We can conceive of running the same program on two machines M and M' in which the same primitive type, say integer, ranges over the sets S and S' respectively. Then if every"integer"input to M represents the corresponding input to M', and if M interprets every primitive operation involving integers in a way which represents the interpretation of M', we expect that every integer output of M should represent the corresponding output of M'. Of course, this idea requires a precise definition of the notion of "represents"; we will supply such a definition after formalizing our illustrative language.

The essential thesis of Reference 5 is that this property of representation independence should hold for user-defined types as well as primitive types. The introduction of a user-defined type t should partition a program into

† Work supported by Rome Air Force Development Center Contract No. 30602-72-C-0281, ARPA Contract No. DAHC04-72-C-0003, and National Science Foundation Grant GJ-41540.

an "outer" region in which t behaves like a primitive type and is manipulated by various primitive operations which are used but not defined, and an "inner" region in which the representation of t is defined in terms of other types, and the primitive operations on t are defined in terms of this representation. We expect that the meaning of such a program will remain unchanged if the inner region is altered by changing the representation of the type and redefining its primitive operations in a consistent manner.

We also wish to consider the old but neglected problem of polymorphic functions, originally posed by Strachey. Consider the construction of a program in which several different types of arrays must be sorted. We can conceive of a "polymorphic sort function" which, for any type t, accepts an array with elements of type t and a binary ordering predicate whose arguments must be of type t, and produces an array with elements of type t. We would like to define such a function, and to have each call of the function syntactically checked to insure that it is type-correct for some t. But in a typed language a separate sort function must be defined for each type, while in a typeless language syntactic checking is lost. We suggest that a solution to this problem is to permit types themselves to be passed as a special kind of parameter, whose usage is restricted in a way which permits the syntactic checking of type correctness.

An Illustrative Language

To illustrate these ideas, we introduce an extension of the typed lambda calculus[8] which permits the binding of type variables. Although this language is hardly an adequate vehicle for programming, it seems to pose the essense of the type structure problem, and it is simple enough to permit a brief but rigorous exposition of its semantics.

We begin with a typed lambda calculus in which the type of every expression can be deduced from the type of its free variables. For this purpose it is sufficient to supply, at each point of variable binding, a type expression describing the variable being bound. For example,

$$\lambda x \in t.\ x$$

denotes the identity function for objects of type t, and

$$\lambda f \varepsilon t \to t.\ \lambda x \varepsilon t.\ f(f(x))$$

denotes the doubling functional for functions over t.

It is evident that the meaning of such expressions depends upon both their free normal variables and their free type variables (e.g., t in the above examples). This suggests the addition of a facility for binding type variables to create functions from types to values, called polymorphic functions. For example,

$$\Lambda t.\ \lambda x \varepsilon t.\ x$$

is the polymorphic identity function, which maps t into the identity function for objects of type t, and

$$\Lambda t.\ \lambda f \varepsilon t \to t.\ \lambda x \varepsilon t.\ f(f(x))$$

is the polymorphic doubling functional, which maps t into the doubling functional for functions over t.

The next step is to permit the application of polymorphic functions to type expressions, and to introduce a new form of beta-reduction for such applications. In general, if r is a normal expression and w is a type expression, then

$$(\Lambda t.\ r)[w]$$

denotes the application of the polymorphic function $\Lambda t.\ r$ to the type w, and is reducible to the expression obtained from r by replacing every free occurrence of t by w (after possible alpha-conversion to avoid collision of variables). For example, the application of the polymorphic identity function to the type integer → real,

$$(\Lambda t.\ \lambda x \varepsilon t.\ x)[\underline{\text{integer}} \to \underline{\text{real}}]$$

reduces to the identity functional for functions from integer to real,

$$\lambda x \varepsilon \underline{\text{integer}} \to \underline{\text{real}}.\ x \quad .$$

Finally, we must introduce a new kind of type expression to describe the types of polymorphic functions. We write $\Delta t.\ w$ to denote the type of polymorphic function which, when applied to the type t, produces a value of type w. Thus if the expression r has the type w, then the expression $\Lambda t.\ r$ has the type $\Delta t.\ w$. For example, the type of the polymorphic identity function is $\Delta t.\ t \to t$, while the type of the polymorphic doubling functional is $\Delta t.\ (t \to t) \to (t \to t)$.

In providing polymorphic functions, we also provide user-defined types. For example, suppose outer is an expression in which cmp is a primitive type (i.e., a free type variable) intended to denote complex numbers, add and magn are primitive functions (i.e., free normal variables) intended to denote addition and magnitude functions for complex numbers, and i is a primitive constant (i.e., a free normal variable) intended to denote the square root of -1. Suppose we wish to represent complex numbers by pairs of reals, and to represent addition, magnitude, and the square root of -1 by the expressions

addrep ε (real × real) × (real × real) → (real × real)
magnrep ε (real × real) → real
irep ε (real × real)

This representation can be specified by the expression

(λcmp. λadd ε cmp × cmp → cmp. λmagn ε cmp → real. λi ε cmp. outer)
[real × real] (addrep) (magnrep) (irep) .

(Our illustrative language does not include the Cartesian product, but its addition should not pose any significant problems.) Admittedly, this is hard to read, but the problem should be amenable to judicious syntactic sugaring.

We now proceed to develop a formal definition of our illustrative language, culminating in a "representation theorem" which asserts its type correctness.

Notational Preliminaries

For sets S and S', we write $S \times S'$ to denote the Cartesian product of S and S', $S \Rightarrow S'$ or S'^{S} to denote the set of functions from S to S', and when S and S' are domains (in the sense of Scott) $S \to S'$ to denote the set of continuous functions from S to S'. If F is a function which maps each member of S into a set, we write $\prod_{x \in S} F(x)$ to denote the set of functions f such that the domain of f is S and, for each $x \in S$, $f(x) \in F(x)$.

For $f \in S \Rightarrow S'$, $x \in S$, $x' \in S'$, we write $[f|x|x']$ to denote the function $\lambda y \in S$. *if* $y = x$ *then* x' *else* $f(y)$.

Syntax

To formalize the syntax of our language, we begin with two disjoint, countably infinite sets: the set T of type variables and the set V of normal variables. Then W, the set of type expressions, is the minimal set satisfying:

(1a) If $t \varepsilon T$ then:

$t \varepsilon W$.

(1b) If $w_1, w_2 \varepsilon W$ then:

$(w_1 \to w_2) \varepsilon W$.

(1c) If $t \varepsilon T$ and $w \varepsilon W$ then:

$(\Delta t.\ w) \varepsilon W$.

(To keep the syntax simple, we have specified complete parenthesization, but in writing particular type expressions we will omit parentheses according to common usage.)

From the fact that $\Delta t.\ w$ is supposed to bind the occurrences of t in w, one can define the notions of free and bound occurrences of type variables, and of alpha-conversion of type expressions in an obvious manner. We write $w \simeq w'$ to indicate that w and w' are alpha-convertible. (In a more complex language, the relation $\simeq$ might be larger; the idea is that it must be a decidable equivalence relation which implies that w and w' have the same meaning.)

One can also define the notion of substitution in an obvious manner. We write $w_1 \left|{}^{w_2}_{t}\right.$ to denote the type expression obtained from w_1 by replacing every free occurrence of t by w_2, after alpha-convering w_1 so that no type variable occurs both bound in w_1 and free in w_2.

To define normal expressions, we must capture the idea that every normal expression has an explicit type. Specifically, an assignment of a type expression to every normal variable which occurs free in a normal expression r must induce an assignment of a type expression to r itself which is unique (to within alpha-conversion). For all $Q \varepsilon V \Rightarrow W$ and $w \varepsilon W$ we write R_{Qw} to denote the set of normal expressions for which the assignment of Q(x) to each normal variable x will induce the assignment of w to the normal expression itself.

Then R_{Qw} is the minimal family of sets satisfying:

(2a) If $Q \in V \Rightarrow W$ and $x \in V$ then:

$$x \in R_{QQ(x)}$$

(2b) If $Q \in V \Rightarrow W$, w_1, w_1', $w_2 \in W$, $w_1 \simeq w_1'$, $r_1 \in R_{Q(w_1 \to w_2)}$, and $r_2 \in R_{Qw_1'}$ then:

$$(r_1\ r_2) \in R_{Qw_2}$$

(2c) If $Q \in V \Rightarrow W$, w_1, $w_2 \in W$, $x \in V$, and $r \in R_{[Q|x|w_1]\ w_2}$ then:

$$(\lambda x \in w_1.\ r) \in R_{Q(w_1 \to w_2)}$$

(2d) If $Q \in V \Rightarrow W$, w_1, $w_2 \in W$, $t \in T$, and $r \in R_{Q(\Delta t.w_1)}$ then:

$$(r[w_2]) \in R_{Q(w_1|^{w_2}_t)}$$

(2e) If $Q \in V \Rightarrow W$, $w \in W$, $t \in T$, $r \in R_{Qw}$, and t does not occur free in $Q(x)$ for any x which occurs free in r, then:

$$(\Lambda t.\ r) \in R_{Q(\Delta t.w)}$$

(Again we have specified complete parenthesization, but will omit parentheses according to common usage.) By structural induction on r, it is easy to show that $r \in R_{Qw}$ and $r \in R_{Qw'}$ implies $w \simeq w'$.

The restriction on t in (2e) reflects the fact that the meaning of t in $\Lambda t.\ r$ is distinct from its meaning in the surrounding context. For example, $Q(x) = t$ does not imply $\Lambda t.x \in R_{Q(\Delta t.t)}$.

Semantics

We will interpret our language in terms of the lattice-theoretic approach of D. Scott.[9-12] Intuitively the effect of a type expression is to produce a Scott domain given an assignment of a domain to each free type variable occurring in the type expression. Thus we expect the meaning of type expressions to be given by a function

$$B \in W \Rightarrow \mathcal{D}^T \Rightarrow \mathcal{D}$$

where $\mathcal{D}$ denotes the class of all domains.

To specify B we consider each of the cases in the syntactic definition of W:

(1a) Obviously,

$$B[t](\bar{D}) = \bar{D}(t)$$

(We will use barred variables to denote functions of T, and square brackets to denote application to syntactic arguments.)

(1b) We intend $w_1 \to w_2$ to denote the domain of continuous functions from the domain denoted by w_1 to the domain denoted by w_2. Thus

$$B[w_1 \to w_2](\bar{D}) = \text{arrow}(B[w_1](\bar{D}),\ B[w_2](\bar{D}))$$

where $\text{arrow} \in (\mathcal{D} \times \mathcal{D}) \Rightarrow \mathcal{D}$ satisfies

$$\text{arrow}(D_1, D_2) = D_1 \to D_2.$$

(1c) We intend $\Delta t.\ w$ to denote a set of functions over the class of domains which, when applied to a domain D will produce some element of the domain denoted by w under the assignment of D to t. Thus

$$B[\Delta t.\ w](\bar{D}) = \text{delta}(\lambda D \in \mathcal{D}.\ B[w][\bar{D} \mid t \mid D])$$

where $\text{delta} \in (\mathcal{D} \Rightarrow \mathcal{D}) \Rightarrow \mathcal{D}$ satisfies

$$\text{delta}(\theta) \subseteq \prod_{D \in \mathcal{D}} \theta(D)$$

We leave open the possibility that delta(θ) may be a proper subset of the above expression. (Indeed, if we are going to avoid the paradoxes of set theory and consider delta(θ) to be a domain, it had better be a very proper subset.)

By structural induction, one can show that $w \simeq w'$ implies $B[w] = B[w']$, and that

$$B[w_1 \mid_t^{w_2}](\bar{D}) = B[w_1][\bar{D} \mid t \mid B[w_2](\bar{D})]$$

The effect of a normal expression is to produce a value, given an assignment of domains to its free type variables and an assignment of values to its free normal variables. (We will call the latter assignment an environment.) However, this effect must conform to the type structure. When given a type assignment $\bar{D}$, a normal expression $r \varepsilon R_{Qw}$ must only accept environments which map each variable x into a member of the domain $B[Q(x)](\bar{D})$, and r must produce a member of the domain $B[w](\bar{D})$. Thus we expect that, for all $Q \varepsilon V \Rightarrow W$ and $w \varepsilon W$, the meaning of the normal expressions in R_{Qw} will be given by a function

$$M_{Qw} \varepsilon R_{Qw} \Rightarrow \prod_{\bar{D} \varepsilon \mathcal{D}^T} (Env_Q(\bar{D}) \to B[w](\bar{D}))$$

where

$$Env_Q(\bar{D}) = \prod_{x \varepsilon V} B[Q(x)](\bar{D}) \ .$$

To specify the M_{Qw} we consider each of the cases in the syntactic definition of R_{Qw}. Essentially the specification is an immediate consequence of the intuitive meaning of the language, guided by the necessity of making the functionalities come out right:

(2a) $M_{QQ(x)}[x](\bar{D})(e) = e(x)$

(2b) $M_{Qw_2}[r_1\ r_2](\bar{D})(e) = (M_{Q(w_1 \to w_2)}[r_1](\bar{D})(e))(M_{Qw_1}[r_2](\bar{D})(e))$

(2c) $M_{Q(w_1 \to w_2)}[\lambda x \varepsilon w_1.\ r](\bar{D})(e)$

$= \lambda a \varepsilon B[w_1](\bar{D}).\ M_{[Q|x|w_1]w_2}[r](\bar{D})[e|x|a]$

(2d) $M_{Q(w_1|_t^{w_2})}[r[w_2]](\bar{D})(e) = (M_{Q(\Delta t.w_1)}[r](\bar{D})(e))(B[w_2](\bar{D}))$

(2e) $M_{Q(\Delta t.w)}[\Lambda t.\ r](\bar{D})(e) = \lambda D \varepsilon \mathcal{D}.\ M_{Qw}[r][\bar{D}|t|D](e)$

Representations

Before we can formulate the representation theorem, we must specify what we mean by representation.

For D, D' ε $\mathfrak{D}$, the set of representations between D and D', written rep(D, D'), is the set of continuous function pairs

$$\text{rep}(D, D') = \{ \langle \phi, \psi \rangle \mid \phi \varepsilon D \to D', \psi \varepsilon D' \to D, \psi \cdot \phi \sqsupseteq I_D, \phi \cdot \psi \sqsubseteq I_{D'} \}$$

where I_D denotes the identity function on D. For $x \varepsilon D$, $x' \varepsilon D'$, and $\rho = \langle \phi, \psi \rangle \varepsilon \text{rep}(D, D')$, we write

$$\rho: x \mapsto x'$$

and say that x represents x' according to ρ if and only if

$$x \sqsubseteq \psi(x')$$

or equivalently,

$$\phi(x) \sqsubseteq x' \quad .$$

A pragmatic justification of this rather ad hoc definition is that it will ultimately make the representation theorem correct. (Although this would still be true if we took rep(D, D') to be the set of projection pairs between D and D', i.e., if we replaced the requirement $\psi \cdot \phi \sqsupseteq I_D$ by $\psi \cdot \phi = I_D$.) However, some intuition is provided by the following connection with the notion of representation between sets. Conventionally, we might say that a representation between a set S and a set S' is simply a function $\pi \varepsilon S \Rightarrow S'$, and that $x \varepsilon S$ represents $x' \varepsilon S'$ according to π iff $\pi(x) = x'$. But if we take D and D' to be the powerset domains 2^S and $2^{S'}$ (with $\subseteq$ as $\sqsubseteq$), and ϕ and ψ to be the pointwise extensions of π and its converse (as a relation), then $\rho = \langle \phi, \psi \rangle$ is a representation between D and D', and $\rho: s \mapsto s'$ iff every $x \varepsilon s$ conventionally represents some $x' \varepsilon s'$ according to π.

The following is an obvious and useful extension of our definition. For $\overline{D}$, $\overline{D}' \varepsilon \mathfrak{D}^T$, we define

$$\text{rep}(\overline{D}, \overline{D}') = \prod_{t \varepsilon T} \text{rep}(\overline{D}(t), \overline{D}'(t)) \quad .$$

The Representation Theorem

At this point, we can formulate a preliminary version of the representation theorem. Consider the set of normal expressions R_{Qw}, and suppose that $\overline{D}$, $\overline{D}' \in \mathcal{D}^{\top}$ and $\overline{\rho} \in rep(\overline{D}, \overline{D}')$, so that for each type variable t, $\overline{\rho}(t)$ is a representation between the domains $\overline{D}(t)$ and $\overline{D}'(t)$. Moreover, suppose that e and e' are environments such that, for each normal variable x, e(x) represents e'(x) according to the relevant representation, i.e., $\overline{\rho}(Q(x))$. Then we expect that the value of any $r \in R_{Qw}$ when evaluated with respect to $\overline{D}$ and e should represent the value of the same normal expression when evaluated with respect to $\overline{D}'$ and e', according to the relevant representation, i.e., $\overline{\rho}(w)$.
More formally:

Let $Q \in V \Rightarrow W$, $w \in W$, $\overline{D}$, $\overline{D}' \in \mathcal{D}^{\top}$, $\overline{\rho} \in rep(\overline{D}, \overline{D}')$, $e \in Env_Q(\overline{D})$, and $e' \in Env_Q(\overline{D}')$. If

$$(\forall x \in V)\ \overline{\rho}(Q(x)): e(x) \mapsto e'(x)$$

then

$$(\forall r \in R_{Qw})\ \overline{\rho}(w): M_{Qw}[r](\overline{D})(e) \mapsto M_{Qw}[r](\overline{D}')(e')$$

However, this formulation has a serious flaw. In choosing $\overline{\rho}$, we assign a representation to every type variable, but not to every type expression, so that the representations $\overline{\rho}(Q(x))$ and $\overline{\rho}(w)$ are not fully defined. Moreover, we can hardly expect to assign an arbitary representation to every type expression. For example, once we have chosen a representation for integer and a representation for real, we would expect that this choice would determine a representation for integer → real and for any other type expression constructed from integer and real.

In brief, we have underestimated the meaning of type expressions. Not only must B[w] map an assignment of domains to type variables into a domain, but it must also map an assignment of representations into a representation. If we can extend the meaning of B to do so, then a correct formulation of the representation theorem is:

Let $Q \in V \Rightarrow W$, $w \in W$, $\overline{D}$, $\overline{D}' \in \mathcal{D}^{\top}$, $\overline{\rho} \in rep(\overline{D}, \overline{D}')$, $e \in Env_Q(\overline{D})$, and $e' \in Env_Q(\overline{D}')$. If

$$(\forall x \in V)\ B[Q(x)](\overline{\rho}): e(x) \mapsto e'(x)$$

then

$$(\forall r \in R_{Qw})\ B[w](\overline{\rho}): M_{Qw}[r](\overline{D})(e) \mapsto M_{Qw}[r](\overline{D}')(e')$$

The Full Semantics of Type Expressions

In order to extend the semantic function B, we first note that the combination of domains and representations forms a category. We write C to denote the category, called the category of types, in which the set of objects is $\mathcal{D}$, the set of morphisms from D to D' is rep(D, D'), composition is given by

$$\langle \phi', \psi' \rangle \cdot \langle \phi, \psi \rangle = \langle \phi' \cdot \phi, \psi \cdot \psi' \rangle$$

and the identity for D is

$$\mathcal{I}_D = \langle I_D, I_D \rangle \quad .$$

From the category of types, we can form two further categories by standard constructions of category theory.[13] We write C^T to denote the category in which the set of objects is $\mathcal{D}^T$, the set of morphisms from $\overline{D}$ to $\overline{D}'$ is $rep(\overline{D}, \overline{D}')$, composition is given by

$$(\overline{\rho}' \cdot \overline{\rho})(t) = \overline{\rho}'(t) \cdot \overline{\rho}(t)$$

and the identity for $\overline{D}$ is given by

$$\mathcal{I}_{\overline{D}}(t) = \mathcal{I}_{\overline{D}(t)} \quad .$$

We write Funct(C, C) to denote the category in which the objects are the functors from C to C and the morphisms from θ to θ' are the natural transformations from θ to θ', i.e., the functions $\eta \in \prod_{D \in \mathcal{D}} rep(\theta(D), \theta'(D))$ such that, for all D, D' $\in \mathcal{D}$ and $\rho \in rep(D, D')$,

$$\theta'(\rho) \cdot \eta(D) = \eta(D') \cdot \theta(\rho)$$

Composition is given by

$$(\eta' \cdot \eta)(D) = \eta'(D) \cdot \eta(D)$$

and the identity for θ is given by

$$\mathcal{I}_\theta(D) = \mathcal{I}_{\theta(D)} \quad .$$

We have seen that the meaning B[w] of a type expression w must map the objects of C^T into the objects of C and the morphisms of C^T into the morphisms of C. Moreover, if our formulation of the representation theorem is to be meaningful, we must have

$$\bar{\rho} \in \mathrm{rep}(\bar{D}, \bar{D}') \quad \text{implies} \quad B[w](\bar{\rho}) \in \mathrm{rep}(B[w](\bar{D}), B[w](\bar{D}'))$$

Thus, at least if we can satisfy the appropriate laws, we expect to extend B[w] from a function from $\mathfrak{D}^T$ to $\mathfrak{D}$ into a functor from C^T to C.

Indeed, by pursuing the analogy of categories with sets and functors with functions, we can induce the main structure of the definition of B[w]. For each of the cases in the syntactic definition of W:

(1a) $B[t](\bar{D}) = \bar{D}(t)$

$B[t](\bar{\rho}) = \bar{\rho}(t)$

(1b) $B[w_1 \to w_2](\bar{D}) = \mathrm{arrow}(B[w_1](\bar{D}), B[w_2](\bar{D}))$

$B[w_1 \to w_2](\bar{\rho}) = \mathrm{arrow}(B[w_1](\bar{\rho}), B[w_2](\bar{\rho}))$

where arrow is a bifunctor from C × C into C.

(1c) $B[\Delta t.\ w] = \mathrm{delta} \cdot \mathrm{abstract}$

where abstract is the functor from C^T into Funct(C, C) such that

$\mathrm{abstract}(\bar{D})(D) = B[w][\bar{D}|t|D]$

$\mathrm{abstract}(\bar{D})(\rho) = B[w][\Phi_{\bar{D}}|t|\rho]$

$\mathrm{abstract}(\bar{\rho})(D) = B[w][\bar{\rho}|t|\Phi_D]$

and delta is a functor from Funct(C, C) into C.

Even before defining the functors arrow and delta, it can be shown that B maps every type expression into a functor from C^T into C, that $w \simeq w'$ implies $B[w] = B[w']$, and that

$$B[w_1|_t^{w_2}](\bar{D}) = B[w_1][\,\bar{D} \mid t \mid B[w_2](\bar{D})\,]$$

$$B[w_1|_t^{w_2}](\bar{\rho}) = B[w_1][\,\bar{\rho} \mid t \mid B[w_2](\bar{\rho})\,] \quad .$$

The Functors arrow and delta

The definition of the functor arrow is fairly obvious. Essentially, its action on representations is to produce the only reasonable composition which matches domains correctly. For all $D_1, D_2 \in \mathcal{D}$,

$$\text{arrow}(D_1, D_2) = D_1 \to D_2 \quad .$$

For all $\langle \phi_1, \psi_1 \rangle \in \text{rep}(D_1, D_1')$ and $\langle \phi_2, \psi_2 \rangle \in \text{rep}(D_2, D_2')$,

$$\text{arrow}(\langle \phi_1, \psi_1 \rangle, \langle \phi_2, \psi_2 \rangle) =$$

$$\langle \lambda f \in D_1 \to D_2 .\ \phi_2 \cdot f \cdot \psi_1,\ \lambda f \in D_1' \to D_2' .\ \psi_2 \cdot f \cdot \phi_1 \rangle$$

(The action of arrow on representations is similar to the method used by Scott to construct retraction or projection pairs for function spaces.)

The definition of arrow and the properties of representations give the following lemma:

Let $f \in D_1 \to D_2$, $f' \in D_1' \to D_2'$, $\rho_1 \in \text{rep}(D_1, D_1')$, and $\rho_2 \in \text{rep}(D_2, D_2')$. Then

$$\text{arrow}(\rho_1, \rho_2): f \mapsto f'$$

if and only if, for all $x \in D_1$ and $x' \in D_1'$,

$$\rho_1: x \mapsto x' \text{ implies } \rho_2: f(x) \mapsto f'(x') \ .$$

which, with the definition of B, gives the following lemma:

Let $w_1, w_2 \in W$, $\bar{\rho} \in \text{rep}(\bar{D}, \bar{D}')$, $f \in B[w_1 \to w_2](\bar{D})$, and $f' \in B[w_1 \to w_2](\bar{D}')$. Then

$$B[w_1 \to w_2](\bar{\rho}): f \mapsto f'$$

if and only if, for all $x \in B[w_1](\bar{D})$ and $x' \in B[w_1](\bar{D}')$,

$$B[w_1](\bar{\rho}): x \mapsto x' \text{ implies } B[w_2](\bar{\rho}): f(x) \mapsto f'(x') \quad .$$

(As an aside, we note that the definition of arrow establishes a connection between our notion of representation and the concept of simulation.[14] Typically, one says that a function $\pi \in S \rightrightarrows S'$ is a simulation of a relation $r \subseteq S \times S$ by a relation $r' \subseteq S' \times S'$ iff $\pi \cdot r \subseteq r' \cdot \pi$ (where $\cdot$ denotes relational composition). But if f, f', ϕ, and ψ are the pointwise extensions of r, r', π, and the converse of π, then $\pi \cdot r \subseteq r' \cdot \pi$ iff $\text{arrow}(\rho, \rho): f \mapsto f'$, where $\rho = \langle \phi, \psi \rangle$.)

The definition of the functor delta is less obvious. For all functors θ from C to C, delta(θ) is the complete lattice with elements

$$\{ f \mid f \in \prod_{D \in \mathcal{D}} \theta(D) \text{ and } (\forall D, D' \in \mathcal{D})(\forall \rho \in rep(D,D'))\theta(\rho): f(D) \mapsto f(D') \}$$

with the partial ordering $f \sqsubseteq g$ iff $(\forall D \in \mathcal{D})\ f(D) \sqsubseteq_{\theta(D)} g(D)$. For all natural transformations η from θ to θ',

$$delta(\eta) = \langle \lambda f \in delta(\theta).\ \lambda D \in \mathcal{D}.\ [\eta(D)]_1(f(D)),\ \lambda f \in delta(\theta').\ \lambda D \in \mathcal{D}.\ [\eta(D)]_2(f(D)) \rangle .$$

At this point, we must admit a serious lacuna in our chain of argument. Although delta(θ) is a complete lattice (with $(\bigsqcup F)(D) = \bigsqcup_{\theta(D)} \{f(D) \mid f \in F\}$), it is not known to be a domain, i.e., the question of whether it is continuous and countably based has not been resolved. Nevertheless there is reasonable hope of evading the set-theoretic paradoxes. Even though $\prod_{D \in \mathcal{D}} \theta(D)$ is immense (since $\mathcal{D}$ is a class), the stringent restrictions on membership in delta(θ) seem to make its size tractable. For example, if $f \in delta(\theta)$, then the value of f(D) determines its value for any domain isomorphic to D.

The definition of delta and the properties of representations give the lemma:

Let η be a natural transformation from θ to θ', $f \in delta(\theta)$ and $f' \in delta(\theta')$. Then

$$delta(\eta): f \mapsto f'$$

if and only if, for all $D, D' \in \mathcal{D}$, and $\rho \in rep(D, D')$,

$$\eta(D') \cdot \theta(\rho): f(D) \mapsto f'(D') .$$

which, with the definition of B, gives:

Let $t \in T$, $w \in W$, $\overline{\rho} \in rep(\overline{D}, \overline{D}')$, $f \in B[\Delta t.\ w](\overline{D})$, and $f' \in B[\Delta t.\ w](\overline{D}')$. Then

$$B[\Delta t.\ w](\overline{\rho}): f \mapsto f'$$

if and only if, for all $D, D' \in \mathcal{D}$, and $\rho \in rep(D, D')$,

$$B[w][\overline{\rho} \mid t \mid \rho]: f(D) \mapsto f'(D') .$$

From the final lemmas obtained about arrow and delta, the representation theorem can be proved by structural induction on r.

Some Syntactic Manipulations

We have explored our illustrative language semantically rather than syntactically, i.e., we have provided it with a mathematical meaning instead of investigating the syntactic consequences of reducibility. However, an obvious question is raised by the fact that every expression in the typed lambda calculus, but not the untyped lambda calculus, has a normal form.(8)

We have been unable to resolve this question for our language. Nevertheless, the language permits some interesting constructions which are not possible in the typed lambda calculus.

For example, consider the following normal expressions:

$$\rho_n \equiv \Lambda t.\ \lambda f \in t \to t.\ \lambda x \in t.\ \underbrace{f(\ \ldots\ f}_{n \text{ times}}(x)\ \ldots\ )$$

of type $\pi \equiv \Delta t.\ (t \to t) \to (t \to t)$,

$$\theta \equiv \lambda h \in \pi.\ \Lambda t.\ \lambda f \in t \to t.\ \lambda x \in t.\ f(h[t]\ f\ x)$$

of type $\pi \to \pi$ (We assume application is left-associative.),

$$\alpha \equiv \lambda g \in \pi \to \pi.\ \lambda h \in \pi.\ g(h[\pi]\ g\ \rho_1)$$

of type $(\pi \to \pi) \to (\pi \to \pi)$, and

$$\beta \equiv \lambda w \in \pi.\ w[\pi \to \pi]\ \alpha\ \theta$$

of type $\pi \to (\pi \to \pi)$. Then the following expressions are interconvertible:

$$\theta\ \rho_n \simeq \rho_{n+1}$$
$$\beta\ \rho_{m+1} \simeq \alpha\ (\beta\ \rho_m)$$
$$\beta\ \rho_0\ \rho_n \simeq \rho_{n+1}$$
$$\beta\ \rho_{m+1}\ \rho_0 \simeq \beta\ \rho_m\ \rho_1$$
$$\beta\ \rho_{m+1}\ \rho_{n+1} \simeq \beta\ \rho_m\ (\beta\ \rho_{m+1}\ \rho_n) \quad .$$

From the last three equations it follows that $\beta\ \rho_m\ \rho_n \simeq \rho_{\psi(n,m)}$, where $\psi(n,m)$ is Ackermann's function.

Further Remarks

Since the writing of the preliminary version of this paper, considerable attention has been given to the "serious lacuna" mentioned above. We have managed to show that delta(θ) is a continuous lattice, but not that it is countably based. Conceivably, our notion of representation is too restrictive, which would tend to make delta(θ) unnecessarily large.

ACKNOWLEDGEMENT

The author would like to thank Dr. Lockwood Morris for numerous helpful suggestions and considerable encouragement.

REFERENCES

1. Van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., and Koster, C. H. A., Report on the Algorithmic Language ALGOL 68. MR 101 Mathematisch Centrum, Amsterdam, October 1969. Also Numerische Mathematik 14 (1969) 79-218.

2. Cheatham, T. E., Jr., Fischer, A., and Jorrand, P., On the Basis for ELF-An Extensible Language Facility. Proc. AFIPS 1968 Fall Joint Comput. Conf., Vol. 33 Pt. 2, MDI Publications, Wayne, Pa., pp. 937-948.

3. Reynolds, J. C., A Set-theoretic Approach to the Concept of Type. Working paper, NATO Conf. on Techniques in Software Engineering, Rome, October 1969.

4. Morris, J. H., "Protection in Programming Languages," Comm. ACM, 16 (1), January 1973.

5. Morris, J. H., Types are not Sets. Proc. ACM Symposium on Principle of Programming Languages, Boston 1973, pp. 120-124.

6. Fischer, A. E., and Fischer, M. J., Mode Modules as Representations of Domains. Proc. ACM Symposium on Principles of Programming Languages, Boston 1973, pp. 139-143.

7. Liskov, B., and Zilles, S., An Approach to Abstraction. Computation Structures Group Memo 88, Project MAC, MIT, September 1973.

8. Morris, J. H., Lambda-calculus Models of Programming Languages. MAC-TR-57, Project MAC, MIT, Cambridge, Mass., December 1968.

9. Scott, D., "Outline of a Mathematical Theory of Computation," Proc. Fourth Annual Princeton Conf. on Information Sciences and Systems (1970), pp. 169-176. Also, Tech. Monograph PRG-2, Programming Research Group, Oxford University Computing Laboratory, November 1970.

10. _____. "Continuous Lattices," Proc. 1971 Dalhousie Conf., Springer Lecture Note Series, Springer-Verlag, Heidelberg. Also, Tech. Monograph PRG-7, Programming Research Group, Oxford University Computing Laboratory, August 1971.

11. _____. "Mathematical Concepts in Programming Language Semantics," AFIPS Conference Proc., Vol. 40, AFIPS Press, Montvale, New Jersey (1972), pp. 225-234.

12. _____. "Data Types as Lattices", Notes, Amsterdam, June 1972.

13. MacLane, S., Categories for the Working Mathematician, Springer-Verlag, New York 1971.

14. Morris, F. L., Correctness of Translations of Programming Languages -- An Algebraic Approach, Stanford Computer Science Department Report STAN-CS-72-303, August 1972.