On the Semantics of "Data Type"

by

James Donahue

TR 77-311

Computer Science Department
Cornell University
Ithaca, N.Y. 14853

On the Semantics of "Data Type"

James Donahue[+]
Computer Science Department
Cornell University
Ithaca, N.Y.

Abstract

This paper considers the general problem of specifying the meaning of programming languages which include "data type definition facilities." The fundamental question posed in attempting to define such languages is: "what meaning should be given to a data type definition," or more simply, "what does data type mean?" In this paper we describe a new approach to defining the meaning of data types and give its application to the definition of a typed lambda calculus extension. We also prove a theorem stating that our language is "strongly typed."

[+]This work was partially supported by National Science Foundation grant MCS76-14293.

1. Introduction

I hate definitions.

— Benjamin Disraeli

A recent trend in programming languages is to allow definitions of data types in programs. Although there is currently much ferment about what form such definitions should take (cf., PASCAL [Jensen and Wirth 1975], Euclid [Lampson et al 1977], and CLU [Schaffert et al 1975]), there seems general agreement about their utility as a programming tool.

The introduction of type definitions requires that we have a clear idea of what we mean by "data type," so that we can know just what is being defined. A clear notion of "data type" is also important for understanding languages without type definition facilities. In attempting to give the semantics (using any kind of formalism) of a language with either a fixed or changeable set of data types, we somehow must give some meaning to the type identifiers and expressions appearing in programs.

This paper presents an approach to defining the meaning of "data type" in programming languages. We do not give the semantics of a language involving type definitions; instead we focus on the interpretation of programs in which data type definitions are used. How specific data type values become associated with type identifiers is unspecified. Indeed, given the lack of agreement as to how this should be done, we are well-advised not to choose any single form of type definition. Hopefully, the reader will see how his favorite form of definition can be viewed using this approach.

The paper is organized as follows. In the next section we describe in informal terms our interpretation of the meaning of "data type," using examples from PASCAL. We then give a formal definition of a far simpler language, a version of the typed lambda calculus, to show how the approach allows a rigorous formulation of the meaning of data types. Finally, we prove a theorem showing that the language we defined is "strongly typed." In doing so, we provide a semantic justification for the common use of "type-checking" to catch erroneous programs.

2. An informal view of data types

Our main interest is the formal semantics of data types; however, before leaping into the morass of notation to follow, we begin with an informal description and justification of our view of data types. We first consider more common interpretations of data type and then show how our approach improves on these interpretations.

The most common view of data types is that they are simply subsets of the "value space" of a programming language. Thus, the meaning of a data type definition is simply to define a new named set of values in the value space. In languages like PASCAL, where variables have immutable type, this notion fits well with our natural view of stack-oriented storage allocation. Thus, "a data type defines the set of values a variable may assume." [Jensen and Wirth 1975, pg. 12]

Recently, work on "encapsulated data type" mechanisms has extended this view of data types to include not only a set of objects (values), but also the behavior of this set under certain operations, e.g., [Morris 1973, Liskov and Zilles 1974, Guttag 1975]. One advantage of this approach is that this behavior can be specified "abstractly," i.e., independently of a particular representation for the set of objects. This abstract specification then becomes the correctness criterion for any particular data type definition (which includes some description of a set of objects) that is said to "implement" the "abstract type." Note however that at the level of the programming language, data types still incorporate some notion of a set of objects; it is simply in the correctness criterion that this notion has been (correctly) removed.

As an informal basis for understanding data types, this set-theoretic approach has the advantages of familarity (sets being common objects to mathematicians) and simplicity. However, when we attempt to use this informal understanding as a basis for specifying a formal semantics, we are faced with some serious problems:

1. What set is defined by a data type definition?

   If any one of several sets may suffice as the meaning of the definition, how are we to choose a particular one as <u>the</u> meaning?

2. Some languages (e.g., CLU) allow types to be parameters to other type or procedure definitions. If types are sets, what meaning is to be given to the appearance of a type identifier as the argument to a procedure?

We discuss both of these questions further before suggesting an alternative approach to data type.

The most obvious difficulty in viewing data types as sets is deciding what set is defined by a data type definition. For example, consider the PASCAL record definition

<u>record</u> re: real; im: real <u>end</u>

The temptation is to view this definition as specifying a Cartesian product of reals; but there are certainly many other suitable representations (e.g., encoding both components in a single value, making the value a function, or simply adding more (unused) components to the Cartesian product). While some representations may be more natural, the formal semantics should somehow reflect the fact that many representations are possible.

The second difficulty in treating types as sets is in the handling of the semantics of languages which allow types as parameters of procedures or type definitions (e.g., CLU). The problem here can be simply stated as follows. If the set of all values which can be passed to procedures (i.e., the allowable types of procedure parameters) includes the set of all sets of allowable values which can be passed to procedures (i.e., the set of all data types), is there any hope of avoiding the paradoxes of set theory? In [1974], Reynolds suggests that there is hope; however, the mathematics of the attempt are daunting.

Our approach to the semantics of data types is based on the observation made above about the specification of "abstract data types"; if we specify the meaning of data types as the meaning of certain operations, we are able to leave unspecified the choice of a particular representation. However, unlike the work on "abstract data types," we do not use programmer-defined operations as the meaning of data types. Instead, we use as the meaning of a data type the interpretation it gives to the basic operations of the language. An example from PASCAL again will help to explain the idea of this approach.

If we consider the PASCAL code fragment

```
var x: T;
var y,z: T';
x:= red;
y.a:= z.b
```

we can see several places where the effect of the program is dependent on the meanings of the data types T and T'. These include:

1. the semantic interpretation of the syntactic constant red (by T, if the program is type-correct),
2. the meaning of x on the left-hand side of the first

assignment statement and the meaning of the assignment. E.g., consider the difference between the effects of this program if preceded by

<u>type</u> T = (red, green, blue)

or

<u>type</u> C = (red, green, blue);
<u>type</u> T = (green .. blue)

In the first case, the program is valid, while in the second case either it is syntactically erroneous or execution of the assignment will produce a semantic error, depending on the view of the implementor (the PASCAL report leaves this point unspecified, stating only that the type of the variable in an assignment may be a subrange of the type of the expression).

3. the meaning of the component selections .a and .b used to access fields of y and z. If we were to change the definition of T' from

<u>type</u> T' = <u>record</u> a,b: real <u>end</u>

to

<u>type</u> T' = <u>array</u> 1..10 <u>of</u> <u>integer</u>,

the program becomes illegal.

Since it is clear that the meanings of these primitive operations are dependent on the types associated with the type identifiers T and T', we adopt the approach that the meanings of T and T' must be the interpretations they provide for these operations.

A similar, but less formal, treatment of data types can be found in the axiomatic definition of PASCAL [Hoare and Wirth 1973]. In the rule of assignment,

$$\{P_x^y\}\ x{:=}y\ \{P\},$$

if the type T of x is a subrange of the type of y, then P_x^y is replaced by $P_x^{T(y)}$ (where T(y) is the restriction of y to a value of type T). In this case then the meaning of the assignment x:=y is explicitly stated to be dependent on the type of x. All we are doing is formalizing this by saying that the meaning of a data type is the semantics it gives to assignment (i.e., the axiom $\{P_x^{T(y)}\}$ x:=y {P}).

In the next section, we give a denotational semantics for an extension of the typed lambda calculus using this approach. The simplicity of the language allows us to keep the semantic description short; however, the language allows us to show that this approach to data types successfully avoids the two problems of the "types as sets" approach described above:

1. Our language allows <u>data types</u> as parameters in lambda expressions. Our semantics handles this added complication in a completely straightforward fashion similar to the treatment of normal lambda abstraction.
2. Using our semantics, we state and prove theorem in section 4 stating that the choice of sets to represent elements of the data types in the language is purely arbitrary. Thus, our semantics can be said to be "representation independent." As a by-product of this theorem, we provide a semantic justification for syntactic "type-checking."

3. A lambda calculus extension and its semantics

3.1 The language

The language we use as an example of this approach to giving meaning to data types is an extension of the typed lambda-calculus first suggested by Reynolds [1974]. As in the basic typed lambda-calculus [Morris 1968], it includes the usual forms of abstraction and application. For example, $\lambda x \epsilon t.x$ defines the identity function of type t ($\lambda x \epsilon t.x$ has type $t \rightarrow t$), and if y has type t, $(\lambda x \epsilon t.x\ y)$ denotes the application of the identity function to the argument y (yielding y). Additionally, we extend the language in two ways:

1. If T is a type, then T$c will be used to denote the constant c of type T. The syntactic form of constants will be left unspecified to shorten the presentation.
2. Just as we allow normal abstraction and application, so we allow type abstraction and application. For example, given the type t identity function $\lambda x \epsilon t.x$, we can abstract it with respect to the type identifier t by writing $\Lambda t.\lambda x \epsilon t.x$ to produce the polymorphic ("of many types") identity function (of type $\Delta t.t \rightarrow t$). From $\Lambda t.\lambda x \epsilon t.x$ we can produce the identity function of any particular type by applying our polymorphic function to the type. Thus,

   $$\Lambda t.\lambda x \epsilon t.x\ [\text{integer}]$$

   produces the integer identity function as its result.

Below, we give a complete formal description of the language using the denotational approach to semantics.

3.2 Formal semantics

We begin by giving a formal syntax for the language (using identifiers beginning with a capital letter to denote nonterminals). Exp, the set of lambda expressions, is defined as follows:

Exp::= Id$Const	constants of type Id (Const will be left unspecified)
| Id	normal variables
| λIdϵW.Exp	normal lambda abstract: (where W is a type expression, as defined below)
| (Exp Exp)	normal application
| Λ Id.Exp	type abstraction
| Exp[W]	type application

W, the set of type expressions, is defined by:

W::= Id	type identifiers
| W→W	functional types (the types of λ terms)
| ΔId.W	polymorphic types (the types of Λ terms)

We give formal semantics to the language by defining a function <u>Me</u> mapping each lambda expression to its meaning as an element of some domain. As in treatments of other forms of the lambda calculus, we will take the meaning of expressions as elements of the domain

$$D = B+[D\to D]+[T\to D]$$

where B is a domain of primitive or basic values (which we assume will be used to interpret constants) and T is the domain of type denotations. (One can think of $D \to D$ values as the meanings of λ-terms, $T \to D$ values as the meanings of Λ-terms.) Obviously, the heart of the matter is the definition of T, and it is here that we apply our previously stated view of data types.

The most basic operation in the lambda calculus is clearly the evaluation of the basic expressions, i.e., identifiers and constants. For the untyped lambda calculus, the usual interpretation of the meaning of a variable x is its value in the current environment, a member of the domain $Id \to D$. However, in the typed calculus, the value produced by evaluating x in the current environment must be consistent with the type associated with x. One approach to satisfying this requirement in the semantics is to say that the environment must be "type-respecting," i.e., that the value associated with x in the environment must be compatible with the type of x. (This approach is adopted in [Reynolds 1974].)

Our approach is to take as the meanings of data types the interpretations of "taking the value of an identifier in an environment." Thus, we take T to be functions of type $D \to D$ such that for any environment e, if x has type t, then the "meaning of t" applied to e(x) produces a "type-respecting" value.

For the basic data types in the language, the meaning of the data type will also include a component of type Const→B giving the meaning of constants of that type. (Since we have included no data type definition facility in the language, how constants are given denotations is left unspecified.)

This interpretation of data types can be informally explained in terms of the operation of an SECD machine [Wegner 196{
evaluation of lambda expressions. In an SECD definition of the la
calculus, the meaning of "taking the value of a variable" is to pu
its current value in the environment onto the stack. Now consider the implementation of the stack on some machine of fixed word length. The action of the machine will involve:

1. allocating a number of words (which may differ for various types) of storage on the top of S to hold the value to be copied, and
2. filling this space with the current value of the variable in E.

Our use of a function of type D→D as the meaning of "data type" can be viewed as simply the denotational encoding of this action of an SECD machine.

This interpretation of the meaning of data types points out one further aspect of our semantics worthy of note. It is natural at first glance to view the D→D component of types that introduce constants (i.e., types in [Const→B]x[D→D]) as simply restrictions of the identity function. For example, if we decide to define the Boolean constants <u>true</u> and <u>false</u> by the integers 1 and 0 respectively, a natural choice for the second component of the meaning of Booleans would be

$\lambda x\epsilon D.$ x <u>is</u> Integer <u>then</u> (x>1 <u>then</u> $\perp$ <u>else</u> x) <u>else</u> $\perp$

This choice can be viewed as choosing a single-bit representation of Booleans and checking to make sure one never moves nonprimitive values (functions, for example) onto the stack. When viewed in this light other possibilities suggest themselves. For example,

$\lambda x\epsilon D.$ x <u>is</u> Integer <u>then</u> (x>255 <u>then</u> $\perp$ <u>else</u> x) <u>else</u> 255

suggests the choice of eight bits to represent Booleans and less care in ruling out the assignment of non-primitive values to Boolean variables. In fact, all we really require is that the interpretations of constants (given by the first component of a primitive type) remain unaltered by the application of the D→D (second) component of the data type. In more formal terms, we require that these functions from D to D be "retracts," i.e., functions f such that $f=f\circ f$, with range including at least the denotations of all constants introduced by the type. And clearly this restriction makes sense in terms of the SECD analogy.

Now, given this basic decision about the meaning of data types, we are now ready to "lay our domains on the table" to specifying the complete semantic structure of the language. Informal commentary follows the formal description.

Basic Domains

Exp	as above
W	as above
$D = B+[D\to D]+[T\to D]$	expression denotations
$T = [D\to D]+[[Const\to B]\times[D\to D]]$	data types

Meaning functions

Me: $Exp\to Q\to Te\to Env\to D$	meaning of expressions
where	
$Q = Id\to W$	syntactic type environme
$Te = Id\to T$	semantic meaning of type identifiers
$Env = Id\to D$	meaning of free identifi
Mt: $W\to Te\to T$	meaning of types

The meaning of an expression (the particular element in $D = B+[D\to D]+[T\to D]$ it denotes) depends in several ways on the free identifiers appearing in the expression; the domains Q, Te, and Env are used to capture these various dependencies. Q is used to provide the "syntactic type" (element of W) associated with each free identifier. For example, inside the body of $\lambda x\epsilon t.exp$, x "has type t", i.e., the element of Q used to interpret exp will associate t with x. The "type environment" Te provides the mechanism for giving a meaning (as elements of T) to these syntactic types by providing the element of T associated with each free type identifier. Finally, the domain Env associates a value in D with each free normal identifier in the expression. Before giving the definitions of Me and Mt, we first must define an auxiliary function Typeof which produces the "syntactic type" of every expression:

Typeof: $Exp \to Q \to W$

is defined by

$TypeOf[\![id\$const]\!](q) = id$
The type of a constant is determined by the constant

$TypeOf[\![id]\!](q) = q[\![id]\!]$
The type of an identifier is found in q.

$TypeOf[\![(exp_1\ exp_2)]\!](q) = Range(TypeOf[\![exp_1]\!](q))$
where

$Range: W \to W$ and
$Dom: W \to W$

extract the range and domain of type expressions of the form $W_1 \to W_2$, i.e.,

$Range[\![W_1 \to W_2]\!] = W_2$ and

$Dom[\![W_1 \to W_2]\!] = W_1$

Note that in specifying the <u>syntactic</u> type of a normal application, all we demand is that the type of the left-hand expression be some functional type; we do not require argument/parameter type correspondence. Expressions where such mismatches occur will have a proper (i.e., defined) type, but improper semantics.

$TypeOf[\![\lambda id \epsilon w.exp]\!](q) = w \to TypeOf[\![exp]\!](q[id \leftarrow w])$

where $q[id \leftarrow w]$ is $\lambda id'.id'=id$ <u>then</u> w <u>else</u> $q[\![id]\!]$. (This notation for "updating" a <u>function</u> <u>will be</u> used extensively in what follows.)

normal abstractions are of functional data types

$TypeOf[\![\Lambda id.exp]\!](q) = \Delta id.TypeOf[\![exp]\!](Replace(q,id,\iota))$
where

$Replace(Q,id,w) = \lambda id'.Q[\![id']\!]|^{w}_{id}$

i.e., all occurrences of the type identifier id in Q are replaced by w. This use of Replace is necessary to reflect the fact that previous bindings of identifiers to type expressions involving t are no longer valid inside the body of $\Lambda t.exp$.

$TypeOf[\![exp[w]]\!](q) = Body(TypeOf[\![exp]\!](q))|^{w}_{BV(TypeOf[\![exp]\!](q))}$

where Body: $W \to W$ and BV: $W \to Id$ extract the body and bound variable of type expressions of the form $\Delta id.w$.

One aspect of our treatment of syntactic types, as specified by TypeOf, is worthy of further comment. That is that expression $\lambda f \epsilon t \rightarrow t . \lambda x \epsilon t .(f\ x)$ always has syntactic type $(t \rightarrow t) \rightarrow (t \rightarrow t)$ independent of surrounding context. Thus, for example, the expression

$$\Lambda t . \lambda f \epsilon t . \lambda x \epsilon t' .(f\ x)\ [t' \rightarrow t'']$$

has improper type because f is not of some functional type, even though normal beta-reduction would reduce this to

$$\lambda f \epsilon t' \rightarrow t'' . \lambda x \epsilon t' .(f\ x)$$

which has a proper type. In implementation-oriented terms, this restriction imposed by TypeOf can be viewed as allowing us to "compile" a polymorphic procedure by being able to type-check its body independent of any type arguments to which the polymorphic procedure may be applied.

Note that the desired effect of the above expression could be achieved by the type-correct expression

$$\Lambda t_1 . \Lambda t_2 . \lambda f \epsilon t_1 \rightarrow t_2 . \lambda x \epsilon t_1 .(f\ x)\ [t'][t''],$$

which can be shown to be semantically equivalent to

$$\lambda f \epsilon t' \rightarrow t'' . \lambda x \epsilon t' .(f\ x)$$

We are finally ready to give the clauses of the definitions of our meaning functions Me and Mt:

$\underline{Me}[\![id\$const]\!](q)(te)(e) = te[\![id]\!]\ \underline{is}\ [Const \rightarrow B] \times [D \rightarrow D]\ \underline{then}\ [te[\![id]\!]]_1 [\![const]\!]\ \underline{else}\ \top$

If id is a type for which constants are defined, produce as value the meaning the constant (i.e., the value of the first component of the meaning of the type applied to the constant); otherwise, the expression is erroneous.

$\underline{Me}[\![id]\!](q)(te)(e) = Apply(\underline{Mt}[\![TypeOf[\![id]\!](q)]\!](te), e[\![id]\!])$

where
Apply: $[T \times D] \to D$
is defined by

$$Apply(t,d) = t\ \underline{is}\ [Const \to B] \times [D \to D]\ \underline{then}\ [t]_2(d)\ \underline{else}\ t(d)$$

The meaning of an identifier is simply its value in the current environment applied to the meaning of the type of the identifier.

$\underline{Me}[\![(exp_1\ exp_2)]\!](q)(te)(e) =$

$$TypeOf[\![exp_2]\!](q) \cong Dom(TypeOf[\![exp_1]\!](q))\ \underline{then}\ (\underline{Me}[\![exp_1]\!](q)(te)(e)|D \to D)\ (\underline{Me}[\![exp_2]\!](q)(te)(e))\ \underline{else}\ \top$$

where $w_1 \cong w_2$ is true iff w_1 is alpha-convertible to w_2

and $x|D \to D$ restrict $x \in D$ to elements of $D \to D$, i.e., $x|D \to D$ is

$\bot$ if $x = \bot$
$\top$ if $x = \top$ or $x \notin D \to D$
x if $x \in D \to D$

The meaning of an application is to produce a proper result only when the type of the argument is consistent with the type of the parameter.

$\underline{Me}[\![\lambda id \in w.exp]\!](q)(te)(e) =$
$\lambda y \in D.\ \underline{Me}[\![exp]\!](q[id \leftarrow w])(te)(e[id \leftarrow y])$

$\underline{Me}[\![\Lambda id.exp]\!](q)(te)(e) =$
$\lambda t \in T.\ \underline{Me}[\![exp]\!](Replace(q,id,\bot))(te[id \leftarrow t])(e)$

$\underline{Me}[\![exp[w]]\!](q)(te)(e) =$
$(\underline{Me}[\![exp]\!](q)(te)(e)|T \to D)\ (\underline{Mt}[\![w]\!](te))$

The meaning of type application is simply to apply the meaning of the expression (as an element of $T \to D$) to the meaning of the type (an element of T).

The meaning of data type expressions is given by:

$\underline{Mt}[\![id]\!](te) = te[\![id]\!]$

$\underline{Mt}[\![w_1 \to w_2]\!](te) = Arrow(\underline{Mt}[\![w_1]\!](te), \underline{Mt}[\![w_2]\!](te))$
where
Arrow: $[T \times T] \to T$ is defined by

$$Arrow(t_1,t_2) = \lambda f \in D.(\lambda x \in D.Apply(t_2,x)) \circ f|D \to D \circ (\lambda x \in D.Apply(t_1,x))$$

where $f_2 \circ f_1$ is normal function composition, i.e.,
$(f_2 \circ f_1)(x) = f_2(f_1(x))$

The meaning of functional types is to map their arguments to "type respecting" functions, i.e., functions that accept values of type w_1 and produce results of type w_2.

$\underline{Mt}[\![\Delta id.w]\!](te) = Delta(\lambda t \in T.\underline{Mt}[\![w]\!](te[id \leftarrow t]))$
where
Delta: $[T \to T] \to T$ is defined by
$Delta(y) = \lambda x \in D.\lambda t \in T.Apply(y(t), (x|T \to D)(t))$

The meaning of polymorphic types is to take polymorphic values to "type-respecting" polymorphic values.

By this point, the reader who has stuck with us so far may be tempted to say, "So what?" Two points of interests in the semantics above should be noticed:

1. Our treatment of data types abstraction and application is a straightforward extension of the treatment of normal abstraction and application in the lambda calculus. In particular, we avoid the "serious lacuna" of Reynolds's semantics for this same language, where the meaning given to polymorphic types may not be elements of a domain (unlike our choice of T). Moreover, his earlier semantics requires the use of category theory to give the meaning of functional and polymorphic types, a mathematical complication we avoid.

2. When read operationally, these meaning functions are consistent with common implementations of statically typed programming languages (ignoring the extra complexities caused by polymorphism). We have already described our interpretation of data types in terms of an SECD machine. The other point to note in this regard is the treatment of parameter-passing and type-checking in the semantics. The meaning of applications involves a simple <u>syntactic</u> check that the type of the argument is consistent with the type of the parameter. This sort of type-checking has been described in many other placed [Morris 1968, Ledgard 1972, Gannon and Horning 1975] and is readily contrasted with the checking of the types of values found in "dynamically typed" languages like Gedanken [Reynolds 1970, Tennent 1976]. Also the interpretation of λ-abstraction involves no test of the "acceptability" of the parameter in the body of the abstraction. This "blind faith" in the validity of an argument is precisely the reason that "compile time" type checking is performed in statically typed languages.

In our introduction, we stated that the semantics of a language with data type definitions should reflect the fact that the choice of representation of elements of the data type was chosen arbitrarily. In the next section, we prove a theorem asserting that the semantics we described above satisfies this requirement.

3.3 A "strong typing" theorem

In the preceding discussion, we stated that by focusing on the operations provided by a data type we could ignore the details of representation. The curious reader might well ask whether this claim holds for the semantics we have given above. In this section we prove that our semantics has the degree of "representation independence" we desire.
It is clear that the meaning given to expressions by our semantics is dependent on the representations chosen for the basic data types in the language. For example, the meaning of the application

$$(\lambda x \epsilon \text{Boolean}.x \;\; \text{Boolean\$true})$$

will change if we change the representation of Boolean constants from integers to two-argument functions (e.g., $\lambda x \epsilon D.\lambda y \epsilon D.x$ for true and $\lambda x \epsilon D.\lambda y \epsilon D.y$ for false). This obvious dependency on representation is unimportant, however; it is like saying that the integer constant 1 will likely have different meanings on machines with 16- or 32-bit words. Of much greater concern is the following sort of representation dependence.

Let us assume that for some expression exp in our language, we have that:

1. TypeOf⟦ exp ⟧(q) = t for some q ϵ Q and
2. when t is bound to some basic type (i.e., a type for which constants are defined), we have that Me⟦ exp ⟧(q)(te)(e) = Me⟦ t$const ⟧(q)(te)(e) for some const ϵ Const.

Our belief in the notion of representation independence suggests that this semantic equivalence of <u>exp</u> and t$const should be independent of any choices made about the representations of <u>any</u> data type in the language, including t itself.

It is important to note that the common practice of "type-checking" can be entirely explicated in terms of our desire to guarantee representation independence. For example, consider the lambda expression

$$(\lambda x \in \text{Integer}.x \ \ \text{Boolean\$true}).$$

As described above, the type of this expression is Integer, while its meaning is $\top$ ("overdefined") because of the type mismatch of argument and parameter. However, consider what meaning this application could have were we not to do the type-checking. Then from the definition of <u>Me</u>, we would have

$$\begin{aligned}&\underline{Me}\llbracket(\lambda x \in \text{Integer}.x \ \ \text{Boolean\$true})\rrbracket(q)(te)(e) = \\ &\quad(\underline{Me}\llbracket\lambda x \in \text{Integer}.x\rrbracket(q)(te)(e)\,|\,D\to D)(\underline{Me}\llbracket\text{Boolean\$true}\rrbracket(q)(te)(e)) \\ &(\lambda y \in D.\underline{Me}\llbracket x\rrbracket(q[x\leftarrow\text{Integer}](te)(e[x\leftarrow y]))(\underline{Me}\llbracket\text{Boolean\$true}\rrbracket(q)(te)(e)) = \\ &\underline{Me}\llbracket x\rrbracket(q[x\leftarrow\text{Integer}])(te)(e[x\leftarrow\underline{Me}\llbracket\text{Boolean\$true}\rrbracket(q)(te)(e)])\end{aligned}$$

Now, the following two simple representations of Booleans induce different meanings for the expression above:

1. Represent Boolean$true and Integer$0 by the same value. Then the meaning of the above expression is Integer$0.

2. Represent Boolean$true and Integer$1 by the same value (in a fashion similar to PL/I). Now the meaning of the expression becomes Integer$1.

(Note that one can achieve precisely this effect in PL/I using EXTERNAL procedures.) We now prove that our semantics disallows the possibility of producing such spurious results.

To do so, we will have to be more specific about exactly what functions in D→D we will allow as data types. In the previous section, we suggested that the retracts in D→D seemed reasonable when viewed in terms of an SECD machine implementation. Now, we will be more precise and require that our data types be retracts.

Definition: A function f:D→D is a retract iff $f = f \cdot f$. (Note that all elements of the range of f are fixed points of f, i.e., $x = f(x)$.) An element $t \in$[Const→B]x[D→D] is a data type iff:

1. $[t]_2$ (the second component) is a doubly strict retract (a function f is doubly strict iff $f(\bot) = \bot$ and $f(\top) = \top$), and
2. $\forall c \in$Const, $[t]_1 [\![c]\!]$ is a fixed point of $[t]_2$, i.e., the meanings of all constants are preservedly the second component of the type.

An element $f \in$D→D is a data type iff f is a doubly strict retract.

Given this restriction on data types, we can now give a simple characterization of what it means for an element of D to be a member of a data type.

Definition: An element $x \in$D is said to be an element of a data type t iff x is a fixed point of $\lambda x \in$D.Apply(t,x), i.e., x is in the range of λx.Apply(t,x).

Several points need to be made about these definitions. Firstly, we will impose no requirement that each element of D belong to a unique data type. We are using our data types to impose a type structure on the basically "typeless" domain D, and it is quite natural to assume that elements of D will serve as elements of many data types. In fact, because of our requirement that these retracts be doubly strict, $\bot$ and $\top$ will be elements of every data type.

Secondly, for the basic data types (those giving meaning to constants), we require only that the denotations of constants be elements of the data type, not that these denotations be <u>the only</u> elements of the data type. The weakness of the assumption can be viewed as allowing an implementor to choose any representation for the data type he finds appropriate, as long as all of the constants can be represented satisfactorily.

Given our precise meaning of data types, two questions immediately suggest themselves:

1. Does the meaning function <u>Mt</u> in fact produce data types as the meaning of type expressions? Clearly, <u>Mt</u> produces elements of T, but does it take retracts to retracts?

2. Is our meaning function <u>Me</u> "type-correct?"

Below, we answer both of these questions affirmatively.

Theorem 1: If $\forall id \in Id$, te⟦id⟧ is a data type, then for all $w \in W$, Mt⟦w⟧(te) is a data type.

Proof: By induction on the structure of w.
Basis: w = id∈Id. Then Mt⟦w⟧(te) = te⟦w⟧ and te⟦w⟧ is a data type by assumption.
Induction: There are two cases:

1. $w=w_1 \rightarrow w_2$. Then Mt⟦w⟧(te) = Arrow(Mt⟦w_1⟧(te), Mt⟦w_2⟧(t
   From the induction hypothesis, we have that both Mt⟦w_1⟧(te) and Mt⟦w_2⟧(te) are data types. So we need to show that for any data types t_1 and t_2 Arrow(t_1,t_2) is a data type (a doubly strict retract), i.e., that
   Arrow(t_1,t_2) = Arrow(t_1,t_2)∘ Arrow(t_1,t_2).
   But
   Arrow(t_1,t_2)(f) = $\lambda x \in D$.Apply(t_2,x)∘ f|D→D∘$\lambda x \in D$.Apply(t_1,x)
   and because t_1,t_2 are data types
   = ($\lambda x \in D$.Apply(t_2,x)∘$\lambda x \in D$.Apply(t_2,x))∘ f|D→D
   ∘ ($\lambda x \in D$.Apply(t_1,x)∘ $\lambda x \in D$.Apply(t_1,x))
   and simply collecting terms
   =(Arrow(t_1,t_2)∘ Arrow(t_1,t_2))(f).

   Q.E.D.

2. w = Δid.w'. Then Mt⟦w⟧(te) = Delta($\lambda t \in T$.Mt⟦w'⟧(te[id←t]
   From the induction hypothesis, we have that $\lambda t \in T$.Mt⟦w'⟧(te[id←t]) maps data types to data types, so we need to show that if $y \in T \rightarrow T$ maps data types that
   Delta(y) = Delta(y)∘ Delta(y)
   But
   Delta(y)(x)(t) = Apply(y(t),(x|T→D)(t))
   and if y(t) is a data type
   = Apply(y(t), Apply(y(t),(x|T→D)(t)))
   = (Delta(y)∘ Delta(y))(x)(t)

   Q.E.D. Theorem 1

Theorem 2: $\forall q \in Q$, $te \in Te$, $e \in Env$, if $\forall id \in Id$, te⟦id⟧ is a data type, then
Me⟦exp⟧(q)(te)(e)
is a element of data type
Mt⟦TypeOf⟦exp⟧(q)⟧(te),
ie.e, the semantics is "correctly typed."

<u>Proof</u>: By induction on the structure of exp.

<u>Basis</u>: There are two cases:

1. exp = id\$const. Then $\underline{Me}[\![exp]\!](q)(te)(e) =$

   $te[\![id]\!]$ <u>is</u> $[Const{\rightarrow}B]x[D{\rightarrow}D]$ <u>then</u> $[te[\![id]\!]]_1[\![const]\!]$ <u>else</u> $\top$

   If $[te[\![id]\!]]_1[\![const]\!]$ is defined, then because $te[\![id]\!]$ is a data type $[te[\![id]\!]]_1[\![const]\!]$ must be an element of the data type; otherwise, $\top$ is an element of every data type.

2. exp = id. Then

   $\underline{Me}[\![exp]\!](q)(te)(e) = Apply(\underline{Mt}[\![TypeOf[\![id]\!](q)(te), e[\![id]\!]]$

   and the theorem is true because

   $\underline{Mt}[\![TypeOf[\![id]\!](q)]\!](te)$ is a data type by Theorem 1.

<u>Induction</u>: There are four subcases:

1. exp = $\lambda id{\in}w.exp'$. Then

   $\underline{Me}[\![exp]\!](q)(te)(e) = \lambda y{\in}D.\underline{Me}[\![exp']\!](q[id{\in}w])(te)(e[id{\leftarrow}y])$

   To show that this function is an element of type $\underline{Mt}[\![w{\rightarrow}TypeOf[\![exp]\!](q[\![id{\leftarrow}w]\!])]\!](te)$, we need the following lemma.

   <u>Lemma</u>: $\underline{Me}[\![exp']\!](q[id{\leftarrow}w])(te)(e[id{\leftarrow}y]) =$
   $\underline{Me}[\![exp']\!](q[id{\leftarrow}w])(te)(e[id{\leftarrow}Apply(\underline{Mt}[\![w]\!](te),y)])$.

   <u>Proof</u>: That this is true can be seen from the fact that $\underline{Mt}[\![w]\!](te)$ is a data type (and thus a retract) and that if id appears in an expression in exp', its meaning will always be an element of $\underline{Mt}[\![w]\!](te)$.

   Then, given this lemma, we have that

   $\underline{Me}[\![exp]\!](q)(te)(e) =$
   $\lambda y{\in}D.\underline{Me}[\![exp']\!](q[id{\leftarrow}w])(te)(e[id{\leftarrow}y]) =$
   $\lambda y{\in}D.\underline{Me}[\![exp']\!](q[id{\leftarrow}w])(te)(e[id{\leftarrow}y])\circ$
   $\lambda x{\in}D.Apply(\underline{Mt}[\![w]\!](te),x)$

   and from the application of the induction hypothesis and our previous theorem

   $= \lambda x{\in}D.Apply(\underline{Mt}[\![TypeOf[\![exp']\!](q[id{\leftarrow}w])]\!](te),x)\circ$
   $\lambda y{\in}D.\underline{Me}[\![exp']\!](q[id{\leftarrow}w])(te)(e[id{\leftarrow}y])\circ$
   $\lambda x{\in}D.Apply(\underline{Mt}[\![w]\!](te),x)$

   Finally, from the definition of <u>Mt</u>, it is clear that this value is an element of $\underline{Mt}[\![w{\rightarrow}TypeOf[\![exp]\!](q[id{\leftarrow}w])]\!](te)$; thus the theorem is true.

2. $exp = (exp_1\ exp_2)$. Then

$$\underline{Me}[\![exp]\!](q)(te)(e) = \text{TypeOf}[\![exp_2]\!](q) \doteq \text{Dom}(\text{TypeOf}[\![exp_1]\!](q))\ \underline{then}\ (\underline{Me}[\![exp_1]\!](q)(te)(e)|D{\to}D)(\underline{Me}[\![exp_2]\!](q)(te)(e))$$

The error case (if the type-checking fails) is taken care of by the doubly strict nature of data types. Otherwise, the simple fact that exp_1 is of a functional type, i.e., $\text{TypeOf}[\![exp_1]\!](q) = w_1 \to w_2$ is sufficient to guarantee the truth of the theorem. By the induction hypothesis, if $\text{TypeOf}[\![exp_1]\!](q) = w_1 \to w_2$, then $\underline{Me}[\![exp_1]\!](q)(te)(e)$ is an element of type $\underline{Mt}[\![w_1 \to w_2]\!](te)$. And from this, we know that <u>for any</u> $d \in D$, $(\underline{Me}[\![exp_1]\!](q)(te)(e)|D{\to}D)(d)$ is an element of data type $\underline{Mt}[\![w_2]\!](te)$, as required by the theorem.

The cases of type abstraction and application are straightforward and are left to the reader.
Q.E.D. Theorem 2

An important aspect of the preceding theorem is that its truth is independent of the type-checking clause introduced in the meaning of normal applications. One is tempted to cite this as a failing of the semantics (after all, type-checking shouldn't be superfluous). But this property is symptomatic of the dangers of using an "untyped" domain to give the semantics of a language (whether it be the domain D defined above or 32-bit wo function applications may always produce results that look reason even though the argument to which the function is applied was not sonable. However, as we show below, this syntactic type-checking important in guaranteeing the representation independence we desi

Our representation-independence result is motivated by the following observation. One can regard the syntax of our language as defining a class of "abstract values" and the meaning of data types as choosing a particular "representation" in D for each "abstract value." For example, we can view Bool$true and λx∈Bool.x as the "abstract values" true and the Boolean identity function. An association of Bool with some data type in T thus induces a particular representation in D for each of these abstract values. What we show below is that the semantics preserves this choice of representation, i.e., after choosing a representation for Boolean constants, the meaning of Boolean functions is consistent with this choice of representation. Thus, we will have that the representation of (λx∈Bool.x Bool$true) will always be the same as the representation of Bool$true (as long as Bool is a data type, as defined above). In other words, our semantics is "representation independent."

To prove this, we first define a notion of "similar data types," i.e., data types that are structurally similar and may be freely substituted one for another.

Definition: Two data types t and t' are similar iff:

1. t, t' ∈ [Const→B]x[D→D], i.e., both allow the introduction of constants, or
2. t = Arrow(t_1,t_2) and t' = Arrow(t_1',t_2') and t_1 similar to t_1' and t_2 similar to t_2', or
3. t = Delta(y) and t' = Delta(y') and for all t_1,t_1' t_1 similar to t_1' implies y(t_1) similar to y'(t_1')

Lemma: If ∀id∈Id, te⟦id⟧ is similar to te'⟦id⟧, then w∈W, Mt⟦w⟧(te) similar to Mt⟦w⟧(te').

Proof: By structural induction on w (omitted).

Now, we define a representation relation between elements of D stating when d and d' are representations of the same "abstract value" according to similar types t and t'.

<u>Definition</u>: Let <u>Rep</u>(t,t'): D⟶̸D be a relation in (D x D) for similar data types t and t' such that <u>Rep</u>(t,t'): d ⊢ d' iff d is an element of t, d' an element of t' and either:

1. t, t'ε [Const→B]x[D→D] and ∃cεConst such that $[t]_1$⟦c⟧ = d and $[t']_1$⟦c⟧ = d', i.e., d and d' are denotations given to the same constant, or
2. <u>Rep</u>(Arrow(t_1,t_2), Arrow(t_1',t_2')): d ⊢ d' iff ∀d_1,d_1' <u>Rep</u>(t_1,t_1'): d_1 ⊢ d_1' implies <u>Rep</u>(t_2,t_2'): d(d_1) ⊢ d'(d_1'), or
3. <u>Rep</u>(Delta(y), Delta(y')): d ⊢ d' iff ∀similar t_1,t_1' <u>Rep</u>(y(t_1), y'(t_1')): d(t_1) ⊢ d'(t_1').

And, now we can prove that <u>Me</u> preserves <u>Rep</u>.

<u>Theorem 3</u>: If we have,

a. ∀idεId, te⟦id⟧ similar to te'⟦id⟧, and
b. ∀id,
   <u>Rep</u>(<u>Mt</u>⟦q ⟦id⟧⟧(te), <u>Mt</u>⟦q⟦id⟧⟧(te')): e⟦id⟧ ⊢ e'⟦
   i.e., each identifier in e,e' refer to the same "abstract value," then ∀expεExp,
   <u>Rep</u>(t,t'): <u>Me</u>⟦exp⟧(q)(te)(e) ⊢ <u>Me</u>⟦exp⟧(q)(te')(e')
   where
   t = <u>Mt</u>⟦TypeOf⟦exp⟧(q)⟧(te) and
   t' = <u>Mt</u>⟦TypeOf⟦exp⟧(q)⟧(te').

Proof: By structural induction on exp. Most cases are simple applications of the definitions of Rep and Me and the preceding theorems (to show that Me produces values in t and t'). The only interesting case is normal application, where we have that if exp = (exp_1 exp_2), then

Rep(Arrow(t_1,t_2), Arrow(t_1',t_2')): Me⟦exp_1⟧(q)(te)(e) ↦ Me⟦exp_1⟧(q)(te')(e')

from the induction hypothesis and the fact that TypeOf⟦exp_1⟧(q) = $w_1 \rightarrow w_2$ for some w_1 and w_2. Now, for our definition of Rep to assure us that Rep(t_2,t_2'): Me⟦exp⟧(q)(te)(e) ↦ Me⟦exp ⟧(q)(te')(e'), we must have

Rep(t_1,t_1'): Me⟦exp_2⟧(q)(te)(e) → Me⟦exp_2⟧(q)(te')(e'),

which we can assure only if TypeOf⟦exp_2⟧(q)≃w_1. (Here we need a trivial lemma that for all w,w', w≃w' implies Mt⟦w⟧(te) = Mt⟦w'⟧(te).) And this is exactly what the type-checking clause of the meaning of application allows us to assert. Q.E.D. theorem 3

One way to view this result is that "the implementor is free to choose any representation he desires." For example, let us assume for the moment that the domain B includes the usual flat domains of integers and Booleans, Int and Bool, i.e., B = Int+Bool+... . Then consider the following data types t_1 and t_2 used to implement Booleans:

1. t_1 = <λc∊C. c = true then 1 else c = false then 0 else ⊥, λx∊D.x> and
2. t_2 = <λc∊C. c = true then true else c = false then false else ⊥, λx∊D.x|Bool>

Note that the retract used in t_2 only produces values in Bool, i.e., the only elements of data type t_2 are the elements of the domain Bool, while all values in D are elements of the data type t_1.

Now from our definition of Rep, $\underline{Rep}_{t_1,t_2}$ relates the following pairs of values

(1,<u>true</u>), (0,<u>false</u>), ($\perp$,$\perp$), and ($\top$,$\top$). And from the theorem above, we know that if TypeOf⟦exp⟧(q) = Boolean, then

$$\underline{\mathrm{Rep}}_{t_1,t_2}(\underline{\mathrm{Me}}⟦\mathrm{exp}⟧(q)(te[\mathrm{Boolean}\leftarrow t_1])(e),\ \underline{\mathrm{Me}}⟦\mathrm{exp}⟧(q)(te'[\mathrm{Boolean}\leftarrow t_2])(e'))$$

if te,te' and e,e' are as required in the theorem. This means that:

1. even using t_1 as the meaning of Boolean, the only values produced by Boolean expressions are 0,1,$\perp$, or $\top$, and
2. the denotation of the same Boolean constant is produced in either case, i.e., if the "abstract meaning" of an expression is Boolean$true, then its meaning will always be the meaning of Boolean$true.

Thus, when read in implementation-oriented terms, this says that the implementor may choose the meanings of the basic data types as he pleases. If we view the purpose of type-checking to guarantee representation independence of the sort we define above, then our semantics is not only "correctly typed," but "strongly typed."

4. Conclusions

We have described a new approach to the meaning of "data type" and applied it to the definition of a typed lambda-calculus extension. In conclusion, we would like to step back from the details of the previous section and suggest the advantages of this approach as illustrated by our lambda-calculus semantics.

Firstly, we claim that our lambda calculus semantics, while admittedly abstract, treats data types in an intuitively appealing manner. As we described above, the choice of

$T = [Const \to B] \times [D \to D] + D \to D$

can be motivated in terms of an SECD implementation and our restriction of T to retracts again makes sense in terms of this machine-oriented view. (It is interesting to note that Scott also uses retracts to define data types of his "universal domain" $P\omega$ [Scott 1976].) And, as is common in statically typed languages our semantics does not involve any checking of the "types" of values, but only the "syntactic types" associated with identifiers. Finally in this regard we note the weakness of the assumptions used in proving the theorems of the previous section. Of particular interest is the fact that no assumptions were made about the domain B of basic values. Although we are using B to interpret constants, we do not require it to be "flat", (i.e., without structure). Indeed, we could even do without B altogether and make elements in $D \to D$ serve double duty as both functions and constants without changing the interpretation of data types at all.

This definition also shows nicely the differences between typed and "typeless" languages like BCPL. One way to view a "typeless" language is to say that in fact there is only a single type. In BCPL, this view can be seen in the single interpretation of assignment; the operational of x:=y is "move k bits from y to x." In the untyped lambda calculus, the same view suggest a semantics with only a single meaning for "take the value of the variable x." This is exactly what one finds in semantic treatments of untyped or "typeless" lambda calculus and its variants (c.f., [Stoy 1974, Scott 1976]).

Finally, two points relating to language design should be made. As languages have included more complex data type definitions, the rules for type-checking have also become far more complex (see, for example, the descriptions of type-checking in Euclid [Lampson et al 1977] or Mesa [Geschke et al 1977]). Although t expressed purpose of such type-checking is to catch program errors, one can't help wondering whether these type-checking rules do not, in fact, introduce subtle semantic errors. Our representation theorem, however, suggests a means of judging the soundness of any proposed type-checking rules; the type-checking must be sufficient to guarantee the representation independence of the semantics in the fashion we described above.

But, most interestingly, our view of data types allows a simple and straightforward treatment of constructs allowing types as parameters. To illustrate the lack of understanding of such constructs, we need only refer to the following comments of Wirth [1975]:

> "I would caution, however, against any further generalization [of array type specifications of procedure parameters]. Allowing the component type of an array to be a parameter too, for example, would destroy many advantages of the Pascal type concept at once."

Although PASCAL data types are certainly more complicated than typed lambda calculus data types, we are convinced that the same principles used in this paper apply. To this end, we (with A. Demers, R. Teitelbaum, and J. Williams) are designing an extension of PASCAL (tentatively called Russell) incorporating polymorphic procedures and data types. Both denotational and axiomatic semantics for this language are also being developed.

References

[Gannon and Horning 1975]
Gannon, J.D. and J.J. Horning, "The Impact of Language Design on the Production of Reliable Software," SIGPLAN Notices 10,6 (June 1975), pp. 10-22.

[Geschke et al 1977]
Geschke, C.M., J.H. Morris, and E.H. Satterthwaite, "Early Experiences with Mesa," SIGPLAN Notices 12,3 (March 1977), pg. 138.

[Guttag 1975]
Guttag, J.V., "The Specification and Application to Programming of Abstract Data Types," Ph.D. thesis, Department of Computer Science, University of Toronto, 1975.

[Hoare and Wirth 1973]
Hoare, C.A.R. and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal," Acta Informatica 2, 1973, pp. 335-355.

[Jensen and Wirth 1975]
Jensen, K. and N. Wirth, PASCAL User Manual and Report, Springer-Verlag, 1975.

[Lampson et al 1977]
Lampson, B., J.J. Horning, R.L. London, and G.L. Popek, "Report on the Programming Language Euclid," SIGPLAN Notices 12,2 (Feb. 1977), pp. 1-79.

[Ledgard 1972]
Ledgard, H., "A Model for Type Checking - with an Application to Algol 60," Comm.ACM 15,11 (Nov. 1972), pp. 956-966.

[Liskov and Zilles 1974]
Liskov, B. and S. Zilles, "Programming with Abstract Data Types," SIGPLAN Notices 9,4 (April 1974), pp. 50-59.

[Morris 1968]
Morris, J.H., "Lambda-Calculus Models of Programming Languages," MAC-TR-57, Project MAC, MIT, 1968.

[Morris 1973]
Morris, J.H., "Types are not Sets," ACM Symposium on Principles of Programming Languages, Boston, 1973, pp. 120-124

[Reynolds 1970]
Reynolds, J.C., "GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," Comm.ACM 13,5 (May 1970), pp. 308-318.

[Reynolds 1974]
Reynolds, J.C., "Towards a Theory of Type Structure," Colloquium on Programming, Pairs, 1974.

[Schaffert et al 1975]
Schaffert, C., A. Synder, and R. Atkinson," The CLU Reference Manual," Project MAC, MIT, 1975.

[Scott 1976]
Scott, D., "Data Types as Lattices," SIAM J. Computing 5,3 (Sept. 1976), pp. 522-580.

[Stoy 1974]
Stoy, J., "The Scott-Strachey Approach to the Mathematical Semantics of Programming Languages," Project MAC, MIT, 1974.

[Tennent 1976]
Tennent, R.D., "The Denotational Semantics of Programming Languages," Comm.ACM 19,8 (Aug. 1976), pp. 437-453.

[Wegner 1968]
Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, 1968.

[Wirth 1975]
Wirth, N., "An Assessment of the Programming Language Pascal," SIGPLAN Notices 10,6 (June 1975), pp. 23-30.