CSc 520

Principles of Programming Languages

15 : Types — Equivalence

Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

Type Checking

- In statically typed languages, when an object is defined, we must also give its type:

TYPE *equivalence* = *type*;
TYPE *subrange* = [*from*..*to*];
TYPE *enumeration* = (*id*,*id*,...);
TYPE *array* = **ARRAY** *range* **OF** *type*;
TYPE *record* = **RECORD**
        field : *type*;
    END;
TYPE *func* = **PROCEDURE** (INTEGER):REAL;
TYPE *set* = **SET OF** *type*;
TYPE *ptr* = **POINTER TO** *type*;
VAR *name* : *type*;
PROCEDURE *name* (*formal-list*) [: *type*];

Type Checking...

- In statically typed languages, whenever a typed object us used, we must check that it occurs in the right type context:

```
var x :  integer;
var y :  record a:float; b:integer; end;
begin
   x := y.a; (* ⇐ error *)
   x := x + y.b;
   x := x + 5.0; (* ⇐ error?  *)
end
```

Type Checking. . .

- type compatibility — can an object of a certain type T be used in a certain context expecting a type U?
  - If $T = U$ then, yes, of course.
  - But, if T is almost the same as U, then what?
- type equivalence — what does it mean for two types T and U to be equivalent?

Type Checking...

- type conversion — how can we convert (cast) a value of one type into another?
- type coercion — which automatic conversions between types should the language allow?
- nonconverting type casts — what are the consequences of allowing values to change type without conversion?
- type inference — given individual types of parts of an expression, what is the type of the whole expression?

Type Equivalence

Structural Type Equivalence

- structural equivalence — two types as the same if they consist of the same components.
- Example:

```
TYPE T1 = RECORD
             a:INTEGER;
             b:ARRAY [0..10] OF CHAR;
          END;
TYPE T2 = RECORD
             a:INTEGER;
             b:ARRAY [0..10] OF CHAR;
          END;
```

  Types `T1` and `T2` are equivalent.
- Algol-68, Modula-3, C, ML use structural type equivalence.

Name Type Equivalence

- name equivalence — each type declaration introduces a new type, distinct from all others.
- Example:

```
TYPE T1 = ARRAY [0..10] OF CHAR;
TYPE T2 = ARRAY [0..10] OF CHAR;
```

  Types `T1` and `T2` are not equivalent.
- Java and Ada use name equivalence.

Declaration Type Equivalence

- declaration equivalence — two types are equivalent if they lead back to the same type.
- Example:

```
TYPE T1 = ARRAY [0..10] OF CHAR;
TYPE T2 = T1;
TYPE T3 = T2;
TYPE T4 = ARRAY [0..10] OF CHAR;
```

  Types `T1`, `T2`, `T3` are equivalent.
- Pascal, Modula-2 use declaration equivalence.
- Type equivalence was left out of Pascal definition: some implementations used name equivalence some structural equivalence some declaration equivalence.

Type Equivalence...

```
TYPE T1 = RECORD a:CHAR; b:REAL END;
TYPE T2 = RECORD a:CHAR; b:REAL END;
VAR x1 :  T1;
VAR x2 :  T2;
VAR x3,x4:  RECORD a:CHAR; b:REAL END;
BEGIN
   x1 := x2;    (* OK, or not?  *)
   x3 := x4;    (* OK, or not?  *)
END
```

- name-equivalence: both assignments are illegal.
- declaration-equivalence: only 2nd assignment is legal.
- structural type equivalence: both assignments are legal.

Structural Type Equivalence

```
TYPE Shape = OBJECT
        METHOD draw (); ···
        METHOD move (X,Y:REAL); ···
     END;
TYPE Cowboy = OBJECT
        METHOD draw (); ···
        METHOD move (X,Y:REAL); ···
     END;
VAR s:S; c:C;
BEGIN s := c; (* OK? *) END
```

- In Modula-3 (which uses structural equivalence) `s` and `c` are compatible! In Object-Pascal (which uses name equivalence) they are not.

Structural Type Equivalence

- Are these types structurally equivalent:

```
TYPE T1 = RECORD
             a : INTEGER;
             b : INTEGER;
          END;
TYPE T2 = RECORD
             b : INTEGER;
             a : INTEGER;
          END;
```

- ML and Algol-68: NO, most other languages, YES.

Structural Type Equivalence

- Are these types structurally equivalent:

```
TYPE T1 = ARRAY [1..10] OF CHAR;
TYPE T2 = ARRAY [1..2*5] OF CHAR;
TYPE T3 = ARRAY [0..9] OF CHAR;
```

- Algol-68: `T1,T2,T3` are equivalent.
- Modula-2: `T1,T2` are equivalent, `T3` not.

Name Equivalence

- Is this assignment legal?

```
TYPE celsius = REAL;
TYPE farenheit = REAL;
VAR c : celsius;
VAR f : farenheit;
c := f;
```

- In Modula-2, YES. This is a problem.
- In Ada, we can construct new types, which are not equivalent:

```
type celsius is new integer;
type farenheit is new integer;
```

Case Study: C

- C uses a combination of declaration and structural equivalence.
- Unions and structs use declaration equivalence.
- Pointers and arrays use structural equivalence.

Case Study: Modula-3

Modula-3

```
TYPE
    T1 = {A, B, C};
    T2 = {A, B, C};
    U1 = [T1.A..T1.C];
    U2 = [T1.A..T2.C];
    V = {A,B};
```

- `T1` and `T2` are the same type. Modula-3 uses *structural type equivalence*. Two types are the same if they look the same once they have been expanded.
- `T1` and `U1` are different types, one is an enumeration, the other a subrange.

Modula-3

```
TYPE
  T = REF INTEGER;
  S = BRANDED "myType" REF INTEGER; (* Explicit *)
  V = BRANDED REF INTEGER; (* Compiler-generated *)
```

- Branded types are different from all other types. It is a way of circumventing structural type equivalence.

Readings and References

- Read Scott, pp. 321–335.

How Modula-3 Got Structural Type Equivalence

How the language got its spots

From Chapter 8, *Systems Programming with Modula-3*, Edited by Greg Nelson, Prentice-Hall, ISBN 0-13-590464-1.

Anonymous

> *I greatly welcomed the chance of meeting and hearing the wisdom of many of the original language designers. I was astonished and dismayed at the heat and even rancour of their discussions. Apparently the original design of ALGOL 60 had not proceeded in that spit-it of dispassionate searchfor truth which the quality of the language had led me to suppose.*
> *-C.A.R. Hoare*

Like many programming languages, Modula-3 was designed by a committee. The meetings were held at the DEC Systems Research Center in Palo Alto, whose director, Bob Taylor, likes to record important events on videotape-including our meetings.

Introduction

At first we found the whirring of the cameras distracting, but eventually we became used to it. We even started to imagine that the tapes might be useful in university courses, to teach students how real scientists approach problems of programming language design.

Unfortunately, when we reviewed the tapes at the end of the project it was obvious that to show them to students was out of the question. Such scenes would probably drive students out of computing, if not all the way out of the sciences. In fact, to show the tapes to anybody at all would be highly embarrassing. But our sense of duty to history prevailed, and we resolved to provide the world with copies of the tapes. Nobody was more disappointed than we when the secretary making the copies inadvertently turned the machine to "erase" instead of "copy", and the record was irretrievably destroyed.

Introduction

As often happens with mishaps of this sort, a few sections of some of the tapes survived, of which a transcript was made for this book. However, even after the usual editing (deletion of expletives, libels, etc.) the publisher still returned the transcript with the tactful suggestion that its truths would be more appealing if they were more fully clothed. As a last resort, we have translated the material into a fictional dialogue, featuring the following characters:

Introduction

Dr. Lambdaman. (An internationally eminent authority on programming languages and their semantics.)

Jo Programmer. (While **Dr. Lambdaman** lectured the committee on Abstract This and Abstract That, Jo amused herself writing microcode in her head.)

Harry Hackwell. (He joined the committee to make sure that Modula-3 would "support his style of programming".)

Noam Wright. (He was the most vocal member of the committee, but his remarks have been drastically abridged in the following account, since they contained almost no information.)

Professor Pluckless. (The patron saint of committee design.)

The casting was not fixed, but varied from day to day, even from minute to minute, and every member of the committee starred at times in every role.

How the types got their identity

PLUCKLESS: The writers working on the language definition have asked us to give them a clear definition of when two types are identical.

HACKWELL: That's easy. Everyone knows that Modula uses name equivalence. Two types are the same if they have the same name.

LAMBDAMAN: The issue of type identity is too fundamental to be decided on the basis of tradition. We should explore the alternatives and decide on the basis of technical merit.

WRIGHT: Well said. All men of principle favor structural equivalence.

PLUCKLESS: Oh yeah? Don't they know the principle "if it ain't broken, don't fix it"?

JO: If you men of principle could bring yourselves to descend occasionally into the realm of specifics, we just might finish this language by the end of the century. I wonder if Hackwell really means what he says. After the declarations

How the types got their identity

```
TYPE A = REF INTEGER;
     B = REF INTEGER;
     C = B;
```

the types A, B, and `C` have distinct names. By Hackwell's definition they would be different types. But in Modula-2, `B` and `C` are the same type.

HACKWELL: I didn't mean to change the semantics from Modula-2. I want `B` and `C` to be the same, and `A` and `B` to be different. I guess "name equivalence" is a misnomer. But everyone knows what it means.

LAMBDAMAN: Should we put that in the manual?

HACKWELL: How about this: two types are the same if they have the same name, or if one of them is a renaming of the other, as in the case of `TYPE C = B` above.

How the types got their identity

PLUCKLESS: I suppose types can be identified by a chain of renamings.

HACKWELL: Yes, of course. The writers are good at phrasing details like that.

LAMBDAMAN: I don't like this wording because it makes a special case of renamings. It shouldn't take any extra words to say that `B` and `C` are the same after the declaration `C = B`. What requires explanation is that `A` and `B` are different after both have been declared to be REF INTEGER.

JO: It's also unclear what happens to anonymous types. For example, in Modula-2, after

```
TYPE R =
   RECORD p, q: REF INTEGER; r: REF INTEGER;
END
```

the p and q fields have the same type, but the r field has a different type. I don't think this follows from Hackwell's wording, since the types involved are anonymous.

How the types got their identity

PLUCKLESS: Let me try. The way I like to think of it, every type comes in a potentially unlimited number of different "brands". Each occurrence of a type constructor puts a distinct brand on the type it creates. You can imagine there's a global counter that gets incremented each time a type constructor is applied. The value of the counter is used to generate the unique brand characterizing that application. In the declaration TYPE `C = B`, no type constructor is applied, no brand is created, and `C` becomes the same type as `B`. But each occurrence of REF, named or anonymous, creates its own unique brand.

LAMBDAMAN: In Modula-2, after the declarations

```
TYPE
   T = INTEGER;
   U = INTEGER
```

How the types got their identity

the types `T` and `U` are the same. But under your theory, wouldn't the two occurrences of the type constructor INTEGER produce different types? Then you can ask about the identity of types from different programs. This seems useful for programming distributed systems.

HACKWELL: For example?

JO: When making a remote procedure call, the type of the actual and type of the formal are types that appear in different programs. To make sense of the requirement that they be the same, you have to compare types in different programs.

HACKWELL: How about a more concrete example?

JO: Essentially the same problem arises with type-safe persistent data. Consider tile call `Pkl.Write(r, f)`, which writes the value r onto the file f. Similarly, `Pkl.Read(f)` reads, a pickle from the file f and builds and returns the corresponding value. Now the question is: when is it type-safe for a pickle of type `T` written by one program to be read and assigned to a variable of type `U` in another?

How the types got their identity

HACKWELL: If `T` and `U` are the same type, obviously.

JO: But `T` and `U` are in different programs, so that only makes sense under structural equivalence.

HACKWELL: Why not allow the operation if the name of `T` in P1 is the same as the name of `U` in P2?

JO: That's hopeless. The types could have completely different structures.

HACKWELL: I suppose you could use structural equivalence for this special case of inter-program typechecking, and still use name equivalence within a single program.

JO: That runs into problems. Suppose that a program contains two types that are structurally equivalent to the type of a pickle that it reads. How does `Pkl.Read` choose the result type? For example, suppose one program is

```
Pkl.Write(NEW(REF INTEGER), "file.pkl")
```

How the types got their identity

and the other is

```
TYPE U1 = REF INTEGER;
     U2 = REF INTEGER;
VAR v := Pkl.Read("file.pk1");
```

What's the final type of v?

HACKWELL: You could treat this as an error. After all, whether you use name equivalence or structural equivalence, it will be an error if the pickle-reading program has no type that is structurally equivalent to the type in the pickle. So it is natural also to make it an error if the program has more than one such type.

JO: It might be natural but it's a gross violation of modularity. It would mean that adding a module containing a private type REF INTEGER could break another section of the program that reads pickles. That's unacceptable.

How the types got their identity

HACKWELL: You could make the caller of `Pkl.Read` specify the type by supplying a typecode.

JO: That only helps for the root of the data structure. `Pkl.Read` will still have to come up with types for any REFANYs that are embedded inside it.

HACKWELL: But the pickles package for Modula-2+ uses name equivalence. So there must be a solution to this problem.

JO: In Modula-2+, `Pkl.Write(r)` puts both the name and structure of r's type into the pickle. The program reading the pickle must have a type with that name and structure.

LAMBDAMAN: What does the Modula-2+ `Pkl.Write` do if it encounters a type with more than one name? Or with no names? For example, after

```
TYPE T = RECORD f: REF INTEGER; g: REF INTEGER END;
VAR t: T;
Pkl.Write(t.g)
```

How the types got their identity

JO: If the type has more than one name, a canonical name is selected by repeatedly undoing type renamings. If the type is anonymous, a name is created for it by some rules having to do with the context in which the type expression occurs.

HACKWELL: So what's the matter with that?

WRIGHT: Can't you recognize a pile of poo when you step in it?

JO: It means that you can invalidate pickles on the disk by changing the name of a type in a program, moving a declaration from one module to another, or, in case of anonymous types, by inserting or deleting declarations that precede a declaration containing an anonymous type.

HACKWELL: Why are we worrying so much about an esoteric facility like pickles? Is it too much to ask that programmers choose a name for each pickled type, and stick to it?

How the types got their identity

JO: Type-safe persistent data is not esoteric. It's Important and it's going to become more important. The current situation with name equivalence is irritating. One Modula-2+ programmer added a type declaration at the top of an interface, never dreaming that such a simple change could break the pickle reading code in a distant module. By the time it was discovered that it was broken, most of the system had been compiled against the new interface. I think it's clear that name equivalence is not quite right.

HACKWELL: Well, I think that structural equivalence is not quite fight either, and my argument is based on something simpler than pickles. For example, consider these types:

```
TYPE
   Apple = REF RECORD ... END;
   Orange = REF RECORD ... END;
```

How the types got their identity

Suppose that by coincidence, the types have the same structure. With structural equivalence, if I declare a procedure that takes an Apple, the type checker will also allow it to take an Orange; even though it's probably a programming error. Structural equivalence weakens typechecking by introducing accidental type coincidences.

LAMBDAMAN: In principle there's something to what you say, but in practice name equivalence is more lenient than you are letting on. Consider these declarations:

```
TYPE
   ExtendedChar = [0..32767];
   ProcessID = [0..32767];
```

In Modula-2, with name equivalence, assignments between Apples and Oranges are forbidden, but assignments between ExtendedChars and ProcessIDs are allowed.

How the types got their identity

WRIGHT: Name equivalence purists preach that all types are created distinct. But some types are more distinct than others!

HACKWELL: I would be happy to explore alternatives in which assignments between ExtendedChar and ProcessIDs require explicit conversions using ORD and VAL.

PLUCKLESS: Really, Hackwell, I don't think any of us would like the result, even you. I think the point is that the danger of accidental coincidences between Apples and Oranges is not so serious a practical problem as you are making out.

LAMBDAMAN: Beside, if it does happen that a programmer erroneously assigns an Apple to an Orange and complains that the type system let it through, we have a perfectly good answer: he should have made the types opaque.

JO: Speaking of opaque types, aren't they a problem for structural equivalence?

LAMBDAMAN: How so?

How the types got their identity

JO: If a client of an opaque type knows or guesses the concrete type, then with structural equivalence, he can violate the abstraction boundary. For example, consider

```
INTERFACE Wr;
   TYPE T <: ROOT; ...
END Wr;
MODULE Wr;
   REVEAL T = OBJECT private: ... END; ...
END Wr.
```

The whole idea of opaque types is that a client of `Wr` can access variables of type `Wr.T` only through procedures that are revealed in the interface. The client is not supposed to be able to deal directly with the object's data fields. But with structural equivalence, the client can use TYPECASE to get at the private fields, like this:

How the types got their identity

```
TYPE WrRep = OBJECT private: ... END;
VAR wr := NEW(Wr.T);

TYPECASE wr OF
   WrRep (w) => ...
END
```

Since with structural equivalence the types `Wr.T` and `WrRep` are the same, the TYPECASE statement will take the first arm, and in that arm the private fields of wr will be accessible to the client via w. This is a disaster for abstraction.

LAMBDAMAN: This is a problem, but I'm sure we can easily fix it. Perhaps the abstract and concrete types shouldn't be the same. Instead they could be related by some kind of abstraction function. For example, the implementation module could contain

```
REVEAL Wr.T = ABSTRACT(OBJECT private: ... END)
```

or something of the sort.

How the types got their identity

PLUCKLESS: That doesn't solve anything. The client could declare `WrRep` to be ABSTRACT(OBJECT private: ... END).

JO: In general, if the concrete type is defined by the declaration REVEAL T = E, then a TYPECASE arm that contains the expression E will succeed. This is an unavoidable consequence of your beloved referential transparency principle.

LAMBDAMAN: I suppose that different occurrences of ABSTRACT could produce different types.

PLUCKLESS: Then your abstract types wouldn't be any different from my branded types.

JO: Furthermore, all the problems that name equivalence poses for distributed programming will reappear. If I write a value of type ABSTRACT(REF INTEGER) into a pickle, just which brand of ABSTRACT(REF INTEGER) will I get when I read it out?

How the types got their identity

LAMBDAMAN: Let's try another tack. Forget about these unprincipled Modula opaque types. What we need are real abstract types.

PLUCKLESS: What is a real abstract type?

LAMBDAMAN: It's an abstract type whose corresponding concrete type is guaranteed to be hidden at runtime as well as at compile time.

PLUCKLESS: How do you define them?

LAMBDAMAN: The basic idea is very simple. We have defined TYPECASE to classify a reference r to be a member of type `T` if r's allocated type is a subtype of `T`. But a subtype in what sense'? In any module there are, in a sense, two subtype relations. There is the global subtype relation on all the types in a program. There is also a smaller relation, consisting of those subtype facts that are statically visible in the module. In our current language, we have defined TYPECASE to use the global subtype relation. If we change it to use the local relation instead, then TYPECASE will no longer be able to violate abstraction boundaries.

How the types got their identity

HACKWELL: Let me see if I understand this. I once got burned in Modula-2+ by constructing a heterogeneous list of TEXTs and OS.ProcessIDs. These are both opaque types, and I tried to use TYPECASE to distinguish them from one another when I read the elements out of the list. Unfortunately, the concrete type of an OS. ProcessID turned out to be TEXT! Under your proposal, my program would have worked?

LAMBDAMAN: Certainly. In the scope of your TYPECASE statement, TEXT and OS.ProcessID were unrelated in the local subtype relation. Therefore in that scope, a TEXT would narrow to a TEXT but not to an OS. ProcessID, and an OS. ProcessID would narrow to an OS. ProcessID but not to a TEXT.

HACKWELL: I like it.

PLUCKLESS: How do you implement it?

LAMBDAMAN: Instead of one table defining the subtype relation, keep one table for each module.

How the types got their identity

PLUCKLESS: Couldn't that take quadratic space? You'll have to do better than that if you expect us to sign up for this scheme.

JO: The implementation is the least of the problems with this proposal. Look at the following program:

```
INTERFACE I; TYPE T <: REFANY; END I.

INTERFACE J; PROCEDURE P(r: REF INT); END J.

MODULE I;
   IMPORT J;
   REVEAL T = REF INT;
BEGIN
   J.P(NEW(T))
END I.
```

How the types got their identity

Is this OK so far?

LAMBDAMAN: Yes. The call to J.P typechecks, since within the scope of the module I it is known that `T` = REF INT; from which it follows of course that `T <: REF INT`. Consequently the NEW (T) actual can be bound to the REF INT formal.

JO: Now look at the implementation of J. P:

```
PROCEDURE P(ri: REF INT) =
   VAR ra: REFANY := ri; BEGIN
   ri := NARROW(ra, REF INT);
END P;
```

The programmer of `J.P` assumed, not unreasonably, that if he assigned a `REF INT` to a `REFANY` then he would be able to narrow that `REFANY` back into a `REF INT`. Unfortunately for him, the allocated type of the actual `ri` is the "real" abstract type `I. T`. In the scope of the call to `J. P`, it was known that I. `T<: REF INT`. But in the scope of the module J this is not known, so the `NARROW` will fail.

How the types got their identity

LAMBDAMAN: Your example is a bit contrived.

JO: It puts a value of type `REF T` into a variable of type `REFANY` and then narrows it back again. Admittedly this is rarely done in two consecutive assignments, but it is common to do indirectly; for example, by putting the value into a table of `REFANY`s. With "real" abstract types, a programmer can never trust that a parameter of type `REF INT` really is a `REF INT`.

PLUCKLESS: So much for real abstract types.

HACKWELL: And so much for structural equivalence, since it makes the world unsafe for abstraction.

JO: But Hackwell, as you found out when you blundered with TEXT and `OSProcessID`, opaque types aren't entirely safe with name equivalence either.

HACKWELL: I don't blame that problem on name equivalence. I blame it on the revelation

```
REVEAL OS.ProcessID = TEXT
```

How the types got their identity

which should be illegal. We should require that the concrete type expression in a revelation must contain a type constructor. For example, it could be

```
REVEAL OS.ProcessID = RECORD t: TEXT END
```

Then the automatic branding of name equivalence will make all opaque types distinct.

JO: This seems like the kind of practical approach that we need. But I'm still concerned with the problems that name equivalence poses for distributed programming. Doesn't your idea work with explicit brands as well as implicit brands?

LAMBDAMAN: An excellent point. We can add a type constructor that applies a brand. If `T` is a type and b is a text constant, let `BRAND (b, T)` be the type that is the same as `T` except that it is branded b.

WRIGHT: I have nothing against brands if they're explicit. Explicit brands preserve referential transparency.

How the types got their identity

LAMBDAMAN: If you write a `BRAND("Wr314", REF INTEGER)` into a pickle, then you get a `BRAND("Wr314" , REF INTEGER)` when you read it out.

PLUCKLESS: But what about the conflict between structural equivalence and abstract types? If the concrete type for an opaque type is `BRAND ("Wr314",REF INTEGER)`, then a client can get at the representation by repeating that type expression in a typecase arm, brand and all.

LAMBDAMAN: We simply prohibit any brand from appearing more than once in a program. That's easy to enforce at link time.

JO: Explicit brands allow the programmer to avoid the kind of accidental type coincidences that worry Hackwell, even when the type involved is not opaque.

How the types got their identity

PLUCKLESS: All this creativity makes me nervous. How can we tell if it hangs together?

LAMBDAMAN: What could go wrong?

PLUCKLESS: Well, one thing that bothers me is the exact definition of `BRAND (b, T)`.

LAMBDAMAN: It has all tile properties of T, except its brand is b.

PLUCKLESS: Oh yeah? Is `T` identical with T?

LAMBDAMAN: Of course.

PLUCKLESS: Then since `BRAND(b, T)` has all the properties of T, one of which is to be identical with T, it follows that `BRAND(b, T)` is identical with T!

How the types got their identity

LAMBDAMAN: Don't be ridiculous. You know what I mean.

HACKWELL: Hall! You always get on your high horse whenever I say that.

PLUCKLESS: I don't think I am being ridiculous. I think your definition is nonsense. Here's another example:

```
TYPE
    T = OBJECT METHODS m() := P END;
    U = BRAND ("X", T) ;
PROCEDURE P(self: T)
```

If `U` has the same properties as `T`, then its m method is `P`. But `P` takes a `T`, not a `U`, so it can't be a method of `U`.

LAMBDAMAN: Oops. Good point. I suppose we could list those properties of `T` that are inherited by `BRAND(b, T)`.

JO: The writers won't like that.

How the types got their identity

LAMBDAMAN: I'm not fond of it myself.

PLUCKLESS: Perhaps `BRAND` shouldn't be a type constructor in its own right, but an optional clause in existing type constructors.

LAMBDAMAN: That will do the trick! You would write something like this:

```
TYPE T = OBJECT fields METHODS methods BRAND b END
```

In the formal semantics, this is an application of the type constructor OBJECT to arguments that include the fields, methods, and brand. Since the brand occurs within the expanded definition, it makes the type unique.

JO: Do you propose that brands he allowed in all type constructors, or only in reference types?

HACKWELL: Can I include the keyword `BRAND` but omit the text literal `b`?

How the types got their identity

PLUCKLESS: You're not serious about that syntax, I hope?

LAMBDAMAN: I'm sure we can reach consensus on these little details.

PLUCKLESS: Maybe we could, but maybe we won't have to. Isn't it time to settle the basic question of name equivalence versus structural equivalence? I think we understand the positions as well as we are going to.

LAMBDAMAN: I vote for structural equivalence with explicit brands.

HACKWELL: I say name equivalence is simpler to think about and to implement, and that structural equivalence is evil because it allows accidental type equivalences.

WRIGHT: You all can vote for whatever you want, but I will always know what was right.

How the types got their identity

PLUCKLESS: To me this whole issue is about as exciting as whether 3.5 should round up to 4 or down to 3. The religious difference between the two proposals may be large, but the practical difference is tiny. If we go with structural equivalence, we'll be letting ourselves in for a lot of unnecessary flak. I vote for name equivalence.

JO: I agree that the practical side of the issue is small compared to the fuss everybody makes about it. Both designs will certainly work out from an engineering point of view. So we should choose on the basis of taste, not tradition. I vote for structural equivalence, since it seems better for distributed programming.

Thus the committee adopted structural equivalence, by a vote of three to two.