

Class 33: Types (3)

Held: Friday, April 20, 2007

Summary: Today we conclude our exploration of types by considering two key readings and exploring some notions of type equivalence.

Related Pages:

- EBoard.
- Reading: Luca Cardelli & Peter Wegner: On Understanding Types.

Overview:

- The CLU Reading.
- Cardelli and Wegner.
- Type Equivalence.
- Structural Equivalence, Revisited.
- Assignment Compatibility.
- Type Coercion.
- Design Issues.

CLU

- My questions
 - What does the year in which the article was published (1977) tell you about the context in which the article was written?
 - What is the thesis of the CLU reading?
 - What seem to be the more interesting type features of CLU?
 - Are there type features that you are surprised have not been adopted (or adopted as quickly) in more modern languages?
- Your questions
 - On pg565, fourth paragraph of second column, Liskov et al says that "the behavior of the data objects must be completely characterized by the set of operations." On pg570, first paragraph of first column, they say that "CLU objects exist independently of procedure activations." My question: The existence of (data) objects seems to be useless without procedure activations since the objects can be manipulated only through the operations provided with them. Hence, should the existence of the objects be dependent on procedure activation? Could you also clarify what they mean by "procedure activation"?
 - One thing in the paper is not clear to me. That is put_n(r,x) method, which the paper says "makes x the n component of the record r (568)". I understand this as that this method can update a component of an object during running time. We have seen the similar thing in Ruby (we can update an object's method). However, CLU does type checking in compiling time,

which sounds contradictory to what I understand. For example, if a program declares an integer as the type of a component and pass the type checking in compiling time, then during running time, it invokes the `put_n(r,x)` to change the type to float. What will happen then?

- The authors said "Thus variables are completely private to the procedure in which they are declared and cannot be accessed or modified by any other procedure". Is that a design choice to limit mistakes of the programmer or was that done for a different reason?

Cardelli and Wegner

- We'll focus on the first half of the paper, rather than on Fun.
- Basic questions
 - What is the difference between untyped, statically typed, and strongly typed?
 - What are the relationships between parametric polymorphism, inclusion polymorphism, ad-hoc polymorphism, and coercion?
 - What are the relationships between typing and polymorphism?
- Some of your questions:
 - Cardelli and Wegner's seem to pretty clearly favor strongly typed languages, and they present monomorphism vs. polymorphism as the only two options for a language. In Automata, the way we look at recursion theory and Turing machines states that anything can be encoded as a string or a number (whichever one we want to work with), including the machines themselves. This sort of encourages polymorphism not by having subtypes, but by making every thing the same type and making it not matter how we treat the data. This approach doesn't seem to jibe with some of the ideas in this article. I realize this is sort of apples and oranges, but I'm curious how it can tie back into what we've read here.
 - Cardelli and Wegner claim that "Static typing is a useful property, but the requirement that all variables and expressions are bound to a type at compile time is sometimes too restrictive." Can you give an example of static typing being too restrictive?
 - Java seems to use mostly ad-hoc polymorphism (overloading, coercion) and some inclusive polymorphism through inheritance. Is that really the case?
 - What's the difference between class parametrization and polymorphism? Is there a formal definition for parametrization?
 - What method of OO code reuse (Polymorphism or inheritance) is preferred?
 - After discussing "Sets" in the previous class, we know that Sets in set theory is a type. The author tells us that "Untyped" means that there is only one type (pg. 472) and that monomorphic also means that there is only one type (pg 475). I am not sure I understand the difference between these terms.

Type Equivalence

- Of the many reasons we use types, one of the most important is in ensuring that objects are only used in "appropriate" places.
 - Arithmetic operations should only be applied to objects on which you can perform arithmetic (and often only to objects of an equivalent type).
 - Values should only be assigned to variables of an equivalent type.
 - The parameters to a function should only be of the appropriate type.

- ...
- Hence, it is important to determine when two types are *equivalent* (in the senses used above).
- Often, type equivalence is attacked from a practical perspective: what equivalencies can we reasonably determine?
 - Recall the definition of `equiv?` in Scheme requires some careful consideration of when functions or pairs are equivalent.
- To answer this question, we'll consider a few basic types and variable declarations. It is often easiest to ask whether we can assign a variable of one type to a variable of another type. Here are some sample types and corresponding variables.

```

type xy = record
    x: integer;
    y: real;
end;
xy_alias = xy;
ab = record
    a: integer;
    b: real;
end;
yx = record
    y: real;
    x: integer;
end;
abc = record
    a: integer;
    b: real;
    c: real;
end;
intrec = record
    x: integer;
end;
iarr = array[1..10] of integer;
iare = array[2..11] of integer;
var
    rec1: xy;
    rec2: xy_alias;
    rec3: ab;
    rec4: abc;
    rec5,rec6: record a: integer; b: real; end;
    rec7: record a: integer; b: real; end;
    rec8: yx;
    rec9: intrec;
    rec10: xy;
    arr1: iarr;
    arr2: iare;
    arr3,arr4: array[1..10] of integer;
    arr5: array[1..10] of integer;

```

- There are a number of strategies one might use to determine equivalence, including
 - structural equivalence,
 - structural equivalence with naming,
 - name equivalence, and
 - declaration equivalence.

- In *structural equivalence*, two types are equivalent if they have the same structure.
 - `rec1` and `rec3` are structurally equivalent as they are both pairs of (integer,real).
 - `rec1` and `rec8` are not structurally equivalent, as they order their elements differently.
- In *structural equivalence under naming*, two types are equivalent if they have the same structure and the named components of each structure have the same names.
 - `rec1` and `rec3` are not structurally equivalent under naming as they name their pairs differently.
 - `rec3` and `rec6` are structurally equivalent under naming.
- In *name equivalence*, two types are equivalent if they are named and have the same name.
 - `rec1` and `rec2` are not name equivalent.
 - `rec1` and `rec10` are name equivalent.
- In *declaration equivalence*, two types are equivalent if they lead back to the same type name. This accommodates variables.

Structural Equivalence, Revisited

- While structural equivalence is attractive because it seems to get closest to “real” equivalence, it may be difficult to determine in the presence of pointers (explicit or implicit) and inclusion.
- As a start, consider

```

type
  alpha = record
    i1: integer;
    i2: integer;
  end;
  aardvark = record
    i1: integer;
    i2: integer;
  end;
  beta = record
    i: integer;
    a: alpha;
  end;
  bandicoot = record
    i: integer;
    a: aardvark;
  end;

```

- `alpha` and `aardvark` are clearly equivalent. Are `beta` and `bandicoot`? How do you tell?
- If you’re willing to follow indirections, consider

```

type
  listp = pointer to list;
  list = record
    i: integer;
    next: listp;
  end;
  lstp = pointer to lst;
  lst = record
    i: integer;
    next: lstp;
  end;

```

- What if indirections aren't direct?

```

type
  gammap = pointer to gamma;
  deltap = pointer to delta;
  gamma = record
    i: integer;
    next: deltap;
  end;
  delta = record
    i: integer;
    next: gammap;
  end;

```

- The cleverer among you might be able to prove that type equivalence (or type constructor equivalence) is uncomputable.

Assignment Compatibility

- A related concept is *assignment compatibility*: can we safely assign a variable of one type to another type?
- This differs from equivalence in that equivalence is bidirectional, but assignment compatibility is unidirectional (just because you can assign a to b, it doesn't mean that you can assign b to a).
- Assignment compatibility often makes the most sense when we think of types as sets of values. You can assign a value in a subset to a variable in a superset, but not vice-versa.

Type Coercion

- Often, we may need to do some *coercion* when working with values of different types. In effect, we change a value of one type to a value of a more general type.
 - When you add the integer 3 and the real number 4.5, we need to coerce 3 to a real.
 - When you print the real number 4.5, you need to coerce it to a string.
 - ...
- *How do we handle such issues?*
 - Disallow them and consider them violations of standard usage.
 - Allow them, but require programmers to explicitly coerce from one type to another.
 - Allow them, and automatically determine an appropriate coercion.
 - ...
- *Are there reasons to make each choice?* Certainly. We'll discuss them.
- *How do we convert, given the underlying representation?*
 - Interpret the bit sequence for one as if it were the bit sequence for the other.
 - Build an equivalent object.
 - ...
- *What kinds of coercions should we allow (explicitly or implicitly)?*
 - "More general" types to "less general" types? (E.g., real to int.)
 - "Less general" types to "more general" types? (E.g., int to real.)
 - What if neither type is more/less general than the other? (E.g., two enumerated types.)

- Should we permit a loss of accuracy? If so, how do we determine that loss of accuracy?
- If multiple coercions are possible, which should we choose? Consider `3 + "4"` in languages that can convert strings to integers.
- *Should we notify the programmer if we coerce?*
- *Should we coerce only simple types, or should we also coerce records?*
 - Consider `ab` and `abc` above.

Design Issues

- Instead of reflecting on “what is” in type systems, we might also reflect on “what might be”.
 - What are some questions we might ask ourselves as we design a type system?
- Some questions have to do with the use of types.
 - Do we type objects in our language?
 - If so, do we type at run time, compile time, both?
 - Are types explicit or implicit?
- Some questions have to do with the “base types” in our system.
 - What primitive types do we provide?
 - What compound types do we provide?
- Some questions might have to do with user-definable types in our system.
 - Can users define new types?
 - Can they define new primitive types?
 - Can they define new compound types?
 - Can they define new type constructors?
- Some might have to do with the interpretation of types.
 - Do we treat types as having fixed-size or arbitrary-size representations?
- Can you think of any others?