CS6110/6116 Lecture 36 Wednesday April 25, 2012

What is a type?

This is a key question for the last quarter of the course. I will post Hoare's paper "Notes on data structuring" which is one of the best intuitive/informal accounts from a computer scientist. In the lectures I am presenting the best precise formal accounts I know which are based on the writings of the logician Per Martin Löf and on the "partial equivalence relation" semantics developed by the PRL group at Cornell, especially Stuart F. Allen, Robert W. Harper, and Nax P. Mendler. I will post my summary of this work from my 1998 article in the Handbook of Proof Theory, "Types in Logic, Mathematics and Programming" (pp. 684–786).

Martin Löf's account of types presents a logically and philosophically deep semantic method. Briefly it is this:

1. types are defined relative to an untyped computation system, specifically an applied λ-calculus with a big step operational semantics derived from a small step one, a la Plotkin (as in course notes and Winskel's book).
2. to define a type A, we give it a canonical name, and we say what its canonical (normalized) elements are as terms in the computation system, $a \in A$.
3. any term a' that reduces to an a in A is also in A.
4. it is also required to say when two terms a_1, a_2 of A are equal by defining the related type $a_1 = a_2$ in A. The only element of this type is atomic, a single element; we denote it as $*$ or axiom (Martin-Löf uses $\star$).

CS6110/6116 Lecture 36 What is a type? continued

Note, based on new work, circa 2011-2012, we will call this treatment of equality types, $x = y$ in A, one-dimensional. This is because there is no structure to the elements, thus the issue of forming the types $x = y$ in $(a = b$ in $A)$, for values of x, y in $a = b$ in A. In higher-dimensional type theory this becomes an issue because evidence in $a = b$ in A can have structure, thereby generating a hierarchy of higher-dimensional types over any $a = b$ in A.

The equality type $x = y$ in A is what allows us to consider types as mathematical objects. Equality is not always defined clearly for types in programming languages, even for finite precision numbers it is a complex idea, and for IEEE floating point numbers, even though equality is defined, it is not advisable to use it in conditional statements. In Lisp equality is well defined on Atoms and Bignums and on lists of these elements.

Equality on function types is extensional in Martin-Löf's 1982 Intuitionistic Type Theory (ITT) and in Computational Type Theory (CTT) implemented in Nuprl and MetaPRL. The definition is

$$f = g \text{ in } A \to B \text{ iff } \forall x{:}A.\ f(x) = g(x) \text{ in } B \quad \text{or}$$

$$f = g \text{ in } x{:}A \to B(x) \text{ iff } \forall x{:}A.\ f(x) = g(x) \text{ in } B(x).$$

In the type theory called the Calculus of Inductive Constructions (CIC), function equality is intensional, that is, the definitions of f and g are the same.

CS6110 Lecture 36 Wed April 25 cont.

Expressing Heyting Arithmetic in type theory.

Kleene Introduction to Metamathematics 1952 presents a version of Heyting Arithmetic (HA) from which one can define Peano Arithmetic (PA) by adding one rule, the "dreaded rule 8.":

8. $\neg\neg A \Rightarrow A$ p. 82 (see Supplemental notes.)

We can express this theory easily in type theory over $\mathbb{N}$ with $x=y$ meaning $x=y$ in $\mathbb{N}$. The normal axioms for $x=y$ as a type handle rule 16 and give us also $a=a$ and $a=b \Rightarrow b=a$, two facts Kleene proves. Let his $a' = S(a)$. We add one rule $a'=b' \Leftrightarrow a=b$ for his 13 and 17.

We define $a+b$ and $a \cdot b$ using λ-terms and the recursion combinator.

$+$ is $\lambda(a,b.\ rec(b; a; u,i.\ S(i)))$

$\cdot$ is $\lambda(a,b.\ rec(b; 0; u,i.\ i+a))$

Instead of the infix $a+b$ and $a \cdot b$ we use $+(a,b)$, $*(a,b)$.

We obtain rules 18, 19 as reductions of $+(a,b)$ and rules 20, 21 as reductions of $*(a,b)$.

To realize the induction rule 13 $P(0) \& \forall x.(P(x) \Rightarrow P(S(x))) \Rightarrow \forall x.P(x)$ we use this realizer:

$$\lambda(h.\ spread(h; p_0, f.\ \lambda(x.\ rec(x; p_0; u,i.f(i)))))$$

Recall

$rec(0; p_0; u,i.h) \downarrow p_0$ $\quad rec(S(n); p_0; u,i.h(u,i)) \downarrow$

$h(n, rec(n; p_0; u,i.h(u,i)))$.

Here are Kleene's rules for Heyting Arithmetic (HA) and Peano Arithmetic (PA). Note PA = HA + Axiom 8°.

Note, Axiom 8° is equivalent to $A \vee \neg A$.

Exercise: Show $(A \vee \neg A) \Leftrightarrow (\neg\neg A \Rightarrow A)$

Kleene Introduction to Metamathematics 1952

GROUP A. Postulates for the predicate calculus.
GROUP A1. Postulates for the propositional calculus.

1a. $A \supset (B \supset A)$.
1b. $(A \supset B) \supset ((A \supset (B \supset C)) \supset (A \supset C))$.
2. $\dfrac{A,\ A \supset B}{B.}$
3. $A \supset (B \supset A \,\&\, B)$.
4a. $A \,\&\, B \supset A$.
4b. $A \,\&\, B \supset B$.
5a. $A \supset A \vee B$.
5b. $B \supset A \vee B$.
6. $(A \supset C) \supset ((B \supset C) \supset (A \vee B \supset C))$.
7. $(A \supset B) \supset ((A \supset \neg B) \supset \neg A)$.
8°. $\neg\neg A \supset A$.

GROUP A2. (Additional) Postulates for the predicate calculus.

9. $\dfrac{C \supset A(x)}{C \supset \forall x A(x).}$
10. $\forall x A(x) \supset A(t)$.
11. $A(t) \supset \exists x A(x)$.
12. $\dfrac{A(x) \supset C}{\exists x A(x) \supset C.}$

GROUP B. (Additional) Postulates for number theory.

13. $A(0) \,\&\, \forall x(A(x) \supset A(x')) \supset A(x)$.
14. $a'=b' \supset a=b$.
15. $\neg a'=0$.
16. $a=b \supset (a=c \supset b=c)$.
17. $a=b \supset a'=b'$.
18. $a+0=a$.
19. $a+b'=(a+b)'$.
20. $a \cdot 0=0$.
21. $a \cdot b'=a \cdot b+a$.

(The reason for writing "°" on Postulate 8 will be given in § 23.)

One may verify that 14—21 are formulas; and that 1—13 (or in the case of 2, 9 and 12, the expression(s) above, and the expression below, the line) are formulas, for each choice of the A, B, C, or x, A(x), C, t, subject to the stipulations given at the head of the postulate list.

The class of 'axioms' is defined thus. A formula is an *axiom*, if it has one of the forms 1a, 1b, 3—8, 10, 11, 13 or if it is one of the formulas 14—21.

The relation of 'immediate consequence' is defined thus. A formula is an *immediate consequence* of one or two other formulas, if it has the form shown below the line, while the other(s) have the form(s) shown above the line, in 2, 9 or 12.

This is the basic metamathematical definition corresponding to Postulates 2, 9 and 12, but we shall restate it with additional terminology

```
∀n:ℕ.  ∃r:ℕ.  r² ≤ n < (r+1)²
BY  allR
  n:ℕ
  ⊢ ∃r:ℕ.  r² ≤ n < (r+1)²
  BY  NatInd 1
  .....basecase.....
      ⊢ ∃r:ℕ.  r² ≤ 0 < (r+1)²
  √  BY  existsR ⌈0⌉ THEN Auto
  .....upcase.....
      i:ℕ+,  r:ℕ,  r² ≤ i-1 < (r+1)²
      ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      BY  Decide ⌈(r+1)² ≤ i⌉ THEN Auto
      .....Case 1.....
          i:ℕ+,  r:ℕ,  r² ≤ i-1 < (r+1)²,  (r+1)² ≤ i
          ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      √  BY  existsR ⌈r+1⌉ THEN Auto'
      .....Case 2.....
          i:ℕ+,  r:ℕ,  r² ≤ i-1 < (r+1)²,  ¬((r+1)² ≤ i)
          ⊢ ∃r:ℕ.  r² ≤ i < (r+1)²
      √  BY  existsR ⌈r⌉ THEN Auto
```

Figure A.1: Proof of the Specification Theorem using Standard Induction.

```
let rec sqrt i
= if i=0 then <0,pf_0>
  else let <r,pf_{i-1}>  = sqrt (i-1)
       in
         if (r+1)²≤n  then
<r+1,pf_i>
         else <r,pf_i'>
```

```
let rec sqrt i
= if i=0 then 0
  else let r = sqrt (i-1)
       in
         if (r+1)²≤n  then r+1
         else r
```

Using standard conversion mechanisms, Nuprl can then transform the algorithm into any programming language that supports recursive definition and export it to the corresponding programming environment. As this makes little sense for algorithms containing proof terms, we only convert the algorithm on the right. A conversion into SML, for instance, yields the following program.

```
fun sqrt n = if n=0 then 0
             else let val r = sqrt (n-1)
                  in
                    if n < (r+1)^2  then r
                    else r+1
                  end
```

A.2 Deriving an Algorithm that runs in $\mathcal{O}(\sqrt{n})$

Due to the use of standard induction on the input variable, the algorithm derived in the previous section is linear in the size of the input n, which is reduced by 1 in each step. Obviously, this is not the most efficient way to compute an integer square root. In the following we will derive more efficient algorithms by proving $\forall n\, \exists r\ r^2 \leq n \wedge n < (r+1)^2$ in a different way. These proof, however, will have to rely on more complex induction schemes to ensure a more efficient computation.