CS6110/6116 Lecture 37 Friday April 27, 2012

The Birth of Constructive Type Theories

Heyting Arithmetic (HA) continued.

See the handout (also on Lecture 36 Web page) for a proof of the integer square root program by standard induction. Note that simple arithmetic reasoning about equality, $<$, $\leq$, $+$, $*$ is accomplished by a decision procedure called Arith used in Nuprl. Note that we can prove that $<$, $\leq$, $=$ are decidable relations, e.g. $\forall x, y: \mathbb{N}. (x = y \vee \neg(x=y))$, $\forall x,y:\mathbb{N}.(x<y \vee \neg(x<y))$, etc. Note, $<$, $\leq$ are defined, e.g. $x \leq y$ iff $\exists z: \mathbb{N}.(x+z = y \text{ in } \mathbb{N}.)$.

Notice that we obtain a computable function for root, but it is slow, order of the value of the input, thus exponential in the length of the input. As an exercise, you will look at a faster way to compute the root using efficient induction. What makes this work is that we prove using ordinary induction that the efficient realizer belongs to the efficient induction type.

One of the key features of constructive type theory in the style of Martin-Löf is that typing judgments can be proved rather than being decided by a type checker. This provides a way of creating efficient realizers.

CS 6110/6116 Lecture 37

HA^ω Heyting Arithmetic of Higher-Order

The types over $\mathbb{N}$ in dependent type theory include quantification over functions. Here is an interesting general theorem, a special case of the Axiom of Choice. Let $R: A \times B \to Prop$ (we discuss Prop below).

$*$ $\forall x{:}A.\ \exists y{:}B.\ R(x,y) \Rightarrow \exists f{:}A \to B.\ \forall x{:}A.\ R(x, f(x))$

We can prove this in HA^ω for A, B the type $\mathbb{N}$, and it holds generally for constructive type theory as we discuss below. We can actually "see" the realizer without resorting to proof, using the techniques we learned for FOL. Here is the realizer: $\lambda(h.\ \langle \lambda(x.\ spread(h(x); b, p.\ b),\ \lambda(x.\ spread(h(x); b, p.\ p)) \rangle)$

The justification that $\lambda(x.spread(h(x); b, p.\ b))$ is a proper function in $A \to B$ depends on the fact that

$a_1 = a_2$ in A implies $f(a_1) = f(a_2)$ in B,

that is $h(a_1)$ and $h(a_2)$ produce b_1, b_2 respectively and $b_1 = b_2$ in B. This is a property of dependent function types, that $h(a_1) = h(a_2)$. Suppose $f(a_1) = \langle b_1, p_1 \rangle$ and $f(a_2) = \langle b_2, p_2 \rangle$. Then
in $\exists y{:}B.\ R(a_1, y)$ in $\exists y{:}B.\ R(a_2, y)$

by the definition of equality over $\exists y.\ R(x,y)$ we have

$b_1 = b_2$ in B and $R(a_1, b_1) = R(a_2, b_2)$.

We look at these equality issues more carefully next and discuss Prop as a type.

Equality relations

We have seen that do define a mathematically meaningful type it is necessary to define equality on its elements. We have studied in Lecture 36 and here these equalities

Type	Equality on the type
$\mathbb{N}$	$n = m$ in $\mathbb{N}$
$n = m$ in $\mathbb{N}$	$e_1 = e_2$ in $n = m$ in $\mathbb{N}$ this is trivial (dimension 1)
$x: A \to B(x)$ (special case $A \to B$)	$f = g$ in $x: A \to B(x)$ extensional equality
$f = g$ in $x: A \to B(x)$	$e_1 = e_2$ in $f = g$ in $x: A \to B(x)$ trivial
$x: A \times B(x)$ (special case is $A \times B$)	$p = q$ in $x: A \times B(x)$ suppose $p = \langle p_1, p_2 \rangle$ $q = \langle q_1, q_2 \rangle$ then $p_1 = q_1$ in A $p_2 = q_2$ in $B(p_1)$
{ we now skip the equality types as types case }	
$A + B$	$u_1 = u_2$ in $A + B$ $u_1 = \text{inl}(u_1')$, $u_2 = \text{inl}(u_2')$ and $u_1' = u_2'$ in A or $u_1 = \text{inr}(u_1')$, $u_2 = \text{inr}(u_2')$ and $u_1' = u_2'$ in B

CS 6110/6116

Lecture 37 continued

The Big Bang

Constructive type theory achieves considerable expressive power by including two more fundamental ideas:

recursive types

types as values in higher types (universes).

Martin-Löf's Intuitionistic Type Theory (ITT) includes one recursive type, the W-types for well-orderings, also called Brouwer ordinals. The Nuprl type theory (Computational Type theory - CTT) adds recursive types of which W-types are a special case. These were added in Constable & Mendler 1985. In 1987, Nax Mendler proved that these types are consistent using an extension of Girard's method. This is a major result that also grounds the Coq type theory based on the Calculus of Inductive Constructions, CIC, circa 1993.

Recursive Types

A very simple example of a recursive type is the definition of lists of elements of type A, List(A). The elements are nil, $\langle a, nil \rangle$, $\langle a, \langle b, nil \rangle \rangle$, $\langle a, \langle b, \langle c, nil \rangle \rangle \rangle$, etc. The definition is written

$List(A) = rec(L.\ Unit + A \times L)$ where $Unit = \{\bullet\}$

and we take $nil := \bullet$ Martin-Löf calls Unit $\mathbb{N}_1 = \{0_1\}$ and $\mathbb{B} = \mathbb{N}_2$ where $\mathbb{N}_2 = \{0_2, 1_2\}$. In general he defines $\mathbb{N}_i = \{0_i, 1_i, \ldots (i-1)_i\}$.

CS6110/6116

Lecture 37 continued

List(A) continued. The idea for how to construct elements of List(A) is very intuitive. We build elements outside-in, or top down, by "unrolling the definition" in a refinement style proof. Here is an example.

$\vdash$ rec(L. Unit + A × L) by intro
  $\vdash$ Unit + A × rec(L.—) by inl(*)
    $\vdash$ Unit by *

$\vdash$ rec(L. Unit + A × L) by intro — inr(⟨a_1, inl(*)⟩)
  $\vdash$ Unit + A × L by inr(⟨a_1, ⟩)
    $\vdash$ A × rec(L.—) by
      $\vdash$ A by a_1
      $\vdash$ rec(L.—) by intro
      $\vdash$ Unit + rec(—) by inl( )
        $\vdash$ Unit by *

The induction principle defines a method of computing on lists. It is the realizer for this induction principle

$P(nil) \;\&\; \forall l:List(A).\, \forall a:A.\, (P(l) \Rightarrow P(\langle a, l\rangle)) \Rightarrow \forall x:List(A).\, P(x)$.

the realizer is list-ind(l; t_b; $x, y, z.\, t_u$) it satisfies

$H \vdash$ list-ind(l; t_b; $x, y.\, t_u$) $\in$
$H \vdash$ list-ind(l; t_b; $x, y, z.\, t_u$) $\in T[l/z]$
  $\vdash l \in List(A)$
  $\vdash t_b \in T[nil/z]$

$H, u:A, v:List(A), w:T[v/z] \vdash t_u[u, v, w/x, y, z] \in T[inr(\langle u, v\rangle)/z]$

This rule tells us what the reduction rule is for the realizer. See also the supplemental notes for a similar account.

CS6110/6116

Lecture 37 continued

Universes – types whose elements are types, i.e. types as values

The other major concept that causes type theory to "inflate" to a theory rich enough to express constructive mathematics as well as the theory of computation is the idea that types are values. The programming language Russell, designed and implemented at Cornell by Alan Demers and Jim Donahue, also treated types as values.

We have introduced canonical names for all of the types we have discussed. The computation system provides the non-canonical names. The notion of equality Martin-Löf used in ITT is very simple, basically he required the minimal equality. For example

$A \to B = A' \to B'$ in $Type_1$ iff $A = A'$ in $Type_1$ and $B = B'$ in $Type_1$.

$x{:}A \to B(x) = x{:}A' \to B'(x)$ iff $A = A'$ and for every $a \in A$, $B(a) = B'(a)$ in $Type_1$.

We have now presented the basic ideas behind ITT and Martin-Löf's ground breaking paper should be accessible. It is posted with this lecture.