CS6110/6116 Type Theory Notes Lecture 38

These are supplemental notes on type theory that summarize several points considered in Lecture 38 and in earlier lectures. They also put into writing some motivational comments made during lectures on constructive type theory.

I included more material on subtyping, partial types, and unsolvable problems than covered during Lecture 38.

Lecture Plan

1. Setting the stage for type theory
2. Programming languages and type theory
3. Foundations – unsolvable problems

Setting the Stage - 1

Principia Mathematica (PM) was designed as a foundational framework for mathematics based on Russell's type theory circa 1908.

It did not work out that way directly, but *PM* has had a large impact on logic and computer science which is exerting an increasing impact on mathematics and creating a foundation for computational mathematics.

Programming Languages and Logics

From the critical historical juncture circa 1908 there was a split in the genotype of mathematics: sets (Zermelo) versus types (Russell).

We now know about sets from mathematics courses and about types from programming courses.

Types also appear in basic CS theory courses such as Algorithms and Data Structures.

Setting the Stage - 2

A major intellectual accomplishment of computer science is the demonstration that computers can automate many high level mental functions, especially those involved in mathematical reasoning. Open problems have been solved this way, two at Cornell, Howe and Murthy, both major accomplishments of computer assisted mathematical reasoning.

Selected Notable Examples

- Four Color Theorem formalization – Gonthier
- Prime Number Theorem – Harrison, Avigad
- Kepler Conjecture Investigation – Halles and HOL-Light team
- Girard's paradox -- Howe
- Constructive Higman's Lemma – Murthy
- Kruskal's Theorem – Seisenberger
- POPLMark Challenge – Coq,Twelf,HOL
- Paris driverless Metro line 14 – Abrial, B-tool
- Mizar's Journal of Formalized Mathematics

Setting the Stage -2 continued

The accomplishments of computer assisted reasoning have added a new dimension to mathematics ---- and to science generally.

This dimension includes a growing library of formalized mathematics -- directly advancing the vision of *PM*. Perhaps surprisingly, the dominant theory being used is type theory, influenced by *PM* and the work of computer scientist such as Hoare (e.g. "Notes on data structuring").

Setting the Stage - 3

It is possible that type theory will remain the dominant framework for formalized mathematics. It was used in all the Automath systems and is now used in Coq, HOL, Isabelle HOL, HOL-Light, MetaPRL, Minlog, Nuprl, and PVS, all important proof assistants whose native logical language is type theory.

Moreover, the use of proof assistants could change the framework used for the foundations of mathematics because mathematicians are also very concerned about correctness based on experience with "major results" that have turned out to be false.

Homotopy Type Theory

Field's medalist mathematician Vladimir Voevodsky is organizing meetings on the use of constructive type theory as a foundation for algebraic topology especially homotopy theory, see the Institute of Advanced Study web page for 2012 – 2013 academic year. He is using Coq to prove some of his results for fear of errors, and he is writing notes on type systems from his viewpoint (available on-line).

Lecture Plan

1. Setting the stage
2. Programming languages and type theory
3. Foundations – unsolvable problems

Increasing Richness of Types in Programming Languages

FORTRAN started with simple type distinctions, fixed and floating point numbers distinguished by the alphabetical range of the variable names, e.g. k,l,m,n for fixed versus x,y,z for floating point.

From that meager beginning we have seen a progressive enrichment of type systems from Algol60, Pascal, Algol 68, PL1, C++, Java, ML, F#, etc.

Standard PL Types

The types we now expect:

Product	$A \times B$	<a,b> ordered pairs
Record	$\{a_1:A_1;\ldots;a_n:A_n\}$	maps a_i into A_i
Union	$A+B$	inl(a), inr(b) inject
Function	$A \rightarrow B$	λ(x.exp) functions
Recursive	$\mu X.F(X)$	e.g. lists,trees, etc.

Richer Type Systems

These ML like types are not the ultimate. There is very active research on new type systems, especially dependent types. One wonders whether there is a natural limit to these extensions. We will look at this question shortly.

Teaching Type Theory

Tony Hoare's 1972 paper Notes on Data Structuring stressed the idea that reasoning about data types was a key part of programming logics, along with rules for control structures such as the famous Hoare while rule.

Proof Assistants for Programming Logics

Some of the first proof assistants in CS were for Programming Logics, such as the Stanford Pascal Verifier, Gypsy, Cornell's PLCV, Stanford LCF, Edinburgh LCF.

These logics include programs. They provide computational content to certain assertions. For example, the Hoare While Rule is an induction principle whose computational content is given by a while program.

Is there a limit to PL type richness?

We have seen programming languages use richer and richer static type systems, is there a limit to what is required?

In the programming logics we need types adequate to mathematics, how much richer is this type system; might it be a limit? Note, we also use logical operators in the types, e.g. {x:A| P(x)}, refinement types. So logic and programming types are intimately connected.

A Very Rich Type System

Certain natural generalizations of the standard types provide a potential limit point because they also characterize the logical operators. This is a surprise -- except to constructive mathematicians and the intuitionists starting with L.E.J. Brouwer.

The semantics for logic in terms of types is called the Brouwer/Heyting/Kolmogorov (BHK) semantics. Also called the propositions as types principle or the proofs as terms principle or realizability semantics. (A much less appropriate name is the Curry-Howard isomorphism.)

Dependent Types

Dependent Product

$y{:}A \times B(y)$ $\quad$ <a,b> with a in A, b in B(a)

Dependent Records

$\{x_1{:}A_1;\ x_2{:}A_1(x_1); x_3{:}A(x_1,x_2); \ldots; x_n{:}A_n(x_1,\ldots,x_{n-1})\}$

functions r from Labels $\{x_1,\ldots,x_n\}$ to values where $r(x_1) \varepsilon\ A_1$, $r(x_2) \varepsilon A_2(r(x_1))$, etc.

Dependent Functions (special case of dep records)

$x{:}A_1 \rightarrow A_2(x)$ $\quad$ $\lambda(x.exp)$ functions

Subtyping and Polymorphism

There is a primitive subtyping relation in CTT.

$A \sqsubseteq B$ means that the elements of *A* are elements of *B* and (a=b in *A*) implies (a=b in *B*). Here are some basic facts about subtyping:

$$\{x : Z \mid x > 0\} \sqsubseteq Z$$
$$(A \sqsubseteq A' \,\&\, B \sqsubseteq B') \Rightarrow A \times B \sqsubseteq A' \times B'$$
$$(A \sqsubseteq A' \,\&\, B \sqsubseteq B') \Rightarrow A + B \sqsubseteq A' + B'$$
$$(A \sqsubseteq A' \,\&\, B \sqsubseteq B') \Rightarrow A' \to B \sqsubseteq A \to B'$$

Record Types and Inheritance

We can define algebraic structures as records. For example, a monoid on carrier S is a record type over S with two components, an associative operator and an identity:

Monoid = { op: S x S -> S; id: S}.

A group extends this record type on S by including an inverse operation.

Group = { op: S x S -> S; id: S; inv:S ->S}

A *Group* is a subtype of a *Monoid* as we show next.

$$Group \sqsubseteq Monoid$$

Groups and Monoids as Records

The basic idea is that the elements of a record type are functions from the field selectors names, e.g. {op,id,inv} to elements of types assigned to them by a mapping called a signature, Sig:{op,id,inv} -> Type. Here are the mappings for a Group over the carrier S.

Sig(op)= SxS->S, Sig(id)=S, Sig(inv)=S->S

Monoid and Group are these dependent function spaces, Group a subtype of Monoid.

$$i:\{op,id,inv\} \to Sig(i) \sqsubseteq i:\{op,id\} \to Sig(i)$$

Inheritance among Algebraic Structures

Notice that the subtyping of algebraic structures depends on the function space subtyping, namely

$$i:\{op,id,inv\} \to Sig(i) \sqsubseteq i:\{op,id\} \to Sig(i)$$

is an instance of the general relation:

$$(A \sqsubseteq A' \,\&\, B \sqsubseteq B') \Rightarrow A' \to B \sqsubseteq A \to B'$$

Intersection and Top Types

We can build records using a binary intersection of types, $A \bigcap B$

These are the elements in both types A and B with x=y in the intersection iff x=y in A & x=y in B.

Top is the type of all closed terms with the trivial equality, x=y for all x, y in Top. Note for any type A, we have $A \sqsubseteq Top$ and $A \bigcap Top = A$.

Lecture Plan

1. Programming languages and logics
2. Programming languages and type theory
3. Foundations – unsolvable problems

Foundational Topics – Partial Types

We look at types denoted Ā derived from type A. Elements of Ā are partial, and we call types with partial elements partial types or "bar types." They have as elements terms which do not converge, like Ω, and terms whose convergence is unknown, yet for which we know that if they converge, they converge to a value in the underlying type A.

Partial Functions

The concept of a partial function is an example of how challenging it is to include all computation in the object theory. It is also key to including unsolvability results with a minimum effort; the halting problem and related concepts are fundamentally about whether computations converge, and in type theory this is the essence of partiality. For example, we do not know that the 3x+1 function belongs to the type N -> N.

Partial Functions

We do however know that the 3x+1 function, call it f, is a partial function from numbers to numbers, thus for any n, f(n) is a number if it converges (halts).

In CTT we say that a value a belongs to the bar type $\bar{A}$ provided that it belongs to A if it converges. So f belongs to $A \rightarrow \bar{A}$ for $\bar{A} = N$.

Unsolvable Problems

It is remarkable that we can prove that there is no function in CTT that can solve the convergence problem for elements of basic bar types.

We will show this for non empty type $\bar{A}$ with element $\bar{a}$ that converges in A for basic types such as Z, N, List(A), etc. We rely on the typing that if f maps $\bar{A}$ to $\bar{A}$, then fix(f) is in $\bar{A}$. This is true even for the bar type over N. For example, fix(λx.x), belongs to any $\bar{A}$. We'll see more interesting examples.

Unsolvable Problems

Suppose there is a function h that decides halting on some Ā. Define the following element of Ā:

d = fix(λ(x. if h(x) then Ω else ā fi))

where Ω is a diverging element, say fix(λx.x).

Now we ask for the value of h(d) and find a contradiction as follows:

Generalized Halting Problem

Suppose that h(d) = *t,* then d converges, but according to its definition, the result is the diverging computation Ω because by computing the fix term for one step, we reduce

d = fix(λ(x. if h(x) then Ω else ā fi))

to d = if h(d) then Ω else ā fi , then to Ω.

If hd(d) = *f,* then we see that d converges to ā.

Why is this result noteworthy?

First notice that the result applies to any purported halting function h. In classical mathematics, there surely is a noncomputable function to decide halting.

Moreover the standard way to present unsolvability constructively is to model Turing machines and prove that no Turing computable function can solve the halting problem. But this result says that no function can solve it.

Why is this result noteworthy?

Another reason to take note of this result is that it is so simple, about the simplest unsolvability result around -- no indexings, no reflection, simple realistic computing model. It also gives a form of incompleteness (not discussed here).

Perhaps the main reason to take note is that this result contradicts classical mathematics and shows that CTT with bar types is not consistent with the law of excluded middle, as the rest of type theory is.