CS 6110 Review for Final Exam

Topics week by week

1. Operational semantics of functional programming languages and main course themes, e.g. prog languages as mathematical objects.
2. The λ-calculus, abstract syntax, evaluators, primitive recursion Classic ML syntax.
3. Evaluation with environments, capsules, closures, primitive recursion and logic (Recursive Number Theory).
4. Comparing evaluators, continuations, continuations and primitive recursion
5. Simple imperative programming languages, IMP, program schemas, while schemes, the Loop Language, induction on computations
6. Axiomatic semantics, partial correctness, Hoare axioms and rules David Gries on proving programs correct.
7. Revisiting continuations, relationship to Loop language, defunctionalization and state machine evaluators
8. Gödel's system T, Taits method (logical relations), strong normalization, Prelim review.
9. Typed λ-calculus, recursive procedures in IMP, Prelim.
10. Applied λ-calculus (×, +, void), compiling with continuations (guest lecture by Andrew Myers).
11. Propositions-as-types and typed λ-calculus, proofs as evidence, First-Order Logic and dependent types.
12. Dependent types, specification languages
13. Type theory, what is a type
14. Distributed computing, message automata, event logic.

CS6110 Review for Final Exam

I will ask questions about bar types (partial types) and you should look carefully at the Lecture 38 notes, e.g.

defining the $3x+1$ function on $\bar{N}$ using fixed points.

knowing the halting problem argument

computing with $\bar{N}$

These partial types are a way to do "domain theory" and semantics of IMP.

You should review IMP, know the structured operational semantics, compare IMP to the λ-calculus, applied lambda calculus and to the Loop language.

There will be stress on type theory especially dependent types, partial types, and the List(A) and Tree(A) types. You should know propositions-as-types thoroughly.

There will be several short answer questions and comparisons between imperative and functional languages. There will be a little bit on event structures, especially causal order.

There will be a bit about "Turing completeness" of programming languages — a key property of the λ-calculus. You should compare the typed and untyped λ-calculus.

Computability in CSA

Here is how to define general recursive functions. Consider the 3x+1 function with natural number inputs.

```
f(x) = if x=0 then 1
            else if even(x) then f(x/2)
                  else f(3x+1)
                  fi
            fi
```

Using Lambda Notation

```
f = λ(x. if x=0 then 1
          else if even(x) then f(x/2)
                else f(3x+1))
```

Here is a related term with function input f

```
λ(f. λ(x. if x=0 then 1
             else if even(x) then f(x/2)
                   else f(3x+1)))
```

The recursive function is computed using this term.

Non-terminating Computations

CTT defines all general recursive functions, hence non-terminating ones such as this

fix(λ(x.x))

which in one reduction step reduces to itself!

This system of computation is a simple functional programming language.

Unsolvable Problems

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

d = fix(λ(x. if h(x) then ↑ else 0 fi))

where ↑ is a diverging term, say fix(λ(x.x)).

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

CS 611 Problem Set 8 *Review & Study Guide* Fall 1993 Due: Wed., Dec. 8

If you want feedback, try these before December 8.

Problem 1: Write an ADT for the concept of a cpo and give two distinct examples.

Problem 2: The Fixed Point Induction Rule is this for *admissable* P:

$$\frac{P(\perp) \;\; P(f) \Rightarrow P(F(f))}{P(fix(f.F(f)))}$$

Use the Fixed Point Induction rule to prove that the $3x+1$ function has the value 0 on all inputs. Here is the function, the rule is in the handout on LCF attached:

$$f(x) = \mathit{if}\ x = 1\ \ \mathit{then}\ 0\ \ \mathit{else\ if}\ \mathit{even}(x)\ \ \mathit{then}\ f(x/2)\ \ \mathit{else}\ f(3x+1)$$

Problem 3: We can also use the Fixed Point Induction Rule to prove this "fact" about the factorial function:

$$\exists x : N.\ \mathit{fact}(x) = \perp .$$

Note: $\mathit{fact} = \ fix(f.\ Fact(f))$.

Here is the proof, what is wrong with it?

Let $P(f)$ be $\exists x{:}N.\ f(x) = \perp$.
Notice $P(\lambda(x.\ \perp))$ is true.
Assume $P(f)$ is true, then
Let Fact$(f) = \mathit{if}\ x = 0\ \ \mathit{then}\ 1\ \mathit{else}\ x \star f(x-1)$ and
Suppose $f(b) = \perp$, then clearly
    Fact$(f)(b+1) = \perp$.
Hence $\exists x{:}N.$ Fact$(f)(x) = \perp$
By fixed point induction,
    $\exists x : N.\ fix(f.Fact)(x) = \perp$.

Problem 4: PCF with **Y** solve Gunter 4.9

Problem 5: Scott Topology solve Gunter 4.22

Problem 6: Halting Problem for PCF

Show without recourse to diagonalization that the predicate is-defined (e) for e a PCF expression which is true iff $e \neq \perp$ is *not* definable in PCF.

$$
\begin{array}{l}
\mathit{LIST} \mathrel{\hat{=}} A{:}\mathit{Type} \rightarrow \\
\quad \mathit{List}{:}\mathit{Type} \\
\quad \#\ \mathit{Nil}{:}\mathit{List} \\
\quad \#\ \mathit{Cons} : A \times \mathit{List} \rightarrow \mathit{List} \\
\quad \#\ \mathit{Lind}{:}\ P{:}(\mathit{List} \rightarrow \mathit{Type}) \rightarrow l{:}\mathit{List} \rightarrow P(\mathit{Nil}) \\
\qquad \rightarrow ((h{:}A \times t{:}\mathit{List} \times P(t)) \rightarrow P(\mathit{Cons}(h,t))) \rightarrow P(l) \\
\quad \#\ P{:}(\mathit{List} \rightarrow \mathit{Type}) \rightarrow l{:}\mathit{List} \rightarrow g{:}P(\mathit{Nil}) \rightarrow i{:}((h{:}A \times t{:}\mathit{List} \times P(t)) \rightarrow P(\mathit{Cons}(h,t))) \\
\qquad\qquad \mathit{Lind}(P)(\mathit{Nil})(g)(i) = g \\
\qquad\qquad \wedge\ \forall h{:}A.\, \forall t{:}\mathit{List}.\, \mathit{Lind}(P)(\mathit{Cons}(h,t))(g)(i) = i(h,t,\mathit{Lind}(P)(t)(g)(i))
\end{array}
$$

$$
\begin{array}{l}
\mathit{MSET} \mathrel{\hat{=}} A{:}\mathit{Type} \rightarrow \\
\quad \mathit{MSet}{:}\mathit{Type} \\
\quad \#\ \mathit{Empty}{:}\mathit{MSet} \\
\quad \#\ \mathit{Add} : A \times \mathit{MSet} \rightarrow \mathit{MSet} \\
\quad \#\ \forall a,b{:}A.\, \forall s{:}\mathit{MSet}.\ \mathit{Add}(a,\mathit{Add}(b,s)) = \mathit{Add}(b,\mathit{Add}(a,s)) \\
\quad \#\ \mathit{Mind}{:}\ P{:}(\mathit{MSet} \rightarrow \mathit{Type}) \rightarrow l{:}\mathit{MSet} \rightarrow P(\mathit{Empty}) \\
\qquad \rightarrow \{i{:}((a{:}A \times s{:}\mathit{MSet} \times P(s)) \rightarrow P(\mathit{Add}(a,s)))\ | \\
\qquad\quad \forall a,a'{:}A.\, \forall s{:}\mathit{MSet}.\, \forall p{:}P(s).\ i(a,\mathit{Add}(a',s),p) = i(a',\mathit{Add}(a,s),p)\} \\
\qquad \rightarrow P(l) \\
\quad \#\ P{:}(\mathit{MSet} \rightarrow \mathit{Type}) \rightarrow l{:}\mathit{MSet} \rightarrow g{:}P(\mathit{Empty}) \\
\qquad \rightarrow i{:}\{i{:}((a{:}A \times s{:}\mathit{MSet} \times P(s)) \rightarrow P(\mathit{Add}(a,s)))\ | \\
\qquad\quad \forall a,a'{:}A.\, \forall s{:}\mathit{MSet}.\, \forall p{:}P(s).\ i(a,\mathit{Add}(a',s),p) = i(a',\mathit{Add}(a,s),p)\} \\
\qquad\qquad \mathit{Mind}(P)(\mathit{Empty})(g)(i) = g \\
\qquad\qquad \wedge\ \forall a{:}A.\, \forall s{:}\mathit{MSet}.\ \mathit{Mind}(P)(\mathit{Add}(a,s))(g)(i) = i(a,s,\mathit{Mind}(P)(s)(g)(i))
\end{array}
$$

Figure 2: List and Multi-set ADT

data-type definition into a single type and the practical advantage of allowing parameterization of one ADT by another. It can also be seen as a natural generalization of signatures in ML, Extended ML [40], and other languages implementing data-abstraction using dependent types. This is the approach we shall take. One complication is that such modularization requires predicative quantification and hence a type hierarchy. For simplicity, we shall use the first universe (which we call *Type*) as the type of ADT parameters and set carriers.

Figure 2 contains two examples of ADTs: parameterized lists and finite multi-sets (sometimes called "bags"). Members of the LIST type are functions: when applied to a parameter type A, they return a tuple in which the first projection, the *carrier* of the ADT, is the type of lists over A whose members are built from the functions inhabiting the Nil and Cons projections. The multi-set ADT is similar except for the defined carrier equality and the elimination and computation rules which are modified to respect this equality (more will be said on this below). Some notation should be explained. First $x : A \# B$ denotes a Σ-type built from the types A and B. Free occurrences of x become bound in B. Sometimes, to emphasize its logical significance, we abbreviate this to $\exists x : A.\, B$, or conjunction (when x is not free in B) $A \wedge B$, or even pairing $A \times B$. Similarly $x : A \rightarrow B$ represents the Π-type built from A and B and may be alternatively displayed as a $\forall x : A.\, B$ or $A \Rightarrow B$. To axiomatize equalities we make