Advanced Progamming Languages	**Lecture 32**
CS 6110 Spring 2015	**Wed. April 15, 2015**

Lecture 32

Topics

1. The Programming Logic idea is alive and well as Sarah Loos mentioned in discussing courses using key based programming logic – built on Java. F* and Lean might be another example, Idris and Haskell.
2. PS4 question on *inclusive* predicates.
3. Integrating programming and logic is a very rich idea, but you need to bet on a programming language.
4. We will look at the *Loop language* of Meyer & Ritchie.
5. PLCV examples continued, page 86, `SQ: PROCEUDRE(N, SUM)`

Fixed Point Induction

$$\frac{\vdash W[\bot/f] \qquad W \vdash W(F(f))/f}{\vdash W[Fix(F)/f]}$$

W admits *ind*.

D is a cpo with $\bot$.

$F : D \to D$ is continuous.

P is *inclusive* iff for all ω-chains $d_0 \sqsubseteq d_1 \sqsubseteq ... \in D$ if $d_n \in P$ for all $n : \mathbb{N}$, then $\bigsqcup_n d_n$, the lub of d_i, is in P.

PS4 problem variations

Find a non-inclusive predicate over $(\overline{\mathbb{N}} \to \overline{\mathbb{N}}) \to (\overline{\mathbb{N}} \to \overline{\mathbb{N}})$ where $\bot = (f(x) = \bot)$. $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all x.

We get function chians from starting with

$f_0 = f(x) = \bot$ for all x.

$f_{i+1} = F(f_i) \qquad f_\omega = \bigsqcup f_n$, the least upper bound of f_n.

```
01  SQ: PROCEDURE(N, SUM);
02  DECLARE N FIXED /*/ READONLY */;
03  DECLARE SUM FIXED;
04  /*/ ASSUME N >= 1;
05      ATTAIN 2*SUM = N * (N+1);
06   */
06  DECLARE I FIXED;
07  /*/ 2*0 = (1-1)*1;
08      ~(1>N) BY ARITH, N>=1; */
09
10  SUM = 0;
11
12  /*/ 2*SUM = (1-1)*1; */
13  DO I = 1 TO N BY 1;
14       /*/ ASSUME P: 2*SUM = (I-1)*I;
15           2*(SUM+I) = I*(I+1) BY ARITH, P, +, 2*I=2*I; */
16
17       SUM = SUM + I;
18
19       /*/ 2 * SUM = I * (I+1);
20           2 * SUM = (I+1-1) * (I+1); */
21  END;
22  /*/ 2*SUM = (N+1-1) * (N+1); */
23  RETURN;
24  END SQ;
```

Figure 19 Computing sum of the first N integers

the loop body is executed. Thus, after the loop is finished executing, P{F+1//I} will be true.

The assumption is made in the above discussion that the loop body is executed. When S>F, the loop body is not executed at all. In that case, the assertion we will conclude after the execution of the loop had better be true right away. Thus, we are obliged to prove S>F => P{F+1//I} before the execution of the loop. Often we know that the loop body will be executed at least once. In that case, we can prove ~(S>F), and the implication we need to prove is vacuously true, and will follow from automatic rules.

Let us see how all of this works in a simple example. Figure 19 contains a program that uses a DO INDEX statement to compute the sum of the first N integers. That is, the program sets S = 1 + 2 + ... N. It is a fact of algebra that the sum of the first N integers is (N*(N+1))/2. Let us see how we can use the DO INDEX rule to prove this fact. (We use the ATTAIN 2*SUM = N * (N+1) rather than SUM = (N*(N+1))/2 to avoid the complexities of reasoning about division.)

The first thing to get out of the way is to show that the loop works properly if the body is not executed. We don't have to worry about that problem in this example, since we are assuming that N >= 1 on line 04. Thus, line 08 takes care of the S>F => P{F+1//I} part of the template.

is indicated by horizontal and vertical dots as with the equality templates. The only difference is that when a group of templates has the same hypothesis, as in the MOD templates, the hypothesis appears in the template only once. It should be understood that whenever you use any of the three MOD templates, that the hypothesis ~(J = 0) should be the conclusion of a previous proof line. (Note: although the given rules completely characterize the division function, there are many additional useful conclusions which may be asserted without laboriously deriving them from these rules. See section 4.12 for the full division template.)

Logical Connectives

The operators &, |, =>, <=>, ~, and the quantifiers are called the "logical connectives." For each logical connective, there is an "introduction rule" and an "elimination rule." The introduction rule for a connective will have an assertion using that connective as a conclusion. Conversely, the elimination rule for a connective will have an assertion using that connective as a hypothesis.

Here are the templates for some of the simpler introduction and elimination rules.

```
& intro     & elim       & elim       | intro     | intro     => elim
P ...       P & Q ...   P & Q ...     P ...       Q ...       P ...
Q ...       .           .           .           .           P => Q ...
.           .           .           .           .           .
.           .           .           .           .           .
.           P;          Q;          P | Q;      P | Q;      .
P & Q;                                                      Q;

<=> intro       <=> elim        <=> elim
P => Q ...      P <=> Q ...     P <=> Q ...
Q => P ...      .               .
.               .               .
.               .               .
.               P => Q;         Q => P;
P <=> Q;
```

The templates given above are for the two-place & and |, but there are also templates for operators with three or more operands. Recall that & and | are combining operators, meaning, for example that P & Q & R means neither (P & Q) & R nor P & (Q & R). The elimination rule allows you to "pull away" any number of combined & operators in one step. Thus, if you have established P & Q & R, you may conclude either P, Q, or R by & elimination. Given (P & Q) & R, you would need two steps to conclude P; you would first have to prove P & Q. The introduction rules work in a similar fashion. If you establish P, you can then prove P | Q | R in one introduction, but to prove (P | Q) | R, you would first have to prove P | Q.

<=> RULES

The special rules we had for the "=" operator reflected the fact that, as an assertion, X=Y meant that the FIXED or BIT expressions X and Y had the same value, and could therefore be substituted for each other in an assertion P, subject to certain restrictions about bound variables which insured