Important Concepts to Remember

Your Name

November 7, 2023

1 Haskell

1.1 Input/Output

Java code:

```java
void f(String out) {
    String inp1 = Console.readLine();
    String inp2 = Console.readLine();
    if (inp2.equals(inp1)) System.out.println(out);
}
```

Listing 1: Java Code

How to convert to Haskell:

```haskell
f :: String -> IO ()
f out = do
inp1 <- getLine
inp2 <- getLine
if inp2 == inp1
    then putStrLn out
    else return ()
```

Listing 2: Haskell Code

1.2 Syntax for IO type

The syntax for the IO type includes:

- The `do` block sequences side effects.
- `<-` extracts values from IO.
- `return` wraps values in IO.
- `show` converts values to Strings.
- `read` converts Strings to values (Always specify the desired type!).
- For α-equivalence, no variables can be free.

2 Syntax Tree

The syntax tree rules include:

- $\wedge$ binds stronger than $\vee$ and stronger than $\rightarrow$.
- $\rightarrow$ associates to the right; $\wedge$ and $\vee$ associate to the left.
- Negation binds stronger than binary operators.
- Quantifiers extend to the right as far as possible.

Proof Rule for Induction Step:

$$\frac{\Gamma \vdash P[n \mapsto 0] \qquad \Gamma \vdash \forall m : \mathit{Nat}.\, P[n \mapsto m] \rightarrow P[n \mapsto m+1]}{\Gamma \vdash \forall n : \mathit{Nat}.\, P} \ (m \ \textit{not free in} \ P)$$

Figure 1: Induction Step Tree

3 Foldr/Foldl

3.1 Foldr

The easiest way to understand `foldr` is to rewrite the list as a series of cons operations.

```
[1,2,3,4,5] => 1:(2:(3:(4:(5:[]))))
```

Listing 3: Haskell Code

Now what `foldr f x` does is that it replaces each `:` with `f` in infix form and `[]` with `x` and evaluates the result.

For example:

```
sum [1,2,3] = foldr (+) 0 [1,2,3]
```

Listing 4: Haskell Code

`[1,2,3] === 1:(2:(3:[]))`

So,

```
sum [1,2,3] === 1+(2+(3+0)) = 6
```

Listing 5: Haskell Code

4 Currying and Uncurrying

Currying is the process of transforming a function that takes multiple arguments in a tuple as its argument into a function that takes a single argument and returns another function that accepts further arguments one by one. You can convert between curried and uncurried forms using the Prelude functions `curry` and `uncurry`.

5 CYP

Proof by induction on List `xs` generalizing `zs`:

```
    Case []
    For fixed \texttt{zs}
    Show: \texttt{rev [] ++ zs .=. qrev [] zs}
    ...
Case y:ys
    Fix \texttt{y, ys}
    Assume
        IH: forall \texttt{zs: rev ys ++ zs .=. qrev ys zs}
    Then for fixed \texttt{zs}
    Show: \texttt{rev (y:ys) ++ zs .=. qrev (y:ys) zs}
    ...
QED
```

Listing 6: Haskell Code

6 η-conversion

The following two terms are equivalent under η-conversion:

$$x \to fx \text{ and } f$$

Converting from left to right is η-contraction, and converting from right to left is η-expansion. η-conversion is sometimes useful to simplify expressions.

Example:

Function `parity` takes a list of Integers and transforms it into a list of 0/1s.

```
    parity xs = map elemPar xs where elemPar x = mod x
```

Listing 7: Haskell Code

General Procedure of `foldr` and `foldl`

1. Identify recursive, dynamic, and static arguments.

```
    foldl f z (x:xs) = foldl f (f z x) xs
```

Listing 8: Haskell Code

2. Write an auxiliary function that has the recursive, then the dynamic arguments. Static arguments can still occur freely (and will come from the final context).

```
    aux [] z = z
aux (x:xs) z = aux xs (f z x)
```

Listing 9: Haskell Code

3. Write the dynamic arguments as lambdas.

```
    aux [] = \z -> z
aux (x:xs) = \z -> aux xs (f z x)
```

Listing 10: Haskell Code

4. Rewrite **aux** in terms of **foldr**. **x** and **aux xs** become arguments of the function for the recursive case.

```haskell
aux = foldr (\x rec -> \z -> rec (f z x)) (\z -> z)
```

Listing 11: Haskell Code

5. Express the original function in terms of **aux** (reorder the dynamic and recursive arguments, if needed).

```haskell
foldl f z xs = aux xs z
```

Listing 12: Haskell Code

6. Replace **aux** with its implementation.

```haskell
foldl f z xs = foldr (\x rec z -> rec (f z x)) (\z -> z) xs z
```

Listing 13: Haskell Code

7 IMP

Remember the following:

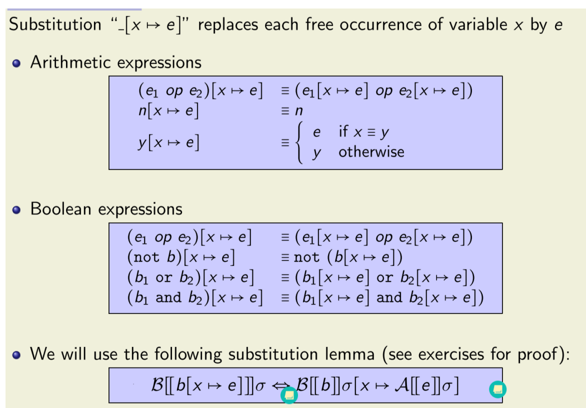

Figure 2: Substitution Rule

7.1 Proof Structure

7.1.1 Free Variables / Arithmetic Expression

Let x, y be arbitrary. Use strong structural induction on e. Thus, we have to prove $P(e)$ for some arbitrary arithmetic expression e and assume $\forall e'' \subset e, P(e'')$ as our induction hypothesis.

- **Case 1:** $e \equiv n$ for some numerical value n. - **Case 2:** $e \equiv y$ for some variable y. - **Case 3:** $e \equiv e_1$ op e_2 for some arithmetic expression e_1, e_2 and some arithmetic operator *op*.

7.1.2 Boolean Expression

- **Case 1:** $b \equiv b_1$ or b_2 for some boolean expressions b_1, b_2. - **Case 2:** $b \equiv b_1$ and b_2 for some boolean expressions b_1, b_2. - **Case 3:** $b \equiv$ not b' for some boolean expression b'. - **Case 4:** $b \equiv e_1$ op e_2 for some arithmetic expression e_1, e_2 and some arithmetic operator *op*.

Arithmetic expressions

$$\begin{aligned} FV(e_1 \; op \; e_2) &= FV(e_1) \cup FV(e_2) \\ FV(n) &= \varnothing \\ FV(x) &= \{x\} \end{aligned}$$

Boolean expressions

$$\begin{aligned} FV(e_1 \; op \; e_2) &= FV(e_1) \cup FV(e_2) \\ FV(\texttt{not } b) &= FV(b) \\ FV(b_1 \texttt{ or } b_2) &= FV(b_1) \cup FV(b_2) \\ FV(b_1 \texttt{ and } b_2) &= FV(b_1) \cup FV(b_2) \end{aligned}$$

Statements

$$\begin{aligned} FV(\texttt{skip}) &= \varnothing \\ FV(x\texttt{:=}e) &= \{x\} \cup FV(e) \\ FV(s_1\texttt{;}s_2) &= FV(s_1) \cup FV(s_2) \\ FV(\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end}) &= FV(b) \cup FV(s_1) \cup FV(s_2) \\ FV(\texttt{while } b \texttt{ do } s \texttt{ end}) &= FV(b) \cup FV(s) \end{aligned}$$

Figure 3: Free Variable

7.1.3 Trees

$R[T] \equiv \forall T, P, Q, b, s ... \text{ root}(T) \equiv ... \implies ...$

We want to prove $\forall T.R(T)$ by strong induction over the shape of T. Assume $\forall T' \subset T.R[T']$. Assume LHS holds. We do a case distinction on the last rule applied in T:

```
Here goes the proof

   \         T1          /
    \                   /
     -------------------
           X, Y
```

Since $T1 \subset T$, and the root has the same statement, we can apply the I.H. We instantiate $P, Q, ...$ as $P', Q', ...$ respectively. Since LHS holds, we know $\exists T'$ s.t. $\text{root}(T') \equiv ....$

8 Find Invariants

8.1 Min, Max (continued)

```
     while (x < y) {
     t := x;
     x := y;
     y := t
}
```

Listing 14: Haskell Code

$\{\downarrow x = \max(X, Y)\}$
Invariant: $\{\max(x, y) = \max(X, Y)\}$
Variant: $y - x = Z$

8.2 Swap

Let $x \geq 0$ and $x = X$.

```
    a := x;
y := 0;
while (a \neq 0) {
    y := y + 1;
    a := a - 1;
}
```

Listing 15: Haskell Code

$\{\downarrow y = X\}$
Invariant: $\{a + y = X \wedge a \geq 0\}$
Variant: a

8.3 A^{2^N}

$\{a = A \wedge A > 0 \wedge n = N \wedge N \geq 0\}$

```
    k := 0;
r := a;
while (k < n) {
    k := k + 1;
    r := r \cdot r
}
```

Listing 16: Haskell Code

$\{\downarrow r = A^{2^N}\}$
Invariant: $\{a = A \wedge A > 0 \wedge n = N \wedge N \geq 0 \wedge r = A^{2^k} \wedge k \leq N\}$
Variant: $n - k$

8.4 Remainder

$\{N \geq 0 \wedge D > 0 \wedge d = D \wedge r = N \wedge q = 0\}$

```
    while (r \geq 0) {
    r := r - d;
    q := q + 1;
}

r := r + d;
q := q - 1;
```

Listing 17: Haskell Code

$\{\downarrow N = q \cdot D + r \wedge r \geq 0 \wedge r < D\}$
Invariant: $\{N = q \cdot d + r \wedge d = D \wedge D > 0 \wedge r + d \geq 0\}$
Variant: $r = Z$

8.5 N^K

$\{k \geq 1 \wedge k = K \wedge n \geq 1 \wedge n = N\}$

```
    i := 0;
r := 1;
while (i < k) {
```

```
    i := i + 1;
    r := r \cdot n;
}
```

Listing 18: Haskell Code

$\{\downarrow r = N^K\}$
Invariant: $\{k = K \land n = N \land r = n^i \land i \leq k\}$
Variant: $k - i = V$

8.6 $N = q \cdot D + r$

$\{N \geq 0 \land D > 0 \land d = D \land r = N \land q = 0\}$

```
    while (r \geq 0) {
    r := r - d;
    q := q + 1;
}
r := r + d;
q := q - 1;
```

Listing 19: Haskell Code

Use the loop invariant in the invariant.
Use post-condition in the loop invariant.
Check if you can already conclude with the invariant your post-condition.

9 Liveness and Safety

Liveness

- Something good will happen eventually.
- If the good thing has not happened yet, it could happen in the future.
- A liveness property does not rule out any prefix.
- Every finite prefix can be extended to an infinite sequence that is in P.
- Liveness properties are violated in infinite time.

Safety

- Something bad is never allowed to happen (and can't be fixed).
- Safety properties are violated in finite time and cannot be repaired.