Today

- programs from proofs
- induction and recursion

Deductive Synthesis

Constructing a program by finding a derivation.

In Amphion, one stage involves

showing that the specification can be satisfied.

We can use inference systems to do this
(many possibilities).

So we want to provide tools to support SE.

But how does this construct a programme?

In general, it doesn't! – we have to find the good derivations (which means, a different logic).

Deductive Synthesis

- Express the desired relation between the input and output in the predicate calculus:

  $$spec(input, output)$$

  Assume that we have properties of the datatypes involved (lists, strings, . . .) as axioms.
- Now *prove* that the specification can be satisfied; find a derivation of

  $$\forall x\ \exists y\ spec(x, y).$$

- If the proof uses *constructive* logic, then we can automatically convert such a proof into a functional program $prog$ such that

  $$\forall x\ spec(x, prog(x)).$$

Some Rules

Standard rules for the quantifiers; a version of these is found in sequent calculus formulations.

`allI`	To show $\forall x\ P(x)$, show $P(x_n)$ for a new variable x_n
`someI`	To show $\exists x\ P(x)$, supply a *term* t, and show $P(t)$
`allE`	If given $\forall x\ P(x)$, can also assume $P(t)$ for any term t
`someE`	If given $\exists x\ P(x)$, can assume $P(x_n)$ for new var x_n

Constructive Logic

Some of the usual rules have to be restricted. For example,

- In *case analysis*, we must have a way of computing which case holds.

  In standard logic, for any formula F, we can always split a proof into two branches, one where F holds and one where $\neg F$ holds.

  In constructive logic, we need a decision procedure before this split is allowed.

  We can use

  $$\forall n, m \ : int \ n = m \vee n \neq m$$

  but not

  $$\forall n, m \ : real \ n = m \vee n \neq m$$

- Proof by contradiction is *not* allowed.

  We can't conclude A is true just because $\neg A$ gives a contradiction.

So we replace the task of constructing a program with that of proving a theorem. This is still a hard task.

If we do find a proof, we know that the program is guaranteed to fit its specification, and this is a big advantage.

Synthesis and reasoning

We can't just use a standard FOL system to show that a specification can be met, and expect that there is a program that computes what we want — maybe the input/output relation always makes sense, but there is no way of computing the output (see Computability course).

So we need another inference system; what about FOL (with some restrictions). We take a standard set of rules before restrictions.

Example

Find the integer square root of x.

Use the specification

$$spec(x, y) =_{def} y^2 \leq x \wedge (y+1)^2 > x$$

And look for a proof of

$$\forall x \, \exists y \; y^2 \leq x \wedge (y+1)^2 > x$$

Apart from usual rules, we have an induction rule.

After induction there are two things to prove.

First we need to show

$$\exists y\ y^2 \leq 0 \wedge (y+1)^2 > 0.$$

Do this by taking the value 0 for y and using arithmetic.
Next assume

$$\exists y\ y^2 \leq k\ \wedge\ (y+1)^2 > k \tag{1}$$

and we need to prove

$$\exists y\ y^2 \leq k+1\ \wedge\ (y+1)^2 > k+1.$$

The `someE` rule applied to hypothesis (1) gives us a new assumption

$$y_0^2 \leq k\ \wedge\ (y_0+1)^2 > k \tag{2}$$

There are now two cases to consider:

either (a) $(y_0+1)^2 = k+1$
or (b) $(y_0+1)^2 > k+1$.

- *Case (a)*: Take $y_0 + 1$ to be the answer (use `someI` rule with that value).

  We now have to check that

  $$(y_0+1)^2 \leq k+1 \wedge (y_0+2)^2 > k+1.$$

- *Case (b)*: Take y_0 to be the answer. We now have to check that

  $$y_0^2 \leq k+1\ \wedge\ (y_0+1)^2 > k+1.$$

Checking this arithmetic finishes the proof.

From proof to program

A program is obtained by attaching bits of program to the proof rules, and building up the program from the way the rules are used in the proof.

For example, the induction rule says:

$$\frac{P(0) \qquad P(n) \to P(n+1)}{P(x)}$$

Suppose programs `A,Q` have been associated with the two formulas above the line. We can then associate with $P(x)$ the program

```
fun Rec 0 = A
  | Rec n = Q (Rec (n-1))
```


Putting together all the parts of the proof, we get the following functional program:

```
fun prog 0 = 0
  | prog n =  let val x = prog (n-1)
              in
                  if (1+x)^2 = n
                      then (1+x)
                  else x
              end
```

This is not an efficient program — but it is a correct program, and we have verified its correctness!

A Synthesis KBS

A KBS to support this sort of program construction has to include:

- A proof system with program constructs;
  Several such systems are known and implemented.
- Some knowledge of how to build derivations, especially where induction is needed.
- Knowledge of which derivations correspond to *efficient* programs, so that synthesis is directed.

Proofs by Induction

Many inductive proofs can be found by following heuristics developed by Boyer and Moore.

The main choices to make are

- which induction rule to use
- which induction variable to apply the rule to

The efficiency of the program is largely dependent on the induction rule chosen.

Complexity of Inductions

Rules contribute to complexity in different ways:

Step Case	Complexity
$P(x) \rightarrow P(x+1)$	Linear
$P(x/2) \rightarrow P(x)$	Log
$P(a), P(b) \rightarrow P(c)$ (for $a, b < c$)	Exponential

This close link between induction (in proof) and recursion (in program) gives us a way of influencing the quality of the implementation.

Summary

- Synthesis via deduction
- Constructive Logic
- Programs from Proofs