REASONING ABOUT RECURSIVE PROGRAMS

Note:

This class and the next ones try to give a pragmatic introduction to the subject of recursive programs and Abstract Data Types. These are subjects on which whole courses are taught. The treatment here is quick and informal, having the purpose of providing a general idea of some important concepts for the user.

Many slides are not here and will be distributed in class.

Natural Numbers (Recursive Definition)

(i) 0 is a number
(ii) if n is a number, n’ is a number

e.g.

0, 0’’’, 0’’’’’ …’’ are numbers

By convention,

0 = 0
0’’’ = 3
etc.

Functions on Numbers

Define function

$$\text{plus: } \mathrm{Nat}, \mathrm{Nat} \to \mathrm{Nat} \quad \text{(* signature of type Nat *)}$$

1) $\mathrm{plus}(0, n) \Leftarrow n$

2) $\mathrm{plus}(m', n) \Leftarrow \mathrm{plus}(m, n)'$

This is not only a definition, but also a program capable of being evaluated for given values of m,n

plus (0’’, 0’’’’) = (2)
plus (0’, 0’’’’)’ = (2)
plus (0, 0’’’’)’’ = (1)
0’’’’’’

(assuming an obvious ‘bracket elimination’ rule)

We say that *plus (0’’, 0’’’’)* has been reduced to its **normal form** *0’’’’’’*

check that plus (0’’’,0’’’)
plus (0’’’’’, 0’)

all reduce to the same normal form

A **normal form** involves only **constructors**, i.e. a set of constants and functions symbols

• in terms of which all data of the given type can be expressed,

and

• for which there are no equations.

In this example, note that all data of type *number* can be represented in terms of 0 or ’, also there are no equations for these (there are only equations for *plus*).

In general, constructors and normal forms are not guaranteed to exist, but it is recommended that equations be written so that they exist (see later).

When recursive definitions are seen as programs or expressions to be evaluated, order of evaluation of subexpressions becomes important.

e.g.

$$\mathrm{plus}(\mathrm{plus}(0',0'),\mathrm{plus}(0'',0))$$

Where do we start?
Does it matter?

From a mathematical point of view, we think of each subexpression as having a value independent of the order of evaluation.

However in computing a specific order of evaluation must be followed.

Some orders of computation may lead to the solution, others may run forever

(result by Cadiou, 1972)

We may decide to accept only orders leading to a result compatible with mathematical interpretation.

(full substitution)

Leftmost innermost is a candidate (call by value)

[Figure from book by Z. Manna, Mathematical Theory of Computation, p. 376]

Or see J. van Leeuwen, Handbook of Theoretical Computer Science, Vol B, p. 469.

Leftmost	plus(plus(0’,0’), plus(0’’,0)) =
Innermost	plus(0’’, plus(0’’,0)) =
(byvalue)	plus(0’’, 0’’) =
	0’’’’

Unfortunately, this rule will loop forever
in cases where parallel won’t

However, we adopt it for convenience

Try function (for naturals)

$$f(0, y) \Leftarrow 0$$
$$f(x', y) \Leftarrow f(x, f(x', y))$$

with x=0, y=0
and with parallel
leftmost innermost
(note: leftmost innermost won’t give a result until all parameters are completely evaluated)

Note also that *leftmost* (call by name) will terminate in this case.

The following is known:

The parallel-outermost, the free argument, and the full-substitution rules are *safe*

However to explain this result would take a while... (what does it mean exactly to be *safe*?)

See Z. Manna, Mathematical Theory of Computation, Wiley, 1974, page 384 ff.

In what follows, we shall assume that the functions are computed in such a way that the computation completes, if it can at all.

Some further examples:

$$\text{times(Nat, Nat)} \rightarrow \text{Nat}$$

$$\text{times }(n,0) \Leftarrow 0$$
$$\text{times}(m',n) \Leftarrow \text{plus}(\text{times}(m,n), n)$$

$$\text{equal(Nat, Nat)} \rightarrow \text{Truthval}$$

$$\text{equal}(0,0) \Leftarrow \text{TRUE}$$
$$\text{equal}(0, n') \Leftarrow \text{FALSE}$$
$$\text{equal}(m', 0) \Leftarrow \text{FALSE}$$
$$\text{equal}(m',n') \Leftarrow \text{equal}(m,n)$$

$$\text{fact(Nat)} \rightarrow \text{Nat}$$
$$\text{fact}(0) \Leftarrow 0'$$
$$\text{fact}(n') \Leftarrow \text{times}(n', \text{fact}(n))$$

$$\text{ack(Nat, Nat)} \rightarrow \text{Nat}$$
$$\text{ack}(0,n) \Leftarrow n'$$
$$\text{ack}(m',0) \Leftarrow \text{ack}(m,0')$$
$$\text{ack}(m',n') \Leftarrow \text{ack}(m, \text{ack}(m',n))$$

try ack(4,4) !

Lists of Naturals

$$:: \quad : \ Nat, List \rightarrow List$$

(1) Nil is a list (constructor)
(2) if n is a Nat and l is a list, then n : : l is a list
(: : is also a constructor)

post: $Nat, List \rightarrow List$

$$post(n, Nil) \Leftarrow n::Nil$$
$$post(n, m::l) \Leftarrow m::post(n, l)$$

(append n at end of l)

e.g.

$$post(4, 1::2::3::Nil) = 1::2::3::4::Nil$$

double:

$List \rightarrow List$

$$double(Nil) \Leftarrow Nil$$
$$double(n::l) \Leftarrow times(2, n)::double(l)$$

rev:

$List \rightarrow List$

$$rev(Nil) \Leftarrow Nil$$
$$rev(n::l) \Leftarrow post(n, rev(l))$$

insert: $\quad \text{Nat}, \text{List} \rightarrow \text{List}$

$\text{insert}(m, \text{Nil}) \Leftarrow m::\text{Nil}$

$\text{insert}(m, n::l) \Leftarrow n::\text{insert}(m,l)$ if m>n

$\text{insert}(m, n::l) \Leftarrow m::n::l$ if not m>n

looks for an element greater than or equal to element to be inserted

sort: $\quad \text{List} \rightarrow \text{List}$

$\text{sort}(\text{Nil}) \Leftarrow \text{Nil}$

$\text{sort}(m::l) \Leftarrow \text{insert}(m, \text{sort}(l))$

for example:

$$\text{sort}(2::3::1::\text{Nil}) =$$
$$\text{insert}(2, \text{sort}(3::1::\text{Nil})) =$$
$$\ldots$$
$$\text{insert}(2, \text{insert}(3, \text{insert}(1, \text{sort}(\text{Nil})))) =$$
$$\text{insert}(2, \text{insert}(3, \text{insert}(1, \text{Nil}))) =$$
$$\text{insert}(2, \text{insert}(3, 1::\text{Nil})) =$$
$$\text{insert}(2, 1::\text{insert}(3, \text{Nil})) =$$
$$\text{insert}(2, 1::3::\text{Nil}) =$$
$$\ldots$$
$$1::2::3::\text{Nil}$$

Axioms for Natural Numbers

Cases

if n is a natural number then
either $n = 0$
or
$n = m'$ for some natural number m

Uniqueness

if n and m are natural numbers
then $n' = m'$ implies $n = m$

Induction

induction hypothesis

if $P(0)$	base
and	
$\forall$ m, $P(m)$ implies $P(m')$	step
then $\forall$ n, $P(n)$	conclusion

$'$ is the constructor of the natural numbers

Proof methods for recursive programs

The formal semantics of recursive programs, and related proof methods, are a very developed subject in theoretical and applied computer science.

In this course, you are only exposed to some very basic ideas.

Two well-known proof methods for recursive programs, which have been automated in tools, are:

- computational induction, i.e. induction on the level of recursion

- structural induction, i.e. induction on the depth of the data structure.

We concentrate on the second method.

Structural induction

Partially ordered set

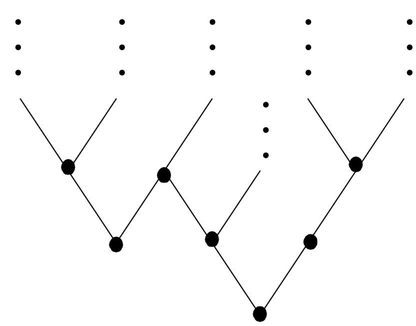
A non-empty set on which a relation < is defined:

1) if a < b and b < c then a < c (transitivity)
2) if a < b then b $\not<$ a (asymmetry)
3) a $\not<$ a (irreflexivity)

Well-founded set:

a partially ordered set which contains no infinite decreasing sequence

$$a_0 < a_1 < a_2 \ldots$$



Structural induction on well founded sets:

Let S be a WFS, and let P be a property

If for all a in S we can prove that
P(a) must be true if it P(b) is true forall b < a in S
then we must conclude P(c) for all c in S

(Note: if there are no b < a , P(a) must be proved unconditionally (induction base))

Structural Induction Theorem (Burstall)

Let S be a well-founded set, and P be a total predicate over S:

if for all a in S we can prove that
P(a) is implied by P(b) for all b < a **(*)**
then P(c) for all c in S

Proof.

By contradiction. We show that if the assumption (*) is satisfied, then there can be no element of S for which P is false.

Consider the set A

$$A = \{a \mid a \text{ in } S \text{ and } \text{ not } P(a)\}$$

Assume that A is nonempty. Then there must be a least element a_0 in A such that a /< a_0 for any a in A, otherwise there would be an infinite descending sequence in S. Then, for any element b in S such that $b<a_0$, P(b) is true. But by (*) then $P(a_0)$ must also be true, contradicting the fact that a_0 is in A. Therefore A must be empty, that is, P(c) is true for all elements of S.

The well-founded ordering to be used can be different from proof to proof.

There are well-founded sets for naturals, the most obvious being:

etc.

0’’

0’

0

and for lists

etc.

0::0::nil   0’::0::nil   0’’::0::nil   etc.

0::nil   0’::nil   0’’::nil

nil

in many cases, the following simpler ordering for lists will suffice (induction on the number of elements in the list):

etc.

m::n::nil

m::nil

nil

Structural induction proofs will then go as follows:

Find a **suitable well-founded set**.

Induction base: Prove the property for the bottom element(s) of the ordering.

Induction hypothesis: Assume that the property holds for an arbitrary element X of the ordering.

Induction step: Under this hypothesis, prove that the property holds for all successors of X in the ordering.

We can then **conclude** that the property holds for all elements in the ordering (all naturals, all lists of naturals...).

[Here prof shows a number of proofs on the board]

Simple proofs we have seen so far proceeded by symbolically evaluating formulas until both sides of an equation were completely reduced.

At that point, hopefully they also had the same form.

Does this always work?

(If it did, everything would be very simple!)

1 $\mathrm{join}(\mathrm{Nil}, l) \Leftarrow l$

2 $\mathrm{join}(s::k, l) \Leftarrow s::\mathrm{join}(\dot{k}, l)$

3 $\mathrm{post}(n, \mathrm{Nil}) \Leftarrow n::\mathrm{Nil}$

4 $\mathrm{post}(n, m::l) \Leftarrow m::\mathrm{post}(n, l)$

5 $\mathrm{rev}(\mathrm{Nil}) \Leftarrow \mathrm{Nil}$

6 $\mathrm{rev}(n::l) \Leftarrow \mathrm{post}(n, \mathrm{rev}(l))$

[More proofs on the board]

Method used:

1) Choose a variable to induct on

2) Set up base and step subproblems

3) Prove base and step:

Rewrite LHS and RHS as far as possible, using

- function definitions
- induction hypothesis

(pattern matching/unification)

4) If stuck, formulate lemma

Note:

Just as the program, this process is not guaranteed to terminate. For example, it won't if rules contain 'loops' (e.g. $\text{plus}(a, b) \Leftarrow \text{plus}(b, a)$)

But if it terminates, we end up in a 'normal form' where no further reduction is possible.

To choose the variable to induct on:

i) Select positions in LHS of definitions which contain *terms* (rather than simple variables)

e.g. $$\text{post}(n, m{::}\text{Nil}) \Leftarrow m{::}\text{post}(n, \text{Nil})$$

position **2** is induction position for post

$$m' + n \Leftarrow (m + n')$$

position **1** is induction position for +

ii) Choose as variables to induct on the variables that appear in these positions in theorem, possibly on both sides.

e.g.
$$\text{join}(s{::}k, l) \Leftarrow s{::}\text{join}(k, l)$$

position **1** is induction position for join

To prove:

$$\forall\, s,k,l \quad \underbrace{\text{join}(k, \text{post}(s, l))}_{\text{LHS}} = \underbrace{\text{post}(s, \text{join}(k, l))}_{\text{RHS}}$$

k is in induction position for join on both sides

$\therefore$ induct on k

e.g.

join(Nil,l) ⇐ l
join(s::k,l) ⇐ s :: join(k,l)

rev(Nil) ⇐ Nil
rev(n::l) ⇐ post(n, rev(l))

1 *is induction position for rev*
1 *is induction position for join*

To prove:

∀ lists l,k

$$rev(join(l, k)) = join(rev(k), rev(l))$$

l is in induction position for join in LHS
l is in induction position for rev in RHS

∴choose l

Using Lemmas

When we are unable to prove a property,
we try to prove a more general one,
by replacing some subexpressions with variables.

Where is the new variable introduced?

Answer:

Best in induction position in the definition
(so we can induct on it)

Induction position:

rev **`1`**
join **`1`**
post **`2`**

How to Formulate Helpful Lemmas

Lemmas enable us to do some replacements that we could not do directly by function definition.

To show

$$join(rev(k), Nil) = rev(k)$$

use lemma

$$join(l, Nil) \quad = \quad l \qquad\qquad (l \Rightarrow rev(k))$$

To show

$$rev(post(s, rev(l))) = s::rev(rev(l)))$$

use lemma

$$rev(post(s, l)) \quad = \quad s::rev(l) \qquad\qquad (l \Rightarrow rev(l))$$

To show

$$post(s, join(rev(k), rev(l))) = join(rev(k), post(s, rev(l)))$$

use lemma

$$post(s, join(k, l)) = join(k, post(s, l))$$

$$(\ k \Rightarrow rev(k), \qquad l \Rightarrow rev(l)\ )$$

ALGEBRAIC ABSTRACT DATA TYPES:

A SPECIFICATION TECHNIQUE FOR DATA

Recursive programs such as the ones we have seen so far are in the category of *rewriting systems*:

rules express that LHS can be *rewritten* as RHS.

One can also think of *equational* systems, where rules express *equality of terms* (rewriting can be done in both directions).

We are then in the area of *algebraic data types* and algebraic specification techniques.

Algebraic specification techniques have their origin in abstract algebra, i.e.

$$
\begin{aligned}
0 + x &= x \\
x + 0 &= x \\
x + \overline{x} &= 0 \\
x + (y + z) &= (x + y) + z
\end{aligned}
$$

this is the well-known set of *axioms* for groups.

A set of axioms defines an equivalence relation over the (normally infinite) set of group terms

$$
\begin{aligned}
& (x + x) + \overline{x} \\
= \quad & x + (x + \overline{x}) \\
= \quad & x + 0 \\
= \quad & x
\end{aligned}
$$

Many-Sorted Algebras

Are a useful model for expressing data abstractions (*data types*)

Such algebras involve variables and operators of different *sorts*

E.g. specifying a *type* stack: there are perhaps three sorts involved:

- the sort *stack*
- the sort of the *elements* that can be put in stack (e.g. integer)
- perhaps the *boolean* sort, necessary to test conditions on stacks, such as: is empty, etc.

The definition of a data abstraction (called data *type*) will consist of two parts:

- a *signature* or *syntax* part, listing the sorts, the operators, and their functionalities

- and an *equational* part, listing the equations that describe the properties of the elements of the type

Expressions involving operators of the sort and variables are called *terms* [note: constants are 0-adic operators]

Example: specifying the type *stack of integers*:

Signature part: listing the sorts, the operators, and their function-alities

NEWSTACK → Stack
PUSH (Stack, Integer) → Stack
TOP (Stack) → Integer ∪ {INTEGERERROR}
POP (Stack) → Stack ∪ {STACKERROR}

Where Stack and Integer are sorts, NEWSTACK, PUSH, TOP, POP are operators, INTEGERERROR and STACKERROR are constants of special sorts used to define error conditions.

Equation part: lists the equations that describe the properties of the elements of sort Stack:

for all S: Stack, I: Integer

equations of sort Integer:
TOP(PUSH(S,I)) = I
TOP(NEWSTACK) = INTEGERERROR

equations of sort Stack:
POP(PUSH(S,I)) = S
POP(NEWSTACK) = STACKERROR

Stacks are now algebraic objects, on which algebraic reasoning is possible.

GUIDELINES FOR DESIGNING DATA TYPE SPECIFICATIONS

- Determine the sorts
- Determine the operators [incl. constants]
- Determine the functionality of each operator: sorts of args and results (signature)
- In general, each type defines only one sort, but uses operations of other sorts.
- Some of the operators will have functionality *out of* the type, others *in the* type.
- A subclass of the ops having functionality in the type are the *constructors:* all the elements of the sort can be represented by using only constructor set operators

  e.g. for stacks, the constructors are: NEWSTACK, PUSH and not POP, TOP,

  Any element of type stack can be represented by these two, e.g. PUSH(PUSH(PUSH(NEWSTACK, 3), 2), 1) represents a stack containing 1 at the top and 3 at the bottom. This is equivalent to many other possibilities, e.g.
  PUSH(PUSH(POP(PUSH(PUSH(NEWSTACK, 3),4)),2),1)
  representing a stack which has the same contents as the previous one, but where a 4 was added and then removed.
  It is also equivalent to:
  PUSH(POP(PUSH
  (PUSH(POP(PUSH(PUSH(NEWSTACK, 3),9)),2),1)),1)

Usually, constructor operators can be recognized by the fact that they do not have explicit equations. They are defined implicitly by their effects on other operators (see stack example).

The *normal form* of a term of a sort includes only constructors.

TREATMENT OF ERROR CASES

There are three main ways of specifying error cases:

• to have an *error* element in each sort. Unfortunately, this requires ‘propagating’ error cases from sort to sort. E.g. if we have *integererror* of sort *integer* resulting from division by 0, then we may have to consider what happens when *integererror* is put in a stack, resulting possibly in *stackerror* of sort *stack*, etc. This is the way it’s done in the ADT part of LOTOS, but it may be cumbersome.

• To have a special ‘error’ sort. This creates complications in the signature of operators, because each operator will have to apply to its specific sort *union* the error sort.

• to equate *error* with ‘undefined’. Then our functions are partially defined. Division by zero is undefined, putting an undefined value in a stack results in an undefined stack, etc. In this technique, one may still use a the word *error*, but it does not denote a value in a sort: it denotes an undefined result (see the papers handed out).

DIRECT IMPLEMENTATION OF ABSTRACT TYPES

Because of the fact that abstract data type equations can be executed as programs (if they are written in such a way as to be executable...), they can be seen as a *direct implementation* of the data type itself.

Execution is done by using the axioms to reduce terms to normal form: this is an evaluation process.

e.g.

```
     TOP(PUSH(POP(PUSH(NEWSTACK, 1), 2)))
  =  TOP(PUSH(NEWSTACK,2))
  =   2

  TOP(POP(PUSH
     (PUSH(POP(PUSH(PUSH(NEWSTACK,5),6)),2),3))
  =  2
```

Recall: normal form is a term which can't be reduced further. It is the simplest representation for its equivalence class of expressions.

Normal forms contain only constructors.

It is possible that normal forms don't exist, or that several exist for a given expression, but this should be avoided [unless one wants to describe nondeterminism in data types].

IMPLEMENTING ABSTRACT TYPES BY OTHER TYPES

Abstract data types can be *implemented* in terms of other data types that are capable of expressing all the operators by respecting all the axioms.

For example, a *stack* data type can be implemented by an *array* data type (it could not be implemented by a *set* data type).

An implementation can be seen as a *functional program* that defines all the operations of the implemented data type in terms of the implementing type.

The correctness of the implementation can be checked by showing that all the equations of the implemented type still hold in the implementation.

[Extensive discussion of two papers by Guttag, Horowitz, and Musser]