Analyse et Conception Formelles

Lesson 3

–

Recursive Functions and Algebraic Data Types

Recursion everywhere... and nothing else

«Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem»

- The «bad» news: in Isabelle/HOL, there is no `while`, no `for`, no mutable arrays and no pointers, . . .
- The good news: you don't really need them to program!
- The second good news: programs are easier to prove without all that!

In Isabelle/HOL all complex types and functions are defined using recursion

- What theory says: expressive power of recursive-only languages and imperative languages is equivalent
- What functional programmers say: it is as it should always be
- What other programmers say: it is tricky but you always get by

Outline

1. Recursive functions
   - Definition
   - Termination proofs with measures
   - Difference between `fun`, `function` and `primrec`
2. (Recursive) Algebraic Data Types
   - Defining Algebraic Data Types using `datatype`
   - Building objects of Algebraic Data Types
   - Matching objects of Algebraic Data Types
   - Type abbreviations

Acknowledgements:
some material is borrowed from T. Nipkow and S. Blazy's lectures

Recursive Functions

- A function is recursive if it is defined using itself.
- Recursion can be direct

```
fun contains:: "'a => 'a list => bool"
where
"contains e []     = False" |
"contains e (x#xs) = (e=x \/ (contains e xs))"
```

- ... or indirect. In this case, functions are said to be mutually recursive.

```
fun even:: "nat => bool"
and odd::  "nat => bool"
where
  "even 0       = True"  |
  "even (Suc x) = odd x" |
  "odd 0        = False" |
  "odd (Suc x)  = even x"
```

Terminating Recursive Functions

In Isabelle/HOL, all the recursive functions have to be terminating!

How to guarantee the termination of a recursive function? (practice)

- Needs at least one base case (non recursive case)
- Every recursive case must go towards a base case
- ... or every recursive case «decreases» the size of one parameter

How to guarantee the termination of a recursive function? (theory)

- If $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$ then define a measure function $g :: \tau_1 \times \ldots \times \tau_n \Rightarrow \mathbb{N}$
- Prove that the measure of all recursive calls is decreasing

$$\frac{\text{To prove termination of } f \quad f(t_1) \twoheadrightarrow f(t_2) \twoheadrightarrow \ldots}{\text{Prove that} \quad g(t_1) > g(t_2) > \ldots}$$

- The ordering $>$ is well founded on $\mathbb{N}$
  i.e. no infinite decreasing sequence of naturals $n_1 > n_2 > \ldots$

Terminating Recursive Functions (II)

How to guarantee the termination of a recursive function? (theory)

- If $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$ then define a measure function $g :: \tau_1 \times \ldots \times \tau_n \Rightarrow \mathbb{N}$
- Prove that the measure of all recursive calls is decreasing

$$\frac{\text{To prove termination of } f \quad f(t_1) \twoheadrightarrow f(t_2) \twoheadrightarrow \ldots}{\text{Prove that} \quad g(t_1) > g(t_2) > \ldots}$$

Example 1 (Proving termination using a measure)

```
"contains e []     = False" |
"contains e (x#xs)= (if e=x then True else (contains e xs))"
```

1. We define the measure $g = \lambda(x, y).(length\ y)$
2. We prove that $\forall$e x xs. g(e, (x#xs)) $>$ g(e, xs)

Proving termination with measure – the quiz

Quiz 1

- *Proving termination of a function f ensures that the evaluations of (f t) will terminate for* | V | some t || R | all possible t |
- *For a function* `f::'a list ⇒ 'a list` *a measure function should be of type* | V | `'a list ⇒ 'a list` || R | `'a list ⇒ nat` |
- *For the function* `f::nat list ⇒ nat list`

```
"f [] = []" |
"f (x#xs) = (if x=1 then [x] else xs)"
```

V	*We do not need a measure function*
R	*The only possible measure is $\lambda x.(length\ x)$*

- *For function* `f::nat list ⇒ nat list`

```
"f [] = []" |
"f (x#xs)= (if x=1 then (f(x#xs)) else (f xs))"
```

V	*There is no measure function*
R	*The only possible measure is $\lambda x.(length\ x)$*

Terminating Recursive Functions (III)

How to guarantee the termination of a recursive function? (Isabelle/HOL)

- Define the recursive function using `fun`
- Isabelle/HOL automatically tries to build a measure[1]
- If no measure is found the function is rejected
- If it is not terminating, make it terminating!
- Try to modify it so that its termination is easier to show

Otherwise

- Re-define the recursive function using `function (sequential)`
- Manually give a measure to achieve the termination proof

[1] Actually, it tries to build a termination ordering but it has the same objective.

Terminating Recursive Functions (IV)

Example 2

A definition of the contains function using function is the following:

```
function (sequential) contains::"'a ⇒ 'a list ⇒ bool"
where
"contains e []     = False" |
"contains e (x#xs)= (if e=x then True else (contains e xs))"

apply pat_completeness
apply auto
done

termination contains
apply (relation "measure (λ(x,y). (length y))")
apply auto
done
```

Prove that the function is "complete" *i.e.* patterns cover the domain

Prove its termination using the measure proposed in Example 1

Terminating Recursive Functions (V)

Exercise 1

Define the following functions, see if they are terminating. If not, try to modify them so that they become terminating.

```
fun f::"nat => nat"
where
"f x=f (x - 1)"

fun f2::"int => int"
where
"f2 x = (if x=0 then 0 else f2 (x - 1))"

fun f3::"nat => nat => nat"
where
"f3 x y= (if x >= 10 then 0 else f3 (x + 1) (y + 1))"
```


Terminating Recursive Functions (VI)

Automatic termination proofs (`fun` definition) are generally enough

- Covers 90% of the functions commonly defined by programmers
- Otherwise, it is generally possible to adapt a function to fit this setting

Most of the functions are terminating by construction (primitive recursive)

Definition 3 (Primitive recursive functions: `primrec`)

Functions whose recursive calls «peels off» exactly one constructor

Example 4 (`contains` can be defined using `primrec` instead of `fun`)

```
primrec contains:: "'a => 'a list => bool"
where
"contains e []     = False" |
"contains e (x#xs)= (if e=x then True else (contains e xs))"
```

For instance, in `List.thy`:

- 26 "`fun`", 34 "`primrec`" with automatic termination proofs
- 3 "`function`" needing measures and manual termination proofs.

Recursive functions, exercises

Exercise 2

Define the following recursive functions

- *A function* `sumList` *computing the sum of the elements of a list of naturals*
- *A function* `sumNat` *computing the sum of the n first naturals*
- *A function* `makeList` *building the list of the n first naturals*

State and verify a lemma relating `sumList`, `sumNat` *and* `makeList`

Outline

1. Recursive functions
   - Definition
   - Termination proofs with orderings
   - Termination proofs with measures
   - Difference between `fun`, `function` and `primrec`
2. (Recursive) Algebraic Data Types
   - Defining Algebraic Data Types using `datatype`
   - Building objects of Algebraic Data Types
   - Matching objects of Algebraic Data Types
   - Type abbreviations

(Recursive) Algebraic Data Types

Basic types and type constructors (list, $\Rightarrow$, *) are not enough to:

- Define enumerated types
- Define unions of distinct types
- Build complex structured types

Like all functional languages, Isabelle/HOL solves those three problems using one type construction: Algebraic Data Types (sum-types in OCaml)

Definition 5 (Isabelle/HOL Algebraic Data Type)

To define type τ parameterized by types $(\alpha_1, \ldots, \alpha_n)$:

$$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau \ = \ C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \mid \ldots \mid C_k\ \tau_{1,k} \ldots \tau_{1,n_k}$$

with $C_1, \ldots, C_n$ capitalized identifiers

Example 6 (The type of (polymorphic) lists, defined using `datatype`)

```
datatype 'a list = Nil    (* Nil and Cons are capitalized *)
                 | Cons 'a  "'a list"
```


Building objects of Algebraic Data Types

Any definition of the form

$$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_n)\tau \ = \ C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \mid \ldots \mid C_k\ \tau_{1,k} \ldots \tau_{1,n_k}$$

also defines constructors $C_1, \ldots, C_k$ for objects of type $(\alpha_1, \ldots, \alpha_n)\tau$

The type of constructor C_i is $\tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_n)\tau$

Example 7

```
datatype 'a list = Nil
                 | Cons 'a  "'a list"
```

defines constructors

`Nil::'a list` and `Cons::'a` $\Rightarrow$ `'a list` $\Rightarrow$ `'a list`

Hence,

- `Cons (3::nat) (Cons 4 Nil)` is an object of type `nat list`
- `Cons (3::nat)` is an object of type `nat list` $\Rightarrow$ `nat list`

Matching objects of Algebraic Data Types

Objects of Algebraic Data Types can be matched using case expressions:

```
(case l of Nil => ... | (Cons x r) => ...)
```

possibly with wildcards, i.e. "_"

```
(case i of 0 => ... | (Suc _) => ...)
```

and nested patterns

```
(case l of (Cons 0 Nil) => ... | (Cons (Suc x) Nil) => ...)
```

possibly embedded in a function definition

```
fun first::"'a list =>'a list"
 where
"first Nil = Nil" |
"first (Cons x _) = (Cons x Nil)"
```

```
fun first::"'a list =>'a list"
 where
"first [] = []" |
"first (x#_) = [x]"
```

Building objects of Algebraic Data Types – the quiz

Quiz 2 (we define `datatype abstInt= Any | Mint int`)

- *How to build an object of type* `abstInt` *from integer 13?*

V	13	R	*(Mint 13)*

- *How to build the object* `Any` *of type* `abstInt`*?*

V	*Any*	R	*(Mint Any)*

- *To check if a variable* `x::abstInt` *contains an integer how to do?*

V	`(case x of (Mint _) => True | Any => False)`
R	`x= (Mint _)`

- *Let* `f` *be defined by*

```
f::abstInt ⇒ abstInt ⇒ abstInt
"f (Mint x) (Mint y) = (Mint x+y)" |
"f _ _ = Any"
```

What is the value of:

`(f (Mint 1) (Mint 2))`	`(f Any (Mint 2))`
V: `Any`	V: `Any`
R: `Mint 3`	R: *Undefined*

Algebraic Data Types, exercises

Exercise 3

Define the following types and build an object of each type using `value`

- *The enumerated type* `color` *with possible values: black, white and grey*
- *The type* `token` *union of types* `string` *and* `int`
- *The type of (polymorphic) binary trees whose elements are of type* `'a`

Define the following functions

- *A function* `notBlack` *that answers true if a* `color` *object is not black*
- *A function* `sumToken` *that gives the sum of two integer tokens and* 0 *otherwise*
- *A function* `merge::color tree ⇒ color` *that merges all colors in a* `color tree` *(leaf is supposed to be black)*

Type abbreviations

In Isabelle/HOL, it is possible to define abbreviations for complex types
To introduce a type abbreviation .... `type_synonym`

For instance:

- `type_synonym name="(string * string)"`
- `type_synonym ('a,'b) pair="('a * 'b)"`

Using those abbreviations, objects can be explicitly typed:

- `value "(''Leonard'',''Michalon'')::name"`
- `value "(1,''toto'')::(nat,string)pair"`

... though the type synonym name is ignored in Isabelle/HOL output ☹