Analyse et Conception Formelle

Lesson 2

–

Types, terms and functions

Outline

1. Terms
   - Types
   - Typed terms
   - λ-terms
   - Constructor terms
2. Functions defined using equations
   - Logic everywhere!
   - Function evaluation using term rewriting
   - Partial functions

Acknowledgements: some slides are borrowed from T. Nipkow's lectures

Types: syntax

τ ::=	(τ)	
|	$bool \mid nat \mid char \mid \ldots$	base types
|	$'a \mid 'b \mid \ldots$	type variables
|	$\tau \Rightarrow \tau$	functions
|	$\tau \times \ldots \times \tau$	tuples (ascii for $\times$: $*$)
|	τ *list*	lists
|	$\ldots$	user-defined types

The operator $\Rightarrow$ is right-associative, for instance:

$$nat \Rightarrow nat \Rightarrow bool \text{ is equivalent to } nat \Rightarrow (nat \Rightarrow bool)$$

Typed terms: syntax

term ::=	$(term)$	
|	a	$a \in \mathcal{F}$ or $a \in \mathcal{X}$
|	*term term*	function application
|	$\lambda y.\ term$	function definition with $y \in \mathcal{X}$
|	$(term, \ldots, term)$	tuples
|	$[term, \ldots, term]$	lists
|	$(term :: \tau)$	type annotation
|	$\ldots$	a lot of syntactic sugar

Function application is left-associative, for instance:

$$f\ a\ b\ c \text{ is equivalent to } ((f\ a)\ b)\ c$$

Example 1 (Types of terms)

Term	Type	Term	Type
y	'a	t1	'a
(t1,t2,t3)	('a × 'b × 'c)	[t1,t2,t3]	'a list
λ y. y	'a ⇒ 'a	λ y z. z	'a ⇒ 'b ⇒ 'b

Types and terms: evaluation in Isabelle/HOL

To evaluate a term t in Isabelle .... value "t"

Example 2

Term	Isabelle's answer
value "True"	True::bool
value "2"	Error (cannot infer result type)
value "(2::nat)"	2::nat
value "[True,False]"	[True,False]::bool list
value "(True,True,False)"	(True,True,False)::bool * bool * bool
value "[2,6,10]"	Error (cannot infer result type)
value "[(2::nat),6,10]"	[2,6,10]::nat list

Exercise 1 (In Isabelle/HOL)

Use `append` *to concatenate 2 lists of bool, 2 lists of nat, and 3 lists of nat.*

Terms and functions: semantics is the λ-calculus

Semantics of functional programming languages consists of one rule:

$$(\lambda\, x.\, t)\; a \;\twoheadrightarrow_\beta\; t\{x \mapsto a\} \qquad (\beta\text{-reduction})$$

where $t\{x \mapsto a\}$ is the term t where all occurrences of x are replaced by a

Example 3

- $(\lambda\, x.\, x+1)\; 10 \;\twoheadrightarrow_\beta\; 10+1$
- $(\lambda\, x.\lambda\, y.\, x+y)\; 1\; 2 \;\twoheadrightarrow_\beta\; (\lambda\, y.\, 1+y)\; 2 \;\twoheadrightarrow_\beta\; 1+2$
- $(\lambda\, (x,y).\, y)\; (1,2) \;\twoheadrightarrow_\beta\; 2$

In Isabelle/HOL, to be β-reduced, terms have to be well-typed

Example 4

Previous examples can be reduced because:

- $(\lambda\, x.\, x+1) :: nat \Rightarrow nat$ and $10 :: nat$
- $(\lambda\, x.\lambda\, y.\, x+y) :: nat \Rightarrow nat \Rightarrow nat$ and $1 :: nat$ and $2 :: nat$
- $(\lambda\, (x,y).y) :: ('a \times 'b) \Rightarrow 'b$ and $(1,2) :: nat \times nat$

A word about curried functions and partial application

Definition 5 (Curried function)

A function is *curried* if it returns a function as result.

Example 6

The function $(\lambda\, x.\lambda\, y.\, x+y) :: nat \Rightarrow nat \Rightarrow nat$ is curried
The function $(\lambda\, (x,y).\, x+y) :: nat \times nat \Rightarrow nat$ is *not* curried

Example 7 (Curried function can be partially applied!)

The function $(\lambda\, x.\lambda\, y.\, x+y)$ can be applied to 2 or 1 argument!

- $(\lambda\, x.\lambda\, y.\, x+y)\; 1\; 2 \;\twoheadrightarrow_\beta\; (\lambda\, y.\, 1+y)\; 2 \;\twoheadrightarrow_\beta\; (1+2) :: nat$
- $(\lambda\, x.\lambda\, y.\, x+y)\; 1 \;\twoheadrightarrow_\beta\; (\lambda\, y.\, 1+y) :: nat \Rightarrow nat$ which is a function!

A word about curried functions and partial application (II)

- To associate the value of a term t to a name n ...... definition "n=t"

Exercise 2 (In Isabelle/HOL)

1. *Define the (non-curried) function* `addNc` *adding two naturals*
2. *Use* `addNc` *to add* 5 *to* 6
3. *Define the (curried) function* `add` *addind two naturals*
4. *Use* `add` *to add* 5 *to* 6
5. *Using* `add`*, define the* `incr` *function adding* 1 *to a natural*
6. *Apply* `incr` *to* 5
7. *Define a function* `app1` *adding* 1 *at the beginning of any list of naturals, give an example of use*

A word about higher-order functions

Definition 8 (Higher-order function)

A *higher-order* function takes one or more functions as parameters.

Example 9 (Some higher-order functions and their evaluation)

- $\lambda x. \lambda f.\ f\ x :: 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b$
- $\lambda f. \lambda x.\ f\ x :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
- $\lambda f. \lambda x.\ f\ (x+1)\ (x+1) :: (nat \Rightarrow nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$

  $(\lambda f. \lambda x.\ f\ (x+1)\ (x+1))\ \mathtt{add}\ 20$
  $\twoheadrightarrow_\beta (\lambda x.\ \mathtt{add}\ (x+1)\ (x+1))\ 20$
  $\twoheadrightarrow_\beta \mathtt{add}\ (20+1)\ (20+1)$
  $= (\lambda x. \lambda y.\ x+y)\ (20+1)\ (20+1)$
  $\twoheadrightarrow_\beta (20+1)+(20+1)$
  $= 42$

A word about higher-order functions (II)

Exercise 3 (In Isabelle/HOL)

1. *Define a function* `triple` *which applies three times a given function to an argument*
2. *Using* `triple`*, apply three times the function* `incr` *on* 0
3. *Using* `triple`*, apply three times the function* `app1` *on* `[2,3]`
4. *Using* `map` $:: ('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ *from the list* $[1,2,3]$ *build the list* $[2,3,4]$

Interlude: a word about semantics and verification

- To verify programs, formal reasoning on their semantics is crucial!
- To prove a property ϕ on a program P we need to precisely and exactly understand P's behavior

For many languages the semantics is given by the compiler (version)!

- C, Flash/ActionScript, JavaScript, Python, Ruby, ...

Some languages have a (written) formal semantics:

- Java [a], subsets of C (hundreds of pages)
- Proofs are hard because of semantics complexity (*e.g.* KeY for Java)

[a] http://docs.oracle.com/javase/specs/jls/se7/html/index.html

Some have a small formal semantics:

- Functional languages: Haskell, subsets of (OCaml, F# and Scala)
- Proofs are easier since semantics essentially consists of a single rule

Constructor terms

Isabelle distinguishes between constructor and function symbols

- A function symbol is associated to a function, *e.g.* `inc`
- A constructor symbol is not associated to any function

Definition 10 (Constructor term)

A term containing only constructor symbols is a constructor term

A constructor term does not contain function symbols

Constructor terms (II)

All data are built using constructor terms **without** variables

...even if the representation is generally hidden by Isabelle/HOL

Example 11

- Natural numbers of type `nat` are terms: 0, $Suc(0)$, $Suc(Suc(0))$, ...
- Integer numbers of type `int` are couples of natural numbers: $\ldots(0,2), (0,1), (0,0), (1,0), \ldots$

  where $(0,2) = (1,3) = (2,4) = \ldots$ all represent the *same* integer -2
- Lists are built using the operators
  - *Nil*: the empty list
  - *Cons*: the operator adding an element to the (head) of the list

  The term $Cons\,0\,(Cons\,(Suc\,0)\,Nil)$ represents the list $[0,1]$

Constructor terms: Isabelle/HOL

For most of constructor terms there exists shortcuts:

- Usual decimal representation for naturals, integers and rationals 1, 2, -3, -45.67676, ...
- [] and # for lists, e.g. $Cons\,0\,(Cons\,(Suc\,0)\,Nil) = 0\#(1\#[]) = [0,1]$ (similar to [] and :: of OCaml)
- Strings using 2 quotes e.g. `''toto''` (instead of `"toto"`)

Exercise 4

1. *Prove that 3 is equivalent to its constructor representation*
2. *Prove that $[1,1,1]$ is equivalent to its constructor representation*
3. *Prove that the first element of list $[1,2]$ is 1*
4. *Infer the constructor representation of rational numbers of type* `rat`
5. *Infer the constructor representation of strings*

Isabelle Theory Library

Isabelle comes with a huge library of useful theories

- Numbers: Naturals, Integers, Rationals, Floats, Reals, Complex ...
- Data structures: Lists, Sets, Tuples, Records, Maps ...
- Mathematical tools: Probabilities, Lattices, Random numbers, ...

All those theories include types, functions and lemmas/theorems

Example 12

Let's have a look to a simple one `Lists.thy`:

- Definition of the datatype (with shortcuts)
- Definitions of functions
- Definitions and proofs of lemmas

  e.g. lemma "length (xs @ ys) = length xs + length ys"
- Code exportation rules for SML, Haskell, Ocaml, Scala

Isabelle Theory Library: using functions on lists

Some functions of `Lists.thy`

- append :: "'a list ⇒ 'a list ⇒ 'a list"
- rev :: "'a list ⇒ 'a list"
- length :: "'a list ⇒ nat"
- map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list"

Exercise 5

1. *Apply the* `rev` *function to list $[1,2,3]$*
2. *Show that for all value x, reverse of the list $[x]$ is equal to $[x]$*
3. *Show that* `append` *is associative*
4. *Show that* `append` *is not commutative*
5. *Using* `map`*, from the list $[1,2,3]$ build the list $[2,4,6]$*
6. *Prove that* `map` *does not change the size of a list*

Outline

1. Terms
   - Types
   - Typed terms
   - λ-terms
   - Constructor terms
2. Functions defined using equations
   - Logic everywhere!
   - Function evaluation using term rewriting
   - Partial functions

Defining functions using equations

- Defining functions using λ-terms is hardly usable for programming
- Isabelle/HOL has a "`fun`" operator as other functional languages

Definition 13 (`fun` operator for defining (recursive) functions)

`fun` f :: "$\tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$"
`where`
" $f\ t_1^1 \ldots t_n^1 \quad = \quad r^1$ " |
...
" $f\ t_1^m \ldots t_n^m \quad = \quad r^m$ "

where for all $i = 1 \ldots n$ and $k = 1 \ldots m$ $(t_i^k :: \tau_i)$ are constructor terms **possibly with** variables, and $(r^k :: \tau)$

Example 14 (The `member` function on lists (2 versions in `cm2.thy`))

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


Total and partial Isabelle/HOL functions

Definition 15 (Total and partial functions)

A function is *total* if it has a value (a result) for all element of its domain. A function is *partial* if it is not total.

Definition 16 (Complete Isabelle/HOL function definition)

`fun` f :: "$\tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$"
`where`
" $f\ t_1^1 \ldots t_n^1 \quad = \quad r^1$ " |
...
" $f\ t_1^m \ldots t_n^m \quad = \quad r^m$ "

f is *complete* if any call $f\ t_1 \ \ldots\ t_n$ with $(t_i :: \tau_i)$, $i = 1 \ldots n$ is covered by one case of the definition.

Example 17 (Isabelle/HOL "`Missing patterns`" warning)

When the definition of f is not complete, an uncovered call of f is shown.

Total and partial Isabelle/HOL functions (II)

Theorem 18

Complete and terminating Isabelle/HOL functions are total, otherwise they are partial.

Question 1

Why termination of f is necessary for f to be total?

Outline

1. Terms
   - Types
   - Typed terms
   - λ-terms
   - Constructor terms
2. Functions defined using equations
   - Logic everywhere!
   - Function evaluation using term rewriting
   - Partial functions

Acknowledgements: some slides are borrowed from T. Nipkow's lectures

Logic everywhere!

In the end, everything is defined using logic:

- data, data structures: constructor terms
- properties: lemmas (logical formulas)
- programs: functions (also logical formulas!)

Definition 19 (Equations (or simplification rules) defining a function)

A function `f` consists of a set of `f.simps` of equations on terms.

To visualize a lemma/theorem/simplification rule .... `thm`

For instance: `thm "length_append"`, `thm "append.simps"`

To *find* the name of a lemma, etc. .... `find_theorems`

For instance: `find_theorems "append" "_ + _"`

Exercise 6

Use Isabelle/HOL to find the following formulas:

- *definition of* `member` *(we just defined) and of* `nth` *(part of List.thy)*
- *find the lemma relating* `rev` *(part of List.thy) and* `length`

Evaluation= Rewriting using equations

Recall that definition of the function `member` consists of the 2 equations:

```
(1) member e []         = False
(2) member e (x # xs)= (if e=x then True else (member e xs))
```

How to use those equations to evaluate the term `(member 2 [1,2,3])`?

Definition 20 (Substitution)

A substitution σ is a function replacing variables of $\mathcal{X}$ by terms of $\mathcal{T}(\mathcal{F},\mathcal{X})$ in a term of $\mathcal{T}(\mathcal{F},\mathcal{X})$.

Example 21

Let $\mathcal{F} = \{f:3, h:1, g:1, a:0\}$ and $\mathcal{X} = \{x,y,z\}$.
Let $\sigma = \{x \mapsto g(a), y \mapsto h(z)\}$ and $t = f(h(x), x, g(y))$. We have $\sigma(t) = f(h(g(a)), g(a), g(h(z)))$.

Evaluation= Rewriting using equations (II)

Definition 22 (Rewriting using an equation)

A term s can be *rewritten* into the term t (denoted by $s \twoheadrightarrow t$) using an Isabelle/HOL equation `l=r` if there exists a subterm u of s and a substitution σ such that $u = \sigma(\mathtt{l})$. Then, t is the term s where subterm u has been replaced by $\sigma(\mathtt{r})$.

Example 23

Let $s = f(g(a), c)$ and `g(x) = h(g(x),b)` the Isabelle/HOL equation.

we have	$f(\ g(a)\ ,c)$	$\twoheadrightarrow$	$f(\ h(g(a),b)\ ,c)$
because	`g(x)`	=	`h(g(x),b)` and $\sigma = \{\mathtt{x} \mapsto a\}$

Remark 1

Isabelle/HOL rewrites terms using equations in the order of the function definition and only from left to right.

Evaluation= Rewriting using equations (III)

```
(1) member e []         = False
(2) member e (x # xs)= (if e=x then True else (member e xs))
```

Evaluation of test: `member 2 [1,2,3]`

- ↠ `if 2=1 then True else (member 2 [2,3])`
  by equation (2), because `[1,2,3] = 1#[2,3]`
- ↠ `if False then True else (member 2 [2,3])`
  by Isabelle equations defining equality on naturals
- ↠ `member 2 [2,3]`
  by Isabelle equation `(if False then x else y = y)`
- ↠ `if 2=2 then True else (member 2 [3])`
  by equation (2), because `[2,3] = 2#[3]`
- ↠ `if True then True else (member 2 [3])`
  by Isabelle equations defining equality on naturals
- ↠ `True`
  by Isabelle equation `(if True then x else y = x)`

Lemma simplification= Rewriting + Logical deduction

```
(1) member e []         = False
(2) member e (x # xs)= (if e=x then True else (member e xs))
```

Proving the lemma: `member y [z,y,v]`

- ↠ `if y=z then True else (member y [y,v])`
  by equation (2), because `[z,y,v] = z#[y,v]`
- ↠ `if y=z then True else (if y=y then True else (member y [v]))`
  by equation (2), because `[y,v] = y#[v]`
- ↠ `if y=z then True else (if True then True else (member y [v])`
  because `y=y` is trivially True
- ↠ `if y=z then True else True`
  by Isabelle equation `(if True then x else y = x)`
- ↠ `True`
  by logical deduction `(if b then True else True)`⟷`True`

Lemma simplification= Rewriting + Logical deduction (II)

```
(1) member e []          = False
(2) member e (x # xs)    = (if e=x then True else (member e xs))

(3) append [] x          = x
(4) append (x # xs) y    = x # (append xs y)
```

Exercise 7

Is it possible to prove the lemma `member u (append [u] v)` *by simplification?*

Exercise 8

Is it possible to prove the lemma `member v (append u [v])` *by simplification?*

Evaluation of partial functions

Evaluation of partial functions using rewriting by equational definitions may not result in a constructor term

Exercise 9

Let `index` *be the function defined by:*

```
fun index:: "'a => 'a list => nat"
where
"index y (x#xs) = (if x=y then 0 else 1+(index y xs))"
```

- *Define the function in Isabelle/HOL*
- *What does it computes?*
- *Why is* `index` *a partial function? (What does Isabelle/HOL says?)*
- *For* `index`*, give an example of a call whose result is:*
  - *a constructor term*
  - *a match failure*
- *Define the property relating functions* `index` *and* `List.nth`

Scala export + Demo

To export functions to Haskell, SML, Ocaml, Scala . . . . export_code

For instance, to export the member and index functions to Scala:

```
export_code member index in Scala file "test.scala"
```

test.scala

```
object cm2 {
  def member[A : HOL.equal](e: A, x1: List[A]): Boolean =
  (e, x1) match {
    case (e, Nil) => false
    case (e, x :: xs) => (if (HOL.eq[A](e, x)) true
                          else member[A](e, xs))
  }
  def index[A : HOL.equal](y: A, x1: List[A]): Nat =
  (y, x1) match {
    case (y, x :: xs) =>
      (if (HOL.eq[A](x, y)) Nat(0)
       else Nat(1) + index[A](y, xs))
```