Analyse et Conception Formelles

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

$$
\begin{array}{lcll}
\tau & ::= & (\tau) & \\
 & | & bool \mid nat \mid char \mid \ldots & \text{base types} \\
 & | & 'a \mid 'b \mid \ldots & \text{type variables} \\
 & | & \tau \Rightarrow \tau & \text{functions} \\
 & | & \tau \times \ldots \times \tau & \text{tuples (ascii for } \times\text{: } *) \\
 & | & \tau\ list & \text{lists} \\
 & | & \ldots & \text{user-defined types}
\end{array}
$$

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

$nat \Rightarrow nat \Rightarrow bool$ is equivalent to $nat \Rightarrow (nat \Rightarrow bool)$

Typed terms: syntax

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

Function application is left-associative, for instance:

$f\ a\ b\ c$ 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

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 $(1,2) :: nat \times nat$

Lambda-calculus – the quiz

Quiz 1

- *Function $\lambda(x,y).\, x$ is a function with two parameters*

V	*True*	R	*False*

- *Type of function $\lambda(x,y).\, x$ is*

V	`'a × 'b ⇒ 'a`
R	`'a ⇒ 'b ⇒ 'a`

- *If* `f::nat ⇒ nat ⇒ nat` *how to call* `f` *on* `1` *and* `2`*?*

V	`f(1,2)`	R	`(f 1 2)`

- *If* `f::nat × nat ⇒ nat` *how to call* `f` *on* `1` *and* `2`*?*

V	`f(1,2)`	R	`(f 1 2)`

Exercises on function definition and function call

Exercise 1 (In Isabelle/HOL)

Use `append::'a list ⇒ 'a list ⇒ 'a list` *to concatenate 2 lists of nat, and 3 lists of nat.*

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

Exercise 2 (In Isabelle/HOL)

1. *Define the function* `addNc`:: $nat \times nat \Rightarrow nat$ *adding two naturals*
2. *Use* `addNc` *to add* 5 *to* 6
3. *Define the function* `add`:: $nat \Rightarrow nat \Rightarrow nat$ *adding two naturals*
4. *Use* `add` *to add* 5 *to* 6

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 (computable) function:
  - all predefined function, *e.g.*, `append`
  - all user defined functions, *e.g.*, `addNc` and `add` (see Exercise 2)
- A constructor symbol is **not** associated to a function

Definition 5 (Constructor term)

A **term** containing only constructor symbols is a constructor term.
A constructor term does not contain function symbols

Example 6

- Term $[0, 1, 2]$ is a constructor term;
- Term (append [0,1,2] [4,5]) is **not** a constructor term (because of append);
- Term 18 is a constructor term;
- Term (add 18 19) is **not** a constructor term (because of add).

Constructor terms (II)

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

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

Example 7

- Natural numbers of type `nat` are terms: 0, (*Suc* 0), (*Suc* (*Suc* 0)), ...
- Integer numbers of type `int` are couples of natural numbers:
  $\ldots(0, 2), (0, 1), (0, 0), (1, 0), \ldots$ represent $\ldots -2, -1, 0, 1 \ldots$
- 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 symbols have types even if they do **not** "compute"

Example 8 (The type of constructor *Cons*)

Cons`::'a ⇒ 'a list ⇒ 'a list`

Constructor terms – the quiz

Quiz 2

- *Nil is a term?* — V True | R False
- *Nil is a constructor term?* — V True | R False
- *(Cons (Suc 0) Nil) is a constructor term?* — V True | R False
- *((Suc 0), Nil) is a constructor term?* — V True | R False
- *(add 0 (Suc 0)) is a constructor term?* — V True | R False
- *(Cons x Nil) is a constructor term?* — V True | R False
- *(add x y) is a constructor term?* — V True | R False
- *(Suc 0) is a constructor subterm of (add 0 (Suc 0))?* — V True | R False

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. $Cons\ 0\ (Cons\ (Suc\ 0)\ Nil) \quad = \quad 0\#(1\#[]) \quad = \quad [0, 1]$
- Strings using 2 quotes e.g. `''toto''` (instead of `"toto"`)

Exercise 3

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 9

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

- Definition of the datatype (with shortcuts)
- Definitions of functions (e.g. `append`)
- Definitions and proofs of lemmas (e.g. `length_append`)
  lemma "length (xs @ ys) = length xs + length ys"
- Exportation rules for SML, Haskell, Ocaml, Scala (`code_printing`)

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`
- `List.member:: 'a list ⇒ 'a ⇒ bool`
- `map:: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list`

Exercise 4

1. *Apply the* `rev` *function to list* $[1, 2, 3]$
2. *Prove that for all value* x*, reverse of the list* $[x]$ *is equal to* $[x]$
3. *Prove that* `append` *is associative*
4. *Prove that* `append` *is not commutative*
5. *Prove that an element is in a reversed list if it is in the original one*
6. *Using* `map`*, from the list* $[(1, 2), (3, 3), (4, 6)]$ *build the list* $[3, 6, 10]$
7. *Using* `map`*, from the list* $[1, 2, 3]$ *build the list* $[2, 4, 6]$
8. *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 10 (fun operator for defining (recursive) functions)

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

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

Example 11 (The contains function on lists (2 versions in cm2.thy))

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


Function definition – the quiz

Quiz 3 (Is this function definition correct? V Yes R No)

```
fun f:: "nat ⇒ nat ⇒ bool"
where
"f x y = (x + y)"
```

Quiz 4 (Is this function definition correct? V Yes R No)

```
fun g:: "nat ⇒ nat ⇒ bool"
where
"g 0 y = False"
```

Quiz 5 (Is this function definition correct? V Yes R No)

```
fun pos:: "nat ⇒ bool"
where
"pos 0 = False" |
"pos (Suc x) = True"
```


Function definition – the quiz (II)

Quiz 6 (Is this function definition correct? V Yes R No)

```
fun pos2:: "nat ⇒ bool"
where
"pos2 0 = False" |
"pos2 (x + 1) = True"
```

Quiz 7 (Is this function definition correct? V Yes R No)

```
fun isDivisor:: "nat ⇒ nat ⇒ bool"
where
"isDivisor x y = (∃ z. x * z = y)"
```


Total and partial Isabelle/HOL functions

Definition 12 (Total and partial functions)

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

Definition 13 (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 14 (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 15

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?

Remark 1

All functions in Isabelle/HOL needs to be terminating!

Outline

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 16 (Equations (or simplification rules) defining a function)

A function `f` consists of a set `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 5

Use Isabelle/HOL to find the following formulas:

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

Evaluating functions by rewriting terms using equations

The append function (aliased to `@`) is defined by the 2 equations:

```
(1) append    Nil  x = x                  (* recall that Nil=[] *)
(2) append (x#xs)  y = (x#(append xs y))
```

Replacement of equals by equals = Term rewriting

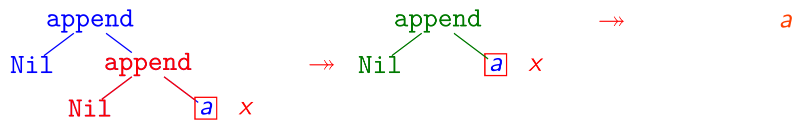
The first equation `(append Nil x) = x` means that

- *(concatenating the empty list with any list x)* is **equal** to *x*
- we can thus replace
  - any term of the form `(append Nil t)` by `t` (for any value `t`)
  - wherever and whenever we encounter such a term `append Nil t`

Term Rewriting in three slides

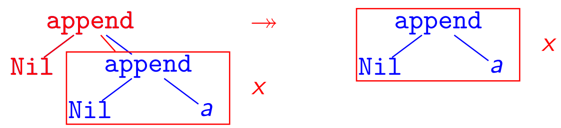
- Rewriting term `(append Nil (append Nil a))` using

```
(1) append     Nil  x  =  x
(2) append (x#xs)   y  =  (x#(append xs y))
```



- We note `(append Nil (append Nil a))` $\twoheadrightarrow$ `(append Nil a)` if
  - there exists a position in the term where the rule matches
  - there exists a substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ for the rule to match.
    On the example $\sigma = \{x \mapsto a\}$
- We also have `(append Nil a)` $\twoheadrightarrow$ `a`

and



Term Rewriting in three slides – Formal definitions

Definition 17 (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 18

Let $\mathcal{F} = \{f : 3, h : 1, g : 1, a : 0\}$ and $\mathcal{X} = \{x, y, z\}$.

Let σ be the substitution $\sigma = \{x \mapsto g(a), y \mapsto h(z)\}$.

Let $t = f(h(x), x, g(y))$.

We have $\sigma(t) = f(h(g(a)), g(a), g(h(z)))$.

Term Rewriting in three slides – Formal definitions (II)

Definition 19 (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 $u = \sigma(\texttt{l})$. Then, t is the term s where subterm u has been replaced by $\sigma(\texttt{r})$.

Example 20

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 $\sigma = \{\texttt{x} \mapsto a\}$

On the opposite $t = f(a, c)$ cannot be rewritten by `g(x) = h(g(x),b)`.

Remark 2

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

Term rewriting – the quiz

Quiz 8

Let $\mathcal{F} = \{f : 1, g : 1, a : 0\}$ and $\mathcal{X} = \{x, y\}$.

- *Rewriting the term $f(g(g(a)))$ with equation $g(x) = x$ is*
  V Possible | R Impossible
- *To rewrite the term $f(g(g(a)))$ with $g(x) = x$ the substitution σ is*
  V $\{x \mapsto a\}$ | R $\{x \mapsto g(a)\}$
- *Rewriting the term $f(g(g(y)))$ with equation $g(x) = x$ is*
  V Possible | R Impossible
- *Rewriting the term $f(g(g(y)))$ with equation $g(f(x)) = x$ is*
  V Possible | R Impossible

Isabelle evaluation = rewriting terms using equations

```
(1) append    Nil   x   =   x
(2) append (x#xs)   y   =   (x#(append xs y))
```

Rewriting the term: `append [1,2] [3,4]` with `(1)` then `(2)` (Rmk 2)

First, recall that `[1,2] = (1#(2#Nil))` and `[3,4] = (3#(4#Nil))`!

`append (1#(2#Nil)) (3#(4#Nil))`
`(1# (append (2#Nil) (3#(4#Nil))))}`
with $\sigma = \{x \mapsto 1, xs \mapsto (2\#Nil), y \mapsto (3\#(4\#Nil))\}$
	$\not\twoheadrightarrow_{(1)}$ $\twoheadrightarrow_{(2)}$

`(1# (append (2#Nil) (3#(4#Nil))))`
`(1# (2#(append Nil (3#(4#Nil)))))`
with $\sigma = \{x \mapsto 2, xs \mapsto Nil, y \mapsto (3\#(4\#Nil))\}$
	$\twoheadrightarrow_{(2)}$

`(1#(2# (append Nil (3#(4#Nil)))))` `(1#(2# (3#(4#Nil)) ))` = `[1,2,3,4]` ! with $\sigma = \{x \mapsto (3\#(4\#Nil))\}$	$\twoheadrightarrow_{(1)}$

Example 21

See demo of step by step rewriting in Isabelle/HOL!

Isabelle evaluation = rewriting terms using equations (II)

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

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

$\twoheadrightarrow$ `if 2=1 then True else (contains 2 [2,3])`
by equation `(2)`, because `[1,2,3] = 1#[2,3]`

$\twoheadrightarrow$ `if False then True else (contains 2 [2,3])`
by Isabelle equations defining equality on naturals

$\twoheadrightarrow$ `contains 2 [2,3]`
by Isabelle equation `(if False then x else y = y)`

$\twoheadrightarrow$ `if 2=2 then True else (contains 2 [3])`
by equation `(2)`, because `[2,3] = 2#[3]`

$\twoheadrightarrow$ `if True then True else (contains 2 [3])`
by Isabelle equations defining equality on naturals

$\twoheadrightarrow$ `True`
by Isabelle equation `(if True then x else y = x)`

Lemma simplification= Rewriting + Logical deduction

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

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

$\twoheadrightarrow$ `if y=z then True else (contains y [y,v])`
by equation `(2)`, because `[z,y,v] = z#[y,v]`

$\twoheadrightarrow$ `if y=z then True else (if y=y then True else (contains y [v]))`
by equation `(2)`, because `[y,v] = y#[v]`

$\twoheadrightarrow$ `if y=z then True else (if True then True else (contains y [v]))`
because `y=y` is trivially `True`

$\twoheadrightarrow$ `if y=z then True else True`
by Isabelle equation `(if True then x else y = x)`

$\twoheadrightarrow$ `True`
by logical deduction `(if b then True else True)`$\longleftrightarrow$`True`

Lemma simplification= Rewriting + Logical deduction (II)

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

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

Exercise 6

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

Exercise 7

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

Demo of rewriting in Isabelle/HOL!

Evaluation of partial functions

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

Exercise 8

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 `contains` and `index` functions to Scala:

```
export_code contains index in Scala
```

test.scala

```
object cm2 {
  def contains[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 contains[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))
  }
```