### Analyse et Conception Formelles

**Lesson 2**

Types, terms and functions

---

### Outline

- **Terms**
  - Types
  - Typed terms
  - $\lambda$-terms
  - Constructor terms

- **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

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

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

\[
\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool} \quad \text{is equivalent to} \quad \text{nat} \Rightarrow (\text{nat} \Rightarrow \text{bool})
\]

---

### Typed terms: syntax

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

Function application is left-associative, for instance:

\[
f a b c \quad \text{is equivalent to} \quad ((f a) b) c
\]

---

### Example 1 (Types of terms)

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>$'a$</td>
</tr>
<tr>
<td>$(t1,t2,t3)$</td>
<td>$('a \times 'b \times 'c)$</td>
</tr>
<tr>
<td>$\lambda y. y$</td>
<td>$'a \Rightarrow 'a$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1$</td>
<td>$[t1,t2,t3]$</td>
</tr>
<tr>
<td>$\lambda y. z. z$</td>
<td>$'a \Rightarrow 'b \Rightarrow 'b$</td>
</tr>
</tbody>
</table>
Types and terms: evaluation in Isabelle/HOL

To evaluate a term \( t \) in Isabelle/HOL, \( \text{value } "t" \)

Example 2

<table>
<thead>
<tr>
<th>Term</th>
<th>Isabelle's answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>value &quot;True&quot;</td>
<td>Error (cannot infer result type)</td>
</tr>
<tr>
<td>value &quot;2&quot;</td>
<td>2::nat</td>
</tr>
<tr>
<td>value &quot;(2::nat)&quot;</td>
<td>Error (cannot infer result type)</td>
</tr>
<tr>
<td>value &quot;[True,False]&quot;</td>
<td>[True,False]::bool list</td>
</tr>
<tr>
<td>value &quot;([2,6,10])&quot;</td>
<td>Error (cannot infer result type)</td>
</tr>
<tr>
<td>value &quot;([2::nat],6,10)&quot;</td>
<td>[2,6,10]::nat list</td>
</tr>
</tbody>
</table>

Exercise 1 (In Isabelle/HOL)

Use append to concatenate 2 lists of bool, 2 lists of nat, and 3 lists of 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 \( (\lambda x.\lambda y. x + y) :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \) is curried.

The function \( (\lambda (x,y). x + y) :: \text{nat} \times \text{nat} \Rightarrow \text{nat} \) 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:

- \( (\lambda x.\lambda y. x + y) 1 \ 2 \Rightarrow (\lambda y. 1 + y) 2 \Rightarrow (1 + 2) :: \text{nat} \)
- \( (\lambda x.\lambda y. x + y) 1 \Rightarrow (\lambda y. 1 + y) :: \text{nat} \Rightarrow \text{nat} \) which is a function!

Terms and functions: semantics is the \( \lambda \)-calculus

Semantics of functional programming languages consists of one rule:

\[
(\lambda x. t) \ a \Rightarrow t \{x \mapsto a\}
\]

(\( \beta \)-reduction)

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

Example 3

\[
(\lambda x. x + 1) 10 \Rightarrow 10 + 1
\]

\[
(\lambda x.\lambda y. x + y) 1 \ 2 \Rightarrow (\lambda y. 1 + y) 2 \Rightarrow 1 + 2
\]

\[
(\lambda (x,y). y) (1,2) \Rightarrow 2
\]

In Isabelle/HOL, to be \( \beta \)-reduced, terms have to be well-typed

Example 4

Previous examples can be reduced because:

- \( (\lambda x. x + 1) :: \text{nat} \Rightarrow \text{nat} \) and \( 10 :: \text{nat} \)
- \( (\lambda x.\lambda y. x + y) :: \text{nat} \Rightarrow \text{nat} \) and \( 1 :: \text{nat} \) and \( 2 :: \text{nat} \)
- \( (\lambda (x,y). y) :: (\text{nat} \times \text{nat}) \Rightarrow \text{nat} \) and \( (1,2) :: \text{nat} \times \text{nat} \)

A word about curried functions and partial application (II)

Exercise 2 (In Isabelle/HOL)

1. Define the (non-curried) function \( \text{addNc} \) adding two naturals
2. Use \( \text{addNc} \) to add 5 to 6
3. Define the (curried) function \( \text{add} \) adding two naturals
4. Use \( \text{add} \) to add 5 to 6
5. Using \( \text{add} \), define the \( \text{incr} \) function adding 1 to a natural
6. Apply \( \text{incr} \) to 5
7. Define a function \( \text{app1} \) 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)
\[
\lambda x. \lambda f. f x :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}
\]

\[
(\lambda f. \lambda x. f (x+1)(x+1)) \quad \text{add}
\]

\[
\Rightarrow^\beta (\lambda x. x + y)(20 + 1)(20 + 1)
\]

\[
= (20 + 1) + (20 + 1)
\]

\[
= 42
\]

Interlude: a word about semantics and verification
- To verify programs, formal reasoning on their semantics is crucial!
- To prove a property \( \phi \) 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 *, subsets of C (hundreds of pages)
- Proofs are hard because of semantics complexity (e.g. KeY for Java)

*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

Exercise 3 (In Isabelle/HOL)
- Define a function \text{triple} which applies three times a given function to an argument
- Using \text{triple}, apply three times the function \text{incr} on 0
- Using \text{triple}, apply three times the function \text{app1} on \([2,3]\)
- Using \text{map} :: \((\text{a} \Rightarrow \text{b}) \Rightarrow \text{a list} \Rightarrow \text{b list}\) from the list \([1,2,3]\) build the list \([2,3,4]\)

Constructor terms

Isabelle distinguishes between constructor and function symbols
- A function symbol is associated to a function, e.g. \text{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: (...0, 2), (0, 1), (0, 0), (1, 0), ...
  where (0, 2) = (1, 3) = (2, 4) = ... 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 (e.g. append)
- Definitions and proofs of lemmas (e.g. 1length_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"
- map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list"

Exercise 5

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. 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
   - $\lambda$-terms
   - Constructor terms

2. Functions defined using equations
   - Logic everywhere!
   - Function evaluation using term rewriting
   - Partial functions

Defining functions using equations

- Defining functions using $\lambda$-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

\[ f \ t_1 \ldots t_n = r_1 \]  

where for all $i = 1 \ldots n$ and $k = 1 \ldots m$

\[ f \ t_1^k \ldots t_n^k = r_m \]  

where $(t_i :: \tau_i)$ are constructor terms possibly with variables, and $(r_i :: \tau)$

**Example 14 (The member function on lists)**

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 elements 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

\[ f \ t_1 \ldots t_n = r_1 \]  

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?

**Remark 1**

All functions in Isabelle/HOL needs to be terminating!
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 \( \texttt{f.simps} \) of equations on terms.

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

For instance: \texttt{thm "length_append"}, \texttt{thm "append.simps"}

To find the name of a lemma, etc. . . . . . . . . . . . . . . . . . . . . . . . . . .

For instance: \texttt{find theorems "append" "+"}

Exercise 6

Use Isabelle/HOL to find the following formulas:
- definition of \texttt{member} (we just defined) and of \texttt{nth} (part of List.thy)
- find the lemma relating \texttt{rev} (part of List.thy) and \texttt{length}

Evaluation= Rewriting using equations

Recall that definition of the function \texttt{member} consists of the 2 equations:

1. \texttt{member e [] = False}
2. \texttt{member e (x # xs)= (if e=x then True else (member e xs))}

How to use those equations to evaluate the term \texttt{(member 2 [1,2,3])}?

Definition 20 (Substitution)

A substitution \( \sigma \) is a function replacing variables of \( \mathcal{X} \) by terms of \( T(\mathcal{F}, \mathcal{X}) \) in a term of \( T(\mathcal{F}, \mathcal{X'}) \).

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

Remark 2

Isabelle/HOL rewrites terms using equations in the order of the function definition and only from \( \texttt{left to right}. \)

Definition 22 (Rewriting using an equation)

A term \( s \) can be rewritten into the term \( t \) (denoted by \( s \rightsquigarrow t \)) using an Isabelle/HOL equation \( \mathbf{1=s} \) if there exists a subterm \( u \) of \( s \) and a substitution \( \sigma \) such that \( u = \sigma(1) \). Then, \( t \) is the term \( s \) where subterm \( u \) has been replaced by \( \sigma(1) \).

Example 23

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

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

T. Genet (ISTIC/IRISA)  ACF-2  23 / 29
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

Evaluation of partial functions

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

Exercise 7
Is it possible to prove the lemma member u (append [u] v) by simplification/rewriting?

Exercise 8
Is it possible to prove the lemma member v (append u [v]) by simplification/rewriting?
Scala export + Demo

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

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

```scala
export_code member index in 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))
        }
}
```