Chapter 2

Functional Programming and Modeling

Overview of Chapter

Functional programming and modeling

1. Review of functional programming
2. Functional modeling in Isabelle/HOL
3. A simple theorem prover, substitution, and unification

HOL = Functional programming + Logic

» Chapter 2 and 3 of Isabelle/HOL Tutorial

Section 2.1

Overview

Functional programming

Fact

A functional program consists of

- *function declarations*
- *data type declarations*
- *an expression*

Functional programs

- do not have variables, assigments, statements, loops, ...
- instead:
  - let-expressions
  - recursive functions
  - higher-order functions

Functional programming

Advantages

- state-independent semantics
- corresponds more directly to abstract mathematical objects
- can express “computational” and “static” aspects

Functional programming and modeling

Functional programming

- function definitions describe execution plan
- functions may be partial
- exception-handling mechanisms

Functional modeling

- function definitions play two roles:
  - representing programs (as above)
  - used to express properties
- functions have to be total

Functional programming and modeling (2)

Common aspects

- recursive definition is central for functions and data
- strongly typed with:
  - type inference
  - parametric polymorphism
  - type classes

Section 2.2

Functional Programming in Isabelle/HOL

Subsection 2.2.1

Primitive datatypes and definitions

Primitive datatypes

Example (Constant definition and use)

```
definition k :: int where "k ≡ 7"

value k
value "k+1"

definition m :: nat where "m ≡ 7"

definition bv :: bool where "bv ≡ True"
```


Overloading of literals

```
value "37+m"     -- nat
value "37+k"     -- int
value 37         -- 'a
```

Non-recursive function definitions

Example (Simple functions)

```
definition inc :: "int ⇒ int" where
"inc j ≡ j+1"

value "inc 1234567890"


definition nand :: "bool ⇒ bool ⇒ bool" where
"nand A B ≡ ¬ (A ∧ B)"

value "nand (¬ bv) False"
```


Higher-order functions

Example (Simple higher-order function)

```
definition appl2 :: "(int ⇒ int) ⇒ int ⇒ int" where
"appl2 f j = f (f j)"

value "appl2 inc (-5)"
```

Example (Lambda abstraction)

```
value "appl2 (λ x::int. x+1) k"

definition plusN :: "int ⇒ (int ⇒ int)" where
"plusN j = (λ x::int. x+j)"

value "plusN 3"
value "plusN 3 45"
```


Pairs, tuples, type “unit”

Example (Pairs)

```
value "(k,inc 7)"              -- "type is int × int"
value "fst (snd (True,(False,bv)))"
```

Example (Tuples)

```
-- "Tuples are realized as pairs nested to the right"

value "(True,True,True) = (True,(True,True))"
```

Example (Type “unit”)

```
value "()"
-- "denotes the only and unique element of type unit"
```


Function arguments

Functions have one argument!

Example (A pair as argument)

```
definition nand2 :: "(bool × bool) ⇒ bool" where
"nand2 PAB ≡ ¬ ((fst PAB) ∧ (snd PAB))"

value "nand2 (¬ bv,False)"
```

Remark

Goal is

- not to execute functions
- but to prove properties

However, functional programs can be generated.

Example (Property formulated as lemma)

```
lemma "nand x y  ≡ nand2 (x,y)"

by (simp add: nand_def nand2_def)
```


Subsection 2.2.2

Type definitions and recursive functions

Type system

- Primitive types
- Predefined type constructor $\alpha \Rightarrow \beta$
- User-defined types and type constructors (“datatype definitions”)
- Type synonyms

Type synonym

```
type_synonym  nat2nat = "nat ⇒ nat"

definition double :: nat2nat where
"double n ≡ 2*n"
```

Datatype definition

```
datatype weekday =
    Mon | Tue | Wed | Thu | Fri | Sat | Sun

lemma "∀ x. x=Mon ∨ x=Tue ∨ x=Wed ∨ x=Thu ∨ x=Fri
                ∨ x=Sat ∨ x=Sun"

apply clarify
apply (case_tac x)
apply auto
done
```


Recursive datatype definition

```
datatype plformula  =  Var   string
                    |  TTrue
                    |  FFalse
                    |  Not   plformula
                    |  Imp   plformula  plformula

value "Imp (Var ''x'') TTrue"
```

Remark

Recursive datatypes are in particularly used to represent the abstract syntax of languages.

Primitive recursive function definitions

Definition

A recursive function definition of f in which

- one argument is of a datatype dt and
- all equations are of the form:

$$f\ x_1 \ldots (C\ y_1 \ldots y_k) \ldots x_n = R$$

where C is a constructor of dt and all recursive calls of f in R may only refer to y_i, i.e., $(f \ldots y_i \ldots)$ for some i, are called primitive recursive.

Remark

For primitive recursive definitions, it is easy to show that the defined function is total/well-defined (“terminates”)

Primitive recursive function definitions

Example

```
primrec varfree :: "plformula ⇒ bool" where
  "varfree (Var s)     =  False"
| "varfree TTrue       =  True"
| "varfree FFalse      =  True"
| "varfree (Not p)     =  varfree p"
| "varfree (Imp p q)   =  ((varfree p) ∧ (varfree q))"

value "varfree (Imp (Var ''x'') TTrue)"
value "varfree (Imp FFalse TTrue))"
```

Datatypes, nat, and primitive recursion

Remark

nat is a defined as datatype with constructors 0 and Suc in Isabelle/HOL.

```
primrec pow :: "nat ⇒ nat ⇒ nat" where
  "pow b 0       =  1"
| "pow b (Suc e) =  ((pow b e) * b)"

value "pow 6 7"
```


More general forms of recursion

Example

```
fun even :: "nat ⇒ bool" where
  "even 0             =  True"
| "even (Suc (Suc n)) =  even n"
| "even _             =  False"
```

Remarks

- Isabelle/HOL supports more general forms of recursive function definition.
- In general, the user has to *verify* that the function is well-defined (see below and later chapters).

Subsection 2.2.3

Parameterized datatypes

Polymorphism

Motivation

To increase reuse, type systems in funtional languages usually support parameterized types, i.e., the definition and use of types that have type parameters.

Example (Pair type with two parameters)

```
datatype ('a,'b) pair = MkPair 'a 'b
```

Parameterized list datatype

```
datatype 'a list = Nil                    ("[]")
                 | Cons 'a "'a list"      (infixr "#" 65)

primrec app::"'a list=>'a list=>'a list" (infixr "@1" 65)
where
"[] @1 ys = ys" |
"(x # xs) @1 ys = x # (xs @1 ys)"

value "rev ([True] @1 [True,False])"


primrec rev :: "'a list => 'a list" where
"rev [] = []" |
"rev (x # xs) = (rev xs) @ (x # [])"
```


Digression: properties

```
lemma rev_app [simp]:
"rev (xs@ys) = (rev ys)@(rev xs)"

by (induct xs) auto

------------------------------------------------------

lemma rev2 [simp]:
"rev (rev xs) = xs"

by (induct xs) auto
```


Parameterized tree datatype

```
datatype 'a btree  =  Tip
                   |  Node "'a btree" 'a  "'a btree"


primrec consbtree :: "nat ⇒ 'a ⇒ 'a btree" where
   "consbtree 0 x         =   Tip"
 | "consbtree (Suc n) x   =
          ( Node (consbtree n x) x (consbtree n x) )"


primrec countnodes :: "'a btree ⇒ nat" where
   "countnodes Tip           =  0"
 | "countnodes (Node l x r) =
          ( (countnodes l) + 1 + (countnodes r) )"
```


Digression: properties

```
lemma pow0 [simp]: "0 < pow 2 x"

by (induct x) auto


lemma "countnodes (consbtree n x) = (pow 2 n) - (1::nat)"

by (induct n) auto
```

Well-definedness of functions

Keep in mind:

- Isabelle/HOL only supports total functions.
- Well-definedness has to be proven:
  - automatically: keywords “primrec”, “fun”
  - interactively: keyword “function”

Well-definedness of functions (2)

Example

Automatic proof fails for:

```
fun consbtree2 :: "nat ⇒ 'a ⇒ 'a btree" where
   "consbtree2 0 x        =  Tip"
 | "consbtree2 (Suc n) x  =
     (if even n
        then  Node (consbtree2 (n div 2) x) x
                   (consbtree2 (n div 2) x)
        else  Node (consbtree2 (n div 2) x) x
                   (consbtree2 ((n div 2)+1) x))"
```


Handling of partial functions

Techniques:

Let D be the domain type and R be the range/result type of the function.
Alternatives to handle partiality:

- use some “defined” value of R as dummy result
- use the specific constant “undefined” that is member of all types
- change the range type into “R `option`”
- use a relation

Illustration of alternatives 1 and 2

```
value "(1::nat) div 0"    (* 1. alternative *)

(* 2. alternative *)
primrec head :: "'a list ⇒ 'a" where
   "head (x#xs) = x"
|  "head []     = undefined"

value "head []"

(*  Automatic completion if pattern list is incomplete *)
primrec head2 :: "'a list ⇒ 'a" where
   "head2 (x#xs) = x"

value "head2 []"
```

Digression: the constant “undefined”

```
datatype myunit = MyUnity

lemma unity: "x = MyUnity"
by (case_tac x)

lemma "undefined = MyUnity"
by (rule unity)
```


Illustration of alternative 3

```
datatype 'a option = None | Some 'a

primrec head3 :: "'a list ⇒ 'a option" where
   "head3 (x#xs) = Some x"
|  "head3 []     = None"

value "head3 []"

value "head3 ([]:: nat list)"
```


Equality

Equality on function types

Unlike in functional programming, equality is defined on function types:

```
primrec facprimrec :: "nat ⇒ nat" where
  "facprimrec 0       = 1"
| "facprimrec (Suc n) = (Suc n) * (facprimrec n)"

definition facfold :: "nat ⇒ nat" where
  "facfold n = foldl (op*) 1 [1..< (Suc n)]"

lemma faceq: "facprimrec = facfold"
apply (rule ext)
apply (induct_tac x)
apply (simp add: facfold_def)
apply (simp add: facfold_def)
done
```


Further typing aspects

Remarks

- Isabelle/HOL infers and checks the types for all expressions/subexpressions
- Isabelle/HOL supports type classes:

  ```
  value "op +"
  "op +"  :: "'a::plus ⇒ 'a::plus ⇒ 'a::plus"
  ```

  Thus, "op+" only works for types providing a plus-operation. E.g., it fails for strings and functions:

  ```
  value "''a'' + ''b''"          -- fails
  value "facfold + facprimrec"   -- fails
  ```