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
2. Functional Programming and Modeling

2.1 Overview

Functional programming

Fact

A functional program consists of

- function declarations
- data type declarations
- an expression

Functional programs

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

Advantages

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

Functional programming and modeling

Function definitions describe execution plan

- functions may be partial
- exception-handling mechanisms

Function definitions play two roles:
  - representing programs (as above)
  - used to express properties

- functions have to be total

Common aspects

- recursive definition is central for functions and data
- strongly typed with:
  - type inference
  - parametric polymorphism
  - type classes
Subsection 2.2.1

Primitive datatypes and definitions

Example (Constant definition and use)

```isabelle
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

```isabelle
value "37+m" -- nat
value "37+k" -- int
value 37 -- 'a
```
Non-recursive function definitions

Example (Simple functions)

\begin{verbatim}
definition inc :: "int ⇒ int" where "inc j ≡ j+1"
value "inc 1234567890"
\end{verbatim}

\begin{verbatim}
definition nand :: "bool ⇒ bool ⇒ bool" where "nand A B ≡ ¬ (A ∧ B)"
value "nand (¬ bv) False"
\end{verbatim}

Higher-order functions

Example (Simple higher-order function)

\begin{verbatim}
definition appl2 :: "(int ⇒ int) ⇒ int ⇒ int" where "appl2 f j = f (f j)"
value "appl2 inc (-5)"
\end{verbatim}

Example (Lambda abstraction)

\begin{verbatim}
value "appl2 (λ x::int. x+1) k"
\end{verbatim}

\begin{verbatim}
definition plusN :: "int ⇒ (int ⇒ int)" where "plusN j = (λ x::int. x+j)"
value "plusN 3"
value "plusN 3 45"
\end{verbatim}

Pairs, tuples, type "unit"

Example (Pairs)

\begin{verbatim}
value "((k,inc 7))" -- "type is int × int"
value "fst (snd (True,(False,bv)))"
\end{verbatim}

Example (Tuples)

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

\begin{verbatim}
value "(True,True,True) = (True,(True,True))"
\end{verbatim}

Example (Type "unit")

\begin{verbatim}
value "()" -- "denotes the only and unique element of type unit"
\end{verbatim}
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 nat2nat = "nat ⇒ nat"
definition double :: nat2nat where "double n ≡ 2*n"
```
### Datatype definition

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

```isabelle
lemma "\( \forall x. x = \text{Mon} \lor x = \text{Tue} \lor x = \text{Wed} \lor x = \text{Thu} \lor x = \text{Fri} \lor x = \text{Sat} \lor x = \text{Sun} \)"
apply clarify
apply (case_tac x)
apply auto
done
```

### Recursive datatype definition

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

```isabelle
value "\text{Imp (Var ''x'') TTrue}"
```

**Remark**

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

### Primitive recursive function definitions

**Definition**

A recursive function definition of \( f \) is called **primitive recursive** if
- the \( i \)-th argument is of a datatype \( dt \) and
- all equations are of the form:

\[
 f x_1 \ldots x_{i-1} (C y_1 \ldots y_k) \ldots x_n = R
\]

where \( C \) is a constructor of \( dt \) and all recursive calls of \( f \) in \( R \) are of the form \( f t_1 \ldots t_{i-1} y_j \ldots \) for some \( j \) where \( t_i \) are arbitrary well-typed terms.

**Remark**

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

```isabelle
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)"
```

© Arnd Poetzsch-Heffter et al.  TU Kaiserslautern
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).

Parameterized datatypes

Motivation

To increase reuse, type systems in functional 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
2. Functional Programming and Modeling

2.2 Functional Programming in Isabelle/HOL

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,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 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
2. Functional Programming and Modeling 2.2 Functional Programming in Isabelle/HOL

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”

Example
Automatic proof fails for:

```plaintext
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 $R$ be the range/result type of the partial 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 "$R\ \text{option}$"
- use a relation

Illustration of alternatives 1 and 2

```plaintext
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 []"
```
Datatype `myunit = MyUnity`

Lemma `unity: "x = MyUnity"`
by (case_tac x)

Lemma "undefined = MyUnity"
by (rule unity)

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)
done

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)"

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
Implementing Simple Theorem Provers

Introduction

Theorem Prover
- theorem prover implements language and proof system
- used for proof checking and automated theorem proving

Goals of this subsection
- understanding the concepts and structure of a theorem prover
- illustrating how Isabelle/HOL can be used for system modeling

A Very Simple Prover for Propositional Logic
Abstract syntax of propositional logic

```plaintext
datatype plformula =
    Var string
  | TTrue
  | FFalse
  | Neg plformula
  | Imp plformula plformula (infixr "~~>" 100)
```

Representation of variable assignments

A variable assignment is a mapping from string to bool.

```plaintext
definition mkVAssign :: "string set ⇒ (string ⇒ bool)"
where
"mkVAssign strs s = (s ∈ strs)"

definition someva :: "(string ⇒ bool)" where
"someva = mkVAssign {''p'', ''q''}"
```

Semantics of propositional logic

```plaintext
primrec plfsem :: "(string ⇒ bool) ⇒ plformula ⇒ bool"
where
"plfsem va (Var s) = va s" |
"plfsem va TTrue = True" |
"plfsem va FFalse = False" |
"plfsem va (Neg p) = (¬ (plfsem va p))" |
"plfsem va (Imp p q) = ((plfsem va p) −→ (plfsem va q))"
```

Sequents and their semantics

```plaintext
datatype plsequent =
    Sq "plformula list" plformula (infix "⊢" 50)

definition plf1 :: plformula where
"plf1 = ((Var ''p'') ~~> TTrue)"

definition pls2 :: plsequent where
"pls2 = ( [Var ''a'', plf1] ⊢ - (Var ''p'') )"
```
Proof system

We build a simple proof system for backward proofs in natural deduction style. It consists of

- a proof state
  
  type_synonym proofstate = "plsequent list"

- a collection of rule-specific functions to manipulate the proof state according to the rules

Application of implication elimination and introduction

fun applImpE :: "proofstate ⇒ plformula ⇒ proofstate" where
  "applImpE ((Sq asms q) # psl) plf = (Sq asms (plf ~~> q)) # (Sq asms plf) # psl "
  | "applImpE psl plf = psl"

fun applImpI :: "proofstate ⇒ proofstate" where
  "applImpI ((Sq asms (p~~>q)) # psl) = (Sq (p#asms) q) # psl)"
  | "applImpI psl = psl"

Remark
The arguments of the application functions depend on the rule.

Application of negation elimination and introduction

fun applNegE :: "proofstate ⇒ plformula ⇒ proofstate" where
  "applNegE ((Sq asms FFalse) # psl) plf = (Sq asms (Neg plf)) # (Sq asms plf) # psl "
  | "applNegE psl plf = psl"

fun applNegI :: "proofstate ⇒ proofstate" where
  "applNegI ((Sq asms (Neg plf)) # psl) = (Sq (plf#asms) FFalse) # psl"
  | "applNegI psl = psl"

Application of assumption axiom

primrec iselem :: "'a ⇒ 'a list ⇒ bool" where
  "iselem x [] = False"
  | "iselem x (y#ys) = (if x=y then True else iselem x ys)"

fun assumpt :: "proofstate ⇒ proofstate" where
  "assumpt ((Sq asms q) # psl) = (if iselem q asms then psl else (Sq asms q) # psl )"
  | "assumpt psl = psl"
2. Functional Programming and Modeling

2.3 Implementing Simple Theorem Provers

Discussion

Properties of implementation

- Pattern matching mechanism of Isabelle/HOL is used to match the rules to the proof goal.
- For additional rules we have to add further functions to the prover.

Properties of logic

- Is the logic sound? I.e., is every derived formula a tautology?
- Is the logic complete? I.e., is every tautology derivable?
- Can these properties be formulated in Isabelle/HOL?

Extensible proof systems

Motivation
User should be able to prove rules and add them to the proof system.

Consequences

- Rules have to be derivable and become syntactical objects.
- We have to distinguish between formulas and formula schemas: ⎕ introduce schema variables
- Generic methods are needed that can apply different rules, in particular rules derived by the user.

Generic Application of Proof Rules

Example (Formula with schema variables)

(SVar ''A'') ~~> (SVar ''B'')

Can be used to express rules (see below)
### Substitutions and matching

**Definition (Substitution)**
A substitution $\sigma$ is a partial function from (schema) variables to formulas (or terms).

**Definition (Matching)**
Let $f_1$, $f_2$ be formulas (or terms).

$f_1$ matches $f_2$ if $f_2$ can be obtained by substituting the (schema) variables in $f_1$ by suitable formulas/terms:

$$\text{matches}(f_1, f_2) \iff \exists \sigma. (\text{applSubst}\ \sigma\ f_1) = f_2$$

### Unification

**Definition (Unifier, unifiability)**
A substitution $\sigma$ is a unifier of two formulas/terms if it makes them equal:

$$\text{unifier}(\sigma, f_1, f_2) \iff (\text{applSubst}\ \sigma\ f_1 = \text{applSubst}\ \sigma\ f_2)$$

Two formulas/terms are unifiable if they have a unifier:

$$\text{unifiable}(f_1, f_2) \iff \exists \sigma. \text{unifier}(\sigma, f_1, f_2)$$

**Remarks**
- Two formulas/terms might have several unifiers. Usually, one is interested in the most general unifier (MGU).
- For many logics, it is a non-trivial task to compute the MGU.

### Rules ($\rightarrow I$) and ($\rightarrow E$) as data

**Rule representation leaves $\Gamma$ implicit.**

```plaintext
datatype plrule = Rule "plsequent list" plsequent

definition impI :: plrule where
  "impI = (Rule
    [ ((SVar "'A'") \implies (SVar "'B'") ]
    ( [] \implies ((SVar "'A'") \iff (SVar "'B'"))) )"

definition impE :: plrule where
  "impE = (Rule
    [ [] \iff (SVar "'A'") \iff (SVar "'B'")), [\implies (SVar "'A'")] ]
    ( [] \iff (SVar "'B'")) )"

Remark
Rule representation leaves $\Gamma$ implicit.
```

```plaintext
definition negI :: plrule where
  "negI = (Rule
    [ [SVar "'A'"] \iff \text{FFalse} ]
    ( [] \iff (Neg (SVar "'A'")) )"

definition negE :: plrule where
  "negE = (Rule
    [ [] \iff (Neg (SVar "'A'")), [\implies (SVar "'A'")] ]
    ( [] \iff \text{FFalse} )"
```
2. Functional Programming and Modeling
2.3 Implementing Simple Theorem Provers

Rule application and matching

Rule application

• Goal: function that applies rules to sequents
• Central aspect: formula matching

Algorithm for matching formulas

• traverse two formulas $f_x, f_y$ where $f_x$ might contain schema variables
• recursively descent into $f_x, f_y$ and build up a substitution $\sigma$
• when a schema variable $s$ is encountered in $f_x$, and $f_{sub_y}$ is the corresponding subformula in $f_y$, check whether the current $\sigma$ already has a binding for $s$:  
  - if yes and $\sigma(s) = f_{sub_y}$, then continue with $\sigma$
  - if yes and $\sigma(s) \neq f_{sub_y}$, then matching fails
  - if no, then add a binding $(s \mapsto f_{sub_y})$ to $\sigma$

type_synonym substitution = "string ⇀ plformula"

fun matchf :: "plformula ⇒ plformula ⇒ substitution ⇒ (substitution × bool)"
where
"matchf (Var s1) (Var s2) σ = (if s1=s2 then (σ, True) else (empty, False))" |
"matchf (SVar s) fy σ = (case (σ s) of None ⇒ (σ(s↦fy), True) | Some f ⇒ (if f=fy then (σ, True) else (empty, False))))" |
"matchf TTrue TTrue σ = (σ, True)" |
"matchf FFalse FFalse σ = (σ, True)" |
"matchf (Neg px) (Neg py) σ = matchf px py σ" |
"matchf (Imp px1 px2) (Imp py1 py2) σ = (let (σ1, success1) = matchf px1 py1 σ in if success1 then matchf px2 py2 σ else (empty, False))" |
"matchf _ _ _ = (empty, False)"

Simplifying sequents:

For the application of rules, we need to match sequents.
In particular, we have to match $\Gamma$ to the list of assumptions.
To keep things simple here, we
• assume that sequent schemas in consequences only have the schema variable $\Gamma$ as assumption list
• handle the binding for $\Gamma$ as the first component of the result

fun match :: "plsequent ⇒ plsequent ⇒ (plformula list × substitution × bool)"
where
"match (gamma ⊢ fx) (fly ⊢ fy) = (fly, (matchf fx fy empty))"

fun applySubst::"substitution ⇒ plformula ⇒ plformula"
where
"applySubst sigma (Var s) = (Var s)" |
"applySubst sigma (SVar s) = (case (sigma s) of None ⇒ (SVar s) | Some f ⇒ f )" |
"applySubst sigma TTrue = TTrue" |
"applySubst sigma FFalse = FFalse" |
"applySubst sigma (Neg plf) = (Neg (applySubst sigma plf))" |
"applySubst sigma (Imp plf1 plf2) = (Imp (applySubst sigma plf1) (applySubst sigma plf2))"
Application of substitutions to sequents

fun applySubstSq ::
"plformula list ⇒ substitution ⇒ plsequent ⇒ plsequent"
where
"applySubstSq gammasubst sigma (Sq asms concl) =
( gammasubst@(map (applySubst sigma) asms)
 ⊢ (applySubst sigma concl)
)
"

Application of rules

Application “methods”
Application of a rule to a proof state might take different arguments. In our setting, applying

- the introduction rules (→I) and (¬I) need only the rule as argument (handled by appl).
- the elimination rules (→E) and (¬E) also need the formula to instantiate the schema variable A in the premisses that does not occur in the consequent; we pass the substitution for A as additional argument (handled by applE).
- the assumption rule needs a special treatment (unchanged from 1st version)

Definition of function appl

fun appl :: "plrule ⇒ proofstate ⇒ proofstate"
where
"appl pr [] = []" |
"appl (Rule prems conseq) (currentgoal#psl) =
( let (gsubst,sigma,success) = match conseq currentgoal
 in if success
 then (map (applySubstSq gsubst sigma) prems) @ psl
 else (currentgoal # psl)
 )"
Concluding remarks

• Note that we only need `applE`, because

  \[
  \text{appl plr pstate} = \text{applE plr pstate empty}
  \]

• Thus, most rules of the natural deduction calculus can be applied to a proof state using one application function.

• To prove a formula, start with a single goal and apply rules until no subgoal is left.

• The simple prover illustrates many concepts of Isabelle/HOL’s proof system like schema variable, rule application, different arguments depending on the used method/tactic.

• Having described the prover in Isabelle/HOL allows to specify and verify its correctness!

Chapter summary

• Introduction to functional programming and modeling in Isabelle/HOL

• Illustration of basic concepts underlying interactive proof systems

• Mentioning the idea that we could develop a theorem prover (or some other software system) in Isabelle/HOL and prove it correct

Questions

1. What is the relationship between the data type construct and the case expression? Illustrate the relationship by an example.

2. Why are there different forms of function definitions in Isabelle/HOL, but only one in ML?

3. Why is there a distinction between types with equality and types without equality in ML, but not in Isabelle/HOL? I.e., why can we compare functions by equality in Isabelle/HOL, but not in ML?

4. What is the distinction between matching and unification?

5. Why did we develop two functions for rule application (`appl`, `applE`) in our prover? Can they be unified?