Specification and Verification in Higher Order Logic

Prof. Dr. K. Madlener

13. April 2011

Chapter 1

Functional Programming: Reminder

Functional Programming

Fact 1.1. A functional program consists of

- data type declarations
- function declarations
- an expression

Functional Programs

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

Functional Programming

Advantages

- clearer semantics
- corresponds more directly to abstract mathematical objects
- more freedom in implementation

The SML Programming Language

Overview

- functional programming language
- interpreter and compiler available
- strongly typed, with:
  - type inference
  - abstract data types
  - parametric polymorphism
- exception-handling mechanisms

Motivation

- ML is similar to functional core of Isabelle/HOL specification language
- ML is the implementation language of the theorem prover

Evaluation and Bindings

Example 1.2. Evaluation

```
- 2 + 3;
val it = 5 : int

- rev [1,2,3,4,5];
val it = [5,4,3,2,1] : int list
```

Example 1.3. Simple Bindings

```
- val n = 8 * 2 + 5;
val n = 21 : int

- n * 2;
val it = 42 : int
```

Bindings

Example 1.4. Special Identifier **it**

```
- it;
val it = 42 : int
```

Example 1.5. Multiple Bindings

```
- val one = 1 and two = 2;
val one = 1 : int
val two = 2 : int
```

Local Bindings

Example 1.6. Simple Local Binding

```
- val n = 0;
val n = 0 : int

- let val n = 12 in n div 6 end;
val it = 2 : int

- n;
val it = 0 : int
```

Example 1.7. Multiple Local Bindings

```
- let val n = 5 val m = 6 in n + m end;
val it = 11 : int
```

Booleans

Example 1.8. Operations

```
- val b1 = true and b2 = false;
val b1 = true : bool
val b2 = false : bool

- 1 = (1 + 1);
val it = false : bool

- not (b1 orelse b2);
val it = false : bool

- (7 < 3) andalso (false orelse 2 > 0);
val it = false : bool
```

Integers

Example 1.9. Operations

```
- val n = 2 + (3 * 4);
val n = 14 : int

- val n = (10 div 2) - 7;
val n = ~2 : int
```

Applying Functions

General Rules

- type of functions from σ_1 to σ_2 is $\sigma_1 \rightarrow \sigma_2$
- application $f\ \ x$ applies function f to argument x
- call-by-value (obvious!)
- left associative: $m\ \ n\ \ o\ \ p = (((m\ \ n)o)p)$

Defining Functions

Example 1.10. One Argument

```
- fun f n = n + 2;
val f = fn : int -> int

- f 22;
val it = 24 : int
```

Example 1.11. Two or More Arguments

```
- fun plus n (m:int) = n + m;
val plus = fn : int -> int -> int

- plus 2 3;
val it = 5 : int
```

Currying

Example 1.12. Curried Addition

```
- fun plus n (m:int) = n + m;
val plus = fn : int -> int -> int

- plus 1 2;
val it = 3 : int
```

Example 1.13. Partial Application

```
- val inc = plus 1;
val inc = fn : int -> int

- inc 7;
val it = 8 : int
```

Higher-Order Functions

Example 1.14. Higher-Order Functions

```
- fun foo f n = (f(n+1)) div 2 ;
val foo = fn : (int -> int) -> int -> int

- foo inc 3;
val it = 2 : int
```

Recursive Functions

Example 1.15. Defining Recursive Functions

```
- fun f n = if (n=0) then 1 else n * f(n-1);
val f = fn : int -> int

- f 3;
val it = 6 : int

- fun member x [] = false |
      member x (h::t) = (x=h) orelse (member x t);
val member = fn : ''a -> ''a list -> bool

- member 3 [1,2,3,4];
val it = true : bool
```

Lambda Abstractions

Example 1.16. The Increment Function

```
- fn x=> x + 1;
val it = fn : int -> int

- (fn x=> x + 1) 2;
val it = 3 : int
```

Lambda Abstractions

Example 1.17. Curried Multiplication

```
- fn x=> fn y=> x * (y:int);
val it = fn : int -> int -> int

- val double = (fn x=> fn y=> x * (y:int)) 2;
val double = fn : int -> int

- double 22;
val it = 44 : int
```

Clausal Definitions

Example 1.18. Fibonacci

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2);
```

Exceptions

Example 1.19. Failure

```
- hd [];
uncaught exception Hd

- 1 div 0;
uncaught exception Div
```

User-Defined Exceptions

Example 1.20. Explicitly Generating Failure

```
- exception negative_argument_to_fact;
exception negative_argument_to_fact

- fun fact n =
        if (n<0) then raise negative_argument_to_fact
        else if (n=0) then 1 else n * fact(n-1);
val fact = fn : int -> int

- fact (~1);
uncaught exception negative_argument_to_fact
```

Example 1.21. Exception Handling

```
- (fact(~1)) handle negative_argument_to_fact => 0;
val it = 0 : int
```

Unit

Example 1.22. Unit

```
- ();
val it = () : unit

- close_theory;
val it = fn : unit -> unit
```

Character Strings

Example 1.23. String Operations

```
- "abc";
val it = "abc" : string

- chr;
val it = fn : int -> string

- chr 97;
val it = "a" : string
```

List Constructors

Example 1.24. Empty Lists

```
- null l;
val it = false : bool

- null [];
val it = true : bool
```

Example 1.25. Construction and Concatenation

```
- 9 :: l;
val it = [9,2,3,5] : int list

- [true,false] @ [false,true];
val it = [true,false,false,true] : bool list
```

List Operations

Example 1.26. Head and Tail

```
- val l = [2,3,2+3];
val l = [2,3,5] : int list

- hd l;
val it = 2 : int

- tl l;
val it = [3,5] : int list
```

Pattern Matching

Example 1.27. Pattern Matching and Lists

```
- fun bigand [] = true
    | bigand (h::t) = h andalso bigand t;
val bigand = fn : bool list -> bool
```

Pairs

Example 1.28. Pair Functions

```
- val p = (2,3);
val p = (2,3) : int * int

- fst p;
val it = 2 : int

- snd p;
val it = 3 : int
```

Records

Example 1.29. Date Record

```
val date = {day=4,month="february",year=1967}
  : {day:int, month:string, year:int}

- val {day=d,month=m,year=y} = date;
val d = 4 : int
val m = "february" : string
val y = 1967 : int

- #month date;
val it = "february" : string
```

Polymorphism

Example 1.30. Head Function

```
- hd [2,3];
val it = 2 : int

- hd [true,false];
val it = true : bool
```

Problem 1.31. What is the type of `hd`?

`int list ->int` or `bool list ->bool`

Polymorphism

Example 1.32. Type of Head Function

```
- hd;
val it = fn : 'a list -> 'a
```

Example 1.33. Polymorphic Head Function

- head function has both types
- `'a` is a type variable.
- `hd` can have any type of the form σ `list ->` σ (where σ is an SML type)

Type Inference

Example 1.34. Mapping Function

```
- fun map f l =
        if (null l)
           then []
           else f(hd l)::(map f (tl l));
val map = fn : ('a -> 'b) -> 'a list -> 'b list

- map (fn x=>0);
val it = fn : 'a list -> int list
```

Fact 1.35. ML Type Inference SML infers the most general type.

Standard List Operations

Example 1.36. Mapping

```
- fun map f [] = []
    | map f (h::t) = f h :: map f t;
val ('a, 'b) map = fn : ('a -> 'b) -> 'a list -> 'b
    list
```

Example 1.37. Filtering

```
- fun filter P [] = []
    | filter P (h::t) = if P h then h::filter P t
                                  else filter P t;
val 'a filter = fn : ('a -> bool) -> 'a list -> 'a
    list
```

Type Inference

Example 1.38. Function Composition

```
- fun comp f g x = f(g x);
val comp = fn:('a -> 'b) -> ('c -> 'a) -> 'c -> 'b

- comp null (map (fn y=> y+1));
val it = fn : int list -> bool
```

Some System Functions

Example 1.39. Load a file called `file.sml`

```
- use;
val it = fn : string -> unit

- use "file.sml";
[opening file.sml]
...
```

Key Commands

- terminate the session: <Ctrl> D
- interrupt: <Ctrl> C

Tuples

Example 1.40. Tuples

```
- val pair = (2,3);
> val pair = (2, 3) : int * int

- val triple = (2,2.0,"2");
> val triple = (2, 2.0, "2") : int * real * string
- val pairs_of_pairs = ((2,3),(2.0,3.0));
> val pairs_of_pairs = ((2, 3), (2.0, 3.0)) : (int *
    int) * (real * real)
```

Example 1.41. Unit Type

```
- val null_tuple = ();
> val null_tuple = () : unit
```

Accessing Components

Example 1.42. Navigating to the Position

```
- val xy1 = #1 pairs_of_pairs;
> val xy1 = (2, 3) : int * int

- val y1 = #2 (#1 pairs_of_pairs);
> val y1 = 3 : int
```

Example 1.43. Using Pattern Matching

```
- val ((x1,y1),(x2,y2)) = pairs_of_pairs;
> val x1 = 2 : int
  val y1 = 3 : int
  val x2 = 2.0 : real
  val y2 = 3.0 : real
```

Pattern Matching

Example 1.44. Granularity

```
- val ((x1,y1),xy2) = pairs_of_pairs;
> val x1 = 2 : int
  val y1 = 3 : int
  val xy2 = (2.0, 3.0) : real * real
```

Example 1.45. Wildcard Pattern

```
- val ((_,y1),(_,_)) = pairs_of_pairs;
> val y1 = 3 : int

- val ((_,y1),_) = pairs_of_pairs;
> val y1 = 3 : int
```

Pattern Matching

Example 1.46. Value Patterns

```
- val 0 = 1-1;

- val (0,x) = (1-1,34);
> val x = 34 : int

- val (0,x) = (2-1,34);
! Uncaught exception:
! Bind
```

Binding Values

General Rules

- The variable binding `val var =val` is irreducible.
- The wildcard binding `val _ =val` is discarded.
- The tuple binding `val(pat1, ... , patN)=(val1, ... , valN )` is reduced to

```
val pat1 = valN

...

val patN = valN
```

Clausal Function Expressions

Example 1.47. Clausal Function Expressions

```
- fun not true = false
    | not false = true;
> val not = fn : bool -> bool
```

Redundant Cases

Example 1.48. Redundant Cases

```
- fun not True = false
    | not False = true;
! Warning: some cases are unused in this match.
> val 'a not = fn : 'a -> bool

- not false;
> val it = false : bool
- not 3;
> val it = false : bool
```

Fact 1.49. Redundant Cases are always a mistake!

Inexhaustive Matches

Example 1.50. Inexhaustive Matches

```
fun first_ten 0 = true | first_ten 1 = true |
     first_ten 2 = true
   | first_ten 3 = true | first_ten 4 = true |
      first_ten 5 = true
   | first_ten 6 = true | first_ten 7 = true |
      first_ten 8 = true
   | first_ten 9 = true;
! Warning: pattern matching is not exhaustive
> val first_ten = fn : int -> bool
- first_ten 5;
> val it = true : bool
first_ten ~1;
! Uncaught exception: Match
```

Fact 1.51. Inexhaustive Matches may be a problem.

Catch-All Clauses

Example 1.52. Catch-All Clauses

```
fun first_ten 0 = true | first_ten 1 = true |
    first_ten 2 = true
  | first_ten 3 = true | first_ten 4 = true |
      first_ten 5 = true
  | first_ten 6 = true | first_ten 7 = true |
      first_ten 8 = true
  | first_ten 9 = true | first_ten _ = false;
> val first_ten = fn : int -> bool
```

Overlapping Cases

Example 1.53. Overlapping Cases

```
- fun foo1 1 _ = 1
    | foo1 _ 1 = 2
    | foo1 _ _ = 0;
> val foo1 = fn : int -> int -> int
- fun foo2 _ 1 = 1
    | foo2 1 _ = 2
    | foo2 _ _ = 0;
> val foo2 = fn : int -> int -> int

- foo1 1 1;
> val it = 1 : int
- foo2 1 1;
> val it = 1 : int
```

Recursively Defined Functions

Example 1.54. Recursively Defined Function

```
- fun factorial 0 = 1
    | factorial n = n * factorial (n-1);
> val factorial = fn : int -> int

- val rec factorial = fn
```

Example 1.55. Recursively Defined Lambda Abstraction

```
- val rec factorial =
fn 0 => 1
         | n => n * factorial (n-1);
```

Mutual Recursion

Example 1.56. Mutual Recursion

```
- fun even 0 = true
    | even n = odd (n-1)
  and odd 0 = false
    | odd n = even (n-1);
> val even = fn : int -> bool
  val odd = fn : int -> bool

- (even 5,odd 5);
> val it = (false, true) : bool * bool
```

Simple data Types: Type Abbreviations

`type` Keyword

- type abbreviations
- record definitions

Example 1.57. Type Abbreviation

```
- type boolPair = bool * bool;
> type boolPair = bool * bool

- (true,true):boolPair;
> val it = (true, true) : bool * bool
```

Defining a Record Type

Example 1.58. Record

```
- type hyperlink =
    { protocol : string, address : string, display :
        string };
> type hyperlink = {address : string, display : string
    , protocol : string}

- val hol_webpage = {
    protocol="http",
    address="rsg.informatik.uni-kl.de/teaching/hol",
    display="HOL-Course" };
> val hol_webpage = {
    address = "rsg.informatik.uni-kl.de/teaching/hol",
    display = "HOL-Course",
    protocol = "http"}
      :{address : string, display : string, protocol :
           string}
```

Accessing Record Components

Example 1.59. Type Abbreviation

```
- val {protocol=p, display=d, address=a } =
    hol_webpage;
> val p = "http" : string
  val d = "HOL-Course" : string
  val a = "rsg.informatik.uni-kl.de/teaching/hol" :
     string
```

```
- val {protocol=_, display=_, address=a } =
    hol_webpage;
> val a = "rsg.informatik.uni-kl.de/teaching/hol" :
    string
```

Accessing Record Components (cont.)

```
- val {address=a, ...} = hol_webpage;
> val a = "rsg.informatik.uni-kl.de/teaching/hol" :
  string
```

```
- val {address, ...} = hol_webpage;
> val address = "rsg.informatik.uni-kl.de/teaching/hol
  " : string
```

Defining *Really* New Data Types

`datatype` Keyword

programmer-defined (recursive) data types, introduces

- one or more new type constructors
- one or more new value constructors

Non-Recursive Data Type

Example 1.60. Non-Recursive Datatype

```
- datatype threeval = TT | UU | FF;
> New type names: =threeval
datatype threeval =
(threeval,{con FF : threeval, con TT : threeval, con
    UU : threeval})
con FF = FF : threeval
con TT = TT : threeval
con UU = UU : threeval

- fun not3 TT = FF
    | not3 UU = UU
    | not3 FF = TT;
> val not3 = fn : threeval -> threeval

- not3 TT;
> val it = FF : threeval
```

Parameterised Non-Recursive Data Types

Example 1.61. Option Type

```
- datatype 'a option = NONE | SOME of 'a;
> New type names: =option
  datatype 'a option =
  ('a option,{con 'a NONE : 'a option, con 'a SOME : '
      a -> 'a option})
  con 'a NONE = NONE : 'a option
  con 'a SOME = fn : 'a -> 'a option
```

- constant `NONE`
- values of the form `SOME` *v* (where *v* has the type `'a`)

Option Types

Example 1.62. Option Type

```
- fun reciprocal 0.0 = NONE
    | reciprocal x = SOME (1.0/x)
> val reciprocal = fn : real -> real option

- fun inv_reciprocal NONE = 0.0
    | inv_reciprocal (SOME x) = 1.0/x;
> val inv_reciprocal = fn : real option -> real
- fun identity x = inv_reciprocal (reciprocal x);
> val identity = fn : real -> real

- identity 42.0;
> val it = 42.0 : real
- identity 0.0;
> val it = 0.0 : real
```

Recursive Data Types

Example 1.63. Binary Tree

```
- datatype 'a tree =
Empty |
Node of 'a tree * 'a * 'a tree;
> New type names: =tree
  datatype 'a tree =
  ('a tree,
   {con 'a Empty : 'a tree,
    con 'a Node : 'a tree * 'a * 'a tree -> 'a tree})
  con 'a Empty = Empty : 'a tree
  con 'a Node = fn : 'a tree * 'a * 'a tree -> 'a tree
```

- `Empty` is an empty binary tree
- (`Node` (t_1, v, t_2)) is a tree if t_1 and t_2 are trees and v has the type `'a`
- nothing else is a binary tree

Functions and Recursive Data Types

Example 1.64. Binary Tree

```
- fun treeHeight Empty = 0
    | treeHeight (Node (leftSubtree, _, rightSubtree))
        =
       1 + max(treeHeight leftSubtree, treeHeight
          rightSubtree);
> val 'a treeHeight = fn : 'a tree -> int
```

Mutually Recursive Datatypes

Example 1.65. Binary Tree

```
- datatype 'a tree =
    Empty |
    Node of 'a * 'a forest
  and 'a forest =
   None |
   Tree of 'a tree * 'a forest;
> New type names: =forest, =tree
  ...
```

Abstract Syntax

Example 1.66. Defining Expressions

```
- datatype expr =
      Numeral of int |
      Plus of expr * expr |
      Times of expr * expr;
> New type names: =expr
  datatype expr =
  (expr,
   {con Numeral : int -> expr,
    con Plus : expr * expr -> expr,
    con Times : expr * expr -> expr})
  con Numeral = fn : int -> expr
  con Plus = fn : expr * expr -> expr
  con Times = fn : expr * expr -> expr
```

Abstract Syntax

Example 1.67. Evaluating Expressions

```
- fun eval (Numeral n) = Numeral n
    | eval (Plus(e1,e2)) =
                let val Numeral n1 = eval e1
                    val Numeral n2 = eval e2 in
                  Numeral(n1+n2) end
    | eval (Times (e1,e2)) =
            let val Numeral n1 = eval e1
                    val Numeral n2 = eval e2 in
                  Numeral(n1*n2) end;
> val eval = fn : expr -> expr

- eval( Plus( Numeral 2, Times( Numeral 5, Numeral 8 )
    ) );
> val it = Numeral 42 : expr
```

Modules: Structuring ML Programs

Modules

- structuring programs into separate units
- program units in ML: *structures*
- contain a collection of types, exceptions and values (incl. functions)
- parameterised units possible
- composition of structures mediated by *signatures*

Structures

Purpose

- structures = implementation

Example 1.68. Structure

```
structure Queue =
struct
  type 'a queue = 'a list * 'a list
  val empty = (nil,nil)
  fun insert( x, (bs,fs)) = (x::bs, fs)
  exception Empty
  fun remove (nil,nil) = raise Empty
  | remove (bs, f::fs) = (f, (bs,fs))
  | remove (bs, nil)= remove (nil, rev bs)
end
```

Accessing Structure Components

Identifier Scope

- components of a structure: local scope
- must be accessed by qualified names

Example 1.69. Accessing Structure Components

```
- Queue.empty;
> val ('a, 'b) it = ([], []) : 'a list * 'b list

- open Queue;
> ...
- empty;
> val ('a, 'b) it = ([], []) : 'a list * 'b list
```

Accessing Structure Components

Usage of `open`

- open a structure to incorporate its bindings directly
- cannot open two structures with components that share a common names
- prefer to use open in `let` and `local` blocks

Signatures

Purpose

- signatures = interface

Example 1.70. Signature

```
signature QUEUE =
sig
  type 'a queue
  exception Empty
  val empty : 'a queue
  val insert: 'a * 'a queue -> 'a queue
  val remove: 'a queue -> 'q * 'a queue
end
```

Signature Ascription

Transparent Ascription

- descriptive ascription
- extract principal signature
  - always existing for well-formed structures
  - most specific description
  - everything needed for type checking
- source code needed

Opaque Ascription

- restrictive ascription
- enforce data abstraction

Opaque Ascription

Example 1.71. Opaque Ascription

```
structure Queue :> QUEUE
struct
  type 'a queue = 'a list * 'a list
  val empty = (nil,nil)
  fun insert( x, (bs,fs)) = (x::bs, fs)
  exception Empty
  fun remove (nil,nil) = raise Empty
  | remove (bs, f::fs) = (f, (bs,fs))
  | remove (bs, nil)= remove (nil, rev bs)
  end
```

Signature Matching

Conditions

- structure may provide more components
- structure may provide more general types than required
- structure may provide a concrete datatype instead of a type
- declarations in any order

Modular Compilation in Moscow ML

Compiler **mosmlc**

- save structure Foo to file `Foo.sml`
- compile module: `mosmlc Foo.sml`
- compiled interface in `Foo.ui` and compiled bytecode `Foo.uo`
- load module `load "Foo.ui"`

```
- load "Queue";
> val it = () : unit
- open Queue;
> type 'a queue = 'a list * 'a list
  val ('a, 'b) insert = fn : 'a * ('a list * 'b) -> 'a
      list * 'b
  exn Empty = Empty : exn
  val ('a, 'b) empty = ([], []) : 'a list * 'b list
  val 'a remove = fn : 'a list * 'a list -> 'a * ('a
      list * 'a list)
```

Implementing a Simple Theorem Prover: Overview

Theorem Prover

- theorem prover implements a proof system
- used for proof checking and automated theorem proving

Goals

- build your own theorem prover for propositional logic
- understanding the fundamental structure of a theorem prover

Data Types

Data Types of a Theorem Prover

- formulas, terms and types
- axioms and theorems
- deduction rules
- proofs

Formulas, Terms and Types

Propositional Logic

- each term is a formula
- each term has the type $\mathbb{B}$

Data Type Definition

```
datatype Term =
    Variable of string |
    Constant of bool |
    Negation of Term |
    Conjunction of Term * Term |
    Disjunction of Term * Term |
    Implication of Term * Term;
```

Syntactical Operations on Terms

Determining the Topmost Operator

```
fun isVar (Variable x) = true
  | isVar _ = false;
fun isConst (Constant b) = true
  | isConst _ = false;
fun isNeg (Negation t1) = true
  | isNeg _ = false;
fun isCon (Conjunction (t1,t2)) = true
  | isCon _ = false;
fun isDis (Disjunction (t1,t2)) = true
  | isDis _ = false;
fun isImp (Implication (t1,t2)) = true
  | isImp _ = false;
```

Syntactical Operations on Terms

Composition

- combine several subterms with an operator to a new one

Composition of Terms

```
fun mkVar s1 = Variable s1;
fun mkConst b1 = Constant b1;
fun mkNeg t1  = Negation t1;
fun mkCon (t1,t2) = Conjunction(t1,t2);
fun mkDis (t1,t2) = Disjunction(t1,t2);
fun mkImp (t1,t2) = Implication(t1,t2);
```

Syntactical Operations on Terms

Decomposition

- decompose a term

Decomposition of Terms

```
exception SyntaxError;

fun destNeg (Negation t1) = t1
  | destNeg _ = raise SyntaxError ;
fun destCon (Conjunction (t1,t2)) = (t1,t2)
  | destCon _ = raise SyntaxError ;
fun destDis (Disjunction (t1,t2)) = (t1,t2)
  | destDis _ = raise SyntaxError ;
fun destImp (Implication (t1,t2)) = (t1,t2)
  | destImp _ = raise SyntaxError ;
```

Term Examples

Example 1.72. Terms

- $t_1 = a \wedge b \vee \neg c$;
- $t_2 = \text{true} \wedge (x \wedge y) \vee \neg z$;
- $t_3 = \neg((a \vee b) \wedge \neg c)$

```
val t1 = Disjunction(
            Conjunction(Variable "a",Variable "b"),
            Negation(Variable "c"));
val t2 = Disjunction(
           Conjunction(Constant true,
                       Conjunction (Variable "x",
                          Variable "y")),
                        Negation(Variable "z"));
val t3 = Negation(Conjunction(
             Disjunction(Variable "a",Variable "b"),
             Negation(Variable "c")));
```

Theorems

Data Type Definition

```
datatype Theorem =
    Theorem of Term list * Term;
```

Syntactical Operations

```
fun assumptions (Theorem (assums,concl)) = assums;
fun conclusion (Theorem (assums,concl)) = concl;
fun mkTheorem(assums,concl) = Theorem(assums,concl);
fun destTheorem (Theorem (assums,concl)) = (assums,
   concl);
```

Rules

Data Type Definition

```
datatype Rule =
    Rule of Theorem list * Theorem;
```

Application of Rules

Application of Rules

- form a new theorem from several other theorems

Application (Version 1)

```
exception DeductionError;

fun applyRule rule thms =
    let
        val Rule (prem,concl) = rule
    in
        if prem=thms then concl else raise
            DeductionError end;
```

Application of Rules

Application of Rules

- premises and given theorems do not need to be identical
- premises only need to be in the given theorems

Application (Version 2)

```
fun mem x [] = false
  | mem x (h::t) = (x=h) orelse (mem x t);
fun sublist [] l2 = true
  | sublist (h1::t1) l2 = (mem  h l2) andalso (sublist
      t1 l2);
fun applyRule rule thms =
     let val Rule (prem,concl) = rule
     in
        if sublist prem thms then concl else raise
            DeductionError end;
```

Application of Rules

Example 1.73. Rule Application

```
val axiom1 = Theorem( [], (Variable "a"));
val axiom2 = Theorem( [], Implication((Variable "a"),(
    Variable "b")));
val axiom3 = Theorem( [], Implication((Variable "b"),(
    Variable "c")));

val modusPonens =
    Rule(
        [Theorem( [], Implication((Variable "a"),(
            Variable "b")) ),
         Theorem( [], (Variable "a") )]
    ,
        Theorem( [], (Variable "b") )
    );
```

Application of Rules

Example 1.74. Rule Application

```
val thm1 = applyRule modusPonens [axiom1,axiom2];
val thm2 = applyRule modusPonens [thm1,axiom3];
```

Problem

- axioms and rules should work for arbitrary variables
- axiom scheme, rule scheme
- definition of substitution and unification needed

Support Functions

Support Functions

```
fun insert x l = if mem x l then l else x::l;

fun assoc [] a = NONE
  | assoc ((x,y)::t) a = if (x=a) then SOME y else
      assoc t a;

fun occurs v (w as Variable _) = (v=w)
  | occurs v (Constant b) = false
  | occurs v (Negation t) = occurs v t
  | occurs v (Conjunction (t1,t2)) = occurs v t1
      orelse occurs v t2
  | occurs v (Disjunction (t1,t2)) = occurs v t1
      orelse occurs v t2
  | occurs v (Implication (t1,t2)) = occurs v t1
      orelse occurs v t2;
```

Substitution

Substitution

```
fun subst theta (v as Variable _) =
              (case assoc theta v of NONE => v | SOME w
                  => w)
| subst theta (Constant b) = Constant b
| subst theta (Negation t) = Negation(subst theta t)
| subst theta (Conjunction (t1,t2)) =
               Conjunction(subst theta t1, subst theta
                   t2)
| subst theta (Disjunction (t1,t2)) =
               Disjunction(subst theta t1, subst theta
                   t2)
| subst theta (Implication (t1,t2)) =
               Implication(subst theta t1, subst theta
                   t2);
```

Substitution

Example 1.75. Substitution

```
val theta1 = [(Variable "a",Variable "b"),(Variable "b
    ",Constant true)];
```

Unification

Definition 1.76. *Matching: A term matches another if the latter can be obtained by instantiating the former.*

$$\text{matches}(M, N) \Leftrightarrow \exists\theta.\text{subst}(\theta, M) = N$$

Definition 1.77. *Unifier, Unifiability: A substitution is a unifier of two terms, if it makes them equal.*

$$\text{unifier}(\theta, M, N) \Leftrightarrow \text{subst}(\theta, M) = \text{subst}(\theta, N)$$

Two terms are unifiable if they have a unifier.

$$\text{unifiable}(M, N) \Leftrightarrow \exists\theta.\text{unifier}(\theta, M, N)$$

Unification Algorithm

General Idea

- traverse two terms in exactly the same way
- eliminating as much common structure as possible
- things actually happen when a variable is encountered (in either term)
- when a variable is encoutered, make a binding with the corresponding subterm in the other term, and substitute through
- important: making a binding (x, M) where x occurs in M must be disallowed since the resulting substitution will not be a unifier occur check.

Unification Algorithm

Unification

```
exception UnificationException;

fun unifyl [] [] theta = theta
  | unifyl ((v as Variable _)::L) (M::R) theta =
      if v=M then unifyl L R theta
      else if occurs v M then raise
             UnificationException
      else unifyl (map (subst [(v,M)]) L)
                  (map (subst [(v,M)]) R)
                  (combineSubst [(v,M)] theta)
     | unifyl L1 (L2 as (Variable _::_)) theta =
         unifyl L2 L1 theta
     ...
```

Unification Algorithm

```
  ...
| unifyl (Negation tl::L) (Negation tr::R) theta =
          unifyl (tl::L) (tr::R) theta
| unifyl (Conjunction (tl1,tl2)::L) (Conjunction (tr1
    ,tr2)::R) theta =
          unifyl (tl1::tl2::L) (tr1::tr2::R)
             theta
| unifyl (Disjunction (tl1,tl2)::L) (Disjunction (tr1
    ,tr2)::R) theta =
          unifyl (tl1::tl2::L) (tr1::tr2::R)
             theta
| unifyl (Implication (tl1,tl2)::L) (Implication (tr1
    ,tr2)::R) theta =
          unifyl (tl1::tl2::L) (tr1::tr2::R)
             theta
| unifyl _ _ _ = raise UnificationException;

fun unify M N = unifyl [M] [N] [];
```

Combining Substitutions

Combining Substitutions

```
fun combineSubst theta sigma =
  let val (dsigma,rsigma) = ListPair.unzip sigma
      val sigma1 = ListPair.zip(dsigma,(map (subst
         theta) rsigma))
      val sigma2 = List.filter (op <>) sigma1
      val theta1 = List.filter (fn (a,_) => not (mem a
          dsigma)) theta
  in
     sigma2 @ theta1
  end;
```

Summary

- programming in Standard ML
  - evaluation and bindings
  - defining functions
  - standard data types
  - type inference
  - case analysis and pattern matching
  - data type definitions
  - modules
- primitive theorem prover kernel
  - terms
  - theorems
  - rules
  - substitution
  - unification