Introduction to (Logic and Functional) Programming

http://www.cse.iitd.ac.in/~sak/courses/ilfp/2014-15.index.html

S. Arun-Kumar
Department of Computer Science and Engineering
I. I. T. Delhi, Hauz Khas, New Delhi 110 016.

March 17, 2015

Contents

Lecture 1: Introduction

Tuesday 26 July 2011

Programming and Algorithms

- A *computation* is a sequence of transformations carried out *mechanically* by means of a number of predefined rules of transformation on finite *discrete* data.
- Computations are specified with the help of *programs* written in a programming language.
- *Algorithms* studies specific classes of problems for which programs may be written on some some universal machine.
- *Programming* is concerned with the logical aspects of program organization.
  1. Draws on the study of algorithms to choose efficient data structures and high-performance algorithms
  2. Main purpose is to provide concepts and methods for writing programs *correctly*, *legibly* in a way that is easy to modify and reuse.

Program Specification

- How do we specify what to expect from a program?
- How do we associate what we expect from a program with a precise description of what the program computes?
- How can we ensure that the program is correct with respect to a given specification?

These questions can be rigorously answered only by means of a formal mathematical specification and by establishing a formal relationship between the specification and the program.

- *Unfortunately, the state of art of these processes is such that they can be done only for small programs.*
- *Without sufficient automation of formal reasoning methods these cannot be done for huge industry scale programs.*

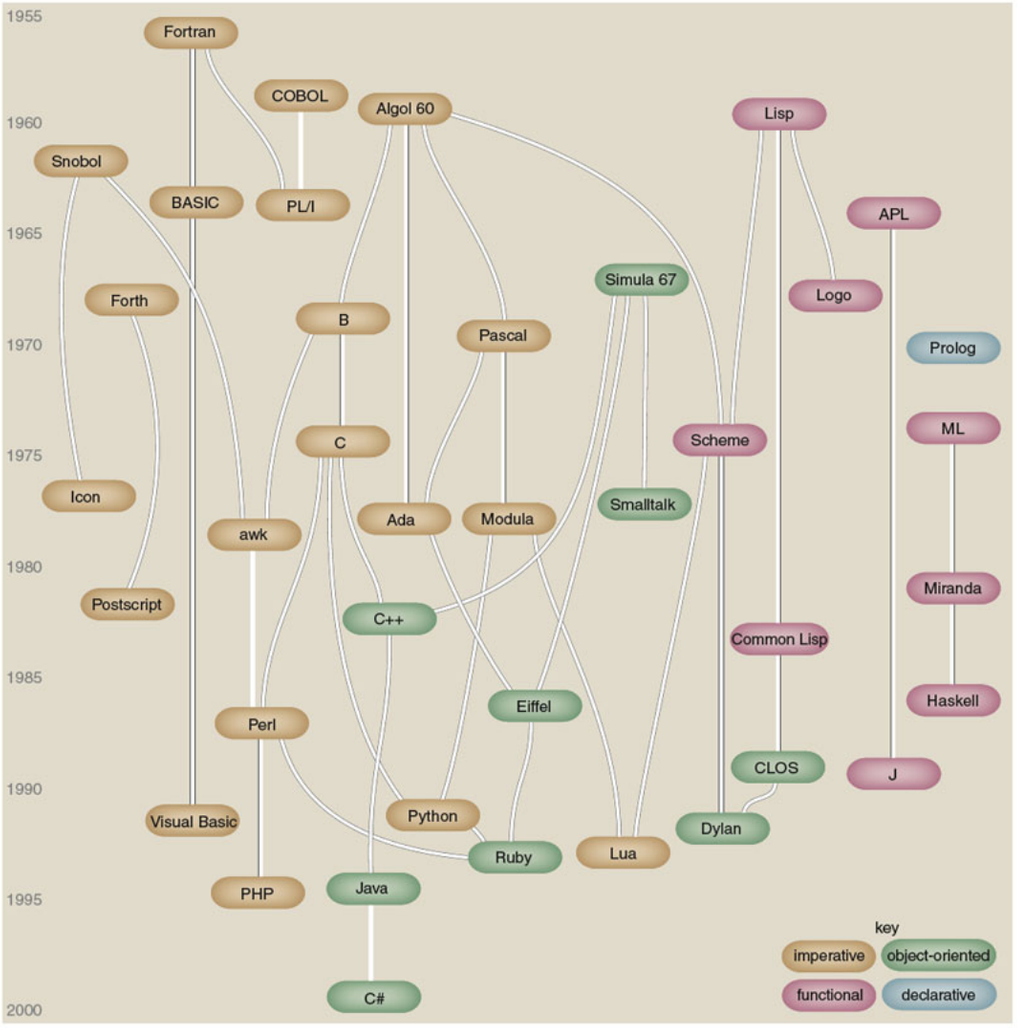
Programming languages History

- A continuous effort to abstract high-level concepts in order to escape low-level details and idiosyncracies of particular machines.
  - machine language (the use of mnemonics)
  - assembly language (assemblers)
  - FORTRAN (compilers)
  - LISP (interpreters)
  - Pascal (compiler on a virtual machine)
  - PROLOG (Abstract machines)
  - Smalltalk (OO with mutable objects)
  - ML (Memory abstraction)

Imperative Programming

- Most conventional programming languages (e.g. C, C++, Java)
- Evolved from the Von-Neumann architecture (machine, assembly, FORTRAN)
- Principally rely on *state changes* (visible updation of memory) through *side-effects*
- Far removed from mathematics (e.g. `x = x+1`).
- Not *referentially transparent*: the same function placed in different contexts behaves differently (side-effects on global variables, aliasing problems etc.).

Chronology of Programming Languages



Lecture 2: Functional Programming

Wednesday 27 July 2011

Functional Programming

- A program consists entirely of functions (some may depend on others previously defined). The "main" program is also a function.
- The "main" function is given input values and the result of evaluating it is the output.
- Most functional progrmaming languages are interactive.
- The notion of function in a pure functional language is the mathematical notion of function.

Imperative vs. Functional

- Imperative programs rely on "side-effects" and state updation. There are no side-effects in "pure" functional programs.
- Side-effects in imperative programs are mainly due to assignment commands (either direct or indirect). There is no assignment command in pure functional languages.
- Most imperative programmers rely on iterative loops. There is no iteration mechanism in a pure functional program. *Instead recursion is used*.
- Variables in imperative programs tend to change values over time. There is no change in the values of variables in a pure functional program. *Variables are constants.*

Referential Transparency

Definition 2.1 *An expression is* **referentially transparent** *if it can be replaced by its value in a program without changing the behaviour of the program in any way.*

In a pure functional language all functions are *referentially transparent*. Hence

- programs are mathematically more tractable than imperative ones.
- compiler optimizations such as common sub-expression elimination, code movement etc. can be incorporated without any static analysis.
- Any expression anywhere may be replaced by another expression which has the same value.

In most imperative languages (because of assignment, and side-effects to non-local variables) there is no (guarantee of) referential transparency.

Higher-order functions & Modularity

Higher Order. Higher order functions characterise most functional programming. It leads to compact and concise code.

Modularity. Modularity can be built into a pure functional language

Objected-orientedness. Object-oriented features require state updation and can be obtained only by destroying referential transparency. So a pure functional programming language cannot be object-oriented, though it can be modular.

Imperative features

Input/Output. All input-output and file-handling (esp. in the Von Neumann framework) is inherently imperative.

Object-Orientation. Object oriented features require updation of state and are hence better served by imperative features.

So most functional languages need to have certain imperative features.

Lecture 3: Standard ML Overview

Friday 29 July 2011

SML: Overview

(Impure) Functional

Strongly and statically typed

Type inferencing

Parameterised Types

Parametric polymorphism

Modularity

SML: Functional

Based on the model of *evaluating expressions* as opposed to the model of *executing sequences of commands* found in imperative languages.

Strongly and statically typed

Type inferencing

Parameterised Types

Parameterised Types

Parametric polymorphism

Modularity

SML: Strong Static Typing

Definition 3.1 *A language is* **statically typed** *if every expression in the language is assigned a type at compile-time.*

Definition 3.2 *A language is* **strongly typed** *if the language requires the provision of a type-checker which ensures that no type errors will occur at run-time.*

Each expression in the ML language is *assigned a type* at compile-time describing the possible set of values it can evaluate to, and no runtime type errors occur.

(Impure) Functional

Type inferencing

Parameterised Types

Parameterised Types

Parametric polymorphism

Modularity

SML: Parameterised Types

ML allows the use of *parameterised types* which allows a single implementation to be applied to all structures which are instances of the parametric type. For this purpose ML also has the notion of a *type variable*.

- Facilitates parametric polymorphism
- Reduces duplication of similar code and allows code reuse.

(Impure) Functional

Type inferencing

Parameterised Types

Parametric polymorphism

Modularity

SML: Type inferencing

Except in a few instances, ML is capable of *deducing* the types of identifiers from the context. There is *no need to declare every identifier* before it is used.

Type-inferencing also works on parametric and polymorphic types in such a way that ML

- assigns the most general parametric polymorphic type to the expression at *compile-time* and
- ensures that each *run-time* value produced by the expression is an appropriate instance of the polymorphic type assigned to it.

(Impure) Functional

Strongly and statically typed

Parameterised Types

Parametric polymorphism

Modularity

SML: Parametric Polymorphism

A function gets a polymorphic type when it can operate uniformly over any value of any given type.

Example 3.3 *One can define types of the form* `stack('a)` *where* `'a` *is a type variable, for stacks of all types including stacks of complex user-defined data structures and types.*
The operations defined for `stack('a)` *work equally well on all instances of the type.*

(Impure) Functional

Strongly and statically typed

Type inferencing

Parameterised Types

Modularity

SML: Modularity

A state-of-the-art module system, based on the notions of *structures* (containing the actual code), *signatures* (the type of structures) and *functors* (creation of parameterised structures from one or more other parametrised structures without the need for writing new code).

(Impure) Functional

Strongly and statically typed

Type inferencing

Parameterised Types

Parametric polymorphism

SML: Overview Summary 1

(Impure) Functional. Based on the model of *evaluating expressions* (as opposed to the model of *executing sequences of commands* found in imperative languages)

Strongly and statically typed. Each expression in the language is *assigned a type* describing the possible set of values it can evaluate to, and *type checking* at the time of compilation ensures that no runtime type errors occur.

Type inferencing. Except in a few instances, ML is capable of *deducing* the types of identifiers from the context. There is *no need to declare every identifier* before it is used.

SML: Overview Summary 2

Parametric polymorphism. A function gets a polymorphic type when it can operate uniformly over any value of any given type.

Modularity. A state-of-the-art module system, based on the notions of *structures* (containing the actual code), *signatures* (the type of structures) and *functors* (parametrized structures).

Functional Pseudocode for writing algorithms

An algorithm will be written in a mixture of English and standard mathematical notation (usually called *pseudo-code*). Usually,

- algorithms written in a natural language are often ambiguous
- mathematical notation is not ambiguous, but still cannot be understood by machine
- algorithms written by us use various mathematical properties. We know them, but the machine doesn't.
- Even mathematical notation is often not quite precise and certain intermediate objects or steps are left to the understanding of the reader (e.g. the use of "$\cdots$" and ":").

However

- *Functional pseudo-code* avoids most such implicit assumptions and attempts to make definitions more precise (e.g. the use of induction).
- *Functional pseudo-code* is still concise though more formal than standard mathematical notation.
- However *functional pseudo-code* is not formal enough to be termed a programming language (e.g. it does not satisfy strict grammatical rules and neither is it linear as most formal languages are).
- But *functional pseudo-code* is precise enough to analyse the correctness and complexity of an algorithm, whereas standard mathematical notation may mask many important details.
- We may freely borrow from the notation of the functional programming language to express various *data-structuring* features.

An Example

Suppose `Real.Math.sqrt` were not available to us!

isqrt(n) of a non-negative integer n is the integer $k \geq 0$ such that $k^2 \leq n < (k+1)^2$

That is,

$$isqrt(n) = \begin{cases} \perp & \text{if } n < 0 \\ k & \text{otherwise} \end{cases}$$

where

$$0 \leq k^2 \leq n < (k+1)^2$$

This value of k is unique!

$$\begin{aligned} & 0 \leq k^2 \leq n < (k+1)^2 \\ \Rightarrow\ & 0 \leq k \leq \sqrt{n} < k+1 \\ \Rightarrow\ & 0 \leq k \leq n \end{aligned}$$

Strategy. Use this fact to close in on the value of k. Start with the interval $[l, u] = [0, n]$ and try to shrink it till it collapses to the interval $[k, k]$ which contains a single value.

If $n = 0$ then $isqrt(n) = 0$.
Otherwise with $[l, u] = [0, n]$ and

$$\boxed{l^2 \leq n < u^2}$$

use one or both of the following to shrink the interval $[l, u]$.

- if $(l+1)^2 \leq n$ then try $[l+1, u]$
  otherwise $l^2 \leq n < (l+1)^2$ and $k = l$
- if $u^2 > n$ then try $[l, u-1]$
  otherwise $(u-1)^2 \leq n < u^2$ and $k = u-1$

$$isqrt(n) = \begin{cases} \bot & \text{if } n < 0 \\ 0 & \text{if } n = 0 \\ shrink(n, 0, n) & \text{if } n > 0 \end{cases}$$

where

$$shrink(n, l, u) = \begin{cases} l & \text{if } l = u \\ shrink(n, l+1, u) & \text{if } l < u \text{ and } (l+1)^2 \leq n \\ l & \text{if } l < u \text{ and } (l+1)^2 > n \\ shrink(n, l, u-1) & \text{if } l < u \text{ and } u^2 > n \\ u-1 & \text{if } l < u \text{ and } (u-1)^2 \leq n \\ \bot & \text{if } l > u \end{cases}$$

In the above algorithm the function *isqrt* uses the function *shrink* which is recursively defined. Beginning with an initial closed interval $[0, n]$, *shrink* reduces the size of the interval by 1 i each recursive call. The complexity of the algorithm is therefore $O(n)$ where n is the input to the function *isqrt*.

A better algorithm would be as follows. Here the interval is "halved" with each recursive evaluation of *shrink*

$$isqrt(n) = \begin{cases} \bot & \text{if } n < 0 \\ 0 & \text{if } n = 0 \\ shrink(n, 0, n) & \text{if } n > 0 \end{cases}$$

where

$$shrink(n, l, u) = \begin{cases} l & \text{if } l = u \text{ or } u = l + 1 \\ shrink(n, m, u) & \text{if } l < u \text{ and } m^2 \leq n \\ shrink(n, l, m) & \text{if } l < u \text{ and } m^2 > n \\ \bot & \text{if } l > u \end{cases}$$

where

$$m = \lfloor (l + u)/2 \rfloor$$

Another Example

```
(* Program for generating primes upto some number *)

fun primeWRT (m, []) = true
  | primeWRT (m, h::t) =
    if m mod h = 0 then false
    else primeWRT (m, t)

fun generateFrom (P, m, n) =
    if m > n then P
    else if primeWRT (m, P)
    then (
           generateFrom ((m::P), m+2, n)
         )
    else generateFrom (P, m+2, n)

fun primesUpto n = if n < 2 then []
                   else if n=2 then [2]
                   else if (n mod 2 = 0) then primesUpto (n-1)
                   else generateFrom ([2], 3, n);
```

Lecture 4: Standard ML Computations

Tuesday 02 Aug 2011

Computations: Simple

For most simple expressions it is

- left to right, and
- top to bottom

except when

- presence of brackets
- precedence of operators

determine otherwise.

Hence

Simple computations

$$
\begin{aligned}
& 4 + 6 - (4 + 6) \texttt{ div } 2 \\
= \; & 10 - (4 + 6) \texttt{ div } 2 \\
= \; & 10 - 10 \texttt{ div } 2 \\
= \; & 10 - 5 \\
= \; & 5
\end{aligned}
$$

Computations: Composition

$$f(x) = x^2 + 1$$
$$g(x) = 3 * x + 2$$

Then for any value say $a = 4$

$$\begin{aligned} & f(g(a)) \\ = \; & f(3 * 4 + 2) \\ = \; & f(14) \\ = \; & 14^2 + 1 \\ = \; & 196 + 1 \\ = \; & 197 \end{aligned}$$

This is the *Leftmost-innermost computation rule*.

Composition: Alternative

$$f(x) = x^2 + 1$$
$$g(x) = 3 * x + 2$$

Why not the following?

$$\begin{aligned} & f(g(a)) \\ = \; & g(4)^2 + 1 \\ = \; & (3 * 4 + 2)^2 + 1 \\ = \; & (12 + 2)^2 + 1 \\ = \; & 14^2 + 1 \\ = \; & 196 + 1 \\ = \; & 197 \end{aligned}$$

This is the *Leftmost-outermost computation rule*.

Compositions: Compare

$$f(x) = x^2 + 1$$
$$g(x) = 3 * x + 2$$

Leftmost-innermost computation

$$\begin{aligned} & f(g(a)) \\ = \; & f(3 * 4 + 2) \\ = \; & f(14) \\ \\ = \; & 14^2 + 1 \\ = \; & 196 + 1 \\ = \; & 197 \end{aligned}$$

Leftmost-outermost computation

$$\begin{aligned} & f(g(a)) \\ = \; & g(4)^2 + 1 \\ = \; & (3 * 4 + 2)^2 + 1 \\ = \; & (12 + 2)^2 + 1 \\ = \; & 14^2 + 1 \\ = \; & 196 + 1 \\ = \; & 197 \end{aligned}$$

Compositions: Compare

Question 1: Which is more correct? Why?
Question 2: Which is easier to implement?
Question 3: Which is more efficient?

Computations in SML: Composition

A computation of $f(g(a))$ proceeds thus:

- $g(a)$ is evaluated to some value, say b
- $f(b)$ is next evaluated

Recursion

$$factL(n) = \begin{cases} 1 & \text{if } n \leq 0 \\ factL(n-1) * n & \text{otherwise} \end{cases}$$

```
fun factL n = if n<=0 then 1 else factL (n-1) * n
```

$$factR(n) = \begin{cases} 1 & \text{if } n \leq 0 \\ n * factR(n-1) & \text{otherwise} \end{cases}$$

```
fun factR n = if n<=0 then 1 else n * factR (n-1)
```

Recursion: Left

$$
\begin{aligned}
& factL(4) \\
= \; & (factL(4-1) * 4) \\
= \; & (factL(3) * 4) \\
= \; & ((factL(3-1) * 3) * 4) \\
= \; & ((factL(2) * 3) * 4) \\
= \; & (((factL(2-1) * 2) * 3) * 4) \\
= \; & \cdots
\end{aligned}
$$

Recursion: Right

$$
\begin{aligned}
& factR(4) \\
= \; & (4 * factR(4-1)) \\
= \; & (4 * factR(3)) \\
= \; & (4 * (3 * factR(3-1))) \\
= \; & (4 * (3 * factR(2))) \\
= \; & (4 * (3 * (2 * factR(2-1)))) \\
= \; & \cdots
\end{aligned}
$$

Factorial: Tail Recursion 1

- The recursive call precedes the multiplication operation. *Change it!*
- Define a state variable p which contains the product of all the values that one must remember
- Reorder the computation so that the computation of p is performed before the recursive call.
- For that redefine the function in terms of p.

Factorial: Tail Recursion 2

$$factL2(n) = \begin{cases} 1 & \text{if } n \leq 0 \\ factL\_tr(n, 1) & \text{otherwise} \end{cases}$$

where

$$factL\_tr(n, p) = \begin{cases} p & \text{if } n \leq 0 \\ factL\_tr(n-1, np) & \text{otherwise} \end{cases}$$

```
fun factL2 n =  if n <= 0 then 1
                else let fun factL_tr (n, p) =
                             if n <= 0 then p
                             else factL_tr (n-1, n*p)
                     in factL_tr(n, 1)
                     end
```

A Tail-Recursive Computation

$$
\begin{aligned}
& factL2(4) \\
\leadsto\ & factL\_tr(4,1) \\
\leadsto\ & factL\_tr(3,4) \\
\leadsto\ & factL\_tr(2,12) \\
\leadsto\ & factL\_tr(1,24) \\
\leadsto\ & factL\_tr(0,24) \\
\leadsto\ & 24
\end{aligned}
$$

Factorial: Issues

Correctness : Prove (by induction on n) that for all $n \geq 0$, $factL2(n) = n!$.

Termination : Prove by induction on n that every computation of $factL2$ terminates.

Space complexity : Prove that $\mathcal{S}_{factL2(n)} = O(1)$ (as against $\mathcal{S}_{factL(n)} \propto n$).

Time complexity : Prove that $\mathcal{T}_{factL2(n)} = O(n)$

Lecture 5: Standard ML Scope Rules

Wednesday 03 Aug 2011

```
(* Program for generating primes upto some number *)

fun primeWRT (m, []) = true
  | primeWRT (m, h::t) =
    if m mod h = 0 then false
    else primeWRT (m, t)

fun generateFrom (P, m, n) =
    if m > n then P
    else if primeWRT (m, P)
    then (
           generateFrom ((m::P), m+2, n)
         )
    else generateFrom (P, m+2, n)

fun primesUpto n = if n < 2 then []
                   else if n=2 then [2]
                   else if (n mod 2 = 0) then primesUpto (n-1)
                   else generateFrom ([2], 3, n);
```

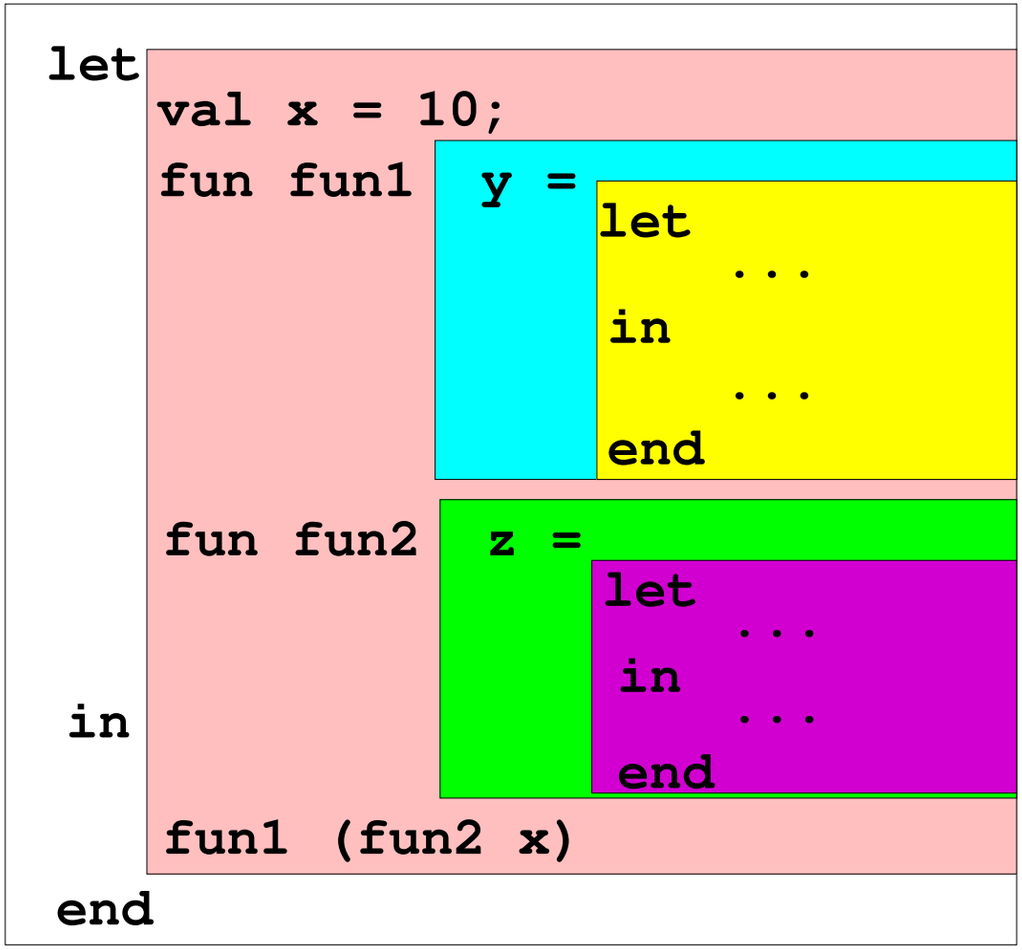
```
(* Program for generating primes upto some number *)
local
    fun primeWRT (m, []) = true
      | primeWRT (m, h::t) = if m mod h = 0 then false
                             else primeWRT (m, t)

    fun generateFrom (P, m, n) =
        if m > n then P
        else if primeWRT (m, P)
        then ( print (Int.toString (m)^" is a prime\n");
               generateFrom ((m::P), m+2, n)
             )
        else generateFrom (P, m+2, n)

in fun primesUpto n =
    if n < 2 then []
    else if n=2 then [2]
    else if (n mod 2 = 0) then primesUpto (n-1)
    else generateFrom ([2], 3, n)
end
```

```
(* Program for generating primes upto some number *)
local
    local
        fun primeWRT (m, []) = true
          | primeWRT (m, h::t) = if m mod h = 0 then false
                                 else primeWRT (m, t)
    in  fun generateFrom (P, m, n) =
            if m > n then P
            else if primeWRT (m, P)
            then ( print (Int.toString (m)^" is a prime\n");
                   generateFrom ((m::P), m+2, n)
                 )
            else generateFrom (P, m+2, n)
    end
in fun primesUpto n =
    if n < 2 then []
    else if n=2 then [2]
    else if (n mod 2 = 0) then primesUpto (n-1)
    else generateFrom ([2], 3, n)
end
```

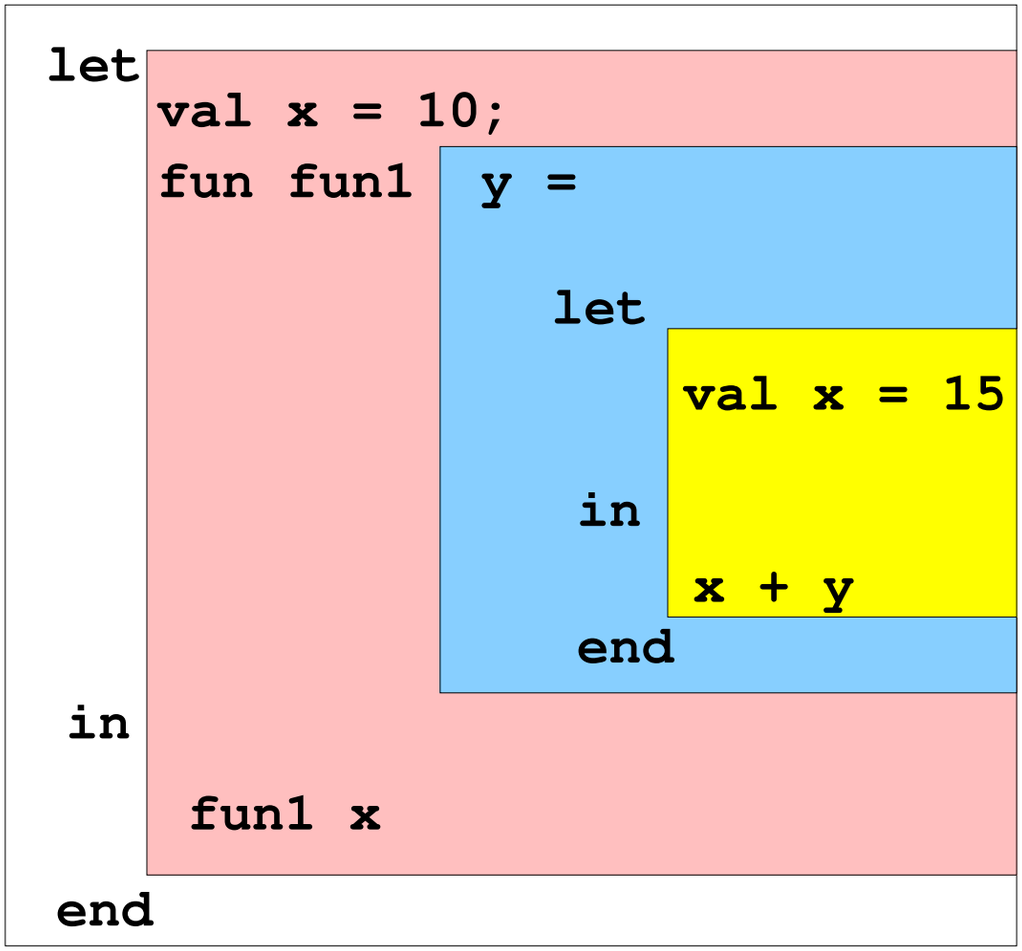
```
(* Program for generating primes upto some number *)
fun primesUpto n =
    if n < 2 then []
    else if n=2 then [2]
    else if (n mod 2 = 0) then primesUpto (n-1)
    else let fun primeWRT (m, []) = true
               | primeWRT (m, h::t) =
                 if m mod h = 0 then false
                 else primeWRT (m, t);
             fun generateFrom (P, m, n) =
                 if m > n then P
                 else if primeWRT (m, P)
                 then ( print (Int.toString (m)^" is a prime\n");
                        generateFrom ((m::P), m+2, n)
                      )
                 else generateFrom (P, m+2, n)
         in generateFrom ([2], 3, n)
         end
```

```
(* Program for generating primes upto some number *)
fun primesUpto n =
    if n < 2 then []
    else if n=2 then [2]
    else if (n mod 2 = 0) then primesUpto (n-1)
    else let fun generateFrom (P, m, n) =
                 let fun primeWRT (m, []) = true
                       | primeWRT (m, h::t) =
                         if m mod h = 0 then false
                         else primeWRT (m, t)
                 in  if m > n then P
                     else if primeWRT (m, P)
                     then ( print (Int.toString (m)^" is a prime\n");
                            generateFrom ((m::P), m+2, n)
                          )
                     else generateFrom (P, m+2, n)
                 end
        in generateFrom ([2], 3, n)
        end
```

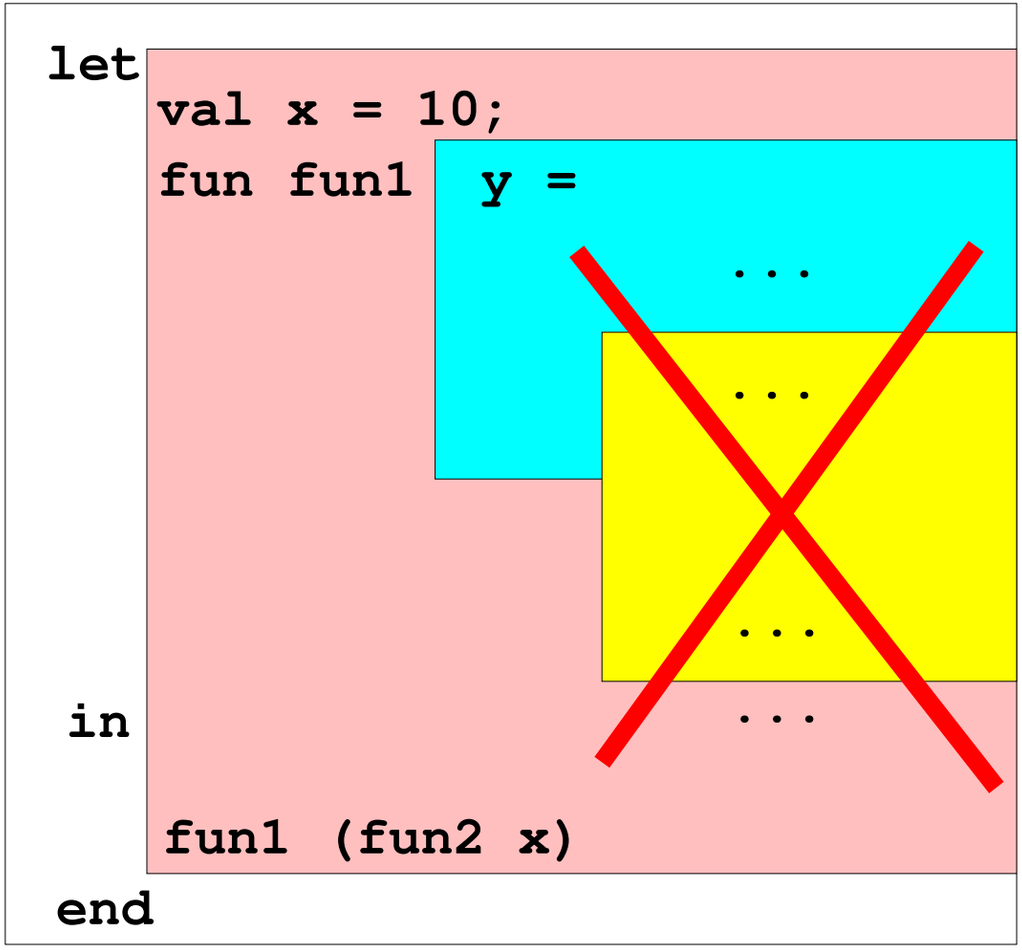
Disjoint Scopes



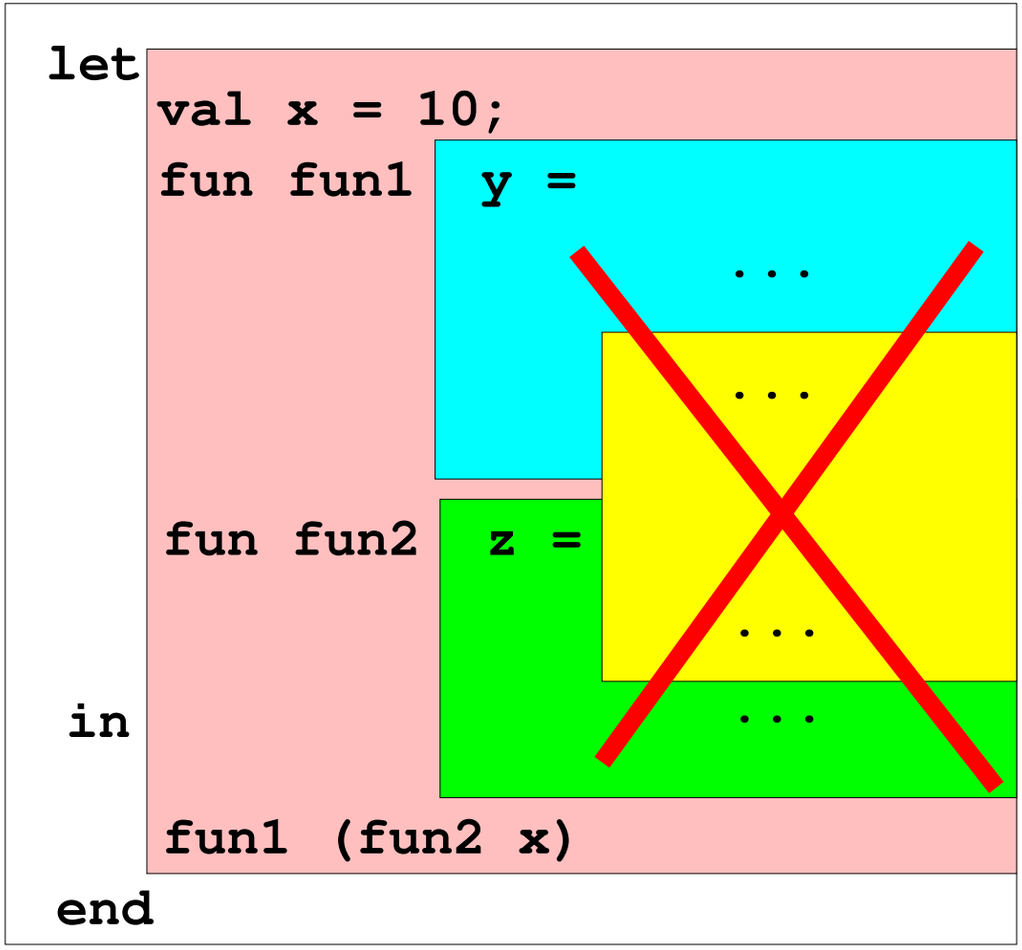
Nested Scopes



Overlapping Scopes



Spannning

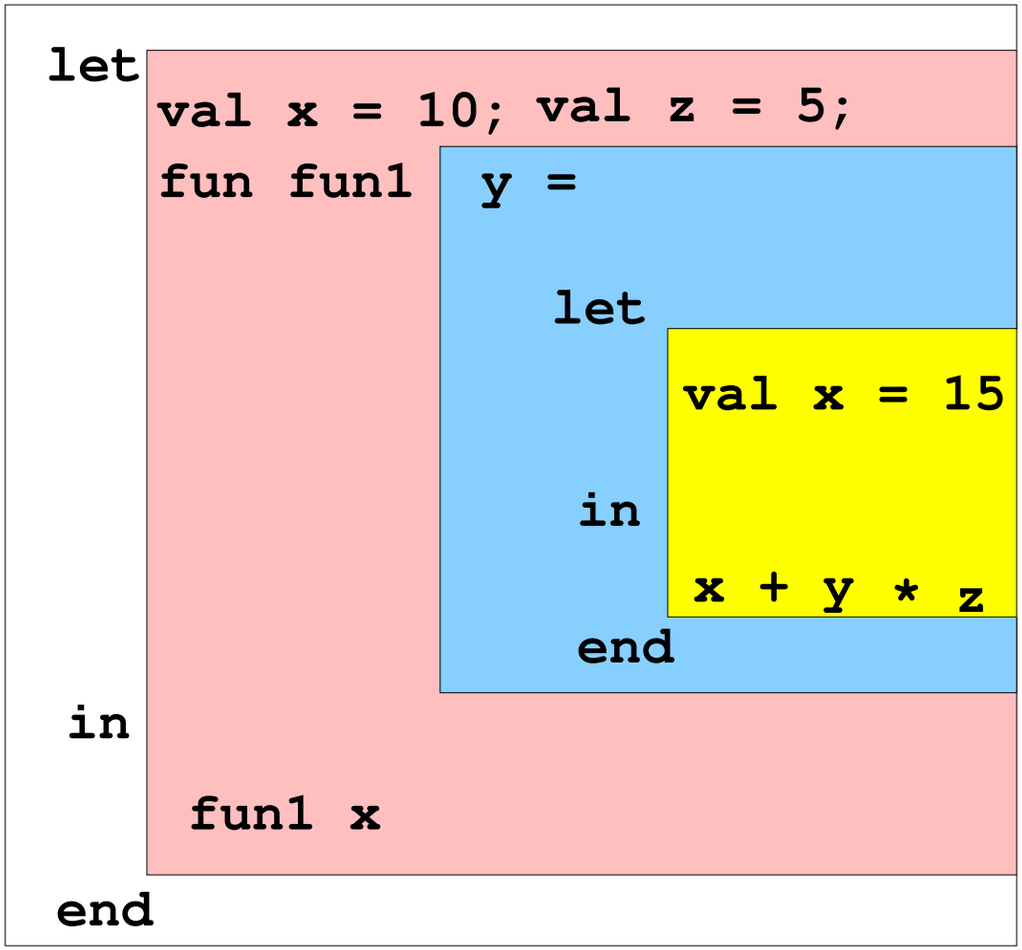


Scope & Names

- A name may occur either as being defined or as a use of a previously defined name
- The same name may be used to refer to different objects.
- The use of a name refers to the textually most recent definition in the innermost enclosing scope

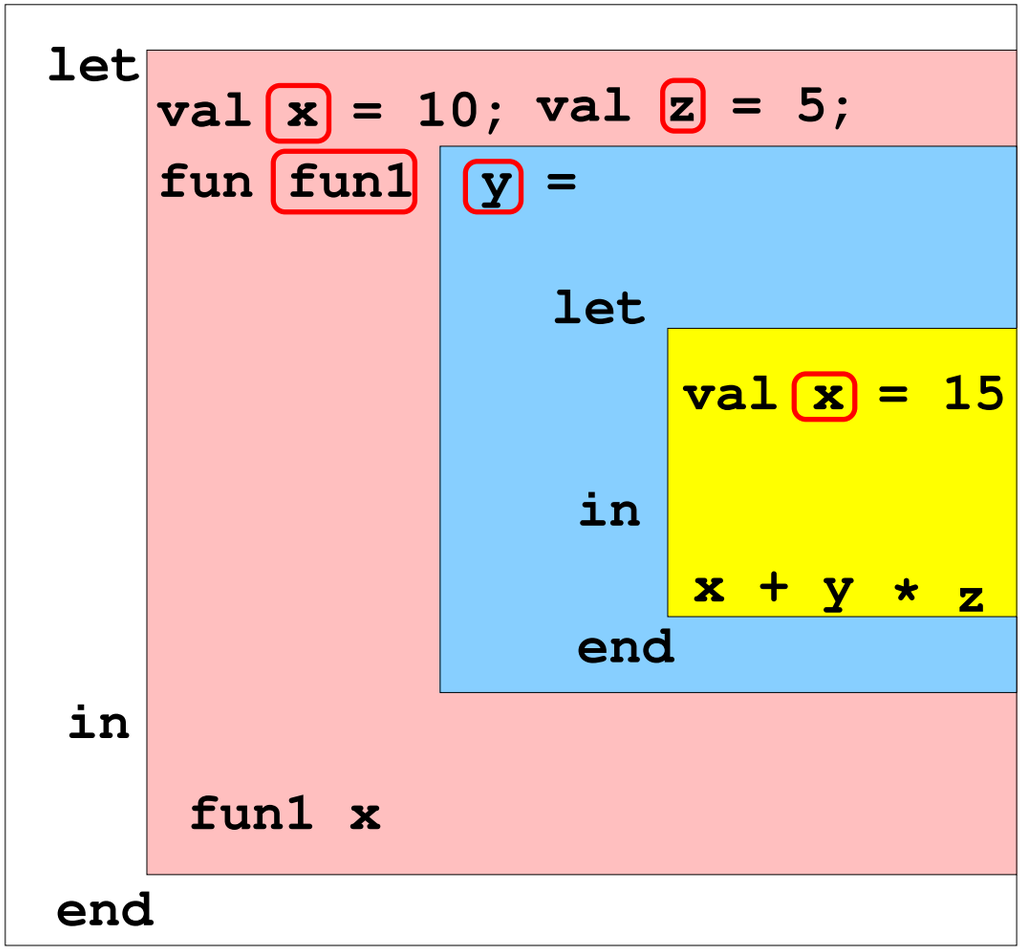
diagram

Names & References: 0



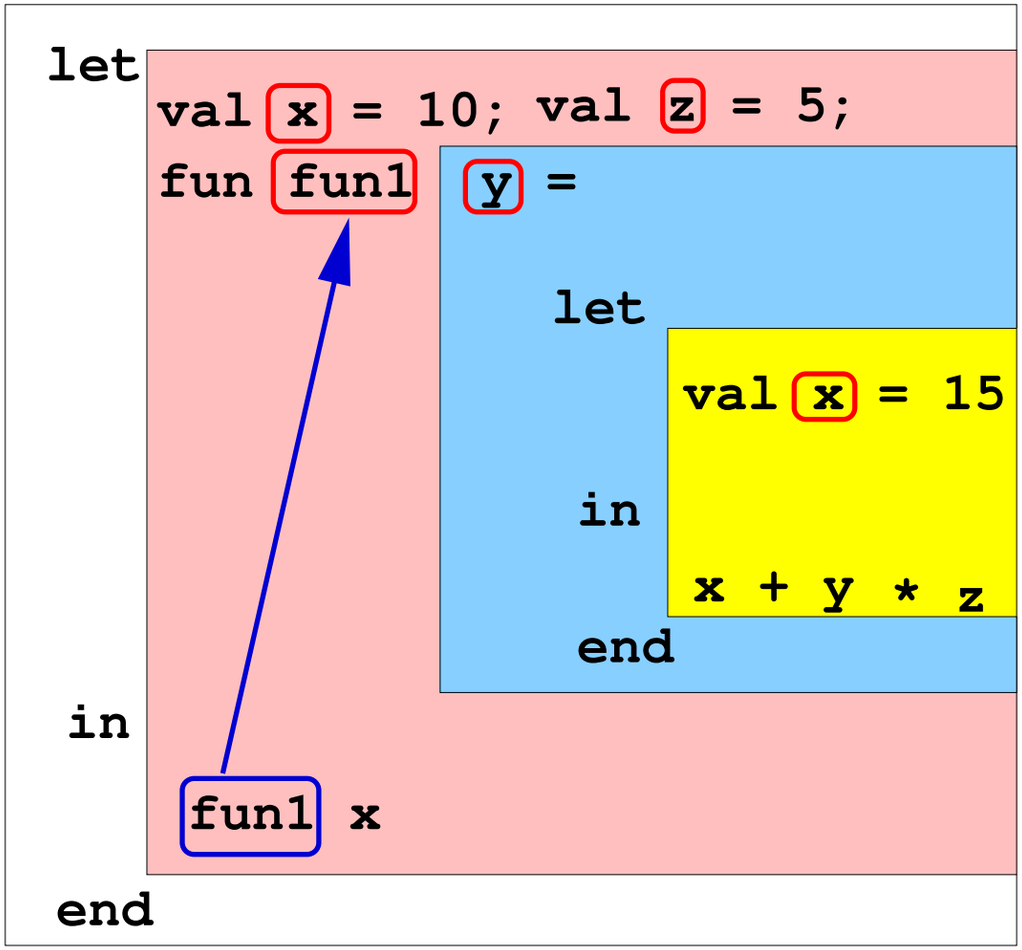
Back to Scope & Names

Names & References: 1



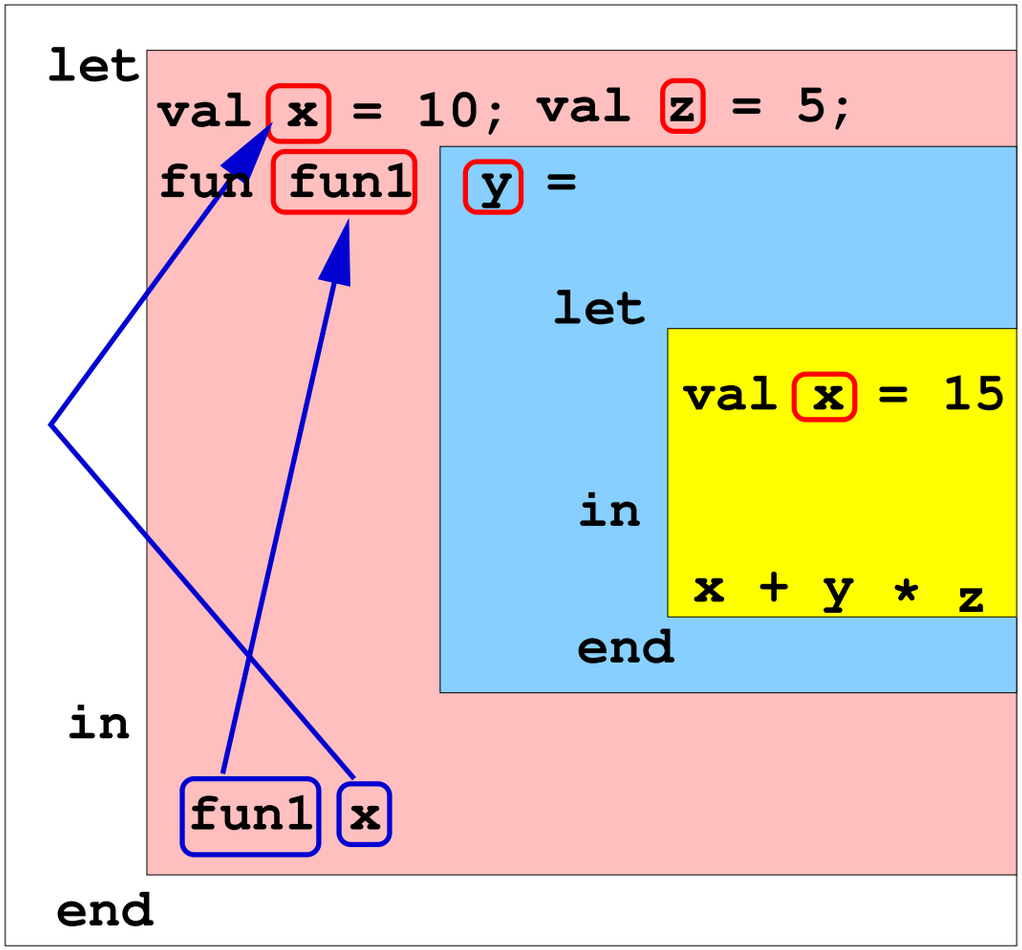
Back to Scope & Names

Names & References: 2



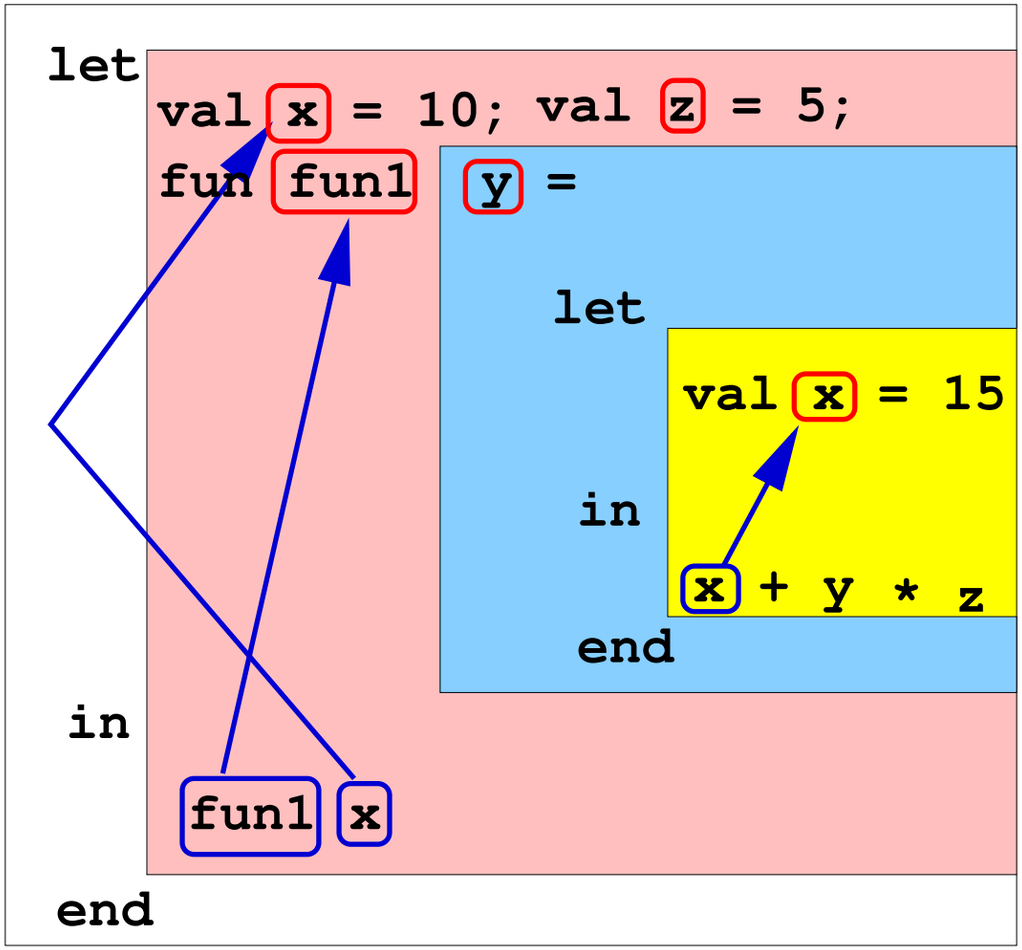
Back to Scope & Names

Names & References: 3



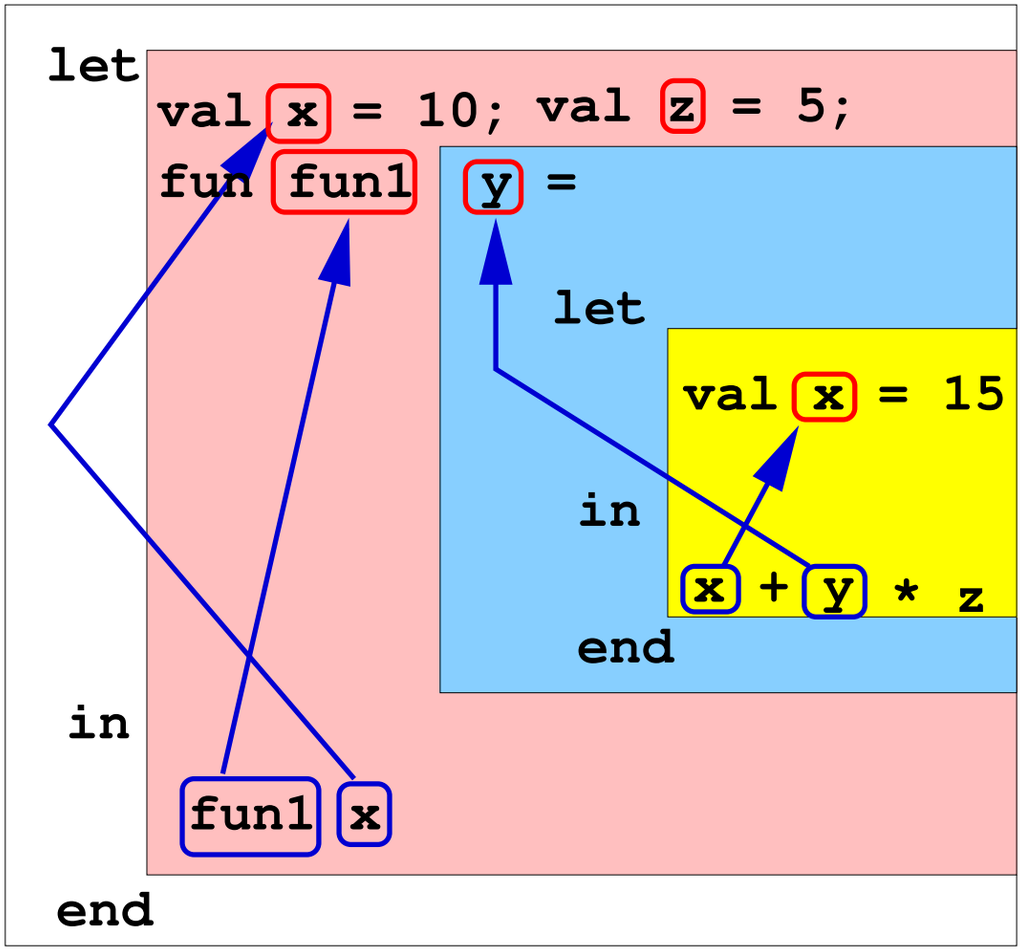
Back to Scope & Names

Names & References: 4



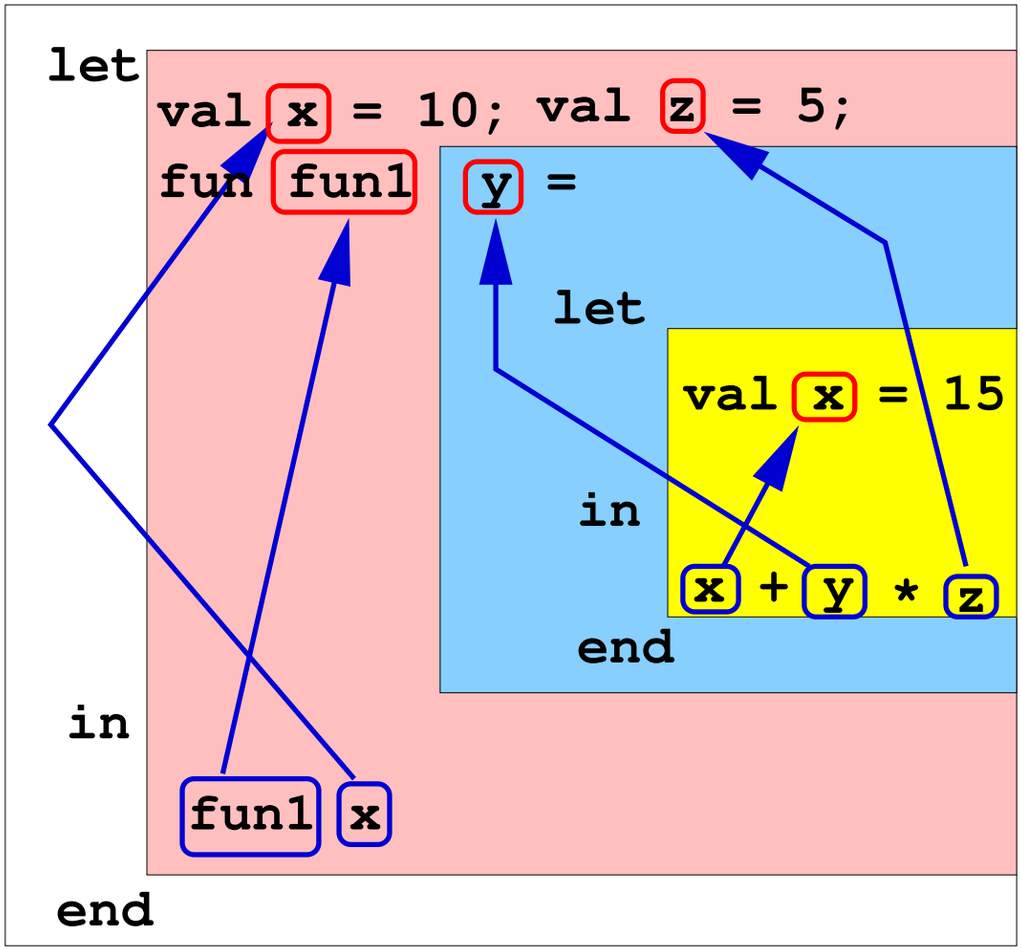
Back to Scope & Names

Names & References: 5



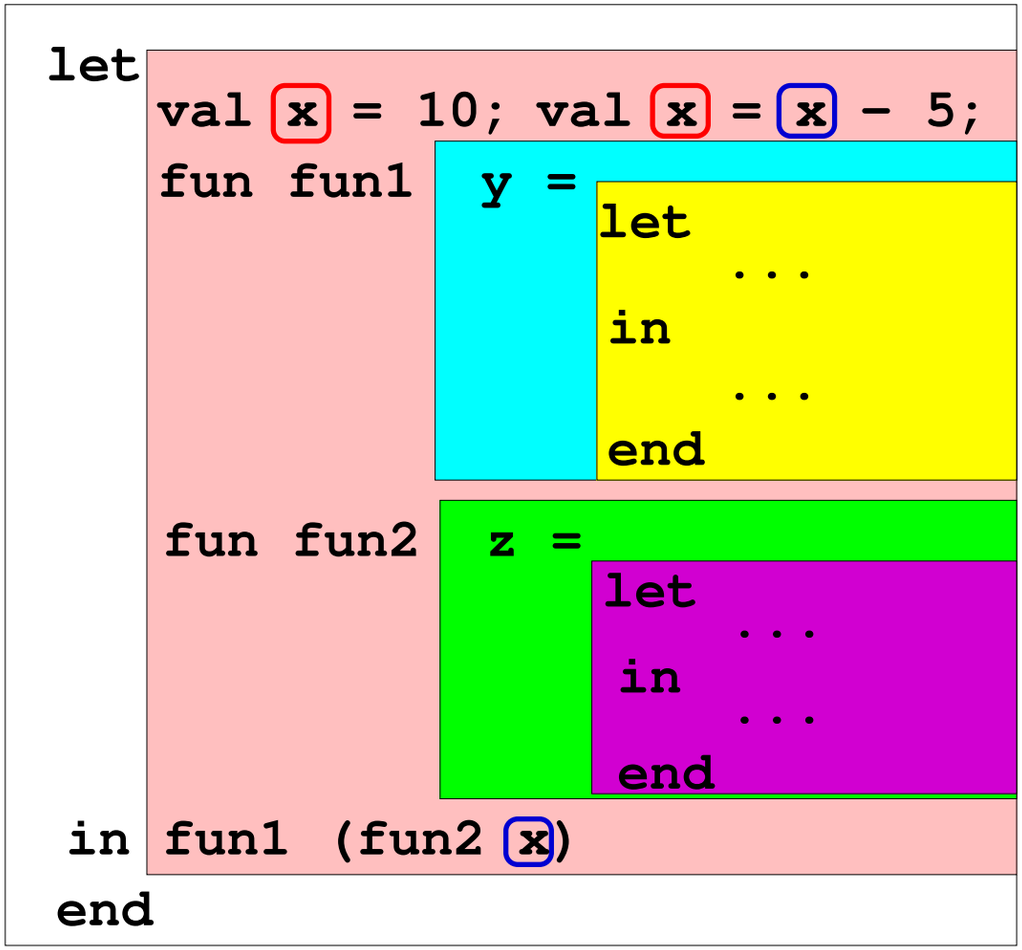
Back to Scope & Names

Names & References: 6



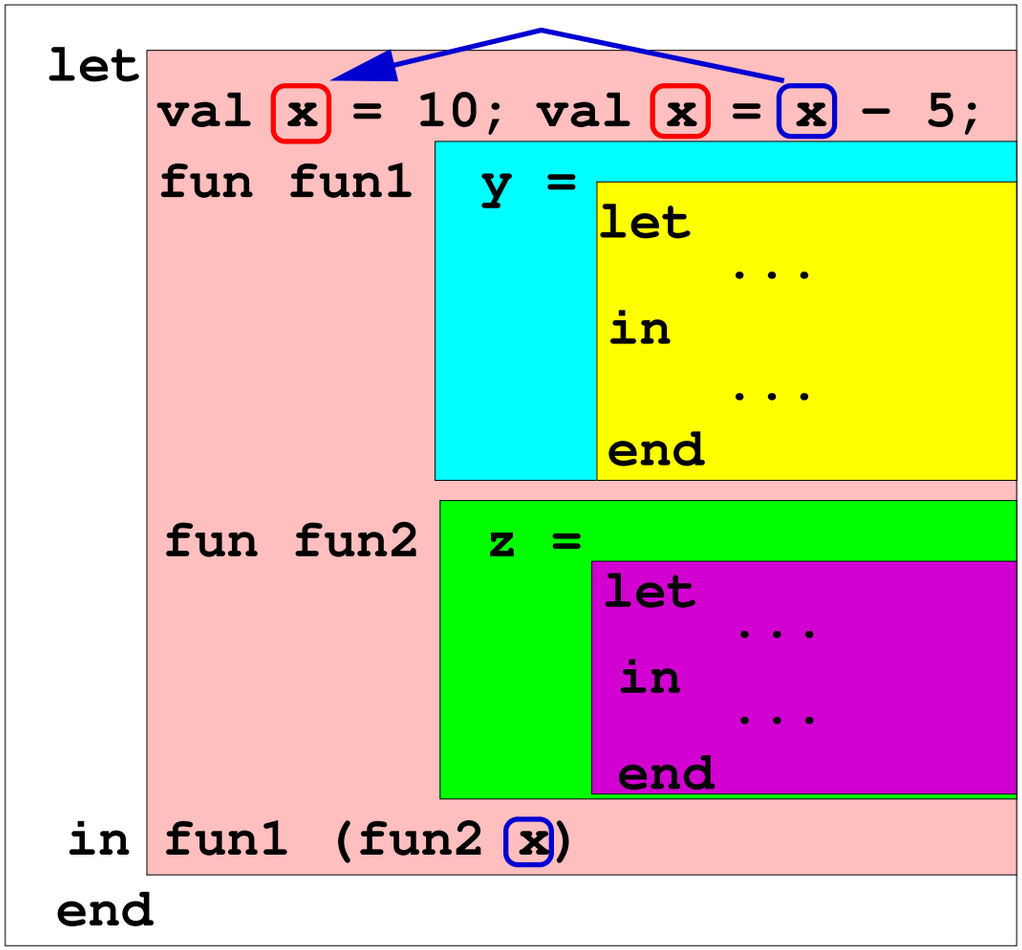
Back to Scope & Names

Names & References: 7



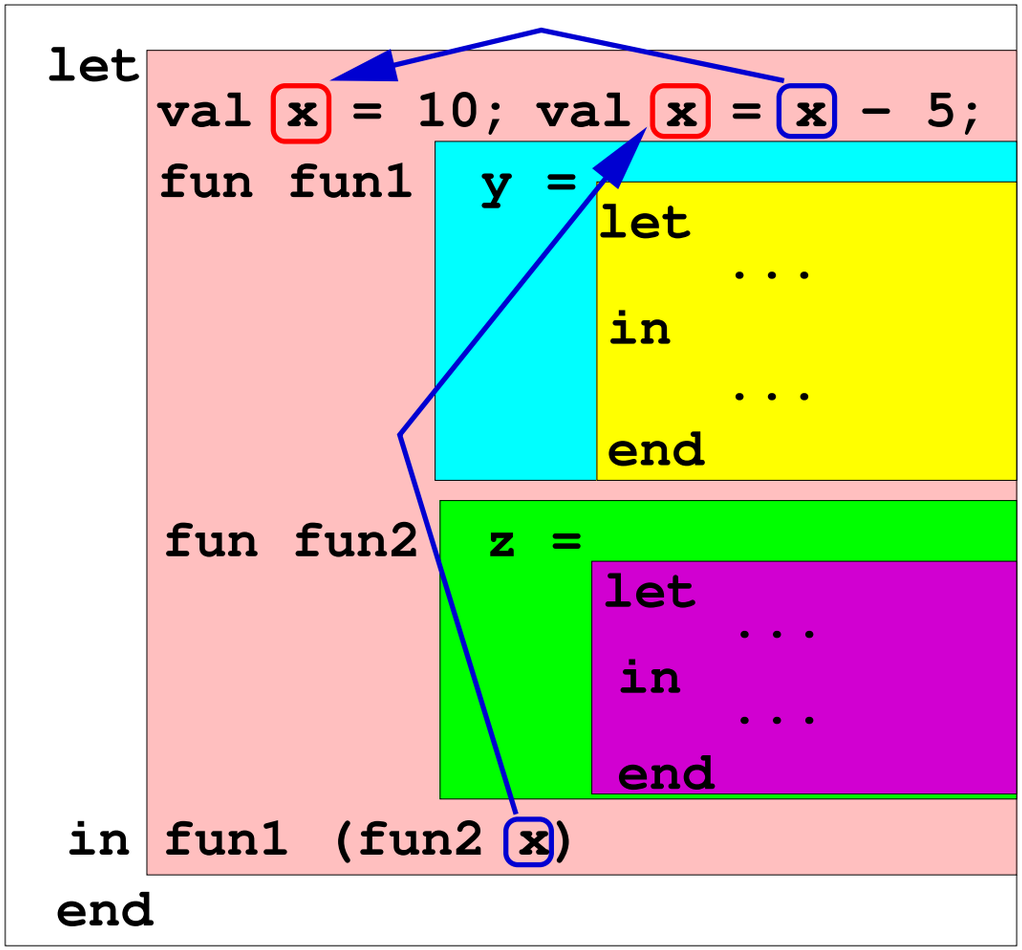
Back to Scope & Names

Names & References: 8



Back to Scope & Names

Names & References: 9



Back to Scope & Names

Definition of Names

Definitions are of the form

qualifier *name* . . . = *body*

- `val` *name* =
- `fun` *name* (*argnames*) =
- `local` *definitions*
  `in` *definition*
  `end`

Use of Names

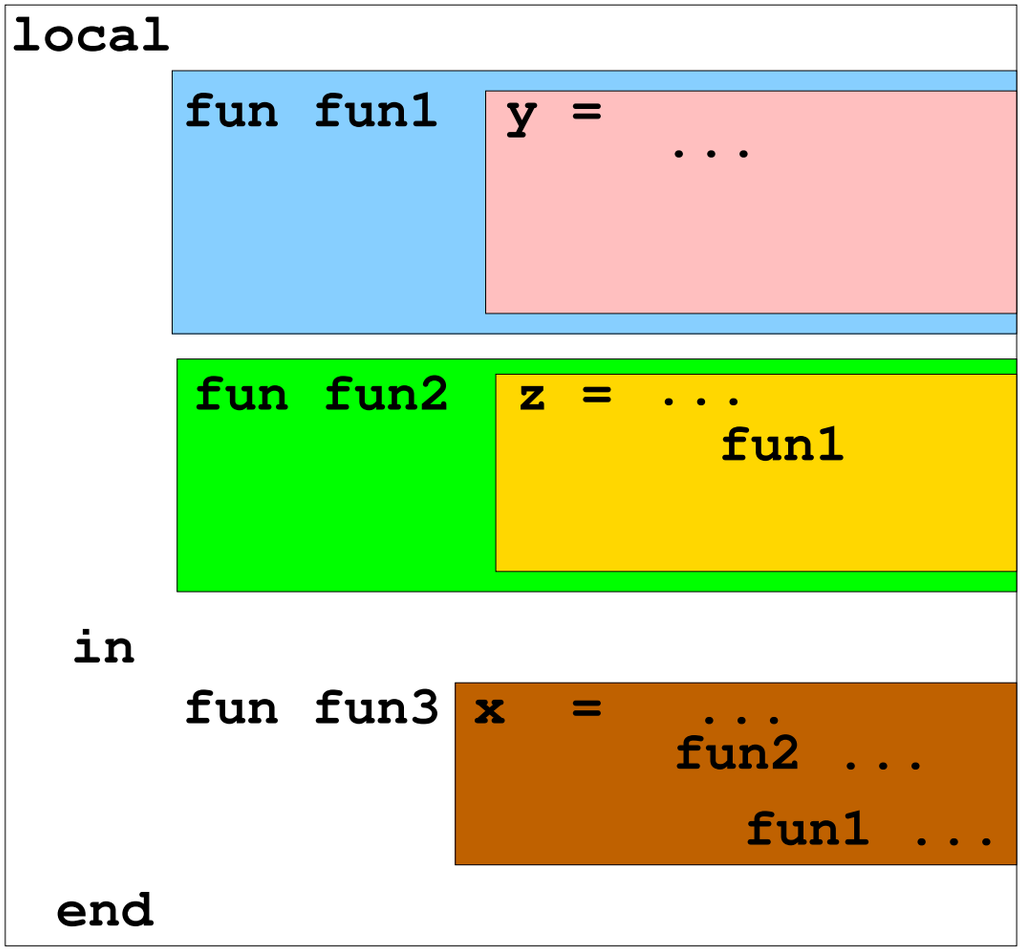
Names are used in expressions.
Expressions may occur

- by themselves – to be evaluated
- as the *body* of a definition
- as the *body* of a let-expression

```
let definitions
in  expression
end
```

use of local

Scope & `local`



Lecture 6: Sample Sort Programs

Friday 05 Aug 2011

6.1. Insertion Sort

Let's consider the development of a program to sort a list using the insertion sort algorithm, which you must have all studied before. Notice the use of induction (basis, hypothesis and induction step) inherent in this algorithm.

Problem How do you sort a list of elements by insertion?

For the purpose of development of this algorithm we assume that we are given

input. A list of elements of some unspecified type such that there exists a pre-defined total ordering relation R on the type of the elements that make up the list.

Our sort function will take this total ordering and the list of elements as parameters.

Strategy. The following cases are to be considered.

Basis. The empty list (and the one-element list) are already sorted.

Induction hypothesis. Assume a list of length $m \geq 0$ can be sorted.

Induction step. Given a list of $n = m + 1$ elements,

1. sort the tail of the list (consisting of $n - 1 = m$ elements). By the induction hypothesis, we know how to do this!
2. insert the first element into this sorted list *at an appropriate position* to obtain a sorted list of length n.

Subproblem How does one insert an element x into a sorted list L of length $m \geq 0$ to obtain a sorted list of length $m + 1$?

Strategy. The following cases need to be considered.

Basis. If the given sorted list L is empty, then $[x]$ is the resulting sorted list.

Induction hypothesis. Assume it is possible to insert x into a sorted list of length $k \geq 0$ to obtain a sorted list of length $k+1$ for $k < m$.

Induction step. Assume given a sorted list L of length $m > 0$. Since $m > 0$, L is non-empty and hence $L = h :: t$ where h is the head of the list and t is the tail. Further t is a list of length $m - 1$.

1. Compute $R(x, h)$.

   Case $R(x, h) = true$**.** Then $x :: L$ is the required sorted list of length $m + 1$.

   Case $R(x, h) = false$**.** Then insert x into t so as to produce a sorted list t' of length m (this is possible by the induction hypothesis). Then $h :: t'$ is the required sorted list of length $m + 1$.

Here is the strategy implemented in functional pseudocode.

$$insertSort\ R\ L = \begin{cases} [] & \text{if } L \sim [] \\ insert\ R\ (insertSort\ R\ t)\ h & \text{elseif } L \sim h :: t \end{cases}$$

where

$$insert\ R\ L\ x = \begin{cases} [x] & \text{if } L \sim [] \\ x :: L & \text{elseif } L \sim h :: t \wedge R(x, h) \\ h :: (insert\ R\ t\ x) & \text{else} \end{cases}$$

We use the notation $\sim$ to indicate "structural pattern-match" rather than equality. Hence in our functioal pseudo-code, "$L \sim h :: t$" denotes the statement *"L is of the form $h :: t$ where h is the head of the list L and t is the tail of the list L". The usual static scope rules for names apply.*

```
(*------------------------- INSERTION SORT ---------------------- *)
(* R is assumed to be a total ordering relation *)
fun insertSort R [] = []
  | insertSort R (h::t) =
    let fun insert R [] x = [x]
          | insert R (h::t) x =
            if R (x, h) then x::(h::t)
            else h::(insert R t x)
        val rest = insertSort R t
    in  insert R rest h
    end;

(* Test
val i = insertSort;
i (op <) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
i (op <=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
i (op >) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
i (op >=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
*)
```

Strategy.

Basis. The empty list and the one-element list are already sorted.

Induction hypothesis. Assume a list of length $m \geq 0$ can be sorted.

Induction step. Given a list of $n > 1$ elements,

1. Find and remove the "R-minimal" element from the list of length $n > 1$.
2. Sort the rest of the list (which is of length $n - 1$).
3. Prepend the list with the "R-minimal" element.

$$selSort\ R\ L = \begin{cases} L & \text{if } L \sim [] \vee L \sim [h] \\ m :: (selSort\ R\ M) & \text{else} \end{cases}$$

where

$$(m, M) \ = \ findMin\ R\ L$$

where

$$findMin\ R\ L = \begin{cases} \bot & \text{if } L \sim [] \\ (h, []) & \text{elseif } L \sim [h] \\ (m, L') & \text{elseif } L \sim h :: t \end{cases}$$

where

$$(m, L') = \begin{cases} (m, h :: t') & \text{if } (m, t') = (findMin\ R\ t) \wedge R(m, h) \\ (h, t) & \text{else} \end{cases}$$

```
(* ------------------------- SELECTION SORT ------------------------- *)
(* R is assumed to be a total ordering relation *)
fun selSort R [] = []
  | selSort R [h] = [h]
  | selSort R (L as h::t) =
    let exception emptyList;
        (* findMin finds the minimum element in the list and removes it *)
        fun findMin R [] = raise emptyList
          | findMin R [h] = (h, [])
          | findMin R (h::t) =
            let val (m, tt) = findMin R t;
            in  if R(m, h) then (m, h::tt) else (h, t)
            end;
        val (m, LL) = findMin R L
    in m::(selSort R LL)
    end;

(* Test
val s = selSort;
s (op <) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
s (op <=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
s (op >) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
s (op >=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
*)
```

```
(* ------------------------- BUBBLE SORT ------------------------- *)
local
    fun bubble R [] = []
      | bubble R [h] = [h]
      | bubble R (f::s::t) = (* can't bubble without at least 2 elements *)
        if R (f, s) then f::(bubble R (s::t))
        else s::(bubble R (f::t))
    fun unsorted R [] = false
      | unsorted R [h] = false
      | unsorted R (f::s::t) =
        if (f=s)  then (unsorted R (s::t))
        else if R (f, s) then (unsorted R (s::t))
        else true
in fun bubbleSort R L =
    if (unsorted R L) then (bubbleSort R (bubble R L))
    else L
end

(* Test
val b = bubbleSort;
b (op <) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
b (op <=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
b (op >) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
b (op >=) [~12, ~24, ~12, 0, 123, 45, 1, 20, 0, ~24];
*)
```

```
(* ------------------------ MERGE SORT ------------------------------- *)
fun mergeSort R [] = []
  | mergeSort R [h] = [h]
  | mergeSort R L = (* can't split a list unless it has > 1 element *)
      let fun split []  = ([], [])
            | split [h] = ([h], [])
            | split (h1::h2::t) =
                let val (left, right) = split t;
                 in (h1::left, h2::right)
                end;
          val (left, right) = split L;
          fun merge (R, [], []) = []
            | merge (R, [], (L2 as h2::t2)) = L2
            | merge (R, (L1 as h1::t1), []) = L1
            | merge (R, (L1 as h1::t1), (L2 as h2::t2)) =
                  if R(h1, h2) then h1::(merge (R, t1, L2))
                  else h2::(merge(R, L1, t2));
          val sortedLeft = mergeSort R left;
          val sortedRight = mergeSort R right;
      in  merge (R, sortedLeft, sortedRight)
      end;
```

```
(* ---------------------------- QUICK SORT ---------------------------- *)
fun quickSort R [] = []
  | quickSort R (h::t) =
    let fun part R p [] = ([], [])
          | part R p (f::r) =
            let val (lesser, greater) = part R p r
            in  if R (f, p) then (f::lesser, greater)
                else (lesser, f::greater)
            end
        val (left, right) = part R h t;
        val sortedLeft = quickSort R left;
        val sortedRight = quickSort R right;
    in  sortedLeft @ (h::sortedRight)
    end;
```

```
(* The lexicographic ordering on strings *)
fun lexlt (s, t) =
    let val Ls = explode (s);
        val Lt = explode (t);
        fun lstlexlt (_, []) = false
        |   lstlexlt ([], (b:char)::M) = true
        |   lstlexlt (a::L, b::M) =
                if (a < b) then true
                else if (a = b) then lstlexlt (L, M)
                     else false
        ;
     in lstlexlt (Ls, Lt)
     end

fun lexleq (s, t) = (s = t) orelse lexlt (s, t)

fun lexgt (s, t) = lexlt(t, s)

fun lexgeq (s, t) = (s = t) orelse lexgt (s, t)
```

Lecture 7: Higher-order Functions

Tuesday 09 Aug 2011

Functions in SML

1. All functions are unary.
   - Parameterless functions take the *empty tuple* as argument
   - Functions with a single parameter take a *single* 1-tuple as argument.
   - Functions of m parameters take a *single* m-tuple as argument.
2. Functions are first-class objects. Any function may be treated as a value (except when ...). So we can have
   - functions on data structures and
   - data structures of functions

Example 7.1 *Creating a list of Functions*

Functions: Mathematics & Programming

Functions in programming differ from mathematical functions in at least two fundamental ways.

1. There is *no notion of function equality* in programming

   Example 7.2 $factL2(n) \stackrel{?}{=} factL\ (n)$ *cannot be checked by a program.*

2. Mathematically equivalent definitions are *not necessarily* program equivalent.

   Example 7.3

$$fL(n) = \begin{cases} 1 & \text{if } n \leq 0 \\ fL(n-1) * n & \text{else} \end{cases} \quad \Big| \quad fL'(n) = \begin{cases} 1 & \text{if } n \leq 0 \\ fL'(n+1)/(n+1) & \text{else} \end{cases}$$

Lists

An $'a\ list\ L$

- is an ordered sequence of elements all of the same type $'a$,
- may be empty (called nil and denoted either by [] or nil).
- only the first element (called the head) of the list is immediately accessible through the unary operation hd.
- the tail of the list for a nonempty list is the list without the head and is obtained by the unary operation tl
- There is an operation (called cons denoted by the infix operation ::) for prepending an element of type $'a$ to a list of type $'a\ list$.
- L satisfies the following conditional equation.

$$L \neq nil \Rightarrow L = (hd\ L) :: (tl\ L) \tag{1}$$

A Progression of Functions

1. Creating a list of Functions
2. Arithmetic Progressions
3. Geometric Progressions

List of Functions: 1

Suppose we want a long list of functions to be generated

$$[incrby1, incrby2, incrby3, \ldots, incrby1000]$$

where the function $incrby$k is a unary function that increments a given input value by k. Here is one way to generate the list

```
fun incrby x = fn y => (x+y);
fun listincrby n = if n <= 0 then []
                   else listincrby (n-1)@[(incrby n)]
```

List of Functions: 2

A more efficient way of doing it would use "::" instead of "@".

```
local
    fun listincrby_tr (m, k, L) =
        if k >= m then L
        else listincrby_tr (m, k+1, (incrby (m-k))::L)

in fun listincrby' n =
       if n <= 0 then []
       else listincrby_tr (n, 0, [])
end
```

```
fun applyl [] x = []
  | applyl (h::t) x = (h x)::(applyl t x)
```

Higher-order Functions on Lists

1. `map` applies a unary function uniformly to all elements of a list and yields the list of result values.

```
fun map f []      = []
  | map f (h::t) = (f h)::(map f t)
```

2. `foldl` applies a binary function to all elements of a list from left to right starting from an identity element.

```
fun foldl f e []  = e
  | foldl f e (h::t) = foldl f (f(h, e)) t
```

3. `foldr` operates from right to left

```
fun foldr f e []  = e
  | foldl f e (h::t) = f (h, foldr f e t)
```

Arithmetic Progressions: 1

$$AP1(a,d,n) = \begin{cases} [] & \text{if } n \leq 0 \\ a :: AP1(a+d,d,n-1) & \text{else} \end{cases}$$

Geometric Progressions: 1

$$GP1(a,d,n) = \begin{cases} [] & \text{if } n \leq 0 \\ a :: GP1(ad,d,n-1) & \text{else} \end{cases}$$

Arithmetic-Geometric Progressions: 1

$$AGP1\ bop\ (a, d, n) = \begin{cases} [] & \text{if } n \leq 0 \\ a :: (AGP1\ bop\ (bop(a, d), d, n - 1)) & \text{else} \end{cases}$$

Arithmetic Progressions: 2

$$AP2(a,d,n) = \begin{cases} [] & \text{if } n \leq 0 \\ AP2\_tr(a,d,n,[]) & \text{else} \end{cases}$$

where

$$AP2\_tr(a,d,n,L) = \begin{cases} L & \text{if } n \leq 0 \\ AP2\_tr(a,d,n-1,(a+d*(n-1)) :: L) & \text{else} \end{cases}$$

Geometric Progressions: 2

$$GP2(a,r,n) = \begin{cases} [] & \text{if } n \leq 0 \\ GP2\_tr(a,r,n,[]) & \text{else} \end{cases}$$

where

$$GP2\_tr(a,r,n,L) = \begin{cases} L & \text{if } n \leq 0 \\ GP2\_tr(a,d,n-1,(a.d^{(n-1)}) :: L) & \text{else} \end{cases}$$

But powering is both an expensive and a wasteful operation.

More Progressions

For any binary operation *bop* define

```
curry2 bop = fn x => fn y => bop(x, y)
```

Then `incrby = curry2 op+` and `multby = curry2 op*`

$$AP3(a,d,n) = \begin{cases} [] & \text{if } n \leq 0 \\ a :: (map(curry2\ op{+}\ d)\ AP3(a,d,n-1)) & \text{else} \end{cases}$$

$$GP3(a,d,n) = \begin{cases} [] & \text{if } n \leq 0 \\ a :: (map\ (curry2\ op{*}\ d)\ AP3(a,d,n-1)) & \text{else} \end{cases}$$

may be generalized to

$$AGP4\ bop\ (a,d,n) = \begin{cases} [] & \text{if } n \leq 0 \\ a :: (map\ (curry2\ bop\ d)(AGP4\ bop\ (a,d,n-1))) & \text{else} \end{cases}$$

$$AP4 = AGP4\ op{+}$$

$$GP4 = AGP4\ op{*}$$

Harmonic Progressions

A **harmonic progression** is one whose elements are reciprocals of the elements of an arithmetic progression.

$$HP4(a,d,n) = map\ reci\ AP4(a,d,n)$$

where $reci\ x = 1.0/(real\ x)$ for each integer x.
We may sum the elements of all the progressions by defining

$$\begin{aligned} sumint &= foldl\ op+\ 0 \\ sumreal &= foldl\ op+\ 0.0 \end{aligned}$$

$$\begin{aligned} AS4(a,d,n) &= sumint(AP4(a,d,n)) \\ GS4(a,d,n) &= sumint(GP4(a,d,n)) \\ HS4(a,d,n) &= sumreal(HP4(a,d,n)) \end{aligned}$$

Lecture 8: Datatypes

Wednesday 10 Aug 2011

Primitive Datatypes

The primitive data types of ML are

Booleans: the type bool defined by the structure Bool

Integers: the type int defined by the structure Int

Reals: the type real defined by the structure Real with a sub-structure Math of useful constants (e.g. Real.Math.pi) and functions (e.g. Real.Math.sin).

Characters: the type char defined by the structure Char

Strings: the type string defined by the structure String

Structured Data

For any data strucure we require the following

Constructors. which permit the creation and extension of the structure.

Destructors or **Deconstructors.** which permit the breaking up of a structure into its component parts.

Defining Equation. *Constructors may be used to reconstruct a data structure pulled apart by its destructors*.

Tuples and Records

Elements of cartesian products of (different or same) types defined by grouping.

Tuples. Records with no field names for the components.

Defining Equation. t = ((#1 t), (#2 t),. . . (#n t))

Records. Tuples with field names for the components. For any record r with field names fn1, ..., fn$m$ we have

Defining Equation. r = {(#fn1 r), (#fn2 r), ..., (#fn$m$ r)}

The List Datatype

A *recursively* defined datatype conforming to the following ML datatype definition

```
datatype 'a list = nil
                 | :: of 'a * 'a list -> 'a list
infix ::
```

Constructors. nil and ::

Destructors. hd and tl

Defining Equation. L = nil ∨ L = (hd L)::(tl L)

The Option Datatype

```
datatype 'a option = NONE | SOME of 'a option
```

Constructors. NONE, SOME

Destructors. valOf

Defining Equation. O = NONE ∨ O = SOME (valOf O)

The Binary Tree Datatype

A recursive user-defined datatype.

```
datatype 'a bintree = Empty | Node of 'a * 'a bintree
```

Constructors. Empty, Node

Destructors. root, leftsubtree, rightsubstree

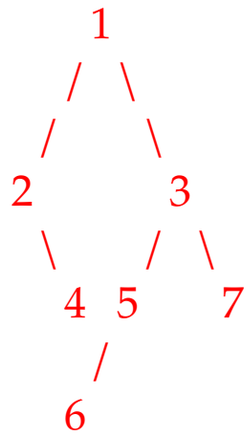
Defining Equation. T = Empty $\vee$ T = Node (root(T), leftsubtree(T), rightsubtree(T))

```
(* The data type binary tree *)

datatype 'a bintree =
         Empty |
         Node of 'a * 'a bintree * 'a bintree

exception Empty_binary_tree

fun isEmpty Empty = true
  | isEmpty _     = false

fun subtrees Empty = raise Empty_binary_tree
  | subtrees (Node(N, Lst, Rst)) = (Lst, Rst)

fun root Empty = raise Empty_binary_tree
  | root (Node(N, _, _)) = N

fun leftsubtree Empty = raise Empty_binary_tree
  | leftsubtree (Node(_, Lst, _)) = Lst

fun rightsubtree Empty = raise Empty_binary_tree
  | rightsubtree (Node(_, _, Rst)) =  Rst

(* Checking whether a given binary tree is balanced *)
```

```
fun height Empty = 0
  | height (Node(N, Lst, Rst)) =
    1+Int.max (height (Lst), height (Rst))

fun isBalanced Empty = true
  | isBalanced (Node(N, Lst, Rst)) =
    (abs (height (Lst) - height (Rst)) <= 1) andalso
    isBalanced (Lst) andalso isBalanced (Rst)

fun size Empty = 0
  | size (Node(N, Lst, Rst)) = 1+size(Lst)+size(Rst)

(* Here is a simplistic definition of preorder traversal *)
fun preorder1 Empty = nil
  | preorder1 (Node(N, Lst, Rst)) =
              [N] @ preorder1 (Lst) @ preorder1 (Rst)

(* The above definition though correct is inefficient because it has
   complexity closer to n^2 since the append function itself is linear in
   the length of the list. We would like an algorithm that is linear in the
   number of nodes of the tree. So here is a new one, which uses an iterative
   auxiliary function that stores the preorder traversal of the right subtree
   and then gradually attaches the preorder traversal of the left subtree
   and finally attaches the root as the head of the list.
*)
```

```
local fun pre (Empty, Llist)  = Llist
        | pre (Node (N, Lst, Rst), Llist) =
          let val Mlist = pre (Rst, Llist)
              val Nlist = pre (Lst, Mlist)
          in  N::Nlist
          end
in fun preorder2 T = pre (T, [])
end

val preorder = preorder2

(* Similarly let's do inorder and postorder traversal *)
fun inorder1 Empty = nil
  | inorder1 (Node(N, Lst, Rst)) =
       inorder1(Lst) @ [N] @ inorder1 (Rst)

local fun ino (Empty, Llist) = Llist
        | ino (Node (N, Lst, Rst), Llist) =
          let val Mlist = ino (Rst, Llist)
              val Nlist = ino (Lst, N::Mlist)
          in  Nlist
          end
in fun inorder2 T = ino (T, [])
end

val inorder = inorder2
```

```
fun postorder1 Empty = nil
  | postorder1 (Node (N, Lst, Rst)) =
    postorder1 (Lst) @ postorder1 (Rst) @ [N]

local fun post (Empty, Llist) = Llist
        | post (Node (N, Lst, Rst), Llist) =
          let val Mlist = post (Rst, N::Llist)
              val Nlist = post (Lst, Mlist)
          in  Nlist
          end
in fun postorder2 T = post (T, [])
end

val postorder = postorder2

(* A map function for binary trees *)

fun BTmap f =
    let fun BTM Empty = Empty
          | BTM (Node(N, Lst, Rst)) =
            Node ((f N), BTM (Lst), BTM (Rst))
    in  BTM
    end

(* Example integer binary tree : Notice that 2 has an empty left subtree
   and 5 has an empty right subtree.
```




```
*)
val t7 = Node (7, Empty, Empty);
val t6 = Node (6, Empty, Empty);
val t4 = Node (4, Empty, Empty);
val t2 = Node (2, Empty, t4);
val t5 = Node (5, t6, Empty);
val t3 = Node (3, t5, t7);
val t1 = Node (1, t2, t3);
```

Lecture 9: Information Hiding

Friday 12 Aug 2011

Datatype: Constructors & Destructors

1. The **Defining Equations** require for each data type a relation between its constructors and destructors.
2. The data type completely reveals its structure through the constructors
3. Even if there are no destructors defined, the structure could be broken down using *pattern-matching*.

1. Information Hiding: Separate Compilation
2. Information Hiding: Abstraction

Information Hiding: Separate Compilation

Information hiding is useful for many reasons

Separate Compilation. Different modules may be compiled separately and

1. a module is loaded only when required by the user program
2. module implementations may be changed, separately compiled and stored. As long as the *interface* to the user remains unchanged the user programs will exhibit no change in behaviour.
3. Many different implementations may be provided for the same package specification, allowing a choice of implementations to the user.

Abstraction.

Information Hiding: Abstraction

Information hiding is useful for many reasons

Separate Compilation.

Abstraction.

1. Requires that the structure of the underlying data should be hidden from user, so that it can be changed whenever found necessary.
2. Reduces clutter in the user code and user code may be read and understood clearly without being distracted by unnecessary details that are not essential for functionality of the user's code.

Abstract Data Types

1. In a datatype definition the constructors of the data type and the structure of the data type are revealed.
2. It is not necessary to program any destructors because of the availability of excellent *pattern-matching* facilites.
3. A datatype may be declared **abstract**, so that absolutely no information about its internal structure is revealed and the only access to its components is through its interface.
4. Without destructor functions available in the interface, no inkling of the components of the structure are made available.

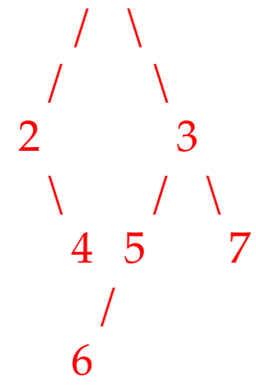
bintree.sml vs. abstype-bintree.sml

abstype-bintree.sml

```
(* The abstract data type binary tree *)

abstype 'a bintree =
         Empty |
         Node of 'a * 'a bintree * 'a bintree
with
        exception Empty_binary_tree

        fun mktree0 () = Empty

        fun mktrees2 (N, TL, TR) = Node(N, TL, TR);

        fun isEmpty Empty = true
          | isEmpty _     = false

        fun subtrees Empty = raise Empty_binary_tree
          | subtrees (Node(N, Lst, Rst)) = (Lst, Rst)

        fun root Empty = raise Empty_binary_tree
          | root (Node(N, _, _)) = N

        fun leftsubtree Empty = raise Empty_binary_tree
          | leftsubtree (Node(_, Lst, _)) = Lst

        fun rightsubtree Empty = raise Empty_binary_tree
```

```
    | rightsubtree (Node(_, _, Rst)) =  Rst

fun height Empty = 0
  | height (Node(N, Lst, Rst)) =
    1+Int.max (height (Lst), height (Rst))

fun isBalanced Empty = true
  | isBalanced (Node(N, Lst, Rst)) =
    (abs (height (Lst) - height (Rst)) <= 1) andalso
    isBalanced (Lst) andalso isBalanced (Rst)

fun size Empty = 0
  | size (Node(N, Lst, Rst)) = 1+size(Lst)+size(Rst)

(* Here is a simplistic definition of preorder traversal *)
fun preorder1 Empty = nil
  | preorder1 (Node(N, Lst, Rst)) =
    [N] @ preorder1 (Lst) @ preorder1 (Rst)

(* The above definition though correct is inefficient because it has
   complexity closer to n^2 since the append function itself is linear in
   the length of the list. We would like an algorithm that is linear in the
   number of nodes of the tree. So here is a new one, which uses an iterative
   auxiliary function that stores the preorder traversal of the right subtree
   and then gradually attaches the preorder traversal of the left subtree
   and finally attaches the root as the head of the list.
```

```
*)

        local fun pre (Empty, Llist)  = Llist
                | pre (Node (N, Lst, Rst), Llist) =
                  let val Mlist = pre (Rst, Llist)
                      val Nlist = pre (Lst, Mlist)
                  in  N::Nlist
                  end
        in fun preorder2 T = pre (T, [])
        end

        val preorder = preorder2

(* A map function for binary trees *)

        fun BTmap f =
            let fun BTM Empty = Empty
                  | BTM (Node(N, Lst, Rst)) =
                    Node ((f N), BTM (Lst), BTM (Rst))
            in  BTM
            end
end (* with abstype *)

(* Example integer binary tree : Notice that 2 has an empty left subtree
   and 5 has an empty right subtree.
```



```
*)
val t0: int bintree = mktree0 ();
val t7 = mktrees2 (7, t0, t0);
val t6 = mktrees2 (6, t0, t0);
val t4 = mktrees2 (4, t0, t0);
val t2 = mktrees2 (2, t0, t4);
val t5 = mktrees2 (5, t6, t0);
val t3 = mktrees2 (3, t5, t7);
val t1 = mktrees2 (1, t2, t3);
```

Lecture 10: Abstract Data Types to Modularity

Wednesday 17 Aug 2011

Information Hiding: Balancing

$$\begin{cases} isBalanced(Empty) & = true \\ isBalanced(Node(N, Lst, Rst)) & = (|(height(Lst) - height(Rst)| \leq 1) \\ & \quad \wedge isBalanced(Lst) \wedge isBalanced(Rst) \end{cases}$$

- Binary tree functions like *size* and *height* require at least one complete traversal of the tree to determine.
- The above code for the binary tree datatype is very expensive.
- One way to make it more efficient:
  - compute information and store it on the node of the tree during construction of the tree and simply read it off from it.
  - Hide this representation from the user.
  - The user cannot tamper with the information
  - There are no updates in functional programming. So once calculated correctly the information is *constant*.

abstype-bintree.sml abstype-bintree2.sml

abstype-bintree2.sml

```
(* The abstract data type binary tree with hidden data *)

(* The bintree.sml and abstype-bintree.sml implementations are both very
   poor when it comes to determining whether a tree is Balanced.

   A look at the code

        fun isBalanced Empty = true
          | isBalanced (Node(N, Lst, Rst)) =
            (abs (height (Lst) - height (Rst)) <= 1) andalso
            isBalanced (Lst) andalso isBalanced (Rst)

   reveals that a large number of traversals of the tree are made
   before deciding whether the tree is balanced. Each call to isBalanced
   also makes calls to height which itself is recursive.

   Given a tree of height h how many times is each node in the tree visited?

   One obvious solution is to keep relevant information about each node at
   the node itself and hide it from a user of the abstype. So with every node
   during the very construction of the tree we keep the following three pieces
   of information:

      h : the height of the node
      s : the size of the tree rooted at this node
```

```
    b : the balance information
        (height of the left-subtree - height of the right-subtree)
    isB: whether the tree is balanced

  This information is hidden from the user and is used internally only
  for the height and isBalanced functions.

  The code of the constructors, destructors and all other functions
  may have to be changed because of this -- pretty much the entire
  implementation changes with only the names remaining the same.
*)

abstype 'a bintree2 =
        Empty |
        Node of {nv:'a, h:int, s:int, b:int, isB:bool} *
                'a bintree2 * 'a bintree2
with
        exception Empty_binary_tree

        fun height Empty = 0
          | height (Node(N, Lst, Rst)) = #h N

        fun size Empty = 0
          | size (Node(N, Lst, Rst)) = #s N

        fun isBalanced Empty = true
```

```
  | isBalanced (Node(N, Lst, Rst)) = #isB N

fun mktree0 () = Empty

fun mktrees2 (n, TL, TR) =
    let val (hL, sL, isBL) = (height(TL), size(TL), isBalanced(TL))
        val (hR, sR, isBR) = (height(TR), size(TR), isBalanced(TR))
        val balinfo = sL-sR
    in  Node ({nv = n, h = 1+Int.max(hL, hR), s=1+sL+sR, b=balinfo,
               isB=(abs(balinfo)<=1) andalso isBL andalso isBR}, TL, TR)
    end

fun isEmpty Empty = true
  | isEmpty _     = false

fun subtrees Empty = raise Empty_binary_tree
  | subtrees (Node(N, Lst, Rst)) = (Lst, Rst)

fun root Empty = raise Empty_binary_tree
  | root (Node(N, _, _)) = #nv N

fun leftsubtree Empty = raise Empty_binary_tree
  | leftsubtree (Node(_, Lst, _)) = Lst

fun rightsubtree Empty = raise Empty_binary_tree
  | rightsubtree (Node(_, _, Rst)) =  Rst
```

```
(* Here is a simplistic definition of preorder traversal *)
      fun preorder1 Empty = nil
        | preorder1 (Node(N, Lst, Rst)) =
          [(#nv N)] @ preorder1 (Lst) @ preorder1 (Rst)

(* The above definition though correct is inefficient because it has
   complexity closer to n^2 since the append function itself is linear in
   the length of the list. We would like an algorithm that is linear in the
   number of nodes of the tree. So here is a new one, which uses an iterative
   auxiliary function that stores the preorder traversal of the right subtree
   and then gradually attaches the preorder traversal of the left subtree
   and finally attaches the root as the head of the list.
*)

      local fun pre (Empty, Llist)  = Llist
              | pre (Node (N, Lst, Rst), Llist) =
                let val Mlist = pre (Rst, Llist)
                    val Nlist = pre (Lst, Mlist)
                in  (#nv N)::Nlist
                end
      in fun preorder2 T = pre (T, [])
      end
```

```
        val preorder = preorder2

(* A map function for binary trees *)

        fun BTmap f =
            let fun BTM Empty = Empty
                  | BTM (Node(N, Lst, Rst)) =
                    Node ({nv = f(#nv  N), s= (#s N), h= (#h N), b=(#b N), isB =
                          BTM (Lst), BTM (Rst))
            in  BTM
            end
end (* with abstype *)
```

Lecture 11: Signatures, Structures & Functors

Friday 19 Aug 2011

Towards Modularity

Consider an abstract data type for which there are not only several different implementations but all the implementations are useful simultaneously.

Example 11.1 *Floating point arithmetic for 32-bit precision as well as for 64-bit.*

Example 11.2 *Various implementations of data types such as binary trees, balanced binary trees, binary search trees etc.*

The Module Facility in SML

The module facility of ML lifts the concepts of type, value and function to a higher level in an analogous fashion

$$\begin{aligned} \textit{signature} &\leftrightarrow \textit{type} \\ \textit{structure} &\leftrightarrow \textit{value} \\ \textit{functor} &\leftrightarrow \textit{function} \end{aligned}$$

Modularity

1. Modularity is essential for large programs
2. Modularity may be used along with hiding of information to provide fine-grained visibility required of a package.
3. Many different implementations for the same package specification may be provided at the same time.
4. Modularity in ML is essentially algebraically defined
5. Each module consists of a **signature** and a **structure**
6. If a **structure** is defined without a signature, then ML *infers* a default signature for the structure without hiding any data or definitions.
7. A signature may be used to define the visibility required of a structure

qu-sig.sml

```
signature Q =

sig
    type 'a que
    exception Qerror
    val emptyq : 'a que
    val nullq  : 'a que -> bool
    val enqueue: 'a que * 'a -> 'a que
    val dequeue: 'a que  -> 'a que
    val qhd    : 'a que -> 'a
end;

(*

In a specification we are concerned with certain behavioural properties
of the object that a module defines. These behavioural properties
pertain to an abstract view of the objects in the specification and
their governing properties.

          DEFINING EQUATIONS

Let us define the state of a queue as the sequence of elements in the
queue. Assume a user of this module performs the
following sequence of operations starting from the emptyq.
```

```
q0 = emptyq;
q1 = enqueue (q0, a1);
q2 = enqueue (q1, a2);
q3 = dequeue (q2);
q4 = enqueue (q3, a3);
q5 = dequeue (q4);
q6 = enqueue (q5, a4);
q7 = enqueue (q6, a5);
q8 = dequeue (q7);                                        (1)
```

At the end of this sequence of operations the queue q8 consists of the sequence of elements <a4, a5>. If we were to abbreviate the "emptyq", "enqueue" and "dequeue" operations respectively by "<>", "e" and "d", this sequence of operations may be regarded as a form of algebraic simplification as follows.

```
   d (e (e (d (e (d (e (e (<>, a1), a2)), a3)),a4),a5))   (1)
                  ----
=  d (e (e (d (e (e (d (e (<>, a1), a2)), a3)),a4),a5))   [7]
                    ------------
=  d (e (e (d (e (e (<>, a2)), a3)),a4),a5))              [6]
            -----
=  d (e (e (e (d (e (<>, a2)), a3)),a4),a5))              [7]
                 ------------
=  d (e (e (e (<>, a3)),a4),a5))                          [6]
```

```
    ----
=   e (d (e (e (<>, a3)),a4),a5)                    [7]
      ----
=   e (e (d (e (<>, a3)),a4),a5)                    [7]
         ------------
=   e (e (<>,a4),a5))                               [6]
```

In other words the sequence of operations (1) may be regarded as being equivalent to the sequence (2), in terms of the net effect on the queue.

```
q9 = emptyq;
q10 = enqueue (q9, a4);
q11 = enqueue (q10, a5)                                (2)
```

At the end of this sequence of operations the queue q consists of the sequence of elements <a4, a5>. We may regard this state as having been obtained by using the equations 6 and 7 to reduce the value of the queue to a "normal form" by algebraic simplification.

The last two equations enable us to view the state of any queue as possessing a normal form expressed only in terms of "emptyq" and a sequence of "enqueue" operations on "emptyq" (with no occurrence of "dequeue" occurring anywhere in the normal form).

Any implementation of this specification must satisfy the above

equations in order to be considered correct. Considering the level of detail that there could be in an implementation, this is often very tedious or cumbersome. However, this seems to be the only way.

DIGRESSION:
The notion of a "normal form" is very pervasive in mathematics; for example every polynomial of degree n in one variable x is written as
(a) a sum of terms written in decreasing order of their degrees,
(b) each term is a product of a coefficient and x raised to a certain power,
(c) no two terms in the representation have the same degree.
Alternatively one could choose other representations -- for instance as a prodeuct of n factors of the form (x-ci)
END OF DIGRESSION
*)

Defining Properties/Equations for Q

For every $q : 'a\ que$ and every $x : 'a$,

0. $nullq(emptyq)$
1. $not\ nullq(enqueue(q, x))$
2. $qhd(emptyq) = Qerror$
3. $qhd(enqueue(q, x)) = x$ if $nullq(q)$
4. $qhd(enqueue(q, x)) = qhd(q)$ if $not(nullq(q))$
5. $dequeue(emptyq) = Qerror$
6. $dequeue(enqueue(q, x)) = emptyq$ if $nullq(q)$
7. $dequeue(enqueue(q, x)) = enqueue(dequeue(q), x)$ if $not(nullq(q))$

We refer to these defining properties/equations as axioms of the queue datatype. These axioms define the behaviour of the queue datatype in much the same way that the group axioms define the class of all groups in mathematics and the monoid axioms define the class of all monoids. The class of monoids contains the class of all groups since every group is a monoid and satisfies all the monoid axioms.

How do these axioms define the "behaviour" of queues? More generally, does every datatype have a set of defining axioms? More particularly, in what way does the behaviour of a queue differ from that of a stack?

To answer some of the above questions, we first define the signature and the axioms of the stack datatype. We do this in a manner analogously to what we have defined for queues.

11.1.1. The Stack Datatype

```
signature S =
sig
   type 'a stk
   exception Serror
   val emptys : 'a stk
   val nulls  : 'a stk -> bool
   val push   : 'a stk * 'a -> 'a stk
   val pop    : 'a stk -> 'a stk
   val top    : 'a stk -> 'a
end
```

Notice that the operations of the stack datatype defined above bear a 1-1 correspondence with the operations of the queue datatype. The correspondence is shown below.

$$
\begin{array}{lcl}
'a\ que & \leftrightarrow & 'a\ stk \\
Qerror & \leftrightarrow & Serror \\
emptyq & \leftrightarrow & emptys \\
nullq & \leftrightarrow & nulls \\
enqueue & \leftrightarrow & push \\
dequeue & \leftrightarrow & pop \\
qhd & \leftrightarrow & top
\end{array}
$$

What about the defining equations of the stack datatype? Well here they are and they are indeed analogous to those of the queue. The correspondence in this case is marked by the numbering of the axioms.

For every $s : 'a\ stk$ and every $x : 'a$,

0. $nulls(emptys)$
1. $not\ nulls(push(s, x))$
2. $top(emptys) = Serror$
3. $top(push(s, x)) = x$

5. $pop(emptys) = Serror$
6. $pop(push(s, x)) = s$

Notice that even though we have tried to maintain the analogy between queues and stacks, they invariably do have different axioms and properties. For example the identity 3 in the case of stacks is unconditional

whereas in the case of queues it requires the corresponding argument q to be empty. Identity 4 in the case of queues is necessary and conditional, but is entirely redundant in the case of stacks (though the analogous identity does hold). In a similar manner identity 6 for queues is again conditional but is unconditional in the case of stacks. As in the case 4, identity 7 is redundant for stacks though necessary for queues.

One obvious question that arises is, "Are these axioms correct?" That is, are all properties derivable from these axioms necessarily true? Another important question is "Are these axioms sufficient to prove all properties of stacks?" These questions are called *soundness* and *completeness* respectively. A third obvious question is "How does one think up such axioms?" The answer to this last question is "programmer intuition" and we will leave it at that. A fourth obvious question could be, "If the above axioms are sound and complete, is there a different set of axioms which is also sound and complete?" The answer to the last question is, "Yes, there could be a different set of sound and complete axioms".

The signature of the stack defines a collection of all stack expressions which have the type `'a stk` for any type `'a` of elements. Given a type `'a`, notice that the only ways of obtaining objects of the type `'a stk` are by composing the operations `emptys`, `push` and `pop`. In effect we may define a language of `'a stk`-*expressions* (ranged over by the meta-variable *se*) by the following BNF.

$$se ::= \texttt{emptys} \mid \texttt{push}\ (se, x) \mid \texttt{pop}\ (se) \tag{2}$$

Notice that the BNF (2) allows stack expressions such as `pop (emptys)`, `pop (pop (emptys))`, `push (pop (emptys)`, x) which do not necessarily yield stacks. The language therefore allows a much larger class of expressions than is actually feasible to describe various kinds of stacks.

Our intuition about stacks (a similar analogy holds for queues as well) tells us that the `pop` operation undoes a `push` operation, and more importantly any stack expression that is made up of a number of `push` and `pop` operations equals a stack in which the `pop` operations and some `push` operations cancel

each other resulting in a stack that is either empty or a non-empty stack obtained by a sequence of pushes on an empty stack. We may therefore define a set of *standard* or *normal* form of expressions (ranged over by the meta-variable *nse*) which describes feasible stacks by the following BNF.

$$nse ::= \texttt{emptys} \mid \texttt{push}\,(nse, x) \tag{3}$$

The BNF (3) reflects the above intuition about stack operations which result in stacks. We call the expressions generated by BNF (3) *normal stack expressions*.

Notice first of all that every *normal stack expression* is also an `'a stk`-*expressions*. Hence the language generated by the BNF (3) is a sub-language of the one generated by the BNF (2). We may also prove that following lemma.

Lemma 11.3 *Every stack expression defined by the BNF (2) which does not yield Serror at any stage, may be reduced to a normal stack expression.*

Proof: By induction on the structure of stack expressions. The proof requires the use of the identity 6 to eliminate all occurrences of `pop` in any stack expression which does not yield *Serror*. We leave the details of the proof to the interested reader. QED

Notice that the normal form is unique.

Lemma 11.4 *Every stack expression defined by the BNF (2) which does not yield Serror at any stage, may be reduced to a* unique *normal stack expression.*

qu1-str.sml

```
structure Q1:Q =

struct

    type 'a que = 'a list;

    exception Qerror;

    val emptyq = [];

    fun nullq ([])    = true
    |   nullq (_::_) = false
    ;

    fun enqueue (q, x) = q @ [x];

    (* enqueue takes time linear in the length of q  *)

    fun dequeue (x::q) = q
    |   dequeue []     = raise Qerror
    ;

    fun qhd (x::q) = x
    |   qhd []     = raise Qerror
    ;
```

```
    (* dequeue and qhd are constant time operations *)

end;

(*
```

Note that the (pre-defined) list data type has the following operations:

```
        []   -- denotes the empty list (also called "nil").
        null -- determines whether a list is empty.
        ::   -- denotes the operation "cons" which prepends a list with
                an element to yield a new list.
        @    -- denotes the operation of "append"-ing one list to another.
        hd   -- which yields the first element of a non-empty list
        tl   -- which yields the rest of the list except its head.
```

Clearly hd and tl are defined only for non-empty lists.

The list data type in turn has a normal form expressible only in terms of "nil" and the "cons" operations. Every list L satisfies the following DATA INVARIANT

0. L = [] or L = hd (L) :: tl(L)

The append operation satisfies the following properties for all lists

L, M.

1. [] @ M = M
2. (h::t) @ M = h :: (t @ M)

There is of course another property of append viz.

3. L @ [] = L

But this property can be shown from properties 1. and 2. by induction on the length of the list L.

*)

qu2-str.sml

```
structure Q2:Q =
struct
    datatype 'a que = emptyq | enqueue of 'a que * 'a
    exception Qerror;

    (* enqueue is a constant time operation *)

    fun nullq emptyq = true
      | nullq _      = false

    fun qhd emptyq                  = raise Qerror
      | qhd (enqueue (emptyq, h)) = h
      | qhd (enqueue (q, l))       = qhd (q)

    fun dequeue emptyq                  = raise Qerror
      | dequeue (enqueue (emptyq, h)) = emptyq
      | dequeue (enqueue (q, l))       = enqueue (dequeue q, l)

    (* dequeue is linear in the length of q *)
end
```

qu3-str.sml

```
structure Q3:Q =

struct

    datatype 'a que = Queue of ('a list * 'a list)

    val emptyq = Queue ([], [])

    fun nullq (Queue ([], [])) = true
      | nullq (_)      = false


    fun reverse (L) =
        let fun rev ([], M) = M
              | rev (x::xs, M) = rev (xs, x::M)
        in
        rev (L, [])
        end
    ;


    fun norm (Queue ([], tails)) = Queue (reverse (tails), [])
    |   norm (q)                 = q
    ;
```

```
    fun enqueue (Queue (heads, tails), x) = norm (Queue (heads, x::tails));

    exception Qerror;

    fun dequeue (Queue (x::heads, tails)) = norm (Queue (heads, tails))
    |   dequeue (Queue ([], _))       = raise Qerror
    ;

    fun qhd (Queue (x::heads, tails)) = x
    |   qhd (Queue ([], _))             = raise Qerror
    ;

    (* Clearly the amortized cost of enqueue-ing and dequeue-ing is less than
       linear in this representation "most of the time".
    *)

end;

(*
This example shows a peculiar representation of queues and the concept
of information hiding. The function "norm" is special to this
particular representation and therefore should not be visible to the
user of the queue. If this implementation is correct and satisfies all
the properties of the specification then one could use
representation-hiding and the hiding of purely representation-dependent
operations like norm to switch between the two implementations without
```

affecting any user of this module.

Any user of the Queue module only requires to know the specification and use only those functions, procedures and operations which are visible via the specification. She would have no use for either issues that properly pertain to representation nor with issues concerning the implementation of the algorithms in the module.

*)

Exercise 11.1

1. *Run through the implementation for the example sequence given in qu-sig.sml given in the specification and satisfy yourself that this implementation is indeed correct.*
2. *Prove that each operation on queues in each of the structures correctly implements the specification.*
3. *Suppose we defined the notion of queues bounded by a certain size, say n. In what way are the behavioural properties of such queues different from those of the the unbounded queues defined here?*
4. *Give a complete set of DATA INVARIANT properties for bounded size queues.*

Signatures

1. A signature (c.f. type) specifies the interface to one or more structures (c.f. values).
2. The different implementations of structures Q1, Q2 and Q3 all implement a common signature.
3. Structures may be *constrained* to signatures by mapping the names of structures to the names of signatures.
   (a) the names of the types should match
   (b) the names of functions and values should also match

qu-sig-str.sml

```
use "qu1-str.sml";
use "qu2-str.sml";
use "qu3-str.sml";
use "qu-sig.sml";

(* Instead if declaring "Q1:Q", "Q2:Q" and "Q3:Q" which bind the structures
   to the given signature "Q", within the structure one could have left the
   structure name unqualified and later bound it as follows
*)

(* Transparent Constraints -- all functions/data not present in the
   signature are hidden.
*)
 structure Q1conc: Q = Q1;
 structure Q2conc: Q = Q2;
 structure Q3conc: Q = Q3; (* norm is a hidden function *)

(* Opaque Constraints -- underlying representation is also hidden *)
 structure Q1abs:> Q = Q1;
 structure Q2abs:> Q = Q2;
 structure Q3abs:> Q = Q3;

(* Try the following and see the difference *)
open Q3conc;
open Q3abs;
```

One of the problems with the BNF (2) is that it permitted the construction of stack expressions which would have yielded the exception *Serror*. As a result we had to state the lemma 11.3 in such a way that reduction to the unique normal form BNF (3) was guaranteed only for stack expressions which did not at at any stage raise the exception *Serror*.

The problems with this formulation are many as noted below:

1. (Unique) normal forms are guaranteed only for a subset of stack expressions, thus leaving out a large number syntactically valid expressions to be essentially undefined.
2. The formulation is not very pleasing because it is not general enough.
3. An exception is different from stack expression een theoretically.

A more algebraically elegant formulation which addresses the above problems could be obtained by taking inspiration from the implementation in "qu2-str.sml" wherein a datatype was defined with `emptyq` and `enqueue` being the constructors of the datatype and `dequeue` is defined as a function which cancels out appropriate occurrences of the `enqueue` constructor. An analogous implementation for stacks would have had the following datatype definition

```
datatype 'a stk = emptys | push of 'a stk * 'a
```

and of course a corresponding function definition for the `pop` operation. Notice that this datatype definition is really no different from the language of normal stack expressions.

But now what we could do is we could simply eliminate the exception *Serror* and instead include a 0-ary

constructor `serr` in the data-type definition to represent an "error stack". This has the advantage we could close the language of stack expressions so that now every stack expression would not just be syntactically valid, it would also be well-defined.

We thus define the language of *extended stack expressions* (ranged over by the meta-variable *xse*) by the following BNF

$$xse ::= \texttt{emptys} \mid \texttt{serr} \mid \texttt{push}\ (xse, x) \mid \texttt{pop}(xse) \tag{4}$$

How does the constructor `serr` differ in behaviour and properties from the constructor `emptys`. Very simply put, every stack operation like `push` or `pop` when applied to `serr` would yield `serr`. And of course any pop operation on `emptys` would also yield `serr`. We may now present a richer and more comprehensive set of equations and properties for this new type of stack. But before that we need to evaluate the other consequences of what we are doing now.

1. By changing the exception to a valid stack expression we even obtain a different signature. We call the new signature `S'`.
2. The functions `top` and `nulls` when applied to `serr` should still yield an exception. Alernatively we could change the results of applications of these functions to yield an option data type. To reflect these changes we call these functions `top'` and `nulls'` respectively.

```
signature S' =
sig
   type 'a stk
   val emptys : 'a stk
   val serr   : 'a stk
   val nulls'  : 'a stk -> bool option
   val push   : 'a stk * 'a -> 'a stk
   val pop    : 'a stk -> 'a stk
   val top'    : 'a stk -> 'a option
end
```

Returning to the elegance of dealing with `serr` as just another stack expression we have the following properties and equational identities.

For every s : *'a stk* and every x : *'a*,

$$
\begin{array}{lll}
0. & nulls'(emptys) = true & \\
0'. & nulls'(serr) = NONE & \\
1. & nulls'(push(s,x)) = false & \text{if } s \neq serr \\
2. & top(emptys) = NONE & \\
2'. & top(serr) = NONE & \\
3. & top(push(s,x)) = SOME\ x & \\
 & & \\
5'. & pop(emptys) = serr & \\
5''. & pop(serr) = serr & \\
6. & pop(push(s,x)) = s & \\
6'. & push(serr,x) = serr &
\end{array}
$$

With these equations we also have a language of extended normal form expressions defined as follows and ranged over by the meta-variable *nxse*.

$$nxse ::= \texttt{emptys} \mid \texttt{serr} \mid \texttt{push}\,(nxse, x) \tag{5}$$

We now have the following lemmata (compare these with lemma 11.3 and lemma lem-unique-normal-form respectively).

Lemma 11.5 *Every stack expression defined by the BNF (4) may be reduced to a normal stack expression.*

Lemma 11.6 *Every stack expression defined by the BNF (4) may be reduced to a* unique *extended normal stack expression.*

As we stated before by making these changes we have changed the signature of the module and all aspects of the data type. An analogous change may be made int he signature and structure of queues too.

qu2′-sig-str-sml

```
(* In this signature and structure we replace the exception Qerror by a
   a 0-ary constructor qerr. This allows all 'a que-expressions to be closed
   under the queue operations. Hence we have the following extra
   identities on queues

   dequeue (emptyq) = qerr
   enqueue (qerr, x) = qerr
   dequeue (qerr) = qerr

   But it raises the additional question of what to do about operations nullq
   and qhd which are supposed to yield respectively a boolean value and an
   'a value?

   The obvious answer to these questions is to either define some other new
   exceptions or use the option datatype. We try using the option datatype

*)

signature Q' =

sig
   type 'a que
   (* exception Qerror *)
   val qerr   : 'a que
   val emptyq : 'a que
```

```
    val nullq' : 'a que -> bool option
    val enqueue: 'a que * 'a -> 'a que
    val dequeue: 'a que  -> 'a que
    val qhd'   : 'a que -> 'a option
end;

structure Q2': Q' =
struct
    datatype 'a que = emptyq | qerr | enq of 'a que * 'a
    (* exception Qerror; *)

    (* enqueue is a constant time operation *)

    fun nullq' emptyq = SOME true
      | nullq' qerr   = NONE
      | nullq' _      = SOME false

    fun qhd' emptyq                 = NONE
      | qhd' qerr                   = NONE
      | qhd' (enq (emptyq, h))      = SOME h
      | qhd' (enq (q, l))           = qhd' (q)

    fun normalise emptyq                = emptyq
      | normalise qerr                  = qerr
      | normalise (enq (emptyq, l))     =  enq (emptyq, l)
      | normalise (enq (qerr, _))       = qerr
```

```
    | normalise (enq (q, l))      = (* enqueue (normalise q, l)) *)
      let val nq = normalise q
      in  case nq of
            emptyq => enq (nq, l)
          | qerr   => qerr
          | enq (_, _) => enq (nq, l)
      end

  fun enqueue (emptyq, l)           = enq (emptyq, l)
    | enqueue (qerr, _)             = qerr
    | enqueue (q, l)                = normalise (enq (q, l))

  fun dequeue emptyq              = qerr
    | dequeue qerr                = qerr
    | dequeue (enq (emptyq, h)) = emptyq
    | dequeue (enq (q, l))        = enq ((dequeue q), l)

  (* dequeue is linear in the length of q *)
end

(* testing *)

open Q2';

val e = enqueue;
```

```
val d = dequeue;

val q1 = e (d (emptyq), 1)

val q7 = e (e (d (e (d (e (e (emptyq, 1), 2)), 3)),4),5)

val h7 = valOf (qhd' q7)

val q8 = d (e (e (d (e (d (e (e (emptyq, 1), 2)), 3)),4),5))

val h8 = qhd' q8

val q9 = d (e (e (d (e (d (e (e (d (emptyq), 1), 2)), 3)),4),5))

val h9 = qhd' q9

val q10 = d (e (e (d (e (d (e (d (d (e (emptyq, 1))), 2)), 3)),4),5))

val h10 = qhd' q10
```

Functors

A **functor** is a structure that takes other structures as parameters and yields a new structure

1. A functor can be applied to argument structures to yield a new structure
2. A functor can be applied only to structures that match certain signature constraints.
3. Functors may be used to test existing structures or to create new structures.
4. Functors may also be used to express **generic** algorithms

bstree-module.sml

```
(* This is a module implementation of BINARY SEARCH TREES *)


(* A Binary Search Tree  or a BST is a tree with nodes labelled by elements
   TOTALLY ordered by an IRREFLEXIVE–TRANSITIVE relation "lt" such that
   for any node y in the tree,

   o   lt (x, y) holds for all nodes x in the LEFT subtree of y,  and
   o   lt (y, z) holds for all nodes z in the RIGHT subtree of y.

*)

(* can one use "bintree.sml" ? *)

(* How does one specialize a binary tree to a BST? *)

(* We assume that we are dealing with BSTs in the following examples. *)

(* Searching in a BST: checking for a node labelled x *)

(* Rather than answering these questions now, we simply follow Ullman's
   construction and address these questions later.
*)

signature TOTALORDER =
```

```
sig
    eqtype eltype
    val lt : eltype * eltype -> bool
end (* sig *);

structure INTLAB: TOTALORDER =
struct
   type eltype = int;
   val  lt = op<
end;

structure STRINGLAB: TOTALORDER =

struct

   type eltype = string;
   (* Lexicographic ordering '<' on strings *)

   fun lexlt (s, t) =
       let val Ls = explode (s);
        val Lt = explode (t);
        fun lstlexlt (_, []) = false
        |   lstlexlt ([], (b:char)::M) = true
        |   lstlexlt (a::L, b::M) =
                  if (a < b) then true
                  else if (a = b) then lstlexlt (L, M)
```

```
                    else false
            ;
        in      lstlexlt (Ls, Lt)
        end
    ;

    val lt = lexlt;

end (* struct *);

functor MakeBST (Lt: TOTALORDER):
    sig
(*        open Lt; *)

(* The use of this "open" is an error in Ullman's book. At least it does not
seem to be allowed in version 109.32. We have instead replaced all occurrences
of "eltype" in the signature by "Lt.eltype" and then it seems to work fine.
*)

        type 'a bintree;
        exception Empty_tree;
        val create: Lt.eltype bintree;
        val lookup: Lt.eltype * Lt.eltype bintree -> bool;
        val insert: Lt.eltype * Lt.eltype bintree -> Lt.eltype bintree;
        val deletemin: Lt.eltype bintree -> Lt.eltype * Lt.eltype bintree;
        val delete: Lt.eltype * Lt.eltype bintree -> Lt.eltype bintree
```

```
    end
=
    struct
        open Lt;
        datatype 'a bintree =
                 Empty |
                 Node of 'a * 'a bintree * 'a bintree
        ;

        val create = Empty;

        fun lookup (x, Empty) = false
        |   lookup (x, Node (y, left, right)) =
             if x=y then true
             else if lt (x, y) then lookup (x, left)
             else lookup (x, right)
        ;

        (* Insert an element into a BST *)

        fun insert  (x, Empty) = Node (x, Empty, Empty)
        |   insert (x, T as Node (y, left, right)) =
             if x=y then T                                    (* do nothing *)
             else if lt (x, y) then Node (y, insert (x, left), right)
             else Node (y, left, insert (x, right))
        ;
```

```
(* Delete (if it is there) from a BST *)

(* Deletion requires the following function which can delete the
   smallest element from a tree; Note that the smallest element in a
   BST is the leftmost leaf-node in the tree (if it exists, otherwise
   the root). The function deletemin should also return the value of the
   smallest element to enable tree reordering for deletion.
*)

exception Empty_tree;

fun deletemin (Empty) = raise Empty_tree
|   deletemin (Node (y, Empty, right)) = (y, right)
|   deletemin (Node (y, left, right)) =
        let val (z, L) = deletemin (left)
        in  (z, Node (y, L, right))
        end
;

(* NOTE that deletemin does not require the comparison function lt as a
   parameter.
*)

fun delete (x, Empty) = Empty
|   delete (x, Node (y, left, right)) =
```

```
            if x=y then
               if left = Empty then right
               else if right = Empty then left
               else (* extract the smallest element from the right subtree
                       and make it the root of the new tree
                    *)
                    let val (z, R) = deletemin (right)
                    in  Node (z, left, R)
                    end
            else if lt (x, y) then Node (y, delete (x, left), right)
                 else    Node (y, left, delete (x, right))
      ;

   end (* struct *);

(* Now we may apply the functor MakeBST to STRINGLAB to obtain a new structure
   StringBST which defines "binary search trees labelled by strings ordered by
   the lexicographic total ordering.
*)

structure StringBST = MakeBST (STRINGLAB); (* applying the functor *)

structure IntBST = MakeBST (INTLAB);
```

Lecture 12: Example: Tautology Checking

Tuesday 23 Aug 2011

Arguments

A typical informally stated argument might go as follows:

If prices rise, then the poor and the salaried class will be unhappy.

If taxes are increased then the businessmen will be unhappy.

If the poor and the salaried class or the businessmen are unhappy, the Government will not be re-elected.

Inflation will rise if Government expenditure exceeds its revenue.

Government expenditure will exceed its revenue unless taxes are increased or the Government resorts to deficit financing or takes a loan from the IMF to cover the deficit.

If the Government resorts to deficit financing then inflation will rise.

If inflation rises, the prices will also rise.

The Government will get reelected.

Therefore the Government will take a loan from the IMF.

Arguments: 2

A typical informally stated argument might go as follows:

If prices rise, then the poor and the salaried class will be unhappy.

If taxes are increased then the businessmen will be unhappy.

If the poor and the salaried class are unhappy or the businessmen are unhappy, the Government will not be re-elected.

Inflation will rise if Government expenditure exceeds its revenue.

Government expenditure will exceed its revenue unless taxes are increased or the Government resorts to deficit financing or takes a loan from the IMF to cover the deficit.

If the Government resorts to deficit financing then inflation will rise.

If inflation rises, the prices will also rise.

The Government will get reelected.

Therefore the Government will take a loan from the IMF.

Arguments in natural language

It turns out that the correctness or otherwise of most arguments depends entirely on the "shapes" of the formulae concerned rather than their intrinsic meaning. Take the previous argument. Suppose we uniformly replace the various atoms by say sentences from nursery rhymes as the following table shows:

Prices rise	Mary has a little lamb
The poor and ...	Little Bo-Peep loses her sheep
Taxes are increased	Jack and Jill go up the hill
The businessmen will be unhappy	Humpty-Dumpty sits on the wall
The Government will get re-elected	I am little teapot
Inflation will rise	Little Jack Horner sits in a corner
Government expenditure ... revenue	The boy stands on the burning deck
The Government resorts ...	Wee Willie Winkie runs through the town
The Government takes a loan ... deficit	Eensy Weensy spider climbs up the water spout

Then we get the following ridiculous sounding argument

If Mary has a little lamb, then Little Bo-Peep loses her sheep.
If Jack and Jill go up the hill, then Humpty-Dumpty sits on a wall.
If Little Bo-Peep loses her sheep or Humpty-Dumpty sits on a wall, then Little Miss Muffet sits on a tuffet.
Little Jack Horner sits in a corner if the boy stands on the burning deck.
The boy stands on the burning deck unless Jack and Jill go up the hill or Wee Willie Winkie runs through the town or Eensy Weensy spider climbs up the water spout
If Wee Willie Winkie runs through the town then Little Jack Horner sits in a corner
If Little Jack Horner sits in a corner then Mary has a little lamb.
Little Miss Muffet sits on a tuffet.

Therefore Eensy Weensy spider climbs up the water spout.

But if the original argument is logically valid then so is the new one, since logical validity depends entirely on the so called connectives that make up the propositions and their effect on truth values.

Validity & Falsification

The validity of such arguments involves showing that the conclusion is a logical consequence of the hypotheses that precede it. The following alternatives exist:

1. Using a truth table.
2. Using theorem 12.2
3. Using one of the parts of theorem 12.3.

If the argument is not valid, then a falsifying assignment needs to be also given.

Translation into propositional Logic

But even after translating the argument into a suitable propositional logic form, it would be quite impossible to verify the validity of the argument by truth table, since the number of “atomic” propositions could be very large.

Atoms in Argument


```
val rise = ATOM "Prices rise";
val pandsun = ATOM "The poor and ... will be unhappy";
val taxes = ATOM "Taxes are increased";
val busun = ATOM "The businessmen will be unhappy";
val reelect = ATOM "The Government will be re-elected";
val inflation = ATOM "Inflation will rise";
val exceeds = ATOM "Government expenditure ... revenue";
val deffin = ATOM "The Government resorts ...";
val imf = ATOM "The Govt. takes a loan ... deficit";
```

In this case the the truth table would have $2^9 = 512$ rows. Further, the truth table will use *all* the atoms, even the irrelevant ones.

The Representation

```
datatype Prop = ATOM of string
              | NOT of Prop
              | AND of Prop * Prop
              | OR of Prop * Prop
              | IMP of Prop * Prop
              | EQL of Prop * Prop
```

Propositional Rendering


```
val hyp1 = IMP (rise, pandsun);
val hyp2 = IMP (taxes, busun);
val hyp3 = IMP (OR (pandsun, busun), NOT(reelect));
val hyp4 = IMP (exceeds, inflation);
val hyp5 = IMP (exceeds, NOT(OR(taxes, OR(deffin, imf))));
val hyp6 = IMP (deffin, inflation);
val hyp7 = IMP (inflation, rise);
val hyp8 = reelect;
val conc1 = imf;
val H = [hyp1, hyp2, ..., hyp8];
val Arg1 = (H, conc1);
```

The Strategy

We need to either show that

$$Arg1 = IMP\ (bigAND\ H, conc1)$$

is a tautology or can be falsified. Using theorem 12.2 for validity

```
val bigAND = leftReduce (AND);
fun Valid ((H, P):Argument) =
    if null (H) then tautology (P)
    else tautology (IMP (bigAND (H), P))
```

and for falsification we use

```
fun falsifyArg ((H, P): Argument) =
    if null (H) then falsify (cnf(P))
    else falsify (cnf (IMP (bigAND (H), P)))
```

Checking Tautology

Checking for tautology crucially involves finding falsifying truth assignments for at last one of the conjuncts in the CNF of the argument.

```
fun tautology2 (P) =
    let val Q = cnf (P);
            val LL = falsify (Q)
    in      if null (LL) then (true, [])
            else (false, LL)
    end
```

Computing the CNF

1. Rewrite implications and equivalences
2. Convert to NNF by pushing negation inward
3. Distribute OR over AND

cnflistlist.sml

```
(*======================== THE SIGNATURE PropLogic =====================
signature PropLogic =

sig
    exception Atom_exception
    datatype Prop =
        ATOM of string      |
        NOT of Prop         |
        AND of Prop * Prop  |
        OR of Prop * Prop   |
        IMP of Prop * Prop  |
        EQL of Prop * Prop
    type Argument = Prop list * Prop
    val show    : Prop -> unit
    val showArg : Argument -> unit
    val falsifyArg : Argument -> Prop list list
    val Valid   : Argument -> bool * Prop list list
end;

(* Propositional formulas *)

======================== THE STRUCTURE PL ====================== *)
(* structure PL:PropLogic = *)
structure PL =  (* This is for debugging purposes only *)
struct

     datatype Prop =
         ATOM of string      |
         NOT of Prop         |
         AND of Prop * Prop  |
         OR of Prop * Prop   |
         IMP of Prop * Prop  |
         EQL of Prop * Prop
     ;

     (* ---------------- Propositions to CNFs --------------------- *)

     exception Atom_exception;
     fun newatom (s) = if s = "" then raise Atom_exception
                       else (ATOM s);
```

```
fun drawChar (c, n) =
            if n>0 then (print(str(c)); drawChar(c, (n-1)))
            else ();
fun show (P) =
    let  fun drawTabs (n) = drawChar (#"\t", n);
         fun showTreeTabs (ATOM a, n) = (drawTabs (n);
                                         print (a);
                                         print("\n")
                                         )
         |   showTreeTabs (NOT (P), n) = (drawTabs(n); print ("NOT");
                                          showTreeTabs (P, n+1)
                                          )
         |   showTreeTabs (AND (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("AND\n");
                          showTreeTabs (Q, n+1)
                          )
         |   showTreeTabs (OR (P, Q), n)  =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("OR\n");
                          showTreeTabs (Q, n+1)
                          )

         |   showTreeTabs (IMP (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("IMPLIES\n");
                          showTreeTabs (Q, n+1)
                          )
         |   showTreeTabs (EQL (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("IFF\n");
                          showTreeTabs (Q, n+1)
                          )
         ;
    in   (print ("\n"); showTreeTabs(P, 0); print ("\n"))

    end
;


(* The function below evaluates a formula given a truth assignment.
```

```
   The truth assignment is given as a list of atoms that are true
   (all other atoms are false).
*)


fun lookup (x:Prop, []) = false
|   lookup (x, h::L)    = (x = h) orelse lookup (x, L)
;

fun eval (ATOM a, L)     = lookup (ATOM a, L)
|   eval (NOT (P), L)    = if eval (P, L) then false else true
|   eval (AND (P, Q), L) = eval (P, L) andalso eval (Q, L)
|   eval (OR (P, Q), L)  = eval (P, L) orelse eval (Q, L)
|   eval (IMP (P, Q), L) = eval (OR (NOT (P), Q), L)
|   eval (EQL (P, Q), L) = (eval (P, L) = eval (Q, L))
;

(* We could also write a tautology checker with out using truth
   assignments by first converting everything into a normal form.
*)

(* First rewrite implications and equivalences *)

fun rewrite (ATOM a)     = ATOM a
|   rewrite (NOT (P))    = NOT (rewrite (P))
|   rewrite (AND (P, Q)) = AND (rewrite(P), rewrite(Q))
|   rewrite (OR (P, Q))  = OR (rewrite(P), rewrite(Q))
|   rewrite (IMP (P, Q)) = OR (NOT (rewrite(P)), rewrite(Q))
|   rewrite (EQL (P, Q)) = rewrite (AND (IMP(P, Q), IMP (Q, P)))
;

(* Convert all formulas not containing IMP or EQL into Negation Normal
   Form.
*)

fun nnf (ATOM a)           = ATOM a
|   nnf (NOT (ATOM a))     = NOT (ATOM a)
|   nnf (NOT (NOT (P)))    = nnf (P)
|   nnf (AND (P, Q))       = AND (nnf(P), nnf(Q))
|   nnf (NOT (AND (P, Q))) = nnf (OR (NOT (P), NOT (Q)))
|   nnf (OR (P, Q))        = OR (nnf(P), nnf(Q))
```

```
|   nnf (NOT (OR (P, Q)))  = nnf (AND (NOT (P), NOT (Q)))
;

(* Distribute OR over AND to get a NNF into CNF *)

fun distOR (P, AND (Q, R)) = AND (distOR (P, Q), distOR (P, R))
|   distOR (AND (Q, R), P) = AND (distOR (Q, P), distOR (R, P))
|   distOR (P, Q)          = OR (P, Q)

(* Now the CNF can be easily computed *)

fun conj_of_disj (AND (P, Q)) = AND (conj_of_disj (P), conj_of_disj (Q))
|   conj_of_disj (OR (P, Q))  = distOR (conj_of_disj (P), conj_of_disj (Q))
|   conj_of_disj (P)          = P
;

fun cnf (P) = conj_of_disj (nnf (rewrite (P)));

(* --------------- Propositions to CNFs ends --------------------- *)

(* --------------- CNFs to lists of lists of literals ------------ *)

(* Convert a clause into a list of literals *)
fun flattenOR (OR (A, B)) = (flattenOR A) @ (flattenOR B)
  | flattenOR C           = [C] (* assuming C is a literal *)

(* Convert a CNF into a list of lists of clauses *)
fun flattenAND (AND (Q, R)) = (flattenAND Q) @ (flattenAND R)
  | flattenAND (P)          = [flattenOR P] (* assuming P is a clause *)

(* Sort the litListList using some ordering and remove duplicates while sorting *)

(* Define an ordering litLess on literals: *)

fun litLess (ATOM (a), ATOM (b))       = a < b (* lexicographic *)
  | litLess (NOT(ATOM a), NOT(ATOM b)) = a < b
    (* every negative literal is smaller than every positive literal *)
  | litLess (NOT(ATOM a), ATOM (b))    = true
  | litLess (ATOM (a), NOT(ATOM b))    = false

(* Extend the ordering to lists of literals *)
```

```
fun clauseLess ([], [])         = false
  | clauseLess ([], _ )         = true
  | clauseLess (_,  [])         = false
  | clauseLess (h1::T1, h2::T2) =
      (litLess (h1, h2)) orelse
      ((h1=h2) andalso clauseLess (T1, T2))

(* Define mergeSortRD to remove duplicates as sorting proceeds *)

fun mergeSortRD R [] = []
  | mergeSortRD R [h] = [h]
  | mergeSortRD R L = (* can't split a list unless it has > 1 element *)
  let fun split []  = ([], [])
      | split [h] = ([h], [])
      | split (h1::h2::t) =
             let val (left, right) = split t;
              in (h1::left, h2::right)
             end;
      val (left, right) = split L;
      fun mergeRD (R, [], []) = []
        | mergeRD (R, [], L2) = L2
        | mergeRD (R, L1, []) = L1
        | mergeRD (R, (L1 as h1::t1), (L2 as h2::t2)) =
                if h1=h2 then mergeRD (R, t1, L2) (* remove a copy *)
                else if R(h1, h2) then h1::(mergeRD (R, t1, L2))
                else h2::(mergeRD (R, L1, t2));
      val sortedLeft = mergeSortRD R left;
      val sortedRight = mergeSortRD R right;
  in  mergeRD (R, sortedLeft, sortedRight)
  end;

(* Now sort the list of lists of literals removing duplicates *)

fun sortRD LL = (* First sort each clause and then the list of clauses *)
    let val sortedClauses = map (mergeSortRD litLess) LL
    in  mergeSortRD clauseLess sortedClauses
    end;

(* Putting everything together *)
```

```
    fun prop2listlist P =  sortRD (flattenAND (cnf P))
end (* struct *);

open PL;

(* Testing prop2listlist ===================================
- val       god = ATOM "There is a God";
val god = ATOM "There is a God" : Prop
- val       oscient = ATOM "God is omniscient";
val       opotent = ATOM "God is omnipotent";
val       evil = ATOM "There is Evil";
val       know = ATOM "God knows there is Evil";
val       prevent = ATOM "God prevents Evil";

val hy1 = IMP (god, AND (oscient, opotent));
val hy2 = IMP (oscient, know);
val hy3 = IMP (opotent, prevent);
val hy4 = evil;
val conc = NOT (god);

val oscient = ATOM "God is omniscient" : Prop
- GC #0.0.0.1.7.187:    (1 ms)
val opotent = ATOM "God is omnipotent" : Prop
- val evil = ATOM "There is Evil" : Prop
- val know = ATOM "God knows there is Evil" : Prop
- val prevent = ATOM "God prevents Evil" : Prop
- - val hy1 = IMP (ATOM "There is a God",AND (ATOM #,ATOM #)) : Prop
- val hy2 = IMP (ATOM "God is omniscient",ATOM "God knows there is Evil") : Prop
- val hy3 = IMP (ATOM "God is omnipotent",ATOM "God prevents Evil") : Prop
- val hy4 = ATOM "There is Evil" : Prop
- val conc = NOT (ATOM "There is a God") : Prop
- - prop2listlist hy1;
val it =
  [[NOT (ATOM "There is a God"),ATOM "God is omnipotent"],
   [NOT (ATOM "There is a God"),ATOM "God is omniscient"]] : Prop list list
- prop2listlist hy2;
val it = [[NOT (ATOM "God is omniscient"),ATOM "God knows there is Evil"]]
  : Prop list list
- val andhyp = AND (hy1, AND (hy2, AND(hy3, hy4)));
val andhyp = AND (IMP (ATOM #,AND #),AND (IMP #,AND #)) : Prop
- prop2listlist andhyp;
```

```
val it =
  [[NOT (ATOM "God is omnipotent"),ATOM "God prevents Evil"],
   [NOT (ATOM "God is omniscient"),ATOM "God knows there is Evil"],
   [NOT (ATOM "There is a God"),ATOM "God is omnipotent"],
   [NOT (ATOM "There is a God"),ATOM "God is omniscient"],
   [ATOM "There is Evil"]] : Prop list list
- val a = ATOM "a";
val a = ATOM "a" : Prop
- val b = ATOM "b";
val b = ATOM "b" : Prop
- val c = ATOM "c";
val c = ATOM "c" : Prop
- val one = IMP (a, OR (a, a));
val one = IMP (ATOM "a",OR (ATOM #,ATOM #)) : Prop
- val two = EQL(b, OR(b, NOT(b)));
val two = EQL (ATOM "b",OR (ATOM #,NOT #)) : Prop
- val three = EQL(AND(a, a), OR(NOT(a), OR(a, NOT(a))));
GC #0.0.0.1.8.221:    (1 ms)
val three = EQL (AND (ATOM #,ATOM #),OR (NOT #,OR #)) : Prop
- val p2ll = prop2listlist;
val p2ll = fn : Prop -> Prop list list
- p2ll one;
val it = [[NOT (ATOM "a"),ATOM "a"]] : Prop list list
- p2ll two;
val it = [[NOT (ATOM "b"),ATOM "b"],[ATOM "b"]] : Prop list list
- p2ll three;
val it = [[NOT (ATOM "a"),ATOM "a"],[ATOM "a"]] : Prop list list
- p2ll (OR(one, OR(two, three)));
val it =
  [[NOT (ATOM "a"),NOT (ATOM "b"),ATOM "a",ATOM "b"],
   [NOT (ATOM "a"),ATOM "a",ATOM "b"]] : Prop list list
-
========================================================== *)
```

Falsifying CNF

1. Suffices to find a falsification of at least one conjunct
2. A conjunct in the CNF can be false iff all the disjuncts in it are false.
3. A disjunct is false iff it does not contain a "complementary pair".

Assume the CNF is $Q \equiv \bigwedge_{i=1}^{m} D_i$ where each $D_i \equiv \bigvee_{j=1}^{n_i} L_{ij}$ where the literals of $D_i = P_i \cup N_i$ where P_i is the set of positive literals (atoms) and N_i consists of the atoms appearing as negative literals.

Then D_i is false iff $P_i \cap N_i = \emptyset$.

tautology1.sml

```
signature PropLogic =

sig
    exception Atom_exception
    datatype Prop =
        ATOM of string      |
        NOT of Prop         |
        AND of Prop * Prop  |
        OR of Prop * Prop   |
        IMP of Prop * Prop  |
        EQL of Prop * Prop
    type Argument = Prop list * Prop
    val show     : Prop -> unit
    val showArg  : Argument -> unit
    val falsifyArg : Argument -> Prop list list
    val Valid    : Argument -> bool * Prop list list
end;

(* Propositional formulas *)

structure PL:PropLogic =
(* structure PL = *) (* This is for debugging purposes only *)
struct

     datatype Prop =
         ATOM of string      |
         NOT of Prop         |
         AND of Prop * Prop  |
         OR of Prop * Prop   |
         IMP of Prop * Prop  |
         EQL of Prop * Prop
     ;

     exception Atom_exception;
     fun newatom (s) = if s = "" then raise Atom_exception
                       else (ATOM s);
     fun drawChar (c, n) =
                  if n>0 then (print(str(c)); drawChar(c, (n-1)))
                  else ();
```

```
fun show (P) =
    let  fun drawTabs (n) = drawChar (#"\t", n);
         fun showTreeTabs (ATOM a, n) = (drawTabs (n);
                                         print (a);
                                         print("\n")
                                         )
         |   showTreeTabs (NOT (P), n) = (drawTabs(n); print ("NOT");
                                          showTreeTabs (P, n+1)
                                          )
         |   showTreeTabs (AND (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("AND\n");
                          showTreeTabs (Q, n+1)
                          )
         |   showTreeTabs (OR (P, Q), n)  =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("OR\n");
                          showTreeTabs (Q, n+1)
                          )

         |   showTreeTabs (IMP (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("IMPLIES\n");
                          showTreeTabs (Q, n+1)
                          )
         |   showTreeTabs (EQL (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("IFF\n");
                          showTreeTabs (Q, n+1)
                          )
         ;
    in   (print ("\n"); showTreeTabs(P, 0); print ("\n"))

    end
;

(* The function below evaluates a formula given a truth assignment.
   The truth assignment is given as a list of atoms that are assigned
   "true" (implicitly all other atoms are assume dto have been
   assigned "false").
```

```
*)


fun lookup (x:Prop, []) = false
|   lookup (x, h::L) =
      if (x = h) then true
      else lookup (x, L)
;

fun eval (ATOM a, L) = lookup (ATOM a, L)
|   eval (NOT (P), L) = if eval (P, L) then false else true
|   eval (AND (P, Q), L) = eval (P, L) andalso eval (Q, L)
|   eval (OR (P, Q), L) = eval (P, L) orelse eval (Q, L)
|   eval (IMP (P, Q), L) = eval (OR (NOT (P), Q), L)
|   eval (EQL (P, Q), L) = (eval (P, L) = eval (Q, L))
;

(* We  first convert every proposition into a normal form.
*)

(* First rewrite implications and equivalences *)

fun rewrite (ATOM a)         = ATOM a
|   rewrite (NOT (P))        = NOT (rewrite (P))
|   rewrite (AND (P, Q)) = AND (rewrite(P), rewrite(Q))
|   rewrite (OR (P, Q))  = OR (rewrite(P), rewrite(Q))
|   rewrite (IMP (P, Q)) = OR (NOT (rewrite(P)), rewrite(Q))
|   rewrite (EQL (P, Q)) = rewrite (AND (IMP(P, Q), IMP (Q, P)))
;

(* Convert all formulas not containing IMP or EQL into Negation Normal
   Form.
*)

fun nnf (ATOM a)       = ATOM a
|   nnf (NOT (ATOM a))         = NOT (ATOM a)
|   nnf (NOT (NOT (P)))        = nnf (P)
|   nnf (AND (P, Q))           = AND (nnf(P), nnf(Q))
|   nnf (NOT (AND (P, Q))) = nnf (OR (NOT (P), NOT (Q)))
|   nnf (OR (P, Q))            = OR (nnf(P), nnf(Q))
|   nnf (NOT (OR (P, Q)))  = nnf (AND (NOT (P), NOT (Q)))
```

```
;

(* Distribute OR over AND to get a NNF into CNF *)

fun distOR (P, AND (Q, R)) = AND (distOR (P, Q), distOR (P, R))
|   distOR (AND (Q, R), P) = AND (distOR (Q, P), distOR (R, P))
|   distOR (P, Q)      = OR (P, Q)

(* Now the CNF can be easily computed *)

fun conj_of_disj (AND (P, Q)) = AND (conj_of_disj (P), conj_of_disj (Q))
|   conj_of_disj (OR (P, Q))  = distOR (conj_of_disj (P), conj_of_disj (Q))
|   conj_of_disj (P)              = P
;

fun cnf (P) = conj_of_disj (nnf (rewrite (P)));

(* A proposition in CNF is a tautology
            iff
   Every conjunct is a tautology
            iff
   Every disjunct in every conjunct contains both positive and negative
   literals of at least one atom

   So we construct the list of all the positive and negative atoms in every
   disjunct to check whether the lists are all equal. We need a binary
   function on lists to determine whether two lists are disjoint
*)

fun isPresent (a, []) = false
|   isPresent (a, b::L) = (a = b) orelse isPresent (a, L)
;

fun disjoint ([], M) = true
|   disjoint (L, []) = true
|   disjoint (L as a::LL, M as b::MM)=
        not(isPresent (a, M)) andalso
        not(isPresent(b, L))  andalso
        disjoint (LL, MM)
;
```

```
(* ABHISHEK : Defining a total ordering on atoms (lexicographic
   ordering on underlying strings), and extending it to a list of atoms.
*)

exception notAtom;

fun atomLess (a, b) = case (a, b) of
        (ATOM(x), ATOM(y)) => x<y
      | (_,_)              => raise notAtom;

fun listLess (a, b) = case (a, b) of
        (_, [])            => false
      | ([], _)            => true
      | (x::lx, y::ly)     => if atomLess(x,y) then true
                              else if atomLess(y,x) then false
                              else listLess(lx,ly);

(* ABHISHEK : Once we have a list of falsifiers, we would want to remove
   any duplication, firstly of atoms within a falsifier, and secondly of
   falsifiers themselves.

   In order to do this, we maintain all lists in some sorted order.
   Instead of sorting a list with a possibly large number of duplicates,
   we check for duplicates while inserting, and omit insertion if a
   previous instance is detected.

*)

fun merge less ([],l2) = l2
|   merge less (l1,[]) = l1
|   merge less (x::l1,y::l2) =
    if less(x,y) then x::merge less (l1,y::l2)
    else if less(y,x) then y::merge less (x::l1,l2)
    else merge less (x::l1,l2);

(* ABHISHEK : Claim is that if all lists are built through the above
   function, then there is no need to sort or remove duplicates.

   Hence all '@' operations have been replaced by merge.
*)
```

```
exception not_CNF;

fun positives (ATOM a)       = [ATOM a]
|   positives (NOT (ATOM _))= []
|   positives (OR (P, Q))    = merge atomLess (positives (P), positives (Q))
|   positives (P)            = raise not_CNF
;

fun negatives (ATOM _)       = []
|   negatives (NOT (ATOM a))= [ATOM a]
|   negatives (OR (P, Q))    = merge atomLess (negatives (P), negatives (Q))
|   negatives (P)            = raise not_CNF
;

(* Check whether a formula in CNF is a tautology *)

fun taut (AND (P, Q)) = taut (P)  andalso taut (Q)
|   taut (P) = (* if it is not a conjunction then it must be a disjunct *)
       not (disjoint (positives (P), negatives (P)))
;

fun tautology1 (P) =
    let val Q = cnf (P)
    in  taut (Q)
    end
;

(* The main problem with the above is that it checks whether a given
   proposition is a tautology, but whenever it is not, it does not yield
   a falsifying truth assignment. We rectify this problem below.

*)

(*
   Firstly, as in the case of the function lookup, we will assume a truth
   assignment is a list of atoms which are assigned the truth value "true"
   and that any atom that is not present in the list has been assigned
   "false".

   Assume Q is a proposition in CNF. Then it is only necessary to list out
   all the lists of truth assignments that can falsify Q.
```

```
   Suppose Q is in CNF, but not necessarily a tautology. Further let

      Q = AND (D1, ..., Dn)

   where each Di is a disjunction of literals. Each Di = Pi + Ni where
   Pi and Ni are the lists of atoms denoting the positive and negative
   literals respectively.

   Q would be "falsified" if at least one of the Di can be made false. Di
   can be made false only if it does not contain a "complementary pair",
   i.e. there exists no atom a such that both a and ~a occur in Di. Hence
   for Di to be falsified it is necessary that the lists Pi and Ni are
   disjoint (if there is no atom common to Pi and Ni, there is no
   "complementary pair" in Di.

   Since Di is a disjunction of literals, it can be falsified only by
   assigning every literal in Di the value "false". This can be done only
   by assigning all the atoms in Pi the value "false" and all the atoms
   in Ni the value "true".

   In other words, if Pi and Ni are disjoint, then Ni is a truth
   assignment which falsifies the proposition Q. We refer to Ni as a
   FALSIFIER of Q.

   Therefore the FALSIFIERS of Q are exactly the list of negative atoms
   of each disjunct which does not contain a complementary pair. By
   checking each disjunct in Q we may list out ALL the possible
   FALSIFIERS of Q.

   If Q has no FALSIFIER then no disjunct Di can be made false i.e. every
   disjunct does indeed have a compementary pair. We may then conclude
   that Q is a tautology.
*)

(* The following function assumes Q is in CNF and outputs a list of list
   of atoms that can falsify Q. If this list of list of atoms is empty then
   clearly Q is a tautology.
*)

fun falsify (Q) =
```

```
    let fun list_Falsifiers (AND (A, B)) =
                  merge listLess (list_Falsifiers (A),list_Falsifiers (B))
        |   list_Falsifiers (A) = (* Assume A is a disjunct of literals *)
                      let val PLA = positives (A) (* no uniq required *)
                          val NLA = negatives (A)
                      in  if disjoint (PLA, NLA) then [NLA]
                          else []
                      end
    in list_Falsifiers (Q)
    end
;

fun tautology2 (P) =
    let val Q = cnf (P);
            val LL = falsify (Q)
    in      if null (LL) then (true, [])
            else (false, LL)
    end
;

val tautology = tautology2;

(*
   We may use the tautology checker to prove various arguments
   logically valid or logically invalid. An argument consists
   of a set of propositions called the "hypotheses" and a (single)
   proposition called the "conclusion". Loosely speaking, an argument
   is similar to a theorem of mathematics. The argument
   is logically valid if the conclusion is a logical consequence of
   of the hypotheses. More accurately, if in every truth assignment
   which makes all the hypotheses true, the conclusion is also invariably
   true then the argument is logically valid.

   Symbolically if H1, ..., Hm are propositions and C is another
   proposition then the argument ({H1, ..., Hm}, C) is logically
   valid (equivalently, C is a logical consequence of {H1, ..., Hm})
   if and only if the (compound) proposition

        (H1 /\ ... /\ Hm) => C

   is a tautology.
```

```
   An argument which is not logically valid is logically
   invalid. In particular if there exists a truth assignment under which
   all the hypotheses are true but the conclusion is false, then the
   argument is invalid.

   Any argument is trivially logically valid if there is no truth
   assignment under which every hypothesis is true. In other words,
   if the set of hypotheses is an inconsistent set then regardless
   of what the conclusion is, the argument is always logically valid.
   The set of hypotheses {H1, ..., Hm} is "inconsistent" if and only if
   (H1 /\ ... /\ Hm) is a "contradiction" (it is false for every truth
   assignment).


*)
type Argument = Prop list * Prop;

fun showArg (A: Argument) =
    let fun printArg (A:Argument as ([], c)) =
            (drawChar (#"-", 80); print("\n");
             show (c); print ("\n\n")
            )
        |   printArg (A:Argument as (p::plist, c)) =
             (show (p); print ("\n");
              printArg (plist, c)
             )
    in (print ("\n\n"); printArg (A))
    end
;

fun leftReduce (F) =
    let exception emptylist;
          fun lr ([]) = raise emptylist
          |   lr ([a]) = a
          |   lr (a::L) = F (a, lr (L))
    in  lr
    end
;

val bigAND = leftReduce (AND);
```

```
    fun Valid ((L, P):Argument) =
        if null (L) then tautology (P)
        else tautology (IMP (bigAND (L), P))
    ;

    fun falsifyArg ((L, P): Argument) =
        if null (L) then falsify (cnf(P))
        else falsify (cnf (IMP (bigAND (L), P)))
    ;

end (* struct *);

(* open PL; *)
```

Logical Consequence: 1

Definition 12.1 *A proposition $\phi \in \mathcal{P}_0$ is called a* **logical consequence** *of a set $\Gamma \subseteq \mathcal{P}_0$ of formulas (denoted $\Gamma \models \phi$) if any truth assignment that satisfies all formulas of Γ also satisfies ϕ.*

- When $\Gamma = \emptyset$ then logical consequence reduces to logical validity.
- $\models \phi$ denotes that ϕ is logically valid.
- $\Gamma \not\models \phi$ denotes that ϕ is not a logical consequence of Γ.
- $\not\models \phi$ denotes that ϕ is logically invalid.

Logical Consequence: 2

Theorem 12.2 *Let* $\Gamma = \{\phi_i \mid 1 \leq i \leq n\}$ *be a finite set of propositions, and let* ψ *be any proposition. Then* $\Gamma \models \psi$ *if and only if* $((\ldots((\phi_1 \wedge \phi_2) \wedge \phi_3) \wedge \ldots \wedge \phi_n) \rightarrow \psi)$ *is a tautology.*

Other Theorems

Theorem 12.3 *Let* $\Gamma = \{\phi_i \mid 1 \leq i \leq n\}$ *be a finite set of propositions, and let* ψ *be any proposition. Then*

1. $\Gamma \models \psi$ *if and only if* $\models \phi_1 \rightarrow (\phi_2 \rightarrow \cdots (\phi_n \rightarrow \psi) \cdots)$
2. $\Gamma \models \psi$ *if and only if* $((\ldots((\phi_1 \wedge \phi_2) \wedge \phi_3) \wedge \ldots \wedge \phi_n) \wedge \neg\psi)$ *is a contradiction.*

Corollary 12.4 *A formula* ϕ *is a tautology iff* $\neg\phi$ *is a contradiction (unsatisfiable).*

■

Lecture 13: Example: Tautology Checking (Contd)

Wednesday 24 Aug 2011

tautology1.sml – the full source

```
signature PropLogic =

sig
    exception Atom_exception
    datatype Prop =
        ATOM of string      |
        NOT of Prop         |
        AND of Prop * Prop  |
        OR of Prop * Prop   |
        IMP of Prop * Prop  |
        EQL of Prop * Prop
    type Argument = Prop list * Prop
    val show     : Prop -> unit
    val showArg  : Argument -> unit
    val falsifyArg : Argument -> Prop list list
    val Valid    : Argument -> bool * Prop list list
end;

(* Propositional formulas *)

structure PL:PropLogic =
(* structure PL = *) (* This is for debugging purposes only *)
struct

     datatype Prop =
         ATOM of string      |
         NOT of Prop         |
         AND of Prop * Prop  |
         OR of Prop * Prop   |
         IMP of Prop * Prop  |
         EQL of Prop * Prop
     ;

     exception Atom_exception;
     fun newatom (s) = if s = "" then raise Atom_exception
                       else (ATOM s);
     fun drawChar (c, n) =
                  if n>0 then (print(str(c)); drawChar(c, (n-1)))
                  else ();
```

```
fun show (P) =
    let  fun drawTabs (n) = drawChar (#"\t", n);
         fun showTreeTabs (ATOM a, n) = (drawTabs (n);
                                         print (a);
                                         print("\n")
                                         )
         |   showTreeTabs (NOT (P), n) = (drawTabs(n); print ("NOT");
                                          showTreeTabs (P, n+1)
                                          )
         |   showTreeTabs (AND (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("AND\n");
                          showTreeTabs (Q, n+1)
                          )
         |   showTreeTabs (OR (P, Q), n)  =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("OR\n");
                          showTreeTabs (Q, n+1)
                          )

         |   showTreeTabs (IMP (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("IMPLIES\n");
                          showTreeTabs (Q, n+1)
                          )
         |   showTreeTabs (EQL (P, Q), n) =
                          (showTreeTabs (P, n+1);
                          drawTabs (n); print("IFF\n");
                          showTreeTabs (Q, n+1)
                          )
         ;
    in   (print ("\n"); showTreeTabs(P, 0); print ("\n"))

    end
;

(* The function below evaluates a formula given a truth assignment.
   The truth assignment is given as a list of atoms that are assigned
   "true" (implicitly all other atoms are assume dto have been
   assigned "false").
```

```
*)


fun lookup (x:Prop, []) = false
|   lookup (x, h::L) =
      if (x = h) then true
      else lookup (x, L)
;

fun eval (ATOM a, L) = lookup (ATOM a, L)
|   eval (NOT (P), L) = if eval (P, L) then false else true
|   eval (AND (P, Q), L) = eval (P, L) andalso eval (Q, L)
|   eval (OR (P, Q), L) = eval (P, L) orelse eval (Q, L)
|   eval (IMP (P, Q), L) = eval (OR (NOT (P), Q), L)
|   eval (EQL (P, Q), L) = (eval (P, L) = eval (Q, L))
;

(* We  first convert every proposition into a normal form.
*)

(* First rewrite implications and equivalences *)

fun rewrite (ATOM a)         = ATOM a
|   rewrite (NOT (P))        = NOT (rewrite (P))
|   rewrite (AND (P, Q)) = AND (rewrite(P), rewrite(Q))
|   rewrite (OR (P, Q))  = OR (rewrite(P), rewrite(Q))
|   rewrite (IMP (P, Q)) = OR (NOT (rewrite(P)), rewrite(Q))
|   rewrite (EQL (P, Q)) = rewrite (AND (IMP(P, Q), IMP (Q, P)))
;

(* Convert all formulas not containing IMP or EQL into Negation Normal
   Form.
*)

fun nnf (ATOM a)       = ATOM a
|   nnf (NOT (ATOM a))         = NOT (ATOM a)
|   nnf (NOT (NOT (P)))        = nnf (P)
|   nnf (AND (P, Q))           = AND (nnf(P), nnf(Q))
|   nnf (NOT (AND (P, Q))) = nnf (OR (NOT (P), NOT (Q)))
|   nnf (OR (P, Q))            = OR (nnf(P), nnf(Q))
|   nnf (NOT (OR (P, Q)))  = nnf (AND (NOT (P), NOT (Q)))
```

```
;

(* Distribute OR over AND to get a NNF into CNF *)

fun distOR (P, AND (Q, R)) = AND (distOR (P, Q), distOR (P, R))
|   distOR (AND (Q, R), P) = AND (distOR (Q, P), distOR (R, P))
|   distOR (P, Q)      = OR (P, Q)

(* Now the CNF can be easily computed *)

fun conj_of_disj (AND (P, Q)) = AND (conj_of_disj (P), conj_of_disj (Q))
|   conj_of_disj (OR (P, Q))  = distOR (conj_of_disj (P), conj_of_disj (Q))
|   conj_of_disj (P)              = P
;

fun cnf (P) = conj_of_disj (nnf (rewrite (P)));

(* A proposition in CNF is a tautology
            iff
   Every conjunct is a tautology
            iff
   Every disjunct in every conjunct contains both positive and negative
   literals of at least one atom

   So we construct the list of all the positive and negative atoms in every
   disjunct to check whether the lists are all equal. We need a binary
   function on lists to determine whether two lists are disjoint
*)

fun isPresent (a, []) = false
|   isPresent (a, b::L) = (a = b) orelse isPresent (a, L)
;

fun disjoint ([], M) = true
|   disjoint (L, []) = true
|   disjoint (L as a::LL, M as b::MM)=
        not(isPresent (a, M)) andalso
        not(isPresent(b, L))  andalso
        disjoint (LL, MM)
;
```

```
(* ABHISHEK : Defining a total ordering on atoms (lexicographic
   ordering on underlying strings), and extending it to a list of atoms.
*)

exception notAtom;

fun atomLess (a, b) = case (a, b) of
        (ATOM(x), ATOM(y)) => x<y
      | (_,_)              => raise notAtom;

fun listLess (a, b) = case (a, b) of
        (_, [])            => false
      | ([], _)            => true
      | (x::lx, y::ly)     => if atomLess(x,y) then true
                              else if atomLess(y,x) then false
                              else listLess(lx,ly);

(* ABHISHEK : Once we have a list of falsifiers, we would want to remove
   any duplication, firstly of atoms within a falsifier, and secondly of
   falsifiers themselves.

   In order to do this, we maintain all lists in some sorted order.
   Instead of sorting a list with a possibly large number of duplicates,
   we check for duplicates while inserting, and omit insertion if a
   previous instance is detected.

*)

fun merge less ([],l2) = l2
|   merge less (l1,[]) = l1
|   merge less (x::l1,y::l2) =
    if less(x,y) then x::merge less (l1,y::l2)
    else if less(y,x) then y::merge less (x::l1,l2)
    else merge less (x::l1,l2);

(* ABHISHEK : Claim is that if all lists are built through the above
   function, then there is no need to sort or remove duplicates.

   Hence all '@' operations have been replaced by merge.
*)
```

```
exception not_CNF;

fun positives (ATOM a)       = [ATOM a]
|   positives (NOT (ATOM _))= []
|   positives (OR (P, Q))    = merge atomLess (positives (P), positives (Q))
|   positives (P)            = raise not_CNF
;

fun negatives (ATOM _)       = []
|   negatives (NOT (ATOM a))= [ATOM a]
|   negatives (OR (P, Q))    = merge atomLess (negatives (P), negatives (Q))
|   negatives (P)            = raise not_CNF
;

(* Check whether a formula in CNF is a tautology *)

fun taut (AND (P, Q)) = taut (P)  andalso taut (Q)
|   taut (P) = (* if it is not a conjunction then it must be a disjunct *)
       not (disjoint (positives (P), negatives (P)))
;

fun tautology1 (P) =
    let val Q = cnf (P)
    in  taut (Q)
    end
;

(* The main problem with the above is that it checks whether a given
   proposition is a tautology, but whenever it is not, it does not yield
   a falsifying truth assignment. We rectify this problem below.

*)

(*
   Firstly, as in the case of the function lookup, we will assume a truth
   assignment is a list of atoms which are assigned the truth value "true"
   and that any atom that is not present in the list has been assigned
   "false".

   Assume Q is a proposition in CNF. Then it is only necessary to list out
   all the lists of truth assignments that can falsify Q.
```

```
   Suppose Q is in CNF, but not necessarily a tautology. Further let

      Q = AND (D1, ..., Dn)

   where each Di is a disjunction of literals. Each Di = Pi + Ni where
   Pi and Ni are the lists of atoms denoting the positive and negative
   literals respectively.

   Q would be "falsified" if at least one of the Di can be made false. Di
   can be made false only if it does not contain a "complementary pair",
   i.e. there exists no atom a such that both a and ~a occur in Di. Hence
   for Di to be falsified it is necessary that the lists Pi and Ni are
   disjoint (if there is no atom common to Pi and Ni, there is no
   "complementary pair" in Di.

   Since Di is a disjunction of literals, it can be falsified only by
   assigning every literal in Di the value "false". This can be done only
   by assigning all the atoms in Pi the value "false" and all the atoms
   in Ni the value "true".

   In other words, if Pi and Ni are disjoint, then Ni is a truth
   assignment which falsifies the proposition Q. We refer to Ni as a
   FALSIFIER of Q.

   Therefore the FALSIFIERS of Q are exactly the list of negative atoms
   of each disjunct which does not contain a complementary pair. By
   checking each disjunct in Q we may list out ALL the possible
   FALSIFIERS of Q.

   If Q has no FALSIFIER then no disjunct Di can be made false i.e. every
   disjunct does indeed have a compementary pair. We may then conclude
   that Q is a tautology.
*)

(* The following function assumes Q is in CNF and outputs a list of list
   of atoms that can falsify Q. If this list of list of atoms is empty then
   clearly Q is a tautology.
*)

fun falsify (Q) =
```

```
    let fun list_Falsifiers (AND (A, B)) =
                  merge listLess (list_Falsifiers (A),list_Falsifiers (B))
        |   list_Falsifiers (A) = (* Assume A is a disjunct of literals *)
                      let val PLA = positives (A) (* no uniq required *)
                          val NLA = negatives (A)
                      in  if disjoint (PLA, NLA) then [NLA]
                          else []
                      end
    in list_Falsifiers (Q)
    end
;

fun tautology2 (P) =
    let val Q = cnf (P);
            val LL = falsify (Q)
    in      if null (LL) then (true, [])
            else (false, LL)
    end
;

val tautology = tautology2;

(*
   We may use the tautology checker to prove various arguments
   logically valid or logically invalid. An argument consists
   of a set of propositions called the "hypotheses" and a (single)
   proposition called the "conclusion". Loosely speaking, an argument
   is similar to a theorem of mathematics. The argument
   is logically valid if the conclusion is a logical consequence of
   of the hypotheses. More accurately, if in every truth assignment
   which makes all the hypotheses true, the conclusion is also invariably
   true then the argument is logically valid.

   Symbolically if H1, ..., Hm are propositions and C is another
   proposition then the argument ({H1, ..., Hm}, C) is logically
   valid (equivalently, C is a logical consequence of {H1, ..., Hm})
   if and only if the (compound) proposition

        (H1 /\ ... /\ Hm) => C

   is a tautology.
```

```
   An argument which is not logically valid is logically
   invalid. In particular if there exists a truth assignment under which
   all the hypotheses are true but the conclusion is false, then the
   argument is invalid.

   Any argument is trivially logically valid if there is no truth
   assignment under which every hypothesis is true. In other words,
   if the set of hypotheses is an inconsistent set then regardless
   of what the conclusion is, the argument is always logically valid.
   The set of hypotheses {H1, ..., Hm} is "inconsistent" if and only if
   (H1 /\ ... /\ Hm) is a "contradiction" (it is false for every truth
   assignment).


*)
type Argument = Prop list * Prop;

fun showArg (A: Argument) =
    let fun printArg (A:Argument as ([], c)) =
            (drawChar (#"-", 80); print("\n");
             show (c); print ("\n\n")
            )
        |   printArg (A:Argument as (p::plist, c)) =
             (show (p); print ("\n");
              printArg (plist, c)
             )
    in (print ("\n\n"); printArg (A))
    end
;

fun leftReduce (F) =
    let exception emptylist;
          fun lr ([]) = raise emptylist
          |   lr ([a]) = a
          |   lr (a::L) = F (a, lr (L))
    in  lr
    end
;

val bigAND = leftReduce (AND);
```

```
    fun Valid ((L, P):Argument) =
        if null (L) then tautology (P)
        else tautology (IMP (bigAND (L), P))
    ;

    fun falsifyArg ((L, P): Argument) =
        if null (L) then falsify (cnf(P))
        else falsify (cnf (IMP (bigAND (L), P)))
    ;

end (* struct *);

(* open PL; *)
```

Lecture 14: The Pure Untyped Lambda Calculus: Syntax

Friday 26 Aug 2011

Lecture 15: The Pure Untyped Lambda Calculus: Basics

Tuesday 30 Aug 2011

Pure Untyped λ-Calculus: Syntax

The language Λ of pure untyped λ-terms is the smallest set built up from an infinite set V of *variables*

$$\begin{array}{rcll} L, M, N & ::= & x & \text{Variable} \\ & | & \lambda x[L] & \text{Abstraction} \\ & | & (L\ M) & \text{Application} \end{array}$$

where $x \in V$.

- A *Variable* denotes a possible binding in the external environment.
- An *Abstraction* denotes a function which takes a formal parameter.
- An *Application* denotes the application of a function to an actual parameter.

Free and Bound Variables

Definition 15.1 *For any term N the set of* **free variables** *and the set of all variables are defined by induction on the structure of terms.*

N	$FV(N)$	$Var(N)$
x	$\{x\}$	$\{x\}$
$\lambda x[L]$	$FV(L) - \{x\}$	$Var(L) \cup \{x\}$
$(L\ M)$	$FV(L) \cup FV(M)$	$Var(L) \cup Var(M)$

- *The set of* **bound variables** $BV(N) = Var(N) - FV(N)$.
- *The same variable name may be used with different bindings in a single term (e.g.* $(\lambda x[x]\ \lambda x[(x\ y)])$*)*
- *The brackets "[" and "]" delimit the scope of the bound variable x in the term $\lambda x[L]$.*
- $\Lambda_0 \subseteq \Lambda$ *is the set of λ-terms with no free variables.*

Notational Conventions

To minimize use of brackets unambiguously

1. $\lambda x_1 x_2 \ldots x_m[L]$ denotes $\lambda x_1[\lambda x_2[\ldots \lambda x_m[L] \cdots]]$ i.e. L is the scope of each of the variables $x_1, x_2, \ldots x_m$.
2. $(L_1\ L_2\ \cdots L_m)$ denotes $(\cdots(L_1\ L_2)\ \cdots L_m)$ i.e. application is *left-associative*.

Substitution

Definition 15.2 *For any terms L, M and N and any variable x, the* **substitution** *of the term N for a variable x is defined as follows:*

$$
\begin{array}{lcll}
\{N/x\}x & \equiv & N & \\
\{N/x\}y & \equiv & y & \text{if } y \not\equiv x \\
\{N/x\}\lambda x[L] & \equiv & \lambda x[L] & \\
\{N/x\}\lambda y[L] & \equiv & \lambda y[\{N/x\}L] & \text{if } y \not\equiv x \text{ and } y \notin FV(N) \\
\{N/x\}\lambda y[L] & \equiv & \lambda z[\{N/z\}\{z/x\}L] & \text{if } y \not\equiv x \text{ and } y \in FV(N) \text{ and } z \text{ is 'fresh'} \\
\{N/x\}(L\ M) & \equiv & (\{N/x\}L\ \{N/x\}M) &
\end{array}
$$

- In the above definition it is necessary to ensure that the free variables of N continue to remain free after substitution.
- The phrase "z is 'fresh'" may be taken to mean $z \notin FV(N) \cup Var(L)$.
- z could be fresh even if $z \in BV(N)$

α-equivalence

Definition 15.3 (α-equivalence) $\lambda x[L] \equiv_\alpha \lambda y[\{y/x\}L]$ *provided* $y \notin Var(L)$.

- Here again if $y \in FV(L)$ it must not be captured by a change of bound variables.

Untyped λ-Calculus: Basic β-Reduction

Definition 15.4

- *Any (sub-)term of the form* $(\lambda x[L]\ M)$ *is called a* β*-redex*
- *Basic* β*-reduction is the relation*

$$\boxed{(\lambda x[L]\ M) \to_\beta \{M/x\}L'} \tag{6}$$

where $L' \equiv_\alpha L$.

Untyped λ-Calculus: 1-step β-Reduction

Definition 15.5 *A 1-step β-reduction $\rightarrow_\beta^1$ is the smallest (under the $\subseteq$ ordering) relation such that*

$$\beta_1 \quad \frac{L \rightarrow_\beta M}{L \rightarrow_\beta^1 M}$$

$$\beta_1\mathbf{Abs} \quad \frac{L \rightarrow_\beta^1 M}{\lambda x[L] \rightarrow_\beta^1 \lambda x[M]}$$

$$\beta_1\mathbf{AppL} \quad \frac{L \rightarrow_\beta^1 M}{(L\ N) \rightarrow_\beta^1 (M\ N)}$$

$$\beta_1\mathbf{AppR} \quad \frac{L \rightarrow_\beta^1 M}{(N\ L) \rightarrow_\beta^1 (N\ M)}$$

- $\rightarrow_\beta^1$ is the compatible closure of basic β-reduction to all contexts.
- We will often omit the superscript ^{1} as understood.

Untyped λ-Calculus: β-Reduction

Definition 15.6

- *For all integers $n \geq 0$, n-step β-reduction $\rightarrow_\beta^n$ is defined by induction on 1-step β-reduction*

$$\beta_{\mathbf{n}}\mathbf{Basis}\ \frac{}{L \rightarrow_\beta^0 L} \qquad \beta_{\mathbf{n}}\mathbf{Induction}\ \frac{L \rightarrow_\beta^m M \quad M \rightarrow_\beta^1 N}{L \rightarrow_\beta^{m+1} N}\ (m \geq 0)$$

- *β-reduction $\rightarrow_\beta^*$ is the reflexive-transitive closure of 1-step β-reduction. That is,*

$$\beta_*\ \frac{L \rightarrow_\beta^n M}{L \rightarrow_\beta^* M}\ (n \geq 0)$$

Untyped λ-Calculus: Normalization

Definition 15.7

- *A term is called a β-normal form (β-nf) if it has no β-redexes.*
- *A term is weakly normalising (β-WN) if it reduces to a β-normal form.*
- *A term L is strong normalising (β-SN) if it has no infinite reduction sequence $L \to^1_\beta L_1 \to^1_\beta L_2 \to^1_\beta \cdots$*

Untyped λ-Calculus: Examples

Example 15.8

1. $\mathsf{K} \stackrel{df}{=} \lambda x\ y[x]$, $\mathsf{I} \stackrel{df}{=} \lambda x[x]$, $\mathsf{S} \stackrel{df}{=} \lambda x\ y\ z[((x\ z)\ (y\ z))]$, $\omega \stackrel{df}{=} \lambda x[(x\ x)]$ *are all* β*-nfs.*

2. $\Omega \stackrel{df}{=} (\omega\ \omega)$ *has no* β*-nf. Hence it is neither weakly nor strongly normalising.*

3. $(\mathsf{K}\ (\omega\ \omega))$ *cannot reduce to any normal form because it has no finite reduction sequences. All its reductions are of the form*

$$(\mathsf{K}\ (\omega\ \omega)) \rightarrow^1_\beta (\mathsf{K}\ (\omega\ \omega)) \rightarrow^1_\beta (\mathsf{K}\ (\omega\ \omega)) \rightarrow^1_\beta \cdots$$

or at some point it could transform to

$$(\mathsf{K}\ (\omega\ \omega)) \rightarrow^1_\beta \lambda y[(\omega\ \omega)] \rightarrow^1_\beta \lambda y[(\omega\ \omega)] \rightarrow^1_\beta \cdots$$

4. $((\mathsf{K}\ \omega)\ \Omega)$ *is* $weakly$ *normalising because it can reduce to the normal form* ω *but it is* not $strongly$ *normalising because it also has an infinite reduction sequence*

$$((\mathsf{K}\ \omega)\ \Omega) \rightarrow^1_\beta ((\mathsf{K}\ \omega)\ \Omega) \rightarrow^1_\beta \cdots$$

Examples of Strong Normalization

Example 15.9

1. ((K ω) ω) *is strongly normalising because it reduces to the normal form* ω *in a single step.*
2. *Consider the term* ((S K) K)*. Its reduction sequences go as follows:*

$$((\mathsf{S}\ \mathsf{K})\ \mathsf{K}) \to^1_\beta \lambda z[((\mathsf{K}\ z)\ (\mathsf{K}\ z))] \to^1_\beta \lambda z[z] \equiv \mathsf{I}$$

Lecture 16: Notions of Reduction

Tuesday 06 Sep 2011

Reduction

For any function such as $p = \lambda x[3.x.x + 4.x + 1]$,

$$(p\ 2) = 3.2.2 + 4.2 + 1 = 21$$

However there is something *asymmetric* about the identity, in the sense that while $(p\ 2)$ deterministically produces $3.2.2 + 4.2 + 1$ which in turn simplifies deterministically to 21, it is not possible to deterministically infer that 21 came from $(p\ 2)$. It would be more accurate to refer to this sequence as a *reduction sequence* and capture the asymmetry as follows:

$$(p\ 2) \rightsquigarrow 3.2.2 + 4.2 + 1 \rightsquigarrow 21$$

And yet they are *behaviourally* equivalent and mutually substitutable in all contexts (*referentially transparent*).

1. Reduction (specifically β-reduction) captures this asymmetry.
2. Since reduction produces behaviourally *equal* terms we have the following notion of equality.

Untyped λ-Calculus: β-Equality

Definition 16.1 *β-equality or β-conversion (denoted $=_\beta$) is the smallest equivalence relation containing β-reduction ($\rightarrow_\beta^*$).*

The following are equivalent definitions.

1. $=_\beta$ is the reflexive-symmetric-transitive closure of 1-step β-reduction.
2. $=_\beta$ is the smallest relation defined by the following rules.

$$=_\beta \textbf{ Basis } \frac{L \rightarrow_\beta^* M}{L =_\beta M} \qquad =_\beta \textbf{ Reflexivity } \frac{}{L =_\beta L}$$

$$=_\beta \textbf{ Symmetry } \frac{L =_\beta M}{M =_\beta L} \qquad =_\beta \textbf{ Transitivity } \frac{L =_\beta M,\ M =_\beta N}{L =_\beta N}$$

Compatibility

Definition 16.2 *A binary relation $\rho \subseteq \Lambda \times \Lambda$ is said to be* **compatible** *if $L\ \rho\ M$ implies*

1. *for all variables x, $\lambda x[L]\ \rho\ \lambda x[M]$ and*
2. *for all terms N, $(L\ N)\ \rho\ (M\ N)$ and $(N\ L)\ \rho\ (N\ M)$.*

Example 16.3

1. *$\equiv_\alpha$ is a compatible relation*
2. *$\to_\beta^1$ is by definition a compatible relation.*

Compatibility of Beta-reduction and Beta-Equality

Theorem 16.4 *β-reduction $\rightarrow_\beta^*$ and β-equality $=_\beta$ are both compatible relations.*

□

Proof of theorem 16.4

Proof: $(\rightarrow_\beta^*)$ Assume $L \rightarrow_\beta^* M$. By definition of β-reduction $L \rightarrow_\beta^n M$ for some $n \geq 0$. The proof proceeds by induction on n

Basis. $n = 0$. Then $L \equiv M$ and there is nothing to prove.

Induction Hypothesis (*IH*).

> *The proof holds for all k, $0 \leq k \leq m$ for some $m \geq 0$.*

Induction Step. For $n = m + 1$, let $L \equiv L_0 \rightarrow_\beta^m L_m \rightarrow_\beta^1 M$. Then by the induction hypothesis and the compatibility of $\rightarrow_\beta^1$ we have

	By definition of $\rightarrow_\beta^n$
for all $x \in V$, $\lambda x[L] \rightarrow_\beta^m \lambda x[L_m]$, $\lambda x[L_m] \rightarrow_\beta^1 \lambda x[M]$	$\lambda x[L] \rightarrow_\beta^n \lambda x[M]$,
for all $N \in \Lambda$, $(L\ N) \rightarrow_\beta^m (L_m\ N)$, $(L_m\ N) \rightarrow_\beta^1 (M\ N)$	$(L\ N) \rightarrow_\beta^n (M\ N)$
for all $N \in \Lambda$, $(N\ L) \rightarrow_\beta^m (N\ L_m)$, $(N\ L_m) \rightarrow_\beta^1 (N\ M)$	$(N\ L) \rightarrow_\beta^n (N\ M)$

End $(\rightarrow_\beta^*)$

($=_\beta$) Assume $L =_\beta M$. We proceed by induction on the length of the proof of $L =_\beta M$ using the definition of β-equality.

Basis. $n = 1$. Then either $L \equiv M$ or $L \to_\beta^* M$. The case of reflexivity is trivial and the case of $L \to_\beta^* M$ follows from the previous proof.

Induction Hypothesis (*IH*).

> *For all terms L and M, such that the proof of $L =_\beta M$ requires less than n steps for $n > 1$, the compatibility result holds.*

Induction Step. Suppose the proof requires n steps and the last step is obtained by use of either $=_\beta$ **Symmetry** or $=_\beta$ **Transitivity** on some previous steps.

Case **($=_\beta$ Symmetry).** Then the $(n-1)$-st step proved $M =_\beta L$. By the induction hypothesis and then by applying $=_\beta$ **Symmetry** to each case we get

		By $=_\beta$ **Symmetry**
for all variables x,	$\lambda x[M] =_\beta \lambda x[L]$	$\lambda x[L] =_\beta \lambda x[M]$
for all terms N,	$(M\ N) =_\beta (L\ N)$	$(L\ N) =_\beta (M\ N)$
for all terms N,	$(N\ M) =_\beta (N\ L)$	$(N\ M) =_\beta (N\ L)$

Case **($=_\beta$ Transitivity).** Suppose $L =_\beta M$ was inferred in the n-th step from two previous steps which proved $L =_\beta P$ and $P =_\beta M$ for some term P. Then again by induction hypothesis and then applying $=_\beta$ **Transitivity** we get

			By $=_\beta$ **Transitivity**
for all variables x,	$\lambda x[L] =_\beta \lambda x[P]$,	$\lambda x[P] =_\beta \lambda x[M]$	$\lambda x[L] =_\beta \lambda x[M]$
for all terms N,	$(L\ N) =_\beta (P\ N)$,	$(P\ N) =_\beta (M\ N)$	$(L\ N) =_\beta (M\ N)$
for all terms N,	$(N\ L) =_\beta (N\ P)$,	$(N\ P) =_\beta (N\ M)$	$(N\ L) =_\beta (N\ P)$

End ($=_\beta$)

QED ■

Eta reduction

Given any term M and a variable $x \notin FV(M)$, the syntax allows us to construct the term $\lambda x[(M\ x)]$ such that for every term N we have

$$(\lambda x[(M\ x)]\ N) \to_{\beta}^{1} (M\ N)$$

In other words,

$$\boxed{(\lambda x[(M\ x)]\ N) =_{\beta} (M\ N) \text{ for all terms } N}$$

We say that the two terms $\lambda x[(M\ x)]$ and M are **extensionally** equivalent i.e. they are *syntactically distinct* but there is no way to distinguish between their *behaviours*.

So we define basic η-reduction as the relation

$$\boxed{\lambda x[(L\ x)] \to_{\eta} L \text{ provided } x \notin FV(L)} \tag{7}$$

Eta-Reduction and Eta-Equality

The following notions are then defined similar to the corresponding notions for β-reduction.

- 1-step η-reduction $\to_\eta^1$ is the closure of basic η-reduction to all contexts,
- $\to_\eta^n$ is defined by induction on 1-step η-reduction
- η-reduction $\to_\eta^*$ is the reflexive-transitive closure of 1-step η-reduction.
- the notions of strong and weak η normal forms η-nf.
- the notion of η-equality or η-conversion denoted by $=_\eta$.

Exercise 16.1

1. *Prove that η-reduction and η-equality are both compatible relations.*
2. *Prove that η-reduction is strongly normalising.*
3. *Define basic $\beta\eta$-reduction as the application of either (6) or (7). Now prove that $\rightarrow^1_{\beta\eta'}$, $\rightarrow^*_{\beta\eta}$ and $=_{\beta\eta}$ are all compatible relations.*

Lecture 17: Confluence Definition

Wednesday 07 Sep 2011

Definition 17.1 *For any binary relation ρ on a set A*

1. ρ^* is the least preorder containing ρ and is defined by the following rules.

Definition 17.2 *Let ρ be any relation on terms. ρ has the* **diamond property** *if for all L, M, N,*

$$\begin{array}{ccc} & M & \\ & \rho & \\ L & & \end{array} \Rightarrow \exists P : \quad \begin{array}{ccc} & M & \\ & \rho & \\ & & P \end{array}$$
$$\begin{array}{ccc} & \rho & \\ & N & \end{array} \qquad \begin{array}{ccc} & \rho & \\ & N & \end{array}$$

Definition 17.3 *Let $\longrightarrow$ be a reduction relation, $\longrightarrow^*$ the least preorder containing $\longrightarrow$ and $\overset{*}{\longleftrightarrow}$ the least equivalence relation containing $\longrightarrow^*$. Then*

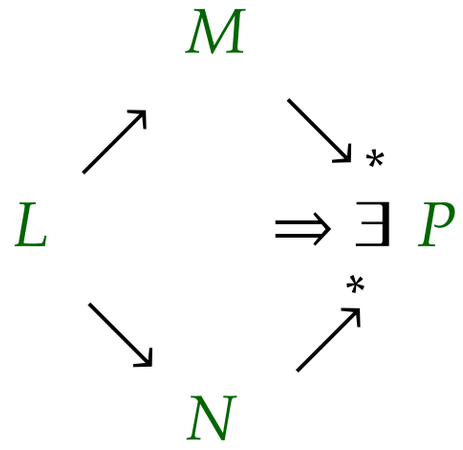
- $\longrightarrow$ *is* **terminating** *iff there is no infinite sequence of the form*

$$L_0 \longrightarrow L_1 \longrightarrow \cdots$$

- $\longrightarrow$ *is* **locally confluent** *if for all L, M, N,*

$$N \longleftarrow L \longrightarrow M \Rightarrow \exists P : N \longrightarrow^* P \ {}^*\!\longleftarrow M$$

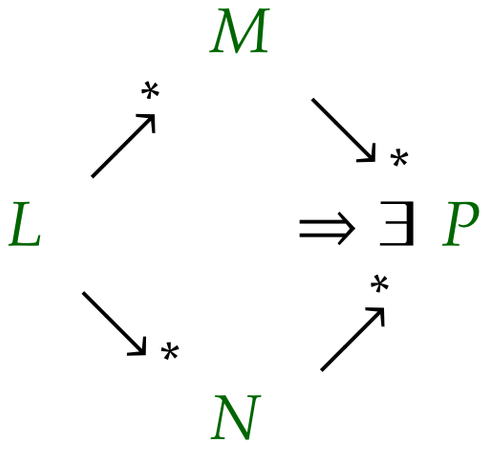
which we denote by



- $\longrightarrow$ *is* **confluent** *if for all* L, M, N,

$$N \,{}^{*}\!\longleftarrow L \longrightarrow^{*} M \Rightarrow \exists P : N \longrightarrow^{*} P \,{}^{*}\!\longleftarrow M$$

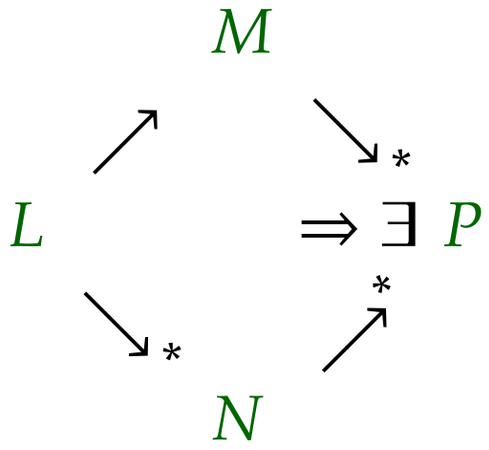
which we denote as



- $\longrightarrow$ *is* **semi-confluent** *if for all* L, M, N,

$$N \longleftarrow L \longrightarrow^{*} M \Rightarrow \exists P : N \longrightarrow^{*} P \,{}^{*}\!\longleftarrow M$$

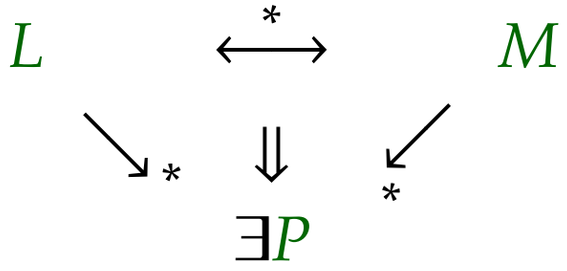
which we denote by



- $\longrightarrow$ *is* **Church-Rosser** *if for all* L, M,

$$L \overset{*}{\longleftrightarrow} M \Rightarrow \exists P : L \longrightarrow^* P \ {}^*\!\longleftarrow M$$

which we denote by



Fact 17.4 *Any confluent relation is also semi-confluent.*

Lemma 17.5

1. $\overset{*}{\longleftrightarrow}$ *is the least equivalence containing* $\longrightarrow$.

2. $\overset{*}{\longleftrightarrow}$ *is the least equivalence containing* $\longrightarrow^*$.

3. $L \overset{*}{\longleftrightarrow} M$ *if and only if there exists a finite sequence* $L \equiv M_0, M_1, \ldots M_m \equiv M$, $m \geq 0$ *such that for each* i, $0 \leq i < m$, $M_i \longrightarrow M_{i+1}$ *or* $M_{i+1} \longrightarrow M_i$. *We represent this fact more succinctly as*

$$L \equiv_\alpha M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_m \equiv_\alpha M \tag{8}$$

Proof:

1. Just prove that $\overset{*}{\longleftrightarrow}$ is a subset of every equivalence that contains $\longrightarrow$.
2. Use induction on the length of proofs to prove this part
3. For the last part it is easy to see that the existence of the "chain equation" (8) implies $L \overset{*}{\longleftrightarrow} M$ by transitivity. For the other part use induction on the length of the proof.

QED ■

Lecture 17: The Church-Rosser Property

Friday 09 Sep 2011

Definition 18.1 *The* **parallel-β** *or* $||\beta$ *reduction is the smallest relation for which the following rules hold.*

$$||\beta_1 \quad \frac{}{L \longrightarrow^1_{||\beta} L}$$

$$||\beta_1\textbf{Abs1} \quad \frac{L \longrightarrow^1_{||\beta} L'}{\lambda x[L] \longrightarrow^1_{||\beta} \lambda x[L']}$$

$$||\beta_1\textbf{App} \quad \frac{L \longrightarrow^1_{||\beta} L', M \longrightarrow^1_{||\beta} M'}{(L\ M) \longrightarrow^1_{||\beta} (L'\ M')}$$

$$||\beta_1\textbf{Abs2} \quad \frac{L \longrightarrow^1_{||\beta} L', M \longrightarrow^1_{||\beta} M'}{(\lambda x[L]\ M) \longrightarrow^1_{||\beta} \{M'/x\}L'}$$

Lemma 18.2

1. $L \longrightarrow^1_{\beta} L' \Rightarrow L \longrightarrow^1_{||\beta} L'$.
2. $L \longrightarrow^1_{||\beta} L' \Rightarrow L \longrightarrow^*_{\beta} L'$.
3. *The smallest preorder containing* $\longrightarrow^1_{||\beta}$ *is* $\longrightarrow^*_{||\beta} = \longrightarrow^*_{\beta}$.
4. *If* $L \longrightarrow^1_{\beta} L'$ *and* $M \longrightarrow^1_{||\beta} M'$ *then* $\{M/x\}L \longrightarrow^1_{||\beta} \{M'/x\}L'$.

Proof: By induction on the structure of terms or by induction on the number of steps in any proof. QED ■

Theorem 18.3 $\longrightarrow^1_{||\beta}$ *has the diamond property.*

Proof: We need to prove for all L

$$N \overset{1}{\underset{\|\beta}{\longleftarrow}} L \overset{1}{\underset{\|\beta}{\longrightarrow}} M \Rightarrow \exists P : N \overset{1}{\underset{\|\beta}{\longrightarrow}} P \overset{1}{\underset{\|\beta}{\longleftarrow}} M$$

We prove this by induction on the structure of L and a case analysis of the rule applied in definition 18.1.

Case $L \equiv x \in V$. Then $L \equiv M \equiv N \equiv P$.

Before dealing with the other inductive cases we dispose of some trivial sub-cases that arise in some or all of them.

Case $L \equiv_\alpha M$. Choose $P \equiv_\alpha N$ to complete the diamond.

Case $L \equiv_\alpha N$. Then choose $P \equiv_\alpha M$.

Case $M \equiv_\alpha N$. Then there is nothing to prove.

In the sequel we assume $N \not\equiv_\alpha L \not\equiv_\alpha M \not\equiv_\alpha N$ and proceed by induction on the structure of L.

Case $L \equiv \lambda x[L_1]$. Then clearly M and N were both obtained in proofs whose last step was an application of rule $\|\beta_1 Abs1$ and so $M \equiv \lambda x[M_1]$ and $N \equiv \lambda x[N_1]$ for some M_1 and N_1 respectively and hence $N_1 \overset{1}{\underset{\|\beta}{\longleftarrow}} L_1 \overset{1}{\underset{\|\beta}{\longrightarrow}} M_1$. By the induction hypothesis we have

$$\exists P_1 : N_1 \overset{1}{\underset{\|\beta}{\longrightarrow}} P_1 \overset{1}{\underset{\|\beta}{\longleftarrow}} M_1$$

Hence by choosing $P \equiv \lambda x[P_1]$ we obtain the required result.

Case $L \equiv (L_1\ L_2)$ *and* L_1 *is not an abstraction.*

The rule $||\beta_1 App$ is the only rule that must have been applicable in the last step of the proofs of $N \overset{1}{||\beta}\longleftarrow L \longrightarrow^1_{||\beta} M$. Clearly then there exist M_1, M_2, N_1, N_2 such that $N_1 \overset{1}{||\beta}\longleftarrow L_1 \longrightarrow^1_{||\beta} M_1$ and $N_2 \overset{1}{||\beta}\longleftarrow L_2 \longrightarrow^1_{||\beta} M_2$. Again by the induction hypothesis, we have

$$\exists P_1 : N_1 \longrightarrow^1_{||\beta} P_1 \overset{1}{||\beta}\longleftarrow M_1$$

and

$$\exists P_2 : N_2 \longrightarrow^1_{||\beta} P_2 \overset{1}{||\beta}\longleftarrow M_2$$

By choosing $P \equiv (P_1\ P_2)$ we obtain the desired result.

Case $L \equiv (\lambda x[L_1]\ L_2)$.

Here we have four sub-cases depending upon whether each of M and N were obtained by an application of $||\beta_1 App$ or $||\beta_1 Abs2$. Of these the sub-case when both M and N were obtained by applying $||\beta_1 App$ is easy and similar to the previous case. That leaves us with three subscases.

Sub-case: Both M *and* N *were obtained by applying rule* $||\beta_1 Abs2$.

Then we have

$$\{N_2/x\}N_1 \equiv N \overset{1}{||\beta}\longleftarrow L \equiv (\lambda x[L_1]\ L_2) \longrightarrow^1_{||\beta} M \equiv \{M_2/x\}M_1$$

for some M_1, M_2, N_1, N_2 such that

$$N_1 \overset{1}{||\beta}\longleftarrow L_1 \longrightarrow^1_{||\beta} M_1$$

and

$$N_2 \overset{1}{||\beta}\longleftarrow L_2 \longrightarrow^1_{||\beta} M_2$$

By the induction hypothesis

$$\exists P_1 : N_1 \longrightarrow^1_{||\beta} P_1 \ {}^1_{||\beta}\!\longleftarrow M_1$$

and

$$\exists P_2 : N_2 \longrightarrow^1_{||\beta} P_2 \ {}^1_{||\beta}\!\longleftarrow M_2$$

and the last part of lemma 18.2 we have

$$\exists P \equiv \{P_2/x\}P_1 : N \longrightarrow^1_{||\beta} P \ {}^1_{||\beta}\!\longleftarrow M$$

completing the proof.

Sub-case: M was obtained by applying rule $||\beta_1 Abs2$ and N by $||\beta_1 App$.

Then we have the form

$$(\lambda x[N_1]\ N_2) \equiv N \ {}^1_{||\beta}\!\longleftarrow L \equiv (\lambda x[L_1]\ L_2) \longrightarrow^1_{||\beta} M \equiv \{M_2/x\}M_1$$

where again

$$N_1 \ {}^1_{||\beta}\!\longleftarrow L_1 \longrightarrow^1_{||\beta} M_1$$

and

$$N_2 \ {}^1_{||\beta}\!\longleftarrow L_2 \longrightarrow^1_{||\beta} M_2$$

By the induction hypothesis

$$\exists P_1 : N_1 \longrightarrow^1_{||\beta} P_1 \ {}^1_{||\beta}\!\longleftarrow M_1$$

and

$$\exists P_2 : N_2 \longrightarrow^1_{||\beta} P_2 \ {}^1_{||\beta}\!\longleftarrow M_2$$

and finally we have

$$\exists P \equiv \{P_2/x\}P_1 : N \longrightarrow^1_{||\beta} P \overset{1}{\underset{||\beta}{\longleftarrow}} M$$

completing the proof.

Sub-case: M was obtained by applying rule $||\beta_1 App$ and N by $||\beta_1 Abs2$.

Similar to the previous sub-case.

QED ■

Theorem 18.4 $\longrightarrow^1_{||\beta}$ *is confluent.*

Proof: We need to show that for all L, M, N,

$$N \overset{*}{\underset{||\beta}{\longleftarrow}} L \longrightarrow^*_{||\beta} M \Rightarrow \exists P : N \longrightarrow^*_{||\beta} P \overset{*}{\underset{||\beta}{\longleftarrow}} M$$

We prove this by induction on the length of the sequences

$$L \longrightarrow^1_{||\beta} M_1 \longrightarrow^1_{||\beta} M_2 \longrightarrow^1_{||\beta} \cdots \longrightarrow^1_{||\beta} M_m \equiv M$$

and

$$L \longrightarrow^1_{||\beta} N_1 \longrightarrow^1_{||\beta} N_2 \longrightarrow^1_{||\beta} \cdots \longrightarrow^1_{||\beta} N_n \equiv N$$

where $m, n \geq 0$. More specifically we prove this by induction on the pairs of integers (j, i) bounded by (n, m), where $(j, i) < (j', i')$ if and only if either $j < j'$ or $(j = j')$ and $i < i'$. The interesting cases are those where both $m, n > 0$. So we repeatedly apply theorem

18.3 to complete the rectangle

$$
\begin{array}{ccccccccc}
L & \longrightarrow^1_{||\beta} & M_1 & \longrightarrow^1_{||\beta} & M_2 & \longrightarrow^1_{||\beta} & \cdots & \longrightarrow^1_{||\beta} & M_m \equiv M \\
||\beta\downarrow 1 & & ||\beta\downarrow 1 & & ||\beta\downarrow 1 & & \cdots & & ||\beta\downarrow 1 \\
N_1 & \longrightarrow^1_{||\beta} & P_{11} & \longrightarrow^1_{||\beta} & P_{12} & \longrightarrow^1_{||\beta} & \cdots & \longrightarrow^1_{||\beta} & P_{1m} \\
||\beta\downarrow 1 & & ||\beta\downarrow 1 & & ||\beta\downarrow 1 & & \cdots & & ||\beta\downarrow 1 \\
\vdots & & \vdots & & \vdots & & \cdots & & \vdots \\
||\beta\downarrow 1 & & ||\beta\downarrow 1 & & ||\beta\downarrow 1 & & \cdots & & ||\beta\downarrow 1 \\
N_n & \longrightarrow^1_{||\beta} & P_{n1} & \longrightarrow^1_{||\beta} & P_{n2} & \longrightarrow^1_{||\beta} & \cdots & \longrightarrow^1_{||\beta} & P_{nm} \equiv P
\end{array}
$$

QED ■

Corollary 18.5 $\longrightarrow^1_\beta$ *is confluent.*

Proof: Since $\longrightarrow^*_\beta = \longrightarrow^*_{||\beta}$ it follows from theorem 18.4 that $\longrightarrow^1_\beta$ is confluent. QED

Corollary 18.6 *If a term reduces to a β-normal form then the normal form is unique (upto $\equiv_\alpha$).*

Proof: If $N_1 \ {}^*_\beta\!\longleftarrow L \longrightarrow^*_\beta N_2$ and both N_1 N_2 are β-nfs, then by the corollary 18.5 they must both be β-reducible to a third element N_3 which is impoosible if both N_1 and N_2 are β-nfs. Hence β-nfs are unique whenever they exist. QED ■

Corollary 18.7 $\longrightarrow^1_\beta$ *is Church-Rosser.*

Proof: Follows from corollary 18.5 and theorem 19.1. QED ■

Lecture 19: Confluence Characterization

Tuesday 13 Sep 2011

Theorem 19.1 *The following statements are equivalent.*

1. $\longrightarrow$ *is Church-Rosser.*

2. $\longrightarrow$ *is confluent.*

3. $\longrightarrow$ *is semi-confluent*

Proof: (1 $\Rightarrow$ 2) Assume

$$N \stackrel{*}{\longleftarrow} L \stackrel{*}{\longrightarrow} M$$

Clearly then $N \stackrel{*}{\longleftrightarrow} M$. If $\longrightarrow$ is Church-Rosser then

$$\exists P : N \stackrel{*}{\longrightarrow} P \stackrel{*}{\longleftarrow} M$$

which implies that it is confluent.

(2 $\Rightarrow$ 3) Obviously any confluent relation is also semi-confluent.

(3 $\Rightarrow$ 2) Assume $L \stackrel{*}{\longleftrightarrow} M$. We proceed by induction on the length of the chain (8).

$$L \equiv_\alpha M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_m \equiv_\alpha M$$

Basis. $m = 0$. This case is trivial since for any P, $L \stackrel{*}{\longrightarrow} P$ iff $M \stackrel{*}{\longrightarrow} P$

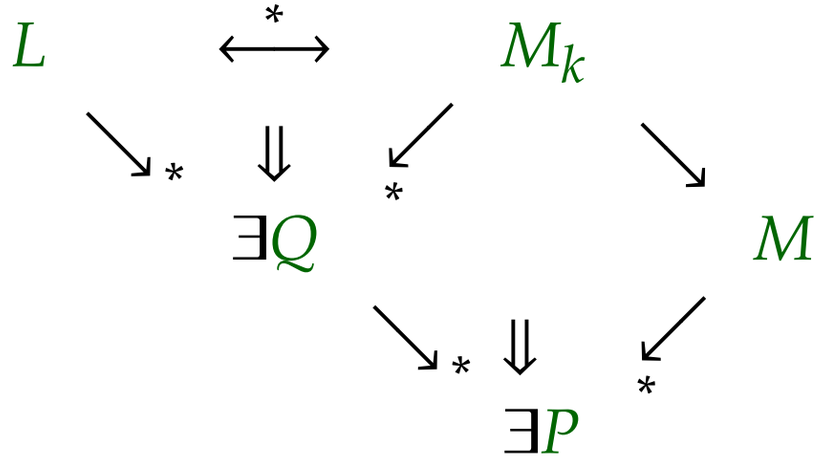
Induction Hypothesis (*IH*).

> *The claim is true for all chains of length k, $0 \leq k < m$.*

Induction Step. Assume the chain is of length $m = k + 1$. i.e.

$$L \equiv_\alpha M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_k \longrightarrow / \longleftarrow M_{k+1} \equiv_\alpha M$$

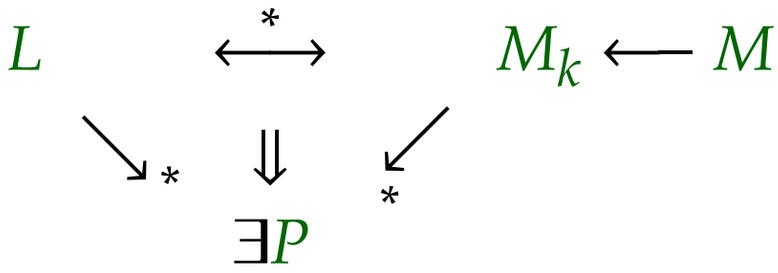
Case $M_k \longrightarrow M$. Then by the induction hypothesis and semi-confluence we have



which proves the claim.

Case $M_k \longleftarrow M$. Then the claim follows from the induction hypothesis

and the following diagram



QED ■

Lemma 19.2 *If a terminating relation is locally confluent then it is semi-confluent.*

Proof: Assume $L \longrightarrow M$ and $L \longrightarrow^* N$. We need to show that there exists P such that $M \longrightarrow^* P$ and $N \longrightarrow^* P$. We prove this by induction on the length of $L \longrightarrow^* N$. If $L \equiv_\alpha N$ then $P \equiv_\alpha M$, otherwise assume $L \longrightarrow N_1 \longrightarrow \cdots \longrightarrow N_n = N$ for some $n > 0$. By the local confluence we have there exists P_1 such that $M \longrightarrow^* P_1$. By successively applying the induction hypothesis we get terms $P_2, \ldots, P_n$ such that $P_{j-1} \longrightarrow^* P_j$ and $N_j \longrightarrow^* P_j$ for each j, $1 \leq j \leq m$. In effect we complete the following rectangle

$$\begin{array}{ccccccccc} L & \longrightarrow & N_1 & \longrightarrow & N_2 & \longrightarrow & \cdots & \longrightarrow & N_n \equiv M \\ \downarrow & & \downarrow & & \downarrow & & \cdots & & \downarrow \\ M & \longrightarrow & P_1 & \longrightarrow & P_2 & \longrightarrow & \cdots & \longrightarrow & P_n \end{array}$$

QED ■

From lemma 19.2 and theorem 19.1 we have the following theorem.

Theorem 19.3 *If a terminating relation is locally confluent then it is Church-Rosser and confluent.*

■

Lecture 25: The Damas Milner Algorithm

Tuesday 13 Sep 2011

The Damas-Milner algorithm for Type Assignment

The algorithm **W** uses *unification*. Assume $\mathbf{U}(\sigma, \tau) = V$, where **U** is a unification algorithm on type expressions, which takes two monotypes σ and τ as arguments and either fails or returns a substitution V which is the *mgu* of σ and τ.

The INPUT: An expression e and a type environment A (which consists of the assumptions about the types of some variables).

The OUTPUT: A substitution S and a type assignment τ to e.

The ALGORITHM: The algorithm $\mathbf{W}(A, e) = (S, \tau)$, is presented by induction on the structure of e.

1. *Case* $e \equiv x$ *and* $A(x) = \forall\alpha_1 \ldots \alpha_n[\sigma]$. Then $S = \varepsilon$ and for each new $\beta_i, 1 \leq i \leq n$ $\tau = \sigma\{\beta_i/\alpha_i \mid 1 \leq i \leq n\}$.

2. *Case* $e \equiv (e_1\ e_2)$. Then let $\mathbf{W}(A, e_1) = (S_1, \tau_1)$ and $\mathbf{W}(A, e_2) = (S_2, \tau_2)$ and $\mathbf{U}(\tau_1 S_2, \tau_2 \to \beta) = V$, where β is new. Then $S = S_1 \circ S_2 \circ V$ and $\tau = \beta V$.
3. *Case* $e \equiv \lambda x[e_1]$. Then let β be a new type variable, and $A' = \{x : \beta\} :: A$ be a modified environment such that $\mathbf{W}(A', e_1) = (S_1, \tau_1)$. Then $S = S_1$ and $\tau = \beta S_1 \to \tau_1$.
4. *Case* $e \equiv$ **let** $x = e_1$ **in** e_2. Let $\mathbf{W}(A, e_1) = (S_1, \tau_1)$, $A' = \{x : \gamma\} :: A$, where $\gamma = \tau_1 S_1$, $A'' = A' S_1$ and $\mathbf{W}(A'', e_2) = (S_2, \tau_2)$. Then $S = S_2 \circ S_1$ and $\tau = \tau_2$.
5. When any of the above conditions is not met then **W** fails.

References

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge Unviersity Press, Cambridge, U.K., 1998.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier Science B. V., Amsterdam, The Netherlands, 1984.

[3] R. Hindley and J. Seldin. *Combinators, λ-calculus*. London Mathematical Society, U.K., 1985.

[4] R. Sethi. *Programming Languages (Second Edition)*. Addison-Wesley Publishing Company, New York, U.S.A., 1996.

Thank You!

Any Questions?