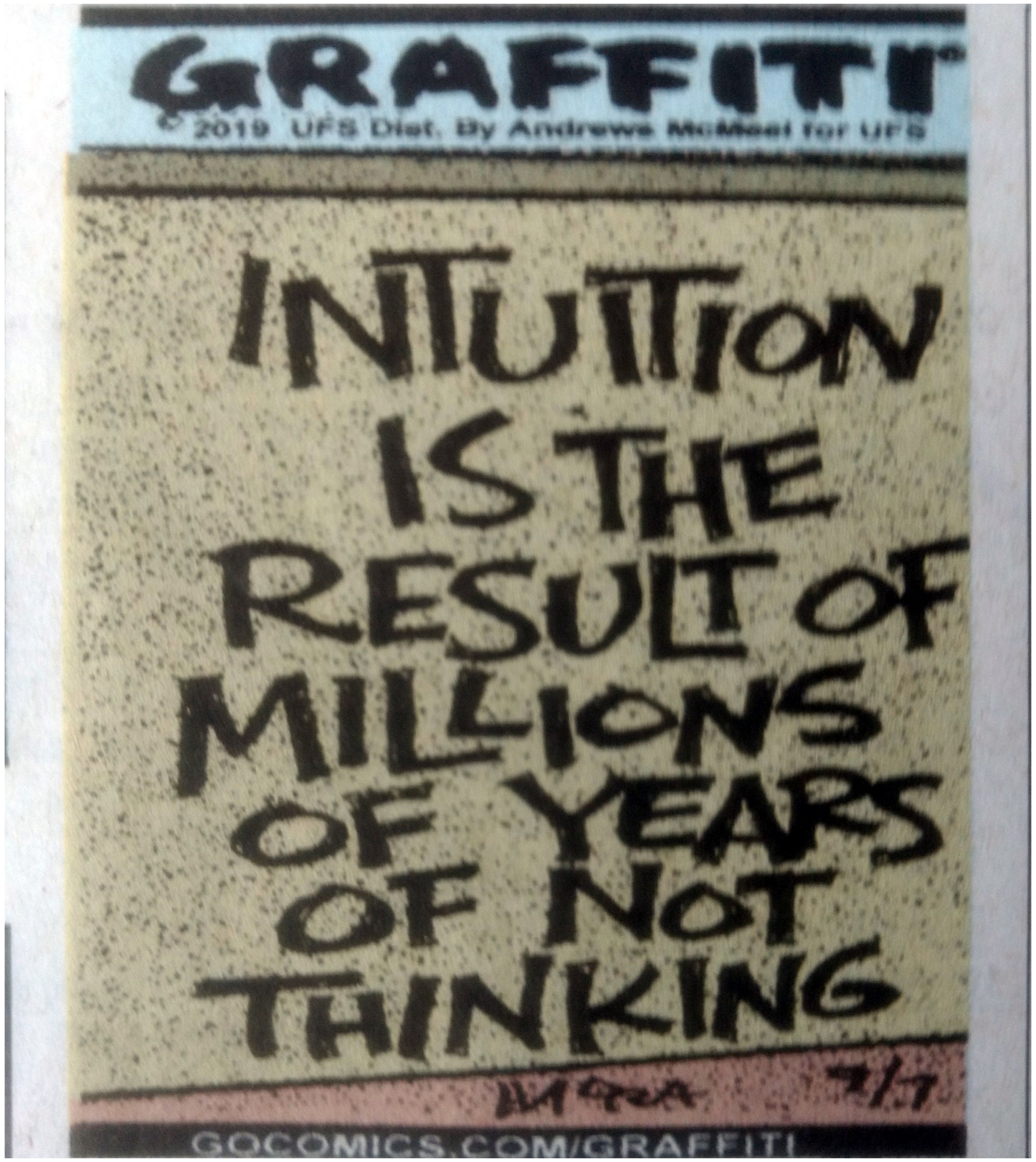
Introduction to (Logic and Functional) Programming

http://www.cse.iitd.ac.in/˜sak/courses/ilfp/2019-20/2019-20.index.html

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

September 25, 2019


GRAFFITI
© 2019 UFS Dist. By Andrews McMeel for UFS
INTUITION IS THE RESULT OF MILLIONS OF YEARS OF NOT THINKING
7/7
GOCOMICS.COM/GRAFFITI

Contents

Introduction

About this course

This covers aspects of

1. Programming paradigms
2. Programming Languages
3. Compilers
4. Logic
5. Software engineering

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.*

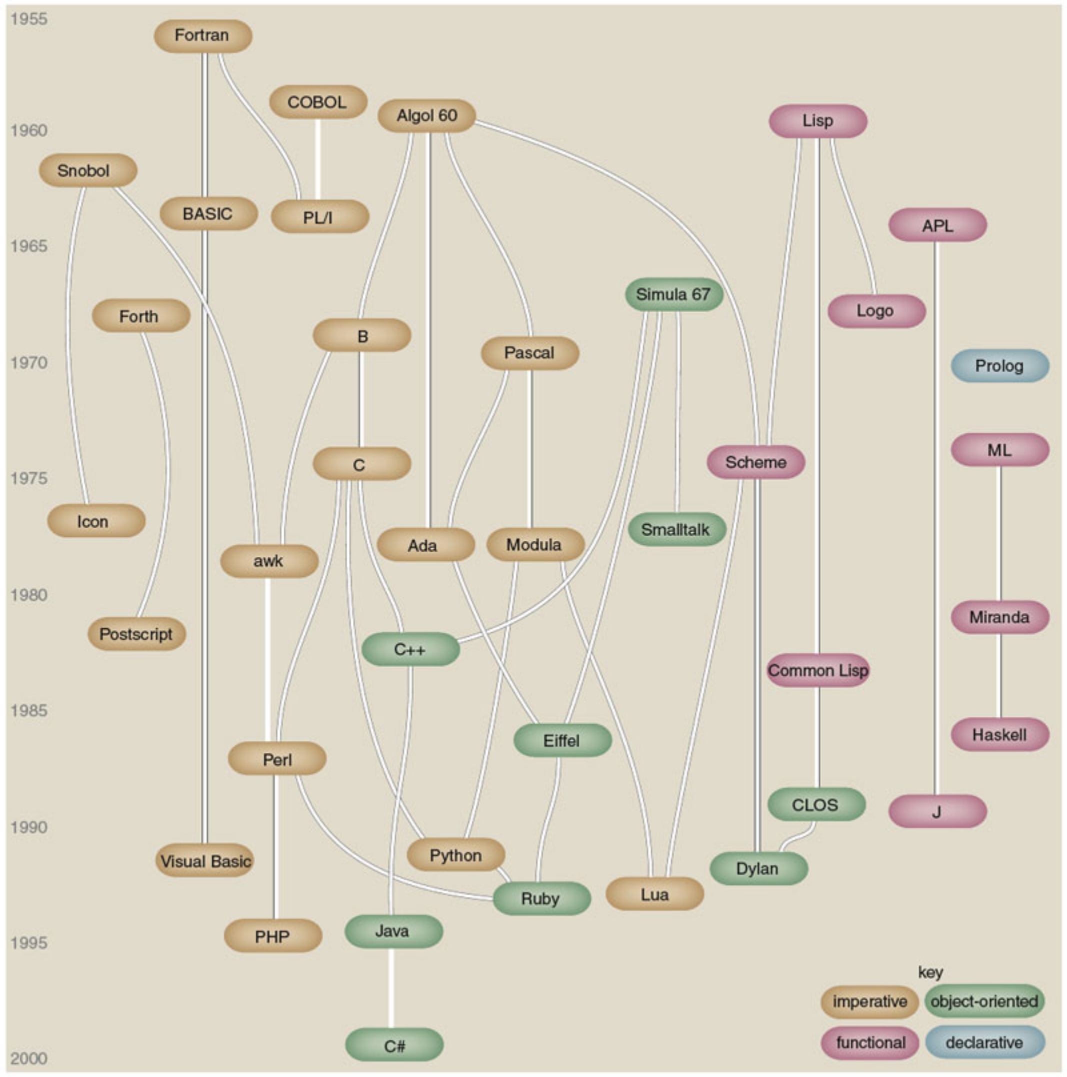
Programming languages History

- 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



Functional Programming

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.

Standard ML Overview

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.

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 $k = u-1$

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

where

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

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*

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

where

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

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);
```

Standard ML Computations

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	*Leftmost-outermost computation*
$f(g(a))$	$f(g(a))$
$= f(3 * 4 + 2)$	$= g(4)^2 + 1$
$= f(14)$	$= (3 * 4 + 2)^2 + 1$
	$= (12 + 2)^2 + 1$
$= 14^2 + 1$	$= 14^2 + 1$
$= 196 + 1$	$= 196 + 1$
$= 197$	$= 197$

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

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

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 $\mathcal{T}_{factL2(n)} = O(n)$

Standard ML Scope Rules

```
(* 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);
```

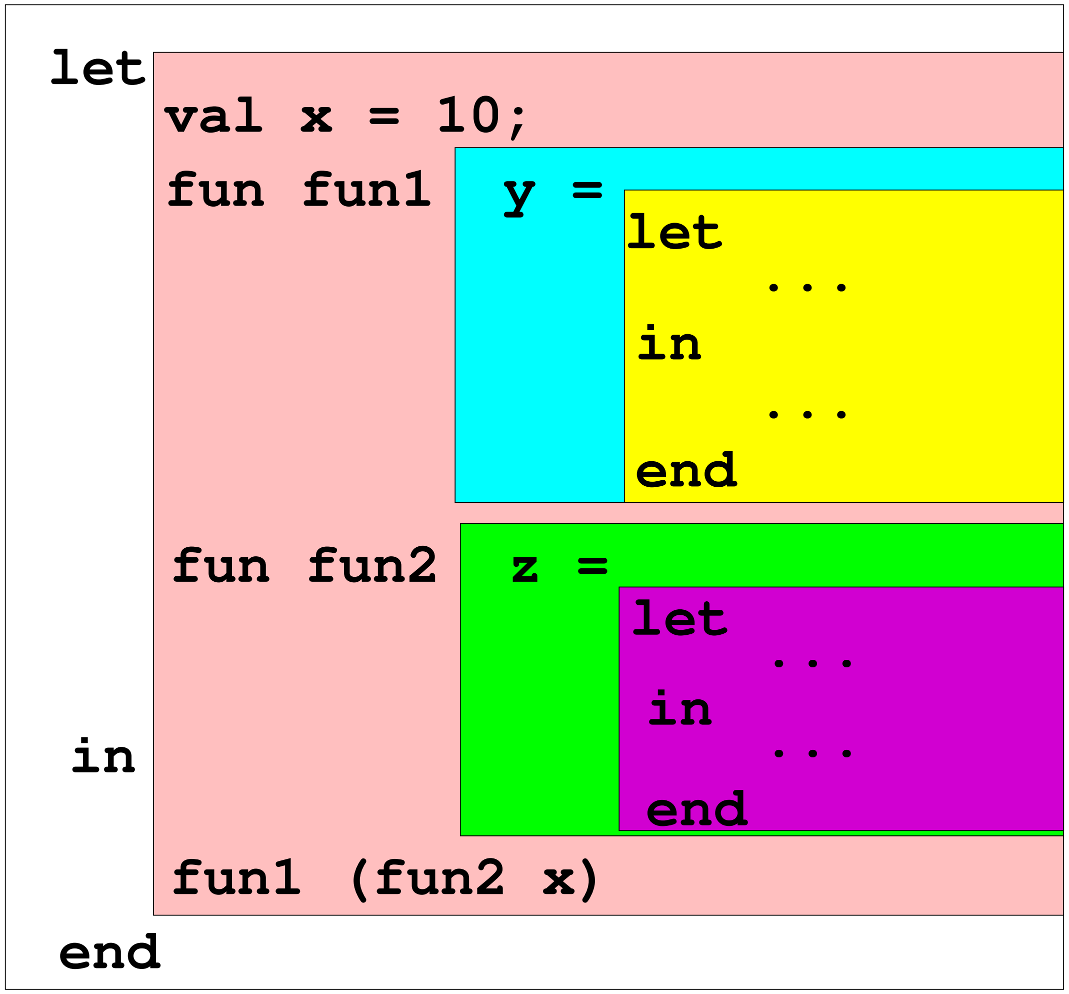
```
(* Program for generating primes upto some number *)
local
    fun primeWRT (m, []) = true
      | primeWRT (m, h::t) = if m mod h = 0 then false
                             else primeWRT (m, t)

    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
```

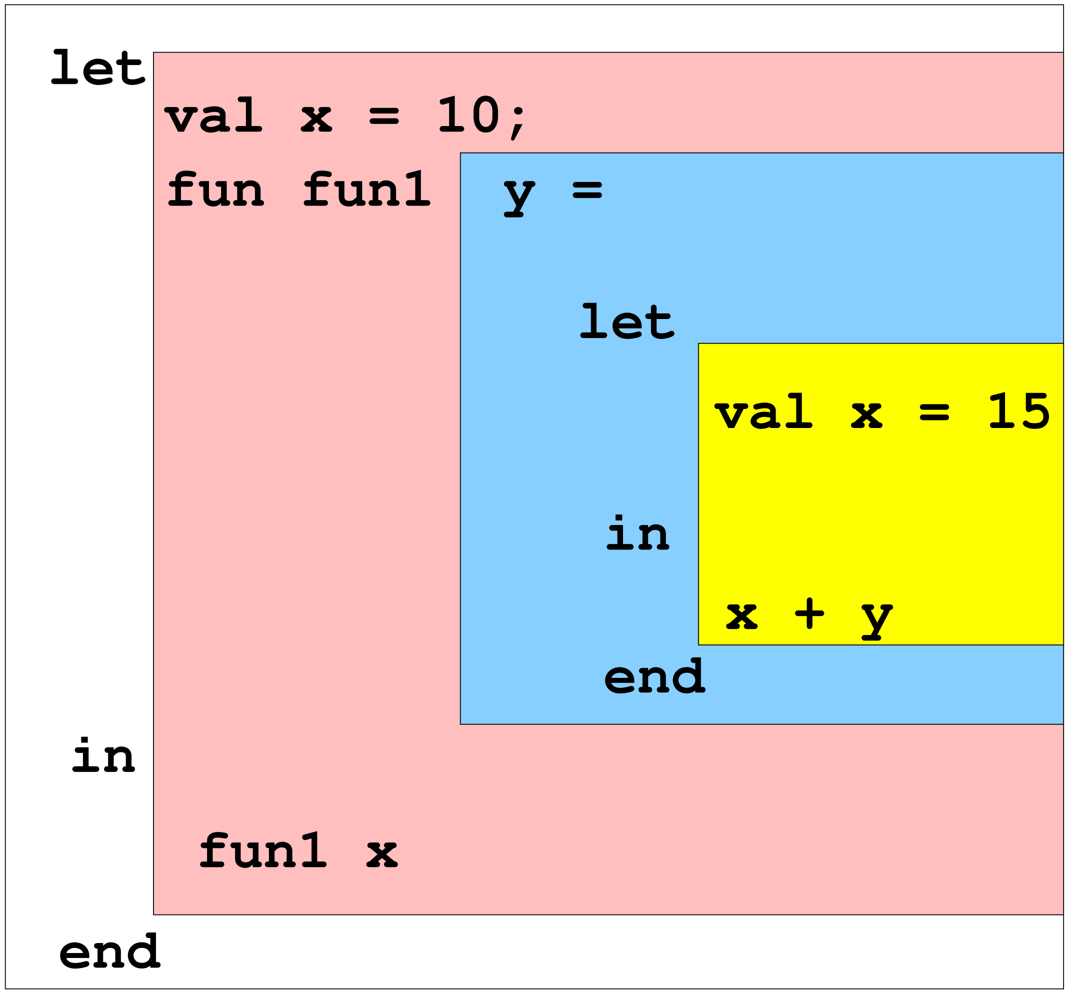
```
(* Program for generating primes upto some number *)
fun primesUpto n =
    if n < 2 then []
    else if n=2 then [2]
    else if (n mod 2 = 0) then primesUpto (n-1)
    else let fun generateFrom (P, m, n) =
                 let fun primeWRT (m, []) = true
                       | primeWRT (m, h::t) =
                         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



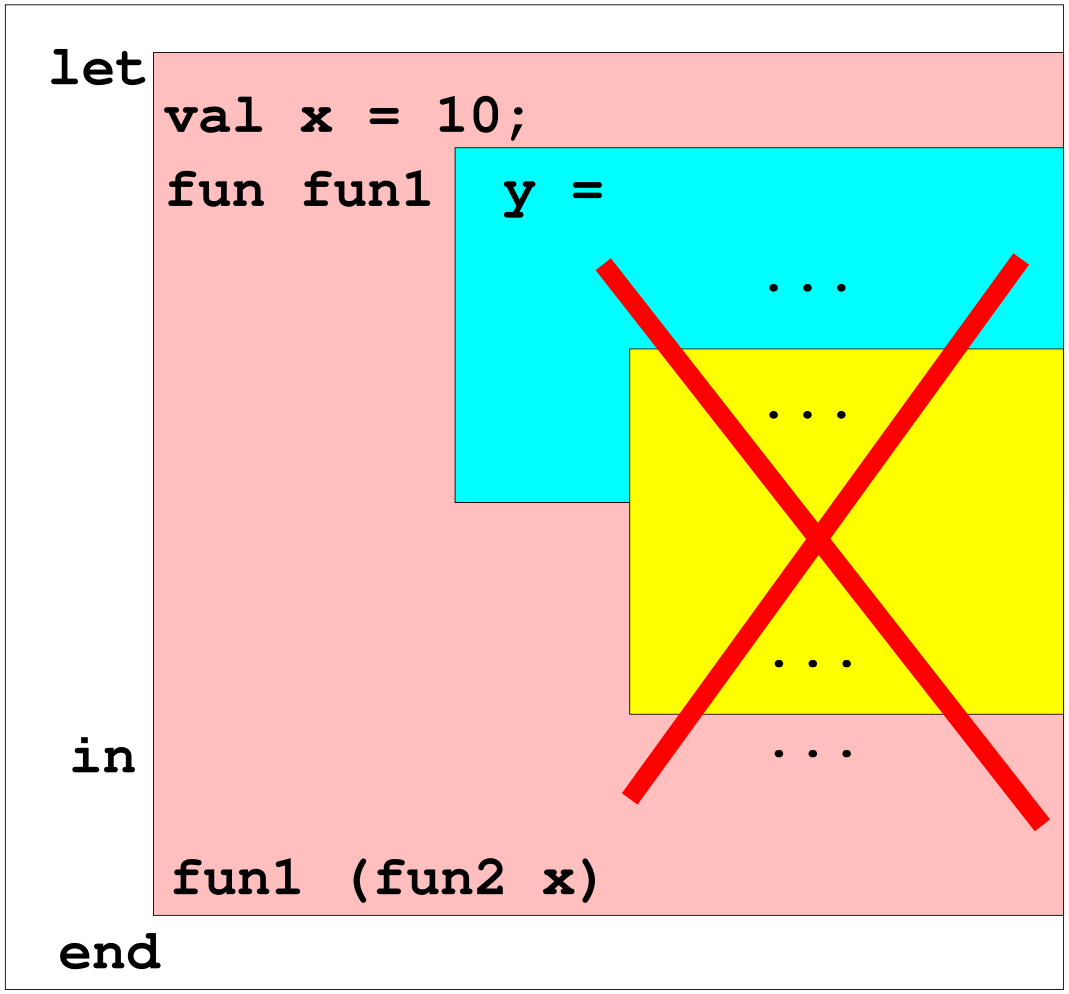
```
let
    val x = 10;
    fun fun1  y =  let
                       ...
                   in
                       ...
                   end

    fun fun2  z =  let
                       ...
                   in
                       ...
                   end
in
    fun1 (fun2 x)
end
```

Nested Scopes



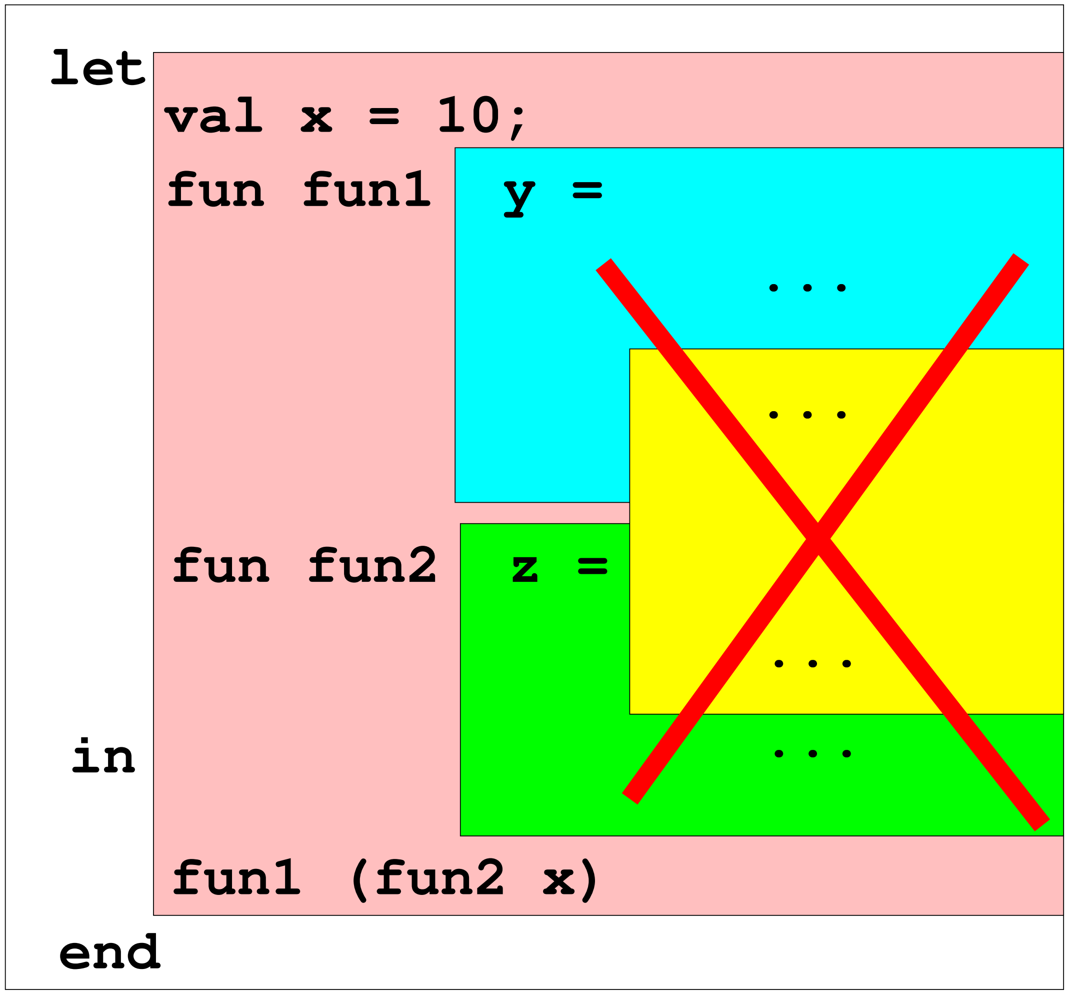
```
let
   val x = 10;
   fun fun1 y =
          let
             val x = 15
           in
             x + y
          end
 in
    fun1 x
 end
```

Overlapping Scopes



```
let
    val x = 10;
    fun fun1  y =
                    ...
                    ...
                    ...
                    ...
in
    fun1 (fun2 x)
end
```

Spannning



```
let
    val x = 10;
    fun fun1  y =
                    . . .
                    . . .
    fun fun2  z =
                    . . .
                    . . .
in
    fun1 (fun2 x)
end
```

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

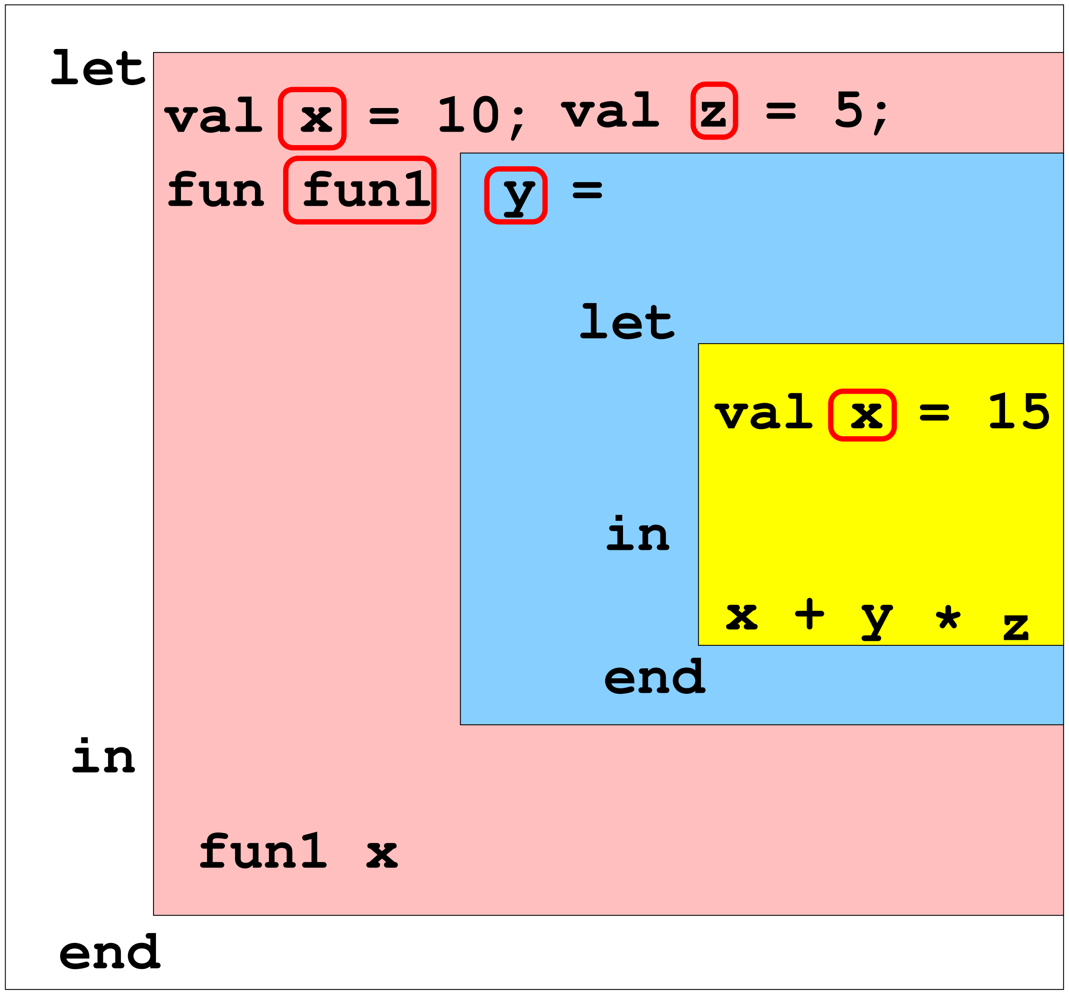
diagram

Names & References: 0

```
let
   val x = 10; val z = 5;
   fun fun1  y =

         let
            val x = 15

          in
            x + y * z
          end
 in
    fun1 x
 end
```

Back to Scope & Names

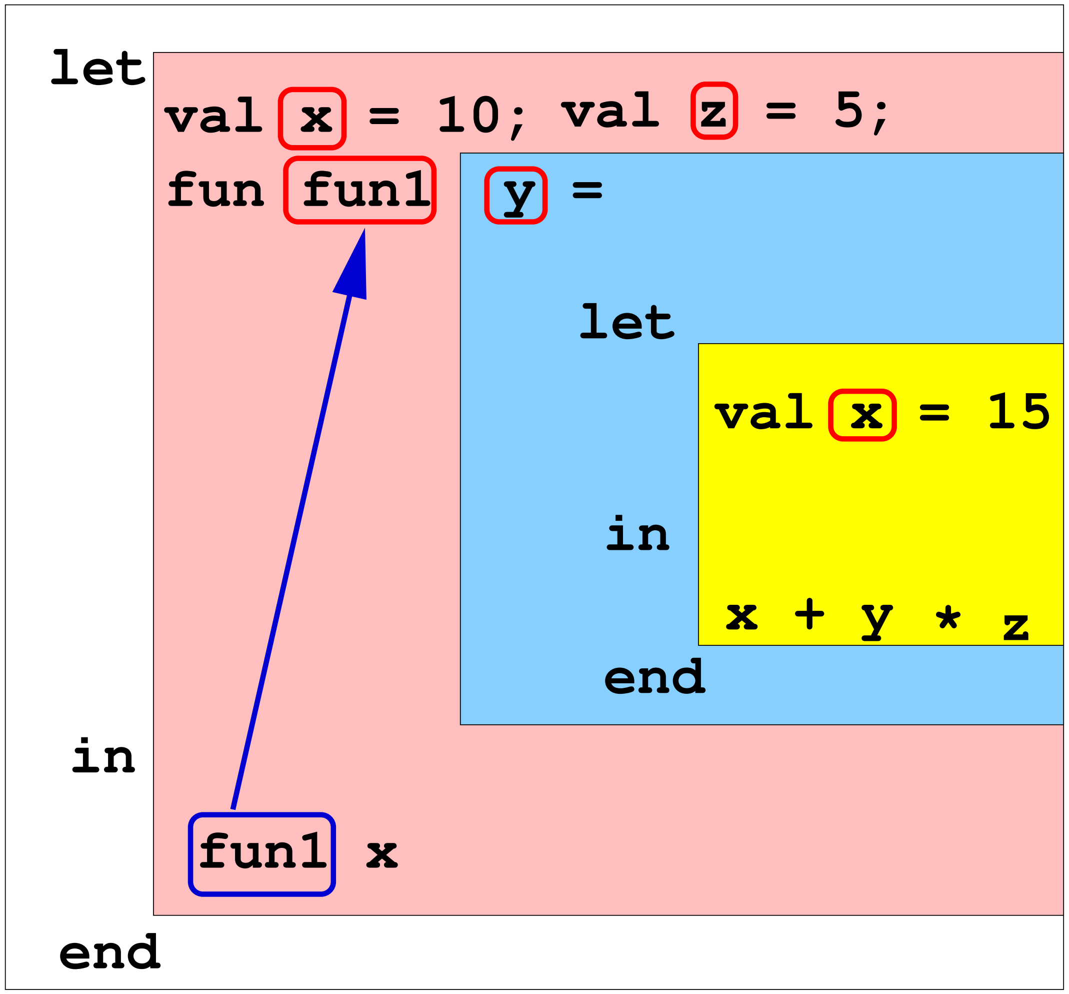
Names & References: 1



```
let
   val x = 10; val z = 5;
   fun fun1 y =
        let
            val x = 15
        in
            x + y * z
        end
in
   fun1 x
end
```

Back to Scope & Names

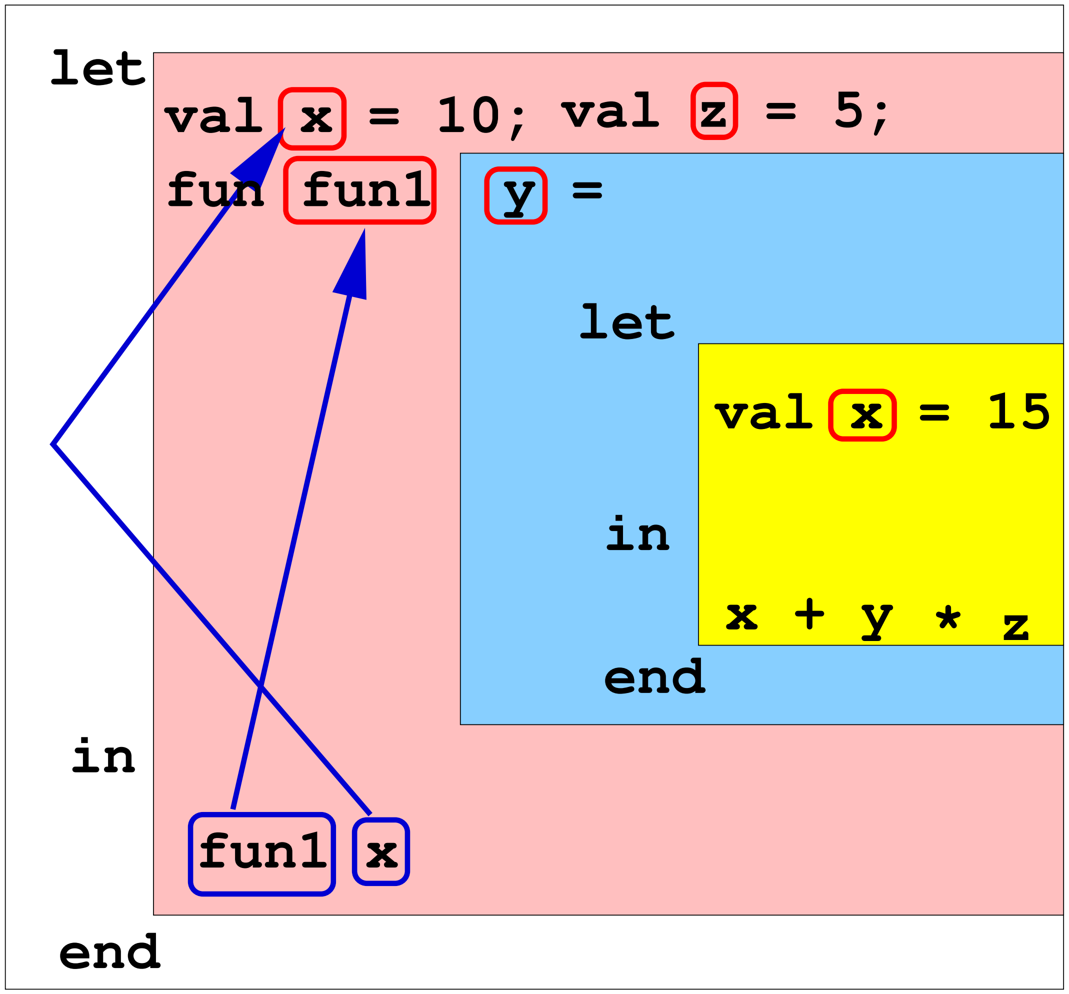
Names & References: 2



```
let
  val x = 10; val z = 5;
  fun fun1 y =
      let
          val x = 15
      in
          x + y * z
      end
in
  fun1 x
end
```

Back to Scope & Names

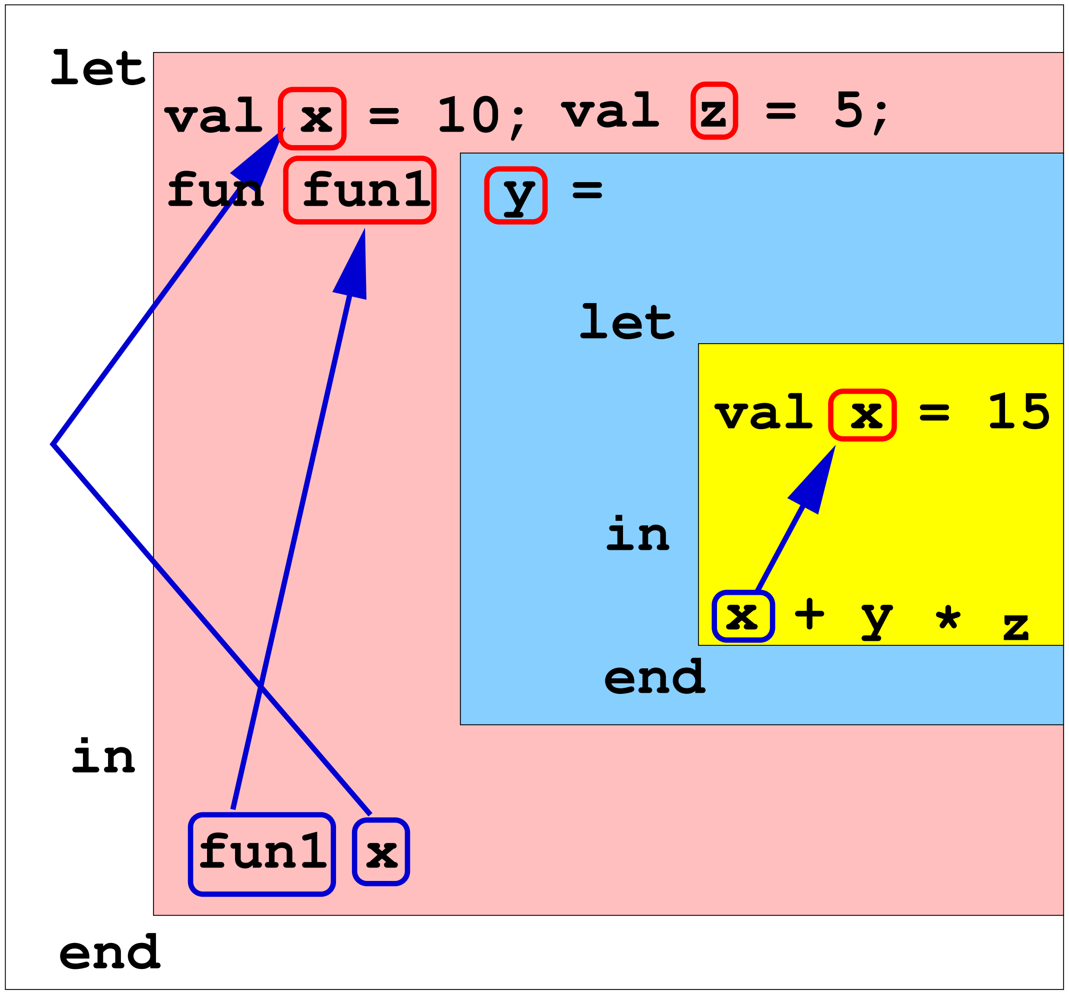
Names & References: 3



```
let
  val x = 10; val z = 5;
  fun fun1 y =
      let
          val x = 15
      in
          x + y * z
      end
in
  fun1 x
end
```

Back to Scope & Names

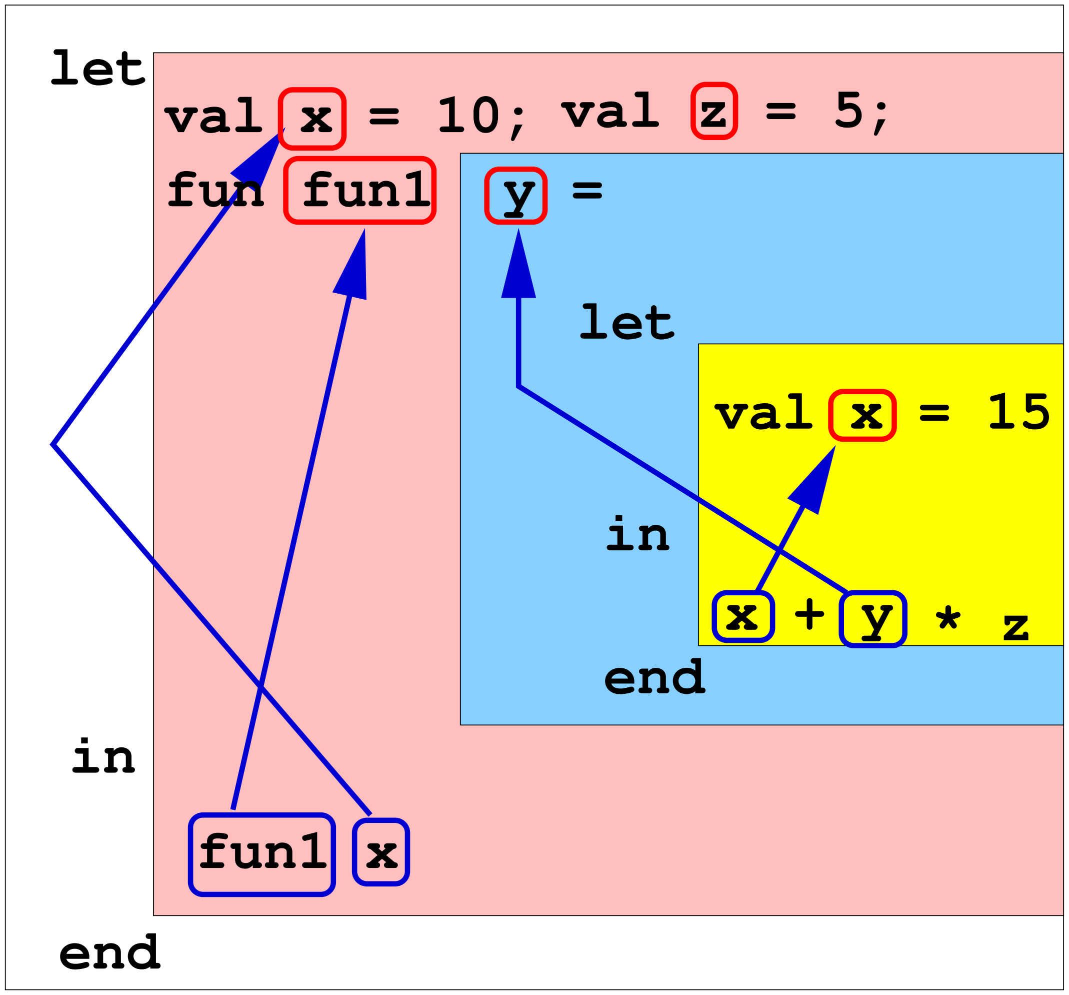
Names & References: 4



```
let
  val x = 10; val z = 5;
  fun fun1 y =
      let
          val x = 15
      in
          x + y * z
      end
in
  fun1 x
end
```

Back to Scope & Names

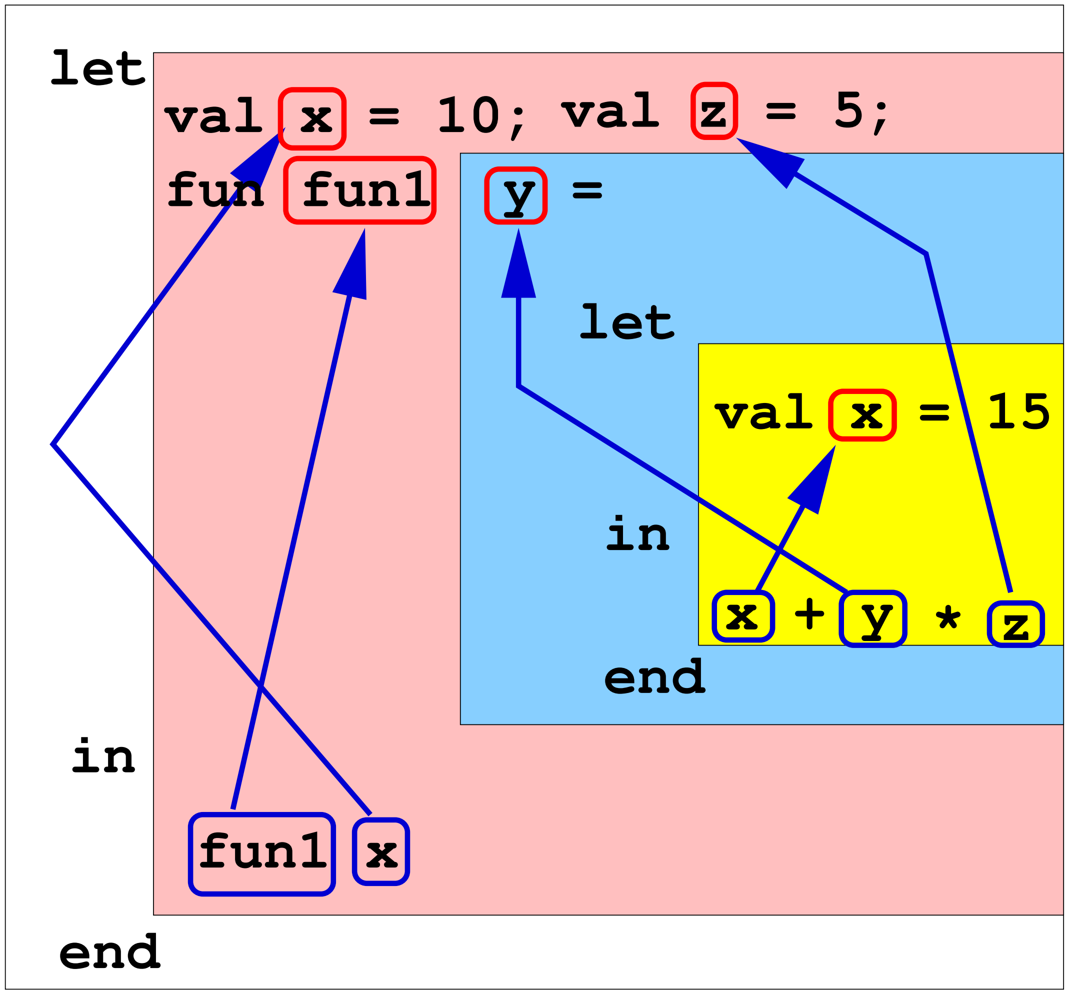
Names & References: 5



```
let
  val x = 10; val z = 5;
  fun fun1 y =
      let
          val x = 15
      in
          x + y * z
      end
in
  fun1 x
end
```

Back to Scope & Names

Names & References: 6



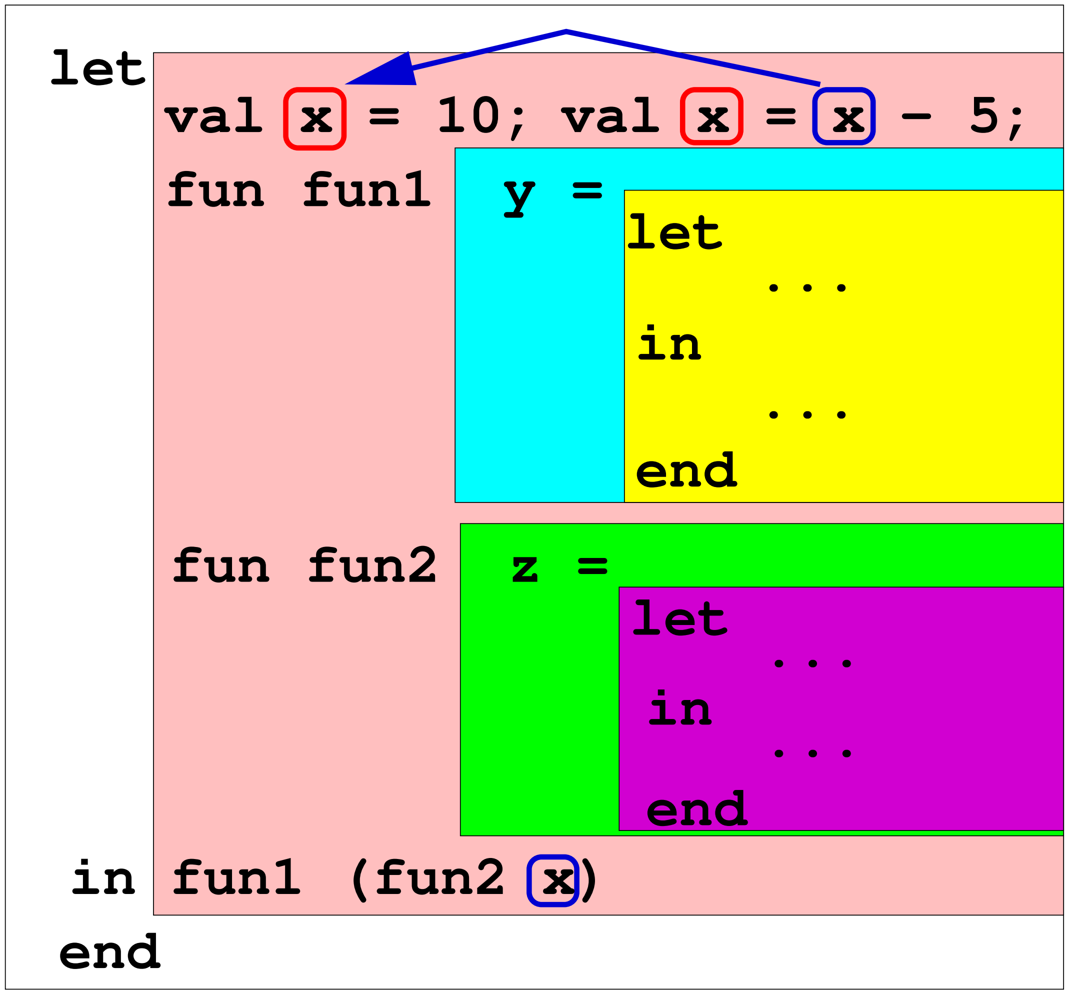
Back to Scope & Names

Names & References: 7

```
let
    val x = 10; val x = x - 5;
    fun fun1  y =
                 let
                       ...
                  in
                       ...
                  end
    fun fun2  z =
                 let
                       ...
                  in
                       ...
                  end
 in fun1  (fun2 x)
 end
```

Back to Scope & Names

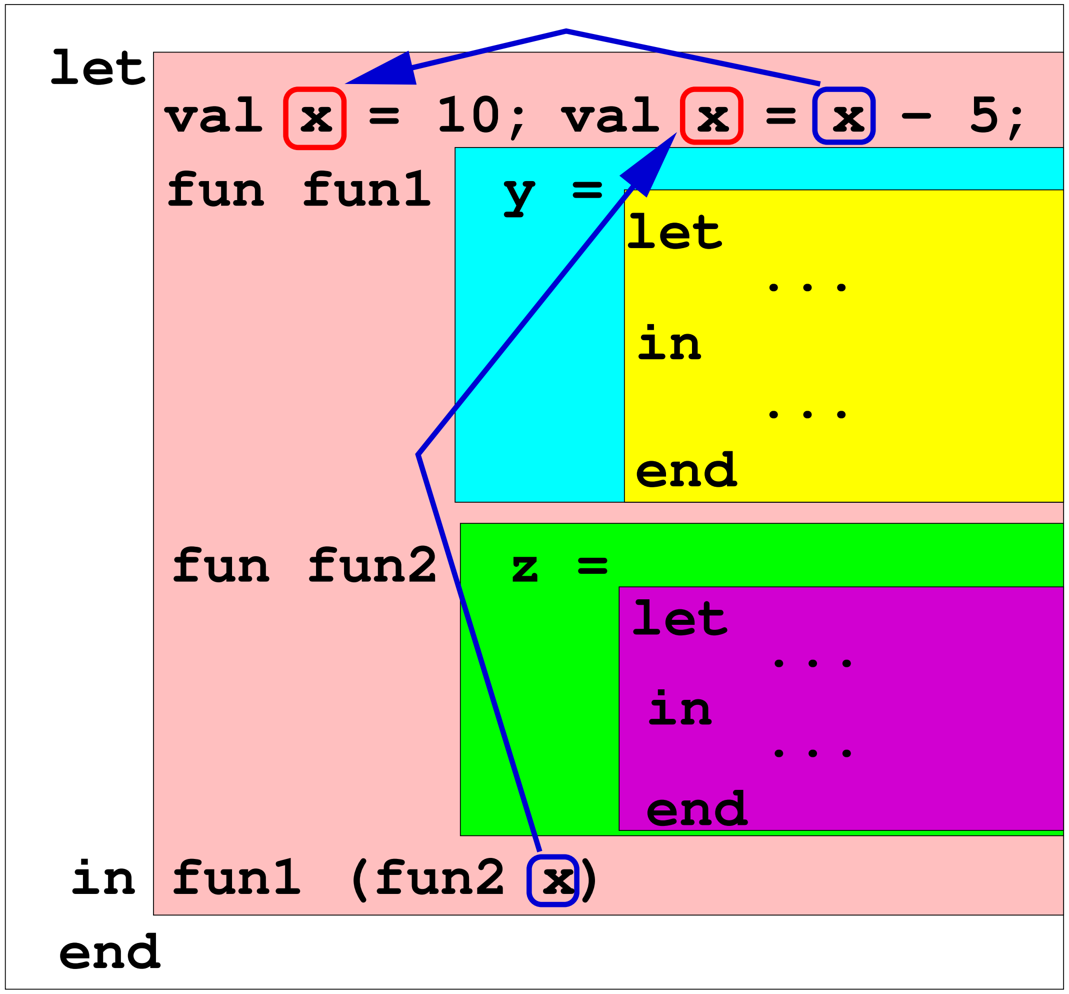
Names & References: 8



```
let
   val x = 10; val x = x - 5;
   fun fun1  y =  let
                     ...
                   in
                     ...
                   end
   fun fun2  z =  let
                     ...
                   in
                     ...
                   end
in fun1 (fun2 x)
end
```

Back to Scope & Names

Names & References: 9



```
let
   val x = 10; val x = x - 5;
   fun fun1  y =
                let
                      ...
                in
                      ...
                end
   fun fun2  z =
                let
                      ...
                in
                      ...
                end
in fun1  (fun2 x)
end
```

Back to Scope & Names

Definition of Names

Definitions are of the form

$\textit{qualifier}\ \underline{\textit{name}} \ldots = \textit{body}$

- `val` $\underline{\textit{name}}$ =
- `fun` $\underline{\textit{name}}$ ($\underline{\textit{argnames}}$) =
- `local` $\textit{definitions}$
  `in` $\textit{definition}$
  `end`

Use of Names

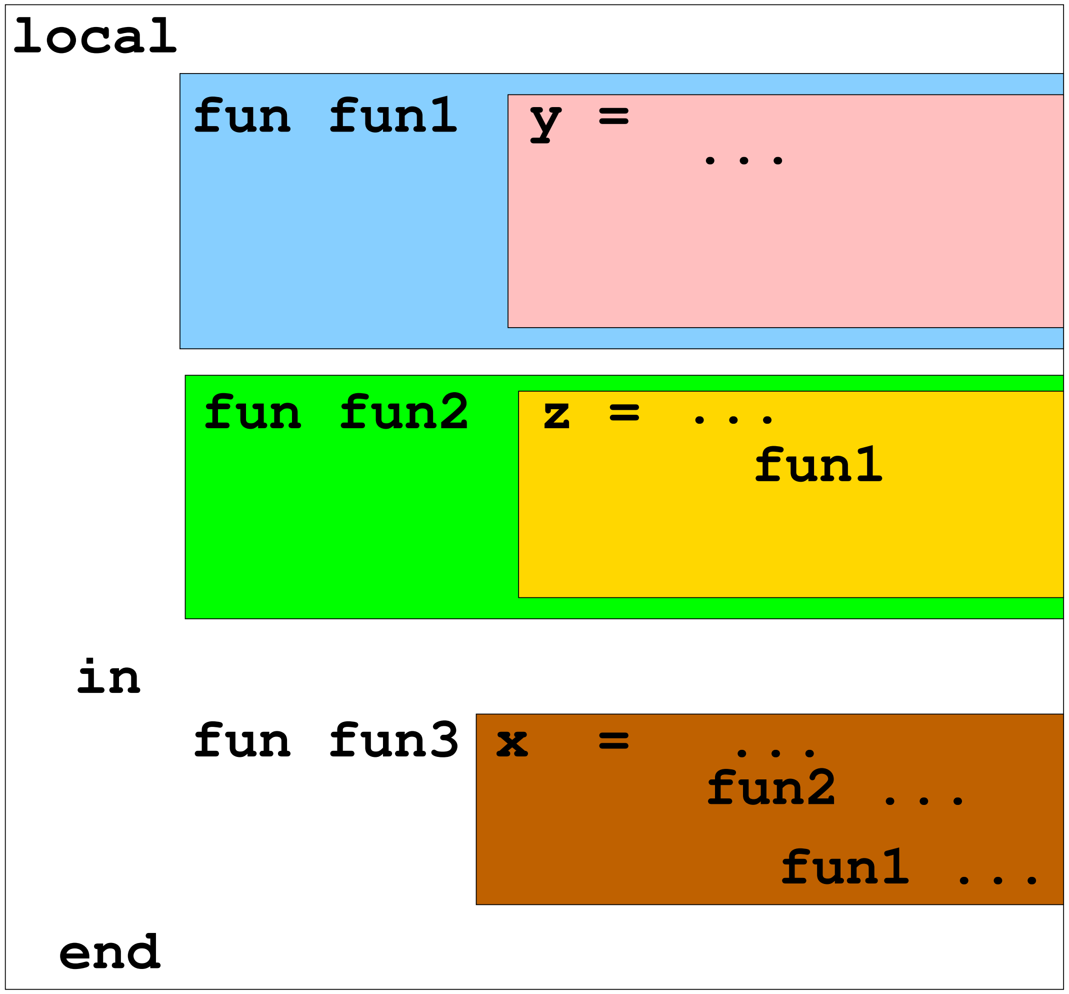
Names are used in expressions.
Expressions may occur

- 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`



```
local
        fun  fun1  y =
                          . . .

        fun  fun2  z =   . . .
                            fun1

  in
        fun  fun3  x   =      . . .
                          fun2  . . .

                             fun1  . . .
  end
```

Sample Sort Programs

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

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

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)
```

Higher-order Functions

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

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

   Example 7.3

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

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.

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

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 i.e.

$$map\,f[a_0,\ldots,a_{n-1}] = [f(a_0),\ldots,f(a_{n-1})]$$

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

2. `filter` “filters” out all elements of a list that do not satisfy a property i.e. it produces an (ordered) sub-list consisting of only those elements in the list for which the property holds

More Higher-order Functions on Lists

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

$$\begin{aligned} AGP4\ bop\ (a,d,n) &= \begin{cases} [] & \text{if } n \leq 0 \\ a :: (map\ (curry2\ bop\ d)(AGP4\ bop\ (a,d,n-1))) & \text{else} \end{cases} \\ AP4 &= AGP4\ op{+} \\ GP4 &= AGP4\ op{*} \end{aligned}$$

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

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

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

More Higher-order Functions on Lists

1. `foldl` applies a binary function to all elements of a list from left to right starting from an initial element e i.e.

$$foldl\ f\ e\ [a_0, \ldots, a_{n-1}] = f(a_{n-1}, f(a_{n-2}, \cdots f(a_0, e) \cdots))$$

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

2. `foldr` operates from right to left starting from an initial element e i.e.

$$foldr\ f\ e\ [a_0, \ldots, a_{n-1}] = f(a_0, f(a_1, \cdots f(a_{n-1}, e) \cdots))$$

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

Datatypes

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 $\vee$ 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.
```

```
                     1
                    / \
                   /   \
                  2     3
                   \   / \
                    4 5   7
                       /
                      6
*)
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 );
```

Information Hiding

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 $\underline{\text{absolutely}}$ 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.

                     1
                /  \
               /    \
              2      3
               \    /  \
                4  5    7
                  /
                6
*)
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);
```

Abstract Data Types to Modularity

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 *)
```

Signatures, Structures & Functors

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

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

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 $x : 'a$,

$$
\begin{array}{rlcll}
0. & nullq(emptyq) & & & \\
1. & not\ nullq(enqueue(q,x)) & & & \\
2. & qhd(emptyq) & = & Qerror & \\
3. & qhd(enqueue(q,x)) & = & x & \text{if } nullq(q) \\
4. & qhd(enqueue(q,x)) & = & qhd(q) & \text{if } not(nullq(q)) \\
5. & dequeue(emptyq) & = & Qerror & \\
6. & dequeue(enqueue(q,x)) & = & emptyq & \text{if } nullq(q) \\
7. & dequeue(enqueue(q,x)) & = & enqueue(dequeue(q),x) & \text{if } not(nullq(q))
\end{array}
$$

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*,

$$\begin{array}{lll}
0.\ nulls(emptys) & & \\
1.\ not\ nulls(push(s,x)) & & \\
2.\ top(emptys) & = & Serror \\
3.\ top(push(s,x)) & = & x \\
 & & \\
5.\ pop(emptys) & = & Serror \\
6.\ pop(push(s,x)) & = & s
\end{array}$$

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.

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

Notice that the BNF (2) allows stack expressions such as $\texttt{pop}\,(\texttt{emptys})$, $\texttt{pop}\,(\texttt{pop}\,(\texttt{emptys}))$, $\texttt{push}\,(\texttt{pop}\,(\texttt{emptys}), x)$ 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 `push`es 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.

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

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*,

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

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 11.4 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 in the 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

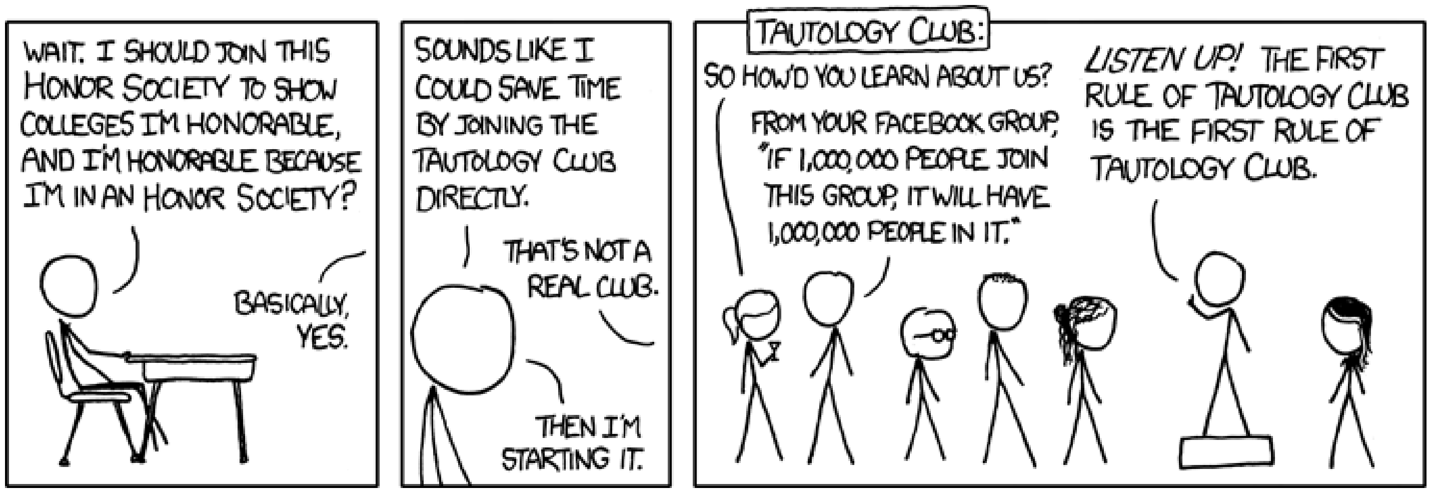
bstree-module.sml

```
(* 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);
```

Example: Tautology Checking


WAIT. I SHOULD JOIN THIS HONOR SOCIETY TO SHOW COLLEGES I'M HONORABLE, AND I'M HONORABLE BECAUSE I'M IN AN HONOR SOCIETY?
BASICALLY, YES.
SOUNDS LIKE I COULD SAVE TIME BY JOINING THE TAUTOLOGY CLUB DIRECTLY.
THAT'S NOT A REAL CLUB.
THEN I'M STARTING IT.
TAUTOLOGY CLUB:
SO HOW'D YOU LEARN ABOUT US?
FROM YOUR FACEBOOK GROUP, "IF 1,000,000 PEOPLE JOIN THIS GROUP, IT WILL HAVE 1,000,000 PEOPLE IN IT."
LISTEN UP! THE FIRST RULE OF TAUTOLOGY CLUB IS THE FIRST RULE OF TAUTOLOGY CLUB.

Arguments and Tautology Checking

Proving logical arguments

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* $\Gamma \models \psi$ $\underline{\textit{if and only if}}$ $((\ldots((\phi_1 \wedge \phi_2) \wedge \phi_3) \wedge \ldots \wedge \phi_n) \to \psi)$ *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. $\Gamma \models \psi$ $\underline{\textit{if and only if}}$ $\models \phi_1 \rightarrow (\phi_2 \rightarrow \cdots (\phi_n \rightarrow \psi) \cdots)$

2. $\Gamma \models \psi$ $\underline{\textit{if and only if}}$ $((\ldots((\phi_1 \wedge \phi_2) \wedge \phi_3) \wedge \ldots \wedge \phi_n) \wedge \neg\psi)$ *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)

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; *)
```

14: The Lambda Calculus: Introduction

As Curry points out in his classic work on Combinatory Logic,

> *Curiously a systematic notation for functions is lacking in ordinary mathematics. The usual notation '$f(x)$' does not distinguish between the function itself and the value of this function for an undetermined value of the argument.*

Let us consider the nature of functions, higher-order functions (functionals) and the use of naming in mathematics, through some examples.

Example 14.1 *Let $y = x^2$ be the squaring function on the reals. Here it is commonly understood that x is the "independent" variable and y is the "dependent" variable when we look on it as plotting the function $f(x) = x^2$ on the $x - y$ axis.*

Example 14.2 *Often a function may be named and written as $f(x) = x^n$ to indicate that x is the independent variable and n is understood (somehow!) to be some constant. Here f, x and n are all names with different connotations. Similarly in the quadratic polynomial $ax^2 + bx + c$ it is somehow understood that a, b and c denote constants and that x is the independent variable. Implicitly by using the names like a, b and c we are endeavouring to convey the impression that we consider the class $\{ax^2 + bx + c \mid a, b, c \in \mathbb{R}\}$ of all quadratic polynomials of the given form.*

Example 14.3 *As another example, consider the uni-variate polynomial $p(x) = x^2 + 2x + 3$. Is this polynomial the same as $p(y) = y^2 + 2y + 3$? Clearly they cannot be the same since the product $p(x).p(y)$ is a polynomial in two variables whereas $p(x).p(x)$ yields a uni-variate polynomial of degree 4. However, in the case of the function f in example 14.1 it does not matter whether we define the squaring function as $f(x) = x^2$ or as $f(y) = y^2$.*

Example 14.4 *The function $f(x) = x^2$ is a continuous and differentiable real-valued function (in the variable x) and its derivative is $f'(x) = 2x$. Whether we regard f' as the name of a new function or we regard the $'$ as an operation on f which yields its derivative seems to make no difference.*

Example 14.5 *Referring again to the functions $f(x)$ and $f'(x)$ in example 14.4, it is commonly understood that $f'(0)$ refers to the value of the derivative of f at 0 which is also the value of the function f' takes at 0. Now let us consider $f'(x+1)$. Going by the commonly understood notion, since $f'(x) = 2x$, we would have $f'(x+1) = 2(x+1)$. Then for $x = 0$ we have $f'(x+1) = f'(0+1) = f'(1) = 2 \times 1 = 2$. We could also think of it as the function $f'(g(0))$ where g is the function defined by $g(x) = x+1$, then $f'(g(0)) = 2g(0) = 2$ which yields the same result.*

The examples above give us some idea of why there is no systematic notation for functions which distinguishes between a function definition and the application of the same function to some argument. It simply did not matter!

However, this ambiguity in mathematical notation could lead to differing interpretationas and results in the context of mathematical theories involving higher-order functions (or "functionals" as they are often referred to). One common higher order function is the derivative (the differentiation operation) and another is the indefinite integral. Most mathematical texts emphasize the higher-order nature of a function by enclosing their arguments in (square) brackets. Hence if O is a functional which transforms a function $f(x)$ into a function $g(x)$, this fact is usually written $O[f(x)] = g(x)$.

Example 14.6 *Consider the functional E (on continuous real-valued functions of one real variable x) defined as follows.*

$$E[f(x)] = \begin{cases} f'(0) & \textit{if } x = 0 \\ \dfrac{f(x) - f(0)}{x} & \textit{if } x \neq 0 \end{cases}$$

The main question we ask now is "What does $E[f(x+1)]$ mean?"

It turns out that there are at least two ways of interpreting $E[f(x+1)]$ and unlike the case of example 14.5, the two interpretations actually yield different results!.

1. *We may interpret $E[f(x+1)]$ to mean that we first apply the transformation E to the function $f(x)$ and then*

substitute $x+1$ for x in the resulting expression. We then have the following.

$$\begin{aligned} & E[f(x)] \\ = & \begin{cases} f'(0) & \textit{if } x = 0 \\ \dfrac{f(x)-f(0)}{x} & \textit{if } x \neq 0 \end{cases} \\ = & \begin{cases} 0 & \textit{if } x = 0 \\ x & \textit{if } x \neq 0 \end{cases} \\ = & \ x \end{aligned}$$

Since $E[f(x)] = x$, $E[f(x+1)] = x+1$.

2. *Since $f(x+1) = f(g(x))$ where $g(x) = x+1$, we may interpret $E[f(x+1)]$ as applying the operator E to the function $h(x) = f(g(x))$. Hence $E[f(x+1)] = E[h(x)]$ where $h(x) = f(g(x)) = (x+1)^2 = x^2+2x+1$. Noting that $h'(x) = 2x+2$, $h(0) = 1$ and $h'(0) = 2$, we get*

$$\begin{aligned} & E[h(x)] \\ = & \begin{cases} h'(0) & \textit{if } x = 0 \\ \dfrac{h(x)-h(0)}{x} & \textit{if } x \neq 0 \end{cases} \\ = & \begin{cases} 2 & \textit{if } x = 0 \\ x+2 & \textit{if } x \neq 0 \end{cases} \\ = & \ x+2 \end{aligned}$$

The last example should clearly convince the reader that there is a need to disambiguate between a function definition and its application.

In function definitions the independent variables are "bound" by a λ which acts as a pre-declaration of the name that is going to be used in the expression that defines a function.

The notation $f(x)$, which is interpreted to refer to "the value of function f at x", will be replaced by $(f\ x)$ to denote an application of a function f to the (known or unknown) value x.

In our notation of the untyped applied λ-calculus the functions and their applications in the examples in subsection 14.1 would be rewritten as follows.

Squaring . $\lambda\ x[x^2]$ is the squaring function.

Example 14.2 . $q \overset{df}{=} \lambda\ a\ b\ c\ x[ax^2 + bx + c]$ refers to any quadratic polynomial with coefficients unknown or symbolic. To obtain a particular member of this family such as $1x^2 + 2x + 3$, one would have to evaluate $(((q\ 1)\ 2)\ 3)$ which would yield $\lambda\ x[1x^2 + 2x + 3]$.

Example 14.3 . $p \overset{df}{=} \lambda\ x[x^2 + 2x + 3]$. Then $p(x)$ would be written as $(p\ x)$ i.e. as the function p applied to the argument x to yield the expression $x^2 + 2x + 3$. Likewise $p(y)$ would be $(p\ y)$ which would yield $y^2 + 2y + 3$. The products $(p\ x).(p\ x)$ and $(p\ x).(p\ y)$ are indeed different and distinct.

Example 14.5 Let us denote the operation of obtaining the derivative of a real-valued function f of one independent variable x by the simple symbol D (instead of the more confusing $\frac{d}{dx}$). Then for any function f, $(D\ f)$ would yield the derivative. In particular $(D\ \lambda\ x[x^2]) = \lambda\ x[2x]$ and the value of the derivative at 0 would be obtained by the application $(\lambda\ x[2x]\ 0)$ which would yield 0. Likewise the value of the derivative at $x + 1$ would be expressed as the application $(\lambda\ x[2x]\ (x + 1))$. Thus for any function f the value of its derivative at $x + 1$ is simply the application $((D\ f)\ (x + 1))$.

The function $g(x) = x + 1$ would be defined as $g \overset{df}{=} \lambda\ x[x + 1]$ and $(g\ x) = x + 1$. Thus the alternative definition of the derivative of f at $x + 1$ is simply the application $((D\ f)\ (g\ x))$.

Example 14.6 The two interpretations of the expression $E[f(x + 1)]$ are respectively the following.

1. $((E\ f)\ (x + 1))$ and
2. $((E\ h)\ x)$ where $h \overset{df}{=} \lambda\ x[(f\ (g\ x))]$

15: The Pure Untyped Lambda Calculus: Basics

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*

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

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* $\lambda x[L]$.
- **Combinators**: $\Lambda_0 \subseteq \Lambda$ *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:*

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

- 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.
- On the other hand if $y \in BV(L)$ then the substitution will replace all free occurrences of x in L and bind some of them to an inner binding of y.

In the sequel we will often omit the subscript α and consider two alpha equivalent terms to be syntactically equivalent.

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*

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

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*

$$\boxed{\beta_1 \ \dfrac{L \rightarrow_\beta M}{L \rightarrow_\beta^1 M}} \qquad \boxed{\beta_1\mathbf{Abs} \ \dfrac{L \rightarrow_\beta^1 M}{\lambda x[L] \rightarrow_\beta^1 \lambda x[M]}}$$

$$\boxed{\beta_1\mathbf{AppL} \ \dfrac{L \rightarrow_\beta^1 M}{(L\ N) \rightarrow_\beta^1 (M\ N)}} \qquad \boxed{\beta_1\mathbf{AppR} \ \dfrac{L \rightarrow_\beta^1 M}{(N\ L) \rightarrow_\beta^1 (N\ M)}}$$

- $\rightarrow_\beta^1$ 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 $\longrightarrow^n_\beta$ is defined by induction on 1-step β-reduction*

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

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

$$\boxed{\beta_*\quad \frac{L \longrightarrow^n_\beta M}{L \longrightarrow^*_\beta M}\quad (n \geq 0)}$$

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* $L \rightarrow^1_\beta L_1 \rightarrow^1_\beta L_2 \rightarrow^1_\beta \cdots$

Untyped λ-Calculus: Examples

Example 15.8

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

2. $\Omega \overset{df}{=} (\omega\ \omega)$ *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 $\mathit{strongly}$ *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. *$((\mathsf{K}\ \omega)\ \omega)$ is strongly normalising because it reduces to the normal form ω in a single step.*
2. *Consider the term $((\mathsf{S}\ \mathsf{K})\ \mathsf{K})$. Its reduction sequences go as follows:*

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

16: Notions of Reduction

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 ($\longrightarrow^*_\beta$).*

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.

$=_\beta$ **Basis** $\dfrac{L \longrightarrow^*_\beta M}{L =_\beta M}$

$=_\beta$ **Reflexivity** $\dfrac{}{L =_\beta L}$

$=_\beta$ **Symmetry** $\dfrac{L =_\beta M}{M =_\beta L}$

$=_\beta$ **Transitivity** $\dfrac{L =_\beta M,\ M =_\beta N}{L =_\beta N}$

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. *$\rightarrow^1_\beta$ is by definition a compatible relation.*

Compatibility of Beta-reduction and Beta-Equality

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

$\square$

Proof of theorem 16.4

Proof: $(\rightarrow^*_\beta)$ Assume $L \rightarrow^*_\beta M$. By definition of β-reduction $L \rightarrow^n_\beta M$ 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*).

$$\boxed{\textit{The proof holds for all } k,\ 0 \leq k \leq m \textit{ for some } m \geq 0.}$$

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

$$\begin{array}{lll|l}
 & & & \text{By definition of } \rightarrow^n_\beta \\
\text{for all } x \in V, & \lambda x[L] \rightarrow^m_\beta \lambda x[L_m], & \lambda x[L_m] \rightarrow^1_\beta \lambda x[M] & \lambda x[L] \rightarrow^n_\beta \lambda x[M], \\
\text{for all } N \in \Lambda, & (L\ N) \rightarrow^m_\beta (L_m\ N), & (L_m\ N) \rightarrow^1_\beta (M\ N) & (L\ N) \rightarrow^n_\beta (M\ N) \\
\text{for all } N \in \Lambda, & (N\ L) \rightarrow^m_\beta (N\ L_m), & (N\ L_m) \rightarrow^1_\beta (N\ M) & (N\ L) \rightarrow^n_\beta (N\ M)
\end{array}$$

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 $L \rightarrow^*_\beta M$. The case of reflexivity is trivial and the case of $L \rightarrow^*_\beta M$ 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 $n \geq 1$, 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

$$\begin{array}{ll|l} & & \text{By } =_\beta \textbf{ Symmetry} \\ \text{for all variables } x, & \lambda x[M] =_\beta \lambda x[L] & \lambda x[L] =_\beta \lambda x[M] \\ \text{for all terms } N, & (M\ N) =_\beta (L\ N) & (L\ N) =_\beta (M\ N) \\ \text{for all terms } N, & (N\ M) =_\beta (N\ L) & (N\ M) =_\beta (N\ L) \end{array}$$

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

$$\begin{array}{lll|l} & & & \text{By } =_\beta \textbf{ Transitivity} \\ \text{for all variables } x, & \lambda x[L] =_\beta \lambda x[P], & \lambda x[P] =_\beta \lambda x[M] & \lambda x[L] =_\beta \lambda x[M] \\ \text{for all terms } N, & (L\ N) =_\beta (P\ N), & (P\ N) =_\beta (M\ N) & (L\ N) =_\beta (M\ N) \\ \text{for all terms } N, & (N\ L) =_\beta (N\ P), & (N\ P) =_\beta (N\ M) & (N\ L) =_\beta (N\ P) \end{array}$$

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

$$(\lambda x[(M\ x)]\ N) \rightarrow^1_\beta (M\ N)$$

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

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

Eta-Reduction and Eta-Equality

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

- 1-step η-reduction $\rightarrow^1_\eta$ is the closure of basic η-reduction to all contexts,
- $\rightarrow^n_\eta$ is defined by induction on 1-step η-reduction
- η-reduction $\rightarrow^*_\eta$ 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 $\longrightarrow^1_{\beta\eta}$, $\longrightarrow^*_{\beta\eta}$ and $=_{\beta\eta}$ are all compatible relations.*

The Paradoxical Combinator

Example 16.5 *Consider Curry's paradoxical combinator*

$$\mathsf{Y_C} \overset{df}{=} \lambda f[(\mathsf{C}\ \mathsf{C})]$$

where

$$\mathsf{C} \overset{df}{=} \lambda x[(f\ (x\ x))]$$

For any term L we have

$$\begin{aligned}
(\mathsf{Y_C}\ L) &\rightarrow^1_\beta (\lambda x[(L\ (x\ x))]\ \lambda x[(L\ (x\ x))]) \\
&\equiv_\alpha (\lambda y[(L\ (y\ y))]\ \lambda x[(L\ (x\ x))]) \\
&\rightarrow^1_\beta (L\ \underbrace{(\lambda x[(L\ (x\ x))]\ \lambda x[(L\ (x\ x))])}) \\
&=_\beta (L\ \overbrace{(\mathsf{Y_C}\ \mathsf{L})})
\end{aligned}$$

Hence $\boxed{(\mathsf{Y_C}\ L) =_\beta (L\ (\mathsf{Y_C}\ L))}$. *However* $(L\ (\mathsf{Y_C}\ L))$ *will never* β*-reduce to* $(\mathsf{Y_C}\ L)$.

Recursion and the Y combinator.

Since the lambda calculus only has variables and expressions and there is no place for names themselves (we use names such as K and S for our convenience in discourse, but the language itself allows only (untyped) variables and is meant to define functions anonymously as expressions in the language). In such a situation, recursion poses a problem in the language.

Recursion in most programming languages requires the use of an identifier which names an expression that contains a call to the very name of the function that it is supposed to define. This is at variance with the aim of the lambda calculus wherein the only names belong to variables and even functions may be defined anonymously as mere expressions.

This notion of recursive definitions may be generalised to a system of mutually recursive definitions.

The name of a recursive function, acts as a place holder in the body of the definition (which in turn has the name acting as a place holder for a copy of the body of the definition and so on ad infinitum). However no language can have sentences of infinite length.

The combinator $\mathsf{Y_C}$ helps in providing copies of any lambda term L whenever demanded in a more disciplined fashion. This helps in the modelling of recursive definitions anonymously. What the $\mathsf{Y_C}$ combinator provides is mechanism for recursion "unfolding" which is precisely our understanding of how recursion should work. Hence it is easy to see from $\boxed{(\mathsf{Y_C}\ L) =_\beta (L\ (\mathsf{Y_C}\ L))}$ that

$$(\mathsf{Y_C}\ L) =_\beta (L\ (\mathsf{Y_C}\ L)) =_\beta (L\ (L\ (\mathsf{Y_C}\ L))) =_\beta (L\ (L\ (L\ (\mathsf{Y_C}\ L)))) =_\beta \cdots$$

Many other researchers have defined other combinators which mimic the behaviour of the combinator

$\mathsf{Y_C}$. Of particular interest is Turing's combinator $\mathsf{Y_T} \overset{df}{=} (\mathsf{T}\ \mathsf{T})$ where $\mathsf{T} \overset{df}{=} \lambda x\, y[(y\ ((x\ x)\ y))]$. Notice that

$$\begin{array}{rl} & (\mathsf{T}\ \mathsf{T}) \\ \equiv & (\lambda x\, y[(y\ ((x\ x)\ y))]\ \mathsf{T}) \\ \to^1_\beta & \lambda y[(y\ ((\mathsf{T}\ \mathsf{T})\ y))] \\ \equiv & \lambda y[(y\ (\mathsf{Y_T}\ y))] \end{array}$$

from which, by compatible closure, for any term L we get

$$\begin{array}{rl} & (\mathsf{Y_T}\ L) \\ \equiv & ((\mathsf{T}\ \mathsf{T})\ L) \\ \to^*_\beta & (\lambda y[(y\ (\mathsf{Y_T}\ y))]\ L) \\ \to^1_\beta & (L\ (\mathsf{Y_T}\ L)) \end{array}$$

Thus $\mathsf{Y_T}$ is also a recursion unfolding combinator yielding

$$(\mathsf{Y_T}\ L) =_\beta (L\ (\mathsf{Y_T}\ L)) =_\beta (L\ (L\ (\mathsf{Y_T}\ L))) =_\beta (L\ (L\ (L\ (\mathsf{Y_T}\ L)))) =_\beta \cdots$$

Notice however that unlike the case of $(\mathsf{Y_C}\ L)$ which never *directly* reduces to $(L\ (\mathsf{Y_C}\ L))$, $(\mathsf{Y_T}\ L)$ does directly reduce to its unfolded version. That is,

$$(\mathsf{Y_T}\ L) \longrightarrow^*_\beta (L\ (\mathsf{Y_T}\ L)) \longrightarrow^*_\beta (L\ (L\ (\mathsf{Y_T}\ L))) \longrightarrow^*_\beta (L\ (L\ (L\ (\mathsf{Y_T}\ L)))) \longrightarrow^*_\beta \cdots$$

The Boolean Constants

$$\mathsf{True} \overset{df}{=} \lambda x[\lambda y[x]] \tag{True}$$

$$\mathsf{False} \overset{df}{=} \lambda x[\lambda y[y]] \tag{False}$$

Negation

$$\mathsf{Not} \overset{df}{=} \lambda x[((x\ \mathsf{False})\ \mathsf{True})] \tag{not}$$

The Conditional

$$\mathsf{Ite} \overset{df}{=} \lambda x\ y\ z[(x\ y\ z)] \tag{ite}$$

Exercise 17.1

1. *Prove that*

$$\begin{aligned} (\mathsf{Not\ True}) &=_{\beta\eta} \mathsf{False} && (8)\\ (\mathsf{Not\ False}) &=_{\beta\eta} \mathsf{True} && (9)\end{aligned}$$

2. *Prove that*

$$\begin{aligned} (\mathsf{Ite\ True}\ L\ M) &=_{\beta\eta} L && (10)\\ (\mathsf{Ite\ False}\ L\ M) &=_{\beta\eta} M && (11)\\ & && (12)\end{aligned}$$

3. *We know from Theorem 7.7 that the boolean constants and the conditional form a functionally complete (adequate) set for propositional logic. Use the conditional combinator* Ite *and the constant combinators* True *and* False *to express the following boolean operators upto $\beta\eta$-equivalence.*
   - Not*. Verify that it is α-equivalent to (not).*
   - And*: conjunction*
   - Or*: disjunction*
   - Xor*: exclusive OR*
4. *Prove the de Morgan laws for the boolean combinators, using only $\beta\eta$-reductions.*
5. *Does* ((And K) I) *have a $\beta\eta$-normal form?*

The Church Numerals There are many ways to represent the natural numbers as lambda expressions. Here we present Church's original encoding of the naturals in the λ-calculus. We represent a natural n as a combinator n.

$$\mathsf{0} \overset{df}{=} \lambda f\ x[x] \qquad \text{(numeral-0)}$$

$$\mathsf{1} \overset{df}{=} \lambda f\ x[(f\ x)] \qquad \text{(numeral-1)}$$

$$\ldots$$

$$\mathsf{n+1} \overset{df}{=} \lambda f\ x[(f\ (f^n\ x))] \qquad \text{(numeral-n+1)}$$

$$\ldots$$

where $(f^n\ x)$ denotes the n-fold application of f to x. That is, $(f^n\ x) = \underbrace{(f\ (f\ \ldots\ (f\ x)\ldots))}_{f \text{ applied } n \text{ times}}$.

"Arithmagic"

We follow the operators of Peano arithmetic and the postulates of first order arithmetic (as treated in any course in first order logic) and obtain "magically"[1] the following combinators for the basic operations of arithmetic and checking for 0.

[1]There are geniuses out there somewhere who manage to come up with these things. Don't ask me how they thought of them!

$$\mathsf{Succ} \overset{df}{=} \lambda n\ f\ x[((n\ f)\ (f\ x))] \tag{Succ}$$

$$\mathsf{Add} \overset{df}{=} \lambda m\ n\ f\ x[((m\ f)\ (n\ f\ x))] \tag{Add}$$

$$\mathsf{IsZero} \overset{df}{=} \lambda n[(n\ \lambda x[\mathsf{False}]\ \mathsf{True})] \tag{13}$$

The only way to convince oneself that the above are correct, is to verify that they do produce the expected results.

Exercise 17.2

1. *Prove the following.*
   (a) $(\mathsf{Succ}\ 0) =_{\beta\eta} 1$
   (b) $(\mathsf{Succ}\ \mathsf{n}) =_{\beta\eta} \mathsf{n} + 1$
   (c) $(\mathsf{IsZero}\ 0) =_{\beta\eta} \mathsf{True}$
   (d) $(\mathsf{IsZero}\ (\mathsf{Succ}\ \mathsf{n})) =_{\beta\eta} \mathsf{False}$
   (e) $(\mathsf{Add}\ 0\ \mathsf{n}) =_{\beta\eta} \mathsf{n}$
   (f) $(\mathsf{Add}\ \mathsf{m}\ 0) =_{\beta\eta} \mathsf{m}$
   (g) $(\mathsf{Add}\ \mathsf{m}\ \mathsf{n}) =_{\beta\eta} \mathsf{p}$ *where* p *denotes the combinator for* $p = m + n$
2. *Try to reduce* $(\mathsf{Add}\ \mathsf{K}\ \mathsf{S})$ *to its* β*-normal form. Can you interpret the resulting lambda term as representing some meaningful function?*

Ordered Pairs and Tuples

$$\begin{aligned}
\mathsf{Pair} &\overset{df}{=} \lambda x\, y\, p[(p\ x\ y)] && (14)\\
\mathsf{Fst} &\overset{df}{=} \lambda p[(p\ \mathsf{True})] && (15)\\
\mathsf{Snd} &\overset{df}{=} \lambda p[(p\ \mathsf{False})] && (16)\\
& && (17)
\end{aligned}$$

We may define an n-tuple inductively as a pair consisting of the first element of the n-tuple and an $n-1$ tuple of the other $n-1$ elements. Let $\langle L, M\rangle$ represent a pair. We then have for any $n > 2$

$$\langle L_1, \ldots, L_n\rangle = (\mathsf{Pair}\ L_1\ \langle L_2, \ldots, L_n\rangle)$$

Note the isomorphism between lists of length n and n-tuples for each $n \geq 2$ (ordered pairs are 2-tuples).

Exercise 17.3

1. *Let* $P \overset{df}{=}$ (Pair L M). *Verify that* (Pair (Fst P) (Snd P)) $=_{\beta\eta} P$.
2. *Let* Sfst $\overset{df}{=}$ (Fst S) *and* Ssnd $\overset{df}{=}$ (Snd S).
   (a) *Compute the* $\beta\eta$ *normal form of* (Pair Sfst Ssnd)*? Is it* $\beta\eta$*-equal to* S*?*
   (b) *Now compute the* $\beta\eta$ *normal forms of* (Fst (Pair Sfst Ssnd)) *and* (Snd (Pair Sfst Ssnd)). *What are their* $\beta\eta$ *normal forms?*
   (c) *What can you conclude from the above?*
3. *For any k,* $0 \leq k < n$*, define combinators which extract the k-th component of an n-tuple.*
4. (a) *Define a combinator* Bintree *that constructs binary trees from* λ*-terms with node labels drawn from the Church numerals.*
   (b) *Define combinators* Root, Lst *and* Rst *which yield respectively the root, the left subtree and the right subtree of a binary tree.*
   (c) *Prove that for any such binary tree* B *expressed as a* λ*-term,* (Bintree (Root B) (Lst B) (Rst B)) $=_{\beta\eta} B$.

18: Confluence

Reduction Relations

Definition 18.1 *For any binary relation ρ on Λ*

1. ρ^1 *is the compatible closure of* ρ
2. ρ^+ *is the transitive closure of* ρ^1
3. ρ^* *is the reflexive-transitive-closure of* ρ^1 *and is a preorder*
4. $((\rho^1) \cup (\rho^1)^{-1})^*$ *(denoted* $=_\rho$*) is the reflexive-symmetric-transitive closure of* ρ^1 *and is an equivalence relation.*
5. $=_\rho$ *is also called the equivalence* **generated** *by* ρ.

We will often use $\longrightarrow$ (suitably decorated) as a reduction relation instead of ρ. Then $\longrightarrow^1$, $\longrightarrow^+$, $\longrightarrow^*$ and $\overset{*}{\longleftrightarrow}$ denote respectively the compaticble closure, the transitive closure, the reflexive transitive closure and the equivalence generated by $\longrightarrow$

The Diamond Property

Definition 18.2 *Let ρ be any relation on terms. ρ has the* **diamond property** *if for all L, M, N,*

$$\begin{tikzcd}
 & M & & M & \\
L \arrow[ur, phantom, "\rho"] \arrow[dr, phantom, "\rho"'] & & \Rightarrow \exists P : & & P \arrow[ul, phantom, "\rho"'] \arrow[dl, phantom, "\rho"] \\
 & N & & N &
\end{tikzcd}$$

We often use a decorated version of the symbol $\longrightarrow$ for a reduction relation and depict the diamond property as

$$\begin{tikzcd}
 & M \arrow[dr] & \\
L \arrow[ur] \arrow[dr] & & \Rightarrow \exists\ P \\
 & N \arrow[ur] &
\end{tikzcd}$$

Reduction Relations: Termination

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

Definition 18.3 $\longrightarrow$ *is* **terminating** *iff there is no infinite sequence of the form*

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

Lemma 18.4 $\longrightarrow_\eta$ *is a terminating reduction relation.*

Proof: By induction on the structure of terms. QED

18.1. Why confluence?

We are mostly interested in β-reduction which is not guaranteed to terminate. We already know that there are several terms which are only weakly normalising (β-WN). This means that there are several possible reduction sequences, some of which may yield β-normal forms while the others may yield infinite computations. Hence in order to obtain normal forms for such terms we need to schedule the β-reductions carefully to be guaranteed a normal form. The matter would be further complicated if there are multiple unrelated normal forms.

Each β-reduction step may reveal fresh β-redexes. This in turn raises the disquieting possibility that each termination sequence may yield a different β-normal form. If such is indeed the case, then it raises fundamental questions on the use of β-reduction (or function application) as a notion of reduction. If *beta*-reduction is to be considered fundamental to the notion of computation then all β-reduction sequences that terminate in β-nfs must yield the same β-nf upto α-equivalence.

Hence our interest in the notion of confluence. Since the issue of confluence of β-reduction is rather complicated we approach it in terms of inductively easier notions such as *local confluence*, and *semi-confluence* which finally lead up to *confluence* and the Church-Rosser property.

Reduction: Local Confluence

Definition 18.5 $\longrightarrow$ *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

$$\begin{tikzcd} & M \arrow[dr, "*"] & \\ L \arrow[ur] \arrow[dr] & & \Rightarrow \exists\ P \\ & N \arrow[ur, "*"'] & \end{tikzcd}$$

Reduction: Semi-confluence

Definition 18.6 $\longrightarrow$ *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

$$\begin{tikzcd}
 & M \arrow[dr, "*"] & \\
L \arrow[ur] \arrow[dr, "*"'] & & \Rightarrow \exists\ P \\
 & N \arrow[ur, "*"'] &
\end{tikzcd}$$

Reduction: Confluence

Definition 18.7 $\longrightarrow$ *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

$$\begin{tikzcd}
 & M \arrow[dr, "*"] & \\
L \arrow[ur, "*"] \arrow[dr, "*"'] & & \Rightarrow \exists\, P \\
 & N \arrow[ur, "*"'] &
\end{tikzcd}$$

Fact 18.8 *Any confluent relation is also semi-confluent.*

■

Reduction: Church-Rosser

Definition 18.9 $\longrightarrow$ *is* **Church-Rosser** *if for all* L, M,

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

which we denote by

$$\begin{tikzcd}
L \arrow[rr, leftrightarrow, "*"] \arrow[dr, "*"'] & \Downarrow & M \arrow[dl, "*"] \\
& \exists P &
\end{tikzcd}$$

Equivalence Characterization

Lemma 18.10

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*

$$L \equiv_\alpha M_0 \longrightarrow / \longleftarrow M_1 \longrightarrow / \longleftarrow \cdots \longrightarrow / \longleftarrow M_m \equiv_\alpha M \qquad (18)$$

Proof of lemma 18.10

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" (18) implies $L \overset{*}{\longleftrightarrow} M$ by transitivity. For the other part use induction on the length of the proof.

QED

20: Confluence Characterization

The Church-Rosser Property

Confluence and Church-Rosser

Lemma 19.1 *Every confluent relation is also semi-confluent*

■

Theorem 19.2 *The following statements are equivalent for any reduction relation* $\longrightarrow$.

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

Proof of theorem 19.2

Proof: (1 ⇒ 2) Assume $\longrightarrow$ is Church-Rosser and let

$$N \;{}^*\!\!\longleftarrow L \longrightarrow^* M$$

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

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

which implies that it is confluent.

(2 ⇒ 1) Assume $\longrightarrow$ is confluent and let $L \overset{*}{\longleftrightarrow} M$. We proceed by induction on the length of the chain (18).

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

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

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.

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

<u>*Case* $M_k \longrightarrow M$</u>. Then by the induction hypothesis and semi-confluence we have

$$\begin{tikzcd}
L \arrow[rr, leftrightarrow, "*"] \arrow[dr, "*"'] & {} \arrow[d, phantom, "\Downarrow"] & M_k \arrow[dl, "*"] \arrow[dr] \arrow[dd, phantom, "\Downarrow"] & \\
& \exists Q \arrow[dr, "*"'] & & M \arrow[dl, "*"] \\
& & \exists P &
\end{tikzcd}$$

which proves the claim.

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

$$\begin{tikzcd}
L \arrow[rr, leftrightarrow, "*"] \arrow[dr, "*"'] & & M_k \arrow[dl, "*"] & M \arrow[l] \\
& \exists P & &
\end{tikzcd}$$

QED

Lemma 19.3 *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

$$\begin{tikzcd}
L \arrow[r] \arrow[d] & N_1 \arrow[r] \arrow[d] & N_2 \arrow[r] \arrow[d] & \cdots \arrow[r] & N_n \equiv M \arrow[d] \\
M \arrow[r] & P_1 \arrow[r] & P_2 \arrow[r] & \cdots \arrow[r] & P_n
\end{tikzcd}$$

QED

From lemma 19.3 and theorem 19.2 we have the following theorem.

Theorem 19.4 *If a terminating relation is locally confluent then it is confluent.*

Proof:

$\longrightarrow$ on Λ is given to be terminating and locally confluent. We need to show that it is confluent. That is for any L, we are given that

1. there is no infinite sequence of reductions of L, i.e. every maximal sequence of reductions of L is of length n for some $n \geq 0$.
2.
$$N_1 \,{}^1\!\!\longleftarrow L \longrightarrow^1 M_1 \Rightarrow \exists P : M_1 \longrightarrow^* P \,{}^*\!\!\longleftarrow N_1 \qquad (19)$$

We need to show for any term L that

$$N \overset{*}{\longleftarrow} L \longrightarrow^* M \Rightarrow \exists S : M \longrightarrow^* S \overset{*}{\longleftarrow} N \tag{20}$$

Let L be any term. Consider the graph $G(L) = \langle \Gamma(L), \longrightarrow^1 \rangle$ such that $\Gamma(L) = \{M \mid L \longrightarrow^* M\}$. Since $\longrightarrow$ is a terminating reduction

Fact 19.5 *The graph $G(L)$ is acyclic for any term L.*

If $G(L)$ is not acyclic, there must be a cycle of length $k > 0$ such that $M_0 \longrightarrow^1 M_1 \longrightarrow^1 \cdots \longrightarrow^1 M_{k-1} \longrightarrow^1 M_0$ which implies there is also an infinite reduction sequence of the form $L \longrightarrow^* M_0 \longrightarrow^k M_0 \longrightarrow^k \cdots$ which is impossible.

Since there are only a finite number of sub-terms of L that may be reduced under $\longrightarrow$, for each L there is a maximum number $p \geq 0$, which is the length of the longest reduction sequence.

Fact 19.6 *For every $M \in \Gamma(L)$,*

1. *$G(M)$ is a sub-graph of $G(L)$ and*
2. *For every $M \in \Gamma(L) - \{L\}$, the length of the longest reduction sequence of M is less than p.*

We proceed by induction on p.

Basis. $p = 0$. Then $\Gamma(L) = \{L\}$ and there are no reductions possible, so it is trivially confluent.

Induction Hypothesis (*IH*).

For any L whose longest reduction sequence is of length k, $0 \leq k < p$, property (20) holds.

Induction Step. Assume L is a term whose longest reduction sequence is of length $p > 0$. Also assume $N \overset{*}{\longleftarrow} L \longrightarrow^* M$ i.e. $\exists m, n \geq 0 : N \overset{n}{\longleftarrow} L \longrightarrow^m M$.

Case $m = 0$. If $m = 0$ then $M \equiv_\alpha L$ and hence $S \equiv_\alpha N$.

Case $n = 0$. Then $N \equiv_\alpha L$ and we have $S \equiv_\alpha M$.

Figure 1: Case $m > 0$ and $n > 0$

$$\begin{tikzcd}
& & M \arrow[dr, "*"] & & \\
& M_1 \arrow[ur, "*"] \arrow[dr, "*"] & & Q \arrow[dr, "*"] & \\
L \arrow[ur, "1"] \arrow[dr, "1"'] & & P \arrow[ur, "*"] \arrow[dr, "*"] & & S \\
& N_1 \arrow[ur, "*"] \arrow[dr] & & R \arrow[ur, "*"] & \\
& & N \arrow[ur, "*"] & &
\end{tikzcd}$$

Case $m, n > 0$. Then consider M_1 and N_1 such that

$$N \overset{*}{\longleftarrow} N_1 \overset{1}{\longleftarrow} L \longrightarrow^1 M_1 \longrightarrow^* M \tag{21}$$

See figure (1). By (19), $\exists P : M_1 \longrightarrow^* P \overset{*}{\longleftarrow} N_1$. Clearly $M_1, N_1, P \in \Gamma(L) - \{L\}$. Hence by fact 19.6, $G(M_1)$, $G(N_1)$ and $G(P)$ are all sub-graphs of $G(L)$ and all their reduction sequences are of length smaller than p. Hence by induction hypothesis, we get

$$P \overset{*}{\longleftarrow} M_1 \longrightarrow^* M \Rightarrow \exists Q : M \longrightarrow^* Q \overset{*}{\longleftarrow} P \tag{22}$$

and

$$N \overset{*}{\longleftarrow} N_1 \longrightarrow^* P \Rightarrow \exists R : P \longrightarrow^* R \overset{*}{\longleftarrow} N \tag{23}$$

But by (22) and (23) and the induction hypothesis we have

$$R \overset{*}{\longleftarrow} P \longrightarrow^* Q \Rightarrow \exists S : Q \longrightarrow^* S \overset{*}{\longleftarrow} R \tag{24}$$

Combining (24) with (21), (22) and (23) we get

$$N \overset{*}{\longleftarrow} L \longrightarrow^* M \Rightarrow \exists S : M \longrightarrow^* S \overset{*}{\longleftarrow} N \tag{25}$$

QED

Theorem 19.7 *If a terminating relation is locally confluent then it is Church-Rosser.*

Proof: Follows from theorem 19.4 and theorem 19.2 QED

19: The Church-Rosser Property

Parallel Beta Reduction

Definition 20.1 *The* **parallel-β** *or* $\|\beta$ *reduction is the smallest relation for which the following rules hold.*

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

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

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

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

Parallel Beta: The Diamond Property

Lemma 20.2

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 20.3 $\longrightarrow^1_{||\beta}$ *has the diamond property.*

Proof of theorem 20.3

Proof: We need to prove for all L

$$N \;{}^{1}_{\|\beta}\!\!\longleftarrow L \longrightarrow^{1}_{\|\beta} M \Rightarrow \exists P : N \longrightarrow^{1}_{\|\beta} P \;{}^{1}_{\|\beta}\!\!\longleftarrow M$$

We prove this by induction on the structure of L and a case analysis of the rule applied in definition 20.1.

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 $N_1 \;{}^{1}_{\|\beta}\!\!\longleftarrow L_1 \longrightarrow^{1}_{\|\beta} M_1$. By the induction hypothesis we have

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

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 $N \;{}^{1}_{\|\beta}\!\!\longleftarrow L \longrightarrow^{1}_{\|\beta} M$. Clearly then there exist M_1, M_2, N_1, N_2 such that $N_1 \;{}^{1}_{\|\beta}\!\!\longleftarrow L_1 \longrightarrow^{1}_{\|\beta} M_1$ and $N_2 \;{}^{1}_{\|\beta}\!\!\longleftarrow L_2 \longrightarrow^{1}_{\|\beta} M_2$. Again by the induction hypothesis, we have

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

and

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

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

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

for some M_1, M_2, N_1, N_2 such that

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

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 the last part of lemma 20.2 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

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

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 20.4 $\longrightarrow^{1}_{||\beta}$ *is confluent.*

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

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

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 20.3 to complete the rectangle

$$\begin{tikzcd}
L \arrow[r, "1", "||\beta"'] \arrow[d, "||\beta"', "1"] & M_1 \arrow[r, "1", "||\beta"'] \arrow[d, "||\beta"', "1"] & M_2 \arrow[r, "1", "||\beta"'] \arrow[d, "||\beta"', "1"] & \cdots \arrow[r, "1", "||\beta"'] & M_m \equiv M \arrow[d, "||\beta"', "1"] \\
N_1 \arrow[r, "1", "||\beta"'] \arrow[d, "||\beta"', "1"] & P_{11} \arrow[r, "1", "||\beta"'] \arrow[d, "||\beta"', "1"] & P_{12} \arrow[r, "1", "||\beta"'] \arrow[d, "||\beta"', "1"] & \cdots \arrow[r, "1", "||\beta"'] & P_{1m} \arrow[d, "||\beta"', "1"] \\
\vdots \arrow[d, "||\beta"', "1"] & \vdots \arrow[d, "||\beta"', "1"] & \vdots \arrow[d, "||\beta"', "1"] & \cdots & \vdots \arrow[d, "||\beta"', "1"] \\
N_n \arrow[r, "1", "||\beta"'] & P_{n1} \arrow[r, "1", "||\beta"'] & P_{n2} \arrow[r, "1", "||\beta"'] & \cdots \arrow[r, "1", "||\beta"'] & P_{nm} \equiv P
\end{tikzcd}$$

QED

■

Corollary 20.5 $\longrightarrow^1_\beta$ *is confluent.*

Proof: Since $\longrightarrow^*_\beta = \longrightarrow^*_{\|\beta}$ it follows from theorem 20.4 that $\longrightarrow^1_\beta$ is confluent. QED

Corollary 20.6 *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 20.5 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 20.7 $\longrightarrow^1_\beta$ *is Church-Rosser.*

Proof: Follows from corollary 20.5 and theorem 19.2. QED

An Applied Lambda-Calculus

A Simple Language of Terms: FL0

Let X be an infinite collection of variables (names). Consider the language (actually a collection of abstract syntax trees) of terms $T_\Omega(X)$ defined by the following constructors (along with their intended meanings).

Construct	Arity	Informal Meaning
`Z`	0	The number **0**
`T`	0	The truth value **true**
`F`	0	The truth value **false**
`P`	1	The **predecessor** function on numbers
`S`	1	The **successor** function on numbers
`ITE`	3	The **if-then-else** construct (on numbers and truth values)
`IZ`	1	The **is-zero** predicate on numbers
`GTZ`	1	The **greater-than-zero** predicate on numbers

FL(X): Language, Datatype or Instruction Set?

The set of terms $T_\Omega(X)$ may be alternatively defined by the BNF:

$$t ::= x \in X \mid \mathtt{Z} \mid (\mathtt{P}\ t) \mid (\mathtt{S}\ t) \mid \mathtt{T} \mid \mathtt{F} \mid (\mathtt{ITE}\ \langle t, t_1, t_0 \rangle) \mid (\mathtt{IZ}\ t) \mid (\mathtt{GTZ}\ t) \tag{26}$$

- It could be thought of as a user-defined data-type
- It could be thought of as the instruction-set of a particularly simple hardware machine.
- It could be thought of as a simple functional programming language without recursion.
- It is a language with two simple types of data: integers and booleans
- Notice that the constructor $(\mathtt{ITE}\ \langle t, t_1, t_0 \rangle)$ is overloaded.

Extending the language

To make this simple language safe we require

Type-checking : to ensure that arbitrary expressions are not mixed in ways they are not "intended" to be used. For example

- t cannot be a boolean expression in $\mathtt{S}(t)$, $\mathtt{P}(t)$, $\mathtt{IZ}(t)$ and $\mathtt{GTZ}(t)$
- $\mathtt{ITE}(t, t_1, t_0)$ may be used as a conditional expression for both integers and booleans, but t needs to be a boolean and either both t_1 and t_0 are integer expressions or both are boolean expressions.

Functions : To be a useful programming language we need to be able to define functions.

Recursion : to be able to define complex functions in a well-typed fashion. Recursion should also be well-typed

Typing FL Expressions

We have only two types of objects – integers and booleans which we represent by $\underline{\texttt{int}}$ and $\underline{\texttt{bool}}$ respectively. We then have the following elementary typing annotations for the expressions, which may be obtained by pattern matching.

1. $\mathtt{Z} : \underline{\texttt{int}}$
2. $\mathtt{T} : \underline{\texttt{bool}}$
3. $\mathtt{F} : \underline{\texttt{bool}}$
4. $\mathtt{S} : \underline{\texttt{int}} \rightarrow \underline{\texttt{int}}$
5. $\mathtt{P} : \underline{\texttt{int}} \rightarrow \underline{\texttt{int}}$
6. $\mathtt{IZ} : \underline{\texttt{int}} \rightarrow \underline{\texttt{bool}}$
7. $\mathtt{GTZ} : \underline{\texttt{int}} \rightarrow \underline{\texttt{bool}}$
8. $\mathtt{ITEI} : \underline{\texttt{bool}} * \underline{\texttt{int}} * \underline{\texttt{int}} \rightarrow \underline{\texttt{int}}$
9. $\mathtt{ITEB} : \underline{\texttt{bool}} * \underline{\texttt{bool}} * \underline{\texttt{bool}} \rightarrow \underline{\texttt{bool}}$

Λ+FL(X): The Power of Functions

To make the language powerful we require the ability to define functions, both non-recursive and recursive. We define an applied lambda-calculus of lambda terms $\Lambda_\Omega(X)$ over this set of terms as follows:

$$L, M, N ::= t \in T_\Omega(X) \;\mid\; \lambda x[L] \;\mid\; (L\ M) \tag{27}$$

This is two-level grammar combining the term grammar (26) with λ-abstraction and λ-application.
While this makes it possible to use the operators of $T_\Omega(X)$ as part of functions (λ-expressions), it does not allow us to use the operators of $T_\Omega(X)$ outside of λ-abstractions and λ-applications.

Λ+FL(X): Lack of Higher-order Power?

Example 21.1 *The grammar (27) does not allow us to define expressions such as the following:*

1. *the successor of the result of an application* $(\texttt{S}\ (L\ M))$
2. *higher order conditionals e.g.* $\lambda x[(\texttt{ITE}\ \langle (L\ x), (M\ x), (N\ x)\rangle)]$ *where* $(L\ x)$ *yields a boolean value for an argument of the appropriate type.*
3. *In general, it does not allow the constructors to be applied to* λ*-expressions.*

So we extend the language by allowing a free intermixing of λ-terms and terms of the sub-language $T_\Omega(X)$.

$\Lambda_{FL}(X)$: Higher order functions

We need to *flatten* the grammar of (27) to allow λ-terms also to be used as arguments of the constructors of the term-grammar (26). The language of applied λ-terms (viz. $\Lambda_{\Omega}(X)$) now is defined by the grammar.

$$
\begin{aligned}
L, M, N ::= x \in X \;&\mid\; \mathtt{Z} \\
&\mid\; (\mathtt{P}\ L) \;\mid\; (\mathtt{S}\ L) \\
&\mid\; \mathtt{T} \;\mid\; \mathtt{F} \\
&\mid\; (\mathtt{IZ}\ L) \;\mid\; (\mathtt{GTZ}\ L) \\
&\mid\; (\mathtt{ITE}\ \langle L, M, N\rangle) \\
&\mid\; \lambda x[L] \;\mid\; (L\ M)
\end{aligned}
\tag{28}
$$

The Normal forms for Integers

We need reduction rules to simplify (non-recursive) expressions.

Zero . Z is the unique representation of the number 0 and every integer expression that is equal to 0 must be reducible to Z.

Positive integers . Each positive integer k is uniquely represented by the expression $\mathtt{S}^k(\mathtt{Z})$ where the super-script k denotes a k-fold application of S.

Negative integers . Each negative integer $-k$ is uniquely represented by the expression $\mathtt{P}^k(\mathtt{Z})$ where the super-script k denotes a k-fold application of P.

δ-rules

$$(\mathrm{P}\ (\mathrm{S}\ x)) \longrightarrow_\delta x \tag{29}$$

$$(\mathrm{S}\ (\mathrm{P}\ x)) \longrightarrow_\delta x \tag{30}$$

Reduction Rules for Boolean Expressions

Pure Boolean Reductions . The constructs T and F are the normal forms for boolean values.

$$(\mathtt{ITE}\ \langle b, x, x\rangle) \longrightarrow_\delta x \tag{31}$$

$$(\mathtt{ITE}\ \langle \mathtt{T}, x, y\rangle) \longrightarrow_\delta x \tag{32}$$

$$(\mathtt{ITE}\ \langle \mathtt{F}, x, y\rangle) \longrightarrow_\delta y \tag{33}$$

Testing for zero .

$$(\mathtt{IZ}\ \mathtt{Z}) \longrightarrow_\delta \mathtt{T} \tag{34}$$

$$(\mathtt{IZ}\ (\mathtt{S}\ n)) \longrightarrow_\delta \mathtt{F}, \text{ where } (\mathtt{S}\ n) \text{ is a } \delta\text{-nf} \tag{35}$$

$$(\mathtt{IZ}\ (\mathtt{P}\ n)) \longrightarrow_\delta \mathtt{F}, \text{ where } (\mathtt{P}\ n) \text{ is a } \delta\text{-nf} \tag{36}$$

Testing for Positivity

$$\begin{aligned}
(\mathtt{GTZ}\ \mathtt{Z}) &\longrightarrow_\delta \mathtt{F} && (37)\\
(\mathtt{GTZ}\ (\mathtt{S}\ n)) &\longrightarrow_\delta \mathtt{T}, \text{ where } (\mathtt{S}\ n) \text{ is a } \delta\text{-nf} && (38)\\
(\mathtt{GTZ}\ (\mathtt{P}\ n)) &\longrightarrow_\delta \mathtt{F}, \text{ where } (\mathtt{P}\ n) \text{ is a } \delta\text{-nf} && (39)
\end{aligned}$$

Other Non-recursive Operators

We may "program" the other boolean operations as follows:

$$\begin{aligned}
\text{NOT} &\overset{df}{=} \lambda x[\mathtt{ITE}\ \langle x, \mathtt{F}, \mathtt{T}\rangle] \\
\text{AND} &\overset{df}{=} \lambda\langle x, y\rangle[\mathtt{ITE}\ \langle x, y, \mathtt{F}\rangle] \\
\text{OR} &\overset{df}{=} \lambda\langle x, y\rangle[\mathtt{ITE}\ \langle x, \mathtt{T}, y\rangle]
\end{aligned}$$

We may also "program" the other integer comparison operations as follows:

$$\begin{aligned}
\text{GEZ} &\overset{df}{=} \lambda x[\mathtt{OR}\ \langle(\mathtt{IZ}\ x), (\mathtt{GTZ}\ x)\rangle] \\
\text{LTZ} &\overset{df}{=} \lambda x[\mathtt{NOT}\ (\mathtt{GEZ}\ x)] \\
\text{LEZ} &\overset{df}{=} \lambda x[\mathtt{OR}\ \langle(\mathtt{IZ}x), (\mathtt{LTZ}\ x)\rangle]
\end{aligned}$$

Recursion in the Applied Lambda-calculus

The full power of a programming language will not be realised without a recursion mechanism. The untyped lambda-calculus has "paradoxical combinators" which behave like recursion operators upto $=_\beta$.

Definition 21.2 *A combinator* Y *is called a* **fixed-point combinator** *if for every lambda term* L, $\boxed{(\mathsf{Y}\ L) =_\beta (L\ (\mathsf{Y}\ L))}$

Curry's Y combinator ($\mathsf{Y_C}$)

$$\mathsf{Y_C} \overset{df}{=} \lambda f[(\mathsf{C}\ \mathsf{C})] \text{ where } \mathsf{C} \overset{df}{=} \lambda x[(f\ (x\ x))]$$

Turing's Y combinator ($\mathsf{Y_T}$)

$$\mathsf{Y_T} \overset{df}{=} (\mathsf{T}\ \mathsf{T}) \text{ where } \mathsf{T} \overset{df}{=} \lambda y\ x[(x\ (y\ y\ x\ ))]$$

But the various Y combinators unfortunately will not satisfy any typing rules that we may define for the language, because they are all "self-applicative" in nature.

$\Lambda_{RecFL}(X)$: Adding Recursion

Instead it is more convenient to use the fixed-point property and define a new constructor with a δ-rule which satisfies the fixed-point property (definition 21.2).

We extend the language FL with a new constructor

$$L ::= \ldots \quad | \quad (\mathrm{REC}\ L)$$

and add the fixed point property as a δ-rule

$$\boxed{(\mathrm{REC}\ L) \longrightarrow_{\delta} (L\ (\mathrm{REC}\ L)) \qquad (40)}$$

Recursion Example: Addition

Consider addition on integers as a binary operation to be defined in this language. We use the following properties of addition on the integers to define it by induction on the first argument.

$$x + y = \begin{cases} y & \text{if } x = 0 \\ (x-1) + (y+1) & \text{if } x > 0 \\ (x+1) + (y-1) & \text{if } x < 0 \end{cases} \tag{41}$$

Using the constructors of $\Lambda_{RecFL}(X)$ we require that any (curried) definition of addition on numbers should be a solution to the following equation in $\Lambda_{RecFL}(X)$ for all (integer) expression values of x and y.

$$(plusc\ x\ y) =_{\beta\delta} \mathtt{ITE}\ \langle(\mathtt{IZ}\ x), y, \mathtt{ITE}\ \langle(\mathtt{GTZ}\ x), (plusc\ (\mathtt{P}\ x)\ (\mathtt{S}\ y)), (plusc\ (\mathtt{S}\ x)\ (\mathtt{P}\ y))\rangle\rangle \tag{42}$$

Equation (42) may be rewritten using abstraction as follows:

$$plusc =_{\beta\delta} \lambda x[\lambda y[\mathtt{ITE}\ \langle(\mathtt{IZ}\ x), y, \mathtt{ITE}\ \langle(\mathtt{GTZ}\ x), (plusc\ (\mathtt{P}\ x)\ (\mathtt{S}\ y)), (plusc\ (\mathtt{S}\ x)\ (\mathtt{P}\ y))\rangle\rangle]] \tag{43}$$

We may think of equation (43) as an equation to be solved in the unknown variable *plusc*.

Consider the (applied) λ-term obtained from the right-hand-side of equation (43) by simply abstracting the unknown *plusc*.

$$\boxed{\mathsf{addc} \stackrel{df}{=} \lambda f[\lambda x\ y[\mathtt{ITE}\ \langle(\mathtt{IZ}\ x), y, \mathtt{ITE}\ \langle(\mathtt{GTZ}\ x), (f\ (\mathtt{P}\ x)\ (\mathtt{S}\ y)), (f\ (\mathtt{S}\ x)\ (\mathtt{P}\ y))\rangle\rangle]] \qquad (44)}$$

Claim 21.3

$$(\mathtt{REC}\ \mathsf{addc}) \longrightarrow_{\delta} (\mathsf{addc}\ (\mathtt{REC}\ \mathsf{addc})) \tag{45}$$

and hence

$$(\mathtt{REC}\ \mathsf{addc}) =_{\beta\delta} (\mathsf{addc}\ (\mathtt{REC}\ \mathsf{addc})) \tag{46}$$

Claim 21.4 $(\mathtt{REC}\ \mathsf{addc})$ *satisfies exactly the equation (43). That is*

$$((\mathtt{REC}\ \mathsf{addc})\ x\ y) =_{\beta\delta} \mathtt{ITE}\ \langle(\mathtt{IZ}\ x), y, \mathtt{ITE}\ \langle(\mathtt{GTZ}\ x), ((\mathtt{REC}\ \mathsf{addc})\ (\mathtt{P}\ x)\ (\mathtt{S}\ y)), ((\mathtt{REC}\ \mathsf{addc})\ (\mathtt{S}\ x)\ (\mathtt{P}\ y))\rangle\rangle \tag{47}$$

Hence we may regard $(\mathtt{REC}\ \mathsf{addc})$ where addc is defined by right-hand-side of definition (44) as the required solution to the equation (42) in which *plusc* is an unknown.

The abstraction shown in (44) and the claims (21.3) and (21.4) simply go to show that $M \equiv_\alpha \lambda f[\{f/z\}L]$ is a solution to the equation $z =_{\beta\delta} L$, whenever such a solution does exist. Further, the claims also show that we may "unfold" the recursion (on demand) by simply performing the substitution $\{L/z\}L$ for each free occurrence of z within L.

Exercise 21.1

1. *Prove that the relation $\longrightarrow_\delta$ is confluent.*
2. *The language FL does not have any operators that take boolean arguments and yields integer values. Define a standard conversion function* B2I *which maps the value* `F` *to* `Z` *and* `T` *to* `(S Z)`.
3. *Prove that* Y_C *and* Y_T *are both fixed-point combinators.*
4. *Using the combinator* add *and the other constructs of* $\Lambda_\Sigma(X)$ *to*
   (a) *define the equation for products of numbers in the language.*
   (b) *define the multiplication operation* mult *on integers and prove that it satisfies the equation(s) for products.*
5. *The equation (41) is defined conditionally. However the following is equally valid for all integer values x and y.*

$$x + y = (x - 1) + (y + 1) \tag{48}$$

   (a) *Follow the steps used in the construction of* addc *to define a new applied* addc′ *that instead uses equation (48).*
   (b) *Is* $(\mathtt{REC}\ \mathsf{addc'}) =_{\beta\delta} (\mathsf{addc'}\ (\mathtt{REC}\ \mathsf{addc'}))$?
   (c) *Is* $\mathsf{addc} =_{\beta\delta} \mathsf{addc'}$?
   (d) *Is* $(\mathtt{REC}\ \mathsf{addc}) =_{\beta\delta} (\mathtt{REC}\ \mathsf{addc'})$?
6. *The function* addc *was defined in curried form. Use the pairing function in the untyped λ-calculus, to define*
   (a) *addition and multiplication as binary functions independently of the existing functions.*
   (b) *the binary 'curry' function which takes a binary function and its arguments and creates a curried version of the binary function.*

Typing FL expressions

We have already seen that the simple language FL has

- two kinds of expressions: integer expressions and boolean expressions,
- there are also constructors which take integer expressions as arguments and yield boolean values
- there are also function types which allow various kinds of functions to be defined on boolean expressions and integer expressions.

The Need for typing in FL

- A type is an important *attribute* of any variable, constant or expression, since every such object can only be used in certain kinds of expressions.
- Besides the need for type-checking rules on $T_\Omega(X)$ to prevent illegal constructor operations,
  - rules are necessary to ensure that λ-applications occur only between terms of appropriate types in order to remain meaningful.
  - rules are necessary to ensure that all terms have clearly defined types at compile-time so that there are no run-time type violations.

TL: A Simple Language of Types

Consider the following language of types (in fully parenthesized form) defined over an infinite collection $'a \in TV$ of type variables. We also have two type constants `int` and `bool`.

$$\sigma, \tau ::= \texttt{int} \mid \texttt{bool} \mid {'a} \in TV \mid (\sigma * \tau) \mid (\sigma \rightarrow \tau)$$

Notes.

- `int` and `bool` are *type constants*.
- $*$ is the product operation on types and
- $\rightarrow$ is the function operator on types.
- We require $*$ because of the possibility of defining functions of various kinds of arities in $\Lambda_\Omega(X)$.
- **Precedence.** We assume $*$ has a higher precedence than $\rightarrow$.
- **Associativity.** $\rightarrow$ is *right* associative whereas $*$ is *left* associative.
- In any type expression τ, $TVar(\tau)$ is the set of type variables

Type-inference Rules: Infrastructure

The question of assigning types to complicated expressions which may have variables in them still remains to be addressed.

Type inferencing. Can be done using type assignment rules, by a recursive travel of the abstract syntax tree.

Free variables (names) are already present in the *environment* (symbol table).

Constants and Constructors. May have their types either pre-defined or there may be axioms assigning them types.

Bound variables. May be necessary to introduce "fresh" type variables in the environment.

Type Inferencing: Infrastructure

The elementary typing previously defined for the elementary expressions of FL does not suffice

1. in the presence of λ abstraction and application, which allow for higher-order functions to be defined
2. in the presence of polymorphism, especially when we do not want to unnecessarily decorate expressions with their types.

Type Assignment: Infrastructure

- Assume Γ is the environment[a] (an association list) which may be looked up to determine the types of individual names. For each variable $x \in X$, $\Gamma(x)$ yields the type of x i.e. $\Gamma(x) = \sigma$ if $x : \sigma \in \Gamma$.
- For each (sub-)expression in FL we define a set C of *type constraints* of the form $\sigma = \tau$, where T is the set of type variables used in C.
- The type constraints are defined by induction on the structure of the expressions in the language FL.
- The expressions of FL could have free variables. The type of the expression would then depend on the types assigned to the free variables. This is a simple kind of polymorphism.
- It may be necessary to generate new type variables as and when required during the process of inferencing and assignment.

[a] usually a part of the symbol table

Constraint Typing Relation

Definition 22.1 *For each term $L \in \Lambda_{\Sigma}(X)$ the* **constraint typing relation** *is of the form*

$$\Gamma \vdash L : \tau \ \ \triangleright_T \ \ C$$

where

- Γ *is called the* context[a] *and defines the stack of assumptions*[b] *that may be needed to assign a type (expression) to the (sub-)expression L.*
- τ *is the type(-expression) assigned to L*
- C *is the set of constraints*
- T *is the set of "fresh" type variables used in the (sub-)derivations*

[a] usually in the symbol table
[b] including new type variables

Typing axioms: Basic

The following axioms (c.f Typing FL Expressions) may be applied during the scanning and parsing phases of the compiler to assign types to the individual tokens.

$$\textbf{Z}\ \frac{}{\Gamma \vdash \mathtt{Z} : \underline{\mathtt{int}} \ \rhd_{\emptyset}\ \emptyset} \qquad \textbf{T}\ \frac{}{\Gamma \vdash \mathtt{T} : \underline{\mathtt{bool}} \ \rhd_{\emptyset}\ \emptyset} \qquad \textbf{F}\ \frac{}{\Gamma \vdash \mathtt{F} : \underline{\mathtt{bool}} \ \rhd_{\emptyset}\ \emptyset}$$

$$\textbf{S}\ \frac{}{\Gamma \vdash \mathtt{S} : \underline{\mathtt{int}}\rightarrow\underline{\mathtt{int}} \ \rhd_{\emptyset}\ \emptyset} \qquad \textbf{P}\ \frac{}{\Gamma \vdash \mathtt{P} : \underline{\mathtt{int}}\rightarrow\underline{\mathtt{int}} \ \rhd_{\emptyset}\ \emptyset}$$

$$\textbf{IZ}\ \frac{}{\Gamma \vdash \mathtt{IZ} : \underline{\mathtt{int}}\rightarrow\underline{\mathtt{bool}} \ \rhd_{\emptyset}\ \emptyset} \qquad \textbf{GTZ}\ \frac{}{\Gamma \vdash \mathtt{GTZ} : \underline{\mathtt{int}}\rightarrow\underline{\mathtt{bool}} \ \rhd_{\emptyset}\ \emptyset}$$

$$\textbf{ITEI}\ \frac{}{\Gamma \vdash \mathtt{ITE} : \underline{\mathtt{bool}}*\underline{\mathtt{int}}*\underline{\mathtt{int}}\rightarrow\underline{\mathtt{int}} \ \rhd_{\emptyset}\ \emptyset}$$

$$\textbf{ITEB}\ \frac{}{\Gamma \vdash \mathtt{ITE} : \underline{\mathtt{bool}}*\underline{\mathtt{bool}}*\underline{\mathtt{bool}}\rightarrow\underline{\mathtt{bool}} \ \rhd_{\emptyset}\ \emptyset}$$

Notice that the constructor `ITE` is *overloaded* and actually is two constructors `ITEI` and `ITEB`. Which constructor is actually used will depend on the context and the type-inferencing mechanism.

Type Rules for FL: 2

$$\textbf{Var}\quad \frac{}{\Gamma \vdash x : \Gamma(x) \;\rhd_{\emptyset}\; \emptyset}$$

$$\textbf{Abs}\quad \frac{\Gamma, x:\sigma \vdash L : \tau \;\rhd_{T}\; C}{\Gamma \vdash \lambda x[L] : \sigma \rightarrow \tau \;\rhd_{T}\; C}$$

$$\textbf{App}\quad \frac{\begin{array}{c}\Gamma \vdash L : \sigma \;\rhd_{T_1}\; C_1 \\ \Gamma \vdash M : \tau \;\rhd_{T_2}\; C_2\end{array}}{\Gamma \vdash (L\; M) : {'a} \;\rhd_{T'}\; C'}\quad \textbf{(Conditions 1. and 2.)}$$

where

- **Condition 1.** $T_1 \cap T_2 = T_1 \cap TVar(\tau) = T_2 \cap TVar(\sigma) = \emptyset$
  Condition 2. ${'a} \notin T_1 \cup T_2 \cup TVar(\sigma) \cup TVar(\tau) \cup TVar(C_1) \cup TVar(C_2)$.
- $T' = T_1 \cup T_2 \cup \{{'a}\}$
- $C' = C_1 \cup C_2 \cup \{\sigma = \tau \rightarrow {'a}\}$

Example 22.2 *Consider the following simple combinator* $\lambda x[\lambda y[\lambda z[(x\ (y\ z))]]]$ *which defines the function composition operator. Since there are three bound variables* x, y *and* z *we begin with an initial assumption* $\Gamma = x : {'a}, y : {'b}, z : {'c}$ *which assign arbitrary types to the bound variables, represented by the type variables* $'a$, $'b$ *and* $'c$ *respectively. Note however, that since it has no free variables, its type does not depend on the types of any variables. We expect that at the end of the proof there would be no assumptions.Our inference for the type of the combinator then proceeds as follows.*

1. $x : {'a}, y : {'b}, z : {'c} \vdash x : {'a} \ \rhd_{\emptyset}\ \emptyset$ (**Var**)
2. $x : {'a}, y : {'b}, z : {'c} \vdash y : {'b} \ \rhd_{\emptyset}\ \emptyset$ (**Var**)
3. $x : {'a}, y : {'b}, z : {'c} \vdash z : {'c} \ \rhd_{\emptyset}\ \emptyset$ (**Var**)
4. $x : {'a}, y : {'b}, z : {'c} \vdash (y\ z) : {'d} \ \rhd_{\{'d\}}\ \{'b = {'c} \to {'d}\}$ (**App**)
5. $x : {'a}, y : {'b}, z : {'c} \vdash (x\ (y\ z)) : {'e} \ \rhd_{\{'d, 'e\}}\ \{'b = {'c} \to {'d},\ 'a = {'d} \to {'e}\}$ (**App**)
6. $x : {'a}, y : {'b} \vdash \lambda z[(x\ (y\ z))] : {'c} \to {'e} \ \rhd_{\{'d, 'e\}}\ \{'b = {'c} \to {'d},\ 'a = {'d} \to {'e}\}$ (**Abs**)
7. $x : {'a} \vdash \lambda x[\lambda y[\lambda z[(x\ (y\ z))]]] : {'b} \to {'c} \to {'e} \ \rhd_{\{'d, 'e\}}\ \{'b = {'c} \to {'d},\ 'a = {'d} \to {'e}\}$ (**Abs**)
8. $\vdash \lambda x[\lambda y[\lambda z[(x\ (y\ z))]]] : {'a} \to {'b} \to {'c} \to {'e} \ \rhd_{\{'d, 'e\}}\ \{'b = {'c} \to {'d},\ 'a = {'d} \to {'e}\}$ (**Abs**)

Hence $\lambda x[\lambda y[\lambda z[(x\ (y\ z))]]] : {'a} \to {'b} \to {'c} \to {'e}$ *subject to the constraints given by* $\{'b = {'c} \to {'d},\ 'a = {'d} \to {'e}\}$ *which yields* $\lambda x[\lambda y[\lambda z[(x\ (y\ z))]]] : ({'d} \to {'e}) \to ({'c} \to {'d}) \to {'c} \to {'e}$

Principal Type Schemes

Definition 22.3 *A solution for* $\Gamma \vdash L : \tau \ \triangleright_T \ C$ *is a pair* $\langle S, \sigma \rangle$ *where* S *is a substitution of type variables in* τ *such that* $S(\tau) = \sigma$*.*

- The rules yield a *principal type scheme* for each well-typed applied λ-term.
- The term is *ill-typed* if there is no solution that satisfies the constraints.
- Any substitution of the type variables which satisfies the constraints C is an instance of the most general polymorphic type that may be assigned to the term.

Exercise 22.1

1. *The language has several constructors which behave like functions. Derive the following rules for terms in* $T_\Omega(X)$ *from the basic typing axioms and the rule* **App**.

$$\textbf{Sx}\quad \frac{\Gamma \vdash t : \tau \ \rhd_T\ C}{\Gamma \vdash (\mathtt{S}\ t) : \underline{\mathtt{int}} \ \rhd_T\ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\textbf{Px}\quad \frac{\Gamma \vdash t : \tau \ \rhd_T\ C}{\Gamma \vdash (\mathtt{P}\ t) : \underline{\mathtt{int}} \ \rhd_T\ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\textbf{IZx}\quad \frac{\Gamma \vdash t : \tau \ \rhd_T\ C}{\Gamma \vdash (\mathtt{IZ}\ t) : \underline{\mathtt{bool}} \ \rhd_T\ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\textbf{GTZx}\quad \frac{\Gamma \vdash t : \tau \ \rhd_T\ C}{\Gamma \vdash (\mathtt{GTZ}\ t) : \underline{\mathtt{bool}} \ \rhd_T\ C \cup \{\tau = \underline{\mathtt{int}}\}}$$

$$\textbf{ITEx}\quad \frac{\Gamma \vdash t : \sigma \ \rhd_T\ C \qquad \Gamma \vdash t_1 : \tau \ \rhd_{T_1}\ C_1 \qquad \Gamma \vdash t_0 : \upsilon \ \rhd_{T_0}\ C_0}{\Gamma \vdash (\mathtt{ITE}\ \langle t, t_1, t_0\rangle) : \tau \ \rhd_{T'}\ C'} \quad (T \cap T_1 = T_1 \cap T_0 = T_0 \cap T = \emptyset)$$

where $T' = T \cup T_1 \cup T_0$ *and* $C' = C \cup C_1 \cup C_0 \cup \{\sigma = \underline{\mathtt{bool}}, \tau = \upsilon\}$

2. *Use the rules to define the type of the combinators* K *and* S*?*

3. *How would you define a type assignment for the recursive function* addc *defined by equation (44).*

4. *Prove that the terms,* $\omega = \lambda x[(x\ x)]$ *and* $\Omega = (\omega\ \omega)$ *are ill-typed.*

5. *Are the following well-typed or ill-typed? Prove your answer.*

   (a) (K S)

   (b) ((K S) ω)

   (c) (((S K) K) ω)

   (d) (ITE $\langle$(IZ x), T, (K x)$\rangle$)

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?