# Advanced Features & Applications

- Having promoted patterns to first-class status means that we have effectively separated the point of definition of patterns from the point where patterns are applied
- This allows for novel applications of patterns.

# Pattern Reuse

- The ability of reusing patterns frees a developer from having to retype the same pattern repeatedly in their code.
- The ability of reusing patterns makes code much more robust from a software engineering perspective
  - In software engineering it is frowned upon to explicitly repeat the same code in your program
  - A maintenance nightmare: if anything ever changes in the repeated code you will have to go through all the repeated instances manually and update them

# Pattern Reuse

```
function fact
    with 0 do
        1
    with (n:%integer) if n > 0 do
        n * fact (n-1).
    with (n:%integer) if n < 0 do
        throw Error("negative value").
end

function sign
    with 0 do
        1
    with (n:%integer) if n > 0 do
        1
    with (n:%integer) if n < 0 do
        -1
end
```

ln015/reuse1.ast

```
let Pos_Int = pattern (x:%integer) if x > 0.
let Neg_Int = pattern (x:%integer) if x < 0.

function fact
    with 0 do
        1
    with n:*Pos_Int do
        n * fact (n-1).
    with *Neg_Int do
        throw Error("negative value").
end

function sign
    with 0 do
        1
    with *Pos_Int do
        1
    with *Neg_Int do
        -1
end
```

ln015/reuse2.ast

# Pattern Factoring

- Patterns can become quite complex given that we can add
  - Conditionals with multiple terms
  - Nested structures such as lists of lists, tuples of lists, lists of tuples, etc.

- First-class patterns allow us to factor patterns into smaller manageable pieces.

# Pattern Factoring

- What exactly is the input structure to the function ‘fold’ – difficult to see…

```
function fold with (x if (x is %integer) or (x is %real) and (x > 0), y) do
    x*y
end
```

In015/factor1.ast

# Pattern Factoring

- …it is a pair where the first component is a positive scalar
  - Using first-class patterns let's us bring that to the forefront

```
let pos_scalar = pattern k if (k is %integer) or (k is %real) and (k > 0).

function fold with (x:*pos_scalar, y) do
    x*y
end
```

ln015/factor2.ast

# Patterns as Constraints

- The use of patterns as constraints is nothing new
- We have seen this before with statements such as,
  - let x : %integer = value.
- where we are not interested in the exact value the pattern %integer matches but just the fact that it matches an integer value rather than anything else.

# Patterns as Constraints

- The following pattern matches any scalar value between 1 and 9
  - let p = pattern k if k > 0 and k < 10.
- We can use this pattern as a constraint,
  - let x : *p = value.
- It works, BUT the pattern instantiates the variable k every time it matches
- …this can lead to difficult to trace bugs

# Patterns as Constraints

```
-- our constraint pattern
let p = pattern k if k in range 10.

-- a simple loop that creates a list of values
let out = [].
let k = 2.
for i in range 10 do
    if i is *p do
        out @append (k).
    end
end
-- should be out == [2,2,2,2,2,2,2,2,2,2]
-- but
assert (not (out == [2,2,2,2,2,2,2,2,2,2])).
-- and
assert (out == [0,1,2,3,4,5,6,7,8,9]).
```

ln015/constraint1a.ast

# Patterns as Constraints

```
-- constraint
let scalar = pattern k if (k is %integer) or (k is %real).

-- fold is a applied to a pair of scalar values
function fold with (x:*scalar, y:*scalar) do -- error: introduces a non-linearity in k
   x+y
end

assert (fold (1,2) == 3).
```

ln015/constraint2a.ast

# Patterns as Constraints

- We saw in each of the previous examples that the first-class pattern introduced an undesirable variable instantiation into the current scope of the program
- We can prevent that with the *scope operator* %[…]% in a first-class pattern
  - Any variable instantiated within the scope operator is not visible outside of the pattern

# Patterns as Constraints

```
-- our constraint pattern
let p = pattern %[k if k in range 10]%.

-- a simple loop that creates a list of values
let out = [].
let k = 2.
for i in range 10 do
    if i is *p do
        out @append (k).
    end
end

assert (out == [2,2,2,2,2,2,2,2,2,2]). -- succeeds!
```

ln015/constraint1b.ast

# Patterns as Constraints

```
-- constraint
let scalar = pattern %[k if (k is %integer) or (k is %real)]%.

-- fold is a applied to a pair of scalar values
function fold with (x:*scalar, y:*scalar) do
    x+y
end

assert (fold (1,2) == 3).
```

ln015/constraint2b.ast

# Managing Pattern Variable Bindings

- As we have seen: repeated first-class patterns lead to non-linearities
  - The scope operator allows us to manage this hiding the variables
- BUT, what if we want the variables of repeated first-class patterns to be bound into our current scope in some shape or form?
  - The scope operator allows us to selectively bind variables into our current scope

# Managing Pattern Variable Bindings

- Consider that we want to compute the dot product of two 2D vectors,
  - (x1,y1) • (x2,y2)
- Writing this as a function
  - dot ((x1,y1),(x2,y2))
- The function takes a pair of pairs, the inner pairs must be pairs of scalars in order for the dot operation to make sense

# Managing Pattern Variable Bindings

- First attempt without first-class patterns
- It’s a mess…the function definition becomes almost unreadable

☞ We can solve this by **pattern factoring** with first-class patterns

```
function dot with ((a1 if (a1 is %real) or (a1 is %integer), b1 if (b1 is %real) or (b1 is %integer)),
                   (a2 if (a2 is %real) or (a2 is %integer), b2 if (b2 is %real) or (b2 is %integer))) do
    a1*a2+b1*b2
end

assert (dot((1,0),(0,1)) == 0).
```

ln015/dot1.ast

# Managing Pattern Variable Bindings

ln015/dot2.ast

```
-- declare a pattern that matches scalar values
let Scalar = pattern %[p if (p is %integer) or (p is %real)]%.

-- declare a pattern that matches pairs of scalars
let Pair = pattern %[(x:*Scalar,y:*Scalar)]%.

-- compute the dot product of two pairs of scalars
function dot with (*Pair bind [x as a1, y as a2], *Pair bind [x as b1, y as b2]) do
    a1*b1 + a2*b2
end

-- define basis vectors of 2D space
let i1 = (1,0).
let i2 = (0,1).
-- the dot product of basis vector is always 0
assert (dot(i1,i2) == 0).
```

**Binding lists** applied to constraint patterns allow us to selectively bind variables into the current scope.