02157 Functional Programming

Lecture 3: Programming as a model-based activity

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- Syntax, semantics and pragmatics (briefly)
  - Overview of F#
  - Semantics of a function declaration
- Programming as a modelling activity
  - Type declarations (type abbreviations)
  - Cash register
  - Map colouring
- Program properties and property-based testing

Syntax, semantics and pragmatics

Programming languages: Syntax, Semantics and Pragmatics

- The syntax is concerned with the (notationally correct) grammatical structure of programs.

Programming languages: Syntax, Semantics and Pragmatics

- The syntax is concerned with the (notationally correct) grammatical structure of programs.

- The semantics is concerned with the meaning of syntactically correct programs.

Programming languages: Syntax, Semantics and Pragmatics

- The syntax is concerned with the (notationally correct) grammatical structure of programs.

- The semantics is concerned with the meaning of syntactically correct programs.

- The pragmatics is concerned with practical (adequate) application of language constructs in order to achieve certain objectives.

Programming languages: Syntax, Semantics and Pragmatics

- The syntax is concerned with the (notationally correct) grammatical structure of programs.
- The semantics is concerned with the meaning of syntactically correct programs.
- The pragmatics is concerned with practical (adequate) application of language constructs in order to achieve certain objectives.

A specification of F# is found at
`https://fsharp.org/specs/language-spec/`

Further characteristics for the functional fragment of F#

F# is a statically typed, compiled language language:

Further characteristics for the functional fragment of F#

F# is a statically typed, compiled language language:

- At compile time: A type for every expression in a program is inferred.
  If this is not possible, then an type error is issued at compile time

Further characteristics for the functional fragment of F#

F# is a statically typed, compiled language language:

- At compile time: A type for every expression in a program is inferred.
  If this is not possible, then an type error is issued at compile time
- Code is only generated by the compiler for well-typed programs.

Further characteristics for the functional fragment of F#

F# is a statically typed, compiled language language:

- At compile time: A type for every expression in a program is inferred.
  If this is not possible, then an type error is issued at compile time

- Code is only generated by the compiler for well-typed programs.

- At runtime: The generated code contains no type information
  well-typed programs do not go wrong

Syntax	Static semantics Type inference $e : \tau$	Semantics
Types τ Patterns pat Expressions e Declarations d	 Types every piece of an expression Types every piece of a declaration	Value v Binding $id \mapsto v$ Environment $e_1 \rightsquigarrow e_2$
indentation sensitive		

Pragmatics: ?

Syntax	Static semantics Type inference $e : \tau$	Semantics
Types τ Patterns pat Expressions e Declarations d	 Types every piece of an expression Types every piece of a declaration	Value v Binding $id \mapsto v$ Environment $e_1 \rightsquigarrow e_2$
indentation sensitive		

Pragmatics: ?

- type and function names are descriptive
- types start with a capital letter
- variables names are short and consistently used
- function types are stated in comments
- a program is composed by small, well-understood pieces
- adequate use of language constructs
- ...
- common computer-science sense

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x $\quad$ _ $\quad$ $(p_1, \ldots, p_n)$ $\quad$ $p_1 :: p_2$ $\quad$ $[p_1; \ldots; p_n]$
  $p_1 | p_2$ $\quad$ p `when` e $\quad$ p `as` x $\quad$ $p : t \ldots$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x $\quad$ _ $\quad (p_1, \ldots, p_n)$ $\quad p_1 :: p_2$ $\quad [p_1; \ldots; p_n]$
  $p_1 | p_2$ $\quad p$ `when` e $\quad p$ `as` x $\quad p : t \ldots$
- Expressions: x $\quad (e_1, \ldots, e_n)$ $\quad e_1 :: e_2$ $\quad [p_1; \ldots; p_n]$

  $e_1\, e_2$ $\quad e_1 \oplus e_2$ $\quad (\oplus)$ $\quad$ `let` $p_1 = e_1$ `in` e_2

  $e : t$ $\quad$ `if` e `then` e_1 `then` e_2 $\quad$ `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e $\quad$ `function` *clauses* $\quad \ldots$

where the construct *clauses* has the form:

`|` $\texttt{p}_1$ `->` $\texttt{e}_1$ `| ... |` $\texttt{p}_n$ `->` $\texttt{e}_n$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x _ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $[p_1; \ldots; p_n]$
  $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions: x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $[p_1; \ldots; p_n]$

  $e_1\, e_2$ $e_1 \oplus e_2$ $(\oplus)$ `let` $p_1 = e_1$ `in` e_2

  $e : t$ `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses*   ...
- Declarations `let` $f\ p_1 \ldots p_n = e$ `let rec` $f\ p_1 \ldots p_n = e$, $n \geq 0$

where the construct *clauses* has the form:

$$| \ \texttt{p}_1 \ \texttt{->} \ \texttt{e}_1 \ | \ \ldots \ | \ \texttt{p}_n \ \texttt{->} \ \texttt{e}_n$$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x $\_$ $(p_1, \ldots, p_n)$ $p_1 :: p_2$ $[p_1; \ldots ; p_n]$
  $p_1 | p_2$ p `when` e p `as` x $p : t \ldots$
- Expressions: x $(e_1, \ldots, e_n)$ $e_1 :: e_2$ $[p_1; \ldots ; p_n]$

  $e_1\, e_2$ $e_1 \oplus e_2$ $(\oplus)$ `let` $p_1 = e_1$ `in` e_2

  $e : t$ `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses* ...
- Declarations `let` $f\ p_1 \ldots p_n = e$ `let rec` $f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  `int` `float` `bool` `string` $'a$ T<$t_1, \ldots, t_n$> ...
  $t_1 * t_2 * \cdots * t_n$ t `list` t_1`->`$t_2 \ldots$

where the construct *clauses* has the form:

| p_1 -> e_1 | ... | p_n -> e_n

Overview: Syntactical constructs in "our part of" F#

- Constants: 0, 1.1, true, ...
- Patterns: $x \quad \_ \quad (p_1,\ldots,p_n) \quad p_1 :: p_2 \quad [p_1;\ldots;p_n]$
  $p_1|p_2 \quad p\ \texttt{when}\ e \quad p\ \texttt{as}\ x \quad p:t\ldots$
- Expressions: $x \quad (e_1,\ldots,e_n) \quad e_1 :: e_2 \quad [p_1;\ldots;p_n]$

  $e_1\,e_2 \quad e_1 \oplus e_2 \quad (\oplus) \quad \texttt{let}\ p_1 = e_1\ \texttt{in}\ e_2$

  $e:t \quad \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{then}\ e_2 \quad \texttt{match}\ e\ \texttt{with}\ \textit{clauses}$

  $\texttt{fun}\ p_1 \cdots p_n \texttt{->} e \quad \texttt{function}\ \textit{clauses} \quad \ldots$
- Declarations $\texttt{let}\ f\ p_1 \ldots p_n = e \quad \texttt{let rec}\ f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  $\texttt{int} \quad \texttt{float} \quad \texttt{bool} \quad \texttt{string} \quad {}'a \quad T\texttt{<}t_1,\ldots,t_n\texttt{>} \quad \ldots$
  $t_1 * t_2 * \cdots * t_n \quad t\ \texttt{list} \quad t_1 \texttt{->} t_2 \ldots$
- Type abbreviations $\texttt{type}\ T = t \quad \texttt{type}\ T\texttt{<}'a_1,\ldots,'a_n\texttt{>} = t$

where the construct *clauses* has the form:

$\texttt{|}\ \texttt{p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n$

Overview: Syntactical constructs in “our part of” F#

- Constants: 0, 1.1, true, ...
- Patterns: x _ $(p_1,\ldots,p_n)$ $p_1 :: p_2$ $[p_1;\ldots;p_n]$
  $p_1|p_2$ p `when` e p `as` x $p:t\ldots$
- Expressions: x $(e_1,\ldots,e_n)$ $e_1 :: e_2$ $[p_1;\ldots;p_n]$

  $e_1\,e_2$ $e_1 \oplus e_2$ $(\oplus)$ `let` $p_1 = e_1$ `in` e_2

  $e:t$ `if` e `then` e_1 `then` e_2 `match` e `with` *clauses*

  `fun` $p_1 \cdots p_n$`->`e `function` *clauses*   ...
- Declarations `let` $f\ p_1 \ldots p_n = e$ `let rec` $f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  `int` `float` `bool` `string` $'a$ T<$t_1,\ldots,t_n$>   ...
  $t_1 * t_2 * \cdots * t_n$ t `list` t_1`->`$t_2\ldots$
- Type abbreviations `type` $T = t$ `type` T<$'a_1,\ldots,'a_n$> $= t$
- Type declarations `type` $T = C_1 \mid \cdots \mid C_i$ `of` $t_i \mid \cdots$

where the construct *clauses* has the form:

| p_1 -> e_1 | ... | p_n -> e_n

What is the value of *a*?

Consider

```
let pi = 3.14;;

let ca r = pi * r * r;;

let a = let pi = 1.0;;
        ca 1.0;;
```

```
 a?
```

Semantics of a function: A closure

Consider a declaration of `f` in an environment
$env = [a \mapsto 4, b \mapsto true]$:

```
let f x = x+a
```

Semantics of a function: A closure

Consider a declaration of `f` in an environment $env = [a \mapsto 4, b \mapsto true]$:

```
let f x = x+a
```

The resulting environment is:

$$[a \mapsto 4, b \mapsto true, f \mapsto cl_f]$$

Semantics of a function: A closure

Consider a declaration of `f` in an environment $env = [a \mapsto 4, b \mapsto true]$:

```
let f x = x+a
```

The resulting environment is:

$$[a \mapsto 4, b \mapsto true, f \mapsto cl_f]$$

where the *value of f* is a closure cl_f =

$$([x], x + a, [a \mapsto 4])$$

consisting of

- the argument list: [x]
- the body of f: x+a
- the environment with bindings for the *free* variables: $[a \mapsto 4]$

Semantics of a function: A closure

Consider a declaration of `f` in an environment
$env = [a \mapsto 4, b \mapsto true]$:

```
let f x = x+a
```

The resulting environment is:

$$[a \mapsto 4, b \mapsto true, f \mapsto cl_f]$$

where the *value of f* is a closure cl_f =

$$([x], x + a, [a \mapsto 4])$$

consisting of

- the argument list: [x]
- the body of f: x+a
- the environment with bindings for the *free* variables: $[a \mapsto 4]$

The bindings in a closure's environment are determined at the place where the function is declared static binding

Function application

Let env be an environment, where the closure for f is

$$cl_f = ([x], e, env_f)$$

Function application

Let *env* be an environment, where the closure for f is

$$cl_f = ([x], e, env_f)$$

The application $f(a)$ is evaluated as follows in *env*:

$$\begin{array}{ll} & (f(a), env) \\ \rightsquigarrow & (e, [x \mapsto v] + env_f) \end{array}$$

Function application

Let *env* be an environment, where the closure for f is

$$cl_f = ([x], e, env_f)$$

The application $f(a)$ is evaluated as follows in *env*:

$$\begin{array}{ll} & (f(a), env) \\ \rightsquigarrow & (e, [x \mapsto v] + env_f) \end{array}$$

where

- v is the value of a in *env*

Function application

Let *env* be an environment, where the closure for f is

$$cl_f = ([x], e, env_f)$$

The application $f(a)$ is evaluated as follows in *env*:

$$\begin{array}{ll} & (f(a), env) \\ \rightsquigarrow & (e, [x \mapsto v] + env_f) \end{array}$$

where

- v is the value of a in *env*
- $[x \mapsto v] + env_f$ is the environment obtained by adding the binding from $x \mapsto v$ to env_f.

Static binding: an example

Consider

```
let pi = 3.14;;

                                env1 = ?

let ca r = pi * r * r;;

                                env2 = ?

let a = let pi = 1.0
                                env3 = ?
        ca 1.0;;

                                env4 = ?
                                a?
```

Programming as a modelling activity

Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Approach: to name the important concepts of the problem and associate types with the names.

- This model should facilitate discussions about whether it fits the problem formulation.

Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Approach: to name the important concepts of the problem and associate types with the names.

- This model should facilitate discussions about whether it fits the problem formulation.

Aim: A succinct, elegant program reflecting the model.

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

A Functional Model

- Name key concepts and give them a type

A Functional Model

- Name key concepts and give them a type

A signature for the cash register:

```
type ArticleCode = string
type ArticleName = string
type Price       = int
type Register    = (ArticleCode * (ArticleName*Price)) list
type NoPieces    = int
type Item        = NoPieces * ArticleCode
type Purchase    = Item list
type Info        = NoPieces * ArticleName * Price
type Infoseq     = Info list
type Bill        = Infoseq * Price

makeBill: Register -> Purchase -> Bill
```

A Functional Model

- Name key concepts and give them a type

A signature for the cash register:

```
type ArticleCode = string
type ArticleName = string
type Price       = int
type Register    = (ArticleCode * (ArticleName*Price)) list
type NoPieces    = int
type Item        = NoPieces * ArticleCode
type Purchase    = Item list
type Info        = NoPieces * ArticleName * Price
type Infoseq     = Info list
type Bill        = Infoseq * Price

makeBill: Register -> Purchase -> Bill
```

Is the model adequate?

Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

A bill is computed as follows:

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Functional decomposition (1)

Type: findArticle: ArticleCode → Register → ArticleName * Price

```
let rec findArticle ac = function
    | (ac',adesc)::_ when ac=ac' -> adesc
    | _::reg                     -> findArticle ac reg
    | _                          ->
          failwith(ac + " is an unknown article code");;
val findArticle : string -> (string * 'a) list -> 'a
```

Note that the specified type is an instance of the inferred type.

Functional decomposition (1)

Type: findArticle: ArticleCode → Register → ArticleName * Price

```
let rec findArticle ac = function
    | (ac',adesc)::_ when ac=ac' -> adesc
    | _::reg                     -> findArticle ac reg
    | _                          ->
          failwith(ac + " is an unknown article code");;
val findArticle : string -> (string * 'a) list -> 'a
```

Note that the specified type is an instance of the inferred type.

An article description is found as follows:

```
findArticle "a2" reg;;
val it : string * int = ("herring", 4)

findArticle "a5" reg;;
System.Exception: a5 is an unknown article code
   at FSI_0016.findArticle[a] ...
```

Note: failwith is a built-in function that raises an exception

Functional decomposition (2)

Type: makeBill: Register → Purchase → Bill

```
let rec makeBill reg = function
    | []            -> ([],0)
    | (np,ac)::pur ->
        let (aname,aprice) = findArticle ac reg
        let tprice         = np*aprice
        let (billtl,sumtl) = makeBill reg pur
        ((np,aname,tprice)::billtl, tprice+sumtl);;
```

Functional decomposition (2)

Type: makeBill: Register → Purchase → Bill

```
let rec makeBill reg = function
    | []            -> ([],0)
    | (np,ac)::pur ->
          let (aname,aprice) = findArticle ac reg
          let tprice         = np*aprice
          let (billtl,sumtl) = makeBill reg pur
          ((np,aname,tprice)::billtl, tprice+sumtl);;
```

The specified type is an instance of the inferred type:

```
val makeBill :
    (string * ('a * int)) list -> (int * string) list
                -> (int * 'a * int) list * int
```

Functional decomposition (2)

Type: makeBill: Register → Purchase → Bill

```
let rec makeBill reg = function
    | []            -> ([],0)
    | (np,ac)::pur ->
          let (aname,aprice) = findArticle ac reg
          let tprice          = np*aprice
          let (billtl,sumtl) = makeBill reg pur
          ((np,aname,tprice)::billtl, tprice+sumtl);;
```

The specified type is an instance of the inferred type:

```
val makeBill :
    (string * ('a * int)) list -> (int * string) list
                -> (int * 'a * int) list * int
```

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

Summary

- A succinct model is achieved using type declarations.

Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.

Summary

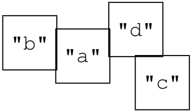
- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.

Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.
- Standard recursions over lists solve the problem.

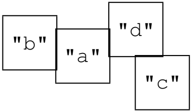
Example: Map Coloring.

Color a map so that neighbouring countries get different colors



Example: Map Coloring.

Color a map so that neighbouring countries get different colors

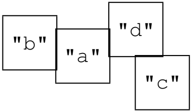


The types for country and map:

- We shall consider different types for countries, so we use a type variable , say `'c`

Example: Map Coloring.

Color a map so that neighbouring countries get different colors



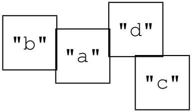
The types for country and map:

- We shall consider different types for countries, so we use a type variable , say `'c`

  Symbols: c, c1, c2, c'; Examples: ”a”, ”b”, . . .

Example: Map Coloring.

Color a map so that neighbouring countries get different colors



The types for country and map:

- We shall consider different types for countries, so we use a type variable , say `'c`

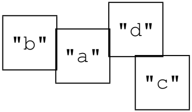
  Symbols: c, c1, c2, c'; Examples: "a", "b", . . .

- The type for `Map` is polymorphic:

```
type Map<'c> = ('c * 'c) list
```

Example: Map Coloring.

Color a map so that neighbouring countries get different colors



The types for country and map:

- We shall consider different types for countries, so we use a type variable , say `'c`

  Symbols: c, c1, c2, c'; Examples: "a", "b", . . .

- The type for `Map` is polymorphic:
  ```
  type Map<'c> = ('c * 'c) list
  ```

  Symbols: m; Example: exMap = [("a","b"); ("c","d"); ("d","a")]

How many ways could above map be colored?

Abstract models for color and coloring

- `type Color<'c> = 'c list`

Be conscious about symbols and examples

Abstract models for color and coloring

- `type Color<'c> = 'c list`

  Symbols: col; Example: ["c"; "a"]

Be conscious about symbols and examples

Abstract models for color and coloring

- `type Color<'c> = 'c list`

  Symbols: col; Example: ["c"; "a"]

- `type Coloring<'c> = Color<'c> list`

Be conscious about symbols and examples

Abstract models for color and coloring

- `type Color<'c> = 'c list`

  Symbols: col; Example: ["c"; "a"]

- `type Coloring<'c> = Color<'c> list`

  Symbols: cols; Example: [["c"; "a"]; ["b"; "d"]]

Be conscious about symbols and examples

Abstract models for color and coloring

- `type Color<'c> = 'c list`

  Symbols: col; Example: ["c"; "a"]

- `type Coloring<'c> = Color<'c> list`

  Symbols: cols; Example: [["c"; "a"]; ["b"; "d"]]

Be conscious about symbols and examples

```
colMap:  Map<'c> -> Coloring<'c>
```

Abstract models for color and coloring

- `type Color<'c> = 'c list`

  Symbols: col; Example: ["c"; "a"]

- `type Coloring<'c> = Color<'c> list`

  Symbols: cols; Example: [["c"; "a"]; ["b"; "d"]]

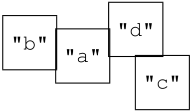
Be conscious about symbols and examples

```
colMap:  Map<'c> -> Coloring<'c>
```

Meta symbol: Type		***Definition***	***Sample value***
`c`	*country*	`'c`	`"a"`
`m:`	`Map<'c>`	`('c * 'c) list`	`[("a","b"),("c","d"),("d","a")]`
`col:`	`Color<'c>`	`'c list`	`["a","c"]`
`cols:`	`Coloring<'c>`	`Color<'c> list`	`[["a","c"],["b","d"]]`

Figure: *A Data model for map coloring problem*

Algorithmic idea



Insert repeatedly countries in a coloring.

	country	old coloring	new coloring
1.	"a"	[]	[["a"]]
2.	"b"	[["a"]]	[["a"] ; ["b"]]
3.	"c"	[["a"] ; ["b"]]	[["a";"c"] ; ["b"]]
4.	"d"	[["a";"c"] ; ["b"]]	[["a";"c"] ; ["b";"d"]]

Figure: Algorithmic idea

Functional decomposition (I)

To make things easy

Functional decomposition (I)

To make things easy

Are two countries neighbours?

```
areNb: Map<'c> -> 'c -> 'c -> bool
let areNb m c1 c2 = List.contains (c1,c2) m
                    || List.contains (c2,c1) m;;
```

Functional decomposition (I)

To make things easy

Are two countries neighbours?

```
areNb: Map<'c> -> 'c -> 'c -> bool
let areNb m c1 c2 = List.contains (c1,c2) m
                    || List.contains (c2,c1) m;;
```

Can a color be extended?

```
canBeExtBy: Map<'c> -> Color<'c> -> 'c -> bool
```

Functional decomposition (I)

To make things easy

Are two countries neighbours?

```
areNb: Map<’c> -> ’c -> ’c -> bool
let areNb m c1 c2 = List.contains (c1,c2) m
                    || List.contains (c2,c1) m;;
```

Can a color be extended?

```
canBeExtBy: Map<’c> -> Color<’c> -> ’c -> bool
let rec canBeExtBy m col c =
  match  col with
  | []        -> true
  | c’::col’  -> not (areNb m c’ c) && canBeExtBy m col’ c;;

canBeExtBy exMap ["c"] "a";;
val it : bool = true

canBeExtBy exMap ["a"; "c"] "b";;
```

Functional composition (I)

Combining functions make things easy

Functional composition (I)

Combining functions make things easy

Extend a coloring by a country:

```
extColoring: Map<'c> -> Coloring<'c> -> 'c -> Coloring<'c>
```

```
Examples:
  extColoring exMap [] "a"        =  [["a"]]
  extColoring exMap [["b"]] "a"   =  [["b"] ; ["a"]]
  extColoring exMap [["c"]] "a"   =  [["a"; "c"]]
```

Functional composition (I)

Combining functions make things easy

Extend a coloring by a country:

```
extColoring: Map<'c> -> Coloring<'c> -> 'c -> Coloring<'c>
```

```
Examples:
  extColoring exMap [] "a"          =  [["a"]]
  extColoring exMap [["b"]] "a"     =  [["b"] ; ["a"]]
  extColoring exMap [["c"]] "a"     =  [["a"; "c"]]
```

```
let rec extColoring m cols c =
    match cols with
    | []          -> [[c]]
    | col::cols'  -> if canBeExtBy m col c
                     then (c::col)::cols'
                     else col::extColoring m cols' c;;
```

Functional composition (I)

Combining functions make things easy

Extend a coloring by a country:

```
extColoring: Map<'c> -> Coloring<'c> -> 'c -> Coloring<'c>
```

```
Examples:
  extColoring exMap [] "a"        =  [["a"]]
  extColoring exMap [["b"]] "a"   =  [["b"] ; ["a"]]
  extColoring exMap [["c"]] "a"   =  [["a"; "c"]]
```

```
let rec extColoring m cols c =
    match cols with
    | []          -> [[c]]
    | col::cols' -> if canBeExtBy m col c
                    then (c::col)::cols'
                    else col::extColoring m cols' c;;
```

Function types, consistent use of symbols, and examples make program easy to comprehend

Functional decomposition (II)

To color a neighbour relation:

- Get a list of countries from the neighbour relation.
- Color these countries

Functional decomposition (II)

To color a neighbour relation:

- Get a list of countries from the neighbour relation.
- Color these countries

Get a list of countries without duplicates:

```
let addElem x ys = if List.contains x ys then ys else x::ys;;

let rec countries = function
    | []            -> []
    | (c1,c2)::m -> addElem c1 (addElem c2 (countries m));;
```

Functional decomposition (II)

To color a neighbour relation:

- Get a list of countries from the neighbour relation.
- Color these countries

Get a list of countries without duplicates:

```
let addElem x ys = if List.contains x ys then ys else x::ys;;

let rec countries = function
    | []           -> []
    | (c1,c2)::m -> addElem c1 (addElem c2 (countries m));;
```

Color a country list:

```
let rec colCntrs m = function
    | []     -> []
    | c::cs -> extColoring m (colCntrs m cs) c;;
```

Functional composition (III)

The problem can now be solved by
combining well-understood pieces

Functional composition (III)

The problem can now be solved by
combining well-understood pieces

Create a coloring from a neighbour relation:

```
colMap:  Map<'c> -> Coloring<'c>
```

Functional composition (III)

The problem can now be solved by
combining well-understood pieces

Create a coloring from a neighbour relation:

```
colMap:  Map<'c> -> Coloring<'c>
```

```
let colMap m = colCntrs m (countries m);;

colMap exMap;;
val it : string list list = [["c"; "a"]; ["b"; "d"]]
```

On modelling and problem solving

- Types are useful in the specification of concepts and operations.

On modelling and problem solving

- Types are useful in the specification of concepts and operations.
- Conscious and consistent use of symbols enhances readability.

On modelling and problem solving

- Types are useful in the specification of concepts and operations.
- Conscious and consistent use of symbols enhances readability.
- Examples may help understanding the problem and its solution.

On modelling and problem solving

- Types are useful in the specification of concepts and operations.
- Conscious and consistent use of symbols enhances readability.
- Examples may help understanding the problem and its solution.
- Functional paradigm is powerful.

Problem solving by combination of well-understood pieces

On modelling and problem solving

- Types are useful in the specification of concepts and operations.
- Conscious and consistent use of symbols enhances readability.
- Examples may help understanding the problem and its solution.
- Functional paradigm is powerful.

Problem solving by combination of well-understood pieces

These points are not programming language specific

Program properties
and
property-based testing

Invariant preservation by example

An integer list $[x_0; x_1; \ldots; x_{n-1}]$ is ordered if

$$x_0 \leq x_1 \leq \cdots \leq x_{n-1} \quad \text{where } n \geq 0$$

Invariant preservation by example

An integer list $[x_0; x_1; \ldots; x_{n-1}]$ is ordered if

$$x_0 \leq x_1 \leq \cdots \leq x_{n-1} \quad \text{where } n \geq 0$$

The function:

```
let rec insert y xs =
  match xs with
  | []              -> [y]               (* C1 *)
  | x::_ when y<=x  -> y::xs             (* C2 *)
  | x::rest         -> x::insert y rest  (* C3 *)
```

inserting y in an ordered list xs should satisfy the property:

If xs is ordered,
then `insert` y xs is ordered as well.

Invariant preservation by example

An integer list $[x_0; x_1; \ldots; x_{n-1}]$ is ordered if

$$x_0 \leq x_1 \leq \cdots \leq x_{n-1} \quad \text{where } n \geq 0$$

The function:

```
let rec insert y xs =
  match xs with
  | []               -> [y]                (* C1 *)
  | x::_ when y<=x   -> y::xs              (* C2 *)
  | x::rest          -> x::insert y rest   (* C3 *)
```

inserting y in an ordered list xs should satisfy the property:

If xs is ordered,
then `insert` y xs is ordered as well.

We say that `insert` y respects (or preserves) the invariant:

- if the argument xs is ordered
  then the value `insert` y xs is ordered as well

Invariant preservation by example

An integer list $[x_0; x_1; \ldots; x_{n-1}]$ is ordered if

$$x_0 \leq x_1 \leq \cdots \leq x_{n-1} \quad \text{where } n \geq 0$$

The function:

```
let rec insert y xs =
  match xs with
  | []                -> [y]                  (* C1 *)
  | x::_ when y<=x -> y::xs                (* C2 *)
  | x::rest           -> x::insert y rest   (* C3 *)
```

inserting y in an ordered list xs should satisfy the property:

If xs is ordered,
then `insert` y xs is ordered as well.

We say that `insert` y respects (or preserves) the invariant:

- if the argument xs is ordered
  then the value `insert` y xs is ordered as well

Function f preserves invariant p:

- if argument a satisfies property p, i.e. $p(a)$ holds,
  then the result $f(a)$ satisfies p as well, i.e. $p(f(a))$ holds

Property-based testing

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs, Claessen and Hughes, 2000

- Random generation of values of arbitrary types
- Properties are expressed as Boolean-valued functions

```
let rec sort xs = .....
let rec ordered xs = ...

// Test that: for all lists xs: ordered(sort xs)
let sortProp (xs: int list) = ordered(sort xs)

let _ = Check.Quick sortProp
 Ok, passed 100 tests.
```

The tool has been ported to many languages. We look at FsCheck for the .Net platform. Consult

- `https://fscheck.github.io/FsCheck/` and
- TipsTricksPrograms DTU Learn

Testing for correctness wrt. a reference model (I)

```
#r "nuget: FsCheck"
open FsCheck

let rec sumA xs acc =
   match xs with
   | []    -> 0
   | x::xs -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

Testing for correctness wrt. a reference model (I)

```
#r "nuget: FsCheck"
open FsCheck

let rec sumA xs acc =
   match xs with
   | []    -> 0
   | x::xs -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

```
let sumRefProp xs = List.sum xs = sumA xs 0;;
let _ = Check.Quick sumRefProp;;
Falsifiable, after 2 tests (2 shrinks) (StdGen ...... :
Original:
[-2; -1]
Shrunk:
 [1]
```

- uses built-in generators for lists
- tool provides a short counterexample

Testing for correctness wrt. a reference model (II)

```
let rec sumA xs acc =
   match xs with
   | []    -> acc
   | x::xs -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

Testing for correctness wrt. a reference model (II)

```
let rec sumA xs acc =
   match xs with
   | []    -> acc
   | x::xs -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

```
let sumRefProp xs = List.sum xs = sumA xs 0;;
let _ = Check.Quick sumRefProp;;
 Ok, passed 100 tests.
```

Testing for correctness wrt. a reference model (II)

```
let rec sumA xs acc =
   match xs with
   | []    -> acc
   | x::xs -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

```
let sumRefProp xs = List.sum xs = sumA xs 0;;
let _ = Check.Quick sumRefProp;;
 Ok, passed 100 tests.
```

- default is 100 random tests
- can be configured

Testing for correctness wrt. a reference model (III)

Test cases are exposed using Check.Verbose as follows:

```
let sumRefProp xs = List.sum xs = sumA xs 0;;
let _ = Check.Verbose sumRefProp;;

0:
[-2]
.....
99:
[-1; 0; -1; -1; 2; 1; -1; 0; 0; 5; -1; 1; -1; 0; 0; -1; 2;
1; -1; 1; -1; 0; -1; -1; -1; -1; 1; -1; 1; 1; 1; 0; -2; 1;
1; 0; -1; 0; -1; -1; -2; 2; 0; 1; -1; -1; 1; 1; 0; 0; -1; 0
0]
Ok, passed 100 tests.
```

Summary

Property-based testing supports testing at a high level of abstraction

- Focus is on fundamental properties – not on concrete test cases
- You write programs for properties – not concrete test cases
- Properties are tested automatically
- Short counterexamples are found — when properties are falsified

The examples given here are just appetizers.

Polynomial project: On checking properties

The list $[a_0; \ldots; a_{n-1}], n \geq 0$ is a *legal* representation of polynomial $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ if $n = 0$ or $a_{n-1} \neq 0$.

- The last element of a representation cannot be 0
- Each polynomial has a unique representation

Polynomial project: On checking properties

The list $[a_0; \ldots; a_{n-1}], n \geq 0$ is a *legal* representation of polynomial $a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$ if $n = 0$ or $a_{n-1} \neq 0$.

- The last element of a representation cannot be 0
- Each polynomial has a unique representation

You should check invariant properties like:

- For every legal representation p: `isLegal(mulX p)`.

Polynomial project: On checking properties

The list $[a_0; \ldots; a_{n-1}]$, $n \geq 0$ is a *legal* representation of polynomial $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$ if $n = 0$ or $a_{n-1} \neq 0$.

- The last element of a representation cannot be 0
- Each polynomial has a unique representation

You should check invariant properties like:

- For every legal representation p: `isLegal(mulX` p`)`.

The script: `PolyGenerator.fsx` contains a generator:

- produces only legal representations of polynomials (type `Poly`)
- having small coefficients and degrees

Polynomial project: On checking properties

The list $[a_0; \ldots; a_{n-1}]$, $n \geq 0$ is a *legal* representation of polynomial $a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$ if $n = 0$ or $a_{n-1} \neq 0$.

- The last element of a representation cannot be 0
- Each polynomial has a unique representation

You should check invariant properties like:

- For every legal representation p: `isLegal(mulX p)`.

The script: `PolyGenerator.fsx` contains a generator:

- produces only legal representations of polynomials (type `Poly`)
- having small coefficients and degrees

You do not need to understand the generator. Just use it, like in:

```
let mulXwrong p = 0::p
let mulXinvWrong (p:Poly) = isLegal(mulXwrong p);;

let testMulXInvWrong = Check.Quick mulXinvWrong;;
>Falsifiable, after 6 tests (0 shrinks) (StdGen (261...
 Original:[]
```