FUNCTIONAL PROGRAMMING AT SIMCORP

FLORIAN BIERMANN
FLBM@SIMCORP.COM

SimCorp

THE SPEAKER

FLORIAN BIERMANN

- **ITU alumn**
  - M.Sc. in Software Development (2014)
  - Ph.D. on parallel, functional programming (2018)
    - “Data Parallel Spreadsheet Programming”
- **Developer at SimCorp since August 2018**
  - IBOR Agile Release Train
  - C# and Ocaml, sometimes a bit of F#
  - Financial contracts, derivatives, OTC

SimCorp

SIMCORP

INVESTMENT MANAGEMENT SOLUTIONS PROVIDER

- Established in 1971
- EUR 382.6 million revenue
- 25 offices globally, main development in DK and UA
- 1900 employees (2020)
- 300+ SimCorp Dimension clients worldwide

SimCorp Dimension

Fully integrated front-to-back investment management solution, powered by an award-winning Investment Book of Record, **offered globally**

190+ clients

FINANCIAL CONTRACTS

WHAT DO WE DO – AND WHY?

- **Bond (loan)**
  - Pay amount up front (nominal)
  - Pay interest on nominal over time
  - Pay back nominal in the end
- **Option (financial instrument)**
  - Right to exercise an underlying contract.
  - E.g. take up loan on already agreed conditions.
  - In a period or at specific points in time.
- **Over-the-Counter (OTC)**
  - Highly customizable financial instruments
  - Not “centrally cleared”, bilateral agreement.

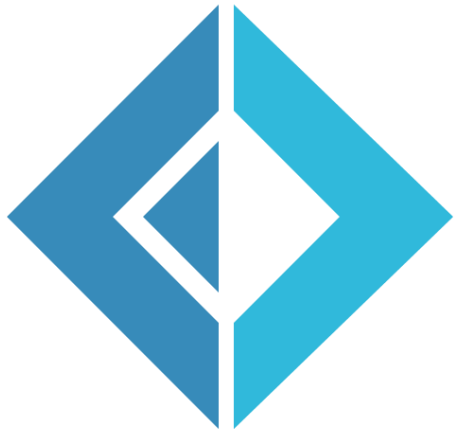
Like an insurance: if it rains in May, you may buy crops at $5 a pound.

Financial contracts have market value!

Cross-currency trades, interest swaps, Sell-buy-back, …

SIMCORP TECHNOLOGIES

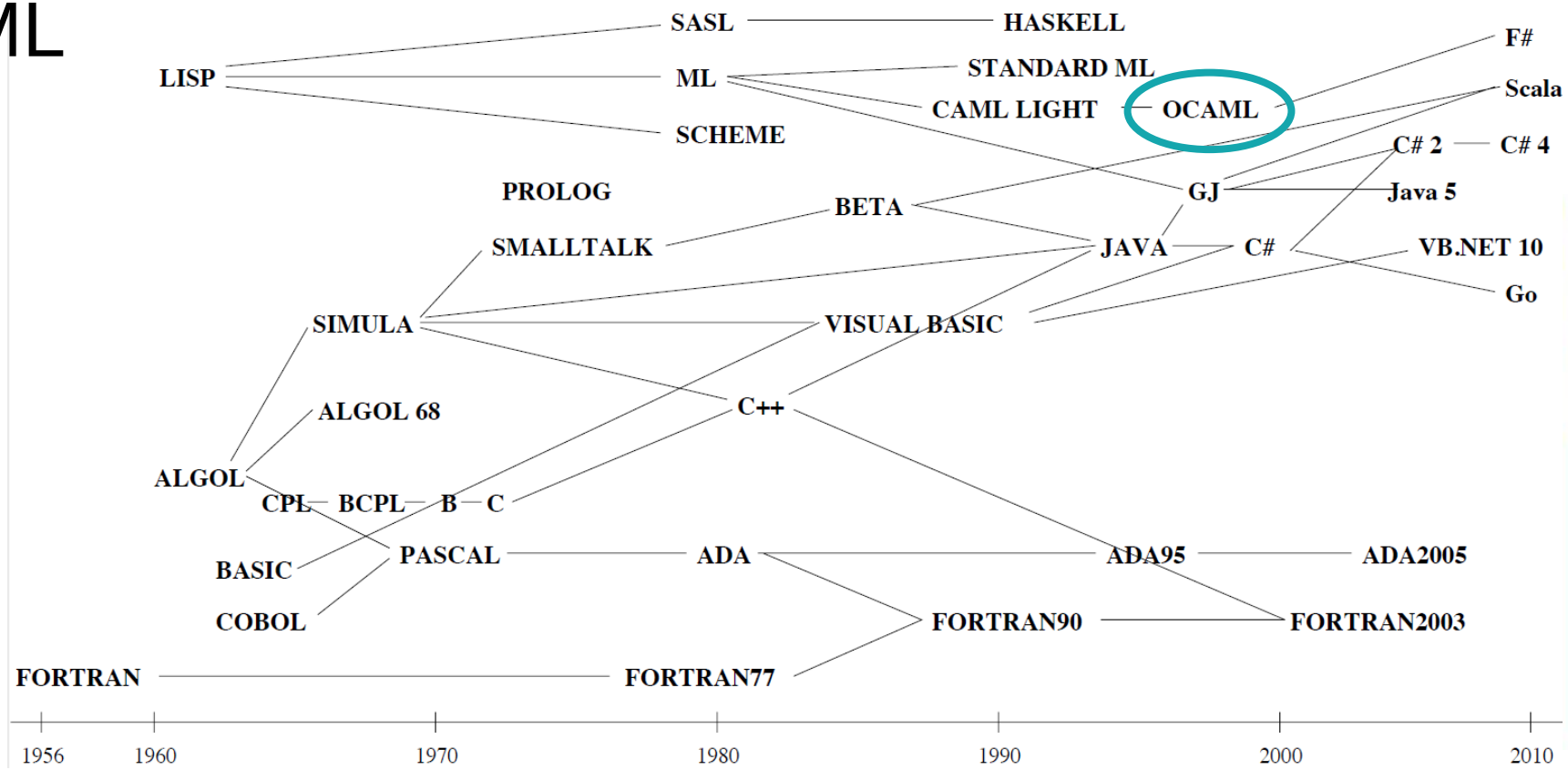
A SMALL SELECTION

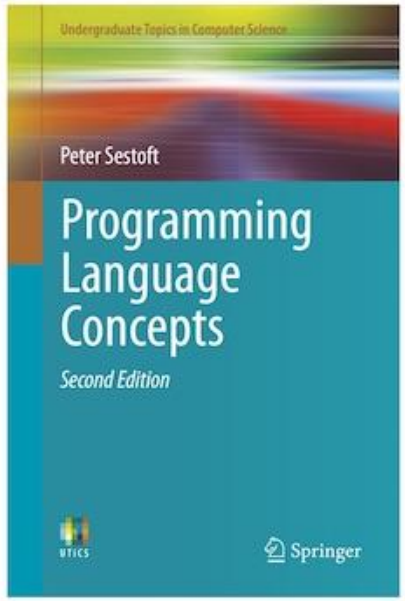


WHAT THIS TALK IS NOT ABOUT

- Dyalog APL
  - Most code at SimCorp is APL code
  - In use since the 1970's
  - Dynamically typed, interpreted, declarative, array-oriented
- C#
  - Most new development on .NET
  - Threading, services, GUI
- F#
  - Some components of SimCorp Dimension
  - SimCorp internal static APL type checker

OCAML

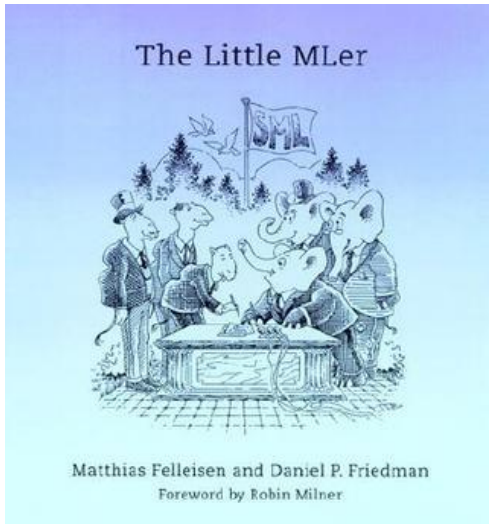




- Multi-paradigm: functional, imperative, object-oriented
- Static types and type inference
- Strict evaluation
- Interactive top-level (REPL)

OCAML TYPES ARE SAFE

```
type pizza =
  | Crust
  | Cheese  of pizza
  | Sausage of pizza
  | Anchovy of pizza
  | Onion   of pizza
```




```
Cheese (Sausage (Onion Crust))
Cheese (Anchovy (Onion Crust))
Crust (Sausage Crust)
```

INDUCTIVE TYPES STRUCTURE PROGRAMS

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Mul of expr * expr
  | Neg of expr
```

```
let rec eval : expr -> int = function
  | Cst i -> i
  | Add (e1, e2) -> eval e1 + eval e2
  | Mul (e1, e2) -> eval e1 * eval e2
  | Neg e -> -(eval e)
```

```
# let e1 = Add (Cst 1, Neg (Cst 1));;
val e1 : expr = Add (Cst 1, Neg (Cst 1))

# let e2 = Add (“one”, Cst 1);;
Error: This expression has type string but an expression was expected of
type expr

# eval e1;;
- : int = 0
```

Composing contracts: an adventure in financial engineering

Functional pearl

Simon Peyton Jones
Microsoft Research, Cambridge
simonpj@microsoft.com

Jean-Marc Eber
LexiFi Technologies, Paris
jeanmarc.eber@lexifi.com

Julian Seward
University of Glasgow
v-sewardj@microsoft.com

23rd August 2000

Abstract

Financial and insurance contracts do not sound like promising territory for functional programming and formal semantics, but in fact we have discovered that insights from pro-

At this point, any red-blooded functional programmer should start to foam at the mouth, yelling "build a combinator library". And indeed, that turns out to be not only possible, but tremendously beneficial.

The finance industry has an enormous vocabulary of jargon

FINANCIAL CONTRACTS MODELLING

- Two parties agree on an amortized loan.
- Pay 30000 DKK up front.
- Pay back over three years with interest.
- Decide when payments are due.

```
let amortized_loan =
  let principal = cst 30000. in
  let coupon = cst 11000. in
  all [ give (flow 2019-01-01 DKK principal);
        flow 2020-01-01 DKK coupon;
        flow 2021-01-01 DKK coupon;
        flow 2022-01-01 DKK coupon ]
```

A COMBINATOR LIBRARY FOR FINANCIAL CONTRACTS

OCAML DIALECT “LEXIFI OCAML”

```
type binop = Add | Sub | Mul | Div | Max

type observable =
  | Underlying of string
  | Const of float
  | Binop of obs * binop * obs
  | Fixed of date * obs

type contract =
  | One     of currency
  | Scale   of observable * contract
  | Acquire of date * contract
  | All     of contract list
  | Give    of contract
  | Either  of contract * contract
  | Anytime of date * date * contract * contract
```

Fix an underlying market rate to its value at some date.

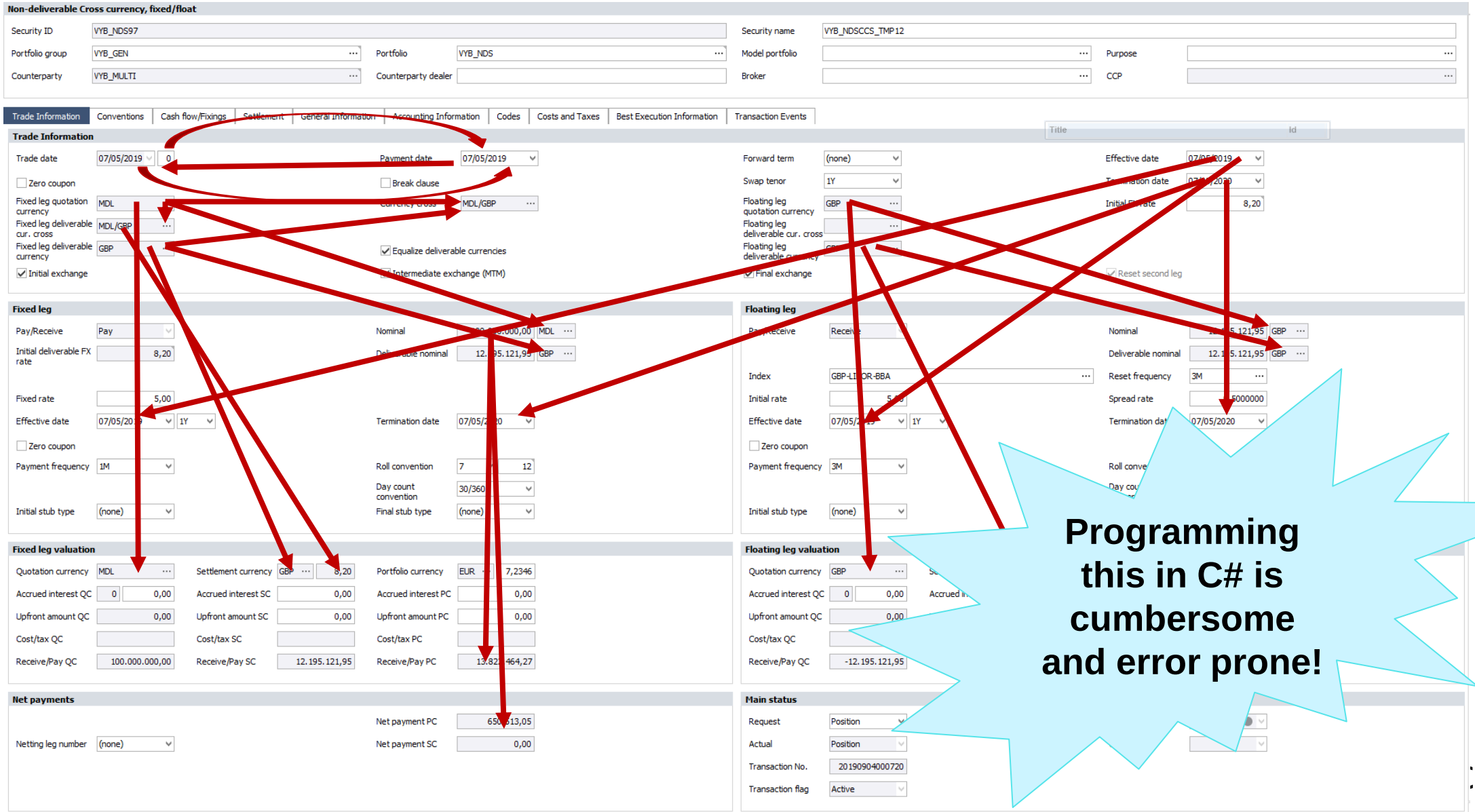
Exercise contract in between dates, or exercise other contract after.

```
let flow t cur obs =
  Acquire (t, Scale (obs, One cur))

let amortized_loan =
  let principal = Const 30000. in
  let coupon = Const 11000. in
  all [ Give (flow 2019-01-01 DKK principal);
        flow 2020-01-01 DKK coupon;
        flow 2021-01-01 DKK coupon;
        flow 2022-01-01 DKK coupon ]
```

Evaluates to a value of type `contract` that we can valuate.

SimCorp

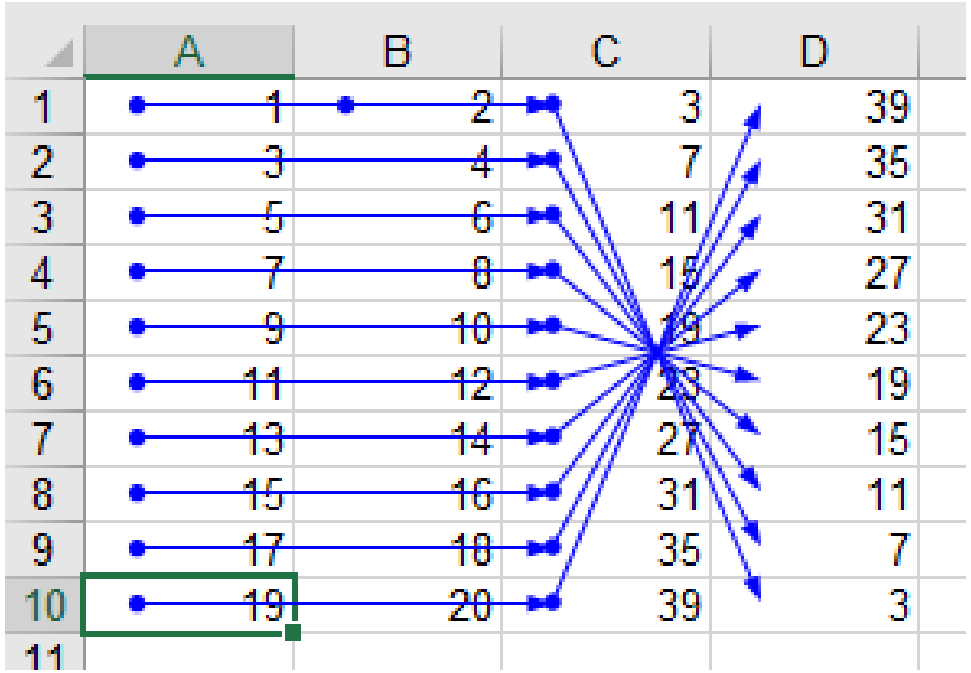
PROGRAMMING COMPLEX GUI LOGIC



LOOKS FAMILIAR?

SPREADSHEET-LIKE EVALUATION MODEL

- Fields depend on each other
  - When the user updates a field, all depending fields must be updated, too.
  - “Reactive programming”
  - “Self-adjusting computation”
- Free of side effects
  - From a programmer’s point of view!
- Programming challenge for domain experts
  - What happens if the user changes the number of settlement days to 5?
- **Solution: pure, type-safe, declarative programming**



SimCorp

Typelets — A Rule-Based Evaluation Model for Dynamic, Statically Typed User Interfaces

Martin Elsman[1] and Anders Schack-Nielsen[2]

[1] University of Copenhagen, Universitetsparken 5, DK-2100 Copenhagen, Denmark
mael@diku.dk
[2] SimCorp, Weidekampsgade 16, DK-2300 Copenhagen, Denmark
anders.schack-nielsen@simcorp.com

Abstract. We present the concept of *typelets*, a specification technique for dynamic graphical user interfaces (GUIs) based on types. The technique is implemented in a dialect of ML, called MLFi,[3] which supports dynamic types, for migrating type-level information into the object level, so-called type properties, allowing easy specification of, for instance, GUI control attributes, and type paths, which allows for type-safe access to type components at runtime. Through the use of Hindley-Milner style type-inference in MLFi, the features allow for type-level programming of user interfaces. The dynamic behavior of typelets are specified us-

RULES DESCRIBE BUSINESS LOGIC

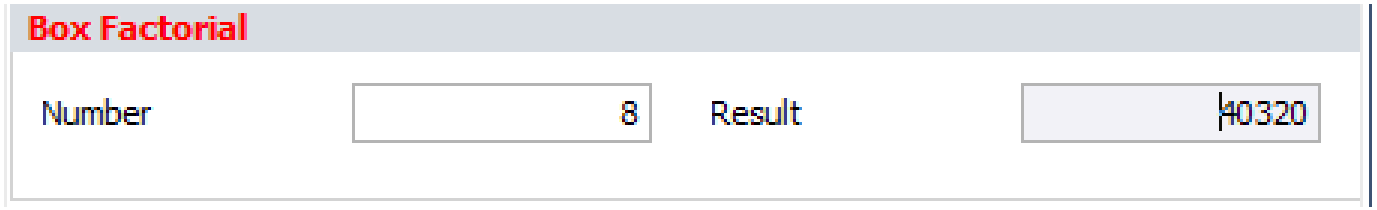
DECLARATIVELY COMPUTE FACTORIAL

```
type t = {
  number: int;
  result: int + [readonly]
}

let rule =
  let rec fact = function
    | 0 -> 1
    | n -> n * fact (n - 1)
  in
  Rule.update
    (Fields.value (.number))
    (Fields.value (.result))
    fact

let layout =
  let open Layout in
  box "Factorial"
      (lpick (.number) % lpick (.result))
```

Use record types to declare fields in the business logic.

Define “normal” OCaml function.

Use it as projection in “update” rule.

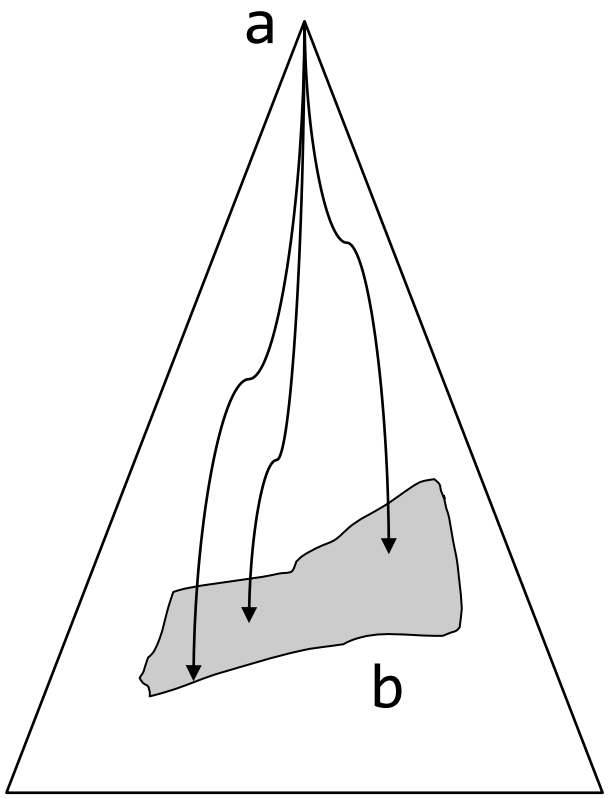


SimCorp

ACCESSING FIELDS THROUGH FIELD API

INTERNAL API FOR DEVELOPING BUSINESS LOGIC

```
module type FIELDS = sig
  type ('i, 'a) t (* 'i : type of the root *)
                  (* 'a : type of elements pointed to *)
  val const    : 'a ttype -> 'a -> ('i, 'a) t
  val value    : ('i, 'a) path -> ('i, 'a) t
  val enabled  : ('i, _) path -> ('i, bool) t
  val readonly : ('i, _) path -> ('i, bool) t
  val restrict : ('i, 'a) path -> ('i, 'a list) t
  val (&)      : ('i, 'a) t -> ('i, 'b) t -> (‘i, 'a * 'b) t
end
```




```
type vec2d = { x : float; y : float }
type vec3d = { x : vedc2; z : float }

let f = Fields.value (.x.x) & Fields.value (.x.y) & Fields.value (.z)
```

What is the type of `f`?

SimCorp

CONSTRUCTING INSTRUMENTS THROUGH RULE API

INTERNAL API FOR DEVELOPING BUSINESS LOGIC

```
module type RULE = sig
  type 'i t
  type ('i,'a) fields = ('i,'a) Fields.t
  val update   : ('i,'a)fields -> ('i,'b)fields -> ('a -> 'b) -> 'i t
  val validate : ('i,'a)fields -> ('a -> string option) -> 'i t
  val button   : ('i,'a)fields -> ('i,'b)fields -> ('a -> 'b) -> ('i,unit)path -> 'i t
  val subpath  : ('i,'a)tpath -> 'a t -> 'i t
  val all      : 'i t list -> 'i t
  val iso      : ('i,'a)fields -> ('i,'b)fields -> ('a -> 'b) -> ('b -> 'a) -> 'i t
  ...
end
```

Isomorphism between two fields.

Lift rule of type `'a t` into context of `'i` if path from `'i` to `'a` exists.

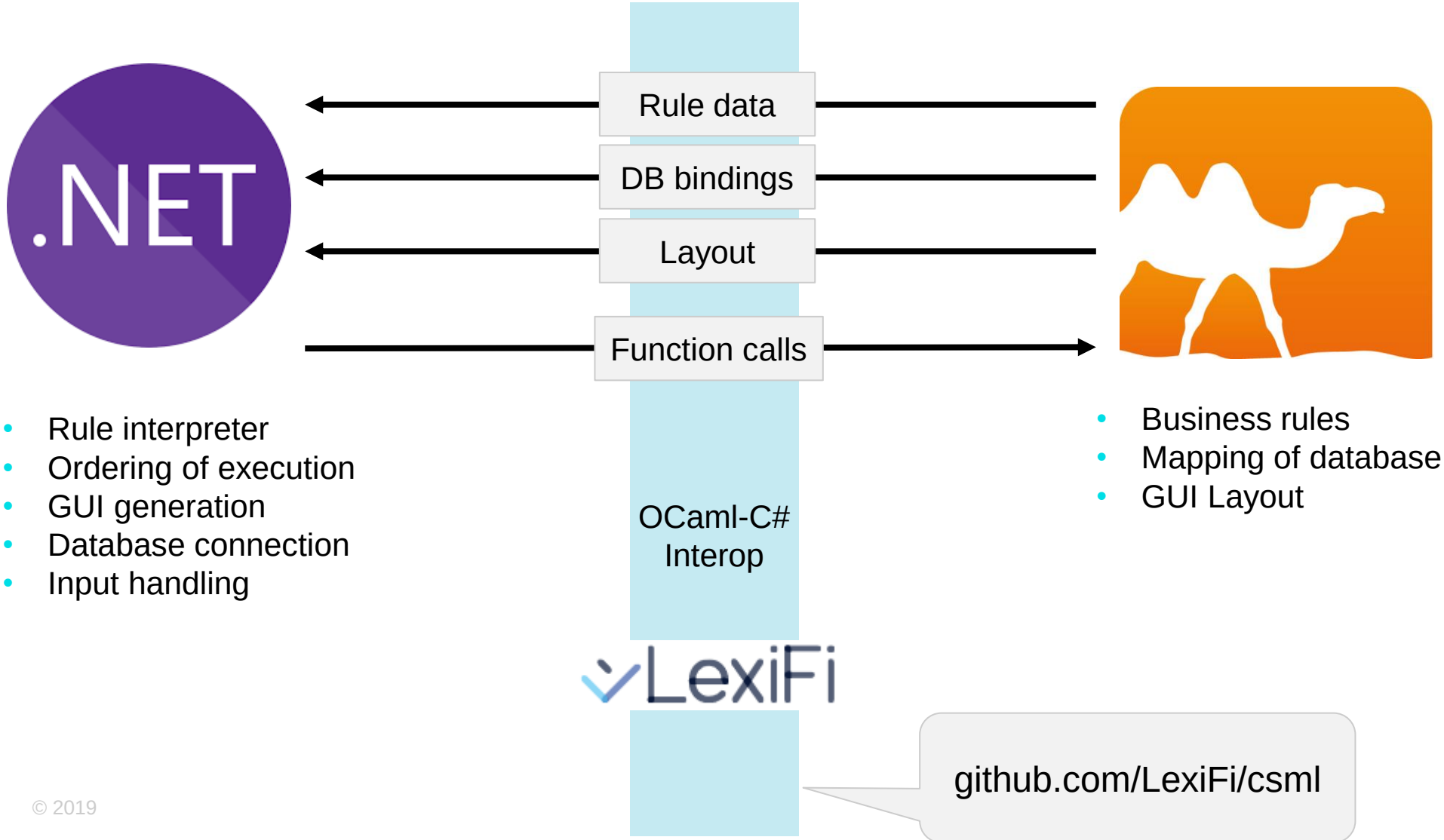
Check whether some property holds for given fields.

What does function `all : 'i t list -> 'i t` do?

SimCorp

RULE EXECUTION

INTERPETER IMPLEMENTED IN .NET – BUSINESS RULES IN OCAML



- Rule interpreter
- Ordering of execution
- GUI generation
- Database connection
- Input handling

- Business rules
- Mapping of database fields
- GUI Layout

MONADIC RULE API

SPLITTING RULES INTO SMALLER STEPS

- Lexifi OCaml run-time is not reentrant.
- Data base access & calls to APL are bottlenecks.
- Solution: split rule computation into small chunks.

Monadic type **'a m**!

```
module Rules : sig =

  val update_m : ('i,'a)fields -> ('i,'b)fields -> ('a -> 'b m) -> 'i t
  val update   : ('i,'a)fields -> ('i,'b)fields -> ('a -> 'b)   -> 'i t

end = struct

  let update src tgt f =
    update_m src tgt (fun x -> return (f x))

end
```

val return : 'a -> 'a m

SimCorp

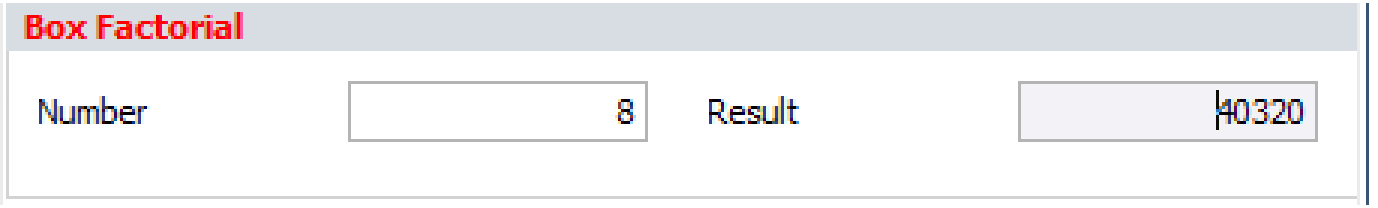
SEPARATION OF CONCERNS

GUI LAYOUT GENERATION VIA LAYOUT API

```
type base = Number | Button | CheckBox | TextBox | Date | DropDown

type ‘p t =
  | Pick   of access_path * string option * base
  | Hseq   of 'p t * ‘p t
  | Vseq   of 'p t * 'p t
  | Halign of halign * 'p t
  | Valign of valign * 'p t
  | Text   of string
  | NamedButton of string
  | Box    of string * ‘p t
```

```
type halign = Left | Center | Right
```

```
let layout =
  let open Layout in
  box "Factorial"
      (lpick (.number) % lpick (.result))
```



SimCorp

HOW WELL DOES THIS WORK IN PRACTICE?

HIGH RE-USE, FAST TIME-TO-MARKET

- Plug-in system using OCaml Functors:
  - A system to generate modules from other modules.
  - Highly composable, type safe, no run-time overhead.
  - Rule composition possible thanks to purity!
- Many business rules are generic!
  - E.g. business calendar functionality.
  - Financial instruments differ only in few places.
- Re-use factor for business rules is **~10**

- **274’845** lines of OCaml code
- **428’771** business rules in production
- **42’132** unique business rules
- Over **100** financial instruments

SUMMARY

FUNCTIONAL PROGRAMMING AT SIMCORP

- **Combinator library for modelling financial contracts**
  - Every-day business for us, revolutionary for the finance sector.
  - Domain experts model contracts.
- **Declarative business logic for type-safe GUI programming**
  - Similar to constructing a spreadsheet.
  - Focus on *what*, not how.
- **Challenges ahead**
  - You gotta know your stuff:
    - Polymorphism, existential types, phantom types
    - Monads & API design
    - Compositionality & catamorphisms

```
let rec flatten = function
  | [] -> []
  | xs :: xxs -> xs @ flatten xss
```

Thank you!

Florian.Biermann@SimCorp.com