

FUNCTIONAL PROGRAMMING AT SIMCORP

FLORIAN BIERMANN

FLORIAN.BIERMANN@SIMCORP.COM

2020-09-30

THE SPEAKER

FLORIAN BIERMANN

- **ITU alumn**
 - M.Sc. in Software Development (2014)
 - Ph.D. in Computer Science (2018)
 - “Data Parallel Spreadsheet Programming”
- **Developer at SimCorp since August 2018**
 - OTC Core
 - C# and OCaml, sometimes a bit of F# and Emacs Lisp
 - Financial contracts, derivatives, framework, application integration



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

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

- **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.

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

**Financial
contracts have
market value!**

SIMCORP TECHNOLOGIES

A SMALL SELECTION



OCaml

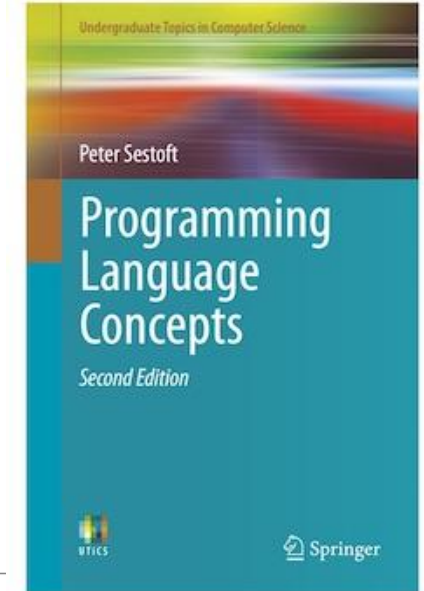
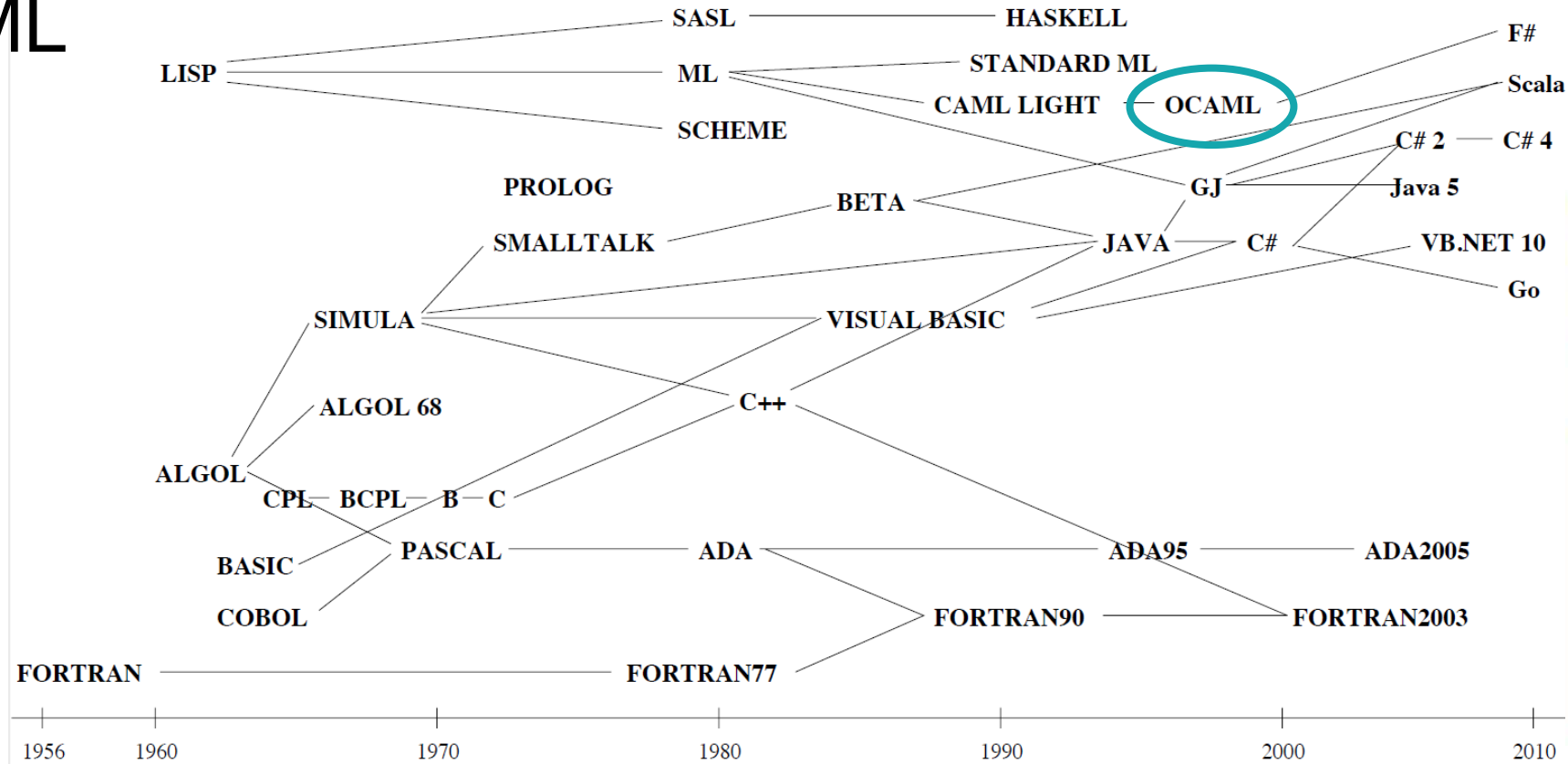


git

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)

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

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

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

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 value.

PROGRAMMING COMPLEX GUI LOGIC

Non-deliverable Cross currency, fixed/float

| | | | |
|-----------------|-----------|---------------------|------------------|
| Security ID | VYB_NDS97 | Security name | VYB_NDSCCS_TMP12 |
| Portfolio group | VYB_GEN | Portfolio | VYB_NDS |
| Counterparty | VYB_MULTI | Counterparty dealer | |
| | | Model portfolio | |
| | | Broker | |
| | | Purpose | |
| | | CCP | |

Trade Information Conventions Cash flow/Fixings Settlement General Information Accounting Information Codes Costs and Taxes Best Execution Information Transaction Events

Trade Information

Trade date: 07/05/2019 Payment date: 07/05/2019

☐ Zero coupon ☐ Break clause

Fixed leg quotation currency: MDL
Fixed leg deliverable cur. cross: MDL/GBP
Fixed leg deliverable currency: GBP
☒ Initial exchange
☐ Equalize deliverable currencies
☐ Intermediate exchange (MTM)
☒ Final exchange

Forward term: (none)
Swap tenor: 1Y
Floating leg quotation currency: GBP
Floating leg deliverable cur. cross: MDL/GBP
Floating leg deliverable currency: GBP
☒ Final exchange
☐ Reset second leg

Effective date: 07/05/2019
Termination date: 07/05/2020
Initial fix rate: 8,20

Fixed leg

Pay/Receive: Pay
Nominal: 100.000,00 MDL
Deliverable nominal: 12.195.121,95 GBP
Fixed rate: 5,00
Effective date: 07/05/2019 Termination date: 07/05/2020
☐ Zero coupon
Payment frequency: 1M
Roll convention: 7 12
Day count convention: 30/360
Initial stub type: (none)
Final stub type: (none)

Fixed leg valuation

| | | | | | | | |
|---------------------|----------------|---------------------|---------------------|----------------|---------------------|------|--------|
| Quotation currency | MDL | Settlement currency | GBP | 8,20 | Portfolio currency | EUR | 7,2346 |
| Accrued interest QC | 0 | 0,00 | Accrued interest SC | 0,00 | Accrued interest PC | 0,00 | |
| Upfront amount QC | 0,00 | 0,00 | Upfront amount SC | 0,00 | Upfront amount PC | 0,00 | |
| Cost/tax QC | | | Cost/tax SC | | Cost/tax PC | | |
| Receive/Pay QC | 100.000.000,00 | Receive/Pay SC | 12.195.121,95 | Receive/Pay PC | 13.821.464,27 | | |

Floating leg

Pay/Receive: Receive
Nominal: 12.195.121,95 GBP
Deliverable nominal: 12.195.121,95 GBP
Index: GBP-LIBOR-BBA
Initial rate: 5,00
Reset frequency: 3M
Spread rate: 0,000000
Effective date: 07/05/2019 Termination date: 07/05/2020
☐ Zero coupon
Payment frequency: 3M
Roll convention: 7 12
Day count convention: 30/360
Initial stub type: (none)
Final stub type: (none)

Floating leg valuation

| | | | | | | | |
|---------------------|----------------|---------------------|---------------------|----------------|---------------------|------|--------|
| Quotation currency | GBP | Settlement currency | GBP | 8,20 | Portfolio currency | EUR | 7,2346 |
| Accrued interest QC | 0 | 0,00 | Accrued interest SC | 0,00 | Accrued interest PC | 0,00 | |
| Upfront amount QC | 0,00 | 0,00 | Upfront amount SC | 0,00 | Upfront amount PC | 0,00 | |
| Cost/tax QC | | | Cost/tax SC | | Cost/tax PC | | |
| Receive/Pay QC | -12.195.121,95 | Receive/Pay SC | 12.195.121,95 | Receive/Pay PC | 13.821.464,27 | | |

Net payments

Netting leg number: (none)
Net payment PC: 650.113,05
Net payment SC: 0,00

Main status

Request: Position
Actual: Position
Transaction No.: 2019090400720
Transaction flag: Active

Programming this in C# is cumbersome and error prone!

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**

| | A | B | C | D |
|----|----|----|----|----|
| 1 | 1 | 2 | 3 | 39 |
| 2 | 3 | 4 | 7 | 35 |
| 3 | 5 | 6 | 11 | 31 |
| 4 | 7 | 8 | 15 | 27 |
| 5 | 9 | 10 | 19 | 23 |
| 6 | 11 | 12 | 23 | 19 |
| 7 | 13 | 14 | 27 | 15 |
| 8 | 15 | 16 | 31 | 11 |
| 9 | 17 | 18 | 35 | 7 |
| 10 | 19 | 20 | 39 | 3 |
| 11 | | | | |

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

Martin Elsmann¹ and Anders Schack-Nielsen²

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 [@t readonly])  
}
```

Use record types to declare fields in the business logic.

```
let rule =  
  let rec fact = function  
    | 0 -> 1  
    | n -> n * fact (n - 1)
```

Define “normal” OCaml function.

```
in  
Rule.update  
  (Fields.value [%p number])  
  (Fields.value [%p result])  
  fact
```

Use it as projection in “update” rule.

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

Box Factorial

Number

8

Result

40320

ACCESSING FIELDS THROUGH FIELD API

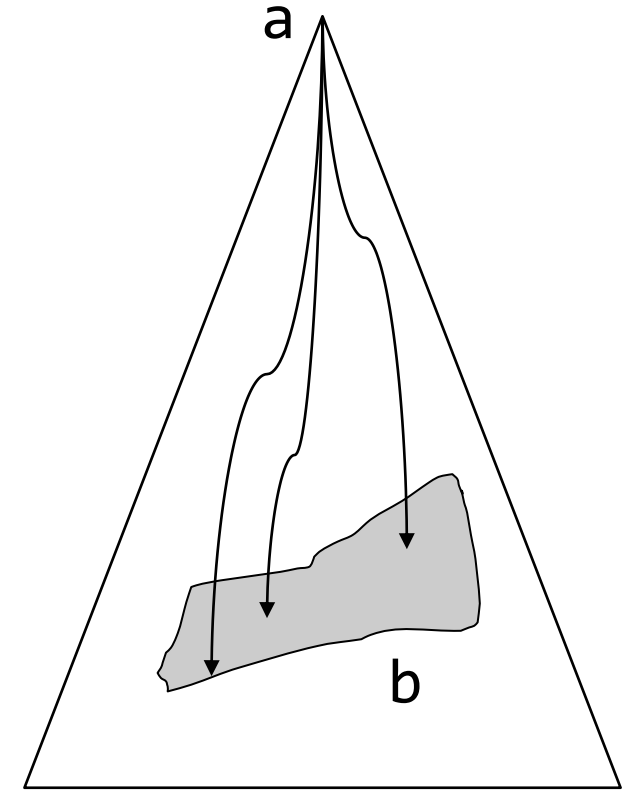
INTERNAL API FOR DEVELOPING BUSINESS LOGIC

```
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 : vec2d; z : float }
```

```
let f = Fields.value [%p x.x] & Fields.value [%p x.y] & Fields.value [%p .z]
```



What is the type of **f**?

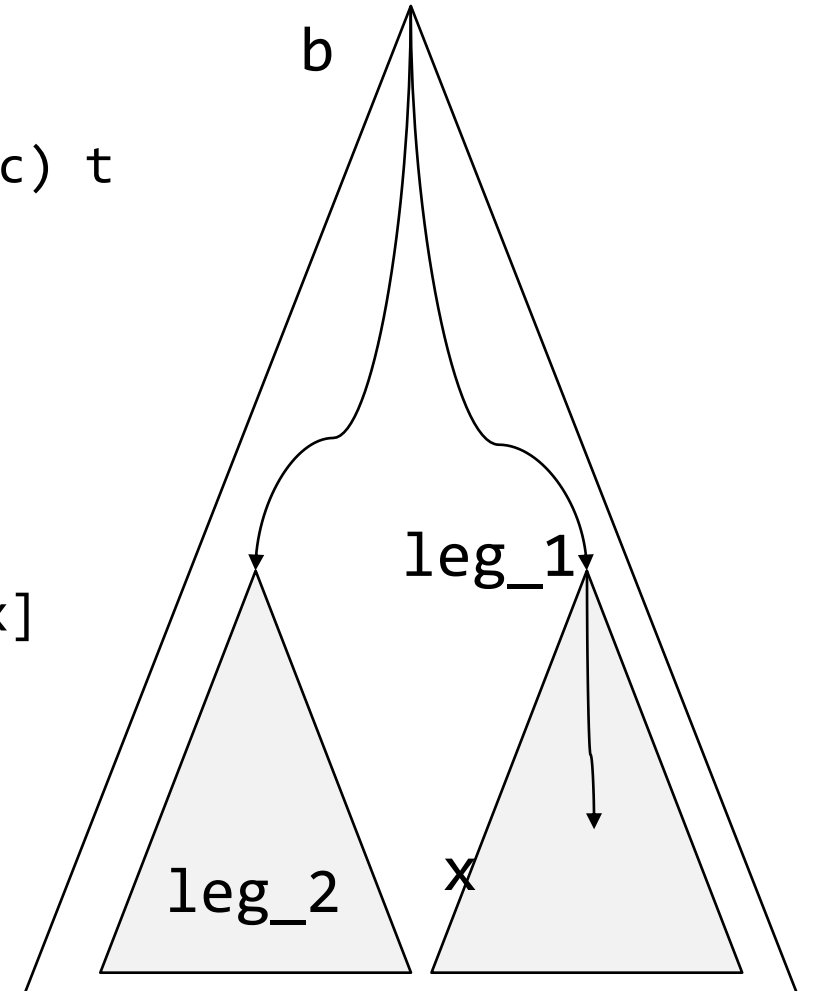
PATHS COMPOSE

ALLOWS TO DIFFERENTIATE BETWEEN REPLICATED STRUCTURES

```
module type PATHS = sig
  val subpath : ('a, 'b) t -> ('b, 'c) t -> ('a, 'c) t
end
```

```
type a = { x : int; y : string }
type b = { leg_1 : a; leg_2 : a }
```

```
let x_leg_1 : (b, int) t = subpath [%p leg_1] [%p x]
```



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)path -> '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.

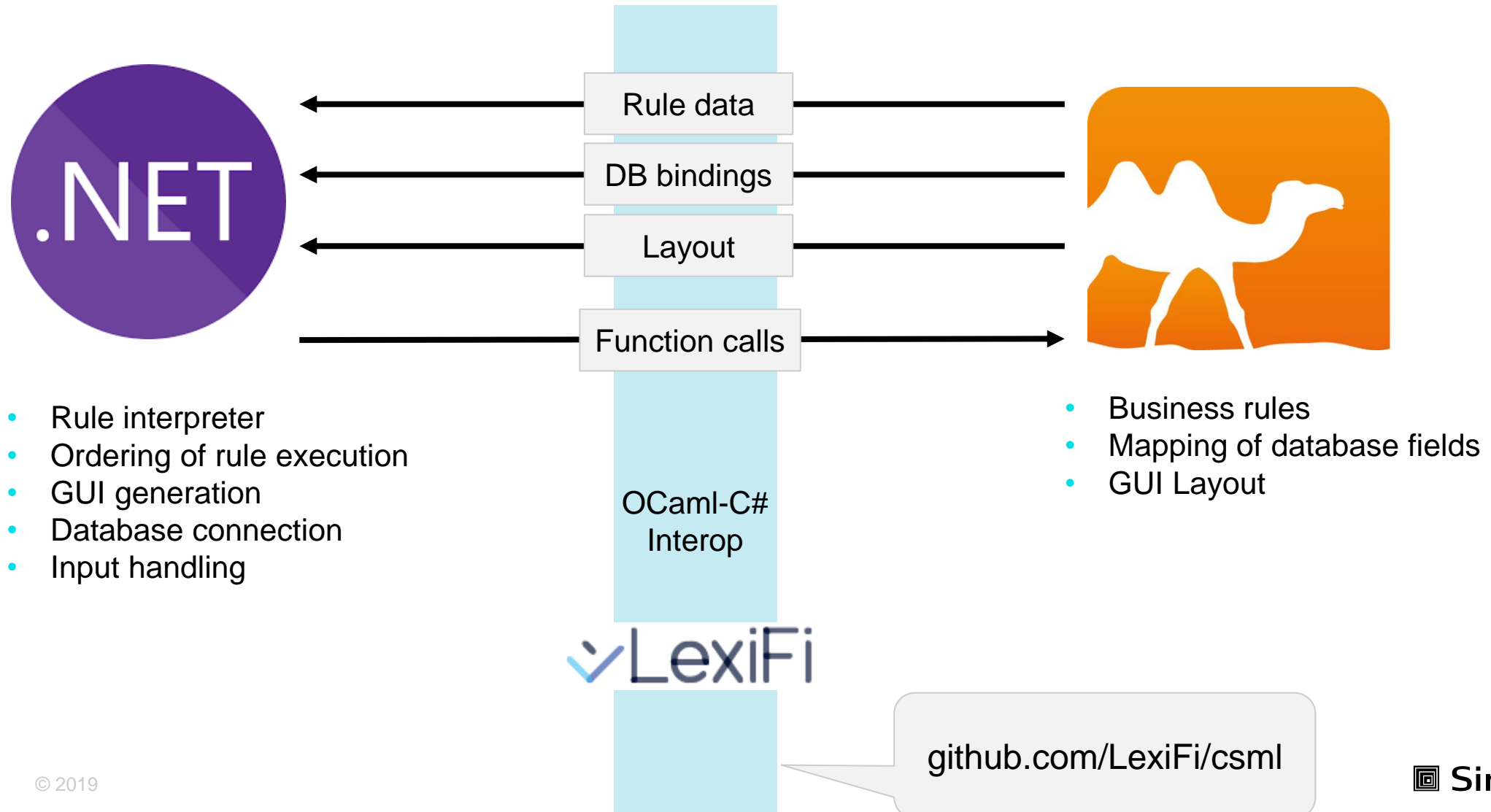
Lift rule of type 'a t into
context of 'i if path from 'i
to 'a exists.

Check whether some
property holds for given
fields.

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

RULE EXECUTION

INTERPETER IMPLEMENTED IN .NET – BUSINESS RULES IN OCAML



STOP: DEMO TIME!



MONADIC RULE API

SPLITTING RULES INTO SMALLER STEPS

- OCaml run-time is not reentrant
- Data base access & calls to APL are long-running and blocking.
- Solution: split rule computation into small chunks – compute step-wise.

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  
val abort  : unit -> 'a m
```

```
update_m  
  (value [%p x])  
  (value [%p w])  
  (call_apl_1 >>= fun (_, y) ->  
    return y >>=  
    call_apl_2)
```

```
update_m  
  (value [%p portfolio_ik])  
  (value [%p currency_ik])  
  (function  
    | 0 -> abort ()  
    | por_ik -> dbhandle >>= fun h ->  
      match select1 h pCURIK (all [pPORIK =. por_ik]) with  
      | [cur_ik] -> return cur_ik  
      | _ -> abort ())
```

orp

EVALUATING RULES AFTER A USER CHANGE

TWO LOOPS TO RULE THE RULES

```
Value EvalRule(RuleComputation rc) {  
    while (true) {  
        if (rc.IsValue) return rc.GetValue();  
        else if (rc.IsAbort) return null;  
        else if (rc.IsCSharpFunction) {  
            rc = Invoke(rc.CSharpFunction, rc.GetValue());  
        } else {  
            rc = rc.Fun(rc.GetValue());  
        }  
    }  
}
```

```
void EvalRules() {  
    while (Rq.HasNext) {  
        RuleInfo r = Rq.Pop();  
        Value inp = Env.GetValue(r.Source);  
        Value res = EvalRule(r.Run(inp));  
        Env.SetValue(r.Target, res);  
    }  
}
```

SetValue() also
updates Rq.

Takes first step

- Iterative algorithm on a queue of rules
- A rule is enqueued when one of its source fields is written to.
- Rules are evaluated step-wise to **enable interleaving**
 - Two clients:
 - One runs a C#-rule
 - The other runs an OCaml rule
 - Minimal mutual exclusion
- **Rule programmer does not really care.**

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 [%p number] % lpick [%p result])
```



Box Factorial

| | | | |
|--------|--------------------------------|--------|------------------------------------|
| Number | <input type="text" value="8"/> | Result | <input type="text" value="40320"/> |
|--------|--------------------------------|--------|------------------------------------|

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

**Psst:
functional
programming
in C#!**

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

Thank you!

Florian.Biermann@SimCorp.com



LEGAL NOTICE

The contents of this publication are for general information and illustrative purposes only and are used at the reader's own risk. SimCorp uses all reasonable endeavors to ensure the accuracy of the information. However, SimCorp does not guarantee or warrant the accuracy, completeness, factual correctness, or reliability of any information in this publication and does not accept liability for errors, omissions, inaccuracies, or typographical errors. The views and opinions expressed in this publication are not necessarily those of SimCorp. © 2019 SimCorp A/S. All rights reserved. Without limiting rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form, by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose without the express written permission of SimCorp A/S. SimCorp, the SimCorp logo, SimCorp®, and SimCorp Services are either registered trademarks or trademarks of SimCorp A/S in Denmark and/or other countries. Refer to www.simcorp.com/trademarks for a full list of SimCorp A/S trademarks. Other trademarks referred to in this document are the property of their respective owners.