SimCorp

Functional Programming for Trade Management and Valuation

Seminar on Functional High Performance Computing in Finance
December 14, 2010

Martin Elsman
SimCorp A/S

The Financial Contracts Market

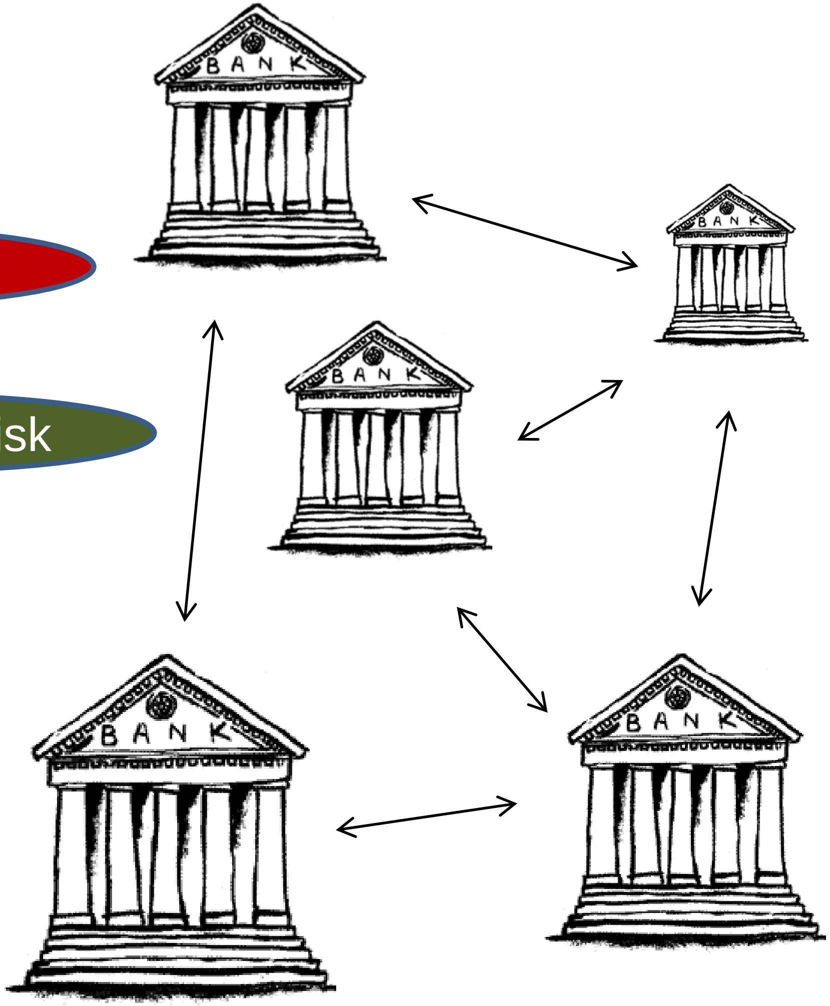
Banks (and other financial institutions) use financial contracts for both

- Speculation
- Insurance (hedging)

Increase risk

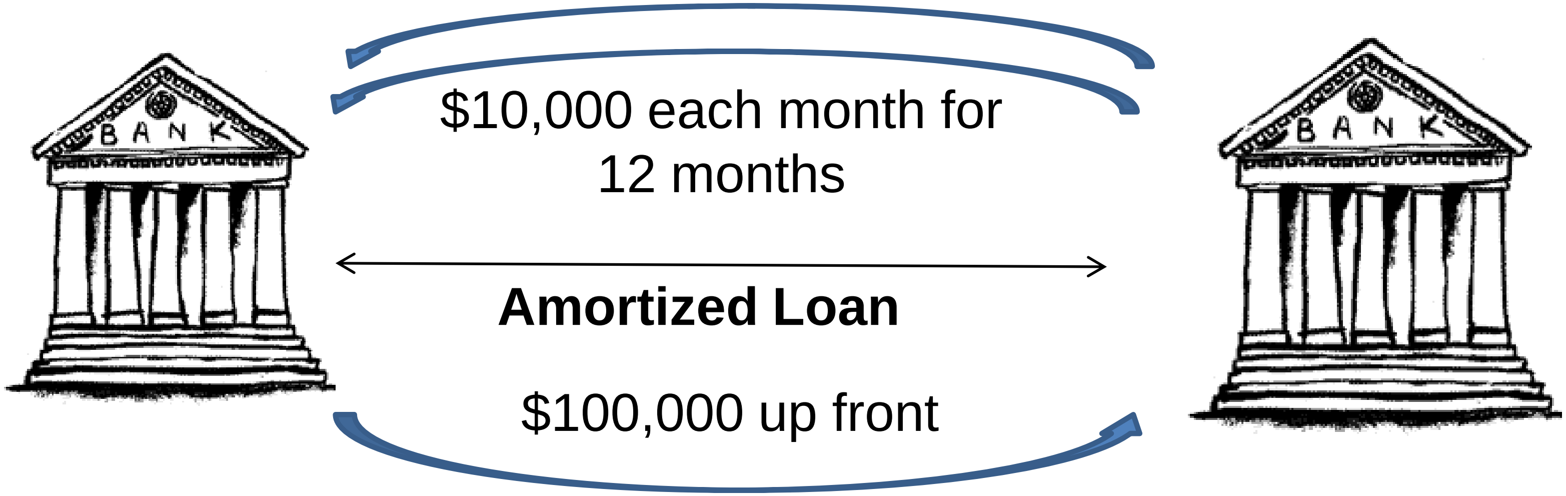
Decrease risk

Many contracts are ”**Over The Counter**” (OTC) contracts, which are negotiated agreements between a bank and another bank (its counter party).



The Term Sheet – the financial contract

- A financial contract is typically agreed upon on a so-called ”Term Sheet”.
- The *term sheet* specifies the financial flows (amounts, dates, etc.) and under which *conditions* a flow should happen.
- Flows can go in both directions.



$10,000 each month for 12 months

Amortized Loan

$100,000 up front

- A ***derivative*** is a contract that depends on an underlying entity (e.g., a stock)

Many Types of Financial Contracts are Traded

- Catastrophy Bonds
- FX Swaps
- Bermudan Options
- Inflation Swaps
- Interest Rate Swaps
- Variance Certificates
- Plain Vanilla Options
- Indexed Linked Bonds
- Credit Default Swaps (CDS)
- Rainbow Options
- Chooser Options
- Himalyan Options
- FX Options
- Forward Rate Agreements
- Callable Range Accrual Notes

How do the Banks Keep Track?

- **Many Problems:**
  - Financial contracts need management
    - ***fixings***, ***decisions***, ***corporate actions***, …
  - Banks must report daily on their total value of assets
  - Banks must control risk (counterparty risk, currency risk, …)
  - Banks need to know about future cash flows

- **A Solution:**
  - **Specify financial contracts in a domain specific language!**
  - **Use a functional programming language (e.g., ML)**

Algebraic properties

Simple reasoning

The SimCorp XpressInstruments Solution

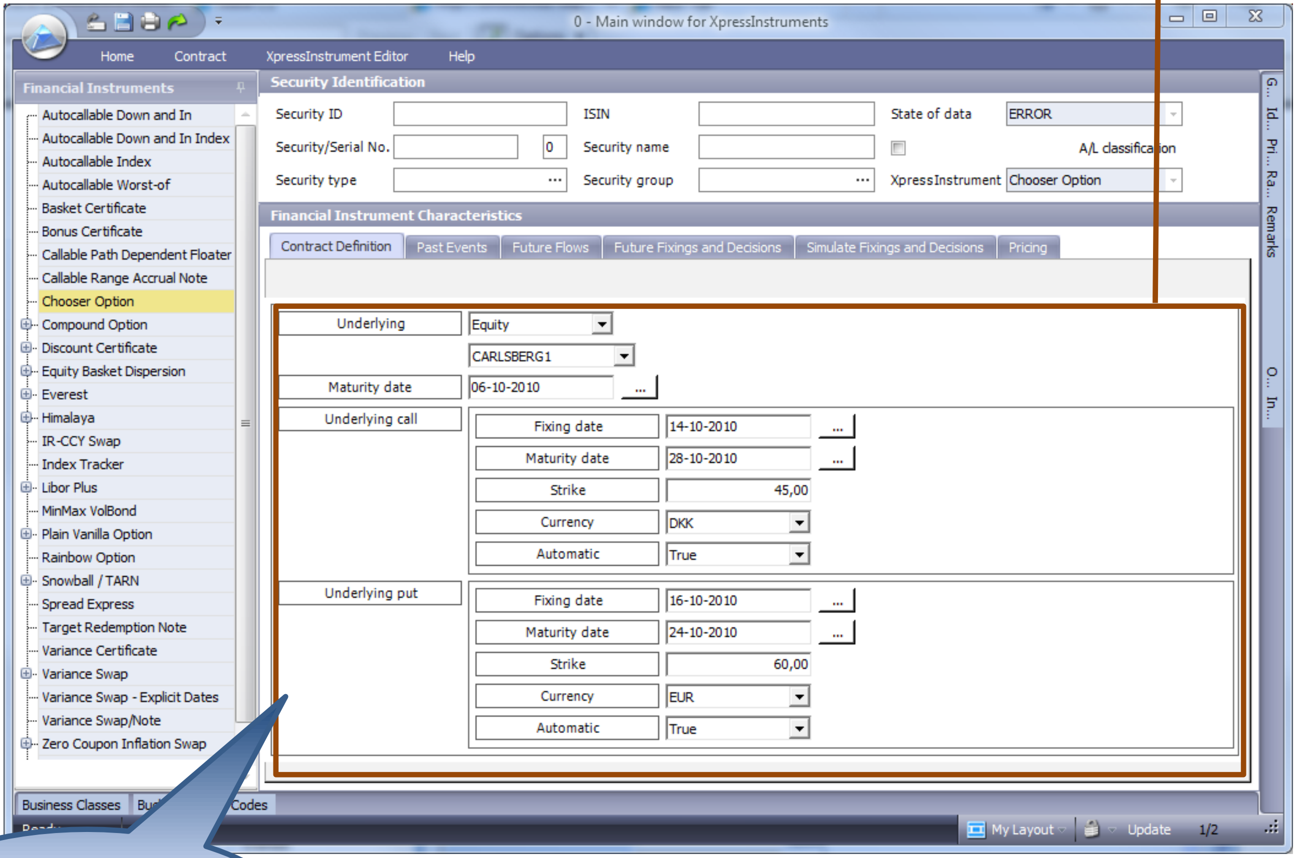
Instrument specific input

Instruments are specified in an ***instrument modeling language***

Once loaded, a **portfolio manager** may **instantiate** an instrument to create contracts.

The **instrument knows** what input to ask for.

Wall-to-wall (Front Office, Middle Office, Back Office) contract management



- Autocallable Down and In
- Autocallable Down and In Index
- Autocallable Index
- Autocallable Worst-of
- Basket Certificate
- Bonus Certificate
- Callable Path Dependent Floater
- Callable Range Accrual Note
- Chooser Option
- Compound Option
- Discount Certificate
- Equity Basket Dispersion
- Everest
- Himalaya
- IR-CCY Swap
- Index Tracker
- Libor Plus
- MinMax VolBond
- Plain Vanilla Option
- Rainbow Option
- Snowball / TARN
- Spread Express
- Target Redemption Note
- Variance Certificate
- Variance Swap
- Variance Swap - Explicit Dates
- Variance Swap/Note
- Zero Coupon Inflation Swap

Security ID
ISIN
State of data: ERROR
Security/Serial No.: 0
Security name
☐ A/L classification
Security type
Security group
XpressInstrument: Chooser Option

Underlying: Equity
CARLSBERG1
Maturity date: 06-10-2010

Underlying call
Fixing date: 14-10-2010
Maturity date: 28-10-2010
Strike: 45,00
Currency: DKK
Automatic: True

Underlying put
Fixing date: 16-10-2010
Maturity date: 24-10-2010
Strike: 60,00
Currency: EUR
Automatic: True

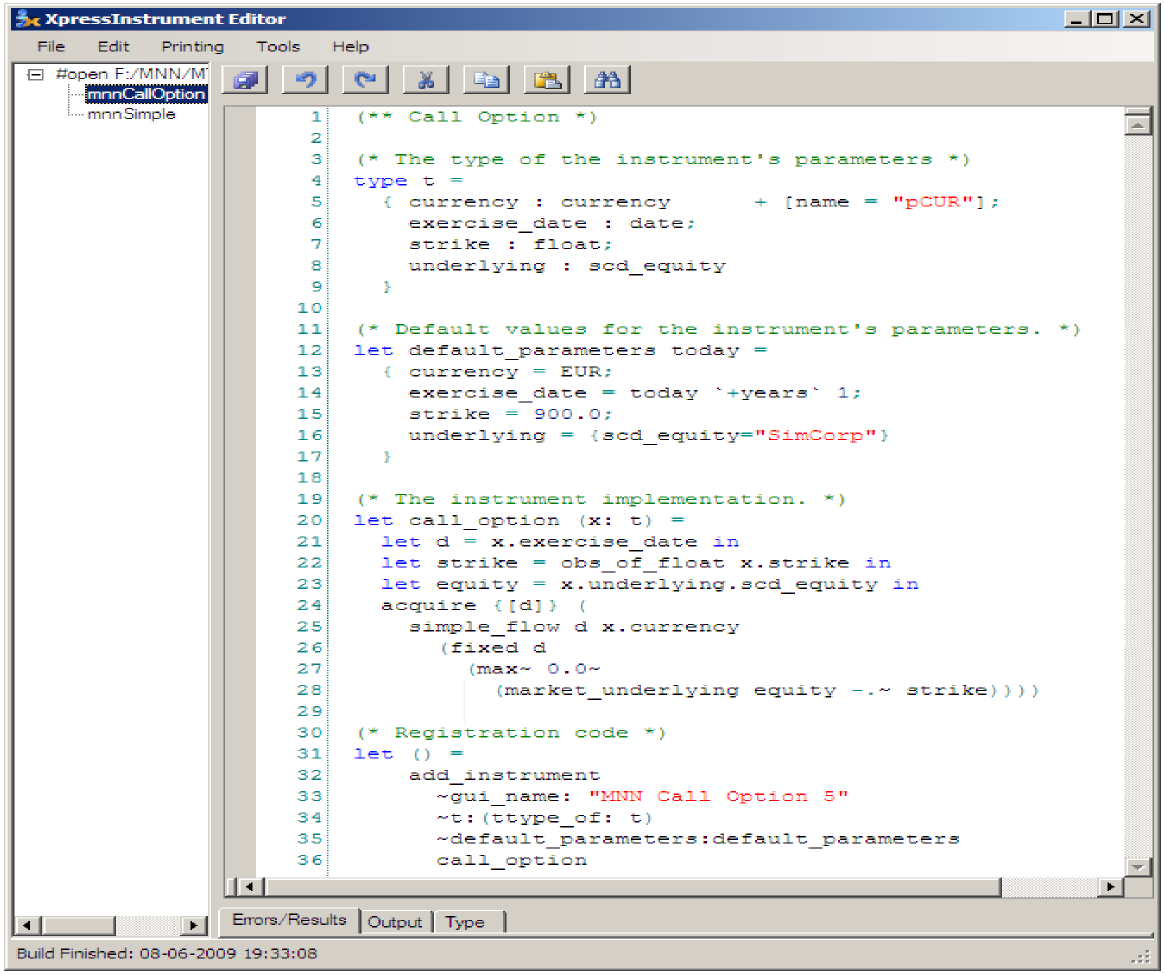
LexiFi Technology Inside!!

SimCorp

The SimCorp XpressInstruments Solution

Instrument IDE

- Instruments are **written** by SimCorp consultants or by banks themselves:
- Newly written instruments may be **loaded** into the system **instantaniously**
- **Notice: Arbitrary short time-to-market**



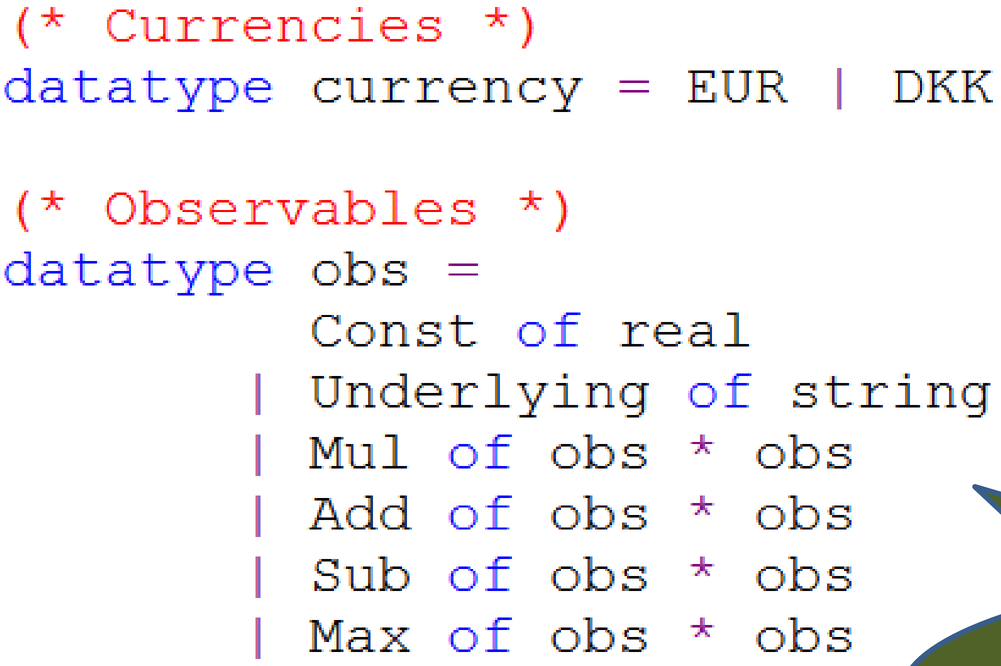
```
(** Call Option *)

(* The type of the instrument's parameters *)
type t =
  { currency : currency      + [name = "pCUR"];
    exercise_date : date;
    strike : float;
    underlying : scd_equity
  }

(* Default values for the instrument's parameters. *)
let default_parameters today =
  { currency = EUR;
    exercise_date = today `+years` 1;
    strike = 900.0;
    underlying = {scd_equity="SimCorp"}
  }

(* The instrument implementation. *)
let call_option (x: t) =
  let d = x.exercise_date in
  let strike = obs_of_float x.strike in
  let equity = x.underlying.scd_equity in
  acquire {[d]} (
    simple_flow d x.currency
      (fixed d
        (max~ 0.0~
          (market_underlying equity -.~ strike))))

(* Registration code *)
let () =
    add_instrument
      ~gui_name: "MNN Call Option 5"
      ~t:(ttype_of: t)
      ~default_parameters:default_parameters
      call_option
```

```
Build Finished: 08-06-2009 19:33:08
```

Constructing Contract Management Software in Standard ML

Basics:

In reality, there are, of course, more currencies…



```
(* Currencies *)
datatype currency = EUR | DKK

(* Observables *)
datatype obs =
         Const of real
       | Underlying of string * Date.date
       | Mul of obs * obs
       | Add of obs * obs
       | Sub of obs * obs
       | Max of obs * obs
```

Observable: algebra over measurable time-changing entities (**e.g., Carlsberg stock**)

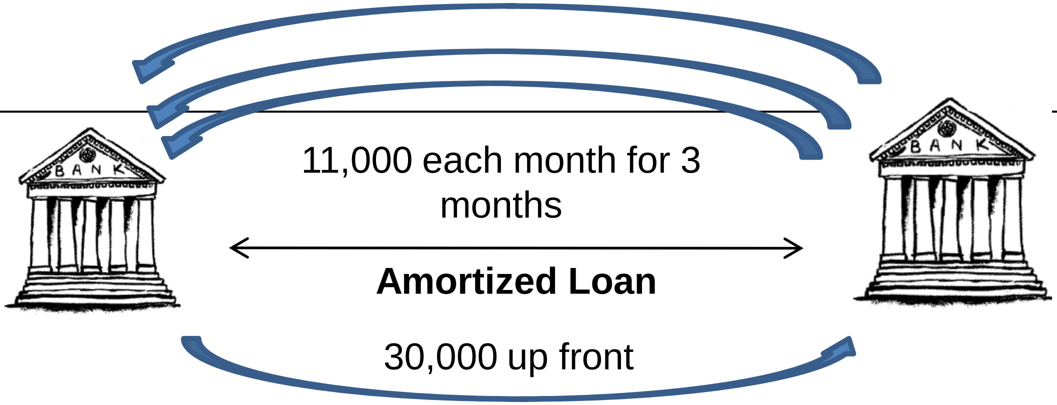
The Contract Language as a Standard ML Datatype

Flow of one unit

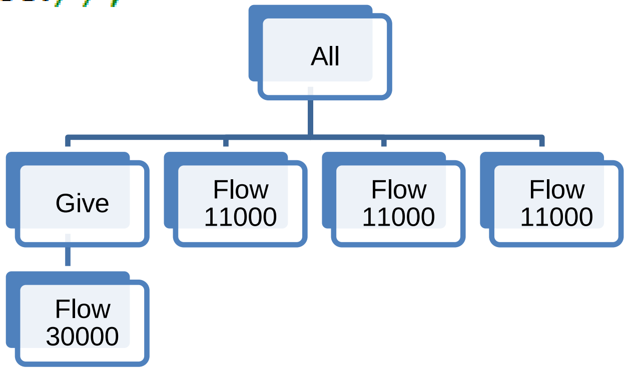
Acquire the underlying contract at specific date

```
(* Contracts *)
datatype contract =
         One of currency
       | Scale of obs * contract
       | All of contract list
       | Acquire of Date.date * contract
       | Give of contract

(* Shorthand notation *)
fun flow(d,v,c) = Acquire(d,Scale(Const v,One c))
val zero = All []
```

Example Financial Contracts



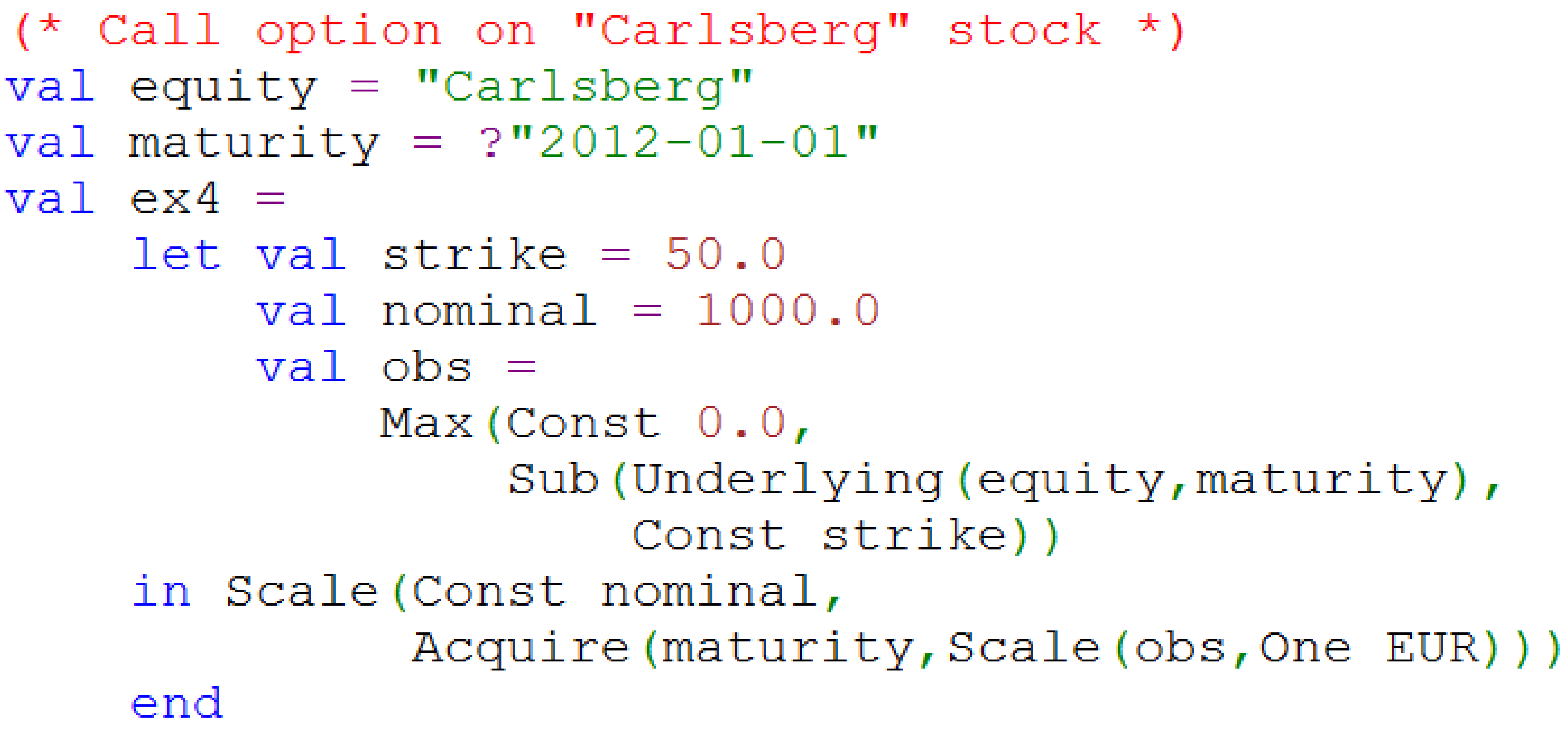
```
(* Simple amortized loan *)
val ex1 =
    let val coupon = 11000.0
        val principal = 30000.0
    in All [Give(flow(?"2011-01-01",principal,EUR)),
            flow(?"2011-02-01",coupon,EUR),
            flow(?"2011-03-01",coupon,EUR),
            flow(?"2011-04-01",coupon,EUR)]
    end
```



```
(* Cross currency swap *)
val ex2 =
    All [Give(
           All[flow(?"2011-01-01",7000.0,DKK),
               flow(?"2011-02-01",7000.0,DKK),
               flow(?"2011-03-01",7000.0,DKK)]),
         flow(?"2011-01-01",1000.0,EUR),
         flow(?"2011-02-01",1000.0,EUR),
         flow(?"2011-03-01",1000.0,EUR)]
```

Notice: flows in different currencies

A Somewhat more Complex Example



```
(* Call option on "Carlsberg" stock *)
val equity = "Carlsberg"
val maturity = ?"2012-01-01"
val ex4 =
    let val strike = 50.0
        val nominal = 1000.0
        val obs =
            Max(Const 0.0,
                Sub(Underlying(equity,maturity),
                    Const strike))
    in Scale(Const nominal,
             Acquire(maturity,Scale(obs,One EUR)))
    end
```

Meaning: Acquire at maturity the amount (in EUR), calculated as follows (*P* is price of Carlsberg stock at maturity):

nominal * max(0, *P* – *strike*)

Now What?

- We have now defined some contracts, but **what can we do with the definitions**?

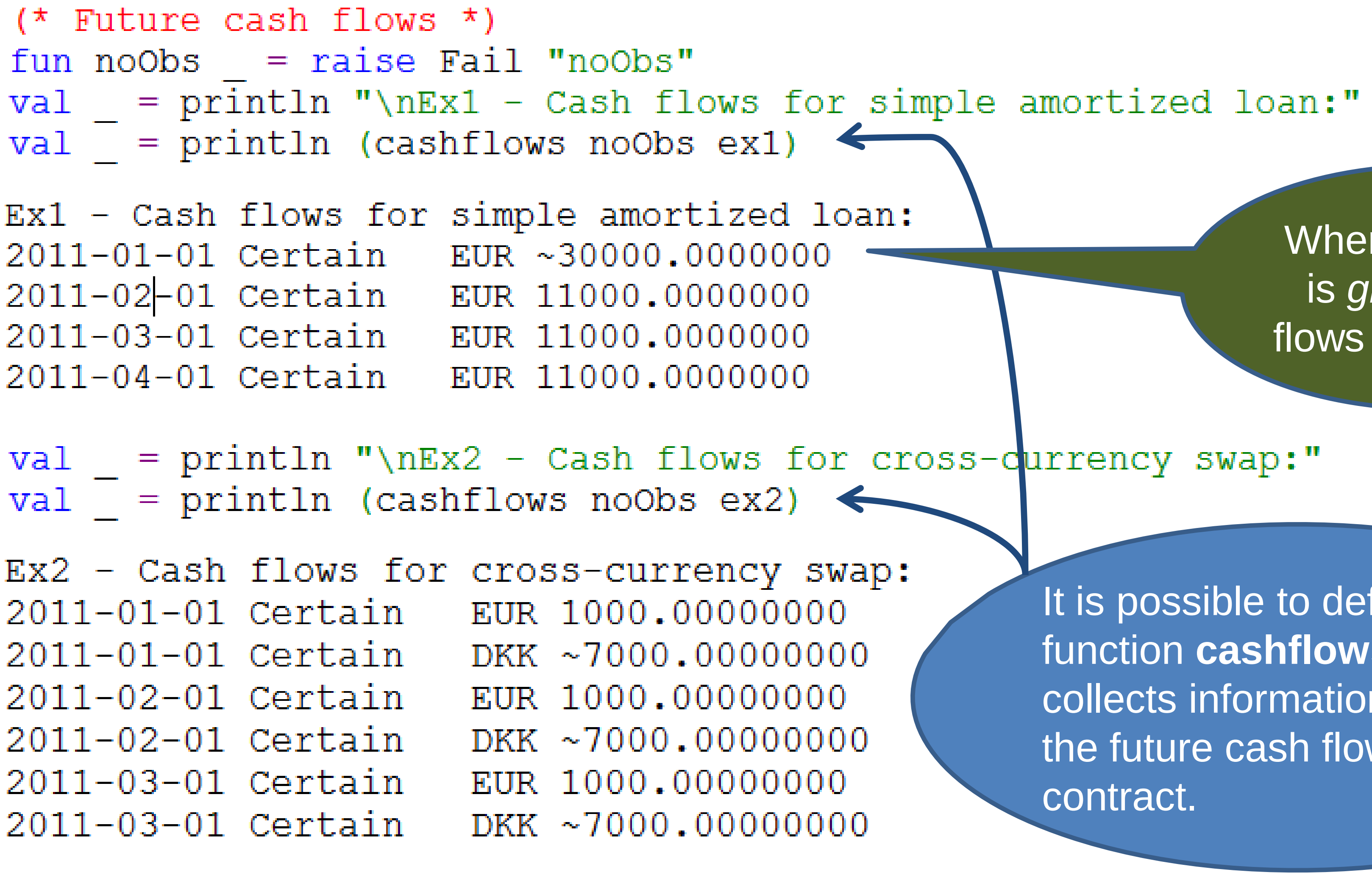
  – Report on the **expected future cash flows**

  – Perform **management operations**:
    - Advancement (simplify contract when time evolves)
    - Corporate action (stock splits, merges, catastrophic events, …)
    - Perform fixing (simplify contract when an underlying becomes known)

  – Report on the **value (price) of a contract**

  – …

Expected Future Cash Flows



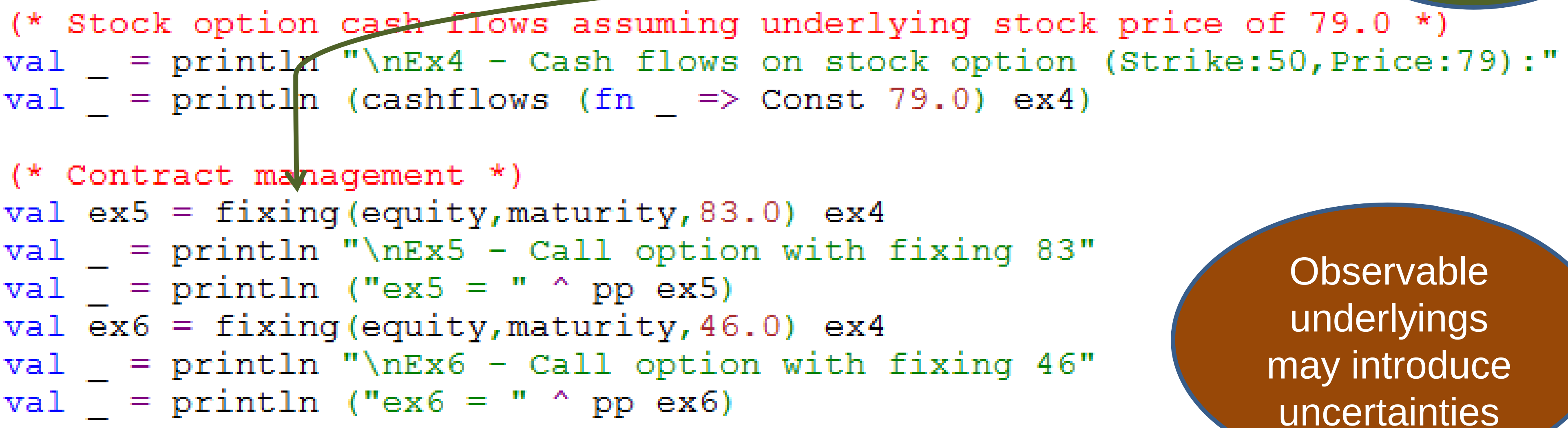
```
(* Future cash flows *)
fun noObs _ = raise Fail "noObs"
val _ = println "\nEx1 - Cash flows for simple amortized loan:"
val _ = println (cashflows noObs ex1)

Ex1 - Cash flows for simple amortized loan:
2011-01-01 Certain   EUR ~30000.0000000
2011-02-01 Certain   EUR 11000.0000000
2011-03-01 Certain   EUR 11000.0000000
2011-04-01 Certain   EUR 11000.0000000

val _ = println "\nEx2 - Cash flows for cross-currency swap:"
val _ = println (cashflows noObs ex2)

Ex2 - Cash flows for cross-currency swap:
2011-01-01 Certain   EUR 1000.00000000
2011-01-01 Certain   DKK ~7000.00000000
2011-02-01 Certain   EUR 1000.00000000
2011-02-01 Certain   DKK ~7000.00000000
2011-03-01 Certain   EUR 1000.00000000
2011-03-01 Certain   DKK ~7000.00000000
```

When a contract is *given* away, flows are inverted

It is possible to define a function **cashflows** that collects information about the future cash flows of a contract.

Contract Management and Contract Simplification

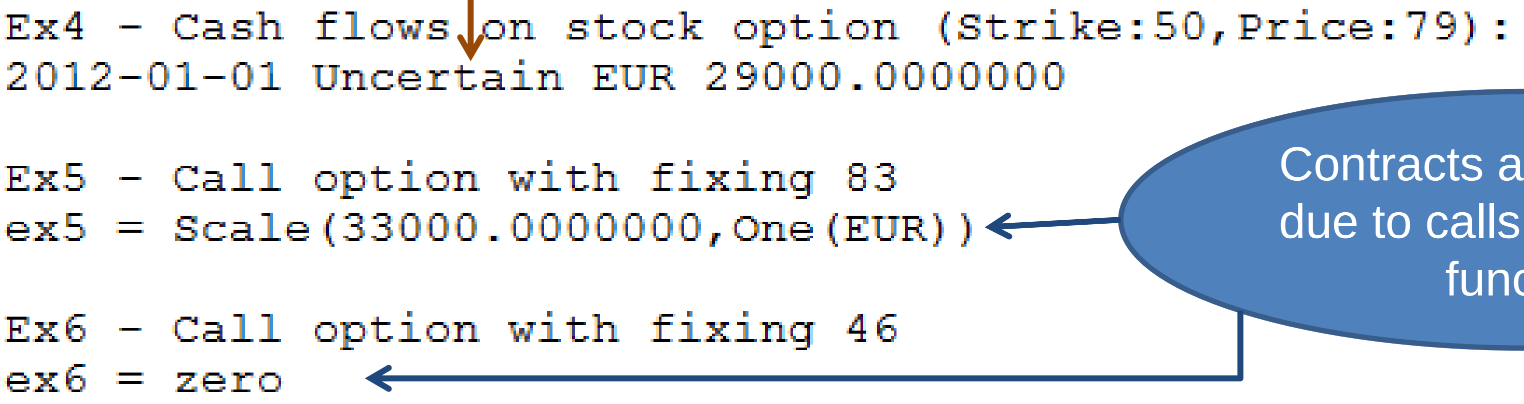
Fixing also advances contract



```
(* Stock option cash flows assuming underlying stock price of 79.0 *)
val _ = println "\nEx4 - Cash flows on stock option (Strike:50,Price:79):"
val _ = println (cashflows (fn _ => Const 79.0) ex4)

(* Contract management *)
val ex5 = fixing(equity,maturity,83.0) ex4
val _ = println "\nEx5 - Call option with fixing 83"
val _ = println ("ex5 = " ^ pp ex5)
val ex6 = fixing(equity,maturity,46.0) ex4
val _ = println "\nEx6 - Call option with fixing 46"
val _ = println ("ex6 = " ^ pp ex6)
```

Observable underlyings may introduce uncertainties

Output:



```
Ex4 - Cash flows on stock option (Strike:50,Price:79):
2012-01-01 Uncertain EUR 29000.0000000

Ex5 - Call option with fixing 83
ex5 = Scale(33000.0000000,One(EUR))

Ex6 - Call option with fixing 46
ex6 = zero
```

Contracts are simplified due to calls to the fixing function

Valuation (pricing)

```
(* Valuation (Pricing) *)
structure FlatRate = struct
  fun discount d0 d amount rate =
      let val time = real(Date.diff d d0) / 360.0
      in amount * Math.exp(~ rate * time)
      end
  fun price d0 (R : currency -> real)
               (FX: currency * real -> real) t =
      let val flows = cashflows0 noE t
      in List.foldl (fn ((d,cur,v,_),acc) =>
                        acc + FX(cur,discount d0 d v (R cur)))
                    0.0 flows
      end
end
```

Notice: This model is a bit too simple – we assume the FX-rate is constant…

```
fun FX(EUR,v)  = 7.0 * v
  | FX(DKK,v)  = v
fun R EUR = 0.04
  | R DKK = 0.05

val p1 = FlatRate.price (?"2011-01-01") R FX ex1
val p2 = FlatRate.price (?"2011-01-01") R FX ex2
val _ = println("\nPrice(ex1) : DKK " ^ Real.toString p1)
val _ = println("\nPrice(ex2) : DKK " ^ Real.toString p2)
```

Output:

```
Price(ex1) : DKK 19465.9718165

Price(ex2) : DKK 17.3909947790
```

What is Missing?

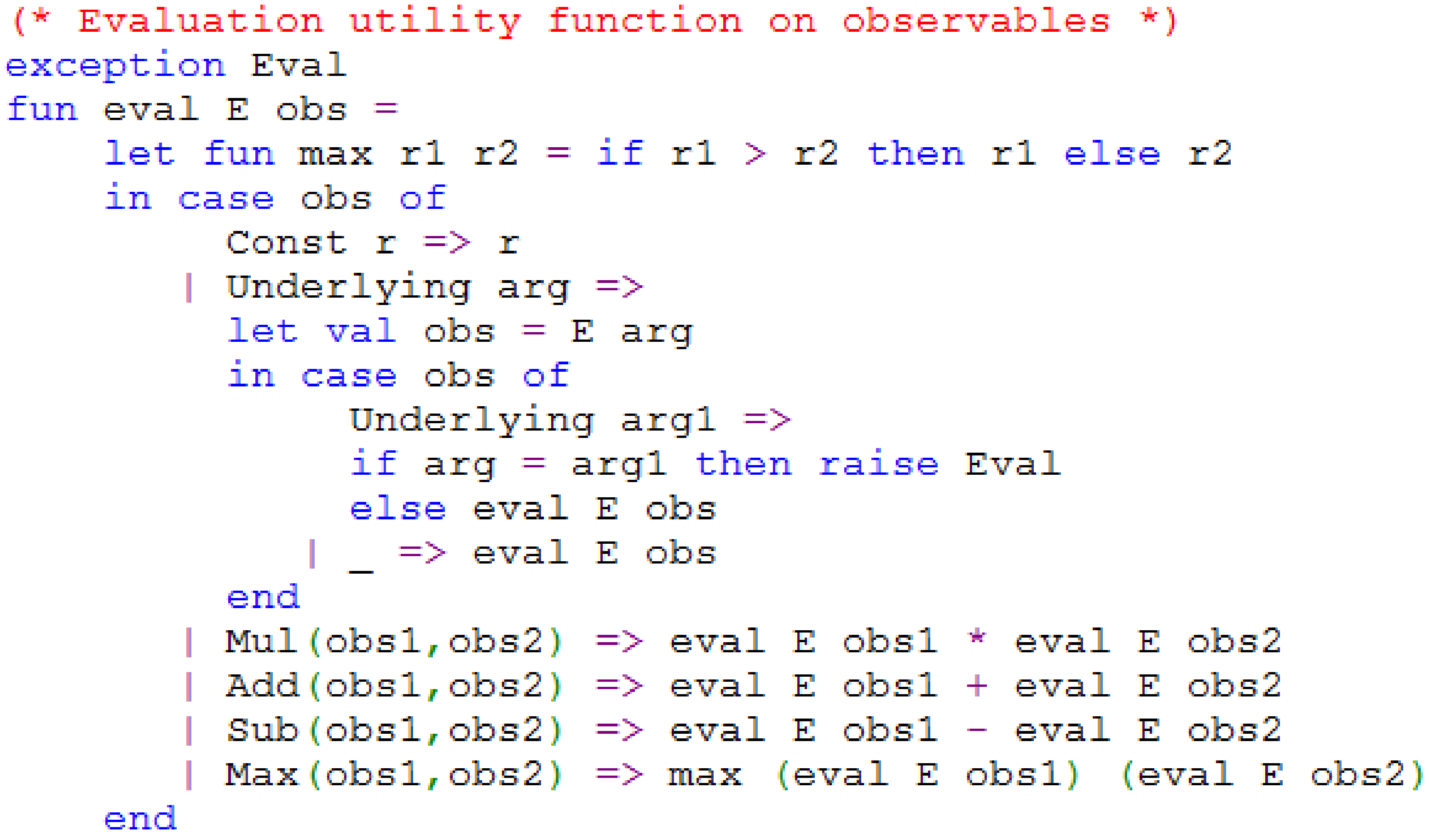
- Proper **date handling** (holidays, business conventions; Act/30, Act/Act, …)
- **Easy GUI** specification
- More **combinators** (e.g., american optionality, dynamic dates, …)
- More **functionality** (e.g., accrual interest)
- Support for **corporate actions** and **catastrophic events**
- **Well-formedness** of contracts… *Disallow acquire of flow in the past*
- Proper **stochastic models** and underlying machinery (**Sobol sequences** for **monte-carlo simulations**) for pricing and calibration
  - Support for linking with external models (e.g., FINCAD)

Conclusions

- **Functional programming**
  - Is **declarative**: **Focus on what** instead of how
  - Is **value oriented** (functional, persistent data structures)
  - Eases **reasoning** (formal as well as informal)
  - Eases **concurrent processing** (e.g., for improved parallelism)

- **SimCorp** not the only company (or bank) that has recognized the value of functional programming for the financial industry
  - LexiFi (See ICFP’00 paper by Peyton-Jones, Eber, Seward)
    - **Engine is used by SimCorp!**
  - Jane Street Capital (focus on electronic trading)
  - Societe Generale, Credit Suisse, Standard Chartered, …
  - Contract ”Pay-off” specifications are often written in a functional style

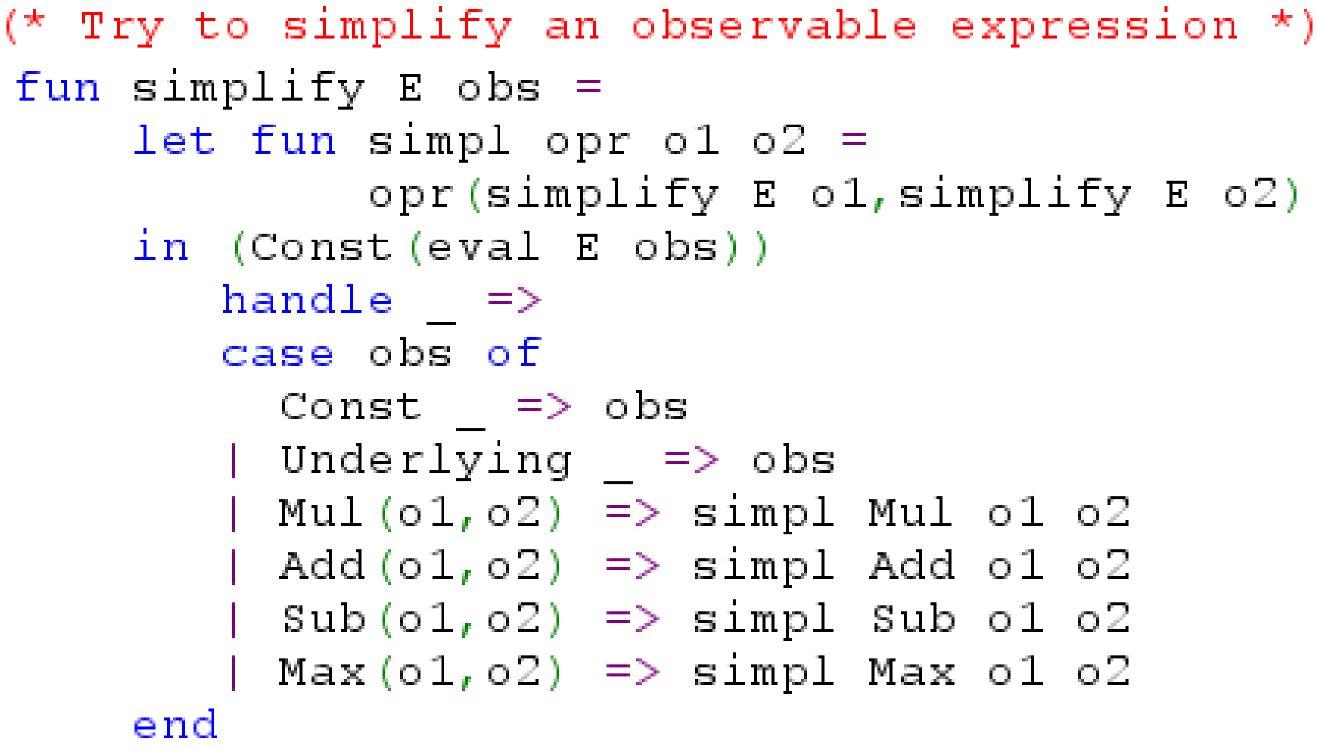
Appendix: Observable evaluation function



```
(* Evaluation utility function on observables *)
exception Eval
fun eval E obs =
    let fun max r1 r2 = if r1 > r2 then r1 else r2
    in case obs of
         Const r => r
       | Underlying arg =>
         let val obs = E arg
         in case obs of
              Underlying arg1 =>
              if arg = arg1 then raise Eval
              else eval E obs
            | _ => eval E obs
         end
       | Mul(obs1,obs2) => eval E obs1 * eval E obs2
       | Add(obs1,obs2) => eval E obs1 + eval E obs2
       | Sub(obs1,obs2) => eval E obs1 - eval E obs2
       | Max(obs1,obs2) => max (eval E obs1) (eval E obs2)
    end
```

SimCorp

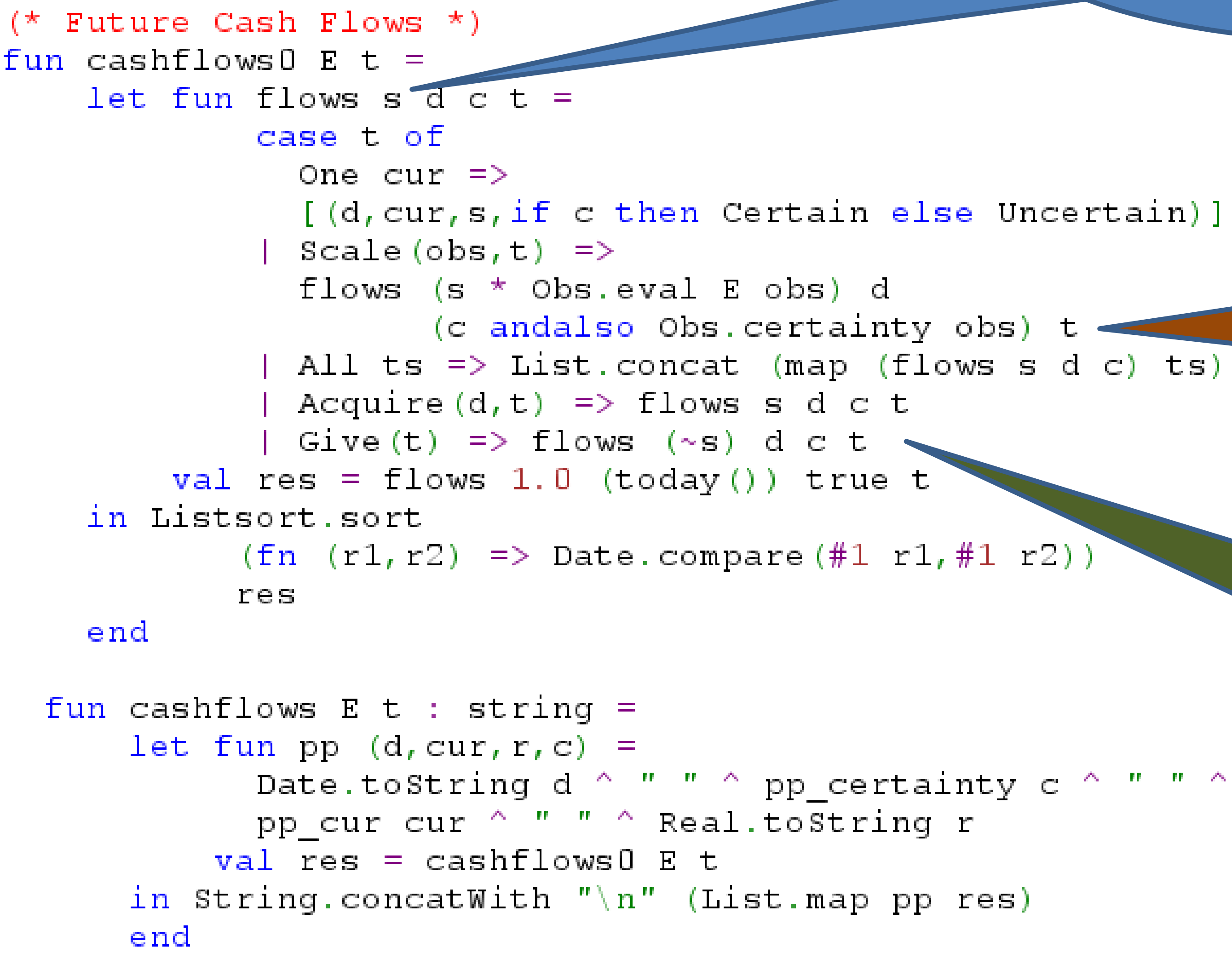
Appendix: Observable Simplification – preparing for Contract Management



```
(* Try to simplify an observable expression *)
 fun simplify E obs =
     let fun simpl opr o1 o2 =
             opr(simplify E o1,simplify E o2)
     in (Const(eval E obs))
        handle _ =>
        case obs of
          Const _ => obs
        | Underlying _ => obs
        | Mul(o1,o2) => simpl Mul o1 o2
        | Add(o1,o2) => simpl Add o1 o2
        | Sub(o1,o2) => simpl Sub o1 o2
        | Max(o1,o2) => simpl Max o1 o2
     end
```

Appendix: Future Cash Flows

Propagate scale factor to resolve amount

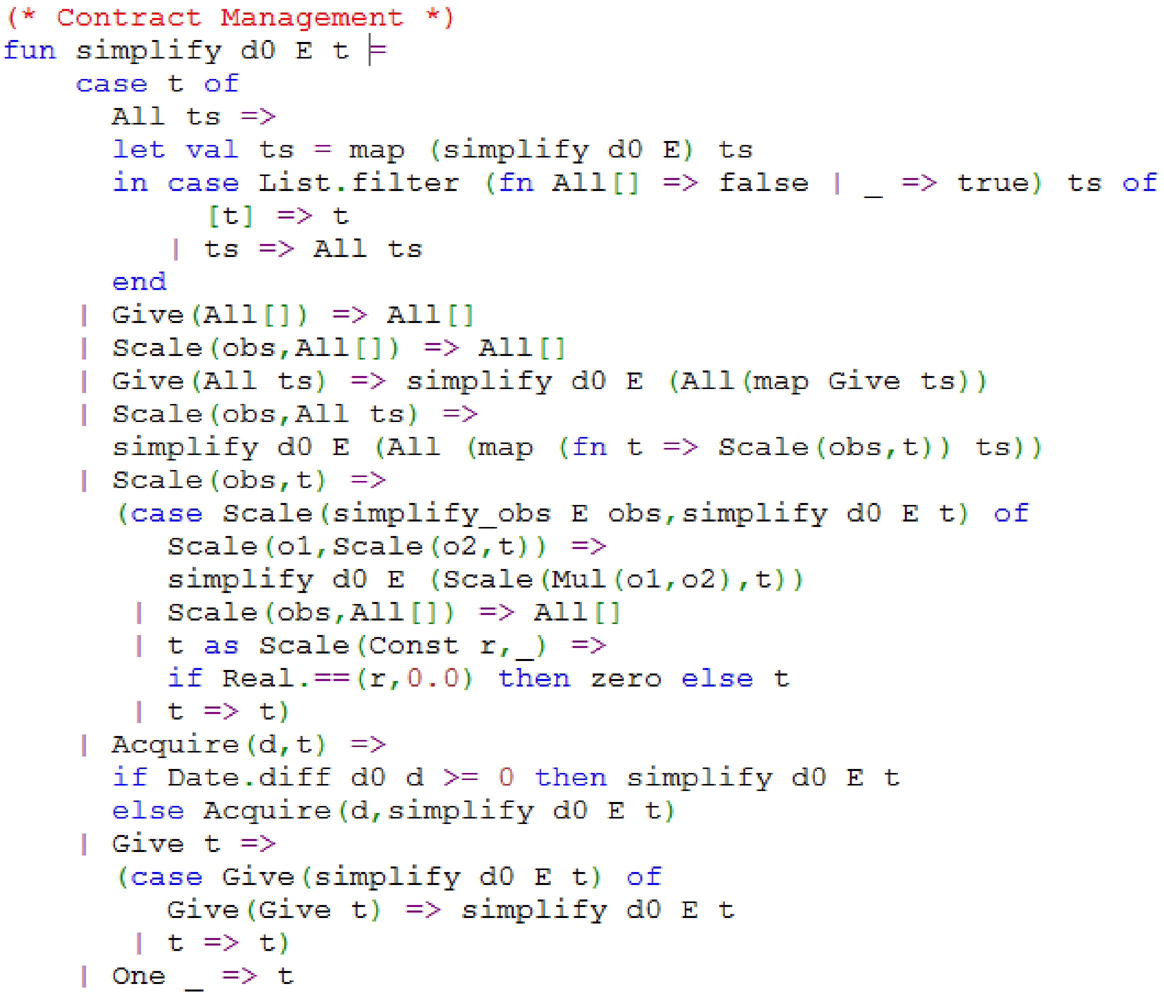


```
(* Future Cash Flows *)
fun cashflows0 E t =
    let fun flows s d c t =
            case t of
              One cur =>
              [(d,cur,s,if c then Certain else Uncertain)]
            | Scale(obs,t) =>
              flows (s * Obs.eval E obs) d
                    (c andalso Obs.certainty obs) t
            | All ts => List.concat (map (flows s d c) ts)
            | Acquire(d,t) => flows s d c t
            | Give(t) => flows (~s) d c t
        val res = flows 1.0 (today()) true t
    in Listsort.sort
           (fn (r1,r2) => Date.compare(#1 r1,#1 r2))
           res
    end

  fun cashflows E t : string =
      let fun pp (d,cur,r,c) =
            Date.toString d ^ " " ^ pp_certainty c ^ " " ^
            pp_cur cur ^ " " ^ Real.toString r
          val res = cashflows0 E t
      in String.concatWith "\n" (List.map pp res)
      end
```

Observable underlyings may introduce uncertainties

When a contract is given away, flows are inverted

Appendix: Contract Simplification



```
(* Contract Management *)
fun simplify d0 E t =
    case t of
      All ts =>
      let val ts = map (simplify d0 E) ts
      in case List.filter (fn All[] => false | _ => true) ts of
           [t] => t
         | ts => All ts
      end
    | Give(All[]) => All[]
    | Scale(obs,All[]) => All[]
    | Give(All ts) => simplify d0 E (All(map Give ts))
    | Scale(obs,All ts) =>
      simplify d0 E (All (map (fn t => Scale(obs,t)) ts))
    | Scale(obs,t) =>
      (case Scale(simplify_obs E obs,simplify d0 E t) of
         Scale(o1,Scale(o2,t)) =>
         simplify d0 E (Scale(Mul(o1,o2),t))
       | Scale(obs,All[]) => All[]
       | t as Scale(Const r,_) =>
         if Real.==(r,0.0) then zero else t
       | t => t)
    | Acquire(d,t) =>
      if Date.diff d0 d >= 0 then simplify d0 E t
      else Acquire(d,simplify d0 E t)
    | Give t =>
      (case Give(simplify d0 E t) of
         Give(Give t) => simplify d0 E t
       | t => t)
    | One _ => t
```

Complete contract simplifier.

Scale and Give constructors are propagated downwards and merged.

Acquire constructors are resolved, given the argument date to simplify (d0).

The environment (E) is propagated to the observable simplifier.

Appendix: Contract Management Using ”simplify”

```
(* Apply a fixing to a contract *)
fun fixing (name,date,value) t =
    let fun E arg =
            if arg = (name,date) then Obs.Const value
            else Obs.Underlying arg
    in simplify date E t
    end

(* Remove the past from a contract *)
fun advance d t =
    let val t = simplify d noE t
        fun adv t =
            case t of
              One _ => zero
            | Scale(obs,t) => Scale(obs, adv t)
            | Acquire _ => t
            | Give t => Give(adv t)
            | All ts => All(map adv ts)
    in simplify d noE (adv t)
    end
```