Beyond Valuation: Past, Present and Future of Domain Specific Languages for Finance Applications: Ten Years of DSL Development, Client Interaction, and Market Feedback at LexiFi

Jean-Marc Eber

LexiFi, www.lexifi.com

DSLFIN: Domain-Specific Languages for Financial Systems
ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems
October 1st, 2013, Miami Beach

Main Driver for Founding LexiFi

Question:

Has the financial industry adopted a method for rigorously describing financial contracts in a computer system?

Response:

No!

Therefore:

Need for a semantically well founded and mechanizable description approach.

The AIG bonus affair

"AIGFP's books also contain a significant number of complex—so-called bespoke —transactions that are difficult to understand and manage. This is one reason replacing key traders and risk managers would not be practical on a large scale. Personal knowledge of the trades and the unique systems at AIGFP will be critical to an effective unwind of AIGFP's businesses and portfolios. [. . .] To the extent that AIGFP were to lose traders who currently oversee complicated though familiar positions and know how to hedge the book, gaps in hedging could result in significant losses. [. . .] Quite frankly, AIG's hands are tied."

From a letter dated 14 March 2009 by Edward Liddy, Chairman and CEO of AIG to Timothy Geithner, U.S. Secretary of the Treasury, regarding compensation at AIG Financial Products.

A language for describing derivatives?

"One of the challenges that we need to address [. . .] is to have a common language to describe derivatives. Every firm uses a different set of terminologies, a different set of representations to describe their derivatives portfolios."

Kenneth Griffin, Founder and CEO, Citadel Investment Group, testifying before the U.S. House Committee on Oversight and Government Reform on November 13, 2008.

Industry Practice

- Contracts are often complex: rights and obligations depend heavily on preceding choices and realizations
- Biggest profit margin in the “tailored product” segment
- Contracts often last for a long time (e.g., 3, 5, 10 years)
- Different treatments must be performed on contracts (valuation, risk analysis, operational management, reporting,...), from inception to maturity
- A financial institution has hundreds or thousands of these contracts on its books, managed by a small group of people (Excel, PDF, post-its,...)
- Computer programs at financial institutions either ignore the "semantics" of contracts (they only record a few salient features) or require heavy error-prone developments for each structure and for each treatment

Term sheets

Term Sheets are informal contract specifications. They contain at least one error (Fundamental Axiom of Financial Engineering), as they typically

- are incomplete
- lack precision
- are contradictory
- contain implicit rules

Business concerns

Financial institutions look for systems that enable them to design, price and process heterogeneous financial products in a framework that scales while containing risks and costs.

- Automation of the full product life cycle, from prototyping to maturity
- Includes contract documentation, automatic ratings, term sheets generation,...
- Agreement on contract: adequacy, quality, and accuracy
  - contracts must be executed precisely between two parties
  - kind of “handshaking protocol”
- Contracts must be hedged, their risk profile must be analyzed, etc.: the valuation problem is crucial
- For some applications, well chosen contract approximations (simplifications) can be used
- Oversight: financial industry is (and will be more and more) regulated

The challenge

Create a product definition that can be read by a human being and processed by a computer, and that satisfies three goals:

- Describe the rights and obligations of the parties both precisely and exhaustively
- Contains therefore all semantics of the contract
- Lend itself to manipulations of various sorts, for example, for the purpose of pricing the contract and evaluating credit risk, or managing its clauses automatically
- Reflect the evolution of the contract through time

Domain Specific Language (DSL)

DSL: Contract Algebra

Heterogeneity of contracts: the contract description

- should be compositional
  - conceptual simplicity (semantics, “traversal” algorithms)
  - software engineering reuse argument (library design)
- should be a value, which cannot only be analyzed but also modified and transformed

Syntax is minor, it’s all about semantics!

Example

```
1  let c =
2    acquire {[2010-01-02]}
3      (either
4        ["exercise", flow 2010-01-10 EUR ((market "XYZ" -.~
             3500.~));
5         "abandon", zero]
6      )
```

on 2010-01-02, holder can choose between

- receiving on 2010-01-10 a flow, in EUR, equal to the quotation of stock XYZ on this same date minus 3500, or
- nothing

A (cash-settled, for simplicity) call with discretionary exercise decision

Previous definition

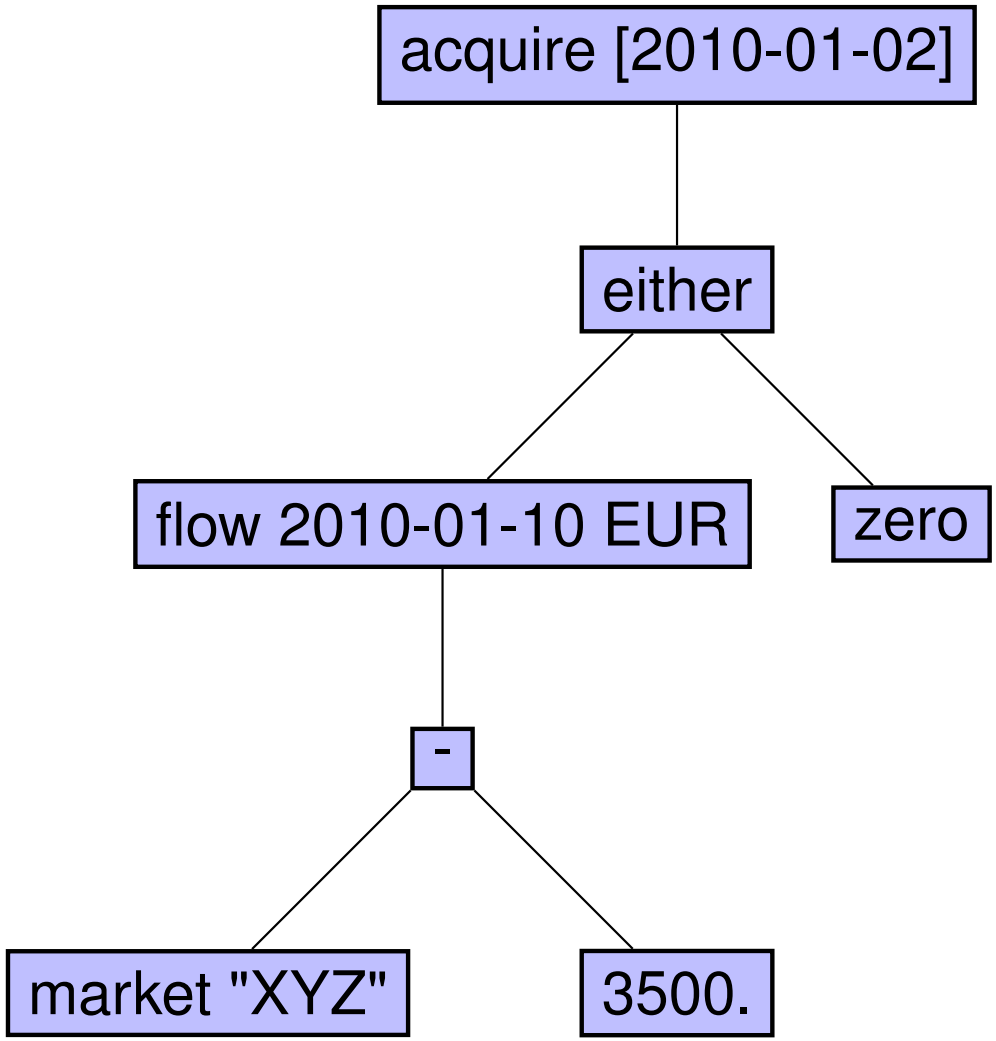
- No reference to any pricing model
- No mathematical notions
- No reference to any `max` operator
  - even if most pricing algorithms will (explicitly or implicitly) "use" a `max`
  - but one could be interested in a sub-optimal pricing in some circumstances
  - pricing scripts do not distinguish discretionary and "automatic" exercise
- Self-contained contract logic description

Semantics

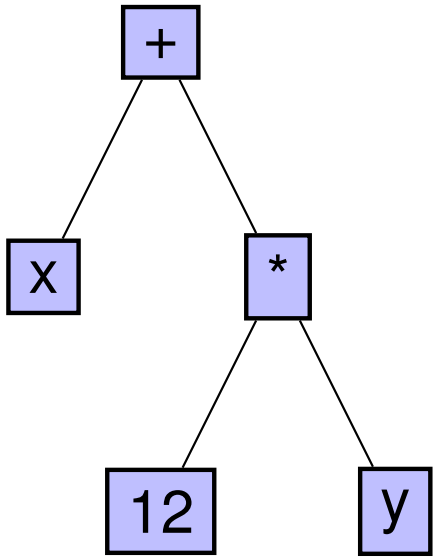
Knowing the meaning of the operators (`acquire`, `either`, ...), we deduce "how the contract works".

Analogy between contracts and algebraic expressions

”acquire {[2010-01-02]} (either [. . .])”



”$x + (12 * y)$”



Informal description of some combinators (simplified)

`acquire` `date -> contract -> contract`
Acquiring `acquire t c` means that you must acquire `c` at future date `t`

`all` `contract list -> contract`
Acquiring `all cs` means that you acquire immediately all contracts in `cs`

`either` `(string * contract) list -> contract`
Acquiring `either cs` means that you must acquire immediately one contract in `cs`

`flow` `date -> currency -> observable -> contract`
Acquiring `flow t cur o` means to receive at `t` the value at
of observable `o`

`give` `contract -> contract`
Acquiring `give c` is to acquire all of `c`'s rights as obligations, and vice versa

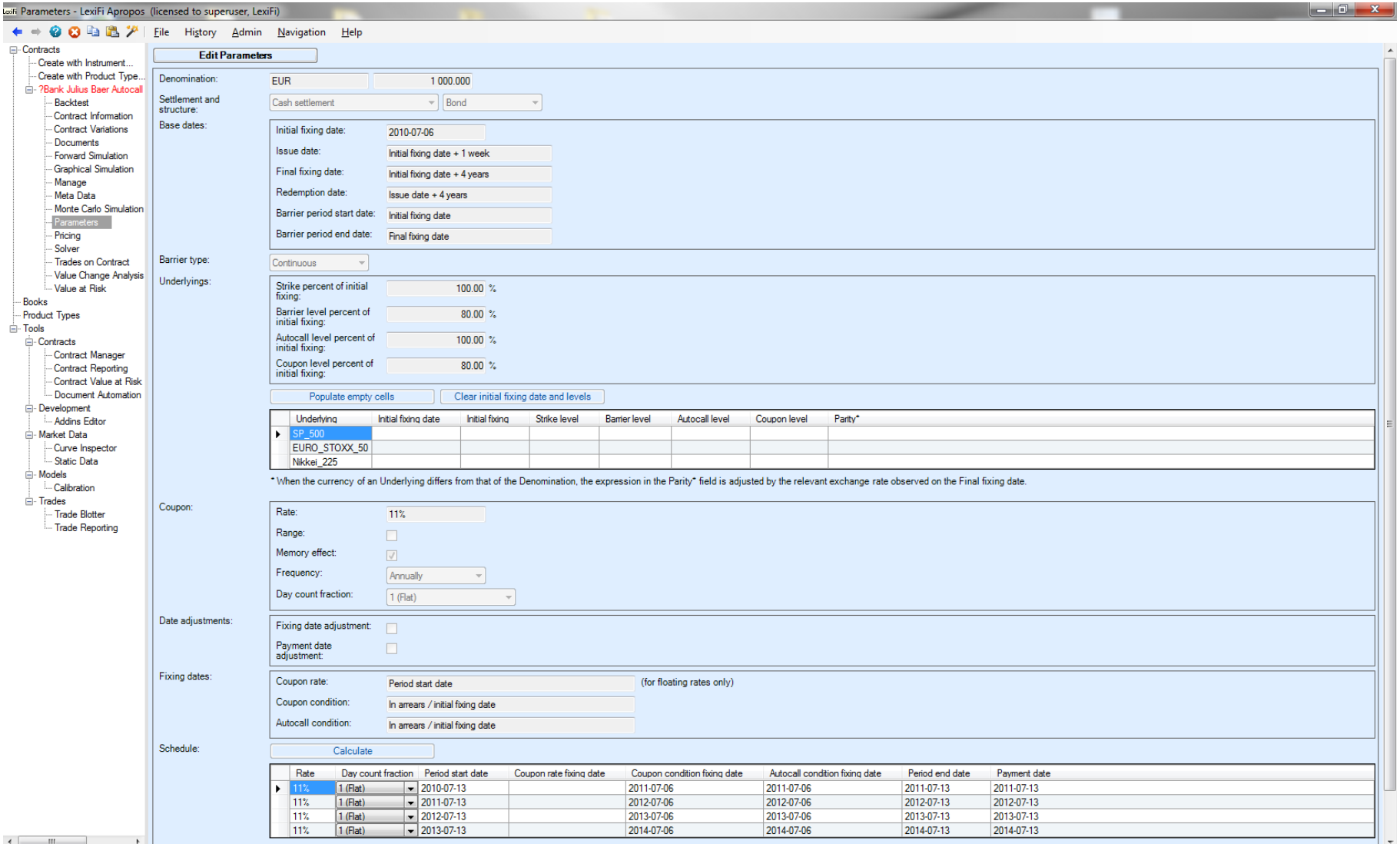
Technical Choices at LexiFi

- LexiFi puts great emphasis on controlling fully its implementation stack and remaining platform independent
  - Critical for ambitious integration projects
  - LexiFi's DSL runs (on client sites) on Windows desktops, on Windows servers, deployed on many Linux servers in big datacenters, etc. Even in the browser (internal experiments)!
- Most of our code is implemented in MLFi, a general purpose functional programming language derived from OCaml
  - Many of the recent advances in OCaml originated from or were funded by LexiFi
  - MLFi is used both a the host language for our contract DSL, but also for numerical code, general application development, etc. (Relying on FFI with C and with .Net where needed)

2-staged approach

- The DSL itself is made of contract and observable combinators
  - Combinators enforce invariants and provide a clean API above the highly-optimized internal representation of DSL terms
  - Restricted expressive power to support rich reasoning, precise analysis and powerful compilation techniques
- The host language (OCaml + extensions) is used to implement instruments, i.e. mapping from user-facing parameters to contracts
  - Full expressivity of the language. In theory, any host language would do the job, but a high-level functional language turns out to be a perfect match for implementing and using the DSL
  - We have developped a library of reusable instrument building blocks, using a “mixin” style
  - The instrument code also specifies properties not stricly related to the contract itself: customized GUI layout or behavior, choice of a default pricing model, hints about specific pricing tricks, etc.

Contract Parameters



Possible contract rewriting (simplification) rules

$$
\begin{array}{lcll}
\ldots & & & \\
all[zero; c_2; \ldots; c_n] & \rightarrow & all[c_2; \ldots; c_n] & \\
give(give\ c) & \rightarrow & c & \\
acquire\ t1\ (acquire\ t2\ c) & \rightarrow & acquire\ t2\ c & if\ t1 \leq t2 \\
 & \rightarrow & \text{"}error\text{"} & otherwise \\
\ldots & & &
\end{array}
$$

what set of equational rules to apply?

applicable set depends on uses (middle office, pricing, . . .)

Reasoning on contract definitions: collecting future cash flows

Similarly, we can write (and implement!) the equations for retrieving a description of the future cash flows of a contract

- Depending on our practical needs, we define a more or less precise semantics

Future flows

Annotated list of future cash flows

Future cash flows with uncertainty flag

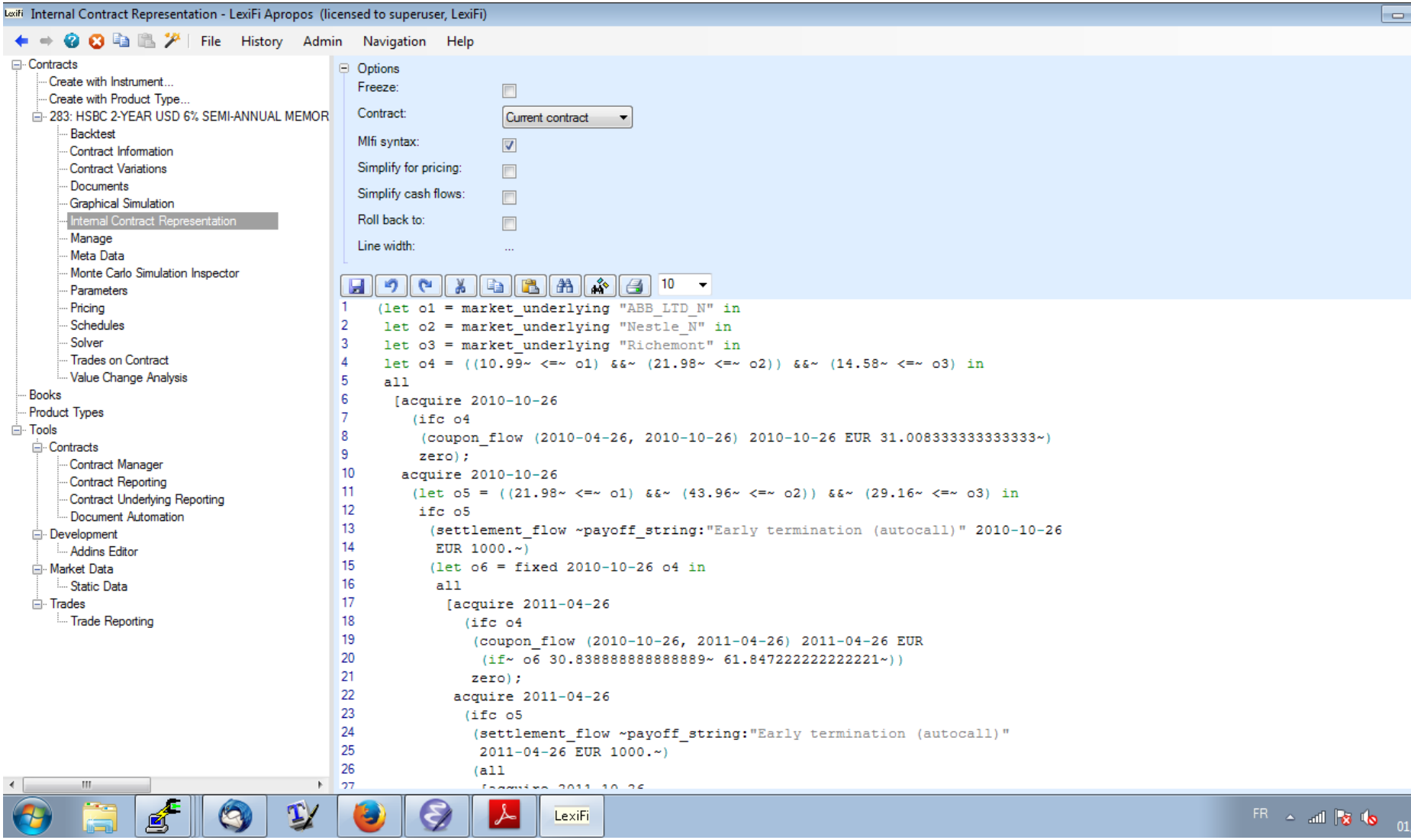
$$
\begin{array}{lcl}
[\![c]\!] & = & [\![\mathit{false}, c]\!]_f^+ \\
\\
[\![p, \mathit{flow}\ t\ \mathit{cur}\ o]\!]_f^+ & = & [(+, p, t, \mathit{cur}, o)] \\
[\![p, \mathit{flow}\ t\ \mathit{cur}\ o]\!]_f^- & = & [(-, p, t, \mathit{cur}, o)] \\
\\
[\![p, \mathit{either}[(s_1, c_1); \ldots; (s_n, c_n)]]\!]_f^+ & = & \cup_{i=1}^n [\![\mathit{true}, c_i]\!]_f^+ \\
[\![p, \mathit{either}[(s_1, c_1); \ldots; (s_n, c_n)]]\!]_f^- & = & \cup_{i=1}^n [\![\mathit{true}, c_i]\!]_f^- \\
\\
[\![p, \mathit{give}(c)]\!]_f^+ & = & [\![p, c]\!]_f^- \\
[\![p, \mathit{give}(c)]\!]_f^- & = & [\![p, c]\!]_f^+
\end{array}
$$

Calendar of a contract

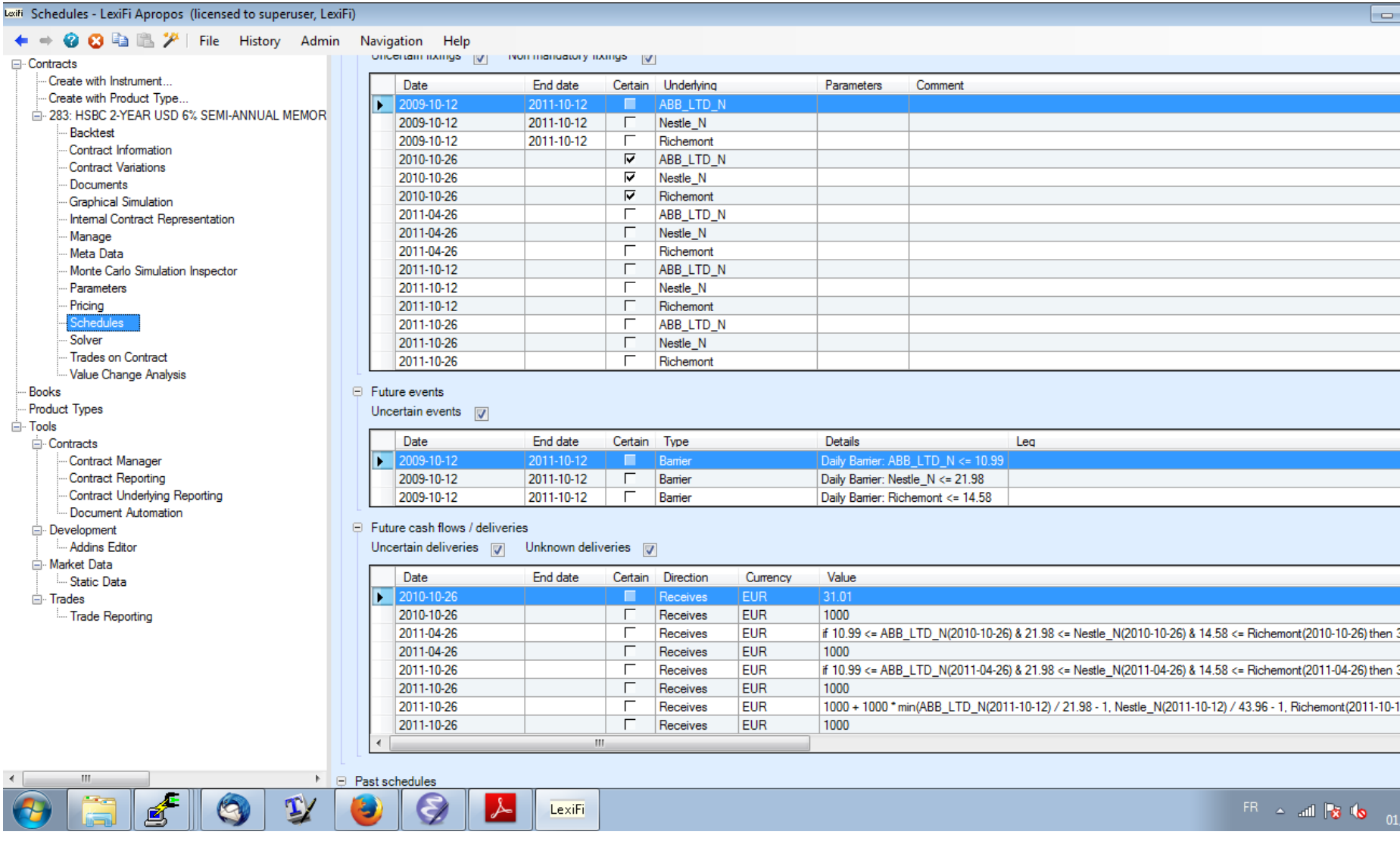
Detect all meaningful events that will or may happen in the future

- Similar (but more complex!) as defining the list of all unknowns in our algebraic expression example
  - recursive traversal of the syntax tree, collecting all the unknowns
- Compositional analysis of the syntax tree of the contract description

Contract Internal Representation

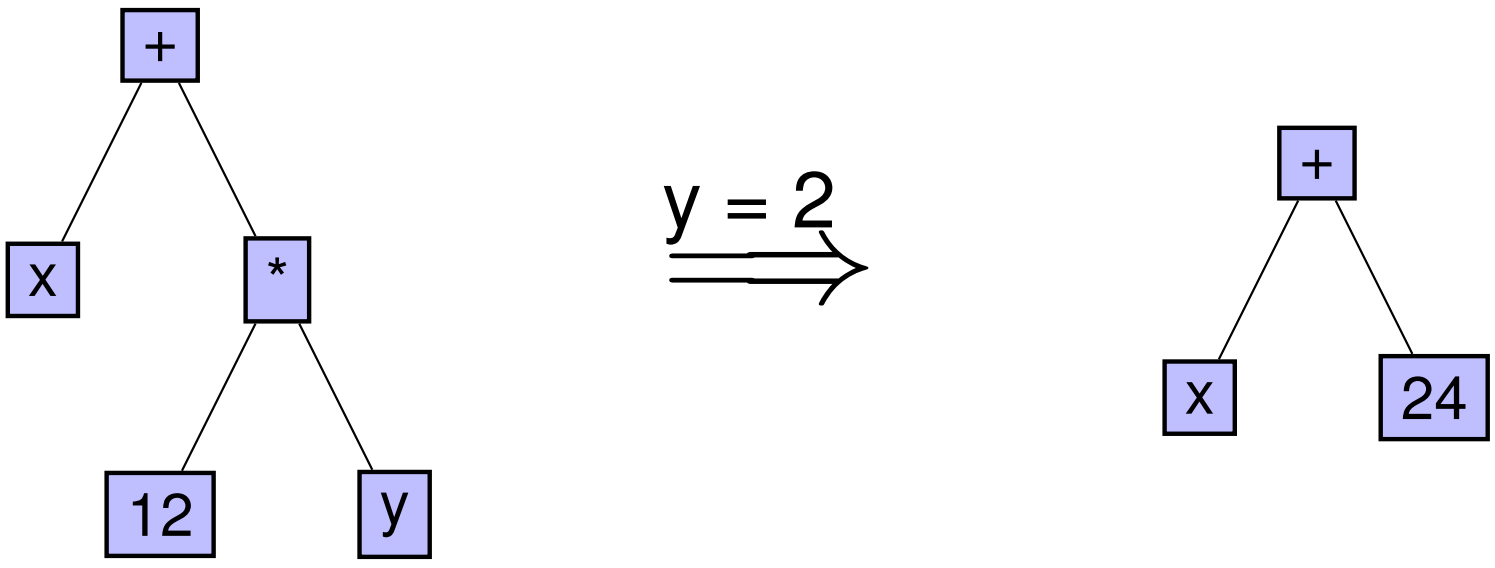


Contract Calendar



Managing a contract: operational semantics

Apply external events (fixings, exercise decisions,...) to the contract. Analogy: resolve an unknown and get a “simpler” expression



Fix an unknown observable

```
let c = manage c (Mgt_fixings[{
    fix_date = 2010-01-10;
    fix_identifier = "XYZ";
    fix_value = Obs_float 3650.}])
...
acquire {[2010-01-02]}
  (either
    [("abandon", zero);
     ("exercise", simple_flow 2010-01-10 EUR 150.~)])
```

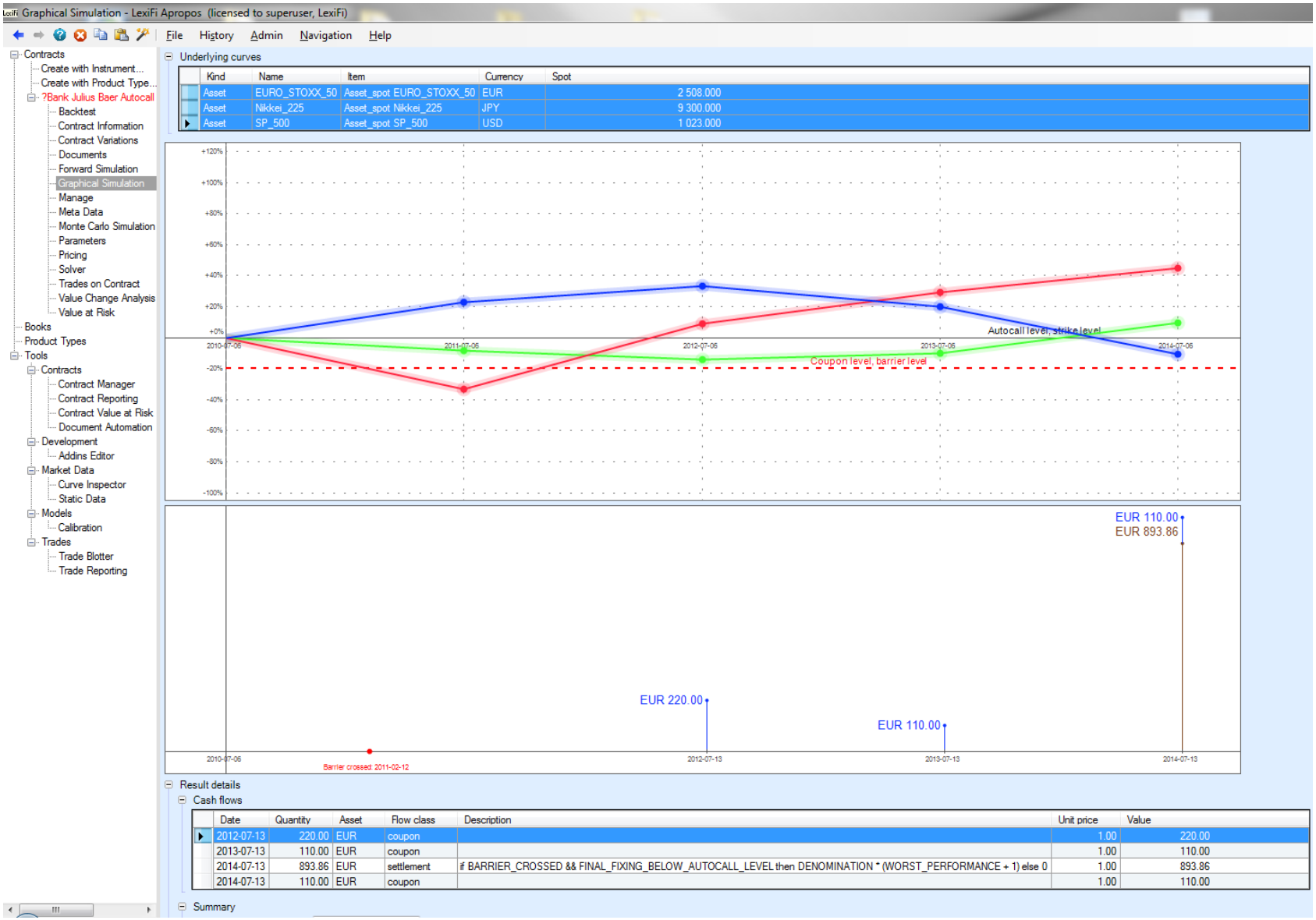
The resulting contract is “simplified”!

Symbolic contract manipulations

Inspection and transformation of contract descriptions has many more interesting applications:

- Decompose a contract into its “bond” and “option” part
- Symbolic netting or approximate netting
- Automatic payoff diagram generation
- Contract debugger
- Calculate (forthcoming) “regulatory” risk indicators
- . . .

Graphical Contract Simulator



Pricing code generation

Valuation is never fast enough as it implies resource hungry numerical calculations (Monte Carlo methods, Partial Differential Equations)

- Run-time code generation of the contract specific payoff part
- Clever caching approach
- Generate code from a “current contract state”
  - by design “in sync” with operational (management) semantics
  - generated code executes a succession of calculation steps, calling model specific primitives
  - usual compilation techniques and optimizations
  - domain specific optimizations

Functional Programming Techniques at LexiFi

Maybe one of the less exotic use of functional programming in the industry!
No fancy web or cloud programming, computer graphics, hardware design, big data, etc. Just a domain specific language, with

- static analysis (to report possible future events)
- rewriting and simplification (to incorporate past events)
- interpretation and compilation (to produce efficient low-level pricing code linked with numerical models)
- run-time type representations (to generate a GUI automatically from types)

LexiFi Milestones

- 1995-2000: first ideas, design, technical choices (functional language, combinator library,...)
- 2000: paper: “Composing Contracts: An Adventure in Financial Engineering” (with Simon Peyton Jones and Julian Seward)
- 2001: LexiFi founded, first implementation
- 2002: switch from 100% proprietary compiler to embedding a combinator algebra into a modified OCaml compiler: MLFi
- 2003: first Technology Client
- 2005: LexiFi Apropos, “end user” application (with automatic GUI generation technology) built on top of the technology
- 2006: LexiFi’s DSL powers the XpressInstruments module in SimCorp Dimension (first OEM client, collaboration continues today!)
- 2007: first client with “real” end-users (no programmers!)
- 2008: LexiFi Apropos embeds a powerful IDE (also integrated in SimCorp Dimension)
- 2009: “LexiFi Apropos for Private Banks”, a special edition automating most business processes of structured products desks (including client-facing documents)
- 2010: first European private bank to adopt LexiFi Apropos
- 2011: the Do-It-Yourself instrument gives most of the DSL flexibility in a GUI; end-users can implement ad hoc structures without programming
- 2012: private banks across Europe (Swiss, Luxembourg, Belgium, UK) adopt LexiFi Apropos
- 2012: switch from native pricing code generation to targeting a highly optimized Monte Carlo Virtual Machine
- 2013: a major French Asset Manager uses LexiFi to manage and price 99% of its entire book (allowing them to meet the new EMIR EU regulation)
- 2013: LexiFi’s DSL integrated into a major US information/service provider
- 2014: switch from native pricing code generation to a Partial Differential Equation Solver Virtual Machine
- 2014?: Algebra3: enhanced Contract Combinators
- 2014,2015?: cloud enabled SaaS offer

Algebra3 project

A more end-user friendly combinator set

- Consolidates more than 10 years of experience; streamlines both the user-facing API and the internal implementation techniques
- Feedback from our OEM customers and advanced end-users on how to make the DSL more accessible to beginner instrument developers
- Give an imperative flavor
- Drop the “point-free” approach for observables (= processes) in favor of a more standard style
- More flexible for date calculations within the DSL itself
- Enlarge the set of contracts that may be described

Lessons learned from Marketing and Sales

- It is very difficult to sell a language/framework/IDE/Workbench/. . . without being able to show an end-user product
- Only “early adopters” will do, and they are very rare
- Once such an end-user product is available, you may/will sell technology alone
- Some features are necessary for a demo, but never used in production
- 95% of prospects aren’t interested in DSLs or technical excellence, but only “features”
- Even if technical excellence will allow you to serve them better in the future, as you explain them
- 90% of clients will not use the product as you thought they would
- Client interaction/feedback/visits is essential

Regulator and Executable Specification: the SEC and the Python

The SEC put out in April 2010 a 667 page proposal regarding disclosures for asset backed securities:

> *"We are proposing to require that most ABS issuers file a computer program that gives effect to the flow of funds, or "waterfall," provisions of the transaction, [...] that the computer program be filed in the form of downloadable source code in Python." (page 205)*
> *"[...] the filed source code, when downloaded and run by an investor, must provide the user with the ability to programmatically input the user's own assumptions regarding the future performance and cash flows from the pool assets, including but not limited to assumptions about future interest rates, default rates, prepayment speeds, loss-given-default rates, and any other necessary assumptions" (page 210)*

- technology as a key regulatory tool
- many comments about language choice, availability, absence of formal semantics of Python and floating point arithmetic etc.

Recent perceived evolution

- As the AIG episode demonstrates, agents in financial institutions who are “in the know” have little incentive to implement systems that clarify exposures or imply a rapid transfer of knowledge;
- over the past decade, agents, shareholders, or regulators have not demonstrated a will to fundamentally improve the way financial contracts are described by the financial industry;
- a political momentum exists today to improve the market transparency, the market infrastructure, and the firm-level management of OTC derivatives.

Standardization

adoption of a common language for describing financial contracts

Conclusion

We argue that only a combination of

- a rigorous finance theory
- a strong numerical implementation
- a clever symbolic contract manipulation framework

can solve in a coherent and scalable way the huge problems encountered in the structured products space.