A Software Language Approach to Derivative Contracts in Finance

Dipl.-Inf. Jean-Marie Gaillourdet

Software Technology Group, Department of Computer Science, University of Kaiserslautern, Germany

***Abstract*— Financial derivatives are an important tool of today's finance sector. Despite their often negative public perception after the crisis of 2008, they are a tool originally developed to reduce risks in trading real goods. Understanding and evaluating derivatives is an important problem in practice. We present a declarative language of derivative contracts which is independent of pricing models. We also present a denotational semantics, enabling a calculus of contracts. The given denotational semantics enables also the application of abstract interpretation and static analysis techniques developed in the programming language community.**

I. Introduction

Financial derivatives are an important tool of today's finance sector. Despite their often negative public perception after the crisis of 2008, they are a tool — originally — developed to reduce risks in trading real goods such as wheat or copper. But at the same time financial derivatives introduce possibilities to speculate over the value of something without owning the actual something. Both uses lead eventually to their important role in banking and investment. Naturally, financial derivatives are a well studied topic in economics and mathematics, e.g. [1], [2]. Both mathematics and economics often use extensive software systems to analyze financial derivatives. Therefore, the apparent lack of intensive research in informatics seems surprising, notable exceptions are: [3], [4], [5], [6], [7].

But the interest to investigate financial contracts from a language perspective seems to be growing. In 2010 the U.S. Securities and Exchange Commission published a "concept release" with a request for comments, which included a proposal to require the publication of Python programs which represent the contract of Asset Backed Securities, a special kind of financial derivative contracts.

But what are financial derivatives? At first they are contracts, i.e. legal documents defining rights and obligations of two or more parties. Second, they are derived, i.e. the rights and obligations defined by the contract depend on some external variables. We call these external variables: *Observables*. They differ from values defined in a contract by their ability to change over time. E.g. possible observables are the temperature at Kaiserslautern or the course of some shares at some stock exchange. This notion of observables was already used in [3].

Henceforth, financial derivative contracts are contracts which are based upon observables describing rights and obligations in terms of payed money from one party to another.

We will further restrict this notion to contracts between two parties. One of them is the holder, he becomes the holder by acquiring the derivative contract. The other party remains anonymous as we will always analyze contracts from the view point of the holder.

The last restriction we place on the notion of derivatives, is that the contracts under consideration may grant the right to choose between alternatives only to the holder of the contract.

The contributions of this article are:

1) An overview of work done in informatics, especially from a software language point of view, on financial derivatives.
2) A formal definition of a language — based on [3] — which is capable of expressing financial derivatives themselves and not some derived notion.
3) A denotational semantics of this language, which enables to a notion of equality of derivative contracts, which strives to model the ideal of a contract and not its syntactic representation.

Section II covers the sparse related work in informatics. Section III defines our language of derivative contracts, which is an extension of the language of [3]. Section IV presents our denotational semantics of contracts and an equality notion of contracts. In Section V we will discuss shortly possible applications of and future work on our language and semantics.

II. Related Work

The earliest attempt at defining a software language for financial products, as far as we know, is a joint academic and industrial project resulting in the definition and implementation of RISLA. It is based on the insight, "that a product can be characterized by describing its cash flows." [4, p. 2]. Its purpose was to simplify the integration of new financial products into the existing business software systems of banks. Therefore, the project developed a compiler from RISLA to COBOL, which generated the necessary routines to integrate a financial product into the existing software infrastructure. The focus on integrating financial products into business software systems lead to a language design containing information about the expected user interface in the business software system. RISLA was not used to analyze the contract or perform pricing[1] on it. From the very sparse publicized documentation of it, it is not clear whether choices of the holder of a derivative are expressible or not.

[1]Pricing is a mathematical analysis which defines a notion of a *fair price*, see e.g. [2] for some recent results on pricing options.

In 2000, Peyton-Jones and Eber [3] presented a domain-specific language to declare derivatives in Haskell. Their language was implemented as combinator library in Haskell. They sketched a *valuation semantics* of derivatives and gave some hints on their implementation of a generic pricing mechanism. The valuation semantics of [3] is tied to the pricing or valuation approach of [1].

In [5] Mogensen introduces a language for cashflow reengineering, which was adopted by a non-disclosed "major Danish bank". The author presents a language, which is similar to data flow languages in programming, and a linear type system, which ensures that every amount of cash is spent only once.

Andersen, et al. present in [6] a language for compositional specification of contracts. That work is also based on [3]. Their focus is on specifying the exchange of resources between any number of involved parties. While our work represents every resource by its monetary value. Additionally, they give denotational and operational semantics for that language, which allows e.g. to decide whether a trace of steps in the real world conforms to a specified contract or not. In comparison to our language, this language is broader as it allows not only to specify derivative contracts, but general contracts, e.g. as they occur when you buy goods in store which provides the right to return the good under certain circumstances.

In [7] Reitz and Nögel present their work in the COMDECO project. That includes an XML based language for derivative contracts. They argue for the use of active documents instead of other approaches to develop applications, which perform pricing of derivatives.

III. A LANGUAGE OF DERIVATIVE CONTRACTS

A. Design Goals

The language for contracts we are going to present has been designed to enable standard mathematical analysis, like option valuation, on contracts written in that language. But at the same time, we did not want to include any assumptions which are introduced by a certain approach of analysis. The language should also allow to resolve any ambiguities about contracts, such might occur e.g. in a text document describing a contract, but must not be possible in this language.

Such a formally defined language should also be amenable to a mathematical treatment of contracts. Such a treatment should include a notion of equality of contracts, that corresponds to our intuition about contracts in real life. Nevertheless, it has to be a mathematical equality, i.e. enabling the substitution of equals and the formulation of mathematical laws on derivative contracts.

Last but not least, a language of contracts should be composable and support abstraction, i.e. support the naming of recurring patterns of contracts. The goal to support composition and abstraction is compatible with the design, we present later in this section, but we will not discuss it in this paper. Instead we will focus on the goals mentioned before.

B. Examples

Before we come to the language definition of derivative contracts, we will have a look at several examples of derivatives specified in it.

1) Zero Coupon Discount Bond: A *zero coupon discount bond* is a contract granting the holder the right to receive an amount of money k at a specified time t in the future. This example has been taken from [3]. This is not a *derivative* contract as it does not depend on external unknown variables.

```
When (At t) (Scale (Const k) One)
```

The meaning of this contract is the following. At time t, the holder acquires the contract Scale (Const k) One). Which is a contract granting the right to immediately receive k units of money. For a more detailed description of the keywords, see Section III-C.

2) European Put Option: A *european put option* is a derivative contract granting the holder the right but not the obligation to sell one item of e.g. a share of Siemens at a previously fixed price, here 100, at a predefined time, in this example 5.

```
When (At 5)
  (Or
    (Scale (Lift2 (−) (Unknown "Siemens") (Const 100)))
    Zero)
```

3) Barriers: The previous derivative contains the possibility to achieve very high pay-offs when the value of a Siemens share drops to almost zero. Therefore, it is common to include, barriers into derivatives. E.g. a derivative contract could state, when the value of one Siemens share gets below 50, the complete option is void. This is easily expressed with Until:

```
Until (Comp < (Unknown "Siemens") (Const 50))
  (When (At 5)
    (Or
      (Scale (Lift2 (−) (Unknown "Siemens") (Const 100)))
      Zero))
```

4) History dependent observables: So called asian variants of options use the mean of price of a share to determine the payoff. In order to construct an observable expression which represents the average of some observable, we have to add the values of observable at different times. In order to allow this, our language contains the constructor Reduce.

```
Lift2 /
  (Reduce + (Between (5,10)) (Unknown "Siemens"))
  (Const 5)
```

The observable above computes the average price of one Siemens share between time 5 and 10. Reduce collects all values of the third argument at times at which the second argument evaluates to true. These values are summed — the first argument. The initial value is the left neutral element of the given operation. The operation here is addition, therefore it is zero in this example.

These history dependent observables are not expressible in the language in [3].

$$
\begin{array}{rcl}
m & \in & \{<, >, >=, <=, ==, \neq\} \\
s & \in & \{+, -, *, \min, \max\} \\
u & \in & \{-, abs, \log, ...\} \\
b & \in & \{+, -, *, /\} \\
o & ::= & \text{Unknown n} \\
 & | & \text{Const k} \\
 & | & \text{Time t} \\
 & | & \text{Lift u o} \\
 & | & \text{D2C o} \\
 & | & \text{C2D o} \\
 & | & \text{Lift2 b } o_1 \text{ } o_2 \\
 & | & \text{Comp m } o_1 \text{ } o_2 \\
 & | & \text{At t} \\
 & | & \text{Between } (t_1, t_2) \\
 & | & \text{Reduce s } o_1 \text{ } o_2
\end{array}
$$

Fig. 1. Syntax of observables

C. Syntax Definition and Language Description

Our language for derivatives is based on the language in [3]. Their language is given as a set of Haskell functions with type signatures. [2] The language itself remains very similar with the exception of some simplifications and one addition, which we will discuss later in this section.

As usual $\mathcal{N}$ denotes the natural numbers including 0, $\mathcal{Z}$ denotes the positive and negative integers, $\mathcal{R}$ denotes the real numbers and $\mathcal{B}$ denotes the set of boolean values. We use $\mathcal{V} = \mathcal{Z} \cup \mathcal{R} \cup \mathcal{B}$ as the values of observables. Every time t is element of $\mathcal{N}$. And $\mathcal{S}$ denotes the set of names of unknown observables. These names are strings. We use the following convention for variable names: $t \in \mathcal{N}$, $k, v \in \mathcal{V}$, and $n \in \mathcal{S}$.

Fig. 1 contains the syntax definition of observable expressions in a style which is common in the programming language community. The single elements of the definitions are either mathematical statements $m \in \{...\}$ or are algebraic data type declaration. In the following, o denotes, depending on the context, the set of all observable expressions, or one concrete expression.

- Unknown n is an observable representing an external variable varying over time. In this work we will restrict the language to unknown observables with $\mathcal{R}$ as value domain.
- Const k is an observable of value $k \in \mathcal{V}$ for all times.
- Time is an observable with domain $\mathcal{N}$. Its value is t at every time t.
- Lift u o is the application of a unary operator u on o.
- Lift2 b o_1 o_2 is the application of a binary operator b on o_1 and o_2.
- Comp m o_1 o_2 is the comparison of o_1 and o_2 with the comparison operator m at every point in time.
- C2D o is the conversion of a continuous-valued observable o, i.e. an observable with domain $\mathcal{R}$, to a discrete-valued observable, i.e. an observable with domain $\mathcal{N}$.
- D2C o is the conversion of a discrete-valued observable o, i.e. an observable with domain $\mathcal{N}$, to a continuous-valued observable, i.e. an observable with domain $\mathcal{R}$.
- Reduce s o_1 o_2 is the fold with the binary function s over the sequence of values of o_2 at, when o_1 evaluates to true. The start value of the fold is the left neutral element of the function s.
- At t is syntactic sugar for: Comp (==) Time (Const t).
- Between (t_1,t_2) is syntactic sugar for:

  Lift2 (&&)
  (Comp (<=) (Const t_1) Time)
  (Comp (<) Time (Const t_2))

Fig. 2 defines the syntax of contracts. We'll give an informal description of the meaning of the constructs here:

- Zero is the contract which grants no rights and no obligations.
- One is the contract which grants the holder the right to receive one unit of money at the time of acquisition.
- Scale o c is a contract which is the same as c with every payment multiplied by the value of the observable o at the time of the payment.
- And c_1 c_2 is the contract which grants all rights and obligations of both c_1 and c_2 at the same time.
- Or c_1 c_2 is the contract which grants the holder the right to choose at time of acquisition to acquire either c_1 or c_2.
- Cond o c_1 c_2 is the contract which grants depending on the value of the boolean observable o at time of acquisition either all rights and obligations of c_1, if o is true, or c_2, if o is false.
- When o c is the contract which acquires the contract c at the first time the boolean observable o becomes true.
- Anytime o c is the contract which grants the holder at every time the boolean observable o is true the right to decide whether he wants to acquire c and release the complete contract Anytime o c or not.
- Until o c is the contract which grants the holder the right to choose between keeping his contract or acquiring c and releasing the complete contract. But this right is only granted as long as the boolean observable o is false. When o becomes true the whole contract becomes equivalent to Zero identical.

The language for derivative contracts is typed. But, because it has a completely standard type system, we omit a formal definition. We describe it only informally:

Every observable expression o has a one of three types: discrete, continuous, and boolean. Their value sets are $\mathcal{Z}$, $\mathcal{R}$, and $\mathcal{B}$. All unary operators take arguments of one type and

[2] Although, we don't present the language as an Haskell-embedded domain specific language our implementation is done as an Haskell-embedded domain specific language. We deliberately choose not to discuss such an embedding and the resulting benefits in this paper. Such benefits would include added expressiveness, modularization and abstraction capabilities, and a simplified implementation.

$$
\begin{array}{rcl}
c & ::= & \text{Zero} \\
& | & \text{One} \\
& | & \text{And } c_1\ c_2 \\
& | & \text{Or } c_1\ c_2 \\
& | & \text{Cond c } c_1\ c_2 \\
& | & \text{Scale o c} \\
& | & \text{When o c} \\
& | & \text{Anytime o c} \\
& | & \text{Until o c}
\end{array}
$$

Fig. 2. Syntax of contracts

return the same type. All binary and comparison operators take two arguments of the same type. Binary operators return the same type as their arguments and comparison operators always return boolean. C2D and D2C are the only ways to convert or cast observables. Contracts c have no type, the observable arguments of the contract constructors are of type continuous, if not given otherwise in the description of the contract constructors above.

IV. Semantics of Contracts

A. Introduction

In this section, we'll give a formal denotational semantics for our language of contracts, for an overview on denotational semantics see e.g. [8]. A formal semantics allows to study what it means to execute a contract, or in more natural terms: to fulfill a contract. By establishing formal semantics and removing all ambiguities — which are all too often present in natural language texts —, we gain insight into the contract itself. We can answer questions like: Is a sequence of payments between the holder and its contract counter party consistent with a certain contract? Is, for a given contract, every consistent sequence of payments between two contract parties of finite length? What is the maximal length of such a sequence? Is the holder able to receive a payment at all under a given contract? And so forth.

Semantics of formal language haven been studied for a long time. The largest fraction of the literature on this topic probably covers semantics of formally specified logics and programming languages. Applying these techniques to other kinds of languages is not new but certainly not widespread. In the context of contract languages, we are aware of only one previous article which gave formal denotational semantics for a contract language: [6].

We choose to give a denotational semantics, because a denotational semantics provides a straight forward notion of equality, and because it is quite naturally defined for our derivative contracts.

B. Denotational Semantics of Observables

A denotation of unknown observables is a function from time to values, $\mathcal{N} \to \mathcal{V}$. Therefore, an environment, i.e. a mapping of names to their denotation, is a function from $e : \mathcal{S} \to \mathcal{N} \to \mathcal{V}$. An environment represents the values of all unknown observables of a contract for all times. Therefore, the denotational semantics of observable expression is a function:

$$[\![\cdot]\!] : o \to (\mathcal{S} \to \mathcal{N} \to \mathcal{V}) \to (\mathcal{N} \to \mathcal{V})$$

We write $[\![o]\!]_e$ to denote the semantics of an observable expression o applied to an environment e. With an additional time argument t we write $[\![o]\!]^t_e$.

$[\![\cdot]\!]$ is defined as follows[3]:

$$
\begin{array}{rcl}
[\![\text{Unknown } n]\!]^t_e & = & e\ n\ t \\
[\![\text{Const } k]\!]^t_e & = & k \\
[\![\text{Time}]\!]^t_e & = & t \\
[\![\text{Lift u o}]\!]^t_e & = & u\ [\![o]\!]^t_e \\
[\![\text{D2C o}]\!]^t_e & = & [\![o]\!]^t_e \\
[\![\text{C2D o}]\!]^t_e & = & \lfloor [\![o]\!]^t_e \rfloor \\
[\![\text{Lift2 } b\ o_1\ o_2]\!]^t_e & = & [\![o]\!]^t_e\ b\ [\![o]\!]^t_e \\
[\![\text{Comp } m\ o_1\ o_2]\!]^t_e & = & [\![o]\!]^t_e\ m\ [\![o]\!]^t_e \\
\underbrace{[\![\text{Reduce } s\ o_1 o_2]\!]^t_e}_{d} & = & \begin{cases} init(s) & \text{if } t < 0 \\ [\![d]\!]^{t-1}_e & \text{if } [\![o_1]\!]^t_e = false \\ [\![d]\!]^{t-1}_e\ s\ [\![o_2]\!]^t_e & \text{if } [\![o_1]\!]^t_e = true \end{cases}
\end{array}
$$

$init(s)$ denotes the left neutral element of s.

C. Denotational Semantics of Contracts

In the following the notation $\widehat{x}$ will denote a set of x. We define the domain $\mathcal{D}$ of the denotation of contracts as the largest fixed point of the following recursive algebraic data type equation.

$$p \quad = \quad o \lhd \widehat{p}$$

$\lhd$ is the only constructor of this data type, it has two arguments, the first is an element of $\mathcal{V}$, the second is a finite set of application of this constructor. Therefore, this data type co-inductively defines the domain of all denotation of contracts. It is a forest of infinite height n-ary trees which are marked with elements of $\mathcal{V}$. The depths of a node of the forest is the time at which the mark of the node is payed to the holder of a contract. E.g. $\{17 \lhd 0 \lhd \dots\}$ is the denotation of a contract which grants the holder the right to receive a payment of 17 money units at time 0 and a payment of zero money units at time 1.

The branching of the trees denotes the choices the *holder* of the contract has.

The empty denotation which consists only of payments of 0 units of money is abbreviated as $\bullet$. It is largest fixpoint of the following equation:

$$\bullet \quad = \quad \{0 \lhd \bullet\}$$

We need the following auxiliary functions on $\mathcal{D}$, before we are able to define the denotational semantics.

$$
\begin{array}{rcl}
\widehat{p} \times \widehat{q} & = & \left\{ (p+q) \lhd (\widehat{p'} \times \widehat{q'}) \middle| (p \lhd \widehat{p'}) \in \widehat{p},\ (q \lhd \widehat{q'}) \in \widehat{q} \right\} \\
o *^t_e \widehat{p} & = & \left\{ ([\![o]\!]^t_e * v) \lhd (o *^{t+1}_e \widehat{p'}) \middle| (v' \lhd \widehat{p'}) \in \widehat{p} \right\}
\end{array}
$$

$\times$ adds two elements of $\mathcal{D}$ such that both arguments are combined into one, as if they were both executed.

[3]We use a curried function application notation which does not use parenthesis

$*_e^t$ multiplies every payment of an element of $\mathcal{D}$ with the current value of an observable expression.

Now, we are able to define the denotation of contracts, which is the following function:

$$[\![c]\!] : (\mathcal{S} \to \mathcal{N} \to \mathcal{V}) \to \mathcal{N} \to \mathcal{D}$$

$$\begin{aligned}
[\![\text{Zero}]\!]_e^t &= \bullet \\
[\![\text{One}]\!]_e^t &= \{1 \multimap \bullet\} \\
[\![\text{And a b}]\!]_e^t &= [\![a]\!]_e^t \times [\![b]\!]_e^t \\
[\![\text{Or a b}]\!]_e^t &= [\![a]\!]_e^t \cup [\![b]\!]_e^t \\
[\![\text{Scale o c}]\!]_e^t &= o *_e^t [\![c]\!]_e^t \\
[\![\text{Cond c a b}]\!]_e^t &= \begin{cases} [\![b]\!]_e^t & \text{if } [\![c]\!]_e^t = false \\ [\![a]\!]_e^t & \text{if } [\![c]\!]_e^t = true \end{cases} \\
[\![\underbrace{\text{When b c}}_{d}]\!]_e^t &= \begin{cases} \{0 \multimap [\![d]\!]_e^{t+1}\} & \text{if } [\![b]\!]_e^t = false \\ [\![c]\!]_e^t & \text{if } [\![b]\!]_e^t = true \end{cases} \\
[\![\underbrace{\text{Anytime b c}}_{d}]\!]_e^t &= \begin{cases} \{0 \multimap [\![d]\!]_e^{t+1}\} & \text{if } [\![b]\!]_e^t = false \\ [\![c]\!]_e^t \cup \{0 \multimap [\![d]\!]_e^{t+1}\} & \text{if } [\![b]\!]_e^t = true \end{cases} \\
[\![\underbrace{\text{Until b c}}_{d}]\!]_e^t &= \begin{cases} [\![c]\!]_e^t \cup \{0 \multimap [\![d]\!]_e^{t+1}\} & \text{if } [\![b]\!]_e^t = false \\ \bullet & \text{if } [\![b]\!]_e^t = true \end{cases}
\end{aligned}$$

When we return to the example of Sec. III-B.2, we can now determine the following: Let x be the price of one Siemens share at time 5 as determined by e, i.e. $[\![\text{Unknown "Siemens"}]\!]_e^5 = x$, and let c be the referenced example contract, then

$$[\![c]\!]_e^t = \{0 \multimap \{0 \multimap \{0 \multimap \{0 \multimap \{0 \multimap (\bullet \cup \{(x - 100) \multimap \bullet\})\}\}\}\}\}$$

The denotation tells us that this European put option is a contract, which can be described as a sequence of 5 payments of zero amounts of money, followed either by an infinite sequence of zero-valued payments or by a payment of value $x - 100$ followed by an infinite sequence of zero-valued payments.

D. Derived Notions

We define the equality of two contracts a and b, now as follows:

$$a = b \text{ if and only if } \forall e. [\![a]\!]_e^0 = [\![b]\!]_e^0$$

Now, we have a notion of equality which is not tied to the syntax of contracts or observables, instead we claim it is tied to the intended meaning, we associate with a contract. Yet, we have a mathematical definition of this equality notion.

Most contracts in real life have a maximal life time. The same is true for derivative contracts in finance. We are going to define a notion of a horizon, which will be the earliest time after which nothing interesting may happen.

The smallest time t for which the following equation holds is called the horizon of a contract c.

$$\forall t'.t' > t \implies \forall e. \exists \widehat{p}_{i \in \{1 \ldots n\}}. [\![c]\!]_e^{t'} = \{0 \multimap \widehat{p_1}, \ldots, 0 \multimap \widehat{p_n}\}$$

Note: Such a horizon t doesn't have to exist for every contract c, while most practical contracts do have a defined horizon. The following contract does not have a defined horizon, because there is no time at which we can guaranty, that the price of a Siemens share has been at least once smaller than the price of a Daimler share.

When
(Comp $<$ (Unknown "Siemens") (Unknown "Daimler"))
One

V. Applications, Future Work and Conclusions

As we said in Sec. III-A, we wanted to design a language which can be used to do option pricing, but which is not tied to one approach. So far, we have implemented option pricing for a subset with help of Qian Liang and Stefanie Müller based on [3],[2], and [1]. We plan to add Monte-Carlo based option pricing to our implementation.

Our language and denotational semantics could also be used to apply abstract interpretation or some specialized variants of it to derivative contracts. That could allow to compute worst or best case scenarios. Or it could allow to determine whether there is horizon, at which times decision have to be made by the holder, and much more.

The semantics itself allows to develop a calculus of contracts, which allows to transform contracts in order to simplify other analysises for example.

We have designed a language to express financial derivatives, which is more general than the language of [3] and more specific than the language of [6]. We believe this is a useful middle ground to work with derivatives and not with more general contracts. We have given a formal semantics which captures the intuitive meaning in a formal definition, which allows to apply the substitution principle. We don't have to resort to bisimulation of processes or similar approaches in order to show the equivalence of two contracts.

References

[1] J. C. Cox, S. A. Ross, and M. Rubinstein, "Option pricing: A simplified approach," *Journal of Financial Economics*, vol. 7, no. 3, pp. 229–263, 1979.

[2] S. Müller, "The binomial approach to option valuation." [Online]. Available: http://kluedo.ub.uni-kl.de/volltexte/2010/2462/

[3] S. Peyton Jones, J.-M. Eber, and J. Seward, "Composing contracts: an adventure in financial engineering (functional pearl)," in *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2000, pp. 280–292.

[4] B. Arnold, A. V. Deursen, and M. Res, "An algebraic specification of a language for describing financial products," in *ICSE-17 Workshop on Formal Methods Application in Software Engineering*. IEEE, 1995, pp. 6–13.

[5] T.Æ. Mogensen, "Linear types for cashflow reengineering," pp. 823–845, 2003.

[6] J. Andersen, E. Elsborg, F. Henglein, J. G. Simonsen, and C. Stefansen, "Compositional specification of commercial contracts," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 8, no. 6, pp. 485–516, November 2006.

[7] M. Reitz and U. Nögel, "Components: A valuable investment for financial engineering," in *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*. New York, NY, USA: ACM, 2006, pp. 153–162.

[8] D. A. Schmidt, *Denotational Semantics*. Allyn And Bacon, Inc., 1986.