# Formalization of Automated Trading Systems in a Concurrent Linear Framework*

Iliano Cervesato Sharjeel Khan Giselle Reis Dragiša Žunić
Carnegie Mellon University
iliano@cmu.edu smkhan@andrew.cmu.edu giselle@cmu.edu dzunic@andrew.cmu.edu

We present a declarative and modular specification of an automated trading system (ATS) in the concurrent linear framework CLF. We implemented it in Celf, a CLF type checker which also supports executing CLF specifications. We outline the verification of two representative properties of trading systems using generative grammars, an approach to reasoning about CLF specifications.

## 1 Introduction

Trading systems are platforms where buy and sell orders are automatically matched. Matchings are executed according to the operational specification of the system. In order to guarantee trading fairness, these systems must meet the requirements of regulatory bodies, in addition to any internal requirement of the trading institution. However, both specifications and requirements are presented in natural language which leaves space for ambiguity and interpretation errors. As a result, it is difficult to guarantee regulatory compliance [3]. For example, the main US regulator, the Securities and Exchanges Commission (SEC), has fined several companies, including Deutsche Bank (37M in 2016), Barclay's Capital (70M in 2016), Credit Suisse (84M in 2016), UBS (19.5M in 2015) and many others [6] for non-compliance.

Modern trading systems are complex pieces of software with intricate and sensitive rules of operation. Moreover they are in a state of continuous change as they strive to support new client requirements and new order types. Therefore it is difficult to attest that they satisfy all requirements at all times using standard software testing approaches. Even as regulatory bodies recently demand that systems must be "fully tested" [1], experience has shown that (possibly unintentional) violations often originate from unforeseen interactions between order types [10].

Formalization and formal reasoning can play a big role in mitigating these problems. They provide methods to verify properties of complex and infinite state space systems with certainty, and have already been applied in fields ranging from microprocessor design [8], avionics [14], election security [11], and financial derivative contracts [12, 2]. Trading systems are a prime candidate as well.

In this paper we use the logical framework CLF [5] to specify and reason about trading systems. CLF is a linear concurrent extension of the long-established LF framework [7]. Linearity enables natural encoding of state transition, where facts are consumed and produced thereby changing the system's state. The concurrent nature of CLF is convenient to account for the possible orderings of exchanges.

The contributions of this research are twofold: (1) We formally define an archetypal automated trading system in CLF [5] and implement it as an executable specification in Celf. (2) We demonstrate how to prove some properties about the specification using generative grammars [13], a technique for meta-reasoning in CLF.

*This paper was made possible by grant NPRP 7-988-1-178 from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

T. Ehrhard, M. Fernández, V. de Paiva, L. Tortora de Falco (Eds.):
Linearity-TLLA 2018
EPTCS 292, 2019, pp. 1–14, doi:10.4204/EPTCS.292.1

$$\frac{\Gamma;\Delta;\Psi \vdash P_0}{\Gamma;\Delta;\Psi,1 \vdash P_0}\ 1_l \quad \frac{\Gamma;\Delta;\Psi,P,Q \vdash P_0}{\Gamma;\Delta;\Psi,P\otimes Q \vdash P_0}\ \otimes_l \quad \frac{\Gamma,a;\Delta;\Psi \vdash P_0}{\Gamma;\Delta;\Psi,!a \vdash P_0}\ !_l \quad \frac{\Gamma;\Delta,a;\Psi \vdash P_0}{\Gamma;\Delta;\Psi,a \vdash P_0}\ \mathsf{st} \quad \frac{\Gamma;\Delta \vdash P_0}{\Gamma;\Delta;\cdot \vdash P_0}\ \mathsf{L}$$

$$\frac{}{\Gamma;\cdot \vdash 1}\ 1_r \quad \frac{\Gamma;\Delta_1 \vdash P \quad \Gamma;\Delta_2 \vdash Q}{\Gamma;\Delta_1,\Delta_2 \vdash P\otimes Q}\ \otimes_r \quad \frac{\Gamma;\cdot \vdash a}{\Gamma;\cdot \vdash !a}\ !_r \quad \frac{\Gamma;\Delta \vdash a}{\Gamma;\Delta \vdash a}\ \mathsf{R} \quad \frac{}{\Gamma;a \vdash a}\ \mathsf{init}$$

$$\frac{\Gamma;\Delta_1 \vdash a \quad \Gamma;\Delta_2,N \vdash F}{\Gamma;\Delta_1,\Delta_2,a\multimap N \vdash F}\ \multimap_l \quad \frac{\Gamma;\cdot \vdash a \quad \Gamma;\Delta,N \vdash F}{\Gamma;\Delta,a\to N \vdash F}\ \to_l \quad \frac{\Gamma;\Delta,N[x\mapsto t] \vdash F}{\Gamma;\Delta,\forall x.N \vdash F}\ \forall_l \quad \frac{\Gamma;\Delta;P \vdash P_0}{\Gamma;\Delta,\{P\} \vdash P_0}\ \{\}_l$$

$$\frac{\Gamma;\Delta,a \vdash N}{\Gamma;\Delta \vdash a\multimap N}\ \multimap_r \quad \frac{\Gamma,a;\Delta \vdash N}{\Gamma;\Delta \vdash a\to N}\ \to_r \quad \frac{\Gamma;\Delta \vdash N[x\mapsto \alpha]}{\Gamma;\Delta \vdash \forall x.N}\ \forall_r \quad \frac{\Gamma;\Delta \vdash P}{\Gamma;\Delta \vdash \{P\}}\ \{\}_r \quad \frac{\Gamma,N;\Delta,N \vdash C}{\Gamma,N;\Delta \vdash C}\ \mathsf{cont}$$

Figure 1: Sequent calculus for a fragment of CLF. $N$ is a negative formula, $P$ and $Q$ are positive formulas, $P_0$ is either an atom or $\{P\}$, $F$ is any formula, $a$ is an atom, $\alpha$ is an eigenvariable and $t$ is a term.

The paper is organized as follows: Section 2 introduces the concurrent logical framework CLF. Section 3 introduces the core concepts related to automated trading systems (ATS), followed by Section 4, which presents the formalization of an ATS in CLF/Celf. Section 5 contains proofs of two properties based on generative grammars, going towards automated reasoning in CLF. We conclude and outline possible further developments in Section 6.

## 2 Concurrent Linear Logic and Celf

The logical framework CLF [5] is based on a fragment of intuitionistic linear logic. It extends the logical framework LF [7] with the linear connectives $\multimap$, $\&$, $\top$, $\otimes$, 1 and ! to obtain a resource-aware framework with a satisfactory representation of concurrency. The rules of the system impose a discipline on when the synchronous connectives $\otimes$, 1 and ! are decomposed, thus still retaining enough determinism to allow for its use as a logical framework. Being a type-theoretical framework, CLF unifies implication and universal quantification as the dependent product construct. For simplicity we present only the logical fragment of CLF (i.e., without terms) needed for our encodings. A detailed description of the full framework can be found in [5].

We divide the formulas in this fragment of CLF into two classes: *negative* and *positive*. Negative formulas have right invertible rules and positive formulas have left invertible rules. Their grammar is:

$$\begin{array}{rcll} N,M & ::= & a\multimap N \mid a\to N \mid \{P\} \mid \forall x.N \mid a & \text{(negative formulas)} \\ P,Q & ::= & P\otimes Q \mid 1 \mid !a \mid a & \text{(positive formulas)} \end{array}$$

where $a$ is an atom (i.e., a predicate). Positive formulas are enclosed in the lax modality $\{\cdot\}$, which ensures that their decomposition happens atomically.

The sequent calculus proof system for this fragment of CLF is presented in Figure 1. The sequents make use of either two or three contexts on the left: $\Gamma$ contains unrestricted formulas, $\Delta$ contains linear formulas and $\Psi$, when present, contains positive formulas. The decomposition phase of a positive formula is indicated in red on the right and in blue on the left. These phases end (by means of rules L or R) after the formula is completely decomposed.

Since CLF has both the linear and intuitionistic implications, we can specify computation in two different ways. Simplifying somewhat, linear implication formulas are interpreted as multiset rewriting: the bounded resources on the left are consumed and those on the right are produced. State transitions can

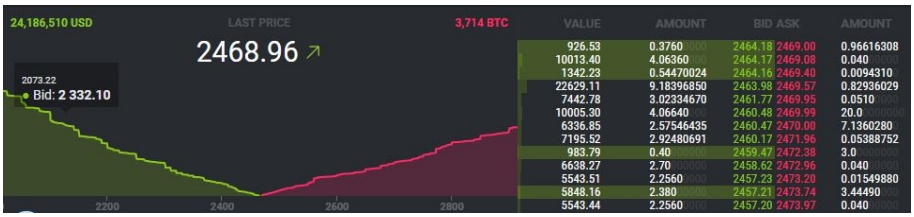



Figure 2: Visualization of the market view

be modeled naturally this way. Intuitionistic implication formulas are interpreted as backward-chaining rules *à la* Prolog, providing a way to compute solutions for a predicate by matching it with the head (rightmost predicate) of a rule and solving the body. In this paper, predicates defined by backward-chaining rules are written in green.

The majority of our encoding involves rules in the following shape (for atomic $p_i$ and $q_i$):
$p_1 \otimes \ldots \otimes p_n \multimap \{q_1 \otimes \ldots \otimes q_m\}$ which is the uncurried version of: $p_1 \multimap \ldots \multimap p_n \multimap \{q_1 \otimes \ldots \otimes q_m\}$.

This framework is implemented as the tool Celf (`https://clf.github.io/celf/`) which we used for the encodings. Following the tool's convention, variable names start with an upper-case letter.

## 3 Automated Trading System (ATS)

Real life trading systems differ in the details of how they manage orders (there are hundreds of order types in use [9]). However, there is a certain common core that guides all those trading systems, and which embodies the market logic of trading on an exchange. We have formalized those elements in what we call an automated trading system, or an ATS. In what follows we introduce the basic notions.

An *order* is an investor's instruction to a broker to buy or sell securities (or any asset type which can be traded). They enter an ATS sequentially and are *exchanged* when successfully matched against opposite order(s). In this paper, we will be concerned with *limit*, *market* and *cancel* orders.

A *limit order* has a specified limit price, meaning that it will trade at that price or better. In the case of a limit order to sell, a limit price $P$ means that the security will be sold at the best available price in the market, but no less than $P$. And dually for buy orders. If no exchange is possible, the order stays in the market waiting to be exchanged – these are called *resident orders*. A *market order* does not specify the price, and will be immediately matched against opposite orders in the market. If none are available, the order is discarded. A *cancel order* is an instruction to remove a resident order from the market.

A *matching algorithm* determines how resident orders are prioritized for exchange, essentially defining the mode of operation of a given ATS. The most common one is *price/time priority*. Resident orders are first ranked according to their price (increasingly for sell and decreasingly for buy orders); orders with the same price are then ranked depending on when they entered.

Figure 2 presents a visualization of a (Bitcoin) market. The left-hand side (green) contains resident buy orders, while the right-hand side (pink) contains resident sell orders. The price offered by the most expensive buy order is called *bid* and the cheapest sell order is called *ask*. The point where they (almost) meet is the *bid-ask* spread, which, at that particular moment, was around 2,468 USD.

Standard regulatory requirements for real world trading systems include: the bid price is always strictly less than the ask price (i.e., no locked – *bid* is equal to *ask* – or crossed – *bid* is greater than *ask* – states), the trade always takes place at either *bid* or *ask*, the price/time priority is always respected when exchanging orders, the order priority function is transitive, among others.

# 4 Formalization of an ATS

We have formalized the most common components of an ATS in the logical framework CLF and implemented them in Celf. The formalization of an ATS is divided into three parts. First, we represent the market infrastructure using some auxiliary data structures. Then we determine how to represent the basic order types and how they are organized for processing. Finally we encode the exchange rules which act on incoming orders.

Since we are using a linear framework, the state of the system is naturally represented by a set of facts which hold at that point in time. Each rule consumes some of these facts and generates others, thus reaching a new state. Many operations are dual for buy and sell orders, so, whenever possible, predicates and rules are parameterized by the action (`sell` or `buy`, generically denoted $A$). The machinery needed in our formalization includes natural numbers, lists and queues. Their encoding relies on the backward-chaining semantics of Celf.

The full encoding can be found at `https://github.com/Sharjeel-Khan/financialCLF`.

## 4.1 Infrastructure

The trading system's infrastructure is represented by the following four linear predicates:

$$\texttt{queue}(Q) \qquad \texttt{priceQ}(A,\ P,\ Q) \qquad \texttt{actPrices}(A,\ L) \qquad \texttt{time}(T)$$

Predicate $\texttt{queue}(Q)$ represents the queue in which orders are inserted for processing. As orders arrive in the market, they are assigned a timestamp and added to $Q$. For an action $A$ and price $P$, the queue $Q$ in $\texttt{priceQ}(A,\ P,\ Q)$ contains all resident orders with those attributes. Due to how orders are processed, the queue is sorted in ascending order of timestamp. We maintain the invariant that price queues are never empty. Price queues correspond to columns in the graph of Figure 2. For an action $A$, the list $L$ in $\texttt{actPrices}(A,\ L)$ contains the exchange prices available in the market, i.e., all the prices on the $x$-axis of Figure 2 with non-empty columns. Note that the bid price is the maximum of $L$ when $A$ is `buy` and the ask price is the minimum when $A$ is `sell`. The time is represented by the fact $\texttt{time}(T)$ and increases as the state changes.

The `begin` fact is the entry point in our formalization. This fact starts the ATS. It is rewritten to an empty order queue, empty active price lists for `buy` and `sell`, and the zero time:

$$\texttt{begin} \multimap \{\texttt{queue}(\texttt{empty}) \otimes \texttt{actPrices}(\texttt{buy},\ \texttt{nil}) \otimes \texttt{actPrices}(\texttt{sell},\ \texttt{nil}) \otimes \texttt{time}(\texttt{z})\}$$

## 4.2 Orders' Structure

An order is represented by a linear fact $\texttt{order}(O,\ A,\ P,\ ID,\ N)$, where $O$ is the type of order, $A$ is an action, $P$ is the order price, $ID$ is the identifier of the order and $N$ is the quantity. $P$, $ID$ and $N$ are natural numbers. In this paper, $O$ is one of `limit`, `market`, or `cancel`. An order predicate in the context is consumed and added to the order queue for processing via the following rule:

$$\begin{aligned} & \texttt{order}(O,\ A,\ P,\ ID,\ N) \otimes \texttt{queue}(Q) \otimes \texttt{time}(T) \otimes \texttt{enq}(Q,\ \texttt{ordIn}(O,\ A,\ P,\ ID,\ N,\ T),\ Q') \\ \multimap\ & \{\texttt{queue}(Q') \otimes \texttt{time}(\texttt{s}(T))\} \end{aligned}$$

The predicate is transformed into a term $\texttt{ordIn}(O, A, P, ID, N, T)$ containing the same arguments plus the timestamp $T$. This term is added to the order queue $Q$ by the (backward-chaining) predicate enq. This queue allows the sequential processing of orders given their time of arrival in the market, thus simulating what happens in reality. The timestamp is also used to define resident order priority. Sequentiality is guaranteed as all state transition rules act only on the first order in the queue. Nevertheless, due to Celf's non-determinism, orders are added to the queue in an arbitrary order.

### 4.3 Limit Orders

According to the matching logic, there are two basic actions for every limit order in the queue: exchange (partially or completely) or add to the market (becomes resident). The action taken depends on the order's limit price (at which it is willing to trade), the bid and ask prices, as well as the quantity of resident orders[1].

**Adding orders to the market** An order is added to the market when its limit price $P$ is such that it cannot be exchanged against opposite resident orders. Namely when $P < ask$ in the case of a buy order, and when $P > bid$ in the case of a sell order. There are two rules for adding an order, depending on whether there are resident orders at that price in the market or not (see Figure 3). The (backward chaining) predicate store is provable when the order cannot be exchanged.

$$\begin{aligned}
\texttt{limit/empty:}\quad & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{store}(A, L', P) \otimes \texttt{actPrices}(A, L) \otimes \texttt{notInList}(L, P) \otimes \texttt{insert}(L, P, LP) \otimes \texttt{time}(T) \\
\multirow{2}{*}{$\multimap$} \quad & \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', L') \otimes \texttt{priceQ}(A, P, \texttt{consP}(ID, N, T, \texttt{nilP})) \\
& \otimes \texttt{actPrices}(A, LP) \otimes \texttt{time}(\texttt{s}(T))\}
\end{aligned}$$

$$\begin{aligned}
\texttt{limit/queue:}\quad & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{store}(A, L', P) \otimes \texttt{priceQ}(A, P, PQ) \otimes \texttt{extendP}(PQ, ID, N, T, PQ') \otimes \texttt{time}(T) \\
\multirow{2}{*}{$\multimap$} \quad & \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', L') \otimes \texttt{priceQ}(A, P, PQ') \otimes \texttt{time}(\texttt{s}(T))\}
\end{aligned}$$

Figure 3: Adding limit orders to the market

The first line is the same for both. Given the order at the front of the order queue, the predicate dual will bind $A'$ to the dual action of $A$ (i.e., if $A$ is buy, $A'$ will be sell and vice-versa). Then $\texttt{actPrices}(A', L')$ binds $L'$ to the list of active prices of $A'$. The incoming order can be added to the market only if there is no dual resident order at an acceptable price. For example, if $A$ is buy at price $P$, any resident sell order with price $P$ *or less* would be an acceptable match. The predicate store holds iff there is no acceptable match.

The second line of each rule distinguishes whether the new order to be added is the first one at that price (limit/empty rule – $\texttt{notInList}(L, P)$) or not. If it is, the active price list is updated by backward chaining on $\texttt{insert}(L, P, LP)$, and rewriting $\texttt{actPrices}(A, L)$ to $\texttt{actPrices}(A, LP)$. Additionally, a new price queue is created with that order alone: $\texttt{priceQ}(A, P, \texttt{consP}(ID, N, T, \texttt{nilP}))$. If there are resident orders at the same price (and action), the existing price queue is extended with the new order by backward chaining on $\texttt{extendP}(PQ, ID, N, T, PQ')$ and by rewriting $\texttt{priceQ}(A, P, PQ)$ to $\texttt{priceQ}(A, P, PQ')$. Both rules increment the time by one unit.

[1] Sometimes an incoming limit order will be partially filled, with the remainder (once resident orders that match this limit price are filled) becoming a new resident order.

**Exchanging orders** The rules for exchanging orders are presented in Figure 4. A limit order is exchanged when its limit price $P$ satisfies $P \leq bid$, in the case of sell orders, or $P \geq ask$ for buy orders.

$$\begin{array}{ll}
\texttt{limit/1:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N', T', \texttt{nilP})) \otimes \texttt{remove}(L', X, L'') \otimes \\
& \texttt{nat-equal}(N, N') \otimes \texttt{time}(T) \\
\multicolumn{2}{r}{\multimap \quad \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', L'') \otimes \texttt{time}(\texttt{s}(T))\}}
\end{array}$$

$$\begin{array}{ll}
\texttt{limit/2:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N', T', \texttt{consP}(ID1, N1, T1, L))) \otimes \\
& \texttt{nat-equal}(N, N') \otimes \texttt{time}(T) \\
\multicolumn{2}{r}{\multimap \quad \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', L') \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID1, N1, T1, L)) \otimes \texttt{time}(\texttt{s}(T))\}}
\end{array}$$

$$\begin{array}{ll}
\texttt{limit/3:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N', T', \texttt{nilP})) \otimes \texttt{remove}(L', X, L'') \otimes \\
& \texttt{nat-great}(N, N') \otimes \texttt{nat-minus}(N, N', N'') \\
\multicolumn{2}{r}{\multimap \quad \{\texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N'', T), Q)) \otimes \texttt{actPrices}(A', L'')\}}
\end{array}$$

$$\begin{array}{ll}
\texttt{limit/4:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N', T', \texttt{consP}(ID1, N1, T1, L))) \otimes \\
& \texttt{nat-great}(N, N') \otimes \texttt{nat-minus}(N, N', N'') \\
\multicolumn{2}{r}{\multimap \quad \{\texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N'', T), Q)) \otimes \texttt{actPrices}(A', L') \otimes} \\
\multicolumn{2}{r}{\texttt{priceQ}(A', X, \texttt{consP}(ID1, N1, T1, L))\}}
\end{array}$$

$$\begin{array}{ll}
\texttt{limit/5:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \\
& \texttt{exchange}(A, L', P, X) \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N', T', L)) \otimes \\
& \texttt{nat-less}(N, N') \otimes \texttt{nat-minus}(N', N, N'') \otimes \texttt{time}(T) \\
\multicolumn{2}{r}{\multimap \quad \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', L') \otimes \texttt{priceQ}(A', X, \texttt{consP}(ID', N'', T', L)) \otimes \texttt{time}(\texttt{s}(T))\}}
\end{array}$$

Figure 4: Exchanging limit orders

The (backward chaining) predicate exchange binds $X$ to the exchange price (either *bid* or *ask*). We distinguish between an incoming order that "consumes" all the quantity available at price $X$, or only a part of the combined quantity available. The arithmetic comparison and operations are implemented in the usual backward-chaining way using a unary representation of natural numbers.

All five rules start the same way: the first line binds $L'$ to the list of active prices of the dual orders. The backward-chaining predicate $\texttt{exchange}(A, L', P, X)$ holds iff there is a matching resident order. In this case, it binds $X$ to the best available market price. The first order in the price queue for $X$ has priority and will be exchanged. Let $N$ be the quantity in the incoming order and $N'$ be the quantity of the resident order with highest priority. There are three cases:

- $N = N'$ (rules limit/1 and limit/2): Both orders will be completely exchanged. The incoming order is removed from the order queue by continuing with $\texttt{queue}(Q)$. The resident order is removed from its price queue and we distinguish two cases:
  - It is the last element in the queue (limit/1): then the fact $\texttt{priceQ}(\_, \_, \_)$ is not rewritten and $X$ is removed from the list of active prices by the backward chaining predicate $\texttt{remove}(L', X, L'')$. This list is rewritten from $\texttt{actPrices}(A', L')$ to $\texttt{actPrices}(A', L'')$.
  - Otherwise (limit/2), the resident order is removed from the price queue by rewriting $\texttt{priceQ}(A', X, \texttt{consP}(ID', N, T', \texttt{consP}(ID1, N1, T1, L)))$ to $\texttt{priceQ}(A', X, \texttt{consP}(ID1, N1, T1, L))$.

- $N > N'$ (rules `limit/3` and `limit/4`): The incoming order will be partially exchanged, and not leave the order queue as long as there are matching resident orders. At each exchange its quantity is updated to $N''$, the difference between $N$ and $N'$, computed by the backward-chaining predicate `nat-minus`$(N, N', N'')$. The order queue is rewritten to `queue`$(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N'', T), Q))$. The resident order will be completely consumed and removed from the market. Therefore, we need to distinguish two cases as before (last element in its price queue – `limit/3` – or not – `limit/4`).
- $N < N'$ (rule `limit/5`): The incoming order is completely exchanged and removed from the order queue (rewritten to `queue`$(Q)$). The resident order is partially exchanged and its quantity is updated to $N''$, computed by the backward chaining rule `nat-minus`$(N', N, N'')$. Notice that the price queue is rewritten with the order in the same position, so its priority does not change.

By convention, the time is only updated once an order is completely processed and removed from the order queue.

### 4.4 Market Orders

A market order is meant to be exchanged immediately at current market prices. As long as there are available sellers or buyers, market orders are exchanged. The remaining part of a market order is discarded. Certainty of execution is a priority over the price of execution.

In this case, there are no rules for adding/storing of market orders. The exchange rules are similar to the ones for limit orders, with subtle differences. Market orders do not have a desired price as an attribute, they only have a desired quantity. Therefore exchanging is continued as long as the quantity was not reached and there are available resident orders (the price $P$ is nominally presented but it is never used). In the rules, the predicate `exchange`$(A, L, P, X)$ is replaced with `mktExchange`$(A, L, X)$, which simply binds $X$ to the best offer in the market. In the unlikely event that there are no more dual resident orders (verified by checking if $L$ in `actPrices`$(A', L)$ is empty), the order is removed from the order queue.

The rules for exchanging market orders are given in Figure 5. The first rule, `market/empty`, addresses the situation when a market order is in the order queue, but there are no opposite resident orders to be matched against. The order is then removed from the order queue. Rules `market/1` and `market/2` address the case when the incoming market order's quantity is the same as the quantity of the best available opposite resident order. In this case these orders are simply exchanged, and again we distinguish cases whether the resident order was the last in the queue (`market/1`) or not (`market/2`). Rules `market/3` and `market/4` address the case when the incoming market order's quantity is greater than the quantity of the best available opposite resident order, i.e., when $N > N'$. In this case the resident order is exchanged and the rest of the market order remains in the order queue. The two rules distinguish whether the resident order was the last in the price queue (`market/3`) or not (`market/4`). Finally, rule `market/5` describes the situation when an incoming market order is strictly less (quantity-wise) compared to the best opposite resident order. The considered market order is exchanged completely whereas the resident order only partially.

### 4.5 Cancel Orders

Cancel order is an instruction to remove a particular resident order from the trading system. Cancel orders refer to a resident order by its identifier. If, by chance, the order to be canceled is not in the

$$\texttt{market/empty:}\quad \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{market}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', \texttt{nilN}) \otimes \texttt{time}(T) \multimap \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', \texttt{nilN}) \otimes \texttt{time}(\texttt{s}(T))\}$$

$$\texttt{market/1:}\quad \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{market}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \texttt{mktExchange}(A, L', Y) \otimes \texttt{priceQ}(A', Y, \texttt{consP}(ID', N', T', \texttt{nilP})) \otimes \texttt{remove}(L', Y, L'') \otimes \texttt{nat-equal}(N, N') \otimes \texttt{time}(T) \multimap \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A', L'') \otimes \texttt{time}(\texttt{s}(T))\}$$

$$\texttt{market/2:}\quad \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{market}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \texttt{mktExchange}(A, L', Y) \otimes \texttt{priceQ}(A', Y, \texttt{consP}(ID', N', T', (\texttt{consP}(ID1, N1, T1, L)))) \otimes \texttt{nat-equal}(N, N') \otimes \texttt{time}(T) \multimap \{\texttt{queue}(Q) \otimes \texttt{priceQ}(A', Y, (\texttt{consP}(ID1, N1, T1, L))) \otimes \texttt{actPrices}(A', L') \otimes \texttt{time}(\texttt{s}(T))\}$$

$$\texttt{market/3:}\quad \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{market}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \texttt{mktExchange}(A, L', Y) \otimes \texttt{priceQ}(A', Y, \texttt{consP}(ID', N', T', \texttt{nilP})) \otimes \texttt{remove}(L', Y, L'') \otimes \texttt{nat-great}(N, N') \otimes \texttt{nat-minus}(N, N', N'') \multimap \{\texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N'', T), Q)) \otimes \texttt{actPrices}(A', L'')\}$$

$$\texttt{market/4:}\quad \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{market}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \texttt{mktExchange}(A, L', Y) \otimes \texttt{priceQ}(A', Y, \texttt{consP}(ID', N', T', (\texttt{consP}(ID1, N1, T1, L)))) \otimes \texttt{nat-great}(N, N') \otimes \texttt{nat-minus}(N, N', N'') \multimap \{\texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{limit}, A, P, ID, N'', T), Q)) \otimes \texttt{priceQ}(A', Y, (\texttt{consP}(ID1, N1, T1, L))) \otimes \texttt{actPrices}(A', L')\}$$

$$\texttt{market/5:}\quad \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{market}, A, P, ID, N, T), Q)) \otimes \texttt{dual}(A, A') \otimes \texttt{actPrices}(A', L') \otimes \texttt{mktExchange}(A, L', Y) \otimes \texttt{priceQ}(A', Y, \texttt{consP}(ID', N', T', L)) \otimes \texttt{nat-less}(N, N') \otimes \texttt{nat-minus}(N', N, N'') \otimes \texttt{time}(T) \multimap \{\texttt{queue}(Q) \otimes \texttt{priceQ}(A', Y, (\texttt{consP}(ID', N'', T', L))) \otimes \texttt{actPrices}(A', L') \otimes \texttt{time}(\texttt{s}(T))\}$$

Figure 5: Exchanging market orders

market, nothing happens and the cancel order is removed from the order queue. If it is there, it is removed from the price queue. Similarly as in exchanging limit orders this results in two sub-cases: the order is the last one in its price queue or not.

The rules for performing order canceling are given in Figure 6.

## 5 Towards a Mechanized Verification of ATS Properties

Using our formalization we are able to check that this combination of order-matching rules does not violate some of the expected ATS properties. Although CLF is a powerful logical framework fit for specifying the syntax and semantics of concurrent systems, stating and proving properties about these systems goes beyond its current expressive power. For this task, one needs to consider states of computation, and the execution traces that lead from one state to another. Recent developments show that CLF contexts (the states of computation) can be described in CLF itself through the notion of generative grammars [13]. These are grammars whose language consists of all possible CLF contexts which satisfy the property being considered. The general idea is to show that every reachable state consists of a context in this language.

Using such grammars plus reasoning on steps and traces of computation, it is possible to state and

$$
\begin{array}{rl}
\texttt{cancel/inListNil:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{cancel}, A, P, ID, N, T), Q)) \otimes \texttt{actPrices}(A, L') \otimes \\
& \texttt{priceQ}(A, P, \texttt{consP}(ID, N, T', \texttt{nilP})) \otimes \texttt{remove}(L', P, L'') \otimes \texttt{time}(T) \\
\multicolumn{2}{r}{\multimap \{\texttt{queue}(Q) \otimes \texttt{actPrices}(A, L'') \otimes \texttt{time}(\texttt{s}(T))\}} \\
\\
\texttt{cancel/inListCons:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{cancel}, A, P, ID, N, T), Q1)) \otimes \\
& \texttt{priceQ}(A, P, Q) \otimes \texttt{inListF}(Q, ID) \otimes \texttt{removeF}(Q, ID, Q') \otimes \texttt{time}(T) \\
\multicolumn{2}{r}{\multimap \{\texttt{queue}(Q1) \otimes \texttt{priceQ}(A, P, Q') \otimes \texttt{time}(\texttt{s}(T))\}} \\
\\
\texttt{cancel/notInListQueue:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{cancel}, A, P, ID, N, T), Q1)) \otimes \\
& \texttt{priceQ}(A, P, Q) \otimes \texttt{notInListF}(Q, ID) \otimes \texttt{time}(T) \\
\multicolumn{2}{r}{\multimap \{\texttt{queue}(Q1) \otimes \texttt{priceQ}(A, P, Q) \otimes \texttt{time}(\texttt{s}(T))\}} \\
\\
\texttt{cancel/notInListActive:} & \texttt{queue}(\texttt{front}(\texttt{ordIn}(\texttt{cancel}, A, P, ID, N, T), Q1)) \otimes \texttt{actPrices}(A, L') \otimes \\
& \texttt{notInList}(L', P) \otimes \texttt{time}(T) \\
\multicolumn{2}{r}{\multimap \{\texttt{queue}(Q1) \otimes \texttt{actPrices}(A, L') \otimes \texttt{time}(\texttt{s}(T))\}}
\end{array}
$$

Figure 6: Cancel orders

prove meta-theorems about CLF specifications. This method is structured enough to become the meta-reasoning engine behind CLF [4], and therefore it is used for the proofs in this paper.

## 5.1 No Locked or Crossed Market

Here we show that the *bid* price $B$ is always less than the *ask* price $S$ in any reachable state using the rules presented in the previous sections. In other words, we show the following invariant:

**Property 1 (No locked-or-crossed market)** *If* $\texttt{actPrices}(\texttt{buy}, L_B)$ *and* $\texttt{actPrices}(\texttt{sell}, L_S)$ *and* $\texttt{maxP}(L_B, B)$ *and* $\texttt{minP}(L_S, S)$*, then* $B < S$*.*

Definition 2 shows a grammar that generates contexts, or states, satisfying Property 1. This is achieved by the guards $\texttt{maxP}(L_B, B)$, $\texttt{minP}(L_S, S)$, $B < S$ on the rewriting rule `gen/0` for the start symbol $\texttt{gen}(Q, L_B, L_S, T)$.

**Definition 2** *The following generative grammar* $\Sigma_{NLC}$[2] *produces only contexts where bid* $<$ *ask.*

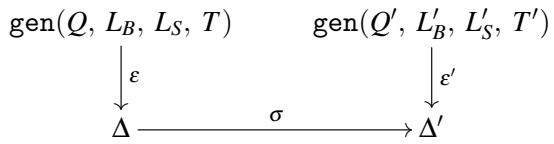
$$
\begin{array}{lcl}
\texttt{gen/0} & : & \texttt{gen}(Q, L_B, L_S, T) \otimes \texttt{maxP}(L_B, B) \otimes \texttt{minP}(L_S, S) \otimes B < S \\
& & \multimap \{\texttt{queue}(Q) \otimes \texttt{actPrices}(\texttt{buy}, L_B) \otimes \texttt{actPrices}(\texttt{sell}, L_S) \otimes \texttt{time}(T) \\
& & \quad \otimes \texttt{gen-buy}(L_B) \otimes \texttt{gen-sell}(L_S)\}. \\
\texttt{gen/buy1} & : & \texttt{gen-buy}(\texttt{nilP}) \multimap \{1\}. \\
\texttt{gen/buy2} & : & \texttt{gen-buy}(P :: L_B) \multimap \{\texttt{priceQ}(\texttt{buy}, P, L) \otimes \texttt{gen-buy}(L_B)\}. \\
\texttt{gen/sell1} & : & \texttt{gen-sell}(\texttt{nilP}) \multimap \{1\}. \\
\texttt{gen/sell2} & : & \texttt{gen-sell}(P :: L_S) \multimap \{\texttt{priceQ}(\texttt{sell}, P, L) \otimes \texttt{gen-sell}(L_S)\}.
\end{array}
$$

Intuitively, to show that the market is never in a locked-or-crossed state, we show that, given a context generated by the grammar in Definition 2, the application of an ATS rule (one step) will result in another context that can also be generated by this grammar. Coupled with the fact that computation starts at a valid context, this shows that the property is always preserved. More formally, we will show the theorem:

**Theorem 3** *For every* $\Delta \in L(\Sigma_{NLC})$ *and rule* $\sigma$*, if* $\Delta \xrightarrow{\sigma} \Delta'$*, then* $\Delta' \in L(\Sigma_{NLC})$*.*

[2] *NLC* stands for no locked-or-crossed.

This theorem can be represented visually as:




The proof consists in showing the existence of $\varepsilon'$.

**Proof** The proof proceeds by case analysis on $\sigma$. We consider only rules that change the linear facts $\texttt{actPrices}(\texttt{buy}, L_B)$ and $\texttt{actPrices}(\texttt{sell}, L_S)$, since otherwise we can simply take $\varepsilon' = \varepsilon$ (possibly with different instantiations for the variables $L$ in `gen/sell2` and `gen/buy2`). Moreover, we restrict ourselves to the case of incoming `buy` orders. The case for `sell` is analogous.

**Case $\sigma = \texttt{limit/empty}$** This rule rewrites $L_B$, the list of buy prices, to a list $L'_B$ which extends $L_B$ by a new price $P$. Since `store` was provable, we know that $P$ is less than the minimum sell price in the market. For `limit/empty` to be applicable, we need:

$$\Delta = \{\texttt{queue}(Q), \texttt{actPrices}(\texttt{buy}, L_B), \texttt{actPrices}(\texttt{sell}, L_S), \texttt{time}(T)\} \cup \Delta_1$$

In which case we conclude that:

$$\begin{array}{rcl} Q & = & \texttt{front}(\texttt{ordIn}(\texttt{buy}, A, P, ID, N, T), Q') \\ \varepsilon & = & \texttt{gen/0}(Q_g, L_B, L_S, T); \varepsilon_1; \varepsilon_2 \end{array}$$

where $\varepsilon_1; \varepsilon_2$ rewrite $\texttt{gen-buy}(L_B)$ and $\texttt{gen-sell}(L_S)$ to the $\texttt{priceQ}(\_, \_, \_)$ facts that form $\Delta_1$.
After applying the rule, the context is modified to:

$$\begin{array}{rcl} \Delta' & = & \{\texttt{queue}(Q'), \texttt{actPrices}(\texttt{buy}, L'_B), \texttt{actPrices}(\texttt{sell}, L_S), \texttt{time}(\texttt{s}(T))\} \\ & & \cup \{\texttt{priceQ}(\texttt{buy}, P, \texttt{consP}(ID, N, T, \texttt{nilP})), \Delta_1\} \end{array}$$

where $L'_B$ is computed by the $\texttt{insert}(L_B, P, L'_B)$ rule and consists of $L_B$ augmented by $P$.
A derivation of $\Delta'$ can be obtained via the following steps:

$$\varepsilon' = \texttt{gen/0}(Q, L'_B, L_S, \texttt{s}(T)); \varepsilon_1; \texttt{gen/sell2}; \varepsilon_2$$

where one extra step `gen/sell2`, with $L = \texttt{consP}(ID, N, T, \texttt{nilP})$, generates $\texttt{priceQ}(\texttt{buy}, P, L)$. Observe that the guard $B < S$ in `gen/0` still holds: if $\texttt{maxP}(L'_B, B)$ and $B \neq P$, then $B < S$ was part of the assumption. In case $\texttt{maxP}(L'_B, P)$, observe that $\texttt{store}(\texttt{buy}, L_S, P)$ only holds if $P < S$, where $\texttt{minP}(L_S, S)$. This property is related only to backward chaining predicates and can be proved in the LF framework Twelf using standard techniques.

**Case $\sigma = \texttt{limit/1}$** This rule rewrites $L_S$, the list of sell prices, to a list $L'_S$ which consists of $L_S$ without a price $X$. For `limit/1` to be applicable, we need:

$$\begin{array}{rcl} \Delta & = & \{\texttt{queue}(Q), \texttt{actPrices}(\texttt{buy}, L_B), \texttt{actPrices}(\texttt{sell}, L_S), \texttt{time}(T)\} \\ & & \cup \{\texttt{priceQ}(\texttt{sell}, X, \texttt{consP}(ID', N, T', \texttt{nilP})), \Delta_1\} \end{array}$$

where $X \in L_S$ is computed by $\texttt{exchange}(\texttt{buy}, L_S, P, X)$.
This can be derived by:

$$\begin{array}{rcl} Q & = & \texttt{front}(\texttt{ordIn}(\texttt{limit}, \texttt{buy}, P, ID, N, T), Q') \\ \varepsilon & = & \texttt{gen/0}(Q_g, L_B, L_S, T); \varepsilon_1; \texttt{gen/sell2}; \varepsilon_2 \end{array}$$

where $\varepsilon_1;\texttt{gen/sell2};\varepsilon_2$ rewrite $\texttt{gen-buy}(L_B)$ and $\texttt{gen-sell}(L_S)$ to the $\texttt{priceQ}(\_,\ \_,\ \_)$ facts that form $\Delta_1$, with the explicit `gen/sell2` generating the fact $\texttt{priceQ}(\texttt{sell},\ X,\ \texttt{consP}(ID',\ N,\ T',\ \texttt{nilP}))$.
After applying the rule, the context is modified to:

$$\Delta' = \{\texttt{queue}(Q'), \texttt{actPrices}(\texttt{buy},\ L_B), \texttt{actPrices}(\texttt{sell},\ L'_S), \texttt{time}(\texttt{s}(T))\} \cup \Delta_1$$

where $L'_S$ is computed by the $\texttt{remove}(L_S,\ X,\ L'_S)$ rule and consists of $L_S$ without $X$.
A derivation of $\Delta'$ can be obtained via the following steps:

$$\varepsilon' = \texttt{gen/0}(Q, L_B, L'_S, \texttt{s}(T)); \varepsilon_1; \varepsilon_2$$

Since $L'_S \subset L_S$, then, considering $\texttt{minP}(L_S,\ S)$ and $\texttt{minP}(L'_S,\ S')$, it is the case that $S \leq S'$. Thus $B < S$ implies $B < S'$. This can be proved in Twelf given the specification of the appropriate relations (such as $\subset$).

**Case $\sigma = \texttt{limit/3}$** This case is analogous to `limit/1`, except that the incoming order is only partially exchanged because its quantity $N$ is greater than the quantity $N'$ of the matching resident order. The initial context is:

$$\begin{array}{rcl} \Delta & = & \{\texttt{queue}(Q), \texttt{actPrices}(\texttt{buy},\ L_B), \texttt{actPrices}(\texttt{sell},\ L_S), \texttt{time}(T)\} \\ & & \cup\ \{\texttt{priceQ}(\texttt{sell},\ X,\ \texttt{consP}(ID',\ N',\ T',\ \texttt{nilP})), \Delta_1\} \end{array}$$

where $X \in L_S$ is computed by $\texttt{exchange}(\texttt{buy},\ L_S,\ P,\ X)$.
Which can be derived as before:

$$\begin{array}{rcl} Q & = & \texttt{front}(\texttt{ordIn}(\texttt{limit},\ \texttt{buy},\ P,\ ID,\ N,\ T), Q''),\ N > N' \\ \varepsilon & = & \texttt{gen/0}(Q, L_B, L_S, T); \varepsilon_1; \texttt{gen/sell2}; \varepsilon_2 \end{array}$$

where $\varepsilon_1;\texttt{gen/sell2};\varepsilon_2$ rewrite $\texttt{gen-buy}(L_B)$ and $\texttt{gen-sell}(L_S)$ to the $\texttt{priceQ}(\_,\ \_,\ \_)$ facts that form $\Delta_1$, with the explicit `gen/sell2` generating the fact $\texttt{priceQ}(\texttt{sell},\ X,\ \texttt{consP}(ID',\ N',\ T',\ \texttt{nilP}))$.
After applying the rule, the context is modified to:

$$\Delta' = \{\texttt{queue}(Q'), \texttt{actPrices}(\texttt{buy},\ L_B), \texttt{actPrices}(\texttt{sell},\ L'_S), \texttt{time}(\texttt{s}(T))\} \cup \Delta_1$$

where $L'_S$ is computed by the $\texttt{remove}(L_S,\ X,\ L'_S)$ rule and consists of $L_S$ without $X$.
A derivation of $\Delta'$ can be obtained via the following steps (note that time does not change for this rule):

$$\begin{array}{rcl} Q' & = & \texttt{front}(\texttt{ordIn}(\texttt{limit},\ \texttt{buy},\ P,\ ID,\ N - N',\ T), Q'') \\ \varepsilon' & = & \texttt{gen/0}(Q', L_B, L'_S, T); \varepsilon_1; \varepsilon_2 \end{array}$$

Since $L'_S \subset L_S$, then, considering $\texttt{minP}(L_S,\ S)$ and $\texttt{minP}(L'_S,\ S')$, it is the case that $S \leq S'$. Thus $B < S$ implies $B < S'$. As before, this argument can be developed in Twelf.

The cases for market orders, $\sigma = \texttt{market/\_}$, are analogous to the `limit/_` cases above. As for the case $\sigma = \texttt{cancel/inListNil}$ - this rule rewrites $L_B$, the list of buy prices, to $L'_B$ which is $L_B$ without a price $P$. This case is analogous to `limit/1` and `limit/3`, except that $\varepsilon$ contains an application of `gen/buy2` which is deleted to obtain $\varepsilon'$.

### 5.2 Exchanges happen at *bid* or *ask*

Every incoming order will either be exchanged or, if this is not possible, stored as a resident order to be exchanged when it is matched (if ever). The `exchange` predicate will be provable exactly when the incoming order can be exchanged, i.e., there exists an opposite resident order at an "acceptable" price. The acceptable price is: lower than the price of an incoming `buy` order, or greater than the price of an incoming `sell` order. Since orders have an associated quantity, the exchange may partially or totally consume the orders.

The price at which the exchange takes place is the best possible with respect to the *incoming* order. Consequently, an incoming `buy` order is exchanged at the minimal sell price (*ask*), while an incoming `sell` order is exchanged against the maximal available buy price (*bid*). Therefore whenever an order is exchanged it happens at either *bid* or *ask* price, and *only* at that price. In this section we show that this is indeed the case for our encoding. We will consider only the rules specifying exchange of *limit* orders (Figure 4). The case for *market* orders follows in a similar fashion, but considering the predicate mktExchange instead of exchange.

**Property 4** *All exchanges happen at and only at the price bid or ask.*

Note that Property 1 is a property of the reachable *states*, while Property 4 is concerns *transitions* between states. We can split it into two parts: (1) in every exchange, only one resident order of price $X$ is consumed; and (2) $X$ is *bid* or *ask*.

Part (1) can be more formally stated as:

**Theorem 5** *Let* $\Delta, \Delta' \in L(\Sigma_{NLC})$*, and* $\sigma$ *be one of the exchange rules* `limit/i` *for* $1 \leq \mathtt{i} \leq 5$*. If* $\Delta \xrightarrow{\sigma} \Delta'$*, then for all* $\mathtt{priceQ}(A,\ Y,\ L) \in \Delta$ *if* $Y \neq X$ *then* $\mathtt{priceQ}(A,\ Y,\ L) \in \Delta'$*, where* $X$ *is determined by* $\mathtt{exchange}(A',\ L,\ P,\ X)$ *on the left side of rule* $\sigma$*.*

**Proof** By inspection of the rules `limit/i`, we observe that the only facts of the shape $\mathtt{priceQ}(A,\ Z,\ L)$ involved in the rewriting are those where $Z = X$, where $X$ is bound by $\mathtt{exchange}(A',\ L,\ P,\ X)$.

Part (2) of the property can be stated more precisely as:

**Theorem 6** *If* $\mathtt{exchange}(\mathtt{buy},\ L,\ P,\ X)$ *then* $\mathtt{minP}(L,\ X)$*.*
*If* $\mathtt{exchange}(\mathtt{sell},\ L,\ P,\ X)$ *then* $\mathtt{maxP}(L,\ X)$*.*

The statement concerns only a backward chaining predicate, so the proof follows standard meta-reasoning techniques from LF, and can be implemented in a few lines in Twelf.

Taken together, Theorems 5 and 6 guarantee Property 4.

## 6 Conclusion and Future Work

We have formalized the core rules guiding the trade on exchanges worldwide. We have done this by formalizing an archetypal automated trading system in the concurrent logical framework CLF, with an implementation in Celf.

Encoding orders in a market as linear resources results in straightforward rules that either consume such orders when they are bought/sold, or store them in the market as resident orders. Moreover the specification is modular and easy to extend with new order types, which is often required in practice. This was our experience when adding market and immediate-or-cancel types of orders to the system.

The concurrent aspect of CLF simulates the non-determinism when orders are accumulated in the order queue, but, as explained, orders from the queue are processed sequentially[3].

Using our formalization we were able to prove two standard properties about a market working under these rules. First we proved that at any given state the *bid* price is smaller than the *ask*, i.e., the market is never in a locked-or-crossed state. Secondly we showed that the trade always take place at *bid* or *ask*. The first property was proved using generative grammars, an approach motivated by our goal to automate meta reasoning on CLF specifications (not implemented in the current version of Celf). Recent investigations indicate that this approach can handle many meta-theorems [13, 4] related to *state* invariants, and ours is yet another example. The second property is a combination of: (1) a property of a backward chaining predicate; and (2) a *transition* invariant. The former can be proved using established methods in LF (in fact, we have proved the desired property in Twelf). The second can be verified by inspection of the rules: the only linear facts that change in the next state, are those rewritten on the right side of $\multimap$.

This specification is an important case study for developing the necessary machinery for automated reasoning in CLF. It is one more evidence of the importance of quantification over steps and traces of a (forward-chaining) computation. It is interesting to note that our example combines forward and backward-chaining predicates, but the generative grammar approach still behaves well. In part because we are only concerned with a linear part of the context. Nevertheless, the proof still relied on some properties of backward chaining predicates. In the second proof, this is even more evident. This indicates, unsurprisingly, that meta-reasoning of CLF specifications must include the already developed meta-reasoning of LF specifications. In the meantime, we are investigating other properties of financial systems that present interesting challenges for meta-reasoning, such as showing that the price/time priority is respected upon exchange.

The difference between the proofs presented might be an indication that we need to follow a more general approach than the one used in LF. In that framework, most theorems that motivated the work have the same shape and their proofs follow the same strategy. Therefore, it is possible to save the user a lot of work by asking them to specify only the necessary parts that fills in a "proof template", which is then checked mechanically. When working with more ad-hoc systems, such as the case of financial exchanges, the properties and proofs are less regular, making it harder to figure out a good "template" that fits all properties of interest. In this case, it may be beneficial to leave more freedom (and consequently more work) to the user, in order to allow more flexible meta-reasoning.

Concurrently, we plan to formalize other models of financial trading systems, as this is a relevant application addressing some critical challenges.

## References

[1] Financial Conduct Authority (2018): *Algorithmic Trading Compliance in Wholesale Markets*. Available at `https://www.fca.org.uk/publication/multi-firm-reviews/algorithmic-trading-compliance-wholesale-markets.pdf`.

[2] Patrick Bahr, Jost Berthold & Martin Elsman (2015): *Certified Symbolic Management of Financial Multi-party Contracts*. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, ACM, pp. 315–327, doi:10.1145/2784731.2784747.

[3] Jan De Bel (1993): *Automated Trading Systems and the Concept of an Exchange in an International Context Proprietary Systems: A Regulatory Headache*. *U. Pa. J. Int'l Bus. L* 14(2), pp. 169–211.

[3] As far as we know, no real life trading system performs parallel order matching and execution.

[4] Iliano Cervesato & Jorge Luis Sacchini (2013): *Towards Meta-Reasoning in the Concurrent Logical Framework CLF*. *Electronic Proceedings in Theoretical Computer Science* 120, p. 216, doi:10.4204/eptcs.120.2.

[5] Iliano Cervesato, Kevin Watkins, Frank Pfenning & David Walker (2003): *A Concurrent Logical Framework I: Judgments and Properties*. Technical Report CMU-CS-02-101, Carnegie Mellon University.

[6] Matthew Freedman (2015): *Rise in SEC Dark Pool Fines*. *Review of Banking and Financial Law* 35(1), pp. 150–162.

[7] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *J. ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[8] Robert B. Jones, John W. O'Leary, Carl-Johan H. Seger, Mark D. Aagaard & Thomas F. Melham (2001): *Practical Formal Verification in Microprocessor Design*. *IEEE Des. Test* 18(4), pp. 16–25, doi:10.1109/54.936245.

[9] Phil Mackintosh (2014): *Demystifying Order Types*. Available at `http://www.smallake.kr/wp-content/uploads/2016/02/KCG_Demystifying-Order-Types_092414.pdf`.

[10] Grant Olney Passmore & Denis Ignatovich (2017): *Formal Verification of Financial Algorithms*. In Leonardo de Moura, editor: *Automated Deduction – CADE 26*, Springer International Publishing, pp. 26–41, doi:10.1007/978-3-319-63046-5_3.

[11] Dirk Pattinson & Carsten Schürmann (2015): *Vote Counting as Mathematical Proof*. In Bernhard Pfahringer & Jochen Renz, editors: *AI 2015: Advances in Artificial Intelligence*, Springer International Publishing, pp. 464–475, doi:10.1007/978-3-319-26350-2_41.

[12] Simon Peyton Jones, Jean-Marc Eber & Julian Seward (2000): *Composing Contracts: An Adventure in Financial Engineering (Functional Pearl)*. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, ACM, pp. 280–292, doi:10.1145/351240.351267.

[13] Robert J. Simmons (2012): *Substructural Logical Specifications*. Ph.D. thesis, Carnegie Mellon University. AAI3534965.

[14] Jean Souyris, Virginie Wiels, David Delmas & Hervé Delseny (2009): *Formal Verification of Avionics Software Products*. In Ana Cavalcanti & Dennis R. Dams, editors: *FM 2009: Formal Methods*, Springer Berlin Heidelberg, pp. 532–546, doi:10.1007/978-3-642-05089-3_34.