Denotational Semantics of General Payment Primitives, and Its Payment System

Miao, ZhiCheng
Co-founder, Superfluid Finance
miao@superfluid.finance

October 31, 2022

Abstract

Payment systems in the information age are still modeled after their analog predecessors. While electronic money payment systems do utilize computing technology and the Internet, this paper presents a case that true modernization can be reached by (a) making payments happening continuously over time, (b) involving more than two parties in payment if necessary, (c) having compositional financial contracts.

This paper first explores the foundation of modern payment systems, which consists of a money distribution model, payment primitives, payment execution systems of financial contracts, and different forms of money mediums. Then the paper uses *denotational semantics* to formally define payment primitives for modern payment systems. By the end, this paper includes an overview of the *Superfluid protocol*, a reference implementation of the payment primitives, and its payment system.

This paper is the first in the series of yellowpapers about modern payment systems dubbed "semantic money."

Revision History

Revision	Date	Author(s)	Description
0.9	2022-10-19	MZC	For reviews
1.0	2022-10-31	MZC	First release

Introduction

It should be fair to say every aspect of money is controversial: the nature of money, the value of money, money and banking, and monetary reconstruction. Two major schools of thought about the theory of money are the *Austrian school* ([1]) and the *Chicago school* ([2]). That was before the appearance of the

Internet-era version of monetary reconstruction, broadly defined as cryptocurrency, which challenges theories of money further and demands their updates ([3] [4]).

This yellow paper does not intend to address these controversies; instead, it focuses on the function of money. According to Von Mises:

> *The function of money is to facilitate the business of the market by acting as a common medium of exchange.*[1]

How do different forms of money perform this function, especially in the information age, when we increasingly use electronic forms of money?

This paper adds a new controversy to money, presenting a foundation of what constitutes a *modern payment system* and a set of *payment primitives* for the system to challenge the preconceived notions of how money can perform its function of medium of exchange.

In part I, we shall first explore the foundation. Here we present a formal definition of a payment system and its components. We then select a few relevant approaches used in computer science useful for modeling and defining formal specifications for the payment system.

One of the approaches is *denotational semantics*, which is used in part II of the paper to define the *general payment primitives*. Along with the denotational semantics, the paper also includes a restatement[2] of it in *Haskell programming language* ([5] [6] [7]).

In part III, we introduce a reference implementation of the general payment primitives and its payment system called *Superfluid Protocol*, along with its overview.

The end of the paper also includes notes on possible further investigations.

Part I
Foundation

1 Payment System

Here we present a definition of a payment system and its components.

Payment system It is solely defined by these components:

- *money distribution* models how monetary value is distributed amongst bearers [3],
- *payment primitives* update money distribution,
- *payment execution environment* performs payment primitives encoded in *financial contracts*,

[1] L. Von Mises, *The theory of money and credit*. Ludwig von Mises Institute, 2009, Chapter One, Chapter I, § 1, p1.

[2] It is a borrowed term from common law: "restatement of the law". In our case, the denotative mathematical laws.

[3] (Banking & Finance) a person who presents a note or bill for payment. - Collins English Dictionary

- and *forms of money medium* are the "user interfaces" of money for the bearers.

Modernization For its modern upgrade, the system should also have these properties:

- Money can be distributed continuously over time, as opposed to being in discrete chunks.
- Payment primitives can involve more than two parties, as opposed to being only for a sender and a receiver.
- Financial systems should be compositional.

1.1 Money Distribution

A representation of money and its distribution proposed in "a unifying theory" by Buldas, Saarepera, Steiner, *et al.* involves the following components:

- U is the set of monetary units.
- $\nu : U \to \mathbb{N}$ is the value function defining the value $\nu(u)$ of every value unit u. The set $\mathbb{N}$ is the set of all natural numbers, but instead, we can use any set of numerals that are totally ordered (*e.g.*, integers, real numbers).
- $\beta : U \to \mathcal{B}$ is the bearer function defining the bearer $\beta(u)$ of a unit. $\mathcal{B}$ is the set of possible bearers. The bearer is usually a legal construction defining any type of legal entity, such as a person, a family, a company, a state institution, etc.

This discrete nature of this money distribution model is schematically depicted in figure 1.

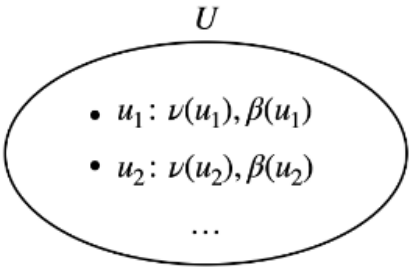

Figure 1: Schematic representation of discrete money distribution

1.1.1 Adding Context γ

But the discrete nature of the model does not provide the necessary element for us to add the desired properties to the payment system. So here we propose a modification that involves the usage of *context (γ)*:

Note that in this model, context can be updated independently, while value functions of the same money distribution can produce different monetary values.

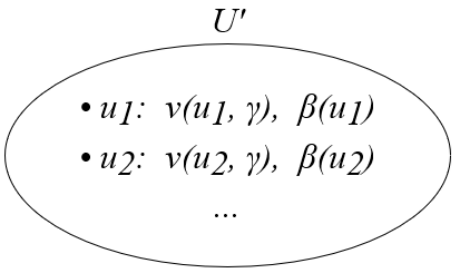

Figure 2: Schematic representation of money distribution with context

Components of Context Here are some possible components of context and how they work:

- If time (t) is included in the context, then the monetary value of each monetary unit can vary continuously over time. Time is part of the physical reality, hence not changeable by the actors in the payment system.
- Any subset of monetary units can also have their value functions depending on the same information in context. This could enable payment primitives that involve many parties. This set of information in context is referred to as *ctx :: SharedContext*; they can be changed over time by the actors in the payment system.

For the purpose of this paper, the model of context is: $\gamma = t \times ctx$.

1.1.2 Haskell Code Conventions Used

Before the first time this paper includes specification in Haskell, here are some highlights of conventions in the code style.

Language Extensions The specification is written in Haskell with *GHC2021 language set*[4].

Other notable extensions used in the paper are: *FunctionalDependencies*, *TypeFamilies*, *TypeFamilyDependencies*, *GADTs*.

Indexed Types and Type Equality To make the specification free of specific choices of core data types in implementation, *FunctionalDependencies*, *TypeFamilies*, and *TypeFamilyDependencies* are extensively used. As a consequence, the type signature can look very cluttered. In order to address that, type quality operator (∼) is also used. Here is an example snippets:

```
type (~) :: forall k. k -> k -> Constraint

monetaryValue :: ( mv  ~ MD_MVAL md
                 , mu  ~ MD_MU md
                 , ctx ~ MD_CTX md
```

[4]The complete set of the extensions is enumerated in `https://downloads.haskell.org/ghc/latest/docs/users_guide/exts/control.html#extension-GHC2021`.

```
      )
    => md -> (mu, ctx) -> mv
```

The otherwise tedious type family synonyms "`MD_XYZ md`" are rewritten with *mv mu ctx* with the help of ($\sim$) operator.

1.1.3 Haskell Definition Of Money Distribution

As the innermost layer of a modern payment system, money distribution models how monetary value is distributed amongst bearers.

```
class (Integral ν, Default ν) => MonetaryValue ν

class (Integral t, Default t) => Timestamp t

class Monoid ctx => SharedContext ctx

class Bearer brr

class Eq u => MonetaryUnit u

-- | Money distribution functions and indexed types.
class ( MonetaryValue (MD_MVAL md)
      , Timestamp (MD_TS md)
      -- t & mval should have the same representational type
      , Coercible (MD_TS md) (MD_MVAL md)
      , MonetaryUnit (MD_MU md)
      , Bearer (MD_BRR md)
      , SharedContext (MD_CTX md)
      , Monoid md
      ) => MoneyDistribution md where
    -- | Set of bearers.
    bearers :: ( brr ~ MD_BRR md
               )
            => md -> [brr]

    -- | Set of monetary units.
    monetaryUnits :: mu ~ MD_MU md
                  => md -> [mu]

    -- | Money distribution β function.
    bearerOf :: ( mu  ~ MD_MU md
                , brr ~ MD_BRR md
                )
             => md -> mu -> brr

    -- | Money distribution ν function.
    monetaryValueOf :: ( mv  ~ MD_MVAL md
                       , mu  ~ MD_MU md
                       , ctx ~ MD_CTX md
                       , t   ~ MD_TS md
                       )
```

```
                => md -> mu -> ctx -> t -> mv

type family MD_MVAL md = (mval :: Type) | mval -> md
type family MD_TS   md = (t    :: Type) | t    -> md
type family MD_MU   md = (mu   :: Type) | mu   -> md
type family MD_BRR  md = (brr  :: Type) | brr  -> md
type family MD_CTX  md = (ctx  :: Type) | ctx  -> md
```

1.2 Payment Primitives

A payment primitive is a data type with a generator function that produces payment primitive from the shared context ctx, primitive specific argument $args$, and timestamp t, along with an update of the shared context. Type signature 1 is for the generator of payment primitive a:

$$genPrim_a :: ctx \to args_a \to t \to prim \times ctx \tag{1}$$

Payment primitive data then can be used to create a delta update of money distribution[5]:

$$runPrim :: prim \to md \tag{2}$$

Loosely speaking, it is considered primitive, if it can not be broken down into other existing primitives, which result in the same money distribution; additionally, primitives should be the only constructs in a payment system that can update money distribution.

Updates are *monoidal* so that they can be incremental, and their parallel executions can be modeled.

The best-known primitive is an instant transfer of monetary value between one monetary unit to another. The introduction of context enables more primitives to be defined, and this will be discussed in part II.

1.3 Payment Execution Environment

The purpose of a payment execution environment is to perform the actual payment primitives, where their computation interface, parallel evaluation strategies, and payment system solvency are defined.

It is out of scope for this paper to survey in-depth the problem space of the operational semantics of payment execution environments. Nonetheless, a simplified model and some potential extensions are discussed briefly to place payment primitives in the big picture.

1.3.1 Composing Financial Contracts

A financial contract is the execution context for payment primitives, including their execution conditions, timing[6], and execution order.

Inspired by the *technique of composing financial contracts* demonstrated in [9], we define the type class for financial contracts as follows:

[5]Specifically being monoidal, that is in short a set that has associative binary operation and an identity element. See https://ncatlab.org/nlab/show/monoid.

[6]Timing is a type of condition of which current system time is a factor

```
-- | Composable financial contracts.
class MoneyDistribution md => FinancialContract fc md | fc -> md where
    -- | Predicate of the execution condition of a financial contract.
    fcPred :: ( ctx ~ MD_CTX md
              , Timestamp t
              )
           => fc -> (md, ctx) -> t -> Bool

    -- | Execute the payment primitives encoded in the financial
    -- contract.
    fcExec :: ( ctx ~ MD_CTX md
              , Timestamp t
              )
           => fc -> (md, ctx) -> t -> ((md, ctx), fc)
```

We know that both *md* and *ctx* are constrained to be monoid, then *(md, ctx)* must be monoidal too. With this, it is possible to build a combinatorial library of financial contracts that can be used to construct more complex financial contracts.

1.3.2 Simplified Execution Environment Models

Here are some models for the different payment execution environments.

Non-deterministic Sequential Execution Environment First, we define a model for a non-deterministic sequential payment execution environment, which includes a set of all financial contracts and a step-through function:

```
-- | Non-deterministic sequential payment execution environment.
class ( MoneyDistribution md
      , FinancialContract fc md
      , Monad env
      ) => NondetSeqPaymentExecEnv env md fc | env -> md, env -> fc where
    -- | Monadically update a financial contract in the execution
    -- environment.
    fcMUpdate :: fc -> env ()

    -- | Monadically select one financial contract from the execution
    -- environment.
    fcMSelect :: ( ctx ~ MD_CTX md
                 , Timestamp t
                 )
              => t -> env (md, ctx, fc)

    -- | Step through the execution environment.
    penvStepThrough :: ( ctx ~ MD_CTX md
                       , Timestamp t
                       )
                    => t -> env (md, ctx)
    -- Default implementation for the step through function.
```

```
penvStepThrough t = do
    (md, ctx, fc) <- fcMSelect t
    if fcPred fc (md, ctx) t
        then do
            let ((md', ctx'), fc') = fcExec fc (md, ctx) t
            fcMUpdate fc'
            -- (<>) operator is the binary operator for monoidal types.
            return ((md, ctx) <> (md', ctx'))
        else return (md, ctx)
```

We do not assume that *fcMSelect* yields a predicate that evaluates to true; since it could be an input from the external world. This won't work with any deterministic financial contract set.

The environment is a Monad, where different side effects for *fcMSelect* can be encoded. However, arrows could be used instead for a more generalized interface to computation[10].

Parallel Execution When the executions of payment primitives can be parallel, resource-sharing problems arise in their data storage when updating money distribution, context, and financial contract sets.

To model the parallel execution, ones must first study the concurrent control of the data storage system used ([11]), while formalism of parallel execution can be best done using Petri Nets ([12], [13])[7].

A model in Haskell will not be provided for now since it is out of the scope of the paper.

Deterministic Execution To make the execution environment deterministic, stronger ordering conditions must be provided to the financial contract type:

```
-- | Financial contract that can be totally ordered.
class ( MoneyDistribution md
      , FinancialContract tofc md
      , Ord tofc)
      => TotallyOrderedFinancialContract tofc md

-- | A partially ordered data type (incomplete definition).
class Poset a
-- omitting detailed interface of it.

-- | Financial contract that can be partially ordered.
class ( MoneyDistribution md
      , FinancialContract tofc md
      , Poset tofc)
      => PartiallyOrderedFinancialContract tofc md
```

Total ordered financial contract could be used to model deterministic sequential execution environment:

[7] Petri Nets World, https://www.informatik.uni-hamburg.de/TGI/PetriNets/index.php

```haskell
-- | Deterministic sequential payment execution environment.
class ( MoneyDistribution md
      , TotallyOrderedFinancialContract tofc md
      ) => DetSeqPaymentExecEnv env md tofc | env -> md, env -> tofc where
    -- | Update a financial contract in the execution environment.
    --
    --   Note:
    --
    --     * In order to keep well-ordering properties, the complexity
    --       of this function can be at least as bad as updating a
    --       sorted data structure £O(log(n))£.
    fcUpdate :: fc -> env -> env

    -- | Deterministically get the next financial contract
    --   executable at a specific time.
    fcNext :: ( ctx ~ MD_CTX md
              , Timestamp t
              )
           => env -> (md, ctx, tofc, t)

    -- | Update execution environment with new money distribution and
    -- context.
    penvUpdate :: ctx ~ MD_CTX md
               => env -> (md, ctx) -> env

    -- | Deterministically step through the execution environment
    penvDetStepThrough :: ( ctx ~ MD_CTX md
                          , Timestamp t
                          )
                       => env -> (env, t)
    -- Default implementation for the step through function.
    penvDetStepThrough env = let
        (md, ctx, fc, t) = fcNext env
        ((md', ctx'), fc') = fcExec fc (md, ctx) t
        -- assert: fcPred fc (md, ctx) t
        in (penvUpdate
            (fcUpdate fc' env)
            ((md, ctx) <> (md', ctx'))
           , t)
```

The environment is no longer monadic; that is to say, it is now fully deterministic. Instead, the monadic interactions with the external world should use *fcInsert* for adding new financial contracts to the environment.

A weaker condition, namely a poset (partially ordered) of financial contracts, may enable deterministic parallel executions of payments. However, model in Haskell will also not be provided for now.

1.4 Payment System Solvency

While money distribution does not assign meaning to the range of monetary values, negative values can have special meanings in real-world applications. In the following analysis, we call any money unit with a negative monetary value *insolvent*.

The detailed analysis of these solvency models is out of the scope of this paper.

Buffer Based Solvency Treatment In a non-deterministic execution environment, we cannot determine when any financial contract will be executed. That means there is always a chance a monetary unit could reach negative monetary value.

To mitigate the uncertainty of execution time, we introduce a concept called *buffer*. A buffer has a monetary value set aside in a solvency conditional financial contract, such that if a solvency condition arises, the buffer may be drawn to cover the loss introduced by the non-deterministic timing of the execution.

Deterministic Solvency Treatment Since *fcNext* returns what is the following financial contract executable at a specific time, this deterministic property eliminates the need for the buffer.

On the other hand, it introduces a different type of systemic risk: a kind of denial-of-service. Because the complexity of $fcUpdate$ is $O(log(n))$, the system may not be able to advance its system time until all the following executable contracts are executed.

1.5 Money Mediums

> *A useful observation about existing money schemes is that they all have some kind of monetary units that are physical or digital representations of money. Examples are bills, coins, bank accounts, Bitcoin UTXOs, etc.*[8]

We call them money mediums, and we further separate them into two big groups:

- *Token and its Accounts* - *e.g.*, bank currency accounts. Each token is its own centralized execution environment; bearers access their monetary value through their accounts and execute financial contracts through the token.
- *Note* - *e.g.*, federal reserve notes, bills, coins and Bitcoin UTXOs, etc. The execution environment is independent of the notes, but it needs notes to complete the execution of financial contracts.

One of the main differences is from the "user interface" perspective. A bearer expects to keep many notes in hands while maybe only needing a few accounts for each token. Also, it is up to bearers to keep track of all their notes, while tokens can keep track of most of the states for bearers; hence notes are more

[8] A. Buldas, M. Saarepera, J. Steiner, *et al.*, "A unifying theory of electronic money and payment systems," *TechRxiv. Preprint*, vol. 2021, 2021, P3.

decentralized and tokens are more centralized. Some also argue note-like model is better for complex concurrent and distributed computing environment[9].

1.5.1 Haskell Definition Of Money Mediums

Here are some toy models for non-deterministic sequential money tokens and money notes:

```
type Address = String

-- | Toy model for non-deterministic sequential money token.
class ( MoneyDistribution md
      , FinancialContract fc md
      , NondetSeqPaymentExecEnv tk md fc
      ) => NondetSeqMoneyToken tk md fc where
    -- | Customary interface for querying one's current account balance.
    balanceOf :: mval ~ MD_MVAL md
              => Address -> tk mval

    -- The rest would be just convenience interfaces for ~fcMInsert~

-- | A money note that is capable of encoding financial contract.
data MoneyNote md fc = ( MoneyDistribution md
                       , FinancialContract fc md
                       ) => FinancialContractNote md fc
                       | MoneytaryUnitNote md

type NoteID = String

-- | A toy model for non-deterministic sequential money notes execution
-- environment.
class ( MoneyDistribution md
      , FinancialContract fc md
      , NondetSeqPaymentExecEnv env md fc
      ) => NondetSeqMoneyNotes env md fc where
    -- | Find note by its ID. This should be used by ~fc~ to rehydrate
    --   the its references to the notes.
    findNote :: note ~ MoneyNote md fc
             => NoteID -> env note

    -- | Customary interface for querying the note's current balance.
    balanceIn ::  ( mval ~ MD_MVAL md
                  , note ~ MoneyNote md fc
                  )
              => note -> env mval
```

It may seem tiny semantic differences between tokens and notes execution

[9] M. M. Chakravarty, J. Chapman, K. MacKenzie, *et al.*, "The extended utxo model," in *International Conference on Financial Cryptography and Data Security*, Springer, 2020, pp. 525–539, P2.

environment, but it is on purpose. Their main difference lies mainly in their "user experience" implementations.

2 Relevant Approaches

The focus of this paper is to formally define a set of payment primitives that modernize our payment systems. However, to prevent reinventing wheels, we should first discuss some approaches in computer science that help tackle this challenge.

2.1 Functional Reactive Programming

Recall that we want our modern payment system to handle money distribution continuously over time and its financial contracts to be compositional. A very closely related software design paradigm best known to address those needs is *functional reactive programming (FRP)*. It was first introduced by Conal Elliott & Paul Hudak in solving multimedia animations ([15]). Later, Hudak also worked on [16] and [17], making FRP a more general framework for programming hybrid systems with continuous behaviors in a high-level, declarative manner.

After FRP got more adoption, it evolved into some variations that support discrete semantics and some variations better suited for interactive systems. However, in this paper, we will stick to and revisit the basic constructs of the original formulation used in [15] from which we will draw inspiration.

Temporal Modeling and Behaviors Values that vary over continuous time are called *behaviors*. They are first-class values and are built up compositionally. That is what we want in modern payment systems also. The semantic function of α-behaviors produces the value of type α of a behavior at a given time:

$$at : Behavior_\alpha \to Time \to \alpha \tag{3}$$

Event Modeling Like behaviors, *events* are first-class values too. The semantic function of α-event describes the time and information associated with an *occurrence* of the event:

$$occ : Event_\alpha \to Time \times \alpha \tag{4}$$

In modern payment systems, the payment primitives executed in payment execution environments are one type of event. More types of events can be read from the original paper[10].

Reactivity They key to modeling the payment execution environment using FRP is the *reactivity*, which makes behaviors reactive. Specifically, the behavior *b untilB e* exhibits b's behavior until *e* occurs, and then switches to a new behavior encoded in *e*:

[10] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, 1997, pp. 263–273, section 2.3 Semantics of Events.

$$
\begin{aligned}
&untilB : Behavior_\alpha \to Event_{Behavior_\alpha} \to Behavior_\alpha \\
&at\ [\![b\ until\ B]\!]\ t = if\ t\ \leqslant\ t_e\ then\ at\ [\![b]\!]\ t\ else\ at\ [\![b']\!]\ t \\
&\qquad where (t_e, b') = occ[\![e]\!]
\end{aligned} \tag{5}
$$

Note that $[\![.]\!]$ is the denotational semantics notation to be introduced later.

In the context of modern payment systems, the occurrences of payment primitives (events) change the behaviors of monetary units in terms of how much monetary value it has over continuous. The building blocks of the financial contracts should be about modeling these events and their reactivity declaratively.

2.2 Denotational Semantics

We also want a formal and precise specification of payment primitives. The title of the paper includes *denotational semantics*: "it is a *compositional* style for precisely specifying the meanings of languages, invented by Christopher Strachey and Dana Scott in the 1960s ([18])", and Conal Elliott proposed that denotational semantics can also be applied to *data types within a programming language*[11].

To create denotational semantics for each syntactic category $\mathcal{C}$, we should specify:

- a mathematical model $[\![\mathcal{C}]\!]$ of meanings, and
- a semantic function $[\![.]\!]_{\mathcal{C}} :: \mathcal{C} \to [\![\mathcal{C}]\!]$.

The syntactic category we are interested in is *payment primitives*, which we should treat as FRP-style *Behavior* data types. In the following chapters, it is also unambiguously referred to in short as $[\![.]\!]$. Various $[\![.]\!]$ must be compositional, *i.e.*, must be defined by structural recursion.

It is important to note that our purpose in using the denotational semantics is to give precise meaning to payment primitives independent of their implementations (which deal with performance, optimization, side effects, etc.). We will spend part II exploring the denotational semantics for general payment primitives in modern payment systems.

Part II
General Payment Primitives

The set of payment primitives supported in a payment system is general if (a) the monetary value of each monetary unit can vary continuously over time (b) monetary values can logically be shifted between two or more monetary units.

The extent of the generality of each payment system may vary. This paper introduces a set of payment primitives that is general and serves as a starting point for the readers to explore the space of modern payment systems.

[11] C. Elliott, "Denotational design with type class morphisms (extended version)," LambdaPix, Tech. Rep. 2009-01, Mar. 2009. [Online]. Available: `http://conal.net/papers/type-class-morphisms`, section 2, denotational semantics and data types.

The specifications will be formally defined using denotational semantics, along with a restatement in Haskell also applied as a constructive artifact friendly to computer environment[12].

3 Denotational Semantics

Here is the convention for symbols used in the formulas:

- **M** is the *model for money distribution.*
- **m** is for *money distributions.*
- **U** is the *set of all money units* in money distribution.
- **u** is for *monetary units.*
- $\boldsymbol{\nu}$ is for *monetary values.*
- **t** is for *time.*

3.1 $\mathcal{M}$ - Syntax for Money Distribution

Recall in the definition of payment system previously, money distribution sits in the core of the system, and that is the syntactic category $\mathcal{M}$ we will be dealing with:

- The meaning of the mathematical *model* $[\![\mathcal{M}]\!]$ is money distribution.
- Semantic function $[\![.]\!] :: \mathcal{M} \to [\![\mathcal{M}]\!]$ evaluates the expression of money distribution, payment primitives, etc.

3.2 Money Distribution

Model This is the model for money distribution.

$$\begin{aligned} [\![M\ u\ t\ \nu]\!] &= u \to t \to \nu \\ [\![.]\!] &= M\ u\ t\ \nu \to (u \to t \to \nu) \end{aligned} \tag{6}$$

Monoidal With this model, $\mathcal{M}$ is also monoidal, hence compositional.

$$\begin{aligned} [\![\emptyset]\!] &= \lambda u \to \lambda t\ \to \emptyset \\ [\![m_a \oplus m_b]\!] &= \lambda u \to \lambda t\ \to [\![m_a]\!]\ u\ t\ +\ [\![m_b]\!]\ u\ t \end{aligned} \tag{7}$$

Knowing that function application category $a \to b$ is also monoidal:

$$\begin{aligned} \emptyset &= \lambda a \to \emptyset \\ f \oplus g &= \lambda a \to f\ a \oplus g\ a \end{aligned} \tag{8}$$

[12]One may argue for a restatement in *Agda* instead, for it has a richer dependently-type system for desirable constructive proofs. This matter will be dealt in the "future investigations" section of this paper.

With some substitutions, we get the desired *monoid homomorphism* property for $\mathcal{M}$.

$$\begin{aligned} [\![\emptyset]\!] &= \emptyset \\ [\![m_a \oplus m_b]\!] &= [\![m_a]\!] \oplus [\![m_b]\!] \end{aligned} \tag{9}$$

3.3 Payment Primitives

A payment primitive *prim* is a model that produces a monoidal money distribution, semantically representing an "update" to a money distribution:

$$[\![prim]\!] = [\![\Delta\ m]\!] \tag{10}$$

Law of Conservation of Value First of all, money distribution must obey the *law of conservation of value*:

$$\forall t \in \mathbb{R} \sum_{u \in U} [\![m]\!]\ u\ t = 0 \tag{11}$$

That is to say, at any given time, the sum of the monetary value of all monetary units shall always equal zero. Zero is used to keep the semantical meaning of payment systems simple and elegant. In its applications, a payment system may use some special monetary unit accounting for negative monetary values to accommodate concepts such as mining, minting, money printing, etc.

Restricted Money Distribution To come up with such lawful money distribution, we can divide and conquer. In a payment system, if we restrict that only payment primitives can provide "updates" to their money distribution so that money distribution is only a result of a sequence of payment primitive updates; then, we can have these axioms as the basis for the proof.

Axiom A of Restricted Money Distributions First, we must impose the law of conservation of value on payment primitives as an axiom in order to prove inductively that such restricted money distribution satisfies the law of conservation of value too.

$$\begin{aligned} &\textit{law of conservation of value for payment primitives} \\ &\forall t \in \mathbb{R}(\sum_{u \in U} [\![prim]\!]\ u\ t = 0) \end{aligned} \tag{12}$$

Axiom B of Restricted Money Distributions

$$\begin{aligned} &\textit{money distribution consists of updates from payment primitives} \\ &[\![m]\!] = [\![prim_1]\!] \oplus [\![prim_2]\!] \oplus [\![prim_3]\!] \oplus \cdots \end{aligned} \tag{13}$$

Proof of Restricted Money Distributions satisfying the law of conservation of value.

In addition to the axioms, given:

(base case with empty money distribution is trivial)

$$\forall t \in \mathbb{R} \sum_{u \in U} [\![\emptyset]\!]\ u\ t = 0$$

(monoid homomorphism)

$$\sum_{u \in U} [\![prim_1 \oplus prim_2]\!]\ u\ t = \sum_{u \in U} [\![prim_2]\!]\ u\ t \oplus \sum_{u \in U} [\![prim_2]\!]\ u\ t$$

We have:

$$\begin{aligned}
&\forall t \in \mathbb{R}. \\
&\sum_{u \in U} [\![m]\!]\ u\ t \\
&\textit{(applying Axiom B)} \\
=&\sum_{u \in U} ([\![prim_1]\!] \oplus [\![prim_2]\!] \oplus [\![prim_3]\!] \oplus \cdots)\ u\ t \\
&\textit{(applying monoid homomorphism)} \\
=&\sum_{u \in U} [\![prim_1]\!]\ u\ t \oplus \sum_{u \in U} [\![prim_2]\!]\ u\ t \oplus \sum_{u \in U} [\![prim_3]\!]\ u\ t \oplus \cdots \\
&\textit{(applying Axiom A)} \\
=&\ 0
\end{aligned} \tag{14}$$

■

3.4 One-to-One Payment Primitives

Here are the primitives involving a sender (u_a) and a receiver (u_b) (one-to-one payments).

Transfer Instant transferring of a fixed amount of monetary value x:

$$[\![transfer\ u_a\ u_b\ x]\!] = \lambda u \to \lambda t \to \begin{cases} -x & (u_a = u) \\ x & (u_b = u) \\ 0 & (otherwise) \end{cases} \tag{15}$$

(Constant) Flow Flowing of monetary value at a constant rate of r at time t':

$$[\![flow\ u_a\ u_b\ r\ t']\!] = \lambda u \to \lambda t \to \begin{cases} -r \cdot (t - t') & (u_a = u) \\ r \cdot (t - t') & (u_b = u) \\ 0 & (otherwise) \end{cases} \tag{16}$$

Decaying Flow Another way monetary value could flow is through an exponential decay function[13].

Symbolically, this process can be expressed by the following differential equation, where N is the quantity and λ is a positive rate called the exponential decay constant:

$$\frac{dN}{dt} = -\lambda N \tag{17}$$

The solution to this equation (see derivation below) is:

$$N(t) = N_0 e^{-\lambda t} \tag{18}$$

But a more convenient semantics of it uses the parameter called "distribution limit" (θ) instead. In this formulation, a decaying flow distributes ϵ amount of monetary value at a rate started at α and halving every time period of $\frac{ln(2)}{\lambda}$:

$$[\![decayingFlow\ u_a\ u_b\ \theta\ \lambda\ t']\!] = \lambda u \to \lambda t \to \begin{cases} \alpha \cdot e^{-\lambda(t'-t)} - \theta & (u_a = u) \\ -\alpha \cdot e^{-\lambda(t'-t)} + \theta & (u_b = u) \\ 0 & (otherwise) \end{cases} \tag{19}$$

For the simplicity of the later discussion, this form of payment primitive is omitted.

3.5 Index Abstraction

To make primitives support more than two parties, we introduce an abstraction called *index*.

An *index* (we use symbol k for them) has a function ρ which produces a real number for each monetary unit:

$$\rho\ k :: u \to \mathbb{R} \tag{20}$$

To make it meaningful, it must satisfy the following law:

$$\sum_{u \in U} \rho\ k\ u = 1 \tag{21}$$

Semantically, it represents a proportion of each money unit, and they must add up to 1.

[13] `https://en.wikipedia.org/wiki/Exponential_decay`.

3.6 Indexed Primitives

Let's generalize the one-to-one payment primitives using index abstraction. We use I subscript for the indexed versions of the primitives.

Indexed Transfer

$$\begin{aligned}[\![transfer_I\ k_a\ k_b\ x]\!] = \lambda u \to \lambda t \to \\ - x \cdot \rho\ k_a\ u + x \cdot \rho\ k_b\ u\end{aligned} \tag{22}$$

Indexed (Constant) Flow

$$\begin{aligned}[\![flow_I\ k_a\ k_b\ r\ t']\!] = \lambda u \to \lambda t \to \\ - r \cdot (t - t') \cdot \rho\ k_a\ u + r \cdot (t - t') \cdot \rho\ k_b\ u\end{aligned} \tag{23}$$

It should be straightforward to prove that they also satisfy the *axiom A of restricted money distribution* thanks to the law of the index.

Universal Index Now one to one payment primitives can be redefined using *universal index* (ku_{any}):

$$\rho\ ku_a = \lambda u \to if\ u\ =\ u_a\ then\ 1\ else\ 0 \tag{24}$$

It means that it is an index *universally* available for each monetary unit, and its proportion is always 1 for that monetary unit and 0 for all others.

Proportional Distribution Primitives A special case of the indexed primitives is to fix the sender side to be an *universal index*.

3.7 Network Abstraction

An even more general abstraction is to model participants involved a payment primitive *network* (we use the symbol w for them).

It has the function ρ as in *index*, and it satisfies a different law:

$$\sum_{u \in U} \rho\ w\ u = 0 \tag{25}$$

3.8 Networked Primitives

Let's generalize the basic payment primitives using network abstraction. We use N subscript for the indexed versions of the primitives.

Shift We rename $transfer$ to $shift$ for networked instant payment primitive:

$$[\![shift_N\ w\ x]\!] = \lambda u \to \lambda t \to x \cdot \rho\ w\ u \tag{26}$$

Networked (Constant) Flow

$$[\![flow_N\ w\ r\ t']\!] = \lambda u \to \lambda t \to r \cdot (t - t') \cdot \rho\ w\ u \tag{27}$$

3.9 Generality vs. Optimization

The question now is, should we use the most general form of semantics to guide implementation?

The answer is most likely a no. Not because it is not useful; after all mathematical formula is about truth and perhaps also elegance in it, but because of we may miss optimization opportunities needed in implementation when more specialized versions are used instead[14].

In part III, we will look into a reference implementation where these optimizations are made.

4 Restatement in Haskell

Here is the denotational semantics of payment primitives of modern payment system.

```
-- | Type synonym for ⟦ℳ⟧.
type MoneyDistributionModel' md = forall ν t u.
    ( MoneyDistribution md
    , ν ~ MD_MVAL md
    , t ~ MD_TS md
    , u ~ MD_MU md
    ) => u -> t -> ν

-- | ⟦ℳ⟧ - methematical model of meaning in money distribution.
data MoneyDistributionModel md = MkMoneyDistributionModel
    (MoneyDistributionModel' md)

-- | Semigroup class instance ⟦ℳ⟧.
instance ( MoneyDistribution md
         ) => Semigroup (MoneyDistributionModel md) where
    -- ⊕: monoid binary operator
    (MkMoneyDistributionModel ma) <> (MkMoneyDistributionModel mb) =
        MkMoneyDistributionModel (\u -> \t -> ma u t + mb u t)

-- | Monoid class instance ⟦ℳ⟧.
instance ( MoneyDistribution md
         ) => Monoid (MoneyDistributionModel md) where
    -- ∅: monoid empty set
    mempty = MkMoneyDistributionModel (\_ -> \_ -> 0)

-- | Index abstraction.
class Eq u => Index k u | k -> u where
    ρ :: k -> u -> Double

-- | Universal index.
data UniversalIndex u = MkUniversalIndex u
```

[14] We will not though discuss the subjective aspects, *e.g.* in software engineering principles such as YAGNI (`https://en.wikipedia.org/wiki/You_aren't_gonna_need_it`), or user experience perspective of the payment primitives.

```haskell
instance Eq u => Index (UniversalIndex u) u where
    ρ (MkUniversalIndex u) u' = if u == u' then 1 else 0

-- | 𝓜' - syntactic category using index abstraction.
data 𝓜' ν t u =
    forall k1 k2. (Index k1 u, Index k2 u) => TransferI k1 k2 ν |
    forall k1 k2. (Index k1 u, Index k2 u) => FlowI k1 k2 ν t

-- | Type synonym for 𝓜' using type family.
type 𝓜 md = forall ν t u.
    ( MoneyDistribution md
    , ν ~ MD_MVAL md
    , t ~ MD_TS md
    , u ~ MD_MU md
    ) => 𝓜' ν t u

-- | ⟦.⟧ - semantic function of 𝓜.
sem :: MoneyDistribution md
    => 𝓜 md -> MoneyDistributionModel' md
sem (TransferI ka kb amount) = \u -> \_ ->
    let x = fromIntegral amount
    in ceiling $ -x * ρ ka u + x * ρ kb u
sem (FlowI ka kb r t') = \u -> \t ->
    let x = fromIntegral $ -r * coerce(t - t')
    in ceiling $ -x * ρ ka u + x * ρ kb u
-- GHC 9.4.2 bug re non-exhaustive pattern matching?
sem _ = error "huh?"

-- | ⟦.⟧ - semantic function of 𝓜.
sem𝓜 :: MoneyDistribution md
     => 𝓜 md -> MoneyDistributionModel md
sem𝓜 s = MkMoneyDistributionModel (sem s)
```

Part III
Superfluid Protocol - A Reference Implementation

Superfluid protocol ("the protocol")[15] is the first implementation of the *denotational semantics of payment primitives* (though before it was formalized and named so) on *Ethereum Virtual Machine* ([20]). The first version of the protocol is written in Solidity programming language [16].

To better serve as a reference implementation of the *modern payment system* formalized by this paper and its sequels, an implementation in Haskell was created. It aims to implement the full specifications of (a) the denotational

[15] The code base is available at `https://github.com/superfluid-finance/protocol-monorepo/`.

[16] `https://soliditylang.org/` - About Solidity programming language.

semantics of payment primitives, (b) compositional financial contracts for the payment primitives, (c) token & note model of money mediums, and their execution environment.

This paper covers the overview of the implementation with regard to (a).

5 Real Time Balance

For the purpose of compositional financial contracts, it is useful to separate monetary values that bearers can use immediately (called *untappedValue* in code) from ones that are set aside for other financial purpose. The concept of *real time balance* is thus created. *Real time balance* is a functorful[17] of monetary values. It can be converted to a single monetary value.

Since this is not the subject of this paper, it suffices to show its definition instead:

```
-- Note that we omit the definition of AnyTypedValue here,
-- since it is not relevant to the idea here.

class ( MonetaryValue v
      , Foldable rtbF
      , Monoid (rtbF v)
      , Eq (rtbF v)
      ) => RealTimeBalance rtbF v | rtbF -> v where

    -- | Convert a single monetary value to a RTB value.
    valueToRTB :: Proxy rtbF -> v -> rtbF v

    -- | Net monetary value of a RTB value.
    netValueOfRTB :: rtbF v -> v
    netValueOfRTB = foldr (+) def

    -- | Convert typed values to a RTB value.
    typedValuesToRTB :: [AnyTypedValue v] -> rtbF v

    -- | Get typed values from a RTB value.
    typedValuesFromRTB :: rtbF v -> [AnyTypedValue v]
```

In the paper, we refer to the real time balance values as *rtb*, and its function $netValueOfRTB$ is what matters the most here, it converts any *rtb* to the *monetary value* which is what the model of money distribution needs.

6 Agreement Framework

The main tasks of the implementer of *denotational payment primitives* are:

- preserve the program correctness (and if possible, with proof of equivalence),

[17] Bartosz Milewski has an excellent series on functors: `https://bartoszmilewski.com/2015/01/20/functors/`, where he uses the term "functorful" to convey the idea of generalized container.

- optimize for efficient computations,
- and provide a software interface for it.

The protocol introduces *agreement framework* to address the optimization needs and to provide a consistent software interface called "agreement" with agreement laws for reasoning about the correctness.

6.1 Monetary Unit Data (MUD)

Recall that all monetary units have a monetary value function: $\nu :: u \to \mathbb{N}$.

Agreement framework defines a concept called *monetary unit data*, and each monetary unit has a set of them:

```
class ( SuperfluidCoreTypes sft
      ) => MonetaryUnitDataClass mud sft | mud -> sft where
    -- | π function - balance provided (hear: π) by the monetary unit data.
    balanceProvided
        :: forall t rtb.
           -- indexed type aliases
           ( t ~ SFT_TS sft
           , rtb ~ SFT_RTB sft
           )
        => mud -> t -> rtb

-- | A semigroup constrained monetary unit data type class.
--
-- Note: a. ~mud~ that doesn't have a binary function may also be referred
--          to as "non-scalable" ~mud~.
--
--       a. What can make a ~mud~ "scalable" then is exactly when it is an
--          actual semigroup. Since a new state can be merged onto the
--          previous state to a new single state. It is still worth mentioning
--          that it is only a sufficient condition, since a monoid could still
--          "cheat" by linearly grow its data size on each binary operation.
class ( MonetaryUnitDataClass smud sft
      , Semigroup smud
      ) => SemigroupMonetaryUnitData smud sft
```

The π function produces *real time balance* at a particular time for *monetary unit data*. How a monetary unit produces monetary value from its set of monetary unit data is described in the "hierarchy of agreements" section.

6.2 Agreement Contract

Recall how a *payment primitive* generator should look like:

$$genPrim_a :: ctx \to args_a \to t \to prim \times ctx \tag{28}$$

Shared context ctx is clearly the only data can be used by optimization, since it is updated each time a primitive is generated. In the agreement framework, a data type *agreement contract* in the shared context is to fulfill this role:

```
-- | Agreement contract type class.
class ( SuperfluidCoreTypes sft
      , Default ac
      , MonetaryUnitDataClass ac sft
      , Traversable (AgreementOperationOutputF ac) -- <= Foldable Functor
      , Monoid (AgreementOperationOutput ac)
      ) => AgreementContract ac sft | ac -> sft where

    -- | ω function - apply agreement operation ~ao~ (hear: ω) to the agreement
    --                operation data ~ac~ to get a tuple of:
    --
    --   1. An updated ~ac'~.
    --
    --   2. A functorful delta of agreement monetary unit data ~mudsΔ~, which
    --      then can be appended to existing ~mudΔ~.  This is what can make an
    --      agreement scalable.
    applyAgreementOperation
        :: forall t ao aoo.
           -- indexed type aliases
           ( t ~ SFT_TS sft
           , ao ~ AgreementOperation ac
           , aoo ~ AgreementOperationOutput ac
           )
        => ac -> ao -> t -> (ac, aoo)

    -- | ϕ' function - functorize the existential semigroup monetary unit data
    --                 of agreement operation output
    functorizeAgreementOperationOutput
        :: forall any_smud muds f.
           ( any_smud `IsAnyTypeOf` MPTC_Flip SemigroupMonetaryUnitData sft
           , MonetaryUnitDataClass any_smud sft
           -- indexed type aliases
           , muds ~ AgreementOperationOutput ac
           , f ~ AgreementOperationOutputF ac
           )
        => Proxy any_smud
        -> muds -> f any_smud

    data family AgreementOperation ac :: Type
    data family AgreementOperationOutputF ac :: Type -> Type
    type family AgreementOperationOutput ac = (smuds :: Type) | smuds -> ac
```

The ω function (*applyAgreementOperation*) is the *genPrim* in the agreement framework. It takes in a agreement contract, an operation onto it and the current time; then it spits out an update of the agreement contract and a functorful of new delta of monetary unit data.

Figure 3 is a illustration of the ω function "machinery".

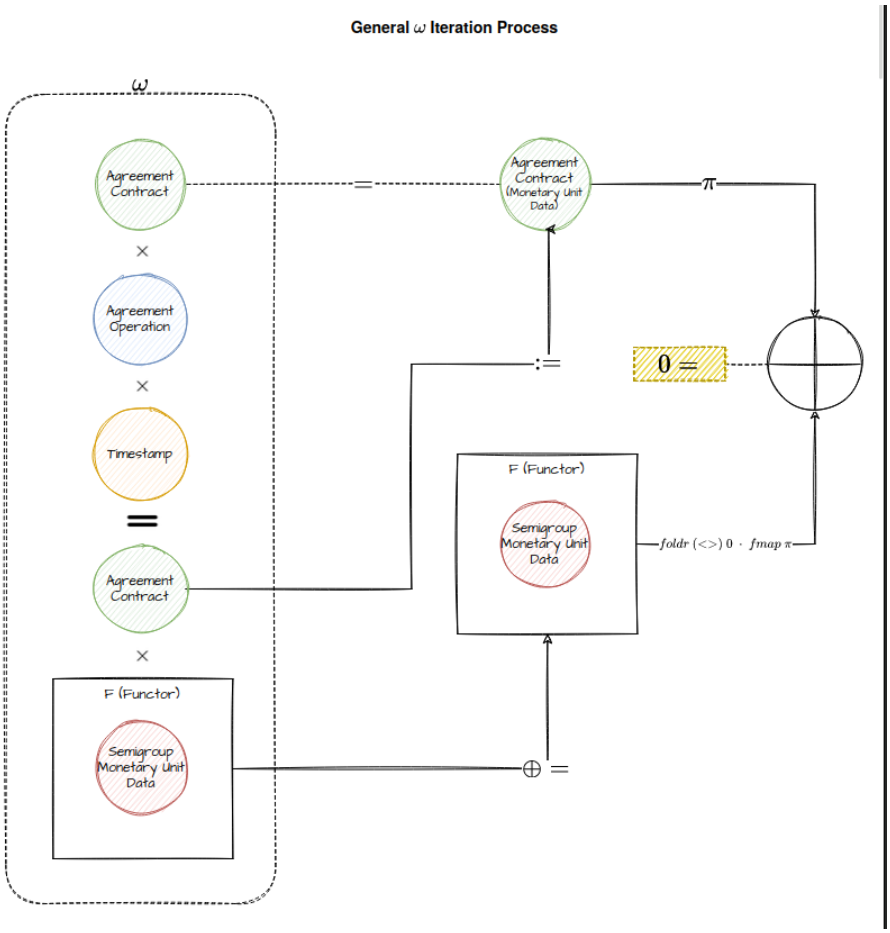

Figure 3: Agreement Contract ω Function

It also illustrates that it is expected the process should respect the *law of conservation of value* mandated by the *restricted money distribution* model.

It is also important to note that *AgreementContract* itself is also a *MonetaryUnitDataClass*, but it is not a semigroup, hence it is used to replace the previous agreement contract data. This has optimization implications, as in that if agreement contracts do not produce zero balance then they must be included in the ν function, that can make the ν function $O(N)$ to the number of agreement contracts a monetary unit is associated with.

It may have been self evident that agreement contract is a analogy to the fact that it encodes "ongoing relationships" like legal contracts do between bearers.

7 Lens Data Accessors

In order to support *index abstraction* in the denotational payment primitives, Monetary unit data are created with *lens data accessors*[18]:

```
-- representation of lenses
data Lens a b s t = Lens { view :: s -> a, update :: (b, s) -> t }
-- the pro-functor version of it
type Lens s t a b = forall f. Functor f => (a -> f b) -> s -> f t
```

The pro-functor version of lens might seem very obscure at first, but its data representation is rather self-explanatory: a lens is simply a pair of getter (*view*) and setter (*update*) encoded in the data.

With the help of Lens, then we can create the payment primitives independent of the choices between 1-to-1, 1-to-N, etc. Here are how they are defined for each class of payment primitives:

8 Useful Agreements

8.1 Instant Value MUD

This is for agreements where value is instantly transferred.

```
class ( Default amuLs
      , SuperfluidSystemTypes sft
      ) => MonetaryUnitLenses amuLs sft | amuLs -> sft where
    untappedValue :: Lens' amuLs (UntappedValue (SFT_MVAL sft))

type MonetaryUnitData :: Type -> Type -> Type
newtype MonetaryUnitData amuLs sft =
    MkMonetaryUnitData { getMonetaryUnitLenses :: amuLs }
    deriving (Default)

instance MonetaryUnitLenses amuLs sft
    => Semigroup (MonetaryUnitData amuLs sft) where
    MkMonetaryUnitData a <> MkMonetaryUnitData b =
        let c = a & over untappedValue (+ b^.untappedValue)
        in MkMonetaryUnitData c

instance MonetaryUnitLenses amuLs sft
    => MonetaryUnitDataClass (MonetaryUnitData amuLs sft) sft where
    balanceProvided (MkMonetaryUnitData a) _ =
        typedValuesToRTB [mkAnyTypedValue $ a^.untappedValue]
```

Note that (a) lens **operator (^.)** is the view function (getter) of data, (b) semigroup **operator (<>)** defines how two monetary unit data can be combined into one.

[18] B. Clarke, D. Elkins, J. Gibbons, *et al.*, "Profunctor optics, a categorical update," *arXiv preprint arXiv:2001.07488*, 2020.

8.2 Constant Flow Agreement (CFA) MUD

A more interesting case is where monetary value is changing continuously over time at a constant rate:

```
class ( Default amuLs
      , SuperfluidSystemTypes sft
      ) => MonetaryUnitLenses amuLs sft | amuLs -> sft where
    settledAt    :: Lens' amuLs (SFT_TS sft)
    settledValue :: Lens' amuLs (UntappedValue (SFT_MVAL sft))
    netFlowRate  :: Lens' amuLs (SFT_MVAL sft)

type MonetaryUnitData :: Type -> Type -> Type
newtype MonetaryUnitData amuLs sft =
    MkMonetaryUnitData { getMonetaryUnitLenses :: amuLs }
    deriving (Default)

instance MonetaryUnitLenses amuLs sft
    => Semigroup (MonetaryUnitData amuLs sft) where
    MkMonetaryUnitData a <> MkMonetaryUnitData b =
        let t  = a^.settledAt
            t' = b^.settledAt
            settledΔ = MkUntappedValue $ a^.netFlowRate * fromIntegral (t' - t)
            c = a & set  settledAt    t'
                  & over netFlowRate  (+ b^.netFlowRate)
                  & over settledValue (+ (b^.settledValue + settledΔ))
        in MkMonetaryUnitData c

instance MonetaryUnitLenses amuLs sft
    => MonetaryUnitDataClass (MonetaryUnitData amuLs sft) sft where
    balanceProvided (MkMonetaryUnitData a) t =
        let b = uval_s + coerce (fr * fromIntegral (t - t_s))
        in  typedValuesToRTB [ mkAnyTypedValue b ]
        where t_s    = a^.settledAt
              uval_s = a^.settledValue
              fr     = a^.netFlowRate
```

Note the implementation of semigroup binary operator, this is where the optimization occurs: to use *netFlowRate* to fold monetary unit data into a single value.

8.3 Decaying Flow Agreement (CFA) MUD

```
class ( Default amuLs
      , SuperfluidSystemTypes sft
      ) => MonetaryUnitLenses amuLs sft | amuLs -> sft where
    decayingFactor  :: Lens' amuLs (SFT_FLOAT sft)
    settledAt       :: Lens' amuLs (SFT_TS sft)
    αVal            :: Lens' amuLs (SFT_FLOAT sft)
    ϵVal            :: Lens' amuLs (SFT_FLOAT sft)
```

```haskell
type MonetaryUnitData :: Type -> Type -> Type
newtype MonetaryUnitData amuLs sft =
    MkMonetaryUnitData { getMonetaryUnitLenses :: amuLs }
    deriving (Default)

instance MonetaryUnitLenses amuLs sft
    => Semigroup (MonetaryUnitData amuLs sft) where
    MkMonetaryUnitData a <> MkMonetaryUnitData b =
        let c = a & set  settledAt     (b^.settledAt)
                  & over αVal          (\α -> α * exp (-λ * t_Δ) - ϵ')
                  & over ϵVal          (+ ϵ')
        in MkMonetaryUnitData c
        where ϵ'  = b^.ϵVal
              λ   = b^.decayingFactor
              t_Δ = fromIntegral (b^.settledAt - a^.settledAt)

instance MonetaryUnitLenses amuLs sft
    => MonetaryUnitDataClass (MonetaryUnitData amuLs sft) sft where
    balanceProvided (MkMonetaryUnitData a) t =
        let b = ceiling $ α * exp (-λ * t_Δ) + ϵ
        in  typedValuesToRTB [ (mkAnyTypedValue . MkUntappedValue) b ]
        where t_s   = a^.settledAt
              α     = a^.αVal
              ϵ     = a^.ϵVal
              λ     = a^.decayingFactor
              t_Δ   = fromIntegral (t - t_s)
```

8.4 Universal Index (UIDX)

An universal index then can be implemented trivially by representing lenses using the record syntax directly. For example, for constant flow monetary unit data:

```haskell
data MonetaryUnitLenses sft = MonetaryUnitLenses
    { settled_at    :: SFT_TS sft
    , settled_value :: UntappedValue (SFT_MVAL sft)
    , net_flow_rate :: SFT_MVAL sft
    } deriving (Generic)

deriving instance SuperfluidSystemTypes sft
    => Default (MonetaryUnitLenses sft)

-- | Monetary unit lenses for the universal index.
instance SuperfluidSystemTypes sft
    => CFMUD.MonetaryUnitLenses (MonetaryUnitLenses sft) sft where
    settledAt          = $(field 'settled_at)
    settledValue       = $(field 'settled_value)
    netFlowRate        = $(field 'net_flow_rate)
```

The *field* is a template Haskell function to generate lens from record fields.

8.5 Proportional Distribution Index (PDIDX)

Proportional distribution index describes on-going *subscription* agreement contracts between one *publisher* and many *subscribers* to the publisher's *distribution*.

Its instant value variance is called *"Instant Distribution Agreement (IDA)"*.

Its constant flow variance is called *"Constant Flow Distribution Agreement (CFDA)"*.

9 Hierarchy of Agreements

To finish, figure 4 is the illustration of data structures for the denotational payment primitives implemented with agreement framework.

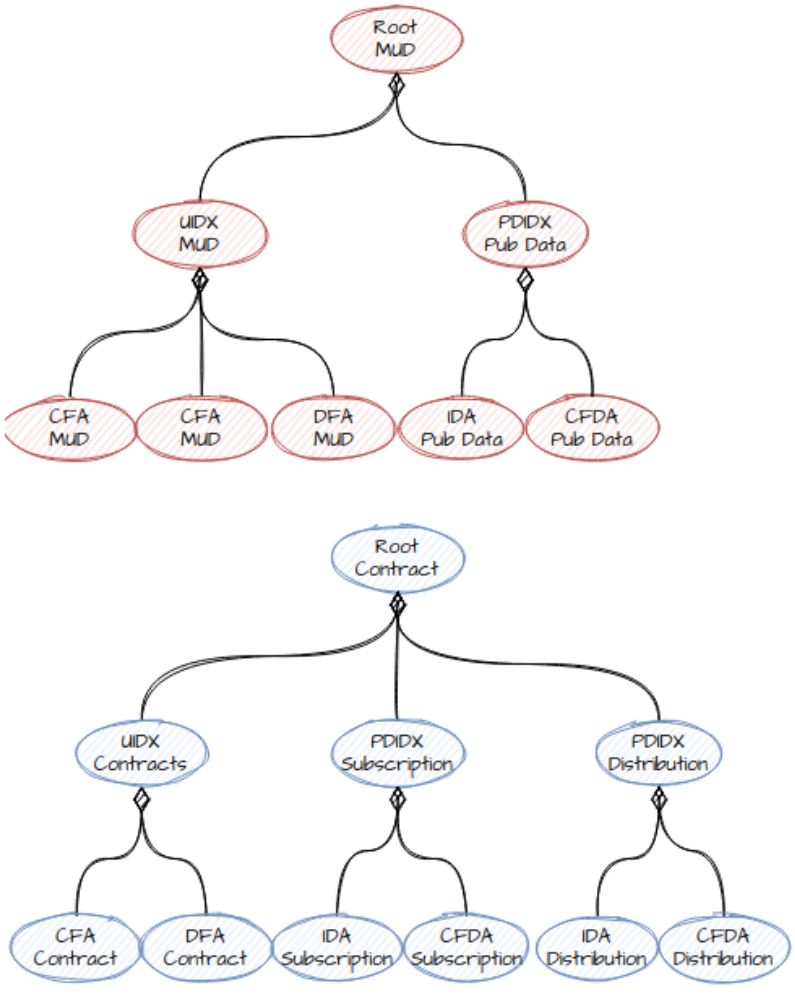

Figure 4: Agreement Data Structures

Note that with *Root MUD*, any monetary unit can traverse all its MUDs;

and with *Root Contract* relations between monetary units can also be searched by a simple algorithm.

This concludes the overview of the Superfluid protocol implementation. For more details, one should refer to the code repository mentioned before.

Part IV
Notes on Future Investigations

10 Restatement in Agda, Correctness and Equivalence Proofs

One of the advantages of using Haskell language to write the reference implementation was its industrial strength. Because of that, the implementation could be easily integrated into a production code base without significant performance compromises.

However, one major disadvantage is that it is not equipped with sufficient apparatus for program correctness proofs. The best it can offer without using more experimental Haskell features is to use *QuickCheck*[19] to test that the necessary laws are not violated using randomized test vectors.

To amend this deficiency, *Agda programming language* comes as a great candidate for a different constructive restatement of the formal specification.

It is based on the insight of the deep connection (equivalence/isomorphism) between logic and dependently typed programming, often called "the Curry–Howard correspondence", as discovered and developed during the 20th century (explained in [22] by Phillip Wadler as "Propositions as Types"). Agda being a dependently typed functional programming language fully embodies this insight, hence a good candidate for creating provable correct programs.

To learn more about Agda, refer to these footnotes [20] [21] [22].

11 Future Papers on Modern Payment System

The next yellow paper will formalize how the denotational payment primitives can be stitched together using a powerful combinator library pattern briefly described in [9].

The execution environments for these financial contracts are also fertile ground for new ideas, from deterministic execution avoiding the usage of buffer, to using distributed ledger technology to deep-embed these financial contracts in the consensus layer.

[19] `https://wiki.haskell.org/QuickCheck` - Haskell wiki page for QuickCheck.
[20] `https://plfa.github.io/` - Programming Language Foundations in Agda.
[21] `https://wiki.portal.chalmers.se/agda/pmwiki.php` - Agda Wiki.
[22] `https://wiki.portal.chalmers.se/agda/Main/Community` - Agda community page.

12 General Accounting Domains & Real Time Finance

> *Accounting, also known as accountancy, is the measurement, processing, and communication of financial and non-financial information about economic entities such as businesses and corporations.*[23]

The *real time balance* introduced in this paper can be seen as an instance in the *general accounting domain*, specifically for financial contracts in dealing with payment primitives, which can be seen as a real time version of cash flow accounting. Along with balance sheet accounting, and income statement accounting, the conversion between these instances of general accounting domains are often called "reconciliations".

More work can be done on how to create a simple and elegant automation system for reconciliations.

Real Time Finance *We define the term* ***real time finance*** *to mean a financial system where its payment system is modernized to handle money distribution continuously over time, its financial contracts are compositional, and its general accounting domain is automated and processed in real time.*

Conclusions

This paper defines what constitutes a modern payment system, with its payment primitives formally specified using denotational semantics and restated using Haskell programming language. Along with a reference implementation of the specification, we want to make a case that marrying with the progress of computer science, the way of money performs its function of medium of exchange can have a modern upgrade.

Absent any claim on whether a modern upgrade is needed; we invite the readers to ask ourselves a question together: with electronic money increasingly being used, should we keep emulating the function of money of its traditional quality, or could we rethink what it may be in the information age?

In future work, we will look into deeper other topics of real time finance and continue to refine the methodology used to achieve *simple and precise* specifications guiding correct and efficient engineering.

[23] B. E. Needles, M. Powers, and S. V. Crosson, *Principles of accounting*. Cengage Learning, 2013.

References

[1] L. Von Mises, *The theory of money and credit*. Ludwig von Mises Institute, 2009.

[2] M. Friedman, "Quantity theory of money," in *Money*, Springer, 1989, pp. 1–40.

[3] S. Ammous, "Can cryptocurrencies fulfil the functions of money?" *The Quarterly Review of Economics and Finance*, vol. 70, pp. 38–51, 2018.

[4] W. K. Härdle, C. R. Harvey, and R. C. Reule, *Understanding cryptocurrencies*, 2020.

[5] P. Hudak, S. Peyton Jones, P. Wadler, *et al.*, "Report on the programming language haskell: A non-strict, purely functional language version 1.2," *ACM SigPlan notices*, vol. 27, no. 5, pp. 1–164, 1992.

[6] S. P. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[7] S. Marlow *et al.*, "Haskell 2010 language report," 2010.

[8] A. Buldas, M. Saarepera, J. Steiner, and D. Draheim, "A unifying theory of electronic money and payment systems," *TechRxiv. Preprint*, vol. 2021, 2021.

[9] S. Peyton Jones, J.-M. Eber, and J. Seward, "Composing contracts: An adventure in financial engineering (functional pearl)," *ACM SIGPLAN Notices*, vol. 35, no. 9, pp. 280–292, 2000.

[10] J. Hughes, "Generalising monads to arrows," *Science of computer programming*, vol. 37, no. 1-3, pp. 67–111, 2000.

[11] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.

[12] C. A. Petri, "Kommunikation mit automaten," 1962.

[13] W. Reisig, *Petri nets: an introduction*. Springer Science & Business Media, 2012, vol. 4.

[14] M. M. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. Peyton Jones, and P. Wadler, "The extended utxo model," in *International Conference on Financial Cryptography and Data Security*, Springer, 2020, pp. 525–539.

[15] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, 1997, pp. 263–273.

[16] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *International School on Advanced Functional Programming*, Springer, 2002, pp. 159–187.

[17] Z. Wan and P. Hudak, "Functional reactive programming from first principles," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000, pp. 242–252.

[18] D. S. Scott and C. Strachey, *Toward a mathematical semantics for computer languages*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971, vol. 1.

[19] C. Elliott, "Denotational design with type class morphisms (extended version)," LambdaPix, Tech. Rep. 2009-01, Mar. 2009. [Online]. Available: `http://conal.net/papers/type-class-morphisms`.

[20] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[21] B. Clarke, D. Elkins, J. Gibbons, *et al.*, "Profunctor optics, a categorical update," *arXiv preprint arXiv:2001.07488*, 2020.

[22] P. Wadler, "Propositions as types," *Communications of the ACM*, vol. 58, no. 12, pp. 75–84, 2015.

[23] B. E. Needles, M. Powers, and S. V. Crosson, *Principles of accounting*. Cengage Learning, 2013.