Cover page

Exam information
NDAA93062E - Computer Science Thesis 30 ECTS, Department of Computer Science - Kontrakt:119856 (Juan-Manuel Torres Garcia)

Handed in by
Juan-Manuel Torres Garcia
rlj119@alumni.ku.dk

Exam administrators
DIKU Eksamen
uddannelse@diku.dk

Assessors
Fritz Henglein
Examiner
henglein@di.ku.dk
+4535335692

Hans Hüttel
Co-examiner
hans@cs.aau.dk

Hand-in information

Titel: Algebraic Resource Accounting for Transfers and Transformations
Vejleder / eksaminator: Fritz Henglein, Hans Hüttel
Tro og love-erklæring: Yes
Indeholder besvarelsen fortroligt materiale: No
Må besvarelsen gøres til genstand for udlån: Yes
Må besvarelsen bruges til undervisning: Yes

UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

MSc Thesis in Computer Science

Juan-Manuel Torres García

Algebraic Resource Accounting for Transfers and Transformations

Supervisor: Fritz Henglein

Submitted on September 30th, 2020

Abstract

In systems where goods and services are produced, exchanged and transformed, one usually resorts to some sort of resource accounting system in order to keep track of resource ownership. We generalize the previous work that has been done on modelling single-resource transformation-less systems to multi-resource systems by introducing an algebraic model, based on McCarthy's REA (Resources, Events, Agents) generalized resource accounting framework, that guarantees certain invariants necessary to ensure the validity of the system's state across events. This model is then extended to allow for production tracking by defining transformations as a type of event in the framework. We then propose an algorithm for transaction netting in the general case of multi-party, multi-resource events (including transfers and transformations). Finally, after conducting a short case-study of the Colombian coffee sector, we discuss how our algebraic model can be used to describe the interactions that take place in the coffee supply chain.

Acknowledgements

I would like to thank my supervisor, Fritz Henglein, for his guidance throughout this project. I owe a great debt of gratitude to all the team at the COWI project for giving me the opportunity to explore the Colombian coffee sector. I would also like to thank my parents, Erika and Yann, and my sisters, Mariana and Isabella, for helping me to keep sanity during these pandemic times, and without whose support I would have never been able to complete this work.

Contents

1 Introduction **5**

2 An introduction to linear algebra **7**

3 REA resource accounting **11**
3.1 Resources, ownership states and transfers 11
3.2 Ownership state predicates and transition functions 13
3.3 Example: the Ethereum Ledger 16

4 The netting problem **18**
4.1 Initial formulation 18
4.2 Two-way, single resource transfers 19
4.3 Generalization to multi-resource, multi-party transfers 21

5 Events and transformations **24**
5.1 Formal definitions 25
5.2 Event predicates 29

6 Application to the coffee supply chain **30**
6.1 Basic terminology 30
6.2 The Colombian coffee supply chain 31
6.3 The `Coffeebrain` smart contract 31
6.4 An algebraic description of the coffee supply chain 33

7 Discussion **36**
7.1 Related work 36
7.2 Future work 36

8 References **37**

A Source code for the Haskell implementation **39**

B Source code for the `Coffeebrain` smart contract **50**

C Copyright attribution **64**

1 Introduction

In a system where several agents perform resource exchanges, there arises the need for a tool to keep track of all transaction among such agents in order to be able to determine at all times the ownership status of resources. A framework that accomplishes such tracking might also be used for the orthogonal goal of keeping track of resource location, since in practice, ownership and location are often decoupled: while an actor might have ownership over a particular resource, they might not have that resource in their physical custody (like for example in the case of futures, or of an item which is currently in transit between its previous owner and its current one).

With his REA accounting framework, McCarthy[3] introduces the idea of viewing such systems in terms of Resources, Events and Agents, which we might informally define as follows:

- *Agents* are not only defined as physical persons, but also as for example companies or, in the case of location tracking, as sets of coordinates, addresses, etc.

- Likewise, we define *resources* as “anything that can be transferred”, that is physical goods as well as intangible concepts such as services or digital currency, among others.

- An *event* would then correspond to any real-world event that changes the state of the system (like for example resource transfers).

More generally, such an accounting model might apply to any system where we have a notion of associating resources to agents, with events being able to mutate such pairings.

We would expect that resource accounting model to prevent *double-spending* by keeping the *resource preservation* invariant across events, which states that the sum of all resources in all consecutive states of the system must remain constant, which would mean that no resource was duplicated or disappeared as a consequence of an event.

We also expect such a model to enforce *credit limits*, that is guarantee that some if not most actors should not be able to spend resources they do not own (or in other words, that their respective account balances cannot ever be negative). We will expand later on why we want this to apply to only *some* (albeit a majority of) agents rather than *all* of them. By extension of such a principle, we can imagine situations where potentially more complex predicates would need to be enforced on account balances, or more generally on the overall state of the system.

Lastly, as opposed to what systems such as Bitcoin or ERC20 Ethereum contracts ([10][12]) offer, we want resource types to be user-definable: our tool must be able to handle arbitrary resource types in order to gain the ability to handle complex transactions within a unified algebraic model.

While a framework that satisfies the aforementioned properties would be sufficient to model any complex monetary system, it does not extent to all potential value chains, since it lacks the concept of *transformations*: in the case of supply chain systems, there exists the notion of combining, or processing resources in order to turn them into something else (as exemplified by the relationship from raw materials to finished goods). It therefore becomes necessary to formulate a theory of such transformations, and of the properties they must satisfy in order for us to be able to assert that they are indeed a valid model of real-world transformation events. Note that not all value chains aim to preserve the same properties: while physical supply chain events might need to operate in a way consistent with the laws of physics (mass and energy conservation, etc.), a purely virtual system might want to enforce all kinds of other properties. We thus need a general way of specifying such constraints on events.

In this thesis, we will leverage the tools of abstract algebra, more specifically the theory of vector spaces, to construct a mathematical framework such that the four concepts described above (resources, events, agents and transformations) can be modelled and manipulated using well-known algebraic operations.

First, we will describe how ownership states can be represented as a vector spaces defined using the concept of coproducts of vector spaces over the field of real numbers. We will then see how transfers can be viewed as a subspace of ownership states, and discuss the benefits of such an approach to resource accounting. Having introduced the notion of ownership states and transfers, we will see how we can model validity predicates over the ownership states, and how state transitions can be represented as a monoidal construct derived from those predicates. We will extend the definition of transfers to also encompass transformations, at which point we will arrive to our final definition of what *events* mean in the context of this framework. This will then lead us to revisit the definition of predicates and the transition construct derived therefrom so that we reach a definition of our model that can describe a wide range of value networks.

Once the concept of events is formally defined, we will investigate how transaction settlement can be accomplished in the context of a resource manager that implements such an accounting framework. This leads us to discuss the concept of *netting*, that is, the selection among a set of pending transactions of the biggest subset thereof that satisfies the various predicates or constraints placed upon the state of the system.

Finally, we will discuss how this model can be applied to a real-world value chain by doing a case study of the Colombian coffee supply chain, supported by data from the COWI Coffeebrain project[5], and by comparing our framework to an existing smart contract for resource accounting in the coffee sector[6].

We provide a full Haskell implementation of this resource accounting mode in Appendix A.

2 An introduction to linear algebra

Below, we will define all the basic linear algebra structures and constructs we will use in the remainder of this text.

Definition 1 (Field) *A field is a set K, together with two operators, $+ : (K, K) \to K$) which we call addition and $\cdot : (K, K) \to K$ which we call multiplication, such that, for all $a, b, c \in K$:*

- $(a + b) + c = a + (b + c)$
- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
- $a + b = b + a$
- $a \cdot b = b \cdot a$
- *there is an element $0 \in K$ such that for all $a \in K$, $a + 0 = a$*
- *there is an element $1 \in K$ such that for all $a \in K$, $a \cdot 1 = a$*
- *for every $a \in K$ there is an element $-a \in K$ such that $a + (-a) = 0$*
- *for every $a \in K \setminus \{0\}$ there is an element $a^{-1} \in K$ such that $a \cdot a^{-1} = 1$*
- $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

Proposition 1 *$\mathbb{R}$ together with addition and multiplication is a field.*

Definition 2 (Vector space) *A vector space over a field K is a set V together with addition ($+$) and multiplication ($\cdot$) operators such that for all $u, v, w \in V$ and $a, b \in K$, we have:*

- $u + (v + w) = (u + v) + w$
- $u + v = v + u$

- *there is a* $\mathbf{0} \in V$ *such that for all* $u \in V$, $u + \mathbf{0} = u$
- *for all* $u \in V$ *there is a* $-u \in V$ *such that* $u + (-u) = 0$
- $a \cdot (b \cdot u) = (a \cdot b) \cdot u$
- $1 \cdot u = u$
- $a \cdot (u + v) = (a \cdot u) + (a \cdot v)$
- $(a + b) \cdot u = (a \cdot u) + (b \cdot u)$

Proposition 2 *All fields are vector spaces over themselves.*

Definition 3 (Linear combination) *Let* V *be a vector space over some field* K *and* S *a subset of* V*. We call* $a : S \to K$ *a linear decomposition of* $v \in V$ *if*

$$v = \sum_{x \in S} a(x) \cdot x.$$

We also call v *a linear combination over* S*.*

Definition 4 (Span) *Let* V *be a vector space over some field* K *and* S *a subset of* V*. We call the set of all linear combinations over* S *the span of* S*, notated* $\mathrm{span} S$*.*

Definition 5 (Linear independence) *Let* V *be a vector space over some field* K *and* S *a subset of* V*.* S *is linearly independent if the only linear decomposition of* 0 *over* S *is the function* $a : S \to K$ *such that, for all* $x \in S$*,* $a(x) = 0$*.*

Definition 6 (Basis) *Let* V *be a vector space over some field* K *and* B *a subset of* V*. We call* B *a basis of* V *if* $\mathrm{span} B = V$ *and* B *is linearly independent.*

Theorem 3 (Basis theorem) *Every vector space has a basis.*

Theorem 4 (Unique decomposition) *Let* V *be a vector space with basis* B*, then every* $v \in V$ *has exactly one linear decomposition over* B*.*

Theorem 5 (Dimension theorem) *All bases of a vector space have the same cardinality.*

Definition 7 (Dimension) *The dimension of a vector space* V*, notated* $\dim V$*, is the cardinality of a basis for it.*

Definition 8 (Support) *Let* V *be a vector space,* X *be a set and* $f : X \to V$ *a function from* X *to* V*. The support of* f *is* $\mathrm{supp}(f) = \{x \in X \mid f(x) \neq \mathbf{0}\}$*.*

Note: we refer to functions with finite support as *finite maps*.

Definition 9 (Homomorphism) *Let V and W be two vector spaces over a field K. A homomorphism (also called linear map) $f : V \to_1 W$ from V to W is a function $f : V \to W$ such that, for all $k \in K$ and $u, v \in V$:*

- $f(u + v) = f(u) + f(v)$
- $f(k \cdot u) = k \cdot f(u)$

If f is injective, then it is a monomorphism, if it is surjective, it is an epimorphism, and if it is bijective, it is an isomorphism. If there exists an isomorphism between two vector spaces V and W, we write $V \cong W$.

Definition 10 (Subspace) *Let V be a vector space over a field K and U a set. If $U \subseteq V$ such that:*

- $\mathbf{0} \in U$
- *For all $u, v \in U$, $u + v \in U$*
- *For all $u \in U$, $-u \in U$*
- *For all $k \in K$ and $u \in U$, $k \cdot u \in U$*

Then U together with the same operations as V is a vector space and we call U a subspace of V.

Definition 11 (Image, Kernel) *Let V, W be vector spaces over $\mathbb{R}$, and $f : V \to_1 W$. The image of f is $\operatorname{im} f = \{f(v) \mid v \in V\}$. The kernel of f is $\ker f = \{v \mid f(v) = \mathbf{0}\}$.*

Proposition 6 *$\ker f$ and $\operatorname{im} f$ are subspaces of V and W respectively.*

PROOF For $\ker f$, we have:

- $\mathbf{0} = f(\mathbf{0})$, therefore $\mathbf{0} \in \ker f$
- Let $u, v \in \ker f$, $f(u + v) = f(u) + f(v) = \mathbf{0} + \mathbf{0} = \mathbf{0}$, therefore $u + v \in \ker f$
- Let $u \in \ker f$, $f(-u) = -f(u) = \mathbf{0}$, therefore $-u \in \ker f$
- Let $k \in K$ and $u \in \ker f$. $f(k \cdot u) = k \cdot f(u) = \mathbf{0}$, therefore $k \cdot u \in \ker f$.

For $\operatorname{im} f$, we have:

- $\mathbf{0} = f(\mathbf{0})$, therefore $\mathbf{0} \in \operatorname{im} f$
- Let $u, v \in \operatorname{im} f$. By definition of $\operatorname{im} f$ there exists $u', v' \in V$ such that $u = f(u')$ and $v = f(v')$. Thus, $u + v = f(u') + f(v') = f(u' + v') \in \operatorname{im} f$.
- Let $u \in \operatorname{im} f$. By definition, there exists $u' \in V$ such that $u = f(u')$. Thus, $-u = -f(u') = f(-u') \in \operatorname{im} f$.
- Let $k \in K$ and $u \in \operatorname{im} f$. By definition, there exists $u' \in V$ such that $u = f(u')$. Thus, $k \cdot u = k \cdot f(u') = f(k \cdot u') \in \operatorname{im} f$.

$\square$

Definition 12 (Coproduct) *Let X be a set and $V_x (x \in X)$ be a family of vector spaces over a field K. We define $\coprod_{x \in X} V_x$ to be the set of functions $f : X \to \bigcup_{x \in X} V_x$ with finite support, such that $f(x) \in V_x$ for all $x \in X$. We also define addition and scalar multiplication as follows:*

- $(f + g)(x) = f(x) + g(x)$
- $(k \cdot f)(x) = k \cdot f(x)$
- $(-f)(x) = -(f(x))$
- $0(x) = 0$

for all $f, g \in \coprod_{x \in X} V_x$, $x \in X$ and $k \in K$.

If for all $x, y \in X$, $V_x = V_y$, we write $\coprod_X V$ for $\coprod_{x \in X} V_x$. Moreover, let $f \in \coprod_{x \in X} V_x$ and $\operatorname{supp}(f) \subseteq \{x_1, x_2, \ldots, x_n\}$. We define the following notation for f:

$$f = \{x_1 : f(x_1), x_2 : f(x_2), \ldots, x_n : f(x_n)\}$$

Note that, while all elements of $\operatorname{supp}(f)$ must be in $\{x_1, \ldots, x_n\}$, there can exist some $1 \leq i \leq n$ such that $f(x_i) = 0$. As a result, the following are all equivalent representations of the same $f \in \coprod_{\mathbb{N}} \mathbb{R}$:

- $\{1 \mapsto \pi, 2 \mapsto \frac{2}{3}, 4 \mapsto 7\}$
- $\{1 \mapsto \pi, 2 \mapsto \frac{2}{3}, 3 \mapsto 0, 4 \mapsto 7\}$
- $\{1 \mapsto \pi, 2 \mapsto \frac{2}{3}, 4 \mapsto 7, 67 \mapsto 0\}$

If $X = \{1, 2, \ldots, n\}$ for some $n \in \mathbb{N}$, we notate f as follows:

$$f = (f(1), \ldots, f(n))$$

Proposition 7 *$\coprod_{x \in X} V_x$, together with addition ($+$) and scalar multiplication ($\cdot$) as defined previously, is a vector space.*

Definition 13 (Direct sum of vector spaces) *Let V_1, V_2 vector spaces over a field K. Their direct sum $V_1 \oplus V_2$ is defined as follows:*

$$V_1 \oplus V_2 = \coprod_{i \in \{1,2\}} V_i \quad (= V_1 \times V_2)$$

In the more general case, given $V_1, \ldots, V_n$ vector spaces over a field K (for some $n \in \mathbb{N}$), we define the associative operator $\oplus$ as:

$$V_1 \oplus \cdots \oplus V_n = \coprod_{i \in \{1,\ldots,n\}} V_i$$

Which naturally follows from the initial definition of $\oplus$ via isomorphism.

Definition 14 (Vector space exponentiation) *Let V be a vector space over a field K and some $n \in \mathbb{N}$. We define V^n as:*

$$V^n \cong \coprod_{\{1,\ldots,n\}} V = \underbrace{V \oplus \cdots \oplus V}_{n \textit{ times}}$$

3 REA resource accounting

3.1 Resources, ownership states and transfers

Definition 15 (Resources) *Let X be the set of resource types. The set of resources R is the vector space over $\mathbb{R}$ such that $R = \coprod_X \mathbb{R}$ (i.e R is the set of finite maps from resource types to $\mathbb{R}$).*

Definition 16 (Ownership states) *Let A be the set of agents in the system. The set S of ownership states is the vector space such that $S = \coprod_A R$ (i.e S is the set of finite maps from agents to resources).*

At first glance, this might seem like a quite unintuitive way of defining resource ownership since it allows on the one hand for an agent to own *negative* amounts of some resources, and on the other hand it for some indivisible resources to be owned in non-integral quantities (for example, owning $\frac{3}{4}$ or π bicycles does not really make sense in a real world situation). However, the introduction of *ownership state predicates* (see Definition 18) solves the latter issue; and as for the former, allowing for negative quantities lets us define transfers as a subspace of ownership states:

Definition 17 (Transfers) *Let* $\mathrm{sum} : S \to_1 R$ *be the homomorphism such that* $\mathrm{sum}(s) = \sum_{a \in supp(S)} s(a)$ *for all* $s \in S$*. The set* T *of transfers is the subspace of* S *such that* $T = \ker \mathrm{sum}$

Example. The following are all transfers:

- $\{\text{Alice} \mapsto 30000 \cdot \text{COP} - 30 \cdot \text{laptop},\ \text{Bob} \mapsto -30000 \cdot \text{COP} + 30 \cdot \text{laptop}\}$
- $\{\}$
- $\{\text{Alice} \mapsto -10 \cdot \text{ETH},\ \text{Bob} \mapsto -10 \cdot \text{ETH},\ \text{Charlie} \mapsto 20 \cdot \text{ETH}\}$

Proposition 8 (Resource preservation) *For any initial state* s_0*, and* s_n *obtained by applying (i.e adding) any sequence of transfers to* s_0*,* $\mathrm{sum}(s_0) = \mathrm{sum}(s_n)$*.*

PROOF Let $t_1, t_2, \ldots, t_n \in T$ be a sequence of transfers and $s_n = s_0 + t_1 + t_2 + \cdots + t_n$ be the ownership state obtained by applying that sequence to our initial state. We have:

$$\begin{aligned}
\mathrm{sum}(s_n) &= \mathrm{sum}(s_0 + t_1 + t_2 + \ldots t_n) \\
&= \mathrm{sum}(s_0) + \mathrm{sum}(t_0 + t_1 + \ldots t_n) \\
&= \mathrm{sum}(s_0) + \mathbf{0} && \text{(since } T = \ker \mathrm{sum}\text{)} \\
&= \mathrm{sum}(s_0)
\end{aligned}$$

$\square$

Theorem 9 *Let* V, W *be vector spaces and* $f : V \to_1 W$*. Then:*

1. $V \cong \operatorname{im} f \oplus \ker f$
2. $\dim V = \dim(\operatorname{im} f) + \dim(\ker f)$

From that theorem and the definitions of sum, S, A, R and T, it follows directly that:

$$S = \coprod_A R \cong \mathrm{im}\,\mathrm{sum} \oplus \ker \mathrm{sum} = R \oplus T$$

In other words, all ownership states can be represented as a pair of a resource (which we will henceforth refer to as *balance*) and a transfer.

What this means intuitively is that a system of n banks maintaining each an ownership state can be turned into a system of n banks which maintain a transfer each, and a central entity which maintains a vector of n balances.

As an example, let us consider a system with two banks, three actors and one resource type, subdivided into two ownership states (one for each bank):

- $s_1 = \{\mathrm{Bank1} \mapsto 2000 \cdot \mathrm{DKK}, \mathrm{Alice} \mapsto 30 \cdot \mathrm{DKK}, \mathrm{Bob} \mapsto 10 \cdot \mathrm{DKK}\}$
- $s_2 = \{\mathrm{Bank2} \mapsto 1500 \cdot \mathrm{DKK}, \mathrm{Thomas} \mapsto 600 \cdot \mathrm{DKK}, \mathrm{Bob} \mapsto 3 \cdot \mathrm{DKK}\}$

As a first step, we can take each ownership state and represent it as an element of $R \oplus T$ (as per Theorem 9):

- $s'_1 = (2040 \cdot \mathrm{DKK}, \{\mathrm{Bank1} \mapsto -40 \cdot \mathrm{DKK}, \mathrm{Alice} \mapsto 30 \cdot \mathrm{DKK}, \mathrm{Bob} \mapsto 10 \cdot \mathrm{DKK}\})$
- $s'_2 = (2103 \cdot \mathrm{DKK}, \{\mathrm{Bank2} \mapsto -603 \cdot \mathrm{DKK}, \mathrm{Thomas} \mapsto 600 \cdot \mathrm{DKK}, \mathrm{Bob} \mapsto 3 \cdot \mathrm{DKK}\})$

3.2 Ownership state predicates and transition functions

Definition 18 (Ownership state predicate) *An ownership state predicate is a boolean function $p : S \to \{0, 1\}$ over ownership states.*

Intuitively, such a function classifies ownership states between valid ones ($p(s) = 1$) and invalid ones ($p(s) = 0$). We want to prevent most agents from spending resources they do not have, and therefore want their balances to always be positive. The reason we allow some agents to have negative balances is that we want to account for agents that are able to *create* resources, such as central banks for money, farmers for agricultural produce, etc. Such predicates also allow us to constrain the ownership quantities of some resources to a certain range within the real numbers (as we alluded to after defining ownership states).

Note that a *credit limit policy*, as we mentioned previously, is a particular type of predicate that sets up a bound on resource quantities for each agent.

Definition 19 (Transition function) *The transition function of a predicate $p \in \{0,1\}^S$ is the function $\delta_p : T \to (S \to S_\perp)$ such that:*

$$(\delta_p(t))(s) = \begin{cases} \perp & \textit{if } p(s+t) = 0 \\ s+t & \textit{otherwise} \end{cases}$$

with $S_\perp = S \cup \{\perp\}$

Definition 20 (Kleisli composition) *Let A, B and C be three sets such that $\perp \notin B$, $\perp \notin C$, and $f : A \to B_\perp$, $g : B \to C_\perp$ two functions. The Kleisli composition of f and g is the function $f \triangleright g : A \to C_\perp$ such that:*

$$(f \triangleright g)(x) = \begin{cases} \perp & \textit{if } f(x) = \perp \\ g(f(x)) & \textit{otherwise} \end{cases}$$

Definition 21 (Monoid) *A monoid is a tuple $(M, \bullet)$ where M is a set and $\bullet : M \times M \to M$ is a binary operator such that:*

1. *for all $a, b, c \in M$, $a \bullet (b \bullet c) = (a \bullet b) \bullet c$*
2. *there is an $x \in M$ such that for all $a \in M$, $a \bullet x = x \bullet a = a$.*

Proposition 10 *Let Δ_S be the set of all functions from S to $S_\perp$. $(\Delta_S, \triangleright)$ is a monoid.*

PROOF

1. Let $f, g, h \in \Delta_S$. By definition, we have:

$$((f \triangleright g) \triangleright h)(x) = \begin{cases} \perp & \text{if } (f \triangleright g)(x) = \perp \\ h((f \triangleright g)(x)) & \text{otherwise} \end{cases}$$

and

$$(f \triangleright g)(x) = \begin{cases} \perp & \text{if } f(x) = \perp \\ g(f(x)) & \text{otherwise} \end{cases}$$

thus, we have:

$$((f \triangleright g) \triangleright h)(x) = \begin{cases} \perp & \text{if } f(x) = \perp \text{ or } g(x) = \perp \\ h(g(f(x))) & \text{otherwise} \end{cases}$$

.

On the other hand, we have:

$$(f \triangleright (g \triangleright h))(x) = \begin{cases} \perp & \text{if } f(x) = \perp \\ (g \triangleright h)(f(x)) & \text{otherwise} \end{cases}$$

and

$$(g \triangleright h)(x) = \begin{cases} \perp & \text{if } g(x) = \perp \\ h(g(x)) & \text{otherwise} \end{cases}$$

thus, we have:

$$(f \triangleright (g \triangleright h))(x) = \begin{cases} \perp & \text{if } f(x) = \perp \text{ or } g(x) = \perp \\ h(g(f(x))) & \text{otherwise} \end{cases}$$

.

Therefore, $f \triangleright (g \triangleright h) = (f \triangleright g) \triangleright h$.

2. Let $\mathrm{id}_\triangleright \in \Delta_S$ such that $\mathrm{id}_\triangleright(x) = x$ and $f \in \Delta_p$. We have:

$$(f \triangleright \mathrm{id}_\triangleright)(x) = \begin{cases} \perp & \text{if } f(x) = \perp \\ \mathrm{id}_\triangleright(f(x)) = f(x) & \text{otherwise} \end{cases}$$

And thus $f \triangleright \mathrm{id}_\triangleright = f$.

We also have:

$$(\mathrm{id}_\triangleright \triangleright f)(x) = \begin{cases} \perp & \text{if } \mathrm{id}_\triangleright(x) = \perp \\ f(\mathrm{id}_\triangleright(x)) = f(x) & \text{otherwise} \end{cases}$$

And since $\perp$ is not in the image of $\mathrm{id}_\triangleright$, we have $(\mathrm{id}_\triangleright \triangleright f) = f$.

□

Let $s \in S$ an ownership state and $p \in \{0,1\}^S$ an ownership state predicate. We define the ownership state $s' \in S_\perp$ obtained by applying a sequence $t_1, \ldots, t_n$ to s via δ_p as follows:

$$s' = (\delta_p(t_1) \triangleright \delta_p(t_2) \triangleright \ldots \triangleright \delta_p(t_n))(s)$$

Proposition 11 *Let $s \in S$ an ownership state and $p \in \{0,1\}^S$ an ownership state predicate, $Q = \{t_1, t_2, \ldots, t_n\}$ a finite multiset of size n such that $t_i \in T$ for all $1 \leq i \leq n$, and q_1, q_2 two sequences of the elements of Q. If both q_1 and q_2 can be applied to s via δ_p to obtain respectively $s', s'' \in S$, then $s' = s''$.*

PROOF Let $q = t_1, \ldots, t_2$ and $q' = t'_1, \ldots, t'_2$ two sequences of the elements of a multiset of elements of T, $s \in S$ and $p \in \{0,1\}^O$, such that both q_1 and q_2 can be applied to s via δ_p to obtain respectively $s_n, s'_n \in O$.

Since $s_n, s'_n \in S$, $o_n \neq \perp$ and $o'_n \neq \perp$, then, by definition of the transition function δ_p, we have:

- $s_n = s + \sum_{i=1}^{n} t_i$
- $s'_n = s + \sum_{i=1}^{n} t'_i$

And since q_1 and q_2 are two sequences that contain exactly the same elements (seeing as they were drawn from the same multiset) and by commutativity of vector addition, it follows that $s_n = s'_n$. □

Note that a permutation of a successful sequence of transactions is not necessarily successful. For example, if we have $s_0 = \{a_1 : 30x, a_2 : \mathbf{0}, a_3 : \mathbf{0}\} \in S$, $t_1 = \{a : -30x, b : 30x\} \in T$ and $t_2 = \{b : -30x, c : 30x\} \in T$, then the sequence t_1, t_2 applied to s_0 will be successful whereas t_2, t_1 will not. However, we know now that all successful orders will result in the same final state.

3.3 Example: the Ethereum Ledger

Using the structures we have defined so far, here is how we can formulate a model of the Ethereum network[11] using the following:

- $X = \{\text{ETH}\}$
- $A = \{s \mid s \text{ is a valid Ethereum address }\}$
- $p(s) = \begin{cases} 0 & \text{if there exists an } a \in A, n \in \mathbb{N} \text{ such that } s(a) < 0 \ \wedge \ a = n \cdot 10^{-18} \\ 1 & \text{otherwise} \end{cases}$

Here, the predicate p limits all amounts to the smallest possible subdivision of Ether (the Wei, with 1 ETH = 10^{18} Wei), and requires all account balances to be positive.

However, the capabilities of an implementation of that model using transfers as defined here would actually be a strict *superset* of Ethereum, since it can handle

n-party transactions (for all $n \in \mathbb{N}$). Let us for example consider a situation where we have three agents, who we will call Alice, Bob and Charles, own 10 Ether each and owe each other Ether with the following configuration:

- Alice owes 30 Ether to Bob
- Bob owes 40 Ether to Charles
- Charles owes 50 Ether to Alice

With a system that supports 2-party transactions only, the transfers needed to settle those debts would be the following:

- $t_1 = \{\text{Alice} \mapsto -30 \cdot \text{ETH}, \text{Bob} \mapsto 30 \cdot \text{ETH}\}$
- $t_2 = \{\text{Bob} \mapsto -40 \cdot \text{ETH}, \text{Charles} \mapsto 40 \cdot \text{ETH}\}$
- $t_3 = \{\text{Charles} \mapsto -50 \cdot \text{ETH}, \text{Alice} \mapsto 50 \cdot \text{ETH}\}$

However, there is no sequence of t_1, t_2 and t_3 that can be successfully be applied to the current state of the system without violating p, even though the netting of those transactions,

$$t = t_1 + t_2 + t_3 = \{\text{Alice} \mapsto 20 \cdot \text{ETH}, \text{Bob} \mapsto -10 \cdot \text{ETH}, \text{Charles} \mapsto -10 \cdot \text{ETH}\}$$

,

can be applied successfully to the current state of the system via δ_p. A state manager that implements our algebraic framework could implement an escrow system via a smart contract that waits for both Bob and Charles to pay it 10 ETH before transferring those 20 ETH to Alice, and would reimburse either Bob of Charles if one of the two failed to pay within a certain timeframe. In a more direct fashion, these three agents could also agree beforehand on this more complex, netted transaction, and submit it together to the system.

The fact that our system can now handle more complex transactions enables an implementation that can effect multi-party state changes *atomically* without the need for a smart contract platform, as is necessary in current blockchain implementations such as Ethereum and its predecessor, Bitcoin. It also allows for a more direct formulation of the *netting problem*, as we will cover in the following section.

4 The netting problem

4.1 Initial formulation

The example we just saw is an instance of the *netting problem* as described in previous literature, wherein a group of transactions that cannot be executed individually are grouped, i.e added, to form a single transaction with the same net effect on the ownership state that can be executed. In order to do that, the state manager maintains a set of all pending transactions and runs a netting algorithm, either periodically or every time the pending transaction set reaches a pre-defined size, whose goal is to find the biggest subset of that transaction set that can be executed.

Unfortunately, this initial formulation of the netting problem is NP-complete, which prompts the need for a more restricted definition of the problem, where we define an ordering on pending transactions, which then constrains the set of possible solutions to the problem.

On the one hand, the case where we have a total order on transactions is rather easy to solve in polynomial time. Let P be a totally ordered set of priorities (e.g timestamps). A centralized resource manager can then solve this restricted definition of the netting problem by maintaining an internal priority-ordered queue of pending transfers and running a simple algorithm that operates as following on reception of a prioritized transfer $(t, p) \in T \times P$:

- If t can be immediately applied to the current state s of the system without violating the ownership state predicate, the state becomes $s' = s + t$. Otherwise, add (t, p) at its corresponding spot to the pending transfer queue.
- Either periodically, or whenever the queue reaches a certain size, and assuming the queue is ordered in decreasing order of priority, find the longest prefix of the queue whose sum can be applied to s without violating the ownership state predicate. Then, remove all elements in that prefix from the queue, obtain s' by adding their sum to s.
- If the prefix we calculated previously is empty, we have reached a *deadlock*, and thus reject all the pending transfers in the queue and empty it.

On the other hand, in practical applications we can only have a partial order on transaction priorities. Let us consider the case where we have a set of actors A, and choose to represent priorities with natural numbers ($i.eP = \mathbb{N}$). We can now visualize the state of the system as a multiset of queues, with a queue $\mathcal{Q}_a$ of elements of $T \times \mathbb{N}$ for each $a \in A$ such that

$$\mathcal{Q}_a = [(t_1, p_1), \ldots, (t_n, p_n)]$$

for some $n \in \mathbb{N}$ and such that $p_i \geq p_{i+1}$ for all $i \in \mathbb{N}$, $1 \leq i < n$. The solution to the problem then becomes the union of the longest prefixes of each $\mathcal{Q}_a$ such that the liquidity constraints of the system hold.

Cao at al.[4] introduce an algorithm for decentralized, privacy-preserving netting in a system with a partial order on transaction such as the one we have just described, but where only two way transactions over a single resource type are allowed. Our goal here is to first describe that algorithm in terms of our resource accounting framework, and then see how an algorithm can be devised that can handle the more general case of n-way transactions ($n \in \mathbb{N}$) over arbitrary resources (i.e the set of transfers T).

Note that we will not describe nor extend the privacy-preserving extension of the algorithm described by Cao et al., since the focus of this document is first and foremost to present a mathematical description of a transaction model rather than develop a concrete production-ready decentralized implementation that would then have to deal with confidentiality issues. Note also that for the following sections, we exclusively allow ownership state predicates which set a lower bound on each actor's balance, since this is an important condition for the algorithm to converge/terminate, as we will see.

4.2 Two-way, single resource transfers

Let us define a restricted version of our REA accounting model, where:

- There is only a single resource, i.e. $X = \{1\}$
- Transfers are only allowed to be two-party transfers, i.e $T_2 = \{t \in T : \mathrm{card}(\mathrm{supp}(t)) = 2\}$
- The ownership state predicate that conditions state transactions is a credit limit policy, that is, it is true only if all actor balances are positive.
- P is the set of all pending transactions at the time the algorithm is run.

Let $s \in S$ the current state of a system with one central resource manager and a set of actors that communicate with it in order to settle their payments. The algorithm described by Cao et al., then proceeds in two phases:

1. Each actor a maintains a local list $\mathcal{Q}_a$ of size s_a of all pending transactions in which they are to spend a resource (i.e for all $(t,p) \in \mathcal{Q}_a$, $t(a) < 0$), sorted in descending priority order (i.e with the highest priority transactions

first). They also maintain a *predicted incoming balance* B_a, which in the first iteration is defined as the sum of all incoming amounts to that actor, i.e $\sum_{t \in N, t(a)>0} t(a)$.

The actor then finds the longest prefix $t_1^a, \ldots, t_k^a$ of Q_a for some $k \in \mathbb{N}, k \leq s_a$ such that $B'_a = s(a) + B_a + t_1^a(a) + \cdots + t_k^a(a) \geq 0$. Finally, it sends its prefix to the central resource manager, which takes care of the second phase of the algorithm.

2. Now, the resource manager calculates *global nettable set of transactions* N, which is the union of all proposals received from the individual actors. If the set at this round is the same as the one at the previous round, the algorithm terminates: we have found the optimal nettable set. If the set is empty, we also terminate, since we have reached a deadlock situation, in which case we reject all pending transactions. Otherwise, it calculates a new incoming balance B'_a for each actor such that for each a, $B'_a = \sum_{t \in N, t(a)>0} t(a)$, sends it to each respective actor, and we go back to phase 1.

Note that transaction priorities are defined on a *per actor* basis, that is, we do not have a total but rather a partial order over transactions with respect to priority.

Theorem 12 *This algorithm always finds a solution which is both unique and optimal.*

PROOF Let N_n denote the value of N at the n-th iteration of the algorithm. Since at each iteration, we remove the transactions that violate the credit limit from N, we have that, for all n:

$$N_{n+1} \subseteq N_n \tag{1}$$
$$\Longrightarrow \mathrm{card}(N_{n+1}) \leq \mathrm{card}(N_n) \tag{2}$$

and therefore, the algorithm always converges.

Let us consider the last iteration n where the algorithm converges, that is, the point at which we have $N_n = N_{n-1}$. Since the two sets are equal, we have, for all $a \in A$

$$B_{a,n} = B_{a,n-1}, \tag{3}$$

and therefore, since the prefixes computed for each actor at iteration n all respect the liquidity constraint for $B_{a,n-1}$, they also respect the liquidity constraint for $B_{a,n}$. Thus, the solution we have found is feasible.

Since N is maximal at each iteration n given $B_{a,n}$ for each $a \in A$, and since equation (1) holds, the first feasible solution we encounter, i.e the one the algorithm terminates on, is also optimal. □

4.3 Generalization to multi-resource, multi-party transfers

In order to generalize this algorithm to multi-resource, multi-party transfers, we use the following theorem:

Theorem 13 *Let $D \subseteq T$ be the set of two-party, single resource transfers, that is, the set of transfers such that for all $d \in D$, there exists $a, b \in A$, $x \in X$ such that:*

1. $\mathrm{supp}(d) = \{a, b\}$

2. $\mathrm{supp}(d(a)) = \mathrm{supp}(d(b)) = \{x\}$

Then D spans T.

PROOF Let $t \in T$. We now devise an algorithm that operates as follows:

1. We initially set n to 1. Let $s_0 = t$.
2. While $\mathrm{supp}(s_{n-1}) \neq \varnothing$:
   (a) Pick $a_n, b_n \in \mathrm{supp}(s_{n-1})$
   (b) Let $t_n = \{a_n \mapsto s_{n-1}(a_n), b_n \mapsto -s_{n-1}(a_n)\} \in T$
   (c) Let $s_n = s_{n-1} - t_n$
   (d) Increment n by 1
3. Terminate

We first prove that step (2a) is always feasible by proving that for all $s_n \in T$, $\mathrm{card}(\mathrm{supp}(s_n)) \neq 1$. Let us assume that there exists an $a^* \in A$ such that $\mathrm{supp}(s_n) = \{a^*\}$. Since $s_n \in T$, we have that

$$\begin{aligned} &\sum_{a \in A} s_n(a) = 0 \\ \implies &s_n(a^*) + \sum_{a \in A \setminus \{a\}} s_n(a) = 0 \\ \implies &s_n(a^*) = 0 \end{aligned}$$

which leads to a contradiction since $a^* \in \mathrm{supp}(s_n)$. Therefore, $\mathrm{card}(\mathrm{supp}(s_n)) \neq 1$.

Since at each value of n, $a_n \in \mathrm{supp}(s_{n-1})$ and $s_n = s_{n-1} - tn$ by step (2a), we have that $s_n(a_n) = s_{n-1}(a) - s_{n-1}(a_n) = 0$, and thus $a \notin \mathrm{supp}(s_n)$, which means that $\mathrm{supp}(s_n) < \mathrm{supp}(s_{n-1})$. Moreover, since $s_n \in T$, its support is finite. Therefore, this algorithm always terminates.

Let $k \in \mathbb{N}$ be the value of n when we enter the last iteration. By step (2c), we have:

$$s_k = s_0 - \sum_{i=1}^{k} t_i$$

And since we terminate right after that iteration, by the exit condition we have that $s_f = 0$. Thus:

$$s_0 = \sum_{n=1}^{k} t_n$$

Therefore, since t_n is a two-party transfer for all $1 \leq n \leq k$, we have found a linear decomposition of t over two-party transfers. Decomposing each of those two-party transfers into a sum of single-resource two-party transfers is then quite simple, since we have:

$$\begin{aligned} & \sum_{a \in A} t_n(a) = 0 \\ \implies & \forall x \in X, \sum_{a \in A} t_n(a)(x) = 0 \end{aligned}$$

Which means that we can decompose each t_n as follows:

$$\begin{aligned} t_n &= \sum_{x \in X} t_n^x \\ &\stackrel{\text{def}}{=} \sum_{x \in X} \{a \mapsto \{x \mapsto t_n(a)(x)\}\}_{a \in A} \end{aligned}$$

with each $t_n^x \in D$ being a single resource two-party transfer. Since $\mathrm{supp}(t_n)$ is finite, and for all $a \in A$, $\mathrm{supp}(t_n(a))$ is also finite, this sum is a finite decomposition of t_n. $\square$

Knowing this, we modify the netting algorithm so that it can handle multi-party, multi-resource transfers in an **atomic** fashion. Let I be the set of *transaction* *identifiers* such that each pending transaction t has a distinct identifier $i(t) \in I$. We then assign to each transaction identifier i a priority $p_a(i) \in \mathbb{N}$ for each actor a. Our modified algorithm then proceeds as follows, with $s \in S$ being the current state of the system:

1. Decompose each pending transaction into a set $N' \subseteq D \times I$ of two-party, single resource transfers, tagged with an id corresponding to the original pending transaction it came from as well as that transaction's priority. From that set, we construct a family of subsets $N_i^a (i \in I, a \in A)$ such that, for all $i \in I$:

$$N_i^a = \{d : ((d, i) \in N) \wedge (a \in \mathrm{supp}(d)) \wedge (\forall x \in X,\ d(a)(x) < 0)\}$$

   that is, we group the transfers in N by original transaction identifier and spender. Finally, for each $a \in A$, $i \in I$, we define $t_i^a \in T$ such that:

$$t_i^a = \sum_{d \in N_i^a} d$$

   We now have for each transaction i and spender a what can be conceived of as a's payout for i, t_i^a.

2. Each actor now constructs a list $\mathcal{Q}_a$ of all t_i^a ordered in decreasing transaction priority as well as a set $I_r \subseteq I$ of all rejected transactions identifiers, which is initially empty, and a predicted incoming balance B_a which is initially defined as the sum of all incoming amounts to that actor. It then selects the longest prefix $t_{i_1}^a, \ldots, t_{i_k}^a$ of $\mathcal{Q}_a$, of length $k \in \mathbb{N}$, such that the following two constraints hold:

   (a) For all $x \in X$, $s(a)(x) + B_a(x) + (\sum_{j=1}^{k} t_{i_j})(x) \geq 0$

   (b) For all $1 \leq j \leq k$, $i_j \notin I_r$

   It then updates I_r by adding to it all the identifiers of the transfers in $\mathcal{Q}_a$ that did not end up in the prefix, and sends the prefix along with its updated I_r to the central resource manager.

3. The resource manager then calculates the global nettable set of transactions by calculating the union of all received prefixes, and the global rejected transaction set by calculating the union of all received I_r. If both of those sets have not changed since the last iteration, the algorithm has converged

and it terminates. Otherwise, the resource manager calculates the new predicted incoming balance B_a of each actor and sends it to each actor as well as the global rejected transaction set, and we go back to step 2.

This algorithm has in common with the original one that the size of the queue of each actor decreases at each iteration, and it also preserves the optimality property of the provisional solution at each round. It can then be proven analogously that it always converges towards the optimal solution.

Note that the fact that the decomposition of a transfer into a sum of two-party single-resource transfers is not unique does not have any impact on the netting algorithm: our use of transaction identifiers ensure that transactions are committed to the state in an atomic fashion, which means that no matter the decomposition we end up with, the only determining factor of whether a transaction is committed or not is whether the sum of the elements of its decomposition satisfies the credit limit, which does not depend on the particular decomposition we pick.

A Haskell implementation of this algorithm is provided in Appendix A (see the `net` function specifically).

5 Events and transformations

In its current form, the RAE accounting framework can handle arbitrary resource transfers, but still falls short if one tries to use it to describe a more complex accounting situation, such as a supply chain for example, because it is not able to handle *transformations* across resource types, which is necessary in order to be able to model full industrial or agricultural processes.

A naive approach when trying to add that capability to our model would be to model transformations as elements of S. That is, given some initial state $s \in S$, we could model a transformation as a vector that subtracts the used resources from the state and adds to it the newly created resources corresponding to the transformation we are modelling. One could imagine for example a simple transformation that could happen at a bar, where s and s' describe respectively the state of the system before and after the transformation takes place:

$$s' = s + \{\text{Bar} \mapsto -5 \cdot \text{cl-gin} - 20 \cdot \text{cl-tonic} + 25 \cdot \text{cl-gin-and-tonic}\}$$

However, this approach fails to preserve some fundamental invariants we expect from real world resource transformation systems. Ideally, we would want our formulation of the system on inherently prevent invalid transformations to occur. Let us consider the example of a system whose initial state is the following:

$$s = \{a \mapsto 300 \cdot x_1\}$$

Let us also consider the following two transformations:

- $e_1 = \{a \mapsto -300 \cdot x_1 + 200 \cdot x_2\}$
- $e_2 = \{a \mapsto 400 \cdot x_1 - 200 \cdot x_2\}$

It is clear that in a well formed formulation of the framework, both those transformations should not be simultaneously valid, since applying both to the initial state of the systems results in the following state s':

$$\begin{aligned} s' &= s + e_1 + e_2 \\ &= \{a_1 \mapsto 400 \cdot x_1\}, \end{aligned}$$

which should be invalid since we have visibly been able to indirectly turn 300 x_1 into 400.

Our goal in formalizing resource transformations is to introduce a device analogous to our definition of transfers in the original formulation of the framework, so that this type of resource manipulations becomes impossible by construction.

5.1 Formal definitions

Definition 22 (Valuation map) *Let V be a vector space over $\mathbb{R}$ and let $w : R \to_1 V$ be a homomorphism from R to V. We call w a valuation map over V.*

Definition 23 (Events) *Let $w : R \to_1 \mathbb{R}$ be a valuation map over some vector space V. The set E_w of valid events with respect to w is defined as*

$$E_w = \{s \in S : \sum_{a \in A} w(s(a)) = 0\}$$

Proposition 14 *Let $w : R \to_1 V$ be a valuation map over some vector space V. Then E_w, together with addition and scalar multiplication, is a subspace of S.*

PROOF Since w is a homomorphism, we have

$$\sum_{a \in A} w(0(a)) = \sum_{a \in A} w(0) = 0$$

And therefore, $0 \in E_w$. Moreover, for all $u, v \in E_w$:

$$
\begin{aligned}
& \sum_{a\in A} w(u(a)) = 0 \wedge \sum_{a\in A} w(v(a)) = 0 \\
\implies & \sum_{a\in A} w(u(a)) + w(v(a)) = 0 \\
\implies & \sum_{a\in A} w(u(a) + v(a)) = 0 \\
\implies & \sum_{a\in A} w((u+v)(a)) = 0 \\
\implies & u + v \in E_w
\end{aligned}
$$

And, finally, for all $u \in E_w$, $k \in \mathbb{R}$:

$$
\begin{aligned}
& \sum_{a\in A} w(u(a)) = 0 \implies \qquad\qquad k \cdot \sum_{a\in A} w(u(a)) = 0 \\
\implies & k \sum_{a\in A} k \cdot w(u(a)) = 0 \\
\implies & k \cdot \sum_{a\in A} w(k \cdot u(a)) = 0 \\
\implies & k \cdot u \in E_w
\end{aligned}
$$

□

Proposition 15 *T is a subspace of E_w.*

PROOF Let $t \in T$ and $w : R \to_1 V$ for some vector space V. Since $T = \ker \mathrm{sum}$:

$$
\begin{aligned}
& \mathrm{sum}(t) = 0 \\
\implies & \sum_{a\in A} t(a) = 0 \\
\implies & \sum_{a\in A} w(t(a)) = 0 \\
\implies & t \in P_w
\end{aligned}
$$

And since T is a vector space, we have:

- $0 \in T$

- For all $u, v \in T$, $u + v \in T$
- For all $k \in \mathbb{R}$, $u \in T$, $k \cdot u \in T$

$\square$

Note that when $w = \mathrm{id}_R$, $E_w = T$.

Definition 24 (Transformations) *Let $w : R \to_1 V$ be a valuation map over some vector space V. The set P_w of transformations with respect to w is defined as*

$$P_w = \{p \in S : \forall a \in A,\ w(p(a)) = 0\}$$

Proposition 16 *P_w is a subspace of E_w.*

PROOF Let $w : R \to_1 V$ for some vector space V and $p \in P_w$. Then we have:

$$\begin{aligned} & \forall a \in A,\ w(p(a)) = 0 \\ \implies & \sum_{a \in A} w(p(a)) = 0 \\ \implies & p \in E_w \end{aligned}$$

Let $u, v \in P_w$ and $k \in \mathbb{R}$, then for all $a \in A$:

$$\begin{aligned} & \begin{cases} w(u(a)) = 0 \\ w(v(a)) = 0 \end{cases} \\ \implies & w(u(a)) + w(v(a)) = 0 \\ \implies & w((u + v)(a)) = 0 \\ \implies & u + v \in P_w \end{aligned}$$

and

$$\begin{aligned} & w(u(a)) = 0 \\ \implies & k \cdot w(u(a)) = 0 \\ \implies & w(k \cdot u(a)) = 0 \\ \implies & k \cdot u \in P_w \end{aligned}$$

$\square$

Proposition 17 *For all $e \in E_w$, there is a $t \in T$, $p \in P_w$ such that $e = t + p$.*

PROOF We define an algorithm, similar to the one in the proof for Theorem 13, that operates as follows:

1. We initially set n to 1. Let $s_0 = e$
2. While $\mathrm{card}(\mathrm{supp}(s_{n-1})) \geq 2$:
   (a) Pick $a_n, b_n \in \mathrm{supp}(s_{n-1})$
   (b) Let $t_n = \{a_n \mapsto s_{n-1}(a_n), b_n \mapsto -s_{n-1}(a_n)\} \in T$
   (c) Let $s_n = s_{n-1} - t_n$
   (d) Increment n by 1
3. Terminate

By steps (2b) and (2c), we have that, for all values n takes:

$$\mathrm{supp}(s_n) \subseteq \mathrm{supp}(s_{n-1}) \setminus \{a_n\}$$

which means that

$$\mathrm{card}(\mathrm{supp}(s_n)) < \mathrm{card}(\mathrm{supp}(s_{n-1}))$$

and therefore, the algorithm terminates.

Let $k \in \mathbb{N}$ be the value of n when we enter the last iteration of the algorithm. By step (2c), we have that

$$s_k = s_0 - \sum_{i=1}^{k} t_i$$
$$\implies s_0 = s_k + \sum_{i=1}^{k} t_i$$

Since $t_i \in T$ for all $1 \leq i \leq k$, and T is closed under addition, $\sum_{i=1}^{k} t_i \in T$, and by the exit condition, we have that $\mathrm{card}(\mathrm{supp}(s_k)) \leq 1$. Which leads us to the following two cases:

1. If $\mathrm{supp}(s_k) = \varnothing$, then $s_k = 0$.

2. If $\mathrm{supp}(s_k) = \{a^*\}$ for some $a^* \in A$, then we have that, since $s_k \in E_w$:

$$\begin{aligned}
&\sum_{a \in A} w(s_k(a)) = 0 \\
\implies & w(s_k(a*)) + \sum_{a \in A \setminus \{a^*\}} w(s_k(a)) = 0 \\
\implies & w(s_k(a*)) + \sum_{a \in A \setminus \mathrm{supp}(s_k)} w(s_k(a)) = 0 \\
\implies & w(s_k(a*)) + 0 = 0 \\
\implies & w(s_k(a*)) = 0
\end{aligned}$$

And therefore, for all $a \in A$, $w(e(a)) = 0$, which means that $s_k \in P_w$. $\square$

Such a decomposition allows us to reformulate the multi-party, multi-resource netting algorithm so that it can handle a set of pending transactions in P_w by defining $(N_i^a)(i \in I, a \in A)$ as follows:

$$N_i^a = \{d : ((d, i) \in N) \wedge (a \in \mathrm{supp}(d)) \wedge (\mathrm{supp}(d) = \{a\} \vee \forall x \in X, d(a)(x) < 0)\}$$

with N being the union of the set of single resource transfers and the set of production steps obtained from the decomposition of the set of pending transactions, in a manner analogous to what we did before transformations were introduced.

Note that even though the decomposition produced by this algorithm is valid, it is not unique, since the way we pick a_n and b_n at each round changes the final result. For example, les us consider the following event:

$$e = \{a \mapsto -3 \cdot x, b \mapsto y + 2 \cdot z\}$$

It can be decomposed in the following two ways

$$e = \begin{cases} \{a \mapsto -3x, b \mapsto 3x\} + \{b \mapsto -3x + y + 2z\} \\ \{a \mapsto -y - 2z, b \mapsto y + 2x\} + \{a \mapsto -3x + y + 2z\} \end{cases}$$

5.2 Event predicates

Definition 25 (Event predicate) *A transformation predicate is a boolean function $E_w \to \{0, 1\}$ on events.*

Definition 26 (Event transition function) *The event transition function of an ownership state predicate $p : S \to \{0, 1\}$ and an event predicate $q : E \to \{0, 1\}$ is the function $\tau_p^q : E \to (S \to S_\perp)$ such that:*

$$(\tau_p^q(e))(s) = \begin{cases} \perp & \textit{if } p(s+e) = 0 \textit{ or } q(e) = 0 \\ s + t & \textit{otherwise} \end{cases}$$

With $S_\perp = S \cup \{\perp\}$.

Note that the set of transition functions defined in the initial formulation of our model is a subset of the family of functions we just defined. Given $p : S \to \{0, 1\}$ and $q : S \to \{0, 1\}$ such that $q(e) = 1$ for all $e \in E$, we have:

$$\delta_p = \tau_p^q$$

Note that such transition functions can also be Kleisli composed using the $\triangleright$ operator as defined previously.

6 Application to the coffee supply chain

In this section, we will introduce the reader to the basics of coffee production as well as provide them with a basic overview of the coffee sector in Colombia. We will then analyze the smart contract developed as part of the COWI coffee supply chain project [5] for modelling the transactions that take place within the Colombian coffee supply chain. We will go through the interface and internals of the contract, and see how our algebraic resource accounting model compares to that prototype in describing the interactions within the coffee supply chain.

6.1 Basic terminology

Coffee cherries are the fundamental raw material of coffee, and are the berries of certain species of *Coffea*, most notoriously *Coffea arabica* (usually referred to as "arabica") and *Coffea canephora* (usually referred to as "robusta"). Most of the coffee grown in Colombia is of the arabica variety, with some of the commonly grown subspecies being *Típica*, *Caturra* and *Castillo*, among others [9].

Wet parchment is the product obtained by the process of wet milling coffee cherries (sp. *despulpado*), that is, separating the seed/bean from the pulp. The seeds, which contain a fair amount of moisture after this process, are then dried and fermented in order to obtain what we call *dry parchment*.

Green coffee is the product obtained by *dry milling* dry parchment coffee (sp. *trilla*). This process consists in removing the outer layer from the beans, as well as sorting them into two categories: *excelso* green coffee, which consists of the heaviest, defect-less beans, and *pasilla*, which consists of all the lower quality beans that did not make it to the other category.

Roasted coffee is the final product and the result of the supply chain. It is obtained by *roasting* (sp. *tostado*) green coffee.

6.2 The Colombian coffee supply chain

The coffee industry in Colombia works in a way that is quite different from other countries, in that since 1927, the National Federation of Coffee Growers of Colombia (*Federación Nacional de Cafeteros de Colombia* in Spanish, often abbreviated as Fedecafé) controls most of the market. It consists of a network of smaller, local cooperatives which buy dry parchment directly from the farmers (who usually carry out the wet milling process themselves), the majority of which operate on a very small scale, with an average farm size of 6.4 hectares (or about 16 acres), out of which on average 1.5 hectares are used for coffee production (or about 3.7 acres)[8].

The dry parchment purchase usually happens at purchasing points (sp. *puntos de compra*) operated by the cooperatives at a price fixed daily by the Federation, which is contingent on the quality of the beans, expressed using a metric called the yield factor (sp. *factor de rendimiento*), which correspond to the weight of dry parchment (in kilograms) necessary to produce 70 kilograms of *excelso* green coffee. The daily reference price of dry parchment is set for a yield factor of

The dry parchment is then usually sent from each purchasing point to a dry milling facility controlled by Almacafé, the logistics arm of the Federation, which then takes care of selling and shipping the resulting green coffee to both Colombian and foreign roasters, which are in charge of roasting and packaging the final product.

6.3 The `Coffeebrain` smart contract

The `coffeebrain-ethereum` project[6] is a prototype for a smart contract that handles resource exchange and transformations along the coffee supply chain as it operates in Colombia. Together with a client that is capable of interpreting the events it emits, it provides its users, with a way of tracing the steps undertaken by the actors in the chain in order to produce any given product, no matter at which stage of production it is (dry parchment, green coffee, etc.).

The model for transformations relies on the fundamental concept of *silos*, which are each represented by a FIFO data structure in which products can be stored. First, the contract implements a `mint` primitive which allows some actors (here, only the ones registered as coffee growers) to emit new products "out of thin air" (from the perspective of the contract), so that we can represent harvests. The contract then implements the following two operations on such silo data structures:

1. `store`, which inserts a product into a silo.
2. `transform`, which creates a new product from the contents of a silo, by dequeuing a certain weight from that silo.

Note that this silo model preserves a weight invariant: the sum of the weights of the inputs of a transformation always equals the sum of the weights of its outputs. In order to be able to model processes that incur weight loss, we introduced an explicit weight loss product type, hence the need for a `burn` operation, as defined in the contract.

From those four primitives, the contract then allows the user to register all kinds of physical transformations on any set of products. Here are some examples of transformation events that happen in the supply chain:

- *Bundling*: Given several products of the same type (dry parchment, green, etc.), we can merge them into a single output product by storing them in a silo and then performing a transformation on that same silo such that the output product's weight equals the sum of the weights of the input products, and such that the output product has the same type as the inputs.
- *Splitting*: Given a product, we can split it into several smaller products by storing it in a silo and then performing one transformation on that silo per output product, such that the sum of the weights of the outputs equal the weight of the input, and such that all output products have the same type as the input product.
- *Transform*: Given a set of products we want to transform, we store them in a silo and then produce a product out of it, which we then split in two in order to obtain both the actual result of the transformation, and a weight loss product, which we discard. Only the following transformations are actually valid:
  - From cherries to wet parchment
  - From wet parchment to dry parchment
  - From dry parchment to green coffee
  - From green coffee to roasted coffee

The contract also implements interface functions for the transfer of resources in the system, such that a resource transfer must be validated by both parties involved: the `ship` method initiates a resource transfer from an actor to another, and the `confirmReceipt` method concludes it.

The contract also implements other operations for the handling of certifications, purchase contracts, etc., but here we chose to only focus on the transfers and transformations of products.

6.4 An algebraic description of the coffee supply chain

Now that we have gained insight into the concrete actors, resources and transformations that take place within the coffee supply chain, as well as into the way the `Coffeebrain` smart contract keeps track of both exchanges and transformations along that value chain. We now describe this supply chain using the extended algebraic framework we have developed.

Let us first define a set K of *kinds*, which represent product types (cherries, parchment, etc.) and is defined as follows:

$$K = \{cherry, wet\_parchment, dry\_parchment, green, roasted, loss\}$$

And such that there is the following total order on K:

$$cherry \leq wet\_parchment \leq dry\_parchment \leq green \leq roasted \leq loss$$

We can then proceed to define the sets we are already familiar with in the context of our resource accounting model:

- The set of actors, $A = \{0,1\} \oplus \mathbb{N}$ with the first component being set to 1 if that actor can create new resources (i.e is a farmer), and 0 otherwise. The second component then denotes the actor's id. We single out the $(0,0)$ element of A and call it the *designated burn account*: its purpose is to store all the weight loss that proceeds from lossy transformations.

- The set of resource types $X = K \oplus \mathbb{N}$, with the first component of each tuple corresponding to that product's type, and the second component corresponding to a particular product.

- The set of resources $R = \coprod_X \mathbb{R}$, with each real associated to a product expressing this product's weight.

We then define the following functions:

- A valuation function over $\mathbb{R}$, $w : R \to_1 \mathbb{R}$, such that, for all $r \in R$:

$$w(r) = \sum_{x \in X} r(x)$$

which given any resource returns the weight contained within a resource.

- An ownership state predicate $p : S \to \{0, 1\}$ such that:

$$p(s) = \begin{cases} 1 & \text{if for all } (t, i) \in A,\, t = 0, \text{ for all } x \in X,\, s(a)(x) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The goal here being to make it so that only farmers can emit new resources.

- A transition predicate $q : E_w \to \{0, 1\}$ for this system must enforce the following properties in order to guarantee that all states are well-formed:

  1. It must be impossible to transfer products of the weight loss kind to any actor other than the designated burn account. Reciprocally, it must also be impossible for the burn account to transfer anything to another actor.
  2. For a transformation to be valid, the kinds of the inputs must be smaller than the kinds of the input (with respect to the total order we defined on K): green coffee can be transformed into roasted coffee and weight loss, but not into dry parchment, for example.

With such a formulation of the framework, the operations allowed by the `Coffeebrain` smart contract become quite simple to implement:

- *Minting* new products becomes trivial since all farmers have an infinite credit limit.

- *Burning* products simply amounts to transferring them to the designated burn account.

- *Splitting* a resource is trivial as well by definition of a vector space. For example, let us say that we have an initial state s_0 such that, for some $a \in A$, $x \in X$:

$$s_0 = \{a \mapsto 3x\}$$

It is trivial to transfer 50% of a's balance in the resource type x to an other actor $b \in A$ via the addition of a transformation to s_0:

$$s_0 + \{a \mapsto -\frac{3}{2} \cdot x, b \mapsto \frac{3}{2} \cdot x\} = \{a \mapsto \frac{3}{2} \cdot x, b \mapsto \frac{3}{2} \cdot x\}$$

- *Bundling* products can be done via a simple transformation that consumes al the inputs and produces an output of the same combined weight, for example:

  $$e_{bundle} = \{a \mapsto -w_1 x_1 - \cdots - w_n x_n + w_{new} x_{new}\}$$

  Where $x_1, \ldots, x_n$ the input types, x_{new} the output type, and $w_1, \ldots, w_n, w_{new} \geq 0$ their respective weights such that $w_{new} = \sum_{i=1}^{n} w_i$.

- Finally, *lossy transformations* can be represented by the set of valid transformations with two outputs: a product of kind less than $loss$ and a weight-loss product, the latter being transferred to the designated burn account.

Even though for the sake of simplicity we have decided to give all farmers an infinite credit limit, in practice this is not completely true: cooperatives usually conduct a land survey of each farmer in order to determine a predicted maximal yearly yield of a farmer's estate in order to ensure that all coffee sold by each individual farmer actually proceeds from that farmer's property, mostly for regulatory reasons (such as requirements imposed upon the cooperatives by certification agencies or the Federation itself). Nonetheless, this calls for a more complex definition of transition and state predicates, which we will not explore in this thesis.

Note also that this formulation of the coffee value chain in terms of our accounting framework does away with the queue/silo concept from the `Coffeebrain` smart contract. The decision to represent production steps using a data structure that functions in a way that emulates the physical production processes of the coffee industry was initially taken out of a desire to model the real world as closely as possible. However, as a consequence of the need to model all possible transformation processes along the production chain, it becomes necessary for each actor to combine the primitives exposed by the contract's interface in order to express processes that might not physically use FIFO silos. Thus, we assume here that the actor has sufficient insight into their production practices to submit transactions that reflect physical reality to the resource manager.

7 Discussion

7.1 Related work

The concept of generalized resource accounting as seen through the lens of resources, agents and events was first introduced by McCarthy[3] in 1982. He then uses those three entities to create a virtual representation of all the processes occurring within a company, with a separate REA model for each of those processes.

Rambaud et al.[1] introduced the use of algebraic structures such as vectors, monoids, etc. to represent the double-entry accounting system. More specifically, they represent the states of the system a balance vectors, where the set of balance vectors is defined as a module over a commutative ring with identity (that is, as a map from actors to balances of a single resource type, seen through the point of view of our framework). In particular, this model describes with systems that have a fixed account number. Henglein[2] then generalizes this model to systems with multiple resource types and arbitrary and infinite balance vectors.

Andersen et al.[7] define a language for the formal specification of contracts in a REA system. Such contracts correspond to state and transition predicates as defined in this dissertation.

Vogelsteller et al. introduce a specification for Ethereum smart contracts that operate following an account model for single resource systems with support for two-party transfers in their ERC-20 token standard specification[12].

Nakamoto's Bitcoin[10] uses an UTXO (unspent transaction output) model to allow for a state machine that allows for multi-party single-resource transfers.

7.2 Future work

In our formulation of the transformation-extended REA accounting model, state and transition predicates are seen merely as *black-box* functions that simply approve or reject ownership states and transfers respectively. This model would benefit from a more explicit definition of how such predicates are constructed, perhaps by leveraging the declarative contract specification language described by Andersen et. al[7]. In the particular case of netting, it would be interesting to see whether having more insight into how predicates are constructed leads us to a more general prioritized netting algorithm that is not restricted to credit limits as predicates. Moreover, the contract specification language introduced by Andersen should also be extended to allow for control over transformations, rather than resource transfers exclusively.

Another way in which our definition of predicates is quite narrow is that we do not allow for predicates to change across state changes, and we do not account for external factors that might influence the validity of transactions. In the coffee supply chain case, we briefly saw that the price at which coffee id bought and sold is set daily, and depends on the calculated yield factor of each individual product. Being able to explicitly account for the influence of external information on transfers and transformations would allow for a more accurate and expressive definition of state transition functions and we would, as a result, be able to formulate a more accurate model of reality.

In their netting paper, Cao et al.[4] leverage Pedersen commitments and zero-knowledge range proofs in order to devise a privacy-preserving extention of their netting algorithm. An interesting extension of our version of the algorithm would be to see how a similar goal can be accomplished so that netting the case of multi-party, multi-resource events can also preserve privacy, and if so, to which extent.

8 References

[1] Rambaud, Perez, Nehmer, Robinson, *Algebraic models for accounting systems*, World Scientific, 2010.

[2] Henglein, *Smart Declarative Contracts: Algebraic Resource Accounting*, Lecture 2 of 4 on smart declarative contracts, Oregon Programming Languages Summer School 2019, University of Oregon, June 2019.

[3] McCarthy, *The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment*, The Accounting Review LVII:3, pp. 554-578, July 1982.

[4] Cao, Yuan, De Caro, Nandakumar, Elkhiyaoui, Hu, *Decentralized Privacy-Preserving Netting Protocol on Blockchain for Payment Systems*, Financial Cryptography and Data Security, 2020.

[5] *Sustainable supply chains for bio-based products: Using blockchain technology to accelerate sustainability in bio-based supply chains*, research project supported by COWIfonden, 2019-2021

[6] Torres, `coffeebrain-ethereum`, git repository, 2020. `https://codeberg.org/juanhebert/coffeebrain-ethereum`

[7] Andersen, Elsborg, Henglein, Simonsen, Stefansen, *Compositional specification of commercial contracts*, International Journal on Software Tools for Technology Transfer (STTT) 8:6, pp. 485–516, November 2006.

[8] García, Ramírez, *Sostenibilidad económica de las pequeñas explotaciones cafeteras Colombianas* [*Economic sustainability in small colombian coffee farms*], Ensayos sobre economía cafetera 15(18):73-89, December 2002.

[9] Orozco, *Descripción de especies y variedades de café* [*Description of coffee species and varieties*], 1986.

[10] Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2009.

[11] Buterin, *A next-generation smart contract and decentralized application platform*, 2014.

[12] Vogelsteller, Buterin, *EIP-20: ERC-20 Token Standard*, 2015.
`https://eips.ethereum.org/EIPS/eip-20`

[13] Torres, `rea-haskell`, git repository, 2020.
`https://codeberg.org/juanhebert/rea-haskell`

A Source code for the Haskell implementation

This code is also available as part of an online Git repository [13].

```haskell
import qualified Data.HashMap.Strict as M
import Data.Bifunctor (first, second)
import Control.Monad ((>=>))
import Data.Tuple (swap, uncurry)
import Data.List (groupBy, sortOn, union)

class Monoid a => LeftMul a where
    (<.>) :: Double -> a -> a

neg :: LeftMul a => a -> a
neg = ((-1) <.>)

sub :: (LeftMul a) => a -> a -> a
sub x y = x <> neg y

instance Semigroup Double where
    (<>) = (+)

instance Monoid Double where
    mempty = 0
    mappend = (<>)

instance LeftMul Double where
    (<.>) = (*)

type Actor = String
type ResourceType = String

-- Resources

newtype Resource = Resource { getResource :: M.HashMap ResourceType Double }
    deriving (Show, Eq)

instance Semigroup Resource where
    (<>) (Resource r1) (Resource r2) =
        Resource $ M.filter (/= 0) $ M.unionWith (+) r1 r2
```

```
instance Monoid Resource where
    mempty = Resource M.empty
    mappend = (<>)

instance LeftMul Resource where
    (<.>) k = Resource . M.map (k <.>) . getResource

isZero :: (Monoid a, Eq a) => a -> Bool
isZero = (== mempty)

makeResource :: [(ResourceType, Double)] -> Resource
makeResource = Resource
    . M.filter (not . isZero)
    . M.fromList


-- Ownership states

newtype OwnershipState =
    OwnershipState { getOwnershipState :: M.HashMap Actor
     ↪ Resource }
    deriving (Show, Eq)

instance Semigroup OwnershipState where
    (<>) (OwnershipState s1) (OwnershipState s2) =
        OwnershipState $ M.filter (not . isZero) $ M.unionWith
         ↪  (<>) s1 s2

instance Monoid OwnershipState where
    mempty = OwnershipState M.empty
    mappend = (<>)

instance LeftMul OwnershipState where
    (<.>) k = OwnershipState . M.map (k <.>) . getOwnershipState

nullOwnershipState :: OwnershipState -> Bool
nullOwnershipState = null . getOwnershipState

makeOwnershipState :: [(Actor, Resource)] -> OwnershipState
makeOwnershipState = OwnershipState
```

```haskell
    . M.filter (not . isZero)
    . M.fromList

makeOwnershipState' :: [(Actor, [(ResourceType, Double)])] -> OwnershipState
makeOwnershipState' = makeOwnershipState . map (second makeResource)

support :: OwnershipState -> [Actor]
support = M.keys . getOwnershipState

supportWith :: (Monoid a, Eq a) => (Resource -> a) -> OwnershipState -> [Actor]
supportWith w = M.keys . M.filter (not . isZero . w) . getOwnershipState

sumOwnershipState :: OwnershipState -> Resource
sumOwnershipState = M.foldl' (<>) mempty . getOwnershipState


-- Transfers

type Transfer = OwnershipState -- zero-sum property enforced via isTransfer

isTransfer :: OwnershipState -> Bool
isTransfer = isZero . sumOwnershipState

isTwoParty :: OwnershipState -> Bool
isTwoParty = (\l -> length l == 2) . M.keys . getOwnershipState

commitTransfer :: (OwnershipState -> Bool) -> Transfer -> OwnershipState
    -> Maybe OwnershipState
commitTransfer p t s = if p s' then Just s' else Nothing
    where s' = s <> t

-- Note: Kleisli composition as defined in section 3.2 is equivalent to the
-- (>=>) function in Control.Monad
kleisli :: (OwnershipState -> Maybe OwnershipState)
```

```haskell
    -> (OwnershipState -> Maybe OwnershipState)
    -> (OwnershipState -> Maybe OwnershipState)
kleisli = (>=>)


-- Events

type Event = OwnershipState -- zero-sum property w.r.t w enforced via isEvent

-- Pre-condition: w must be a homomorphism
isEvent :: (Monoid a, Eq a) => (Resource -> a)  -> OwnershipState -> Bool
isEvent w = isZero . w . sumOwnershipState

isTransformation :: (Monoid a, Eq a) => (Resource -> a) -> OwnershipState
    -> Bool
isTransformation w = M.foldl' f True . getOwnershipState
    where f acc curr = if isZero $ w curr then acc else False

commitEvent :: (OwnershipState -> Bool) -> (Event -> Bool) -> Event
    -> OwnershipState
    -> Maybe OwnershipState
commitEvent p q e s = if p s' && q e then Just s' else Nothing
    where s' = s <> e

-- Note that we have commitTransfer p = commitEvent p (\_ -> True)

decomposeEventLoop :: OwnershipState -> [Transfer]
    -> ([Transfer], Event)
decomposeEventLoop e ts
    | M.size (getOwnershipState e) > 1 =
        let
            [(a, r1), (b, _)] = take 2 $ M.toList $ getOwnershipState e
            t = makeOwnershipState [(a, r1), (b, neg r1)]
            ts' = t:ts
            e' = sub e t
```

```haskell
        in
            decomposeEventLoop e' ts'
    | otherwise = (ts, e)

-- Pre-condition: t must be a two party transfer
twoPartyToSingleResource :: Transfer -> [Transfer]
twoPartyToSingleResource t =
    let
        [(a, r1), (b, _)] = M.toList $ getOwnershipState t
        makeTransfer (k, v) = makeOwnershipState' [
                (a, [(k, v)]),
                (b, [(k, -v)])
            ]
    in map makeTransfer $ M.toList $ getResource r1

decomposeEvent :: Event -> ([Transfer], Event)
decomposeEvent e =
    first (>>= twoPartyToSingleResource) $ decomposeEventLoop e []

decomposeTransfer :: Transfer -> [Transfer]
decomposeTransfer = fst . decomposeEvent

fuse :: ([Transfer], Event) -> [Event]
fuse (ts, e) | e == mempty = ts
             | otherwise = e:ts

-- Pre-condition: t must be either a single-resource two-party transfer
-- or a one-party transformation.
spender :: Transfer -> Actor
spender t =
    let
        ((a, r1):ts) = M.toList $ getOwnershipState t
        [(k, v)] = M.toList $ getResource r1
    in
        case ts of
            ((b, _):[]) -> if v < 0 then a else b
            [] -> a
```

```haskell
-- Pre-condition: t must be either a single-resource two-party transfer
-- or a one-party transformation.
recipient :: Transfer -> Actor
recipient t =
    let
        ((a, r1):ts) = M.toList $ getOwnershipState t
        [(k, v)] = M.toList $ getResource r1
    in
        case ts of
            ((b, _):[]) -> if v > 0 then a else b
            _ -> a

sameSpender :: Transfer -> Transfer -> Bool
sameSpender t1 t2 = spender t1 == spender t2

type TransactionId = Integer

taggedGroupBySpender :: TransactionId -> [Transfer]
    -> [(TransactionId, Actor, Transfer)]
taggedGroupBySpender i =
    (map (\ts -> (i, spender $ head ts, mconcat ts)))
    . groupBy sameSpender

prependGroupSpender :: [(TransactionId, Actor, Transfer)]
    -> (Actor, [(TransactionId, Transfer)])
prependGroupSpender ts@((_, a, _):_) = (a, map (\ (a, _, b) -> (a, b)) ts)

mergeByTransaction :: [(TransactionId, Actor, Transfer)]
    -> [(TransactionId, Actor, Transfer)]
mergeByTransaction =
    map combine
    . groupBy (\a b -> getId a == getId b)
    . sortOn getId
    where
        getId (a, _, _) = a
        op (i1, a1, t1) (_, _, t2) = (i1, a1, t1 <> t2)
        combine = foldl1 op

positiveBalance :: Resource -> Bool
```

```haskell
positiveBalance = foldl f True . M.toList . getResource
    where f acc (_, v) = v >= 0 && acc

allBalancesPositive :: OwnershipState -> Bool
allBalancesPositive =
     (== mempty)
     . M.filter (not . positiveBalance)
     . getOwnershipState

getBalance :: Actor -> OwnershipState -> Resource
getBalance a = M.lookupDefault mempty a . getOwnershipState

-- Given the initial set of pending transactions, tagged with
↪ their ids,
-- this function generates the initial state of the algorithm.
makeNettingQueues :: [(TransactionId, Event)]
    -> [(Actor, [(TransactionId, Transfer)])]
makeNettingQueues =
    map (second (sortOn fst))
     . map prependGroupSpender
     . map mergeByTransaction
     . groupBy (\ a b -> getActor a == getActor b)
     . sortOn getActor
     . (>>= (\ (i, e) -> taggedGroupBySpender i $ fuse $
     ↪ decomposeEvent e))
    where
        getActor (_, a, _) = a

firstNettingPhase ::
    OwnershipState -> [(Actor, [(TransactionId, Transfer)])] ->
    ↪ [TransactionId]
    -> (Actor, [(TransactionId, Transfer)])
    -> (Actor, [(TransactionId, Transfer)], [TransactionId])
firstNettingPhase s qs rejected (a, q) =
    let
        (prefix, rejected', _) = foldl f ([], rejected, True) q
    in
        (a, prefix, rejected')
    where
        f (prevPrefix, prevRejected, stillLooking) curr@(id, t) =
            let
```

```haskell
                incoming = mconcat $ approvedByOthers a rejected qs
                predicted = getBalance a $
                    s <> incoming <> (mconcat (map snd prevPrefix)) <> t
            in
                if positiveBalance predicted
                    && (not $ elem id prevRejected)
                    && stillLooking
                    then (curr:prevPrefix, prevRejected, stillLooking)
                    else (prevPrefix, id:prevRejected, False)

secondNettingPhase ::
    [(Actor, [(TransactionId, Transfer)], [TransactionId])]
    -> ([Event], [TransactionId], [(Actor, [(TransactionId, Transfer)])])
secondNettingPhase = foldl f ([], [], [])
    where
        f (pApproved, pRejected, pState) (cActor, cApproved, cRejected) =
            let
                approved = pApproved ++ map snd cApproved
                rejected = union pRejected cRejected
                state = (cActor, cApproved):pState
            in
                (approved, rejected, state)

approvedByOthers :: Actor -> [TransactionId]
    -> [(Actor, [(TransactionId, Transfer)])]
    -> [Transfer]
approvedByOthers a rejectedIds =
    map snd
    . filter (\(id, _) -> not $ elem id rejectedIds)
    . concat
    . map snd
    . filter (\(a', _) -> a /= a')

nettingLoop :: OwnershipState -> [Event] -> [Event] -> [TransactionId]
    -> [(Actor, [(TransactionId, Transfer)])]
```

```
    -> ([Event], [TransactionId])
nettingLoop s prevApproved approved rejected qs =
    if prevApproved == approved
        then (approved, rejected)
        else
            let qs'' = map (firstNettingPhase s qs rejected) qs
                (approved', rejected', qs') = secondNettingPhase
                ↪  qs''
            in
                nettingLoop s approved approved' rejected' qs

net :: OwnershipState -> [(TransactionId, Event)] -> ([Event],
↪  [TransactionId])
net s ts =
    let
        queues = makeNettingQueues ts
        initialApproved = map snd ts
    in nettingLoop s [] initialApproved [] queues



-- Sample values for testing

w :: Resource -> Double
w = M.foldl' (<>) mempty . getResource

e :: Event
e = makeOwnershipState' [
        ("a", [("x", -30)]),
        ("b", [("x", 15), ("y", 7), ("z", 8)]),
        ("c", [("z", -10)]),
        ("d", [("x", 5), ("z", 5)])
    ]

d1 :: ([Transfer], Event)
d1 = decomposeEvent e

t :: Transfer
t = makeOwnershipState' [
        ("a", [("x", -30), ("y", 10)]),
        ("b", [("x", 25), ("y", -2), ("z", 5)]),
```

```
            ("c", [("x", 5), ("y", -8), ("z", -5)])
        ]

s :: OwnershipState
s = makeOwnershipState' [
            ("a", [("USD", 40)]),
            ("b", [("Bike", 10)]),
            ("c", [("USD", 10)])
        ]

p1 :: Event
p1 = makeOwnershipState' [
            ("a", [("USD", -50), ("Bike", 2)]),
            ("b", [("USD", 50), ("Bike", -3)]),
            ("c", [("Bike", 1)])
        ]

p2 :: Event
p2 = makeOwnershipState' [
            ("a", [("USD", 10)]),
            ("c", [("USD", -10)])
        ]

p3 :: Event
p3 = makeOwnershipState' [
            ("a", [("Bike", -3)]),
            ("c", [("Bike", 3)])
        ]

ps :: [(TransactionId, Event)]
ps = zip [1, 2, 3] [p1, p2, p3]

n = first mconcat $ net s ps

-- The following are all True

-- Event decomposition
ppt0 = isEvent w e
ppt1 = isEvent w e' && isTransformation w e'
    where e' = snd d1
ppt2 = and $ map (\ x -> isTransfer x && isTwoParty x) $ fst d1
```

```
ppt3 = (mconcat $ fst d1) <> snd d1 == e
ppt4 = isZero $ w $ M.foldl' (<>) mempty $ getOwnershipState $ snd d1

-- Transfer decomposition
ppt5 = isTransfer t
ppt6 = and $ map (\x -> isTransfer x && isTwoParty x) $ decomposeTransfer t
ppt7 = (mconcat $ decomposeTransfer t) == t
ppt8 = isZero $ snd $ decomposeEvent t

-- Netting

ppt9 = snd n == [3]
ppt10 = fst n == (p1 <> p2)
ppt11 = allBalancesPositive $ (s <> fst n)

-- Global test

pass = and [
        ppt1,
        ppt2,
        ppt3,
        ppt4,
        ppt5,
        ppt6,
        ppt7,
        ppt8,
        ppt9,
        ppt10,
        ppt11
    ]
```

B Source code for the `Coffeebrain` smart contract

This code is also available as part of an online Git repository [6].

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;
pragma experimental ABIEncoderV2;

/// @title Coffeebrain
/// @notice A smart contract for the coffee supply chain.
/// @author Juan Manuel Hébert
contract Coffeebrain {

    struct File {
        string location; // URL
        bytes32 hash; // SHA-3
        string MIMEType;
        bool isDefined;
    }

    struct Product {
        uint256 kind; // dry parchment, green, roasted, etc.
        uint256 variety; // tipica, borbon, castillo, etc.
        uint256 weight; // in grams
        bool isDefined;
    }

    struct MaybeIndex {
        uint256 index;
        bool isDefined;
    }

    struct Set {
        bytes32[] array;
        mapping(bytes32 => MaybeIndex) map;
    }

    struct StoredProduct {
```

```
        bytes32 product;
        uint256 weight;
    }

    struct Silo {
        uint256 weight;
        uint256 first;
        StoredProduct[] content;
    }

    struct Certificate {
        address certifier;
        uint256 startDate;
        uint256 endDate;
        uint256 kind;
        bool isDefined;
    }

    struct Practice {
        address observer;
        uint256 date;
        uint256 kind;
        bool isDefined;
    }

    struct Actor {
        bool canEmit;
        Set inventory;
        Set ownership;
        mapping(bytes32 => Silo) silos;
        bool isDefined;
    }

    struct Transaction {
        bytes32 product;
        address sender;
        address recipient;
        uint256 price;
        uint256 currency;
        bool isDefined;
        bool isCancelled;
```

```
        bool isConfirmed;
    }

    address payable private _owner;
    address public burnAccount;

    mapping (address => Actor) private _actors;
    mapping (bytes32 => Product) private _products;
    mapping (bytes32 => Certificate) private _certificates;
    mapping (bytes32 => Practice) private _practices;
    mapping (bytes32 => Transaction) private _custodyTransactions;
    mapping (bytes32 => Transaction) private _ownershipTransactions;

    constructor() public {
        _owner = msg.sender;
        burnAccount = address(0);
        _actors[burnAccount].isDefined = true;
    }

    // Utility functions (private)

    function setAdd(Set storage set, bytes32 productId) private {
        set.array.push(productId);
        set.map[productId] = MaybeIndex(set.array.length - 1, true);
    }

    function setDelete(Set storage set, bytes32 productId) private {
        MaybeIndex storage mIndex = set.map[productId];
        uint256 index = mIndex.index;
        uint256 length = set.array.length;
        require(mIndex.isDefined, 'Invalid index (undefined in map).');
        require(index < length, 'Invalid index (out of bounds).');

        bytes32 lastProductId = set.array[length-1];
        set.array[index] = lastProductId;
```

```
        set.array.pop();
        set.map[productId] = MaybeIndex(0, false);
        set.map[lastProductId].index = index;
    }

    function siloNQ(Silo storage silo, bytes32 productId) private {
        uint256 weight = _products[productId].weight;

        silo.weight += weight;
        StoredProduct memory pending = StoredProduct({
            weight: weight,
            product: productId
        });
        silo.content.push(pending);
    }

    event AddInput(bytes32 input, uint256 weight, bytes32 output);

    function siloDQ(Silo storage silo, uint256 weight, bytes32 output) private {
        require(weight <= silo.weight, 'Cannot pop more than is contained in the silo.');

        uint256 remainingWeight = weight;
        while(remainingWeight > 0) {
            StoredProduct storage current = silo.content[silo.first];

            if(current.weight <= remainingWeight) {
                emit AddInput(current.product, current.weight, output);
                remainingWeight -= current.weight;
                silo.weight -= current.weight;
                silo.first += 1;
            } else {
                emit AddInput(current.product, remainingWeight, output);
                silo.content[silo.first].weight -= remainingWeight;
```

```
                    silo.weight -= remainingWeight;
                    remainingWeight = 0;
                }
            }

            // If the silo is empty, we reset it
            if(silo.weight == 0) {
                silo.first = 0;
                delete silo.content;
            }
        }

        // Modifiers

        modifier authenticated {
          require(_actors[msg.sender].isDefined, 'Message sender not
    ↪   registered.');
          _;
        }

        modifier registered(address actor) {
          require(_actors[actor].isDefined, 'Actor does not exist.');
          _;
        }

        // Public API functions

        event Registration(address actor, string name, string email,
    ↪   string location, bool canEmit, string pictureURL, bytes32
    ↪   pictureHash);

        /// @notice Register a new actor.
        /// @dev Only the owner of the contract can register new
    ↪   actor.
        /// @param actorId The address of the actor to be registered.
        /// @param name The name of the actor to be registered.
        /// @param location Human-readable location of the actor.
        /// @param pictureURL URL of the actor's profile picture
        /// @param pictureHash Hash of the actors profile picture
        /// @param canEmit A boolean that determines whether the
    ↪   actor will be able to mint new products.
```

```
    function register(address actorId, string memory name, string
    ↪ memory email, string memory location, string memory
    ↪ pictureURL, bytes32 pictureHash, bool canEmit) public {
        Actor storage actor = _actors[actorId];
        require(msg.sender == _owner, 'Only the contract owner
    ↪ can register new actors');
        require(!actor.isDefined, 'Actor already exists.');

        actor.canEmit = canEmit;
        actor.isDefined = true;

        emit Registration(actorId, name, email, location,
    ↪ canEmit, pictureURL, pictureHash);
    }

    event Transformation(address emitter, bytes32 productId,
    ↪ uint256 kind, uint256 variety, uint256 weight);

    /// @notice Mint a new product.
    /// @dev Only actors whose canEmit flag is set to true can
    ↪ use this function.
    /// After being created, the product is added to its
    ↪ creator's ownership and
    /// custody sets.
    /// @param productId Fresh ID to be assigned to the new
    ↪ product.
    /// @param kind Type of product (see integer correspondences
    ↪ in README)
    /// @param variety Coffee variety (see integer
    ↪ correspondences in README)
    /// @param weight Weight of the newly minted product.
    function mint(bytes32 productId, uint256 kind, uint256
    ↪ variety, uint256 weight) public authenticated {
        Product storage product = _products[productId];
        Actor storage sender = _actors[msg.sender];
        require(!product.isDefined, 'Product already exists.');
        require(sender.canEmit, 'User is not allowed to mint new
    ↪ products.');

        product.kind = kind;
        product.variety = variety;
```

```
        product.weight = weight;
        product.isDefined = true;

        setAdd(sender.inventory, productId);
        setAdd(sender.ownership, productId);

        emit Transformation(msg.sender, productId, kind, variety, weight);
    }

    event Shipment(bytes32 transaction, bytes32 product, address recipient, address sender);

    /// @notice Initiate a shipping transaction.
    /// @dev An actor can only call this function on products in their inventory.
    /// If the transaction is created successfully, the product is removed from
    /// the calling actor's inventory.
    /// @param transactionId Fresh ID to be assigned to the new shipping transaction.
    /// @param productId ID of the product to be shipped.
    /// @param to Address of the intended recipient.
    function ship(bytes32 transactionId, bytes32 productId, address to) public authenticated registered(to) {
        Transaction storage transaction = _custodyTransactions[transactionId];
        Actor storage sender = _actors[msg.sender];
        MaybeIndex storage mIndex = sender.inventory.map[productId];
        require(!transaction.isDefined, 'Transaction already exists');
        require(mIndex.isDefined, 'Product is not in the inventory of the sender.');

        transaction.product = productId;
        transaction.sender = msg.sender;
        transaction.recipient = to;
        transaction.isDefined = true;

        setDelete(sender.inventory, productId);
```

```
        emit Shipment(transactionId, productId, to, msg.sender);
    }

    event Reception(bytes32 transaction, address recipient);

    /// @notice Successfully conclude a shipping transaction.
    /// @dev An actor can only call this function open transactions destined to them.
    /// If successful, this operation adds the product to the actor's inventory.
    /// @param transactionId ID of the transaction to be concluded.
    function confirmReceipt(bytes32 transactionId) public authenticated {
        Transaction storage transaction = _custodyTransactions[transactionId];
        require(transaction.isDefined, 'No shipping transaction found.');
        require(transaction.recipient == msg.sender, 'Actor cannot finalize the transaction.');
        require(!transaction.isCancelled && !transaction.isConfirmed, 'Transaction already finalized.');

        transaction.isConfirmed = true;

        bytes32 productId = transaction.product;
        setAdd(_actors[msg.sender].inventory, productId);

        emit Reception(transactionId, msg.sender);
    }

    event Sale(bytes32 transaction, bytes32 product, address buyer, address seller, uint256 price, uint256 currency);

    /// @notice Initiate an ownership transfer transaction.
    /// @dev An actor can only call this function on products they currently own.
    /// If the transaction is created successfully, the product is removed from
    /// the calling actor's ownership set.
```

```
    /// @param transactionId Fresh ID to be assigned to the new ownership transaction.
    /// @param productId ID of the product to be sold.
    /// @param to Address of the intended buyer.
    /// @param price Amount of currency to be paid out.
    /// @param currency Currency (see integer correspondences in README).
    function sell(bytes32 transactionId, bytes32 productId, address to, uint256 price, uint256 currency) public authenticated registered(to) {
        Transaction storage transaction = _ownershipTransactions[transactionId];
        Actor storage sender = _actors[msg.sender];
        MaybeIndex storage mIndex = sender.ownership.map[productId];
        require(!transaction.isDefined, 'Transaction already exists.');
        require(mIndex.isDefined, 'Seller does not own the product.');

        transaction.product = productId;
        transaction.sender = msg.sender;
        transaction.recipient = to;
        transaction.price = price;
        transaction.currency = currency;
        transaction.isDefined = true;

        setDelete(sender.ownership, productId);

        emit Sale(transactionId, productId, to, msg.sender, price, currency);
    }

    event Purchase(bytes32 transaction, address buyer);

    /// @notice Successfully conclude an ownership-transfer transaction.
    /// @dev An actor can only call this function open transactions destined to them.
    /// If successful, the product is added to the calling actor's ownership set.
```

```
    /// @param transactionId ID of the transaction to be concluded.
    function confirmPurchase(bytes32 transactionId) public authenticated {
        Transaction storage transaction = _ownershipTransactions[transactionId];
        require(transaction.isDefined, 'No purchase transaction found.');
        require(transaction.recipient == msg.sender, 'Product was not sold to the requesting actor.');
        require(!transaction.isCancelled && !transaction.isConfirmed, 'Transaction is already finalized.');

        transaction.isConfirmed = true;

        setAdd(_actors[msg.sender].ownership, transaction.product);

        emit Purchase(transactionId, msg.sender);
    }

    event Storage(bytes32 product, bytes32 silo, address actor);

    /// @notice Store a product in a silo.
    /// @dev An actor can only store products they both own and have custody over.
    /// Removes the product from the actor's inventory.
    /// @param productId Product to be stored.
    /// @param siloId ID of the silo.
    function store(bytes32 productId, bytes32 siloId) public authenticated {
        require(_actors[msg.sender].inventory.map[productId].isDefined, 'Actor cannot store a product they do not have. ');

        siloNQ(_actors[msg.sender].silos[siloId], productId);
        setDelete(_actors[msg.sender].inventory, productId);

        emit Storage(productId, siloId, msg.sender);
    }
```

```
    /// @notice Use a silo's contents to create a new product.
    /// @dev The new product is added to the actor's custody and ownership sets.
    /// @param siloId ID of the silo.
    /// @param weight Weight to be taken out of the silo in grams.
    /// @param resultId Fresh ID to be assigned to the resulting product.
    /// @param resultKind Kind of the resulting product (see integer correspondences in README).
    function transform(bytes32 siloId, uint256 weight, bytes32 resultId, uint256 resultKind) public authenticated {
        Product storage product = _products[resultId];
        Actor storage sender = _actors[msg.sender];
        require(!product.isDefined, 'Product with that id already exists.');

        product.kind = resultKind;
        product.weight = weight;
        product.isDefined = true;

        siloDQ(sender.silos[siloId], weight, resultId);
        setAdd(sender.inventory, resultId);
        setAdd(sender.ownership, resultId);

        emit Transformation(msg.sender, resultId, resultKind, 0, weight);
    }

    event Burn(bytes32 product, address actor);

    /// @notice Burn a product.
    /// @dev Sends the product to the burn account (both ownership and custody are affected).
    /// @param productId ID of the product to be burned.
    function burn(bytes32 productId) public authenticated {
        Actor storage sender = _actors[msg.sender];
        Actor storage burner = _actors[burnAccount];
        require(sender.inventory.map[productId].isDefined, 'An actor can only burn products they have.');
```

```
        require(sender.ownership.map[productId].isDefined, 'An actor can only burn products they own.');

        setDelete(sender.inventory, productId);
        setDelete(sender.ownership, productId);
        setAdd(burner.inventory, productId);
        setAdd(burner.ownership, productId);

        emit Shipment(bytes32(0), productId, burnAccount, msg.sender);
        emit Burn(productId, msg.sender);
    }

    event Certification(bytes32 certificate, address actor, address certifier, uint256 kind, uint256 startDate, uint256 endDate);

    /// @notice Certify an actor.
    /// @param actor Actor to be certified.
    /// @param certificateId Fresh ID for the new certificate.
    /// @param kind Certificate kind (see integer correspondences in README).
    /// @param startDate UNIX timestamp / 1000 of the date from which the certificate applies.
    /// @param endDate UNIX timestamp / 1000 of the certificate's expiration date.
    function certify(address actor, bytes32 certificateId, uint256 kind, uint256 startDate, uint256 endDate) public authenticated {
        Certificate storage certificate = _certificates[certificateId];
        require(!certificate.isDefined, 'Certificate already exists');

        certificate.certifier = msg.sender;
        certificate.startDate = startDate;
        certificate.endDate = endDate;
        certificate.kind = kind;
        certificate.isDefined = true;

```

```
        emit Certification(certificateId, actor, msg.sender, kind, startDate, endDate);
    }


    event Observation(bytes32 practice, address actor, address observer, uint256 kind, uint256 date);

    /// @notice Add a sustainability practice observation.
    /// @param actor Address of the actor the observation applies to.
    /// @param practiceId Fresh ID for the newly created practice.
    /// @param kind Practice kind (see integer correspondences in README).
    /// @param date UNIX timestamp / 1000 of the date of the observation.
    function observe(address actor, bytes32 practiceId, uint256 kind, uint256 date) public authenticated {
        Practice storage practice = _practices[practiceId];
        require(!practice.isDefined, 'Practice already exists');

        practice.observer = msg.sender;
        practice.date = date;
        practice.kind = kind;
        practice.isDefined = true;

        emit Observation(practiceId, actor, msg.sender, kind, date);
    }

    event Evidence(bytes32 eventId, uint8 eventType, address author, string hash);

    function addEvidence(bytes32 eventId, uint8 eventType, string memory hash) public authenticated {
        require(eventType < 11, 'Invalid event type');
        emit Evidence(eventId, eventType, msg.sender, hash);
    }

    // Test functions
```

```

    function lookupInventory(address actor, bytes32 productId) public view returns (bool) {
        return _actors[actor].inventory.map[productId].isDefined;
    }

    function lookupOwnership(address actor, bytes32 productId) public view returns (bool) {
        return _actors[actor].ownership.map[productId].isDefined;
    }

    function getProductWeight(bytes32 productId) public view returns (uint256) {
        return _products[productId].weight;
    }

    function getSiloWeight(address actor, bytes32 siloId) public view returns (uint256) {
        return _actors[actor].silos[siloId].weight;
    }

    // Kill function
    function kill() public {
        require(msg.sender == _owner, "Only the owner of the contract can kill it");
        selfdestruct(_owner);
    }
}
```

C Copyright attribution