Z^{Ac}—FORMALISING ACCOUNTING SYSTEMS IN Z

STEVE REEVES

1. The basic systems

1.1. **Single-entry.** Here a transaction leads to an entry in one single account. And the record for this account is just a sequence of transactions, perhaps written as lines in a ledger.

Typically a transaction is given by a date, a description and an amount, which is either added to the money a company owns (i.e. someone pays it money, say or goods or services) or taken from (it pays someone money, say because they buy some raw materials or for running-costs). So, an entry might be

Date	Description	Amount
1st January 2017	Sold Tissot watch	$250.00
3rd March 2017	Bought can of oil	-$40.00

A slightly more refined version might split the "Amount" column into "Income" and "Expenditure"

Date	Description	Income	Expenditure
1st January 2017	Sold Tissot watch	$250.00	
3rd March 2017	Bought can of oil		$40.00

just to make things rather clearer. But there would still be just one account involved: the money we currently hold in the company. The name single-entry comes from the fact that each transaction talks about (the same) single account. (See Ijiri's point about what the "double" in double-entry refers to [].)

This way of running a company (the experts tell us, and the laws allow us) is OK for small, simple companies that just deal in cash (or at least don't deal in credit for itself or customers) and who want to keep a simple record of where they stand in terms of what money the have at any one moment in time.

Once a company wants to start selling things on credit, for example, it needs, though, to have a way of keeping track not just of the cash is has now, but also what cash is still owed to it. That is not available to use at this moment, but it will eventually appear (in a perfect world). This starts to allow a company to predict its future, for example. This also means that when it sells something, cash might not immediately flow into its account (whereas it does for the first transaction in the example above). But in order to keep track of what it is owed, another account needs to be created, so when a customer buys on credit the transaction does not mean money moves into its cash account, but

into another account that records other sorts of asset. So, we now have two accounts, and thus we enter the world of double-entry.

Just to anticipate what is coming, we might amend our transactions above further, as in:

Date	Description	Account	Amount
1st January 2017	Sold Tissot watch	Capital	$250.00
3rd March 2017	Bought can of oil	Capital	-$40.00

and even to:

Date	Description	Account	Income	Expenditure
1st January 2017	Sold Tissot watch	Capital	$250.00	
3rd March 2017	Bought can of oil	Capital		$40.00

1.2. **Double (and why, and its mad (well...conventional) naming for credits and debits etc).** In order to keep track of two or more accounts (which give the benefits of being able to deal with credit, loans, invoices etc.) we have to be more sophisticated with our recording methods. This in turn requires that we have more checks in place so that the more complicated system does not get out of hand. Continuing our example from above shows what is needed.

Our transactions above in the single case would, assuming we now have asset accounts (what the company owns, like stock and cash) and liabilities accounts (like some debt owed to a bank due to a loan from the bank, or an invoice comes in from another company for some asset we bought from them) now become something like:

Date	Description	Account	Income	Expenditure
1st January 2017	Sold Tissot watch	Stock		$250.00
		Bank	$250.00	
3rd March 2017	Bought can of oil	Bank		$40.00
		Equipment	$40.00	

and the transaction now has two lines since information about two accounts needs to be recorded.

All the accounts in this expanded example are *asset* accounts, since they are where we hold things we own. (We squashed these three accounts into the Capital account in the previous section.) The main point of this new idea of several accounts, though, is that we can have other sorts of account, so that we can keep track of invoices, say, where we give, essentially, a promise to pay someone for an asset at a later agreed date. So, we don't use an asset to pay for an asset in that transaction, we use a liability (that is, the promise to pay later).

Continuing to make our example more sophisticated, we can introduce a liability account called *Promise to pay* where we record these promises, i.e. invoices sent with goods we receive, to pay later. So our second transaction above now becomes:

Date	Description	Account	Income	Expenditure
3rd March 2017	Bought can of oil	Promise to pay		$40.00
		Equipment	$40.00	

At this point the headings on the last two columns start to be misleading. They made sense when there was just one account that held out assets. But now we have accounts that hold liabilities, and the whole point of them is that we have not, at this moment, spent anything, so *Expenditure* as a title needs changing. The promise to pay later is an increase in liability, so by convention this is viewed as a credit to that account (our liabilities got bigger). Similarly, if our liabilities decrease, because we pay off a promise, then we have debited our liability account. This suggests the following improved headings:

Date	Description	Account	Debit	Credit
3rd March 2017	Bought can of oil	Promise to pay		$40.00
		Equipment	$40.00	

Well, improved in the sense that the rightmost heading now makes sense. But the one next to it? Hm!

But note that the second line of the transaction is talking about an asset account, Equipment, whereas the first was talking about a liability account, so perhaps that difference explains the rather counter-intuitive name in relation to the second line and the penultimate column?

Underlying the double-entry method is the so-called *accounting equation*, which simply says that[1]

$$assets = liabilities$$

being true defines "the company is keeping its accounts correctly". We have already seen that it makes intuitive sense to say that an increase in liability is a credit in that account. To allow the accounting equation to hold, then, the assets must also increase in this case, but to make the "balancing" in that transaction apparent (when we see a credit we expect to see a matching debit) then we are forced (to make all this work) to call an increase in an asset a debit. Once this is accepted then we can summarise: increases in liabilities are called credits, and decreases in liabilities are called debits; increases in assets are called debits, and decreases in assets are called credits.

The accounting equation holds for all the accounts if we require, for every transaction, that

$$debit = credit$$

So, a global property "the accounting equation holds" is reduced to a local one "debit = credit for this transaction". (Room for some Z-style promotion here, surely!)

This reduction of a global property to a local one (or at least one we can keep track of just by checking each transaction as it happens) probably makes the strange naming

[1] in fact it's a bit more complicated since the right-hand side also mentions "ownership interest", but here we will just talk about liabilities.

convention (which is all it is, we could use any words here) in terms of credit and debit worthwhile.

Finally, here are both our transactions with the new naming:

Date	Description	Account	Debit	Credit
1st January 2017	Sold Tissot watch	Stock		$250.00
		Bank	$250.00	
3rd March 2017	Bought can of oil	Promise to pay		$40.00
		Equipment	$40.00	

Note that the first transaction all takes place within asset accounts, but the rules still work given the naming conventions.

1.3. **Triple (useful tool for a business planner).**

1.4. **UTXO and Bitcoin.**

1.5. **Ethereum and beyond.**

2. Building a Z model

2.1. **Single-entry accounting.** We have given sets

$[DATE, TEXT, MONEY]$

and the useful schema

Transaction$_0$

$date : DATE$
$description : TEXT$
$amount : MONEY$

This exactly mirrors a line in the table in Section 1.1.

Note the schema has no predicate part, so we impose no conditions on the transactions. (Should there be any??)

Now the state space for the account is just a sequence of transactions (with an initial state with no transactions):

SingleState

$ssaccount : \text{seq } Transaction_0$

InitSingleState

SingleState

$saccount = <>$

Again, this exactly mirrors the table in Section 1.1.

Note this schema also has no predicate part, so we impose no conditions on the account. (Should there be any??)

Then we finally have a single state-changing operation, which itself uses an operation that just sets-up a new transaction (this follows a, simplified, version of the Z idiom of promotion...done here just because this might be a useful bit of structuring when things get more complicated later..)

$$
\begin{array}{|l}
\hline
NewTransaction_0 \\
Transaction' \\
date? : DATE \\
description? : TEXT \\
amount? : MONEY \\
\hline
date' = date? \\
description' = description? \\
amount' = amount? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\Phi DoSomeBusiness \\
\Delta SingleState \\
Transaction' \\
\hline
saccount' = saccount^\frown < \theta Transaction' > \\
\hline
\end{array}
$$

We then promote this local operation (which works on a single transaction) to the whole system ("the accounts") using the following[2]

$$DoSomeBusiness_0 \mathrel{\widehat=} \exists\, Transaction'_0 \bullet \Phi DoSomeBusiness \land NewTransaction_0$$

We can generalise a bit (as we did in the previous section) to a transaction that tells us what account name the transaction is dealing with. Clearly in the single-entry case, since there is one, single account under consideration, this is always the same (which is why it isn't mentioned in real systems since it is pointless and wasteful to record this redundant piece of information).

So we amend $Transaction_0$ to

$$
\begin{array}{|l}
\hline
Transaction_1 \\
date : DATE \\
description : TEXT \\
account : ACCOUNT \\
amount : MONEY \\
\hline
\end{array}
$$

and also add the new given type

$$[ACCOUNT]$$

which causes obvious changes to the above Z as follows:

[2]We have used the convention established by some authors make decades ago of using an initial Φ to name the *framing schema* for the promotion (Φ for F in "Framing", eye-catchingly rendered).

NewTransaction$_1$

$$
\begin{array}{l}
Transaction' \\
date? : DATE \\
description? : TEXT \\
account? : ACCOUNT \\
amount? : MONEY \\
\hline
date' = date? \\
description' = description? \\
account' = account? \\
amount' = amount?
\end{array}
$$

$$DoSomeBusiness_1 \mathrel{\widehat=} \exists\, Transaction'_1 \bullet \Phi DoSomeBusiness \land NewTransaction_1$$

Finally, again as in the previous section, we introduce the separation of *amount* to give

Transaction

$$
\begin{array}{l}
date : DATE \\
description : TEXT \\
account : ACCOUNT \\
income : MONEY \\
expenditure : MONEY
\end{array}
$$

which flows on to give

NewTransaction

$$
\begin{array}{l}
Transaction' \\
date? : DATE \\
description? : TEXT \\
account? : ACCOUNT \\
income? : MONEY \\
expenditure? : MONEY \\
\hline
date' = date? \\
description' = description? \\
account' = account? \\
income' = income? \\
expenditure' = expenditure?
\end{array}
$$

$$DoSomeBusiness \mathrel{\widehat=} \exists\, Transaction' \bullet \Phi DoSomeBusiness \land NewTransaction$$

and *SingleState* has the obvious definition in terms of this final form for *Transaction*.

(Predicates we might want...keep track of totals amount in and out to make sure it's never below zero? Constrain *account* to always be the same (our only) account name?

Also operations like "what is my total balance?" and "compile a report for the month's income and expenses, in summary".)

2.2. **Double-entry accounting.** Double—some schema calculus with the single version
But also, some refinement.

It looks to me like single can be refined to double.

Very clear from the tables in the first section. The first transaction in the single system

Date	Description	Account	Income	Expenditure
1st January 2017	Sold Tissot watch	Capital	$250.00	

can first be rewritten as

Date	Description	Account	Income	Expenditure
1st January 2017	Sold Tissot watch	Capital		$250.00
		Capital	$250.00	

which is simply related to

Date	Description	Account	Debit	Credit
1st January 2017	Sold Tissot watch	Capital		$250.00
		Capital	$250.00	

We aim to formalise the double-entry system in the same way as we have done for single-entry, and then show (as the tables above suggest) that every single-entry transaction and be made more concrete (refined to) a double-entry one. The various operations will follow the same pattern and the usual Z refinement properties will be proved.

First we define what a double-entry transaction looks like and, as before, what a double-entry state is. And we need to know that their is a special sort of money amount, the blank sort.

$$blank : MONEY$$

DoubleTransaction

$$ddate : DATE$$
$$ddescription : TEXT$$
$$account1, account2 : ACCOUNT$$
$$debit1, debit2, credit1, credit2 : MONEY$$

$$(debit1 = credit2 \land debit2 = credit1 = blank)$$
$$\lor$$
$$(debit2 = credit1 \land debit2 = credit1 = blank)$$

$$DoubleState \mathrel{\widehat=} [daccount : \operatorname{seq} DoubleTransaction]$$

And similarly to the single-entry case we also have

InitDoubleState

$$DoubleState$$

$$daccount = <>$$

DoubleNewTransaction

$$DoubleTransaction'$$
$$ddate? : DATE$$
$$ddescription? : TEXT$$
$$daccount1?, daacount2? : ACCOUNT$$
$$debit1?, debit2? : MONEY$$
$$credit1?, credit2? : MONEY$$

$$ddate' = ddate?$$
$$ddescription' = ddescription?$$
$$daccount1' = daccount1?$$
$$daccount2' = daccount2?$$
$$debit1' = debit1?$$
$$debit2' = debit2?$$
$$credit1' = credit1?$$
$$credit2' = credit2?$$

$\Phi DoSomeDoubleBusiness$

$\Delta DoubleState$
$DoubleTransaction'$

$daccount' = daccount^\frown < \theta DoubleTransaction' >$

$DoSomeDoubleBusiness \mathrel{\widehat=} \exists\, DoubleTransaction' \bullet \Phi DoSomeDoubleBusiness \land DoubleNewTransaction$

Now, since we have different state space we need to invoke the rules for data refinement (in a Z setting) and to do that we need to say how the two state spaces are related. Here, at last, is where we formalise the picture suggested by the tables above.

We have the schema R which expresses the relationship between the two state spaces:

R

$SingleState$
$DoubleState$

$\#saccount = \#daccount$
$\forall\, i : \mathrm{dom}\, account \bullet (saccount(i).date = daccount(i).ddate \land$
$saccount(i).description = daccount(i).ddescription \land$
$saccount(i).account = daccount(i).account1 = daccount(i).account2 \land$
$saccount(i).income = daccount(i).debit2 \land$
$saccount(i).expenditure = daccount(i).credit1)$

Then we need to show that the refinement rules hold for any single-entry operation and the corresponding double-entry operation. The rules are standard (see Derrick and Boiten, for example) and state that for any two state spaces $AState$ and $CState$ connected by a relation given by a schema R, for any corresponding operation schemas AOp and COp the following must hold:

Applicability:

$\forall\, AState;\ CState \bullet \mathrm{pre}\ AOp \land R \Rightarrow \mathrm{pre}\ COp$

Correctness:

$\forall\, AState;\ CState;\ CState' \bullet \mathrm{pre}\ AOp \land R \land COp \Rightarrow \exists\, AState' \bullet R' \land AOp$

and also that there must be corresponding initial states

Init

$\forall\, CState' \bullet CInit' \Rightarrow \exists\, AState' \bullet AInit' \land R'$

In our case, if we take the single-entry system to be the abstract state, the double-entry to be the concrete and first consider the appropriate initialisation schemas, we have to prove:

$\forall\, DoubleState' \bullet InitDoubleState' \Rightarrow \exists\, SingleState' \bullet InitSingleState' \land R'$

Unpacking definitions gives the obligation:

$$\forall\, daccount' : \text{seq}\, DoubleTransaction \bullet daccount' = <> \Rightarrow \\ \exists\, saccount' : \text{seq}\, SingleTransaction' \bullet saccount' = <>\ and \wedge R'$$

We can use the one-point rule on both

$$\forall\, daccount' : \text{seq}\, DoubleTransaction \bullet daccount' = <> \Rightarrow \ldots$$

and

$$\exists\, saccount' : \text{seq}\, SingleTransaction' \bullet saccount' = <>\ and \wedge R'$$

to get

$$\begin{array}{|l} SingleState' \\ DoubleState' \\ \hline \# <> = \# <> \\ \forall\, i : \text{dom} <> \bullet\ (saccount'(i).date = daccount'(i).ddate \wedge \\ saccount'(i).description = daccount'(i).ddescription \wedge \\ saccount'(i).account = daccount'(i).account1 = daccount'(i).account2 \wedge \\ saccount'(i).income = daccount'(i).debit2 \wedge \\ saccount'(i).expenditure = daccount'(i).credit1) \end{array}$$

and here the first line is clearly true, and the large universally-quantified predicate is vacuously true. So, initialisation for our system holds.

Next we consider the operations *DoSomeBusiness* and *DoSomeDoubleBusiness*, which give rise to the obligations of having to prove:
Applicability:

$$\forall\, AState;\ CState \bullet \text{pre}\ AOp \wedge R \Rightarrow \text{pre}\ COp$$

Correctness:

$$\forall\, AState;\ CState;\ CState' \bullet \text{pre}\ AOp \wedge R \wedge COp \Rightarrow \exists\, AState' \bullet R' \wedge AOp$$

which by substitution specialise to:
Applicability:

$$\forall\, SingleState;\ DoubleState \bullet \text{pre}\ DoSomeBusiness \wedge R \Rightarrow \text{pre}\ DoSomeDoubleBusiness$$

Correctness:

$$\forall\, SingleState;\ DoubleState;\ DoubleState' \bullet \text{pre}\ DoSOmeBusiness \wedge R \wedge DoSomeDoubleBusiness \Rightarrow \exists\, S$$

We consider Applicability first:

2.3. **Triple-entry accounting (aka Momentum accounting).** Triple—and again?

2.4. **UTXO accounting.** For a UTXO model, an account's balance is not explicitly stored anywhere (officially; it might be kept track of in the "wallet" that someone keeps, but that is not part of the blockchain).

We start with some basic constants and types.

The cash that gets moved around in a UTXO system is not fixed in size (so when we want to try out this model we will probably concretise $CASH$ by $\mathbb{N}$, but for now we stay most abstract with a given type. The same goes for $ACCOUNTNO$ and $TRANSACTIONNAME$.

Note that the mechanisms for secure account use (spending, mainly) are abstracted (hidden) in this version.

Any inputs to a transaction indicate where value is being taken from: some previous transaction (identified by its name) , and which of its outputs (identified by who is spending their money). And outputs are identified here by the person who owns them, how much they are worth and in which transaction they were given the money:

$$TransInput \mathrel{\widehat=} [transaction : TRANSACTIONNAME, spender : ACCOUNTNO]$$

Something to prove here is that an input only refers to a previous transaction.

Any outputs from a transaction say how much cash they contain and who owns it

$$TransOutput \mathrel{\widehat=} [name : TRANSACTIONNAME, cash : CASH, owner : ACCOUNTNO]$$

(I guess we don't need to show that the owner is a current live account? There are accounts with no live owners.)

We are representing a transaction by how much is being transferred from where (which inputs) and to where (which outputs), together with its name. So we have:

Transaction

$$\begin{array}{l} name : TRANSACTIONNAME \\ inputs : \mathbb{P}\ TransInput \\ outputs : \mathbb{P}\ TransOutput \\ \hline \forall\, o : outputs \bullet o.name = name \end{array}$$

Finally, this is the state of the blockchain abstracted to just show all the accepted, recorded transactions:

$$BCstatetype == \text{seq}\ Transaction$$

The outputs of a transaction form a set (transactions only output one coin to each account that they pay to, so if there are equal amounts of cash they are distinguishable by the owner's account number).

Also an owner might have, say, two coins with the same cash value. But they will each "reside" in different transactions' outputs (transactions only output one coin to each account that they pay to), so they remain distinguishable.

When it comes to identifying all the unspent coins at a certain point in the history of a blockchain, what then? At any given point in the evolution of the blockchain we have a list of valid, accepted (by consensus) transactions. At this point, the unspent coins are the unspent outputs in any transaction that has appeared so far. An output is unspent if it has not been used as an input to some subsequent transaction. Now consider the

smallest blockchain, or list of transactions: it has just one block and all the coins in its outputs are unspent coins (and note that this starting block breaches the value inout equals value output requirement, so it must be special, of course).

Now consider adding a transaction: the inputs to this transaction must be amongst those output by the first, special, transaction, and after this second transaction the unspent coins are the outputs from the first transaction, (set) minus the coins input to the second transaction (which must be amongst those output by the first transaction), unioned with the coins output from the second transaction (none of which can have been used yet since this is the last transaction in the list, i.e. on the blockchain.)

We can see that, by induction, as a generalisation of this story, the unspent output for a list of transactions of the form $h : t$ is the unspent outputs from the tail (list) t, (set) minus the coins input to the (head) transaction h, unioned with the outputs of h. And to start the inductive definition the unspent outputs of the initial blockchain are all the outputs of that first transaction. To summarise, we define:

$$unspentOutputs\ [h] =_{def} h.outputs$$
$$unspentOutputs\ (h : t) =_{def} unspentOutputs(t) \setminus ItoO(h, t)\ \cup h.outputs$$

Note that we cannot simply use $h.inputs$ as the second term in the second right-hand side above (consider the types!), so we need the function $ItoO$ to map $h.inputs$ to the outputs within t that they are each formed from. We have:

$$ItoO(h, []) =_{def} \emptyset$$
$$ItoO(h, (h' : t)) =_{def} ItoO(h, t) \cup \bigcup_{i \in h.inputs} \{ c : CASH \mid i.transaction = h'.name \wedge \exists\, o' : TransOutput \mid o' \in h'.outputs \bullet i.spender = o'.owner \wedge o'.cash = c \bullet \langle\!| name \rightsquigarrow h'.name, cash \rightsquigarrow c, owner \rightsquigarrow h.spender |\!\rangle \}$$

A transaction in UTXO is *valid* as long as the cash input and the cash output are the same (remember we are hiding anything to do with the *security* of account use for now; not to do so would mean also requiring that the sender can prove ownership of the coin that is being sent—orthogonal, and perhaps something to do later), and all the inputs in the current transaction were amongst the outputs of previously occurring valid transactions.

Here are the two components for validity for any blockchain $bc : BCstatetype$—the first is the equality of inputs and outputs for any transaction and the second says that all the inputs of a transaction must be unspent outputs from previous transactions:

$$\forall\, trans : Transaction;\ i : \mathbb{N} \mid i \mapsto trans \in bc \bullet sum\_out\ trans.outputs = sum\_in\ trans.inputs$$

where *sum_out* sums all the cash values in the outputs, and *sum_in* sums all the cash values "in" the inputs by referring back to the output that input is formed from in each case.

$$\forall\, trans : Transaction \mid i \mapsto trans \in bc \bullet \forall\, input : TransInput \mid input \in trans.inputs \bullet input \in unspentOutputs((0..i) \lhd bc)$$

2.5. **Ethereum-style contracts and accounting.** To come.