Anticipation Flowing Backwards in a Functional Monetary Economics Simulation

Pierre Boudes, Antoine Kaszczyc and Luc Pellissier

{boudes, kaszczyc, pellissier}@lipn.univ-paris13.fr
Université Paris 13, Laboratoire d'Informatique de Paris-Nord,
93430 Villetaneuse – France

Abstract

In [2], we presented a SFC-safe framework for programming monetary economics simulations, which we succesfully tested on top of Pascal Seppecher's Jamel. We present here a functional implementation of a macro-economics simulator. Interaction is mediated through constraints, accounting for agent's anticipation.

The dynamics of constraints is handled in the background by the simulation itself, while the economist-programmer is provided a convenient domain-specific language for programming behavioural bricks. We present an implementation of such a system in the Scala language.

1 Safe programming of monetary economics simulations

As economics simulation grow in size and complexity, it gets increasingly important to have tools discharging the programmer-economist from any burden whose handling can be automatized. In particular, we are interested in increasing compositionality and modularity of functions and sections of programs. Indeed, a solution to a problem (be it economic or otherwise) that does not scale can not be considered satisfactory. Compositionnality offers a path to scaling: the possibility of dissecting a problem into smaller components, each one being able to be solved easily, and then recomposed onto the original problem is of paramount importance. Functionnal programming is a progamming paradigm that aims at providing compositionnality, by isolating the internal behaviour of functions from the effects they can have, thus enable simpler composition. We advocate the financial system can be seen as a side effect of the real economy (and thus that the financial and real spheres of the economy, distinct but correlated, have to be treated as such in simulations).

The noble goal of programming in such a way can be complicated for the programmer: in particular, isolating functions from the financial system creates difficulties in interacting with one person's account, for instance for checking it. It creates important difficulties for managing anticipations. Our vision would be pretty useless if it imposed to big a burden on the economist-programmer. We present here a solution to the management of expectations in the framework of functional stock-flow consistent programming.

In a previous work [2], we presented a framework handling the Stock-Flow Consistency for the programmer. Let us review the main points of this framework.

Functional programming is a paradigm really close to mathematical practice: once defined, an object never changes value; functions have first-class status and can be passed as arguments to higher-order functions; the output of the application of a function to an input only depends on the argument, and not on hidden parameters such as a global

state. As such, it allows both for an ease of translating specifications (often expressed mathematically) into a program and of reasonning about the program.

Strongly-typed functional programming adds another aspect, which we will use heavily: from a static analysis of the code, it is possible to infer a specific kind of invariant of a program, its type. Much can be recovered from the type of a program in a strongly-typed setting [3]. Consider, for instance, an HASKELL program of the type $f : \forall \alpha, [\alpha] \rightarrow \textbf{int}$.

Its meaning is that f can take as an input a list of elements of any type (but homogeneous: every element of the list must be of the same, unknown *a priori* type) and returns an integer. It can be proved that such a function can not look at the elements of the list (as f should be the same for every type α) and so can only be a function on the length of the list and not its content.

As this example shows, types encode invariants of a program. An interesting invariant in monetary economics simulation is Stock-Flow Consistency [1]: the fact that every monetary transaction has an exact counterpart. In our previous work, we presented a couple of type constructions expressing that a program typed with these constructions is SFC.

1.1 The SFC monadic duet

In order to make possible for the type system to know of the financial system, we first introduce some abstract types:

`Id`. The `Id` type is a type of identifiers, that identifies an account and its owner (an owner may have multiple accounts).

`Amount`. The `Amount` type is just an abstract numerical type, expressing any amount of money.

`Accounts`. The `Accounts` type is a type of key-value map: each key (of type `Id`) is mapped to a value (of type `Amount`). It can be thought of as a dictionnary, an array or a list.

`Constraints` and `Satisfactions`. The `Constraints` and `Satisfactions` types are implemented like `Accounts`. A constraint aims to represent a required amount on the account with the corresponding id and a satisfaction as to be though as computed as the difference between the real amount on the account and the amount required by a certain constraint.

`Transaction`. The `Transaction` type contains at least triples `Id` $\times$ `Amount` $\times$ `Id`.

It is straightforward to define addition, maximum, minimum, extensional order on `Accounts`, `Constraints` or `Satisfactions`. We use a special difference $\ominus$ and addition $\oplus$, defined only where their left-hand-side operand are defined (example Figure 1). Given a list of transactions we can compute the balance it implies on the accounts and also the lowest amount each account reaches while firing the transactions in the order of the list. See Figure 1.

Functions inside the simulator are of the type:

$$\alpha \rightsquigarrow \beta \stackrel{\text{def}}{=} (\alpha \times (\texttt{Constraints} \rightarrow \texttt{Satisfactions})) \rightarrow \beta \times (\texttt{Transaction List})$$

We will use an example function of this type, named f. You can see that this type is a function. Let us start with the left part of the type,
$\alpha \times (Constraints \rightarrow Satisfactions)$. These are the two arguments that f will receive. The α is the value, like the simulation objects (firms, ...). The $Constraints \rightarrow Satisfactions$

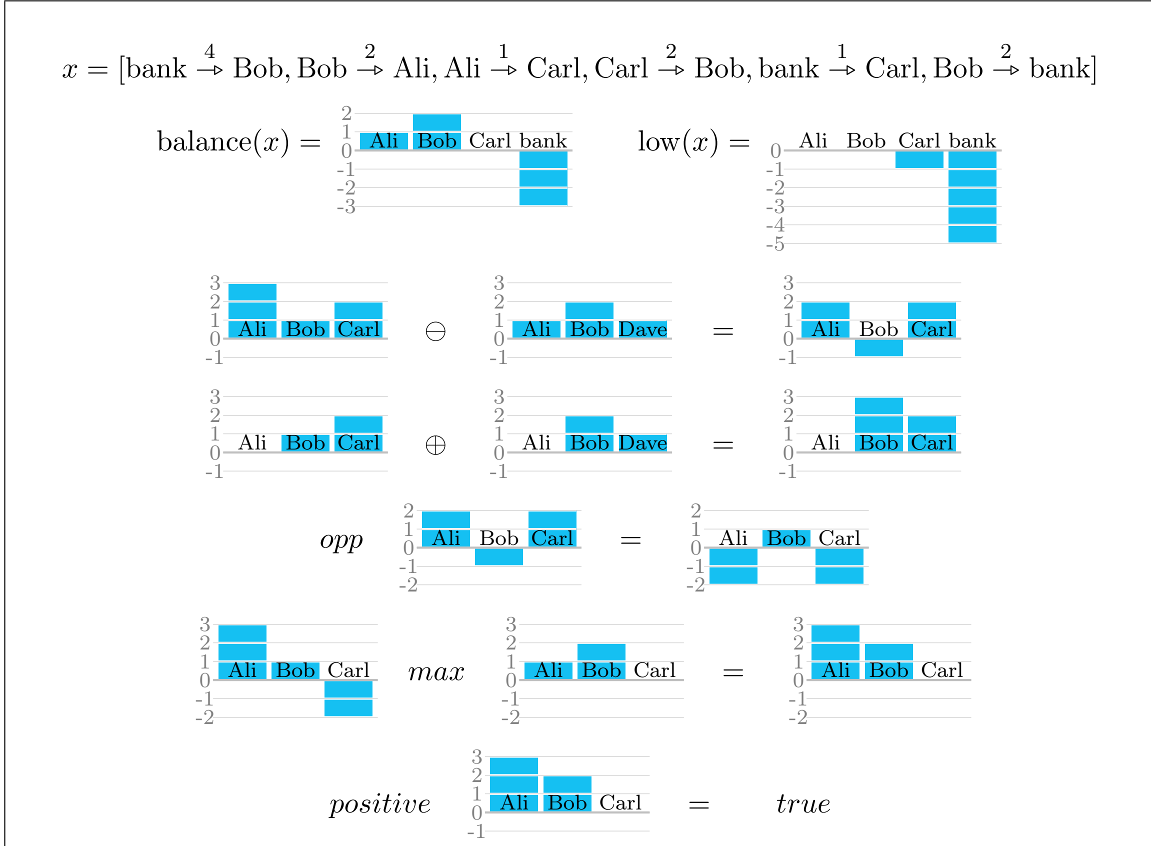

Figure 1: Operations on accounts and operations balance and low

part is the way for f to obtain the information about bank accounts. f will give some constraints about some accounts, and will receive in exchange the satisfactions. For the right part of the type, $\beta \times TransactionsList$, it is what f will return. The β is a value optionally different from α. The transactions list is the financial action of f, expressed as a list of transactions.

This programming style ensures that a function programmed this way is SFC, through two different observations:

- a function must give a financial constraint in order to access its input. If the constraint is not fulfilled, it will not receive it, or receive a fallback value;
- a function does not access the financial system by itself, but produces a list of transactions for the financial system to execute. A list of transaction guarantees consistency by definition.

Before going on with this type, let us introduce Jamel, Pascal Seppecher's monetary economics simulator, as we are going to use it in an example.

2 One Jamel period

Many simulators are structured as a sequence of periods. In a functional style, a *period* is a function from a state in input to updated state in output. In another, more imperative, style, it can be a procedure which updates some state stored somewhere. In this section we describe the period of the simulator *Jamel*.

Jamel defines a state composed of three types of agents:

- a unique *bank*, which lends money, earns interests for loans and shares dividends with its owner (an household);
- several *firms*, which hire workforce, pay wages, sell goods, borrow and pay back money and share dividends with their owners (some households);
- several *households*, which earn wages, work for firms and buy goods.

A period is composed of many sequential steps, among which one where firms plan production and one on which they pay the wages. In many of these steps, an agent receives information about its environment (its account current balance, the markets) and reacts to it, sometimes also with a memory of the past. The agent often has to forecast and anticipate for a step in the future. It is the case with the two steps "firms plan production" and "firms pay the wages". The firm has to compute how much money is needed to pay the wages. In a future step, it has to pay the wages. On a concrete example:

- firm A plans production: it computes the sum of wages it will have to pay this month (10×100 units); checks the amount of its bank account (800 units); asks the bank for a loan (200 units).
- firm A do other things, such as hiring workers
- firm A pays wages: it computes the sum of wages it has to pay (10×100 units); and pays the wages (1000 units).

The computation during the firm forecast and the one happening really at a later point are the same. In a sense, the firm projects itself onto the future when forecasting, computes some constraints on its state in the present, and deduces from it actions that must be undertaken in the present. It might be interesting to fuse the two steps of forecasting and of actually acting by exploring mechanisms allowing to have some limited control of the past from the future (allowing for this sort of constraints flowing from the future to the past).

In the next section we present the composition which will allow information from the future to flow back to the past and allow functions to prepare it. The explanation will be accompanied with an example directly inspired from the "borrow and pay wages" one of Jamel.

3 Composition

We defined in a previous section the type and the behaviour of one function. Now the most important thing to do is define the composition of two functions. Recall that a fonctional simulator is built as a composition of functions.

The goal is simple : we dispose of two functions, f of type $\alpha \rightsquigarrow \beta$ (see the previous section about this type) and g of type $\beta \rightsquigarrow \gamma$. What we want to define is method to obtain a function h of type $\alpha \rightsquigarrow \gamma$ as a combination of f and g.

We illustrate the composition with Figure 2, an example with two functions : `borrow` that borrow if the amount on the account is negative and `payWages` that spend money if there are some. We compose `borrow` with `payWages` and the goal is that `borrow` eventually learn about `payWages` constraints and borrow money for it. The type of both functions are $Sim \rightsquigarrow Sim$ or $(Sim \times (Constraints \rightarrow Satisfactions)) \rightarrow (Sim \times TransactionList)$. `Sim` contains objects of the simulation, here a bank, a firm, some workers and some loans.

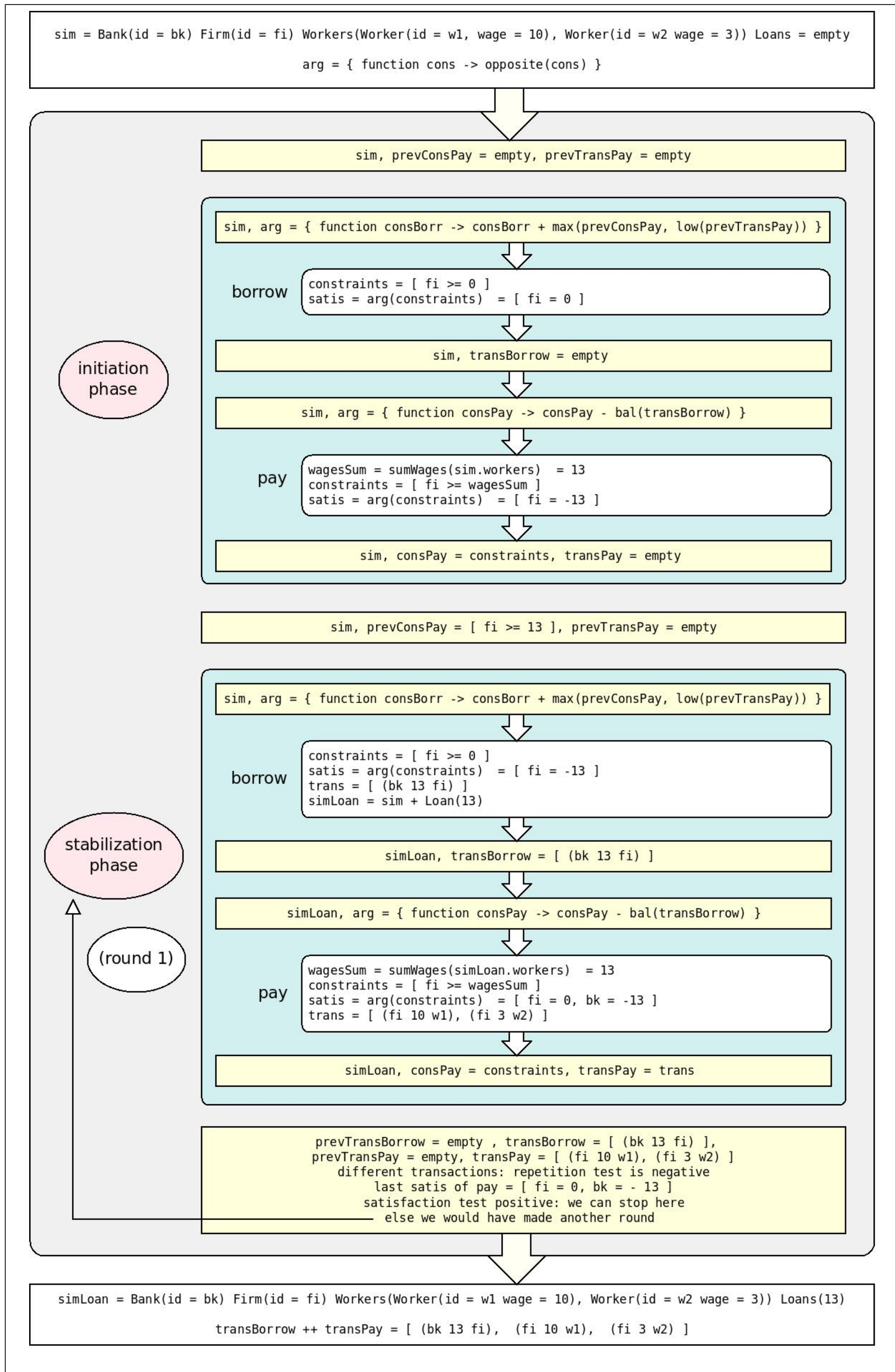

Figure 2: Diagram of the composition of *borrow* and *payWages*.

3.1 Initiation phase

We want the constraints and transactions of `payWages` to flow backward to `borrow`, so it can have a vision of its 'future' and make actions to prepare it. Of course, to know the actions of `payWages`, we need to execute it once, and for this we need to execute `borrow` once because `payWages` needs the `Sim` that `borrow` produces. So, the composition always starts by the initiation phase, where we execute `borrow` then `payWages`.

3.2 Stabilization phase

Once we got the previous constraints of `payWages`, we replay `borrow`. To grant `borrow` with the ability of knowing about `payWages` constraints, we are going to interfere in its call of satisfaction. Indeed we modify the constraints of `borrow` by adding the ones of `payWages`. In this way the constraint grow and the satisfaction become negative. Then `borrow` know it has to make some loan.

This is a stabilization phase because we replay it until `payWages` obtain a positive satisfaction. In this case it means that `borrow` successfully provided the money asked by `payWages` and the composition has achieved its purpose.

3.3 A bit more complicated

As you can see in Figure 2 things are more complicated. To have a correct composition, more information has to be shared.

First, `borrow` has to know about the constraints of `payWages` but also about its transactions. To achieve this, what is added to the contraints of `borrow` is the *max* of [contraints of `payWages` and the *low* of the transactions of `payWages`]. We are using the *max* because we do not want these two informations to be duplicated. We are using *low* on the transactions to obtain the constraints they express.

Then, `payWages` should know about `borrow` transactions. Unless, it would never know when `borrow` have lend some money. This is achieved by *minus* the constraints of `payWages` by the *bal* of `borrow`'s transactions. By reducing the constraints, we are raising the satisfaction, so `payWages` will be able to spend money. We use *bal* here and not *low* because we do not need the constraints but the resulting amounts of money on the accounts.

3.4 End phase

When we reach satisfaction (or when the transactions become constants), we can stop the composition and give the result. Of course, the result value is the last `Sim` given by `payWages` and the concatenation of the last transactions of `borrow` and `payWages`.

4 Implementation and usability

The system we described is powerful and removes a lot of unduly burden from the shoulder of the programmer: in particular, nothing relating to the management of the financial system or the way agents project themselves in the future is to be handled by the programmer herself. All that needs to be written are small blocks handling atomic operations (such as the wages payment, the decision to work or not for a household,...) and global information about the simulation (which operations are chained? In which order? Is there a notion of periods? If so, how do they compose?...).

We propose a small Domain-Specific Language suited for describing such global information as well as atomic functions, meant to be composed. The control-flow of each atomic

```
val payWages = (sim:Sim) => {
    val fid = sim.firm.id
    val sumWages = sim.firm.workers.map(_.wage).sum
    withConstraints (fid -> sumWages) (
       { satisfied => make =>
           sim.firms.workers foreach (w => make order (fid, w.wage, w.id))
       },
       { unsatisfied => make => make nothing })
}
```

Figure 3: the function "payWages" written with the DSL

```
val borrow = (sim:Sim) => {
    val fid = sim.firm.id
    withConstraints (fid -> 0) (
        { satisfied => make => make nothing },
        { unsatisfied => make =>
            val loanAmount = (-1) * unsatisfied(fid)
            make loan (fid, loan) })
}
```

Figure 4: the function "borrow" written with the DSL

function is really simple. The function declares a list of minimum amounts on some accounts. They are the wishes of the function, it is telling to anterior functions that it would be best to have these amounts. Take for example Figure 3 where the function `payWages` meant for paying wages is written. Observe that the function first computes the sum of the wages, and then declare that it would be best if the firm have this amount on its bank account, with the keyword `withConstraints`. After the wishes comes the actual results and reactions. There are two possibilities: either the constraints are satisfied, either they are not. You can see there is indeed two corresponding constructs on the figure, `satisfied` and `unsatisfied`. The (un)satisfaction received is the difference between the constraints and the actual amounts on accounts. In the case of `payWages`, a satisfied constraint means there is enough on the firm's account to pay the wages. So for each worker in the firm, the function order a transaction by using the DSL provided utility `make order`. In the other hand, an unsatisfied constraint would mean that the firm has not enough money, so `payWages` cannot pay the workers. It is not definitive as the composition will often need several steps to stabilize and give satisfaction to `payWages`.

The other function `borrow` at Figure 4 is very similar, except it reacts on an unsatisfied constraint. There is also the code `unsatisfied(fid)`, which reflects the fact that the satisfaction is indeed a `Map` of `Id` to `Amount` and that we can get the amount of the firm by giving the firm's id. Now that we have seen how to describe atomics functions using the DSL, what remains is the combination of these little parts. All that is needed is the ordered sequence of atomic parts. To complete our example, the following code permits to sequence (compose) the two atomics `borrow` and `payWages`.

```
val period = new PeriodBuilder() + borrow + payWages
```

The DSL is implemented as a small collection of functions written in the Scala language. It is a good trade-off between a perfect syntax for the DSL and the full power of a complete

programming language. It is worth nothing that it is fully compatible with Java, the language used in Jamel.

References

[1] W. Godley and M. Lavoie. *Monetary Economics: An Integrated Approach to Credit, Money, Income, Production and Wealth.* Palgrave Macmillan, 2007.

[2] Antoine Kaszczyc Pierre Boudes and Luc Pellissier. Monetary economics simulation: Stock-flow consistent invariance, monadic style. In *Artificial Economics 2015.*

[3] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.