It's promises (almost) all the way down...

Daniel Tang, David Pugh, J. Doyne Famer

June 5, 2015

1 Motivation

Roughly speaking our work is motivated by two key ideas:[1]

1. "All economic agents are banks" - Hyman Minksy
2. "All interactions between banks involve a swap of IOUs" - Perry Mehrling

We have designed our low-level API around a slight translation of the above ideas:

1. each economic `Actor` is a "bank";
2. each interaction between economic `Actors` involves a `transfer` of a `Promise`.

This immediately raises the following questions:

1. What do we mean by an economic `Actor`?
2. What we we mean when we say that all economic `Actors` are "banks"?
3. What is a `Promise`?
4. What does it mean to `transfer` a `Promise`?

2 Actor

In this section we define the low-level API for an economic `Actor` and define what it means to say that each economic `Actor` is a "bank."

1. each actor has a balance sheet containing assets and liabilities;
2. each actor faces a sequence of cash-flow constraints.

[1] Don't forget to cite work by John G. and various papers by Kiyotaki and Moore (possibly others as well)!

3 In the beginning...

3.1 Defining a `Good`

3.2 `Actions` over `Goods`

4 Promises, promises...

Borrowed title of this section from 1997 paper by John G. John G. defines an asset as a promise together with some collateral.

4.1 Definition of a `Promise`

A `Promise` represents a commitment between a `promisor` and some `promisee` to undertake certain `actions`, potentially involving both `goods` and additional `promises`, specified by some `Sentence` `when` a certain `StateofAffairs` has occurred.

These considerations lead to the following low-level interface for the `Promise` trait.

```
trait Promise {

  def promisor: ActorRef

  def promisee: ActorRef

  def actions: Sentence

  def when: StateOfAffairs

}
```

Want to allow for the possibility that some promises can not be given (and therefore not transferred or exchanged) from one `Actor` to another `Actor`. To capture this feature we define a separate `GiveablePromise` trait.

```
trait GiveablePromise extends Promise {

  val isGiveable: Boolean

}
```

4.2 Actions over Promises

4.2.1 Uni-lateral Actions

An `Actor` i can *unilaterally* decide to perform any of the following `Actions` with a `Promise`.

- `create` a new `Promise`: When an `Actor` creates a new `Promise`, the `Actor` becomes its `promisor` and the new `Promise` becomes a liability for that `Actor`. Similarly, the new `Promise` becomes an asset for which ever `Actor` is the `promisee`.
- `accept` a `Promise`: If an `Actor` i chooses to `accept` a `Promise` from another `Actor` j, then the `Promise` is added as an asset to the balance sheet of `Actor` i and as a liability to the balance sheet of `Actor` j.
- `reject` a `Promise`: An `Actor` i can always choose to `reject` (i.e., *not* `accept`) a `Promise` from another `Actor` j. Rejected promises are not added to balance sheets.
- `fulfill` a `Promise`: An `Actor` i who is the `promisor` of a `Promise` may choose to perform `actions` specified in the `Sentence` `when` a certain `StateOfAffairs` has occurred. Once a `Promise` is successfully fulfilled, it is removed from both the balance sheet of its `promisor` and the balance sheet of its `promisee`.
- `break` a `Promise`: An `Actor` i who is the `promisor` of a `Promise` may choose *not* to perform `actions` specified in the `Sentence` `when` a certain `StateOfAffairs` has occurred. A decision to `break` a `Promise` is the same as a decision *not* to `fulfill` a `Promise`.[2]
- `destroy` a `Promise`: An `Actor` i who is the `promisee` of a `Promise` may choose to `destroy` that `Promise` prior to a certain `StateOfAffairs` occurring. Once destroyed a `Promise` is removed from both the balance sheet of the `promisee` *and* the balance sheet of the `promisor`.
- `give` a `Promise`: An `Actor` i who is the `promisee` of a `Promise` may choose to `give` that `Promise` to another `Actor` j.
- `redeem` a `Promise`: An `Actor` i who is the `promisee` of a `Promise` may choose to `redeem` that `Promise` after a certain `StateOfAffairs` has occurred. Redemption of a `Promise` can be thought of as a request that the `promisor` of that `Promise` `fulfill` the `Promise`.

The above actions can be combined into a low-level, `PromiseMaker` trait that extends the `Actor` base trait:

```
trait PromiseMaker extends Actor {

  def create(promisee: ActorRef,
```

[2]Breaking a `Promise` may or may not have consequences. Should consequences be specified in the original `Promise`? I think so.

```
              actions: Sentence,
              when: StateOfAffairs): Promise

  def destroy(promise: Promise): Unit

  def accept(promise: Promise): Unit

  def reject(promise: Promise): Unit

  def fulfill(promise: Promise): Unit

  def break(promise: Promise): Unit

  def give(promise: Promise, other: ActorRef): Unit

  def redeem(promise: Promise): Unit

}
```

4.2.2 Cooperative `Actions`

Two `Actors` i and j can cooperate *bi-laterally* to perform additional actions over `Promises`:

- `transfer` a `Promise`: An `Actor` i who is the `promisee` of the `Promise` can `transfer` a `Promise` to `Actor` j as follows:
  1. `Actor` i `give` `Promise` to `Actor` j
  2. `Actor` j `accept` `Promise` from `Actor` i
- `exchange` of `Promises`: Two `Actors` i and j who are the `promisees` of different `Promises` can `exchange` these `Promises` with one another as follows:
  1. `Actor` j `create` new `Promise` {`give` existing `Promise` to `Actor` i}.
  2. `Actor` j `transfer` new `Promise` to `Actor` i.
  3. `Actor` i `transfer` existing `Promise` to `Actor` j
  4. `Actor` j `fulfill` `Promise` to `Actor` i

A few things are worth noting about a bi-lateral `exchange`. First, by choosing to `fulfill` the new `Promise` in step 4, `Actor` j gives his existing `Promise` to `Actor` i (which completes the `exchange`). Second, the new `Promise` issued by `Actor` j in step 1 involved a promise to `give` and *not* a promise to `transfer` an existing `Promise` to `Actor` i.[3] Finally, note that an `exchange` could take place with `Actor` i creating the new `Promise` in step 1. The important point is that either `Actor` i or `Actor` j (not necessarily both) must be able to *credibly* commit to `give` its `Promise` upon receipt of the other's `Promise`.[4]

[3]Should it be possible to for an `Actor` to create a `Promise` that commits *other* `Actors` to perform actions? Do we have any real world examples?.

[4]The credibility of any particular `Promise` should be endogenously determined within the model and *not* imposed by us *a priori*

It is interesting to compare the above bi-lateral exchange mechanism with a multi-lateral exchange mechanism involving cooperation between three `Actors` i, j, and k. Two `Actors` i, j who are the `promisees` of different `Promises` can `exchange` these `Promises` using `Actor` k as an intermediary as follows:

1. `Actor` k `create` new `Promise` {`give Actor` j `Promise` to `Actor` i}.
2. `Actor` k `create` new `Promise` {`give Actor` i `Promise` to `Actor` j}.
3. `Actor` k `transfer` new `Promise` to `Actor` i.
4. `Actor` k `transfer` new `Promise` to `Actor` j.
5. `Actor` i `transfer` existing `Promise` to `Actor` k
6. `Actor` j `transfer` existing `Promise` to `Actor` k
7. `Actor` k `fulfill Promise` to `Actor` i
8. `Actor` k `fulfill Promise` to `Actor` j

An important feature of this multi-lateral process is that, so long as `Actor` k can *credibly* commit to *both* `Actors` i *and* j, then the exchange between i and j can take place even if *neither* `Actor` i *nor* `Actor` j can bi-laterally commit to `give` its `Promise` upon receipt of the other's `Promise`.[5] There are several interpretations of `Actor` k's role in the above process. One interpretation is that `Actor` k is functioning as a central clearing party (CCP) for transactions between other `Actors`; another more institutional interpretation is that `Actor` k is an actual `Market`.

One final cooperative action needs to be specified: `transfer` of a `Promise` that is liability for one `Actor` to some other `Actor`. An `Actor` i who is the `promisor` on a `Promise` can only `transfer` that `Promise` to another `Actor` j with permission from the `promisee` of that `Promise`, `Actor` k.

1. `Actor` i `create` new `Promise` {`give Actor` j existing `Promise` }.
2. `Actor` k `accept` new `Promise`.
3. `Actor` i `fulfill Promise` to `Actor` k.
4. `Actor` j `accept Promise` from `Actor` i.

4.3 A language for `Promises`

Having defined the concepts of a `Good` and a `Promise` as well as sets of `actions` over `Goods` and `Promises` that can be performed by an `Actor` or groups of `Actors` to complete the API we need to define a language (grammar?) for building `Sentences` that describe valid `Promises`.

[5]The difference between multi-lateral and bi-lateral commitment has been stressed by many monetary theorists, in particular Kiyotaki and Moore in a series of papers.

4.3.1 Examples

Need to build a catalogue of examples demonstrating how to build common contracts using our language.

5 `Markets` API

1. Must be able to add/remove routees from the router.
2. Must be able to handle situations when market has no buyers (sellers).

The low-level `PromiseMaker` API already incorporates markets in the sense that an individual `PromiseMaker` can act is an intermediary between other `PromiseMakers` in order to facilitate the exchange of `Promises`. In this instance a `PromiseMaker` is acting as a broker or "market maker" and is functionally equivalent to a market institution.

However in order to streamline the implementation of market institutions in actual models we have designed a separate `Market` API on top of the low-level `PromiseMaker` API. The design of the `Market` API is based around the notion that a market needs to have mechanisms for:

1. matching `Promises` made by "buyers" with `Promises` made by "sellers;"
2. clearing (i.e., finding an appropriate price and quantity) for each matched set of `Promises`.

```
trait Market extends PromiseMaker {

  val router: akka.routing.Router

}
```

5.1 Routing

Note that each `Market` has a `Router`.

`MarketParticipants` can easily be added (removed) as `Routees` to a `Router` by sending `AddRoutee` or `RemoveRoutee` messages.

5.1.1 Routing logic

5.2 TODO:

- Router will need to have a `DeathWatch` on all its `Routees`.

5.3 `MarketParticipant` API

Since not all `Actors` in the model will need to participate in markets, seems a good idea to have a separate `MarketParticipant` trait that must be implemented in order for an `Actor` to participate in market transactions.

```
trait MarketParticipant {
  ???
}
```

We implement separate interfaces for each of these functions: a `MatchingEngine` interface for generating matched sets of `Promises` and a `ClearingEngine` interface for finding an appropriate price and quantity for each matched set of `Promises`.

```
trait MatchingEngine {
  // details to be implemented
}

trait ClearingEngine {
  // details to be implemented
}
```

Having separate interfaces for matching and clearing improves the clarity and maintainability of the code base while allowing for greater flexibility in the modeling of markets.