Agent-based and the neoclassical growth models

Domain Specific Languages for Economical and Environmental Modelling
Gothenburg- June 16-17, 2011

Outline

The neoclassical growth model: Solow

- A production function f :

$$y_t = f(l_t, k_t),$$

where $y_t \in \mathbb{R}_+$ is output, $l_t \in \mathbb{R}_+$ is labor, and $k_t \in \mathbb{R}_+$ is capital at date $t = 0, 1, 2, \cdots$.

- A law of motion for the capital stock:

$$k_{t+1} = (1 - \delta)k_t + \sigma y_t,$$

where $\delta \in]0, 1[$ is the depreciation rate and $\sigma \in]0, 1[$ is the savings rate.

The neoclassical growth model: Solow continued

- h_t assumed constant and normalized to 1 $\Rightarrow$ a dynamical system:

$$k_{t+1} = (1 - \delta)k_t + \sigma F(k_t),$$

  where $F(k_t) = f(1, k_t)$
- Standard assumptions (f homogeneous of degree 1, increasing, concave, and C^2) guarantee convergence to a stable steady state $k^* > 0$ such that:

$$\overline{k} = (1 - \delta)\overline{k} + \sigma F(\overline{k}),$$

RA Equilibrium

- The “micro-founded" neoclassical growth model
- A social planner/ representative agent solves for:

$$\max \sum_{t=0}^{+\infty} (1+\rho)^t u_{RA}(c_t)$$

$$s.t\ k_{t+1} = F(k_t) + (1-\delta)k_t - c_t$$

where $c_t \in \mathbb{R}_+$ denotes consumption, $\rho \in]0,1[$ is the discount rate and u_{RA} the utility function is increasing and concave.

RA Equilibrium

- Under standard assumptions, there exists a unique solution $(c_t^*, k_t^*)_{t \in \mathbb{N}}$ characterized by the path of capital accumulation:

$$h_{RA}(k_t^*) = k_{t+1}^*$$

- And a a stable steady state characterized by k^* such that

$$h_{RA}(k^*) = k^*$$

OLG equilibrium

- At each time $t \geq 0$, an household borns and lives for two periods. It supplies inelastically one unit of labor at date t and solves for

$$\max u_{OLG}(c_t^t, c_{t+1}^t)$$

$$s.t\ p_t c_t + p_{t+1} c_{t+1} \leq w_t$$

- At each time t, the firm maximizes profits:

$$\max p_t(F(k_t) - k_{t+1}) + q_{t+1} k_{t+1} - w_t - q_t k_t$$

- where p_t, w_t, q_t, are prices of date t output, labor and capital

OLG equilibrium

- Under standard assumptions, there exists a unique equilibrium $(c_t^{t,**}, c_t^{t-1,**}, k_t^{**}, p_t^{**}, w_t^{**}, q_t^{**})$, completely characterized by the path of capital accumulation:

$$h_{OLG}(k_t) = k_{t+1}$$

- And a stable steady state characterized by k^{**} such that

$$h_{OLG}(k^{**}) = k^{**}$$

Observational equivalence (Aygari JET 1985)

- The OLG and the representative agent models are observationnaly equivalent if they “lead to identical time paths for aggregate capital, output, consumption, investment, real wage, and the real interest rate.”
- Formally, for any u_{RA} there exists u_{OLG} such that

$$h_{RA} = h_{OLG}$$

  and conversely.
- Under specific assumptions, obervational equivalence holds (Aygari JET 1985).
- “In general, the range of dynamics that could be exhibited by OLG models is much larger than that exhibited by [RA] models and observational equivalence cannot possibly hold” (Aygari JET 1985).

The economy as a complex system

- A large number of interacting units.
- Emergent properties.
- The emergent behavior does not result from the existence of a central controller
- Can RA-type dynamics be seen as emergent properties of economies as complex systems ?

ABM as models of complex systems

- Agent-based computational models are models in which:
  - A multitude of objects interact with each other and with the environment.
  - The objects are autonomous, i.e. there is no central, or “top down” control over their behavior.
  - The outcome of their interaction is numerically computed.

  (M.Richiardi)
- “ACE is the computational study of economic processes modeled as dynamic systems of interacting agents.” (L. Tesfatsion)

Equilibrium as an emergent property ?

Can one build ABM which are observationally equivalent (or observationally imply) the RA neoclassical growth model ?

Outline

Characteristics of ABM

- Heterogeneity
- Autonomy
- Explicit Space (and time)
- Local Interactions
- Bounded Rationality
- Non-Equilibrium Dynamics/ dynamically complete

Primitive types

```
data Good = Capital Int | Labor | Consumption | Money deriving Eq

data AgentKind = Household | CapitalFirm | ConsumptionFirm deriving Eq

type Input = RealFunction Good

type Output = RealFunction Good

type Technology = (Input,Output) -> (Input,Output)

type Date = Int

type Hour = Int

type Time = (Date,Hour)
```

Agents and contracts

```
data Agent = Ag AgentId AgentKind Technology Input Output

data Contract = Ctr AgentId AgentId Good Price Quantity Time Time

type Economy = (Date,[Contract],Array AgentId Agent)
```

- A system of heterogeneous agents
- An agent is a stock, a technology to transform stocks autonomously and an identity which will also serve as position/hour .
- The contracts embed memory, expectations, interactions, networks.

Primitive types

```
step :: Economy -> Economy
step eco = foldr act eco ids where
     ids = map identity (elems (agents eco))

 act :: AgentId -> Economy -> Economy
 act id eco = let
  eco1 = updateFinancialStatus ((agents eco)!id) eco
  eco2 = executePendingContracts ((agents eco1)!id) eco1
  eco3 = updateAndExecuteContracts Money  ((agents eco2)!id) eco2
  eco4 = updateAndExecuteContracts (Capital 1)  ((agents eco3)!id) eco
  eco5 = updateAndExecuteContracts Consumption  ((agents eco4)!id) eco
  eco6 = produce  ((agents eco5)!id) eco5
         in eco
```

- "act" only modifies the "acting" agent and the list of contracts $\Rightarrow$ Autonomy.
- The agent identity "is" also its hour: it defines a unique moment in time (and possibly in space also): time and space are explicit. One can derive notions of locality.

Production

```
produce :: Agent -> Economy -> Economy
produce ag eco =
          (date eco, contracts eco, (agents eco)//[(identity ag,ag2)])
 where
  ag2 = setOutput (snd addProduction) ag1
  ag1 = setInput (fst addProduction) ag
  addProduction = (technology ag) (input ag, output ag)
```

Contracts building blocks

```
demand :: Economy -> Good -> Agent -> Price -> Quantity

supply:: Hour -> Economy -> Good -> Agent -> Price -> Quantity

suppliers ::  Economy -> Good -> Agent -> [Agent]

setContract::  Economy -> Good -> Agent -> Agent -> Contract
setContract eco g buyer seller =
              Ctr (identity buyer) (identity seller) g p q d1 d2
 where
    q = min (demand eco g buyer p) (supply (hour buyer) eco g seller p)
    d1 = if ( elem g [Money,Labor]) then (date eco+1, hour buyer)
                                    else time eco buyer
    d2 = time eco buyer
    p = case g of  ...
```

- Good specific conventions on payment and delivery dates.
- Good specific price setting mechanism (flexible).

Contracts updating

```
updateContracts :: Good -> Agent -> Economy -> Economy
updateContracts g buyer eco =
  foldrUntil noDemand purchase ecoBis (suppliers eco g buyer) where
    noDemand xeco = (demand xeco g buyer 1 <= 0)
    purchase seller xeco =
        if (quantity (setContract xeco g buyer seller) <= 0)
        then  xeco
        else addContracts xeco [setContract xeco g buyer seller,
                          reiterateContract (setContract xeco g buyer seller)]
    ecoBis = (date eco, filter (not.isExpected) (contracts eco) ,agents eco)
    isExpected c = ((good c ==g) && (deliveryTime c == time eco buyer)
                                 &&  (paymentTime c >= time eco buyer))
```

- An agent only updates the contracts for which he is a buyer
- Expected contracts are delayed and replaced by binding contracts.
- Binding contracts are reiterated to serve as expected contracts for next period.

Contracts execution

```
executeContract :: Contract -> Economy -> Economy
executeContract ctr eco=  let
 buyer =  (agents eco)!(buyerId ctr)
 seller = (agents eco)!(sellerId ctr)
 value = (price ctr)* (quantity ctr)
 sellerStock = eval (output seller) (good ctr)
 delivery =  min sellerStock (quantity ctr)
 (buyer1,seller1) = if (paymentTime ctr == time eco buyer)
                    then  (addToOutput Money (-value) buyer,
                           addToOutput Money (value) seller)
                    else  (buyer,seller)
 (newBuyer,newSeller) = if (deliveryTime ctr == time eco buyer)
                        then (addToOutput (good ctr) (delivery) buyer1,
                              addToOutput (good ctr) (-delivery) seller1)
                        else (buyer1,seller1)
 newCtrs = [setQuantity delivery ctr,
            delayDelivery (setQuantity ((quantity ctr)-delivery) ctr)]
           ++ (delete ctr (contracts eco))
       in (date eco, newCtrs,
          ((agents eco)//[(buyerId ctr,newBuyer),(sellerId ctr,newSeller)]))
```

- The contracts are executed only at the buyer's hour.
- Flows always go from output to input.
- Payments can not be delayed, deliveries can.

Update and execute contracts

```
updateAndExecuteContracts :: Good -> Agent -> Economy -> Economy
updateAndExecuteContracts g ag eco =
  foldr executeContract eco1 ctrs1
 where
  eco1 = updateContracts g ag eco
  ctrs1 = filter
              (\c -> ((good c == g) && (buyerId c == identity ag)))
              (contracts eco1)


executePendingContracts :: Agent-> Economy -> Economy
executePendingContracts ag eco = foldr executeContract eco pendingCtrs
  where
  pendingCtrs = filter pendingCharacteristics (contracts eco)
  pendingCharacteristics c = (((deliveryTime c < time eco ag) ||
                              (paymentTime c < time eco ag)) &&
                              (buyerId c == identity ag))
```

Outline

Accounting and money market

- Households receive wages for labor and dividends from money invested (supplied) to firms.
- They have a subsistence and a target income (both parametric):
  - If the actual income is below the target level: buy consumption up to the subsistence income, supply the remaining money to firms inelastically.
  - If the actual income is above the target level : consume everything.
  - $\Rightarrow$ steady state of the neoclassical growth model.

Accounting and money market

```
updateFinancialStatusHousehold :: Agent -> Economy -> Economy
 updateFinancialStatusHousehold ag eco =
   (date eco, contracts eco, (agents eco)//[(identity ag, newAg)])
    where
    newAg = addToInput Money (-income) (addToOutput Money income ag)
    income = eval (input ag) Money

 moneySupply :: Hour -> Economy -> Agent -> Price -> Quantity
 moneySupply h eco ag p = savings - lendings where
  savings = eval (output ag) Money
  lendings = sum (map quantity (filter isLending (contracts eco)))
  isLending c =((deliveryTime c >= begin) &&
                                (deliveryTime c < end) &&
                                (good c == Money) &&
                                (sellerId c == identity ag))
  (begin,end) = if (h > hour ag)
                then ((date eco,hour ag),(date eco +1 ,hour ag))
                else ((date eco -1, hour ag),(date eco, hour ag))
```

Accounting and money market

- Firms form expectations about demand for their product, capital and labor prices on the basis of existing contracts.
- They determine the combination of inputs which produces the expected demand at the least possible cost.
- They demand the money required to finance the purchase of the corresponding amount of new capital.
- They try to collect the corresponding amount of money from households starting with the closest.
- The money collected is added to the firm social capital: it can not be recovered by the household but entails a dividend payment every future period.
- The operating result (sales-wages-amortization) is distributed as dividend.
- Money is delivered immediately and paid at the buyer's hour next period.

Accounting and money market

```
moneyDemand :: Economy -> Agent -> Price -> Quantity
 moneyDemand eco ag p =expectedCapitalDemand* pCapital-(investmentFunds eco ag)
  where
   pCapital = average (map price (filter isCapital (contracts eco)))
   isCapital c = ((deliveryTime c == time eco ag) &&
                            (good c == Capital 1) && (buyerId c == identity ag))
   expectedCapitalDemand = factorDemand (technology ag) (target) ps (Capital 1)
                           - (capital ag)
   target = sum (map quantity plannedDeliveries)
   plannedDeliveries = filter isDelivery (contracts eco)
   isDelivery c = ((deliveryTime c = time eco ag ) &&
                                (deliveryTime c < (date eco +1, hour ag)) &&
                                (good c == productKind ag) &&
                                (sellerId c == identity ag))
   ps =listToFunc [(Labor,expectedWage),(Capital 1,p)]
   expectedWage=  price (head (filter isLabor (contracts eco)))
   isLabor c = ((deliveryTime c == time eco ag)
                && (good c == Labor) && (buyerId c == identity ag))


 moneySuppliers :: Economy -> Agent -> [Agent]
 moneySuppliers eco ag = currentSuppliers++potentialSuppliers
   where
   potentialSuppliers = sortBy compareDistance
                                ((localSellers eco Money)\\currentSuppliers)
   currentSuppliers = [(agents eco)!i | i <- suppliersId]
   suppliersId = map sellerId (filter isShareholder (contracts eco))
   isShareholder c = ((deliveryTime c < time eco ag) && (good c == Money) &&
                      (buyerId c == identity ag))
   compareDistance ag1 ag2 = compare (relativeDistance eco ag ag1 )
```

Accounting and money market

```
addedValue :: Economy -> Agent -> Value
addedValue eco ag = sales + deltaStocks where
 deltaStocks = production ag -sum (map quantity deliveries)
 sales = sum (map value deliveries)
 deliveries = filter isDelivery (contracts eco)
 isDelivery c = ((deliveryTime c < time eco ag ) &&
                              (deliveryTime c = (date eco -1, hour ag)) &&
                               (good c == productKind ag) &&
                               (sellerId c == identity ag))

ebidta :: Economy -> Agent -> Value
ebidta eco ag = addedValue eco ag -wagesPaid where
 wagesPaid = sum (map value (workContracts))
 workContracts =filter isLabor (contracts eco)
 isLabor c = ((paymentTime c == (date eco, hour ag))
                           && (good c == Labor) && (buyerId c == identity ag))

--Amortization is computed in a linear way.

operatingResult ::  Economy -> Agent - > Value
operatingResult eco ag = (ebidta eco ag) - amortization where
              amortization = (usedCapital ag) / (fromIntegral capitalLifeTime)
```

Accounting and money market

```
financialResult ::  Economy -> Agent ->Value
financialResult eco ag = (operatingResult eco ag) + advances
 where
 advances = paidValue-deliveredValue
 deliveredValue = sum (map value deliveries)
 paidValue = sum (map value orders)
 deliveries = filter isDelivery (contracts eco)
 isDelivery c = ((deliveryTime c < time eco ag ) &&
                              (deliveryTime c = (date eco -1, hour ag)) &&
                              (good c == productKind ag) &&
                              (sellerId c == identity ag))
 orders = filter ordersCharacteristics (contracts eco)
 ordersCharacteristics c = ((paymentTime c < time eco ag ) &&
                              (paymentTime c = (date eco -1, hour ag)) &&
                              (good c == productKind ag) &&
                              (sellerId c == identity ag))

netResult ::  Economy -> Agent -> Value
netResult = financialResult
```

Capital market

- Only capital of age 1 can be exchanged.
- A firm demands new capital until it has exhausted the money holdings it had collected for that purpose.
- The price a supplier proposes is such that the value of the production is equal to the value of the expected sales evaluated at a normal price.
- The firm chooses the cheapest suppliers.
- If the whole demand can not be accommodated, deliveries are delayed until the buyer's hour next period
- The consumption good market functions alike.

Capital market

```
capitalDemand :: Economy - Agent - Price - Quantity
capitalDemand eco ag p =  ((eval (input ag) Money) - purchases)/ p where
 purchases = sum (map value (filter isPurchase (contracts eco)))
 isPurchase c = ((paymentTime c== time eco ag) &&
      (good c == Capital 1) && (buyerId c == identity ag))

miscSupply :: Hour - Good - Economy - Agent - Price - Quantity
miscSupply h g eco ag p = (p*prod -p0*expectedSales )/ p0 where
 prod = production ag
 p0 = (targetPrice h eco ag)
 expectedSales = sum (map quantity plannedDeliveryCtrs)
 plannedDeliveryCtrs = filter isDelivery (contracts eco)
 isDelivery c =((deliveryTime c = begin) &&
                              (deliveryTime c < end) &&
                              (good c == productKind ag) &&
                              (sellerId c == identity ag))
 (begin,end) = if (h  hour ag)
               then ((date eco,hour ag),(date eco +1 ,hour ag))
               else ((date eco -1, hour ag),(date eco, hour ag))


miscSuppliers :: Good - Economy - Agent - [Agent]
miscSuppliers g eco ag = sortBy compareSupplyPrice (localSellers eco g)
  where
 compareSupplyPrice ag1 ag2 = compare (supplyPrice ag1) (supplyPrice ag2)
 supplyPrice agX = miscSupplyPrice (hour ag) eco agX (demand eco g ag 1)
```

Labor market

- Each firm demands the amount of labor required to produce the demand it expects, taking as given its capital stock
- Each household supplies one unit of labor at its preceding wage if he was already employed by the firm, at the minimal wage prevailing in the firm if he was previously unemployed and at a price equal to the marginal productivity of its current employer otherwise.
- Labor is delivered immediately and paid at the buyer's hour next period.

Labor market

```
laborDemand :: Economy - Agent - Price - Quantity
laborDemand eco ag w =
 missingToInput (technology ag) target (input ag) Labor - workforce
  where
  target = sum (map quantity plannedDeliveries)
  plannedDeliveries = filter isDelivery (contracts eco)
  isDelivery c = ((deliveryTime c = time eco ag ) &&
                                (deliveryTime c < (date eco +1, hour ag)) &&
                                (good c == productKind ag) &&
                                (sellerId c == identity ag))
  workforce = sum (map quantity (filter isLabor (contracts eco)))
  isLabor c = ((deliveryTime c == time eco ag)
                           && (good c == Labor) && (buyerId c == identity ag))

laborSupply :: Economy - Agent - Price - Quantity
laborSupply eco ag w = if (w = (reservationWage eco ag)) then eval (output ag) Labor  else 0

laborSuppliers :: Economy - Agent - [Agent]
laborSuppliers eco ag = currentSuppliers++potentialSuppliers
  where
  potentialSuppliers = sortBy compareReservationWage
                                ((localSellers eco Labor)\\currentSuppliers)
  currentSuppliers = [(agents eco)!i | i <- suppliersId]
  suppliersId = map sellerId (sortBy comparePrice
                               (filter isEmployed (contracts eco)))
  isEmployed c = ((deliveryTime c == time eco ag) && (good c == Labor) &&
                   (buyerId c == identity ag))
  compareReservationWage ag1 ag2 = compare (reservationWage eco ag1)
                                           (reservationWage eco ag2)
```

Production

- Each household consume all the consumption goods in its input stock and produce one unit of labor.
- Each firm uses its input of capital,k, and of labor, l, to produce $f(k, l)$ units of its product: consumption if the firm is a consumptionFirm capital of age 1 if it is a capitalFirm.
- The production process increments the age of all vintages of capital and the capital whose age is above the capital life time (an exogenous parameter) is deleted.
- In practice, we use Cobb-Douglas production function $f(k, l) = k^{\alpha} l^{1-\alpha}$ where α is the output elasticity.

Production

```
produce :: Agent - Economy - Economy
produce ag eco =
        (date eco, contracts eco, (agents eco)//[(identity ag,ag2)])
 where
  ag2 = setOutput (snd addProduction) ag1
  ag1 = setInput (fst addProduction) ag
  addProduction = (technology ag) (input ag, output ag)

householdTechnology :: Technology
householdTechnology (input,output) = (setFuncValue (Consumption,0) input,
                                      setFuncValue (Labor,1) output)

cobbDouglasTechnology :: Good - Technology
cobbDouglasTechnology g (input,output) = (input3,output3)
  where
  output3 = sumFunc output productionFlow
  productionFlow =  multFunc prod (dirac g)
  prod = (k**alpha)*(l**(1-alpha))
  k = sum  [eval input (Capital x)| x <- [1..capitalLifeTime] ]
  l = eval input Labor
  alpha = capitalElasticity
  input3 =  setFuncValue (Labor,0) input2
  input2 = setFuncValue (Capital  (capitalLifeTime+2),0) input1
  input1 = foldr ageCapital input [1..(capitalLifeTime+1)]
  ageCapital x inputx =
             setFuncValue (Capital (x+1),eval inputx (Capital x)) inputx
```

Outline

Summary/Questions

- An example of computational structure for abm.
- The beginning of an implementation.
- Observational equivalence as a potential relation to equilibrium models
- Can those be used as elements of a domain specific language ?