Dynamic Microeconomic Models and the Taxonomy of HARK

Christopher D. Carroll Alexander M. Kaufman David C. Low
Nathan M. Palmer Matthew N. White

June 2016

Introduction

Heterogeneous Agents

Two dimensions of heterogeneity

- *Ex post* heterogeneity: Agents differ because a different sequence of events or shocks has happened to them
- *Ex ante* heterogeneity: Agents differ in objectives, preferences, expectations, etc before anything in the model "happens"

Microeconomic Models

Microeconomic models in HARK:

- Concern an agent's independent problem
- Discrete time
- Sequence of choices
- Possibly subject to risk
- Agents treat inputs to problem as exogenous

Key restriction: Model solution can be interpreted as iteration on sequence of "one period problems", conditional on solution to subsequent period.

Macroeconomic Models

Macroeconomic models in HARK:

- Additional layer on top of micro model(s)
- Some inputs to micro model are actually endogenous
- Agents' collective states & controls generate macro outcomes
- Expectational equilibrium
- "Rational" is an option
    - If agents act optimally while believing X...
    - ...generates history of macro outcomes consistent with X

Key restriction: Model solution can be stated in Bellman form

A Framework for Micro Models

Building a "universal microeconomic solver"

- A focus of HARK: *modularity* and *interoperability*
- Want elements/modules to play nicely with each other
- Reduce "Tower of Babel problem" and "duct tape coding"

A Framework for Micro Models

Building a "universal microeconomic solver"

- A focus of HARK: *modularity* and *interoperability*
- Want elements/modules to play nicely with each other
- Reduce "Tower of Babel problem" and "duct tape coding"
- Want a universal framework for microeconomic agent problems in discrete time, easy to combine models
- Make a "universal solver" so models speak the same language

The AgentType Class

Our solution: `AgentType`

- General purpose class for representing economic agents
- Each model creates a subclass of `AgentType`
  - Includes model-specific attributes, functions, and methods...
  - ...And how to solve the "one period problem" for that model
  - Instances of subclass are *ex ante* heterogeneous "types"

The AgentType Class

Our solution: `AgentType`

- General purpose class for representing economic agents
- Each model creates a subclass of `AgentType`
  - Includes model-specific attributes, functions, and methods…
  - …And how to solve the “one period problem” for that model
  - Instances of subclass are *ex ante* heterogeneous “types”
- All `AgentType` subclasses use the same `solve()` method
- Just a universal backward induction loop…
- …That lets different models “play nicely” together

The Market Class

- Instance represents an "outcome aggregation"
- Turns microeconomic outcomes into macroeconomic outcomes
  - E.g. asset holdings of agents –> aggregate capital –> R and w

The Market Class

- Instance represents an "outcome aggregation"
- Turns microeconomic outcomes into macroeconomic outcomes
    - E.g. asset holdings of agents –> aggregate capital –> R and w
- Also has "universal solver" to find general equilibrium
    - Seeks consistency agent beliefs about macro outcomes...
    - ...and the macro outcomes that occur when they act on beliefs
- Beliefs might be about dynamics, not static values

Core Code Structure

How This is Implemented in Code

Three broad types of modules:

- Tools: "HARK" modules
  - eg. non-parametric kernel regression
  - discrete approximations to distributions
  - "HARK" in name
  - includes `AgentType` and `Market`
- Models
  - classes to represent agents and markets
  - functions for solving "one period" problems
- Applications
  - use HARK-Tools and Models for specific research question

Overview of Major Modules

Will briefly discuss these components, followed by illustrative examples:

- Main `HARK` modules
- `AgentType` class
- `Market` class

HARK Modules: General Purpose Tools

Main HARK modules are:

- `HARKcore`
- `HARKutilities`
- `HARKinterpolation`
- `HARKsimulation`
- `HARKestimation`
- `HARKparallel`

HARK Modules

`HARKcore`

- `AgentType` class, basic class to extend
- `Market` class, basic class to extend
- `HARKobject`, set attributes and generic "distance" function

HARK Modules

`HARKcore`

- `AgentType` class, basic class to extend
- `Market` class, basic class to extend
- `HARKobject`, set attributes and generic "distance" function

`HARKutilities`

- Utility functions, derivatives, inverses: CRRA, CARA, etc
- Discrete approximations to distributions: lognormal, etc
- Assorted convenience functions: plotting, etc

HARK Modules

`HARKinterpolation`

- Classes for representing function approximations
- Some extended from *scipy.interpolate*, some custom
- Modularity: common interface / methods across classes
  - All subclasses of `HARKinterpolator`
  - Common method names for evaluate, derivative, etc
  - Distance metric as `distance()` method

HARK Modules

`HARKinterpolation`

- Classes for representing function approximations
- Some extended from *scipy.interpolate*, some custom
- Modularity: common interface / methods across classes
  - All subclasses of `HARKinterpolator`
  - Common method names for evaluate, derivative, etc
  - Distance metric as `distance()` method

`HARKsimulation`

- Tools for generating simulated data and shocks
- Takes (sequence of) distribution parameters as input, returns array of simulated draws
- ConsumptionSavingModel : Draws of ψ_t and θ_t

HARK Modules

`HARKestimation`

- Local optimization routines for parameter search
- Simple data bootstrapping tools

HARK Modules

`HARKestimation`

- Local optimization routines for parameter search
- Simple data bootstrapping tools

`HARKparallel`

- Default Python processes single-threaded
- Provides basic tools for multiple CPU cores
  - `multiThreadCommands`
- Parallel Nelder-Mead Simplex
- `Spark` multi-core and grid tools in future extension

`AgentType` Class

The foundational class definition

- `AgentType` class defined in `HARKcore`
- Each specific model defines subclass of `AgentType`
  - define model-specific methods
  - inherits superclass methods

`AgentType` Class

The foundational class definition

- `AgentType` class defined in `HARKcore`
- Each specific model defines subclass of `AgentType`
    - define model-specific methods
    - inherits superclass methods
- Key `solve` method acts as a “universal solver” applicable to any (properly formatted) discrete-time model
- Well-formed instance of `AgentType` must have specific attributes

Attributes of `AgentType`

- `solveOnePeriod`: function (or list of functions) representing a solution method for a single period

Attributes of AgentType

- `solveOnePeriod`: function (or list of functions) representing a solution method for a single period
- `time_inv`: list of names of time-invariant variables needed for `solveOnePeriod`, correspond to attributes in `AgentType`

Attributes of `AgentType`

- `solveOnePeriod`: function (or list of functions) representing a solution method for a single period
- `time_inv`: list of names of time-invariant variables needed for `solveOnePeriod`, correspond to attributes in `AgentType`
- `time_vary`: same as `time_inv` but vary over time. Corresponding `AgentType` attributes must be lists

Attributes of AgentType

- solveOnePeriod: function (or list of functions) representing a solution method for a single period
- time_inv: list of names of time-invariant variables needed for solveOnePeriod, correspond to attributes in AgentType
- time_vary: same as time_inv but vary over time. Corresponding AgentType attributes must be lists
- solution_terminal: solution of terminal period (or first guess if ∞-horizon)

Attributes of `AgentType` (cont.)

- `cycles`: non-negative integer indicating number of times agent experiences sequence of all periods
  - `cycles = 1` : Sequence happens once, lifecycle model
  - `cycles = 40` : Sequence occurs 40 times in a loop
  - `cycles = 0` : Sequence repeats forever, infinite horizon

Attributes of `AgentType` (cont.)

- `cycles`: non-negative integer indicating number of times agent experiences sequence of all periods
  - `cycles = 1` : Sequence happens once, lifecycle model
  - `cycles = 40` : Sequence occurs 40 times in a loop
  - `cycles = 0` : Sequence repeats forever, infinite horizon
- `tolerance`: positive real number representing maximum acceptable "distance" between cycle solutions

Attributes of AgentType (cont.)

- cycles: non-negative integer indicating number of times agent experiences sequence of all periods
  - cycles = 1 : Sequence happens once, lifecycle model
  - cycles = 40 : Sequence occurs 40 times in a loop
  - cycles = 0 : Sequence repeats forever, infinite horizon
- tolerance: positive real number representing maximum acceptable "distance" between cycle solutions
- time_flow: boolean flag indicating the direction time is "flowing"
  - **True**: variables listed in time_vary in ordinary chronological order: $t = 0$ is first period
  - **False**: variables in time_vary in reverse chronological order: $t = 0$ is terminal period

`AgentType` Universal Solver Method

Where the magic happens: `AgentType.solve()`

- Recursively call `solveOnePeriod`
- Uses successive period's solution as `solution_next`
  - first call: `solution_next` ≡ "solution_terminal"'
  - finite horizon: sequence of `periods` solved `cycles` times
  - infinite horizon: terminate using `distance` metric and `tolerance`

`AgentType` Universal Solver Method

Where the magic happens: `AgentType.solve()`

- Recursively call `solveOnePeriod`
- Uses successive period's solution as `solution_next`
  - first call: `solution_next` $\equiv$ "solution_terminal"'
  - finite horizon: sequence of `periods` solved `cycles` times
  - infinite horizon: terminate using `distance` metric and `tolerance`
- Output: model-specific solution class
  - includes behavioral & value functions
- `preSolve` and `postSolve` methods
  - eg. `TractableBufferStock`: `postSolve` constructs interpolated consumption function
- `AgentType` is really a framework (API) to fill out when extending

Important Aside: The Flow of Time

- Dynamic optimization problems solved recursively
  - natural to list time-varying values in reverse chronological order
- However *simulation* of agents typically in chronological order

Important Aside: The Flow of Time

- Dynamic optimization problems solved recursively
  - natural to list time-varying values in reverse chronological order
- However *simulation* of agents typically in chronological order
- `AgentType` has attribute `time_flow` to address this
- Methods to manipulate include:
  - `timeFlip()`: Flip direction of time and change appropriate details
  - `timeFwd()`, `timeRev()`: set direction of time
- *Beauty of inheritance*: you don't need to write these!

Simple Example: Perfect Foresight Consumption-Savings

$$V_t(M_t) = \max_{C_t} u(C_t) + \beta \not{D}_t \mathbb{E}_t \left[V_{t+1}(M_{t+1}) \right]$$

$$\text{s.t.}$$

$$A_t = M_t - C_t$$

$$M_{t+1} = \mathsf{R} A_t + Y_t,$$

where

$$Y_{t+1} = \Gamma_{t+1} Y_t,$$

$$u(C) = C^{1-\rho}/(1-\rho),$$

See class `PerfForesightConsumerType` in `HARK/ConsumptionSaving/ConsIndShockModel.py`

Microeconomic Problem Parameters

Agent's problem characterized by values of:

- ρ, R, β, and
- sequences of survival probs $\not D_t$ and income growth Γ_t for $t = 0, ..., T$

Setting up problem in code:

- Module `ConsIndShockModel.py` defines class `PerfForesightConsumerType( AgentType )`
- Inherits from `AgentType`

A Complete Code Example

```
MyConsumer = PerfForesightConsumerType(time_flow=True,
cycles=1)

MyConsumer.CRRA = 2.7

MyConsumer.Rfree = 1.03

MyConsumer.DiscFac = 0.98

MyConsumer.LivPrb =
[0.99,0.98,0.97,0.96,0.95,0.94,0.93,0.92,0.91,0.90]

MyConsumer.PermGroFac =
[1.01,1.01,1.01,1.01,1.01,1.02,1.02,1.02,1.02,1.02]

MyConsumer.solve()
```

Dive Into Code

Let's look at and run:

- Perfect foresight
- Idiosyncratic income shocks: infinite & finite life cycles
- Idiosyncratic income shocks: infinite + 4 seasons
- Kinked interest rate

Market Class

- So far we've talked about microeconomics / partial equilibrium
  - Some inputs treated as fixed (eg. prices taken as exogenously given)
- How to construct macroeconomics / general equilibrium?
  - Fixed inputs endogenized
  - Rational expectations: agent beliefs about world are consistent

Market Class

- So far we've talked about microeconomics / partial equilibrium
  - Some inputs treated as fixed (eg. prices taken as exogenously given)
- How to construct macroeconomics / general equilibrium?
  - Fixed inputs endogenized
  - Rational expectations: agent beliefs about world are consistent
- Dynamic general equilibrium is fixed point:
  - Agent acting optimally, believing in aggregate rule
  - Individual actions, when aggregated, produce history consistent with aggregate rule
- This is accomplished in `Market` class

A Farm Metaphor

- Imagine all `AgentTypes` in an economy believe some dynamic rule
  - rule stored in a new AgentType attribute
- Each AgentType finds their optimal solution using `solve` + info from dynamic rule

A Farm Metaphor

- Imagine all `AgentTypes` in an economy believe some dynamic rule
  - rule stored in a new AgentType attribute
- Each AgentType finds their optimal solution using `solve` + info from dynamic rule
- `Market` generate a history of GE outcomes by looping over these steps:
  1. `sow`: Distribute the macroeconomic state to all `AgentTypes` in the market
  2. `cultivate`: Each `AgentType` executes their `marketAction` method
  3. `reap`: Microeconomic outcomes are gathered from each `AgentType` in the market
  4. `mill`: Data gathered by `reap` is processed into new macroeconomic states
     - via to some "aggregate market process"
  5. `store`: Relevant macroeconomic states added to running history of outcomes

A Farm Metaphor (cont.)

- This procedure is conducted by the `makeHistory` method of `Market`
  - executed in `Market.solve` method

A Farm Metaphor (cont.)

- This procedure is conducted by the `makeHistory` method of `Market`
  - executed in `Market.solve` method
- After making histories, the market executes `calcDynamics`
  - takes macroeconomic history as inputs
  - generates new rule to distribute to `AgentTypes` in the market

A Farm Metaphor (cont.)

- This procedure is conducted by the `makeHistory` method of `Market`
  - executed in `Market.solve` method
- After making histories, the market executes `calcDynamics`
  - takes macroeconomic history as inputs
  - generates new rule to distribute to `AgentTypes` in the market
- The process then begins again:
  - all agents solve their updated microeconomic models given the new dynamic rule
  - `solve` loop continues until the "distance" between successive dynamic rules is sufficiently small

Market Attributes

A well-formed `Market` instance has these attributes:

- `agents`: A list of `AgentTypes`, representing the agents in the market
- `sow_vars`: list of names of variables output from the aggregate market process (macroeconomic outcomes)
- `reap_vars`: names of variables collected "'agents"' in the "'reap"' step
- `const_vars`: names of variables used by the aggregate market process that *do not* come from "'agents"'
- `track_vars`: names of variables generated by the aggregate market process to record in history, to use to calculate a new dynamic rule

Market Attributes (cont.)

- `dyn_vars`: names of variables that constitute a dynamic rule (to be stored as agent attributes)
- `millRule`: a function for the "aggregate market process", transforming microeconomic outcomes into macroeconomic outcomes
- `calcDynamics`: a function that generates a new dynamic rule from a history of macroeconomic outcomes
- `act_T`: the number of times that the `makeHistory` method should execute the "farming loop" when generating a new macroeconomic history
- `tolerance`: The minimum acceptable "distance" between successive dynamic rules produced by `calcDynamics` for a sufficiently converged solution

Market Attributes (cont.)

The `AgentTypes` which participate in an aggregate market need new methods as well:

- `marketAction`: the microeconomic process to be run in the `cultivate` step
- `reset`: reset, initialize, or prepare for a new "farming loop" to generate a macroeconomic history (eg. reset internal RNG, initial state variables, clear individual history, etc.)

Simple Example: `FashionVictimType(AgentType)`

- Each period, fashion victims[1] make a binary choice of style s
  - dress as jock, $s = 0$
  - dress as punk, $s = 1$
- Utility:
  - comes directly from the outfit they wear...
  - ...and as a function of the proportion of the population who *just wore* the same style
  - they also pay switching costs (c_{pj}, c_{jp}) if they change styles
- Moreover, they receive idiosyncratic type 1 extreme value (T1EV) preference shock to each style, in each period.

[1] This model is inspired by the paper "The hipster effect: When anticonformists all look the same" by Jonathan Touboul.

Simple Example: "FashionVictim"

Define:

- Proportion of population punk as $p \in [0, 1]$
- Conformity utility function as $f : p \to \mathbb{R}$,
- No restrictions on f; fashion victims might be conformists ($f'(p) > 0$) or hipsters like to style themselves in the minority ($f'(p) < 0$)

Single period utility function is:

$$\begin{aligned} u(s_t; s_{t-1}, p_t) = & \; s_t f(p_t) + (1 - s_t) f(1 - p_t) \\ & + s_t U_p + (1 - s_t) U_j \\ & - c_{pj} s_{t-1} (1 - s_t) - c_{jp} (1 - s_{t-1}) s_t . \end{aligned}$$

- Agents are forward looking, discount future at β

FashionVictim Problem

$$V(s_{t-1}, p_t) = \mathbb{E}\left[\max_{s_t \in \{0,1\}} u(s_t; s_{t-1}, p_t) + \eta_{s_t} + \beta \mathbb{E}\left[V(s_t, p_{t+1})\right]\right],$$

$$p_{t+1} = ap_t + b + \pi_{t+1}, \qquad \pi_{t+1} \sim U[-w, w], \qquad \eta_0, \eta_1 \sim T1EV.$$

- Assume agents believe that the p_{t+1} is a linear function of p_t, subject uniform shock.
- Problem characterized by U_p, U_j, c_{pj}, c_{jp}, a function f, and beliefs: a, b, w.

Aggregate Solution

- Given U_p, U_j, c_{pj}, c_{jp}, a function f, and beliefs: a, b, w, a `FashionVictimTypes` infinite horizon microeconomic model can be solved in a few lines

Aggregate Solution

- Given U_p, U_j, c_{pj}, c_{jp}, a function f, and beliefs: a, b, w, a `FashionVictimTypes` infinite horizon microeconomic model can be solved in a few lines
- Individual agents treat the dynamics of p_t as exogenous ...
  - ... however they are in fact endogenously determined by all fashion victims in the market.

Aggregate Solution

- Given U_p, U_j, c_{pj}, c_{jp}, a function f, and beliefs: a, b, w, a `FashionVictimTypes` infinite horizon microeconomic model can be solved in a few lines
- Individual agents treat the dynamics of p_t as exogenous ...
  - ... however they are in fact endogenously determined by all fashion victims in the market.
- A dynamic general equilibrium of the "macroeconomic fashion model" is thus characterized by a triple of (a, b, w) such that when fashion victims believe in this "punk evolution rule" and act optimally, their collective fashion choices exhibit this same rule when the model is simulated.

Finding Self-Consistent Dynamic Rule

The search for dynamic general equilibrium is implemented in `Market` with these definitions:

`sow_vars = ['pNow']` (macroeconomic outcome is p_t)

`reap_vars = ['sNow']` (micro outcomes are s_t for many agents)

`track_vars = ['pNow']` (must track history of p_t)

`dyn_vars = ['pNextSlope','pNextIntercept','pNextWidth']` (dynamic rule (a, b, w))

`millRule = calcPunkProp` (aggregate process: average the style choices of all agents)

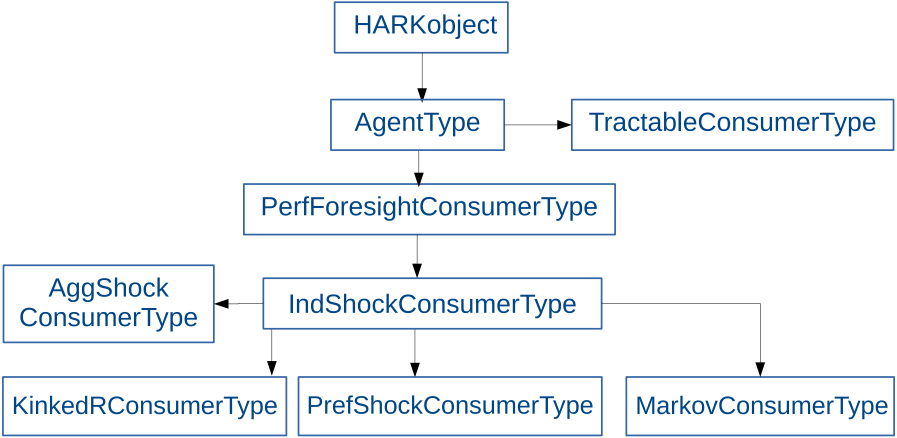
`calcDynamics = calcFashionEvoFunc` (calculate new (a, b, w) with autoregression of p_t)

`act_T = 1000` (simulate 1000 periods of the fashion market)

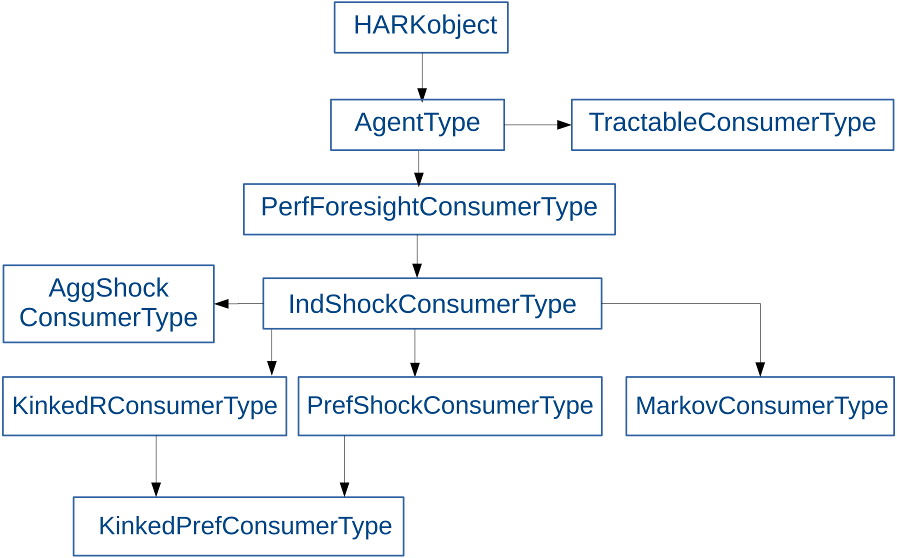
`tolerance = 0.01` (terminate when (a, b, w) changes by less than 0.01)

Extending with Inheritance

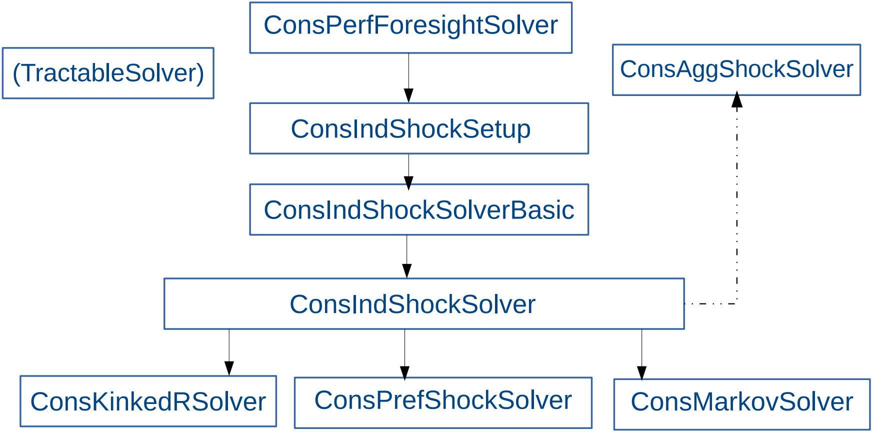
Playing with HARK: Class Inheritance



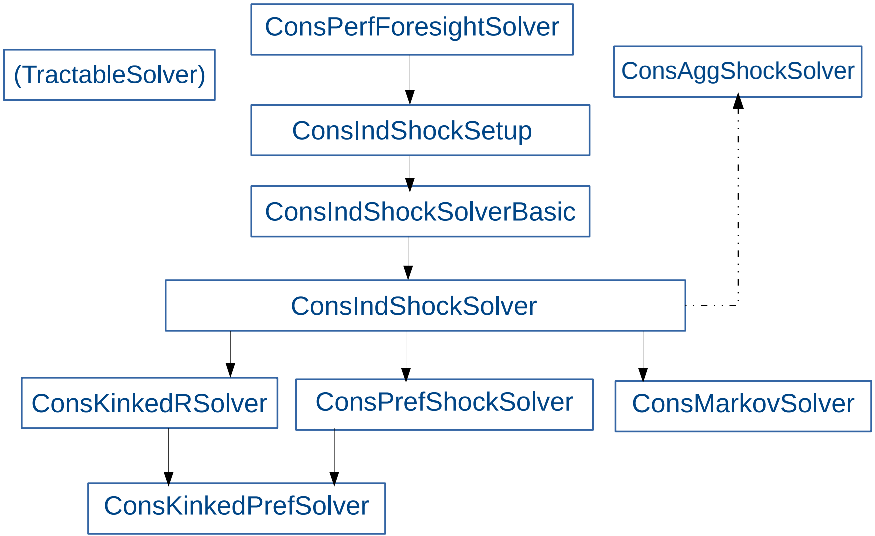
Playing with HARK: Class Inheritance



Playing with HARK: Class Inheritance



Playing with HARK: Class Inheritance



Playing with HARK: Class Inheritance