Memorandum for Meeting with Pablo Winant — December 15, 2023

Purpose: The Econ-ARK team has been internally discussing the development of the next iteration of `HARK`, and seeks feedback on our ideas from Pablo.

Background: The current version of `HARK` does not have a formal relationship among the mathematical model being represented, the operations performed to generate a "solution" of the model (policy functions and aggregate dynamics), and the model as acted out in simulation. Each of these components is written out separately for each model, and end users are meant to trust that there is consistency among them. Consequently, modifying an existing `HARK` model is not straightforward nor easy, as the user must make changes in several places in the code and understand the relationships among various modules. The solution methods in the current version of `HARK` are specifically tailored for each model, and the package is designed with *ex ante* heterogeneity at front of mind.

In contrast, the approach taken by Pablo's `dolo` and `dolark` projects enforces consistency among the model concept, solution, and simulation by using a "single source of truth": a clear specification of the model's dynamics (equations) and characterization of its solution, which is directly used when solving and simulating the model. User-specified models in `dolo` and `dolark` are solved automatically by several methods, which has the upside of being easy to use and the downside of potentially less efficient computation from forgoing hand-crafted code. Pablo's projects were designed primarily for macroeconomic modeling. *Ex ante* heterogeneity and life-cycle models are possible to represent in these packages, but were not the primary intended use-case. Models with mixed choices or that change fundamentally over the lifecycle ("phases of life") are not what `dolo` and `dolark` are designed for.

Pitch Summary: The Econ-ARK team wants to develop a next iteration of `HARK` that incorporates the structural consistency of `dolark`, while maintaining its focus on models with extensive *ex ante* heterogeneity and life-cycle dynamics. This encompasses two main components: first, a format for precisely specifying dynamic models that can handle a wide variety of features (e.g. continuous and discrete choices, lifecycle problems, mortality and population dynamics, etc), like an extended `dolang`; and second, a software package that imports model specifications and allows the user to interactively construct a model piece by piece. Our intent is for the model specification language to be useful *outside* of our own software package: a clear and consistent way to specify model information that could also be used by *other* economists for their own purposes (e.g. another software package).

Basic Example Model: To fix ideas, consider the one period problem of an agent who must choose how much to consume c_t out of their market resources m_t and how much to retain as assets a_t. They have CRRA preferences over consumption, can earn a risk free return on retained assets, and discount future utility flows by a constant factor.

$$\mathrm{v}_t(m_t) = \max_{c_t} \frac{c_t^{1-\rho}}{1-\rho} + \beta \overline{\mathrm{v}}_{t+1}(k_{t+1}) \text{ s.t.}$$

$$k_{t+1} = a_t = m_t - c_t, \qquad a_t \geq 0,$$

$$b_t = \mathsf{R} k_t, \qquad m_t = b_t + y_t, \qquad y_t \sim F,$$

$$\overline{\mathrm{v}}_t(k_t) = \mathbb{E}[\mathrm{v}_t(m_t) \mid k_t].$$

In this statement, which is intentionally more belabored and elaborate than strictly necessary, $\mathrm{v}_t(m_t)$ is the ordinary Bellman value function over the decision-time state variable (market resources), while $\overline{\mathrm{v}}_t(k_t)$ is the value function at the *very beginning* of period t, before capital returns or labor income uncertainty are resolved. To notationally separate periods, we emphasize that the same quantity represented by end-of-period-t assets is *also* beginning-of-period-$t+1$ capital holdings. These terms and notation will be used below.

Modeling Language: The core element of our proposed modeling language is a "stage", representing a sequence of events for an agent (or actor). An agent's dynamic programming problem is constructed by linking these stages together, potentially recurring on itself. Model stages are meant to compartmentalize information, explicitly specifying the variables that are "inbound" to the stage as information (i.e. "state variables") and objects that can be observed from its successor when solving (e.g. the continuation value function). Like in `dolo`, a stage declares its parameters, distributions, functions, and algebraic substitutions (definitions); unlike `dolo`, the *values* or *details* of these parameters and distributions do *not* need to be provided, and can instead be left symbolic.

The dynamic model content of a stage could be as small as one model "step", or it could encompass as much as what we would ordinarily describe as entire period of the model. For example, in the context of a basic consumption-saving model with transitory labor income risk and a riskless return, a stage could be as small as going from beginning-of-period capital holdings k_t to bank balances b_t by multiplying the former by return factor R; or it could be the next step, where labor income y_t is drawn from some distribution and added to b_t to yield market resources m_t. Alternatively, a stage could be specified as having *all* of the steps in one

period of the consumption-saving problem: earning interest on capital, receiving stochastic labor income, choosing consumption, and retaining non-consumed market resources as assets.

Suppose a user chose to specify the simple consumption-saving model using (say) four very small stages. Under our proposed modeling language, they could further specify that these stages are sequentially linked together. The resulting linked group of stages would itself then represent a stage (the entire period) that could be further linked.

The Econ-ARK team has been developing draft specifications of this modeling language, formatted as YAML files. This allows model information to exist alongside parameterization choices, but not inextricably paired together. Instead, the modeling language will include a directive for assigning parameters (or distribution information, etc) from a YAML dictionary entry to a stage, filling in previously unspecified information. As discussed below, our proposed software package would also allow a user to import a stage into their modeling environment without parameter values, and *then* to assign values.

Our draft language specification includes concepts for flow control among stages, so that (e.g.) an agent's discrete control can govern the *nature* of the next choice they face, or a mortality shock can determine whether they continue to the next period or simply terminate. A group of connected stages thus acts like (and can be represented by) a directional graph among a set of nodes. A graph that loops back on itself is interpreted as an infinite horizon model in which the solution for optimal controls is recursively defined, while a graph that reaches a definite terminal node is a finite lifecycle problem.

The `dolo`, `dolark`, and `dynare` packages are all designed for solving macroeconomic models that include endogenous objects such as factor prices and (approximations of) aggregate dynamics. We intend for our new language to also be able to describe these important features, but our draft proposals have not yet progressed to discussing how to do so.

Software Package: The software package we plan to develop (sometimes internally called "HARK 2.0") will act as an interpreter of the YAML model statements, interactive model development environment, and platform for applications using the models (e.g. structural estimation). Several features of the model specification language will also be included as programming directives in our software package– the user can link stages, create a sequence of periods, assign or change parameters, etc.

The new version of `HARK` will read in YAML model files and generate representations of model objects using `sympy`'s various capabilities, which now include so-called "pleuripotent" functions: objects that are simultaneously both a symbolic representation of a function's

mathematical form *and* a numeric approximation of it (e.g. a representation of the optimal policy or value function) that can actually be executed. The `sympy` package now has well developed interfaces with various numeric back-ends (including `numpy` and `jax`), so that carrying symbolic representations of the model does not (necessarily) impose grave efficiency costs on computations using them.

Likewise, `sympy`'s automatic differentiation and algebraic substitution capabilities have now been sufficiently well developed that an Econ-ARK team member has prototyped code for generating an algebraic representation of the first order and envelope conditions for an intertemporal optimization problem. These representations can be automatically converted into code that numerically solves the choice problem using the endogenous grid method (EGM). The new version of `HARK` will thus be able to quickly generate (a draft of) solver code from the model input, which could be used as-is or manually refined by the user. The new version of `HARK` will also maintain the ability to specify custom solver code.

We have also already prototyped automatic Monte Carlo simulation based on YAML model input, ensuring consistency between intended and actual model behavior. The new version of HARK will support multiple modes of population dynamics, including explicit mortality Monte Carlo, cumulative survival probability weights, and discretizations of continuous state variables (for Markov-style approximations).

Example Usage: So as not to distract with specification details that have not yet been implemented (let alone finalized), we present a *hypothetical* example of command-line interaction in the future `HARK` modeling environment, omitting the YAML files with the model and parameters; some imagination is required. Analogous concepts for most of these operations would also exist in the modeling language, so that more complex (or more completely specified) model objects could be directly imported.

[Hypothetical code example on next page for easier reading]

```
> from HARK import read_yaml, AgentType, Connector, link
> CS_model_components = read_yaml('BasicCSmodel.yml')
> CS_model_components.stages
OUT: [basic_consumption_saving_period, risk_free_return_stage, transitory_income_stage,
consumption_choice_stage, discount_stage]
> CS_model_components['basic_consumption_saving_period'].parameters
OUT: [Rfree, DiscFac, CRRA]
> CS_model_components['basic_consumption_saving_period'].distributions
OUT: [TranShkDstn]
> some_CS_parameters = read_yaml('ExampleCSparams.yml')
> my_CS_period = CS_model_components['basic_consumption_saving_period'].copy()
> my_CS_period.parameterize(some_CS_parameters)
> my_CS_period.label('perpetual youth basic CS period')
> time_loop = Connector(tick=True, twist={'aLvl':  'kLvl', 'vFunc_continuation':
'vFunc_before_return'})
> InfiniteCSproblem = link(my_CS_period, time_loop, my_CS_period)
> MyConsumerType = AgentType(model=InfiniteCSproblem, count=10000)
> MyConsumerType.solve(method='EGM')
> MyConsumerType.solution.policy
OUT: [cLvl(mLvl) for perpetual youth basic CS period]
> MyConsumerType.track('aLvl','cLvl')
> MyConsumerType.simulate(method='montecarlo')
```