Using Probabilistic Programming Languages to Model Game Heuristics

Karl Cronburg
karl@cs.tufts.edu
Dept. of Computer Science, Tufts University, Medford MA 02155

1. Introduction

A **game-play heuristic** is a description of what a player should do to maximize her utility in a game. Due to the probabilistic nature of many games, this description can be formulated as a **probabilistic model**. Repeated sampling from this model gives the modeler a top-down view of what the heuristic does. The modeler can then, given an inference engine, query the state of the game given some observations about a state of the game at a different point in time.

It is apparent from this description that game-play heuristics fit into the model-observe-query pattern used by existing Probabilistic Programming Languages (PPLs). In this work we present an argument favouring the use of PPLs for describing game heuristics and reasoning about their performance.

We now continue by studying a particular game with sufficiently complicated probabilistic game mechanics, Dominion, discussed in Section 2. The complexity of Dominion and our attempts to model and make queries gives us insight into the desired features of a PPL, and advantages and disadvantages of existing PPLs. We discuss these insights in Section 3. This is followed in Section 4.1 with a discussion of Dominion-specific future work. We also touch on in Section 4.2 the future of PPLs in relation to their applicability to the domain of game heuristics.

1.1 Probabilistic Techniques: Markov Chain Monte-Carlo

In this work we use rejection sampling to infer distributions over the latent variables in our model. In the future we would like to further study instances in which rejection sampling is insufficient due to high rejection rates. Section 4 goes into more detail about future work likely requiring more clever inference techniques.

High rejection rates in the case of deck building games like Dominion will most likely be the result of inference questions involving very restrictive conditioning. While it is possible that a high rejection rate can be the result of a non-diffuse true distribution, this is not indicative of an interesting deck of cards. For example you could design a card which turns of shuffling in the game resulting in a pathologically non-diffuse distribution. This however is not the kind of card game designers wish to study.

2. Dominion

Dominion is a game we are familiar with. This game falls into the category of **turn-based deck building game**, where a player *builds* a good deck by choosing what cards to buy on her turn. The probabilistic component of deck building games arises from shuffling decks of cards at certain points in the game. In particular in Dominion the cards a player plays can affect both when cards get shuffled and the composition of the deck being shuffled. As a result, game-play heuristics for Dominion are inherently intertwined with the probabilistic game state.

2.1 Game State

The exact details of what a model of Dominion needs to keep track of are described in Appendix C. The most important part of this state information when formulating a buy heuristic is the set of cards available to be bought. This set of cards is known as the *supply*. Each card in the supply has a name, a cost, and a number of copies available for purchase. A player

then, during her buy-phase, chooses a card to buy costing less than or equal to the number of *treasure* cards played during her money-phase.

In addition to the buy-phase transforming the state of the game (moving a card from the supply into a player's discard pile), there is an action-phase (preceding the money and buy phases) during which a player plays cards from her hand. Actions transform the state of the game. One such action card is described in detail in Appendix B.

2.2 Greedy Heuristic

In Dominion a game heuristic comprises how a player plays cards and how she buys cards. In general the play-heuristic is easier to reason about because it usually just involves maximization of the amount of money in the player's hand by the end of the action phase. In probabilistic programming terms, this can be implemented as a sampling of the distribution over the amount of money resulting from each possible chain of actions the player can perform. A good play heuristic might then choose the set of actions which result in the highest expectation value for amount of money.

For simplicity, the play-heuristic used in the experiments in this work is one which greedily plays cards which give the player more actions. After playing all cards which give her additional actions, the player plays all remaining cards in decreasing order of cost of the card. The haskell implementation of this is:

```
greedyAct :: Game -> Measure (Maybe CardName)
greedyAct g = do
  let as = filter isAction $ (cards.hand.p1) g
  case length as of
    0 -> return Nothing
    _ -> if elem VILLAGE as
         then return $ Just VILLAGE
         else return $ Just $
              maximumBy (comparing cost) as
```

In contrast to the play-heuristic, the buy-heuristic is a much more interesting heuristic to study. This is because a 'good' buy-heuristic will change throughout the course of the game. Again for simplicity however, we study a heuristic which greedily buys the most expensive card it can with a high probability regardless of the turn number in the game. The exact intricacies of the buy heuristic can be gleaned from the Hakaru model implementation [Cronburg 2014].

2.3 Greedy Theory

A simple query one might wish to ask of the greedy heuristic is:

> **Question 1:** Given that we buy card X with probability p_0 and card Y with probability $(1-p_0)$, what is the distribution over the length of the game?

In this query, we have model parameters comprising:

- X `::` **`Card`**
- Y `::` **`Card`**
- p_0 `::` **`Probability`**

We also have implicit hyper-parameters comprising (see Appendix A for an overview of the Haskell implementation):

- `kingdomCards` `::` `[`**`Card`**`]`
- `turnRules` `::` `[`**`GameParameters`**`]`

One observe-query inference question to ask the model based on the probable game lengths Question 1 gives us is:

> **Question 2:** In a greedy buy heuristic a player buys card X with some unknown probability p_0 and card Y with probability $(1-p_0)$. We observe a game length of t'_e turns. What is the probability p_0 the greedy heuristic used?

The semantics of this question are slightly different than those of Question 1 . The key difference is in the perspective of the asker. In this question p_0 is a latent variable because the asker is not told the value of this parameter to the greedy heuristic model. In the previous question p_0 was simply a parameter to the model because the perspective of the question is from the player executing the buy heuristic. See Section 4.2 for a discussion of the linguistic ramifications of the context-dependence of whether or not a variable is latent, observable, a parameter, or a hyper-parameter. For the greedy model presented in this work, we do not constrain the type of p_0 with any of these type abstractions. This is primarily because Hakaru neither requires nor enforces such types.

The simplicity of this question makes for an easier to follow domain analysis of the greedy heuristic. It however has the drawback of not being as interesting as more complex queries. An obvious extension to this query is to expand the state space to include a parameter p_i for every possible `(money,card)` `::` `(`**`Int`**`,`**`Card`**`)` pair. The lack of probabilistic abstractions just discussed however makes the implementation of such a model in Hakaru error-prone. Therefore for the sake of ensuring correctness of our implementation, we only present results for the one-parameter case. See the future work section (Section 4.2) for further discussion of the abstractions a PPL might support.

2.4 Greedy Results & Analysis

We now present results for the greedy heuristic, as described in Section 2.2. Before answering the inference question (Question 2), we first take unconditioned samples from the greedy-vs-greedy model to determine characteristic game lengths. The result of this sampling is realized as a Gaussian-like distribution of Figure 1. Sampling the model with other values of the **`VILLAGE`** - **`CHANCELLOR`** buy ratio p_0 shows that p_0 is inversely related to the mean of the Gaussian μ_t.

From this information we infer that the **`VILLAGE`** card has a greater impact on how quickly the game ends. This is interesting to a player of Dominion attempting to come up with

a ranking of cards. Performing an automated analysis on every possible pair of cards using this sampling technique, one could:

- Design effective greedy buy heuristics
- Determine the balance of the hyper-parameter corresponding to which cards are being used in the game.

The **VILLAGE** having a greater impact on the game ending is also in line with the personal belief of the author that a **VILLAGE** is, in general, more useful than a **CHANCELLOR**. However, what the Gaussian form of Figure 1 does not tell us is the characteristic distribution of values of p_0 for a given game length. This is what the inference question (Question 2) is trying to give us insight into.

In Figure 2 we see the characteristic form of the turn-conditioned distribution over the **VILLAGE** - **CHANCELLOR** buy ratio (p_0). At first glance these distributions appear to be linear in p_0. In terms of our parameters we can therefore write the equation:

$$P(p_0|t_e = t'_e) = m(p_0, X, Y) * p_0 + b(p_0, X, Y) \quad (1)$$

In this equation we now have a name for the observed random variable 'game length' (t_e, the end turn number). We also have t'_e a specific (conditional) value for t_e. Finally the latent random variables m and b together comprise a characteristic description of the conditional distribution. The key to this equation is that m and b are variables which cannot be directly sampled / observed and have a probabilistic causal affect on the observed variable t_e. Since we have no feasible means for determining the closed analytical forms of m and b we rely on the Markov Chain Monte-Carlo (MCMC) method discussed in Section 1.1.

For the specific case of X = **VILLAGE** and Y = **CHANCELLOR**, we see (Figure 2) a relationship where as the number of turns t'_e we condition on goes down:

- the slope m of the conditional distribution increases from negative to positive
- the intercept b decreases
- the area under the distribution gradually shifts from low values of p_0 to high values of p_0.

These trends are unsurprisingly in accordance with the trend of μ_{t_e}, the mean of the Gaussian distribution of Figure 1. Namely that the **VILLAGE** to **CHANCELLOR** ratio increases as the number of turns in the game decreases. The new insight we have gained from our MCMC methods is the form of Equation 1 as well as values for the latent variables m and b at various conditioned points. The form of the conditional distribution is in fact itself a latent variable which can be dependent on both the hyper-parameters to the game engine and parameters to the probabilistic model. Presently it is a latent variable which can only be computed by a human, but given a good fitting algorithm could be automated. See Section 4 for further discussion of analysis automation.

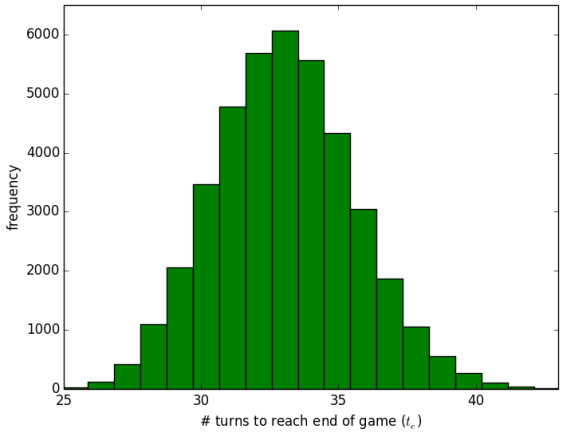

Figure 1. Probability distribution over number of turns in a 2-player game of Dominion. Both players play a greedy strategy with model parameters of X = **VILLAGE**, Y = **CHANCELLOR**, and $p_0 = 0.5$. See Appendix D for a complete description of the semantics behind these cards and why this pair of cards is interesting for the game Dominion. This distributions is unconditioned and does not involve any latent variables, therefore only requiring sampling directly from the model (no inference).

3. Probabilistic Language Analysis

Existing probabilistic languages seem to rely heavily on the linguistic constructs given to them by their host languages. From our experience in implementing probabilistic game heuristic models with BLOG [Milch 2014] and Hakaru [Hakaru 2014], we see that PPLs which tend to defer to their host language's abstraction capabilities are more expressive and powerful. At the same time such languages also rely heavily on the model of computation given to them by their host language. Hakaru relies heavily on Haskell's monadic model of computation. BLOG on the other hand has developed the **many-worlds** model of computation.

Probabilistically, the latter model of computation can be very satisfying in terms of its similarity to the mathematical description of a probabilistic process. In many situations declaratively describing a probabilistic model of the possible worlds is more natural. In the domain of games, one can envision a declarative enumeration of the rules in a game and how they affect the game state.

Such rules form the underlying mechanics of a game. The probabilistic components of these mechanics are, as discussed throughout this work, implementable in PPLs. The difficulties in creating these implementations in Hakaru in particular are described below (Section 3.1).

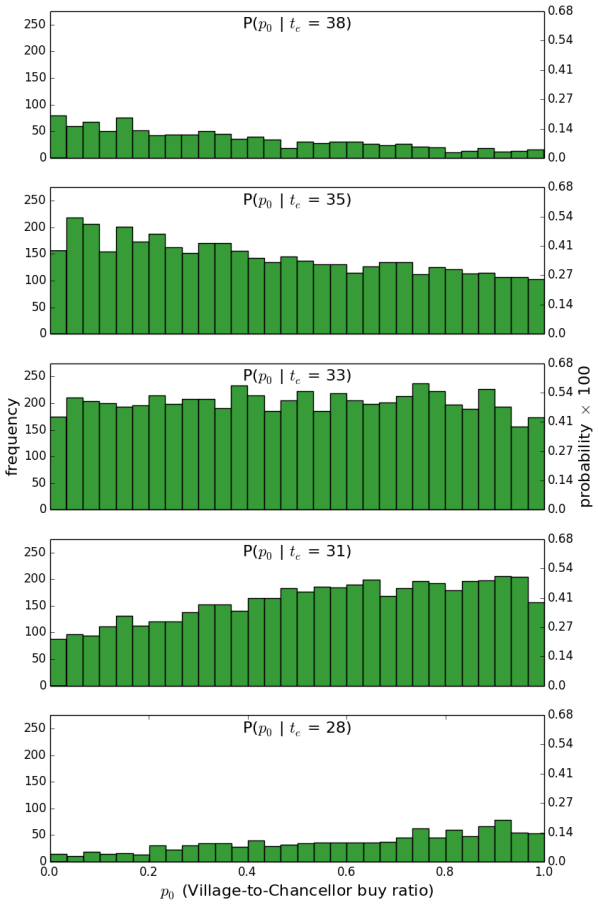

Figure 2. Inferred probability distributions using rejection sampling. On the x-axis we have the inferred value of the latent variable p_0. A value of $p_0 = 1.0$ indicates the greedy player buys a `VILLAGE` with probability 1.0 and a `CHANCELLOR` with probability 0.0 when it has 3 money (the cost of both cards) in play. On the y-axis we have both frequency and probability. Note that the frequency is relative to a total sample size of $40,590$. Similarly the probability is normalized by the total sample size. As a result the probabilities in each individual distribution will not sum to 1. The distributions shown are formed using 30 bins each of size $1.0/30$. We show the inferred distribution only for $t_e \in$ `[38,35,33,31,28]` because these give a sufficient view of how the distribution over p_0 changes for typical values of t_e from the Gaussian in Figure 1

3.1 Hakaru Analysis

When implementing in Haskell a complex model like that of Dominion, a need for various monadic structures arises. In the case of Dominion this includes the `State` and `Measure` monads (and eventually the `IO` monad). It would therefore have been helpful for Hakaru's `Measure` monad to support standard Haskell monads (`State` in particular). The current implementation of our model manages to combine these two monads using the `StateT` monad transformer and the use of Haskell's default `lift` to move values between the `State` and `Measure` monads. This model and more complex models using other monads would certainly benefit from Hakaru having its `Measure` monad derive more typeclasses.

Similarly Hakaru's interface into Haskell's random number support was unsatisfying to use. In particular the modeler cannot specify a source of randomness at the top-level operations Hakaru exposes. While the default source of randomness Hakaru uses is applicable for many situations, it makes meta-programming difficult. In the implementation of our model we needed to re-write some of the sampling interfaces to consume a parametric source of randomness rather than a hard-coded one in order to get the desired effect.

Another design flaw currently baked into Hakaru's runtime system is the use of runtime errors to unconditionally terminate the program when Hakaru sees something it is unable to handle. The existence of one of these errors is likely indicative of a bug in the model supplied to Hakaru. However it is unsatisfying because it both inhibits meta-programming and debugging.

Meta-programming is inhibited because a modeler may for instance want to programmatically test for the correctness of a model by seeing if it produces a runtime error. This is only feasible if the runtime error is in some way 'catchable'. Hakaru could for instance use Haskell's `State` monad to maintain both a list of warnings and errors along with the contents of its `Measure` monad.

Similarly runtime errors inhibit future Hakaru-specific support for debugging because they rely on an inherently imperative feature in a functional host language. In particular one can modularize the implementation of model debugging support in Hakaru if one takes advantage of the full expressive power of Haskell in lieu of relying solely on Haskell's `error` semantics. It would also be very helpful for Hakaru to print user-friendly error messages and suggestions about common mistakes seen in probabilistic models.

3.2 BLOG Analysis

In addition to Hakaru we also look at the feasibility of defining a probabilistic model of Dominion as a BLOG model. The general form of Hakaru's issues just described in Section 3.1 is 'we have a powerful host language, the abstractions just need to be refined.' The form of BLOG's issues we discuss in this section is 'we have a powerful model of computation and a set of natural probabilistic abstractions,

but the linguistic components of BLOG are presently undeveloped or poorly abstracted.'

One such poorly designed abstraction is the use of comprehensions in BLOG's grammar. BLOG currently supports set-comprehensions but not list-comprehensions. Grammatically (and syntactically) these two features are virtually identical. It is evident from this that BLOG's grammar has done a poor job of distinguishing between general purpose linguistic features and probabilistic-specific features. It is this author's opinion that there should be a clean distinction between probabilistic and general purpose features. BLOG in particular is, to some extent, embedded in Java. This embedding could be exploited to a similar extent that Hakaru exploits features of Haskell. Such an embedding would forgo the need to 'reinvent the wheel' in terms of making BLOG a more expressive language.

The current trajectory of BLOG appears to be towards tailoring general purpose language features to BLOG's particular model of computation. The ambitious nature of this goal could eventually give BLOG a highly expressive modeling form in conjunction with an efficient model of computation. Presently however, BLOG falls significantly short of the expressive power needed to naturally model significantly complex probabilistic processes.

4. Future Work

This work touches on the language design aspects of PPLs in general. We also discuss the applicability of such PPLs to a specific game (Dominion). In Section 4.1 we discuss Dominion-specific future work. In Section 4.2 we discuss possible future extensions of the linguistic analyses from Section 3.

4.1 Dominion

This work has demonstrated that modeling Dominion is, by itself, a difficult task. The modeler must have both an intimate knowledge of Dominion's mechanics and some level of experience using PPLs. This is further complicated by the fact that games like Dominion are described most naturally in a recursive form. Future work should therefore include a guarantee of correctness of the models presented in this paper. We are confident in the MCMC techniques used in this paper. We are however less confident in the exact numeric results shown in the distributions in the results section. The results presented do cross-validate with existing beliefs, we should however perform a more statistically rigorous analysis of these results.

This future analysis could involve computing values for the intercepts and slopes of the three distributions in Figure 2. We believe this slope m can be used to compute the relative balance between the cards X and Y in our greedy model. Namely that balance is proportional to $1/abs(m)$. A slope of exactly $m = 0$ would mean the cards X and Y are perfectly balanced (infinite balance). In this case we have a uniform conditional probability distribution over p_0.

In contrast a slope approaching $m = \pm\infty$ indicates either X or Y completely dominates the other (perfectly imbalanced). The conditional distribution we infer should, for perfect imbalance, approach the form of a Dirac delta function:

$$P(p_0|t_e = t'_e) \rightarrow \begin{cases} \delta(0.0) & : \text{as } m \rightarrow -\infty \\ \delta(1.0) & : \text{as } m \rightarrow +\infty \end{cases}$$

In similar fashion to the slope m, we believe the intercept b to be an absolute measure of the balance between cards X and Y. Future work involves both testing these hypotheses about the form of the distributions and quantifying the distributions for interesting conditions.

We hope to make analyses like this one more friendly to 'game design' domain experts who are not steeped in the statistical and linguistic aspects of the analysis. Such a goal is best suited to a domain-specific probabilistic programming language. This DSL would provide built-in functionality for computing metrics like *balance* from the inferred conditional distributions. The output of programs in the DSL would be a set of statistics and visualizations of interest to a game designer. This output is in lieu of (or in addition to) the probability distribution(s) printed by existing general-purpose PPLs.

The tools and support we envision a game designer having in conjunction with a probabilistic DSL comprise:

- Automated analysis of the form of inferred probability distributions using existing curve fitting algorithms.
- Support for analysis of multiple variables at the same time
  - Coordinated multiple views visualization of the conditioned distributions over the latent variables.

These tools and features compliment Dominion-specific future work comprising:

- Implementation of more Dominion-specific heuristics.
- Designing heuristics capable of maintaining probabilistic models corresponding to belief states. What is the feasibility of recursive use of probabilistic models in existing PPLs?
- Cross-validation with currently held beliefs of human Dominion players / domain experts. [Qvist 2013]
- Applying Machine Learning algorithms to publicly available Dominion data sets. [Zonker 2013]
- Design of a probabilistic inference engine to determine **balanced** sets of cards to play with.

4.2 PPL Design

Hakaru does not enforce any sort of type abstractions related to latent and observable variables. This makes for easy

implementation of probabilistic models which can be used to answer various questions regardless of the exact interpretation of the variables in the question. This however in the best case requires a reader to implicitly (e.g. by reading code comments) determine which variables are latent or observed on a particular run of the program. In the worst case it can result in design flaws in probabilistic models which invalidate the results.

One possible solution to this is for PPLs like Hakaru to strictly enforce the primitive probabilistic types (observable, latent, ...) of a variable using an existing strongly-typed host language. The PPL then defines a library of combinator functions for transforming and composing model variables. For example the '+' combinator for adding a latent and an observable variable together would result in a latent variable. The Haskell type signature might look something like:

```
(+) :: (Num a, Num b, Num c)
    => Latent a -> Observable b -> Latent c
```

Such abstractions would have allowed us to express our inference question (Question 2) as an unambiguous Haskell function. In particular the variable p_0 is used in one context as a known parameter to the model, and in another context as the latent variable we would like to infer. While our inference question is relatively simple (single variable), keeping track of which variables are observable and which are latent becomes error-prone when we look at inference questions involving multiple parameters of interest.

In the immediate future we would like to investigate:

- The usability of Church, Figaro, probability monad, and other PPLs in the context of this work.
- The utility of a domain specific language for modeling deck building games. What do we gain from card-game specific probabilistic abstractions?

References

K. Cronburg. Dominion runtime system, 2014. URL https://github.com/cronburg/deckbuild.

K. Cronburg, R. Veroy, and M. Ahrens. Deckbuild: A declarative domain-specific language for card game design, 2014. URL https://github.com/cronburg/deckbuild-lang.

Hakaru. A probabilistic programming embedded dsl, 2014. URL https://hackage.haskell.org/package/hakaru.

G. B. Mainland. Why it's nice to be quoted: Quasiquoting for haskell. *ACM 978-1-59593-674-5/07/0009*, 2007.

B. Milch. Bayesian logic, 2014. URL https://bayesianlogic.github.io/.

Qvist. Dominion card survey, 2013. URL http://wiki.dominionstrategy.com/index.php/List_of_Cards_by_Qvist_Rankings.

D. Vaccarino. Dominion, 2009.

D. Zonker. Isotropic, 2013. URL https://dominion.isotropic.org/.

A. Dominion Runtime System Cards

```
data CardName = CELLAR | CHAPEL | VILLAGE | CHANCELLOR | COPPER
              | SILVER | GOLD   | ESTATE  | DUCHY      | PROVINCE
deriving (Eq, Typeable, Show, Ord)

data RuntimeCard = RuntimeCard { cID    :: !CardName
                               , cType  :: !CardType
                               , cDescr :: !CardDescr
                               , cCost  :: !CardCost }
deriving (Eq, Typeable, Show, Ord)

kingdomCards = [ RuntimeCard
                   { cID    = CELLAR
                   , cType  = ACTION
                   , cDescr = CardDescr
                                { primary = [Effect {amount = 1, effectType = ACTIONS}]
                                , other   = "Discard any number of cards. +1 Card per card discarded"
                                }
                   , cCost   = 2 }

                 , ...

                 , RuntimeCard
                   { cID    = PROVINCE
                   , cType  = VICTORY
                   , cDescr = CardDescr
                                { primary = [Effect {amount = +6, effectType = VICTORYPOINTS}]
                                , other   = []
                                }
                   , cCost = 8 }
                 ]

turnRules = [ GameParameters
              { turnID     = "Dominion_Standard"
              , turnPhases =
                [ Phase {phaseName = ActionP,  phaseInt = PhaseInt 1 }
                , Phase {phaseName = BuyP,     phaseInt = PhaseInt 1 }
                , Phase {phaseName = DiscardP, phaseInt = All       }
                , Phase {phaseName = DrawP,    phaseInt = PhaseInt 5 }
                ] } ]
```

The code above is indicative of what the runtime system is given as hyper-parameters to the model. The function `kingdomCards` specifies which cards are to be used during a single game. The `turnRules` function further specifies parameters to the underlying mechanics of Dominion. For instance in the code above the `Dominion_Standard` game parameters specify that a player draws 5 cards during her draw phase, starts with 1 action during her action phase, starts with 1 buy during her buy phase, and discards `All` of her cards during her discard phase. For an in-depth analysis of the abstractions used to define these hyper-parameters see the DeckBuild language [Cronburg et al. 2014] paper. In short, the above code is the quasiquoted [Mainland 2007] output of the embedded DSL DeckBuild.

B. Complex Card Effects - Probabilistic Implications

```haskell
type STMeasure a b = StateT a Measure b

-- Discards any number of cards, returning the number of cards discarded
cellarEffect' :: STMeasure Game Int
cellarEffect' = do
  g  <- get
  c' <- lift $ ((mayPick.p1) g) g CELLAR
  case c' of
    Just c  -> if   elem c ((cards.hand.p1) g)
               then discard c >> cellarEffect' >>= \n -> return $ n + 1
               else return 0
    Nothing -> return 0

-- Discard any number of cards, then draw that many cards:
cellarEffect :: STMeasure Game ()
cellarEffect = addActions 1 >> cellarEffect' >>= \n -> draw n
```

The code above defines the state transformation to be performed on the game when a **CELLAR** is played. When played, a **CELLAR** requires the player to pick any number of cards from her hand and discard them. For each card she discards, she draws one card from her deck and places it into her hand.

The state monad is required because the effect moves cards around between piles in the state of the game. Hakaru's Measure monad is also required because the **CELLAR** effect requires asking the player's pick-heuristic to pick a card to discard.

This example also illustrates the use of:

- `cards :: Pile -> [CardName]`,
- `hand :: Player -> Pile`
- `p1 :: Game -> Player`
- `discard :: STMeasure Game ()`
- `draw :: STMeasure Game ()`
- `addActions :: STMeasure Game ()`
- `mayPick :: Game -> CardName -> Measure (Maybe CardName)`

In the context of a game-play heuristic, the effect a Cellar has on the game state has probabilistic implications. One implication is that the number of cards a player decides to discard will indirectly determine how many more turns until she needs to reshuffle her deck. Since shuffling is a probabilistic action, this implication creates a probabilistic causal link from actions made by the player to the observable states of the game. The same probabilistic causal link exists for **VILLAGE** and **CHANCELLOR**, the two cards discussed in detail in this paper.

C. Runtime System - State Format

```
λ> runGreedy (0.5, 0.5)

Player1:
    name   = "Greedy1"
    hand   = [ESTATE, GOLD, PROVINCE, PROVINCE, SILVER]
    inPlay = []
    deck   = [SILVER,   PROVINCE, COPPER, SILVER,  COPPER
             ,ESTATE,   COPPER,   COPPER, VILLAGE, VILLAGE
             ,PROVINCE, ESTATE,   SILVER, COPPER]
    dscrd  = [SILVER, SILVER, SILVER, COPPER, COPPER, PROVINCE]
    buys=1, actions=1, money=0

Player2:
    name   = "Greedy2"
    hand   = [COPPER, COPPER, COPPER, COPPER, VILLAGE]
    inPlay = []
    deck   = [SILVER,   SILVER, GOLD, COPPER, COPPER,   ESTATE
             ,GOLD,     ESTATE, GOLD, ESTATE, PROVINCE, VILLAGE
             ,PROVINCE, GOLD]
    dscrd  = [SILVER, COPPER, SILVER, SILVER, SILVER, PROVINCE]
    buys=1, actions=1, money=0

Trash: []
Supply: [(COPPER,60),  (CELLAR,10),  (MOAT,10),       (ESTATE,8)
        ,(SILVER,27),  (VILLAGE,6),  (WOODCUTTER,10), (WORKSHOP,10)
        ,(MILITIA,10), (REMODEL,10), (SMITHY,10),     (MARKET,10)
        ,(MINE,10),    (DUCHY,8),    (GOLD,25),       (PROVINCE,0)]
Turn #: 30
```

Above is a snapshot of the state of the runtime system after two greedy models played a game against each other. The models are given direct access to this game-state information on each turn of the game. In future iterations of the runtime system there will be a feature whereby the players can remember what they did on previous turns.

In the snapshot, we have a `Supply :: [(CardName,Int)]` where the `Int` corresponds to the number of copies of that particular card in the supply. The runtime system is implemented in this manner so as to reduce the space-complexity constant of functions which interact with the supply.

The immutable nature of data is an advantage of using Haskell as the host language for probabilistic modeling. In particular, the probabilistic heuristic models used in this work were designed in a manner which expects immutable data. This is true of probabilistic state models in general - many of the transformations done to the state are purely functional, making immutable semantics in the language useful.

D. Village and Chancellor

©[Vaccarino 2009]

©[Vaccarino 2009]

The two cards above are the **VILLAGE** and **CHANCELLOR** cards referred to throughout this paper. In this case the model parameter X is the **VILLAGE** and the model parameter Y is the **CHANCELLOR**. This is an interesting first pair of cards to look at because the first buy in a game of Dominion involving these two cards usually requires the player to choose between buying one of these two cards. Additionally these cards allow us to make a number of simplifications in the first iteration of the greedy heuristic. Namely that the optimal play heuristic for a hand which can only contain these two action cards is to always play **VILLAGE** cards first followed by playing any remaining **CHANCELLOR** cards. This simple action heuristic is optimal because it maximizes the amount of money the player has during her subsequent buy phase.