Game Aspect: An Approach to Separation of Concerns in Crowdsourced Data Management

Shun Fukusumi, Atsuyuki Morishima(✉), and Hiroyuki Kitagawa

University of Tsukuba, Tsukuba, Japan
shun.fukusumi.2011b@mlab.info, mori@slis.tsukuba.ac.jp,
kitagawa@cs.tsukuba.ac.jp

Abstract. In data-centric crowdsourcing, it is well known that the incentive structure connected to workers' behavior greatly affects output data. This paper proposes to use a declarative language to deal with both of data computation and the incentive structure explicitly. In the language, computation is modeled as a set of Datalog-like rules, and the incentive structures for the crowd are modeled as games in which the actions taken by players (workers) affect how much payoff they will obtain. The language is unique in that it introduces the game aspect that separates the code for the incentive structure from the other logic encoded in the program. This paper shows that the game aspect not only makes it easier to analyze and maintain the incentive structures, it gives a principled model of the fusion of human and machine computations. In addition, it reports the results of experiments with a real set of data.

Keywords: Crowdsourcing · Declarative languages · Databases · Separation of concerns

1 Introduction

Much crowdsourcing is *data-centric*, i.e., we ask workers to perform microtasks to enter data or to help collect, process, and manage data [5] [20]. In data-centric crowdsourcing, it is well known that the incentive structure connected to workers' behavior greatly affects output data [10]. However, for many existing data-centric declarative frameworks [6] [19] [22], the design space for the incentive structure is relatively limited. For example, each worker receives a fixed amount of payment for each task, and the payment for each task is the only parameter for the incentive space. The reason for this is that complex incentive structures are often strongly connected to the logic of applications. In addition, such incentive structures are difficult to analyze and maintain.

This paper proposes to use a declarative language to deal with both of data computation and the incentive structure explicitly. In the language, computation is modeled as a set of Datalog-like (or Prolog-like) rules, and the incentive structures for the crowd are modeled as *games* in which the actions taken by players (workers) affect how much payoff they will obtain. The language can be used to implement both microtask-based and game-style crowdsourcing applications.

J. Zdravkovic et al. (Eds.): CAiSE 2015, LNCS 9097, pp. 3–19, 2015.
DOI: 10.1007/978-3-319-19069-3_1

The language is unique in that it adopts *separation of concerns*, which is an important principle in software development. The language introduces the *game aspect* that separates the code for the incentive structures from the other logic encoded in the program. With the game aspect, the code for the incentive structure is localized and described in terms taken from game theory[1]. Game theory is known to be useful when discussing not just real "games" but any system that involves incentive structure, and the game aspect makes analyzing and maintaining the incentive structure easier. In contrast, in traditional programming abstractions, it is difficult to find what kind of game is implemented in the code.

Interestingly, a logic-based language with the game aspect provides us with a natural and principled model of integration of human and machine intelligence. In the model, the incentive structure affects workers' behavior to determine two things that cannot be determined by the machine. Workers determine: (1) which rule to execute first when we have multiple rules but the evaluation order cannot be determined by logic, and (2) what value will be entered when the values cannot be derived by logic and the stored data. We will show that given an appropriate incentive, human intelligence can be used to find an effective ordering for evaluating rules to find good results without exploring the whole search space.

As a running example, we discuss four variations of a game-style crowdsourcing application for extracting structured data from tweets. To show the potential of the language, some variations are complex: the players (workers) are asked to not only enter the extraction results, but also to enter *extraction rules to be processed by the machine*. Therefore, we obtain the extraction rules as the result of crowdsourcing. In addition, the obtained rules are used during the same crowdsourcing process so that the main contributor of the extraction is gradually changed from humans to the machine. Although there are many approaches to apply machine learning techniques to find rules [16], there are many cases wherein we cannot apply machine learning techniques, such as a case in which the rules are very complex or we need to find rules with a small set of training data. In such cases, crowdsourcing is a promising approach.

Related Work. There have been many attempts to develop languages for data-centric crowdsourcing. Most of them are SQL-like languages [6] [19] [21]. Other approaches to help develop crowdsourcing applications include toolkits (e.g., TurKit [15]) and abstractions for crowdsourcing [13][17]. Existing languages have no explicit component to describe the incentive structure, and how workers behave is out of their scope.

There are several crowdsourcing systems that introduce the notion of games. Verbosity [4] is a Game-With-A-Purpose system (GWAP) [2] that collects common-sense knowledge during gameplay. The ESP Game [1] collects tags for

[1] Strictly speaking, the game aspect localizes the code required to define the unit of games and compute feedback to workers, regardless whether the feedback serves as a meaningful incentive or not. However, the game aspect helps us analyze and maintain the incentive structure. Detail is given in Section 4.

images, wherein each player is shown an image and guesses the tag another player would enter for the image. Our paper shows that declarative languages with the game aspect description offer a simple and powerful approach to help analyze the behavior of the code.

Recently, integration of human and machine computations is a hot issue [9] [14]. This paper shows that we can use a logic-based formalization to bridge the gap between human-only and machine-only computations in a unique way.

There have been many attempts to investigate connections between game theory and logic [12]. Our language is unique in that games are defined in the code, and that the games are used to not only define the semantics of the code, but leverage human intelligence to solve problems efficiently. The literature on algorithmic game theory has addressed various aspects involving both algorithms and games, such as complexities of computing equilibrium of games [23]. We hope that results from the area are helpful in discussing computational complexity of programs involving human activities.

This paper proposes an implementation language for crowdsourcing. Applying the research results on higher-level issues [7][8] to our language is an interesting challenge. Aspect oriented programming were first introduced in [11]. Applying various results on AOP to our context, such as finding aspects (in our case, games) in the early stages of software development [25], is also our interesting future work.

Summary of Contributions. First, we introduce a declarative framework that supports the game aspect. The game aspect not only makes it easier to analyze and maintain the incentive structures, it gives a principled model of the fusion of human and machine computations. Second, with a running example, we show that the game aspect allows us to easily apply game theory to prove some properties of complex crowdsourcing applications. We believe that this is an important first step to discuss the semantics of the code involving human behavior, although the assumption that workers are rational is not always true. We also conducted experiments with a real set of data. The results show that appropriately designed complex crowdsourcing applications obtain good results, especially in terms of the quality of data extraction rules. The results are consistent with the results of theoretical analysis using the game aspect.

2 Running Example: TweetPecker

As a running example, we explain TweetPecker [18], a game-style Web application to crowdsource extracting structured data from a set of tweets to populate a relation.

Figure 1 shows the dataflow in TweetPecker. It takes as inputs (1) a set of tweets and (2) the relation (table) schema $St(\texttt{tw}, a_1, a_2, \ldots, a_N)$ to store extracted values (Figure 1(1)) where `tw` is a mandatory attribute to store tweets, and a_is are attributes to store values extracted from the tweet `tw`. Then, TweetPecker shows workers each tweet one by one and asks them to enter values for attributes a_is for the tweet. Each result is represented as a tuple (row)

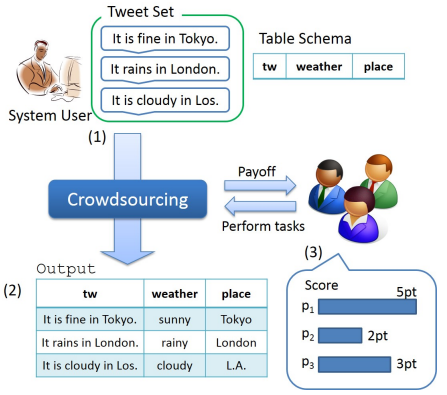

Fig. 1. Overview of TweetPecker

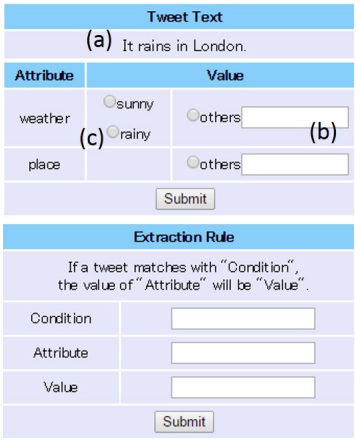

Fig. 2. Interface for entering values (top) entering extraction rules (bottom)

and inserted into the `Output` relation (Figure 1(2)). Workers receive feedbacks (Figure 1(3)) according to their inputs.

We can design variations of TweetPecker by changing what workers do and how they receive feedback. The following are four variations of TweetPecker: VE, VE/I, VRE, and VRE/I.

(1) Value-Entry (VE) crowdsources workers directly extracting values from tweets. Figure 2 (top) is the interface for workers. In VE (and VE/I we explain next), the radio buttons in Figure 2 (c) do not appear. In VE, the player (worker) is given a tweet "It rains in London" (Figure 2 (a)) and is asked to enter values for `weather` and `place` attributes into the text form (Figure 2 (b)). If she enters the values, the next tweet shows up. Each attribute value of an `Output` tuple is determined when two distinct workers give the same value. Each worker receives a fixed score (e.g., 1) whenever she performs a task regardless of the entered value.

(2) Value-Entry with Incentive (VE/I) is the same as VE except that workers receive positive scores only if their entered values match with each other (Section 4 for details).

(3) Value-Rule-Entry (VRE) allows workers to not only directly extract values from tweets but give extraction rules to be used by the machine with the interface in Figure 2 (bottom) (Section 5.1).

(4) Value-Rule-Entry with Incentives (VRE/I) is the same as VRE except that workers receive scores according to their behaviors (Section 5.2).

3 CyLog

CyLog [18] is an executable abstraction for crowdsourcing. It is a rule-based language whose syntax is similar to Datalog. CyLog is adopted by Crowd4U [27],

```
rules: Pre1: TweetOriginal(tw:"It rains in London", loc:"London");
Pre2: ValidCity(cname:"London"); Pre3: Tweet(tw) <-
TweetOriginal(tw, loc), ValidCity(cname:loc); Pre4: Worker(pid:1,
name:"Shun"); Pre5: Worker(pid:2, name:"Ken");

 VE1: Input(tw, attr:"weather", value, p)/open[p] <- Tweet(tw), Worker(p);
 VE2: Output(tw, weather:value) <- Input(tw, attr:"weather", value, p:p1),
                                   Input(tw, attr:"weather", value, p:p2), p1!=p2;
```

Fig. 3. Fragment of a CyLog program

an open crowdsourcing platform being developed and operated by universities. Crowd4U is similar to Amazon Mechanical Turk but workers (most of them are students and faculty members of universities) perform tasks voluntarily. This section first summarizes the basics of CyLog. Then, it explains the code for VE, the simplest variation of TweetPecker.

Overview. The basic data structure in CyLog is a *relation*, which is a table to deal with a set of *tuples* that conform to the *schema* of the relation. A program written in CyLog consists of four sections. The `schema` section describes the schema of relations. The `rules` section has a set of rules each of which *fires* (is executed) if its condition is satisfied. The `views` section describes the interface with workers in HTML (e.g., the ones in Figure 2). The `games` section describes the game aspect of the program (Section 4). In the following discussions, we explain only the `rules` section and the `games` section. The `schema` and `views` sections are straightforward and we assume that they are appropriately given.

Facts and Rules. The main component of a CyLog program is the set of *statements* written in the `rules` section. Figure 3 shows a set of statements, each of which is preceded by a label for explanation purposes. A statement is either a *fact* or a *rule*. In the figure, Pre1, Pre2, Pre4, and Pre5 are facts, and Pre3, VE1, VE2 are rules. A rule has the form of *head* ← *body*. Each *fact* or *head* is given in the form of an *atom*, while each atom consists of a predicate name (e.g., `Tweet`) followed by a set of *attributes* (e.g., `loc`). Optionally, each attribute can be followed by a colon with a value (e.g., `:"London"`) or an alias name (e.g., `:p1`). Each *body* consists of a sequence of atoms.

A fact describes that the specified tuple is inserted into a relation. For example, Pre1 is a fact that inserts a tuple whose values for attributes `tw` and `loc` are ``It rains in London'' and ``London'' into relation `TweetOriginal`[2].

A rule specifies that, for each combination of tuples satisfying the condition specified in the *body*, the tuple described in *head* is inserted to a relation. Atoms in the body are evaluated from left to right and variables are bound to values that are stored in the relation specified by each atom. For example, Pre3 is a rule that inserts a tuple having a tweet `tw` into relation `Tweet` if `tw` is in the `TweetOriginal` and its location is a valid city contained in `ValidCity`. In other words, for each combination of a tuple in `TweetOriginal` and a tuple in

[2] CyLog adopts the *named perspective* [3] which means that variables and values in each atom are associated to attributes by explicit *attribute names*, not by their positions in the attribute sequence.

`ValidCity` whose `loc` attributes match to each other, it inserts a tuple having `tw` value into relation `Tweet`.

Open Predicates. CyLog allows predicates to be *open*, which means that the decision as to whether a tuple exists in the relation or not is performed by humans when the data cannot be derived from the data in the database. For example, the head of VE1 is followed by `/open` and is an open predicate. If a head is an open predicate, CyLog asks humans to give values to the variables that are not bound to any values in the body (e.g., `value` in VE1)[3]. Optionally, each `/open` can be followed by `[..]` (e.g., `[p]`) to specify the worker CyLog asks for the values through the interface. Therefore, VE1 means that for each combination of a tweet `tw` and a worker `p`, the code asks the worker `p` to enter a value for the `value` attribute.

Evaluation Order. Each rule fires when its condition is satisfied. If more than one rule are ready to fire *at the same time* because all of their body conditions are satisfied, logic cannot determine in which order the rules should be executed. As with many languages, CyLog evaluates all such rules with a default ordering if the rules have no open predicates: a rule that appears earlier in the code with tuples appearing at earlier rows in relations is given higher priority. However, evaluation of any rule with an open predicate is suspended until a worker enters values for the open predicate, even if its body condition is satisfied. This allows workers to determine which rule to or not to fire, under the control by the incentive mechanism we explain in Section 4.

Block Style Rules. Each rule $P \leftarrow P_1, P_2, \ldots, P_n$ can be written in the *block style* $P_1\{P_2\{\ldots\{P_n\{P;\}\ldots\}$. For example, Pre3 in Figure 3 can be written as:

```
TweetOriginal(tw, loc) {
   ValidCity(cname:loc) {
      Tweet(tw);
   }
}
```

where `Tweet(tw)` is the head of the rule. The block style provides a concise expression when we have many rules that have the same body atoms, because we can write more than one atom inside each bracket (e.g., $P_1\{P_2; P_3;\}$ for $P_2 \leftarrow P_1; P_3 \leftarrow P_1;$).

Value-Entry: A Variation of TweetPecker. The code in Figure 3 implements VE. we assume that the relation generated by TweetPecker is `Output(tw, weather)`[4]. Pre1 to Pre3 construct a set of tweets. Pre4 and Pre5 define two workers. VE1 and VE2 implement the essential part of VE. VE1 asks the two workers to extract values for attribute `weather` from tweets (there is only one tweet in the code) and generates tuples for relation `Input`, an intermediate relation to store the workers' inputs. The omitted code in the view section shows

[3] If all variables in the head are bounded to values in the rule body, CyLog asks workers whether the tuple should exist in the relation.

[4] For simplicity, we deal with only one attribute for extracted values. It is straightforward to deal with more than one attribute.

Player A/Player B	Fine	Rainy
Fine	(1,1)	(0,0)
Rainy	(0,0)	(1,1)

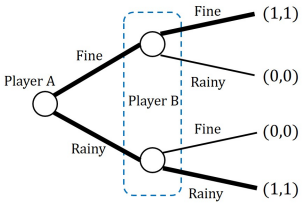

Fig. 4. Payoff matrix for a game and its extensive form

workers the tweet (Figure 2(a)) and the text input form (Figure 2(b)). VE2 states that if two different workers enter the same value for `weather` attribute of the same tweet, the agreed value is stored in the `Output`.

4 Game Aspect

Semantics of Open Predicates. Open predicates make it difficult to define the semantics of the code. First, we have no clue on what value each worker gives for open predicates. Second, the order in which we evaluate rules with open predicates is undefined. CyLog introduces a mechanism to support for describing the incentive structure at the language level to give the code a clear semantics using game theory.

The idea is to model the incentive structure as "games" in which the behaviors of players (workers) determine payoffs to them. We use VE/I (Section 2) to explain this concept. VE/I can be seen as having a collection of games, each of which is associated to one tweet: In each game, a tweet is shown to players, and each player is required to predict a term to represent the weather written in the tweet, which others would give for the same tweet. If two players give the same term, the players are rewarded, and the matched term will be stored into `Output` as the value of the `weather` attribute of the tuple for the tweet.

In game theory, a game is often written as a *payoff matrix*; Figure 4 (left) shows a part of the payoff matrix of the game (only two terms are shown in the matrix). The Y and X axis show the possible *actions* of Player A and B, respectively. The matrix shows that each player can enter `fine` or `rainy` for a given tweet. It also describes how payoffs are given to players. In each cell, (v_1, v_2) means that Players A and B receive v_1 and v_2 as their payoffs when they choose the actions on the X and Y axes. In the game, if they give the same term, they receive the payoffs. Such an incentive structure is known as a coordination game [24] in game theory.

Figure 4 (right) illustrates the same game in a tree style called the *extensive form* [24]. Each *path* from the root to a terminal node corresponds to each cell in the payoff matrix and represents a possible play of the game. The leaf nodes are associated with payoffs to the players. The dotted circle means that the player B does not know the choice Player A took for her action. Then, we can define the semantics of open values as actions in the *solutions* of the game, which are the paths taken by rational workers. For example, the solution of the game in Figure

```
games:
      VEI(tw,attr){   // Skolem function
        /* Path definition */
VEI1:   Path(action:["value",value],player:p)
              <- Inputs(tw,attr,value,p);
        /* Payoff definition */
VEI2:   Path(action,player:p1),
        Path(action,player:p2),
        p1!=p2 {
VEI2.1:      Payoff[p1+=1,p2+=1]
        }
      }
```

Fig. 5. Game aspect of VE/I

Order	Date	Player	Action
1	10:10am	Kate	["value", "fine"]
2	10:11am	Pam	["value", "rainy"]
3	10:12am	Ann	["value", "fine"]

Fig. 6. Path table

4 (right) is the paths in which the players provide the same term (bold lines), because the best strategy for them is to choose the same one that the other player would choose. To compute the payoff values for them, it is important to maintain the information on the path in each game play.

Separation of Concerns by the Game Aspect. If we write code to maintain paths of the game and to compute payoffs to players in existing programming languages, the code fragments related to the games are implicitly encoded in many different places in the code. Therefore, analyzing and changing the incentive structure will be a cumbersome task. As our example will show, the incentive structure is often complicated, which makes it almost impossible to analyze and maintain the incentive structure.

An important principle in software development is the *separation of concerns.* We propose the *game aspect*, which separates the code for the incentive structures from the other logic encoded in the program, by allowing the code to be localized and described in terms taken from game theory. Therefore, the game aspect makes analyzing and maintaining the incentive structure easy.

Figure 5 shows the game aspect of VE/I. The whole code for VE/I is the combination of the `rules` section (Figure 3) and the game aspect. Therefore, the code clearly shows that VE/I is the same as VE except its incentive structure.

A game aspect consists of three parts: a Skolem function, the path definition, and the payoff definition.

1. Skolem Function. The first line has a function named a Skolem function to create a game for each specified parameters. Intuitively, it defines the unit of games. For example, `VEI(tw,attr)` creates a `VEI` game for each combination of a tweet and one of its attributes (e.g., `weather`). We call each game a *game instance.*

For each game instance, a special table called a *path table* is automatically constructed (Figure 6). The path table maintains the *path* (i.e., a line from the root to a leaf in Figure 4 (right)) of the play of the game instance to show how the game reached the last state. Its schema is `Path(Order,Date,Player,Action)`, where each tuple records when and who took what action on the executed path. Figure 6 shows an example of the path table. Here, `["value",`*value*`]` is a list

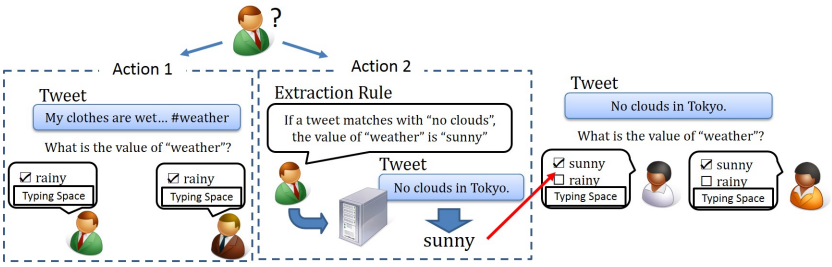

Fig. 7. VRE and VRE/I

containing two strings, which means that the action is to enter *value* for the "`value`" attribute.

2. Path Definition. The rule whose head is `Path` (i.e., VEI1) supplies tuples inserted to the path table of a game instance created for the parameters of the Skolem function (i.e., a combination of `tw` and `attr`). VEI1 states that the inputs to the `value` attribute of the relation `Inputs` are inserted into the table to be recorded as actions of players.

3. Payoff Definition. The rule whose head is `Payoff` (i.e., VEI2 to VEI2.1. Note that VEI2.1 is the head in the block style rule) computes the payoffs to players. `Payoff` is a relation that maintains payoff values to players. The head `Payoff[p1+=1, p2+=1]` is a syntactic sugar of a rule to update the values of payoffs for players (we can write rules without the syntactic sugar by writing a more complicated rule). The rule implements the same game as that in Figure 4 except that it extends the payoff matrix with an infinite number of players and terms (values).

Value-Entry with Incentive (VE/I). To summarize, the combination of the codes in Figures 3 and 5 implements VE/I. Each `VRI` game is instantiated for each combination of a tweet and an attribute. If people behave rationally, it is expected that values are computed by the solution of the coordination game. Assume that we have the path table shown in Figure 6 for a game instance, which was constructed by the game aspect in Figure 5. Then, the payoff values for `Kate`, `Pam`, and `Ann`, are 1, 0, and 1, respectively, because `Kate` and `Ann` agreed on the value. The players can see their accumulated payoff values as the scores shown in the screen (the code is omitted) (Figure 1(3)).

5 Complex Crowdsourcing

5.1 Value-Rule-Entry (VRE)

In contrast to VE and VE/I, VRE allows each worker to take two types of actions in arbitrary order (Figure 7):

```
rules:
VRE1: Rules(cond,attr,value,p)/open[p] <- Workers(p);
VRE2: Extracts(tw,attr,value,rid) <- Tweets(tw), Output(tw,weather:null),
                                     Rules(rid,cond,attr:"weather",value),
                                     matches(cond, tw);
VRE3: Tweets(tw), Workers(p){
  VRE3.1: Inputs(tw,attr:"weather",value, p)/open[p];
  VRE3.2: Inputs(tw,attr,value,p)/open[p] <- Extracts(tw,attr:"weather",value);
      }
VRE4: Output(tw, weather:value) <- Inputs(tw,attr:"weather",value,p:p1),
                                   Inputs(tw,attr:"weather",value,p:p2), p1!=p2;
```

Fig. 8. The code for VRE

Action 1: directly enter values extracted from tweets as in VE and VE/I. The default interface of VRE is the same as that for VE (Figure 2 (top)) in which the worker takes Action 1, except that it shows workers the candidate values extracted by the machine (Figure 2(c)).

Action 2: give extraction rules to be used by the machine. If the worker chooses to take Action 2, the interface is changed to the one shown in Figure 2 (bottom), in which she enters extraction rules. We allow regular expressions in the *condition* part.

The entered rules are used by the machine to extract values from the tweets whose values have not been determined yet. The values extracted from a tweet by the machine using the extraction rules are shown to other workers who are taking Action 1, as possible candidates for attribute values of the tweet (Figure 2(c)). This allows the workers to choose a shown value, instead of directly extracting and typing values in the text input form in Figure 2 (b).

Note that workers take Actions 1 and 2 in *arbitrarily order*. As we will explain in Section 5.1, the two actions are implemented by two independent CyLog rules with open predicates. And as explained in Section 3, the rules with open predicates are suspended until workers enter values, even if the both rules are ready to fire (i.e., logically, they have the same priority in the evaluation order). Therefore, the order of user's taking actions determines the order of evaluating such rules. Here, we want to utilize the intelligence of workers, because we expect that workers are able to determine whether they should directly extract values or enter extraction rules for future tweets, for efficient extraction of values. We will be back to this issue in Section 5.2.

Extraction Rules. In VRE, workers enter extraction rules in an HTML form (Figure 2 (bottom)). Logically, each extraction rule is a triple (*condition*, *attribute*, *value*). It means that if a tweet matches with the *condition*, the value of *attribute* will be *value*. For example, extraction rule ("`clear`", "`weather`", "`sunny`") means that if a tweet contains "`clear`", the value of `weather` will be "`sunny`".

CyLog Description. Figure 8 is the CyLog code for VRE. We assume that we already have valid tweets and workers. Compared to the code of VE (Figure 3), Figure 8 has two additional rules for relations `Rules` (VRE1) and `Extracts`

(VRE2). However, VRE3 and VRE4 are similar to VE1 and VE2, respectively. Details are shown below.

VRE1. The rule asks workers to enter extraction rules. The `Rules` relation maintains the extraction rules entered by workers. Its schema is `Rules(rid, cond, attr, value, p)`. Here, we have two additional attributes for management purposes: `rid` is an auto-increment key and `p` is the worker who entered the rule.

VRE2. The rule extracts a value if `Output` records no value for the `weather` attribute of a tweet `tw` and there is an extraction rule that matches with the tweet. Relation `Extracts(tw, attr, value, rid)` records the values extracted by the machine. Each tuple records the fact that an extraction rule with id `rid` extracted `value` as the value for attribute `attr` of tweet `tw`. In the omitted schema section, we define the key of the `Extracts` as a combination of `tw`, `attr`, `value` attributes. Hence the machine can extract `value` for an attribute `attr` of a tweet `tw` only once.

VRE3. The rule from VRE3 to VRE3.1 is the same as VE1 except that it is written in the block style. The rule from VRE3 to VRE3.2 deals with the case wherein the machine extracted a value and shows it in the interface in Figure 2 (c): if `Extracts` has a tuple that records `value` for "`weather`" attribute of `tw`, we ask the worker if the `value` is correct or not.

VRE4. This is the same as VE2.

5.2 Value-Rule-Entry with Incentives (VRE/I)

In VRE, there was no theoretical guarantee that the workers use their intelligence to take actions so that the results are generated efficiently and effectively, and we left the results up to fate. VRE/I is a solution to the problem. It is the same as VRE but implements the `VREI` game that defines the incentive for workers that affects workers' decision on how to interleave and take Action 1 (encoded as VRE3) and Action 2 (encoded as VRE1).

Incentive Structure of VRE/I. The incentive structure for VRE/I is as follows:

Payoffs related to Action 1: As with VE/I, the worker receives w_1 if she entered a value for a tweet in the interface in Figure 2 (top) and the value matched with the value entered by another worker. Note that if the machine used extraction rules to extract values for the tweet, candidate values show up as in Figure 2 (c). Regardless which interface the worker uses ((b) or (c)), she receives payoffs if another worker gives the same value. We call this *payoff 1*.

Payoffs related to Action 2: The worker receives w_2 if all of the following conditions hold: (1) she enters an extraction rule in the form of Figure 2 (bottom), (2) the machine uses the extraction rule to extract a value for a tweet, and (3) the value is adopted as an agreed value by two other workers with the interface

in Figure 2 (top). We call this *payoff 2a*. If there are more than one worker that satisfy the conditions above, the worker who entered the extraction rule *earliest* receives w_2. However, she loses w_3 if the value extracted by her extraction rule is *not* adopted as an agreed value. We call this *payoff 2b*.

In VE/I, we defined one game for each combination of a tweet and an attribute. However, the incentive structure of VRE/I involves all tweets and extraction rules in the process. Therefore, we cannot divide VRE/I into many small independent game instances. Instead, we define only one game for VRE/I as explained below.

CyLog Description. Figure 9 shows the game section of VRE/I (the rule section is the same as that of VRE in Figure 8). Note that since `VREI` has no parameter, there is only one VREI game instance in VRE/I.

As the incentive structure of VRE/I contains that of VE/I, the game aspect `VREI` naturally contains that of `VEI`. In the path definition, VREI1 is the same as VEI1 except that each action records `tw` and `attr`, which are used to differentiate the action from the ones for other tweets. We need to do so because `VREI` has only one game instance for all tweets and attributes. In the payoff definition, the rule from VREI3 to VREI 3.1 is the same as the rule from VEI2 to VEI2.1, and computes *payoff 1*.

Other lines deal with the case where extraction rules are involved. In the path definition, VREI2 means that the path table records an action if a worker enters an extraction rule. In the payoff definition, the rules VREI3 and VREI3.2 implement *payoff 2a*: if a worker entered an extraction rule to extract a value and other two workers agreed on the value, she receives w_2. Similarly, the rules VREI3 and VREI3.3 implement *payoff 2b*.

Note that in VREI3.2, the key of `Extract` relation is a combination of `tw`, `attr` and `value`. As explained in Section 3, the evaluation priority of CyLog rules guarantees that the extraction rule entered earliest will be used for extracting the value. Therefore, payoff is given to the worker who entered the first extraction rule that extracted `value` for `attr` of `tw`.

6 Theoretical Analysis

Since the game aspect directly describes the incentive structure embedded in the program using the terms of game theory, it is easy to analyze the game aspect of the code. Moreover, it is easy to change the incentive structure because the game aspect is separated from other logic. In this section, we prove two theorems on properties of VRE/I, which we cannot guarantee to hold without an appropriate incentive structure. We can prove the theorems easily by looking at the code in the game aspect. Due to the space limitation, the complete proofs are given in [26].

Theorem 1. (Data Quality) *Let p_1 be a worker of VRE/I who enters an extraction rule. Let p_2 and p_3 be workers who enter values for an attribute of a tweet*

```
games:
      VREI(){  /* Skolem function */
       /* Path definition */
VREI1: Path(action:["value",tw,attr,value], player:p)
                 <- Inputs(tw,attr,value,p);
VREI2: Path(action:["rule",rid], player:p) <- Rules(rid,p);
       /* Payoff definition */
VREI3: Path(action:["value",tw,attr,value],player:p2),
       Path(action:["value",tw,attr,value],player:p3),
       p2!=p3 {
VREI3.1:      Payoff[p2+=w1,p3+=w1];
VREI3.2:      Payoff[p1+=w2]
                   <- Extracts(tw,attr,value,rid),
                      Path(action:["rule",rid], player:p1),
                      p1!=p2, p1!=p3;
VREI3.3:      Payoff[p1-=w3]
                  <- Extracts(tw,attr,value:v1,rid),
                     Path(action:["rule",rid], player:p1),
                     v1!=value, p1!=p2, p1!=p3;
     }}
```

Fig. 9. Game aspect of VRE/I

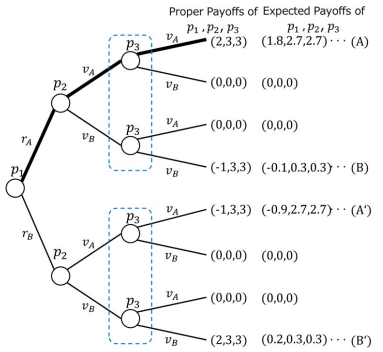

Fig. 10. Game tree of VRE/I

for which the extraction rule extracted an value. Then, if the workers behave rationally, all of them enter correct extraction rules or values.

Proof Outline. From the game aspect of VRE/I, we can easily develop the game tree for `VREI` (A fragment of the game tree is shown in Figure 10 with expected payoffs for workers). A simple game-theoretic analysis proves the theorem holds. □

Theorem 2. (Termination) *VRE/I terminates if the number of tweets is finite.*

Proof outline. In VREI3.2 of the game aspect, payment is given to only the worker who entered the first extraction rule that extracted `value` for `attr` of `tw`, because the key attributes of `Extract` are `tw`, `attr` and `value`. Thus, the number of extraction rules that yield payoffs is finite and rational workers eventually stop entering rules. □

7 Experiment

We conducted an experiment to compare the four variations of TweetPecker. Workers in our experiment were basically diligent and their extracted values were generally good in the quality even with variations without incentives for good data quality (i.e., VE and VRE). This is not surprising, because the theorem 1 does not guarantee that workers without incentives generate bad-quality data. However, an interesting finding is that *even for such diligent workers*, the incentive structure heavily affected the quality of extraction rules and workers' behavior, as supported by theorems 1 and 2. The result suggests that giving workers an appropriate incentive structure improves data quality. The comparision of this approach and other approaches such as training workers is an interesting issue but is out of the scope of this paper.

Due to the space limitation, we cannot show all results of our experiments and detailed analysis. The complete set of results is given in [26].

Table 1. Quality of acquired data

Technique		VE	VE/I	VRE	VRE/I
A: Agreed values	Correct	73.5%	72.2%	71.2%	72.0%
	Incorrect	6.7%	7.9%	7.2%	7.6%
	Neither	19.8%	19.9%	21.6%	20.4%
B: Average confidence of rules		-	-	60.9%	77.0%
C: Average support of rules		-	-	2.71%	6.32%

Method. In the first phase, we recruited four sets of five workers, and told each set of workers to work for one of VE, VE/I, VRE, and VRE/I. We used a common set of tweets as explained below. Each variation terminates when all of attribute values for all tweets are determined.

In the second phase, we asked an independent set of three persons to evaluate the quality of the extracted (agreed) attribute values. This phase was performed off-line. They discussed the quality of each value to classify it to one of "correct", "incorrect", and "neither" groups (Row A of Table 1).

Data. We collected tweets tweeted for a successive 16 days in 2013 with the tag "#tenki" (Japanese word to denote the weather). The number of tweets was 463. The relation schema was defined as `Output(tw, weather, place)`.

Quality of Generated Attribute Values and Extraction Rules. As Row A of Table 1 shows, the quality of data generated by the four variations are almost the same and the difference is not significant in statistics. The reason is that the workers are university students, and they are relatively reliable even without the incentive for improving the quality of their work.

However, an interesting fact was that even if they are reliable workers who work diligently without an incentive that is connected to the quality of their work, the quality of extraction rules became much better with an appropriate incentive structure. Given an extraction rule r_i, we compute the confidence ($conf_i$) and support (sup_i) for r_i and compared them to each other. Here, $conf_i$ and sup_i are defined as follows:

$$conf_i = \frac{\#\text{agreed values}}{\#\text{values extracted by } r_i} \quad sup_i = \frac{\#\text{tweets matched with } r_i}{\#\text{all tweets}}$$

Rows B and C of Table 1 shows that the average confidence and support for VRE/I are clearly higher than those for VRE. In fact, our statistical analysis shows that the differences are significant at the 0.01 significance level, assuming that population variances of VRE and VRE/I are the same in computing $conf_i$ and sup_i.

Workers' Behavior. For the detailed analysis, we examined the log of actions to compare the workers' behavior in both VRE/I and VRE. Figure 11 has two graphs each of which visualizes the behavior of workers for VRE or VRE/I. In each graph, the X axis is the completion rate of extracting attribute values from

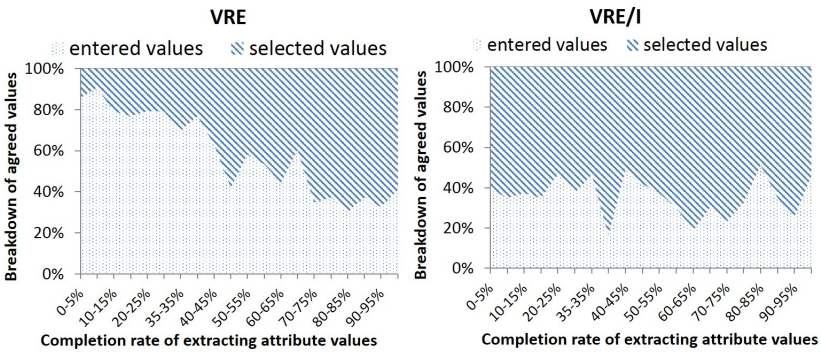

Fig. 11. Breakdown of agreed values into entered and selected values

all the tweets. The Y axis shows the breakdown of agreed values: whether each agreement was on entered values or on selected values. If we compare the two graphs, we can see that the percentage of agreement on selected values (i.e., the percentage of the values extracted by the machine, out of all adopted values) is clearly higher in the early stages in VRE/I.

We examined the log of actions in more detail [26]. Then, we found that workers in VRE/I took the strategy that they enter high-quality extraction rules in earlier stages to try (1) to maximize the number of values extracted by the rules and agreed by workers and (2) to obtain more payoff by taking Action 1 in later stages, instead of entering extraction rules that they cannot expect to give them much payoff. The strategy is consistent with Theorems 1 and 2, and in fact, generated high-quality extraction rules.

8 Conclusion

This paper introduced a declarative language that supports the game aspect for data-centric crowdsourcing. With a running example, we showed that the game aspect not only makes it easier to maintain and analyze the code using game theory, but gives a principled model of the fusion of human and machine computations. In addition, we showed experimental results with a real data set. The results are consistent with the results of the theoretical analysis, and showed that appropriately designed complex crowdsourcing applications obtain good results.

Acknowledgments. The authors are grateful to the contributors to Crowd4U, whose names are partially listed at http://crowd4u.org. They are also grateful to Prof. Yasuhiro Hayase for giving valuable comments on AOP for CyLog. This research was partially supported by PRESTO from the Japan Science and Technology Agency, and by the Grant-in-Aid for Scientific Research (#25240012) from MEXT, Japan.

References

1. Ahn, L., Dabbish, L.: ESP: labeling images with a computer game. In: AAAI Spring Symposium: Knowledge Collection from Volunteer Contributors, pp. 91–98 (2005)
2. Ahn, L., Dabbish, L.: Designing games with a purpose. Commun. ACM **51**(8), 58–67 (2008)
3. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995), ISBN 0-201-53771-0
4. Ahn, L., Kedia, M., Blum, M.: Verbosity: a game for collecting common-sense facts. In: CHI, pp. 75–78 (2006)
5. Artikis, A., Weidlich, M., Schnitzler, F., Boutsis, I., Liebig, T., Piatkowski, N., Bockermann, C., Morik, K., Kalogeraki, V., Marecek, J., Gal, A., Mannor, S., Gunopulos, D., Kinane, D.: Heterogeneous stream processing and crowdsourcing for urban traffic management. In: EDBT, pp. 712–723 (2014)
6. Franklin, M.J., Kossmann, D., Kraska, T., Ramesh, S., Xin, R.: CrowdDB: answering queries with crowdsourcing. In: SIGMOD Conference, pp. 61–72 (2011)
7. Gustas, R., Gustiene, P.: Conceptual Modeling Method for Separation of Concerns and Integration of Structure and Behavior. IJISMD **3**(1), 48–77 (2012)
8. Geiger, D., Rosemann, M., Fielt, E., Schader, M.: Crowdsourcing information systems - definition, typology, and design. In: ICIS (2012)
9. Goldberg, S.L., Wang, D.Z., Kraska, T.: CASTLE: Crowd-assisted system for text labeling and extraction. In: HCOMP (2013)
10. Jain, S., Parkes, D.C.: The role of game theory in human computation systems. In: HCOMP 2009, pp. 58–61 (2009)
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241. Springer, Heidelberg (1997)
12. Hodges, W.: Logic and Games. The Stanford Encyclopedia of Philosophy (Spring 2013 Edition), Zalta, E.N., (ed.). http://plato.stanford.edu/archives/spr2013/entries/logic-games
13. Kittur, A., Smus, B., Kraut, R.: CrowdForge: crowdsourcing complex work. In: CHI Extended Abstracts, pp. 1801–1806 (2011)
14. Kondreddi, S.K., Triantafillou, P., Weikum, G.: Combining information extraction and human computing for crowdsourced knowledge acquisition. In: ICDE, pp. 988–999 (2014)
15. Little, G., Chilton, L.B., Goldman, M., Miller, R.C.: TurKit: human computation algorithms on mechanical turk. In: UIST, pp. 57–66 (2010)
16. Langley, P., Simon, H.A.: Applications of Machine Learning and Rule Induction. Commun. ACM **38**(11), 54–64 (1995)
17. Minder, P., Bernstein, A.: CrowdLang: a programming language for the systematic exploration of human computation systems. In: Aberer, K., Flache, A., Jager, W., Liu, L., Tang, J., Guéret, C. (eds.) Social Informatics. LNCS, vol. 7710, pp. 124–137. Springer, Heidelberg (2012)
18. Morishima, A., Shinagawa, N., Mitsuishi, T., Aoki, H., Fukusumi, S.: CyLog/Crowd4U: a declarative platform for complex data-centric crowdsourcing. VLDB **5**(12), 1918–1921 (2012)
19. Marcus, A., Wu, E., Karger, D., Madden, S., Miller, R.: Human-powered Sorts and Joins. PVLDB **5**(1), 13–24 (2011)
20. Nebeling, M., Speicher, M., Grossniklaus, M., Norrie, M.C.: Crowdsourced web site evaluation with crowdstudy. In: Brambilla, M., Tokuda, T., Tolksdorf, R. (eds.) ICWE 2012. LNCS, vol. 7387, pp. 494–497. Springer, Heidelberg (2012)

21. Parameswaran, A.G., Park, H., Garcia-Molina, H., Polyzotis, N., Widom, J.: Deco: declarative crowdsourcing. In: CIKM, pp. 1203–1212 (2012)
22. Park, H., Pang, R., Parameswaran, A.G., Garcia-Molina, H., Polyzotis, N., Widom, J.: An overview of the deco system: data model and query language; query processing and optimization. SIGMOD Record **41**(4), 22–27 (2012)
23. Roughgarden, T.: Algorithmic game theory. Commun. ACM **53**(7), 78–86 (2010)
24. Vega-Redondo, F.: Economics and Theory of Games. Cambridge University Press (2003)
25. Yu, Y., Cesar, J., Leite, S.P., Mylopoulos, J.: From goals to aspects: discovering aspects from requirements goal models. In: RE, pp. 38–47 (2004)
26. Fukusumi, S., Morishima, A., Kitagawa, H.: Game Aspect: An Approach to Separation of Concerns in Crowdsourced Data Management, Technical Report. http://mlab.info/t-reports/game_aspect-full.pdf
27. Crowd4U. http://crowd4u.org