CyLog/Game Aspect: An Approach to Separation of Concerns in Crowdsourced Data Management[☆]

Atsuyuki Morishima^{a,*}, Shun Fukusumi^a, Hiroyuki Kitagawa^a

^a*University of Tsukuba
Tsukuba, Japan*

Abstract

In data-centric crowdsourcing, the output data are sensitive to the incentive structure connected to the workers' behavior. This paper proposes to use a declarative language to explicitly handle both data computation and the incentive structure. The language models computation as a set of Datalog-like rules, and the incentive structures for the crowd as games in which players' (workers') actions affect their received payoff. 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 enables easier analysis and maintenance of the incentive structures but also provides a principled model of the fusion of human and machine computations. In addition, we formally discuss how the rule-based language using the game concept integrates human and machine computations, and discuss its limitation and expressive power.

Keywords: Crowdsourcing, Declarative Languages, Databases, Separation of Concerns

1. Introduction

Much crowdsourcing is *data-centric*, i.e., workers are requested to perform microtasks to enter data or to help collect, process, and manage data [3] [24]. In data-centric crowdsourcing, it is well known that the incentive structure connected to workers' behavior greatly affects output data [13]. However, in many existing data-centric declarative frameworks [5] [21] [26], the design space of the incentive structure is relatively limited. For example, each worker receives a fixed payment for each task, which is the only parameter for the incentive

[☆]This paper is an extended version of "Game Aspect: An Approach to Separation of Concerns in Crowdsourced Data Management" published in Proc. CAiSE 2015 [6].

*Corresponding author

Email addresses: mori@slis.tsukuba.ac.jp (Atsuyuki Morishima), shun.fukusumi.2011b@mlab.info (Shun Fukusumi), kitagawa@cs.tsukuba.ac.jp (Hiroyuki Kitagawa)

space. The underlying reason 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 explicitly handle both data computation and the incentive structure. In this 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 players’ (workers’) actions affect their received payoff. The language can be used to implement both microtask-based and game-style crowdsourcing applications.

The language is unique in that it adopts *separation of concerns*, an important principle in software development. The language introduces the *game aspect*, which separates the code for the incentive structures from the other logic encoded in the program. The game aspect localizes the code of the incentive structure and describes it in terms of game theory¹. Game theory is known to be applicable not merely to real “games” but to any system involving an incentive structure. The game aspect facilitates the analysis and maintenance of the incentive structure. For example, we will show that two programs implementing the same logic but different incentive structures have the same code except for their game aspects. In contrast, it is difficult to find what kind of game is implemented in the code of traditional programming abstractions.

Interestingly, adopting the game aspect in a logic-based language provides a natural and principled model of integrated human and machine intelligence. In this model, the incentive structure drives the workers to determine two items that cannot be determined by machines; the workers determine: (1) which rule to execute first in the presence of multiple rules whose evaluation order cannot be determined by logic, and (2) what value to enter when the values cannot be derived by logic and the stored data. We will show that given an appropriate incentive, human intelligence can find an effective ordering of rule evaluations, yielding good results without exploring the whole search space.

As a running example, we discuss four variations of a game-style crowdsourcing application that extracts the structured data from tweets. To show the potential of the language, some variations introduce complex situations in which the players (workers) are asked not only to enter the extraction results but also to enter extraction rules to be processed by the machine. Therefore, we obtain the extraction rules as a result of crowdsourcing. In addition, applying the obtained rules during the same crowdsourcing process, the main contributor of the extraction is gradually changed from humans to the machine. Although many machine-learning techniques are available for rule extraction [17], these techniques are not applicable in some cases; for instance, when the rules are very complex or must be extracted from a small training dataset. In such cases,

¹Strictly speaking, the game aspect localizes the code that defines the unit of games and computes the 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. Details are given in Section 5.

crowdsourcing is a promising approach.

To more precisely understand the essence of this model, we also formally discuss how the language achieves this integration. We discuss the limitation of the language and clarify its expressive power in terms of games the language can implement.

Summary of Contributions. First, we introduce CyLog, a rule-based declarative language that supports the game aspect. The game aspect not only assists the analysis and maintenance of the incentive structures but also provides a principled model of the fused human-machine computations.

Second, in a running example, we show that the game aspect allows us to easily apply game theory to prove some properties of complex crowdsourcing applications. In experiments on a real dataset, we demonstrate that appropriately designed complex crowdsourcing applications yield 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. Note that the example is intended to show the usefulness of the game aspect, and that we do not argue that the game-style crowdsourcing is always the best way to improve data quality. CyLog can also be used to implement other techniques for improving the quality of task results, such as statistics-based ones.

Third, we formally discuss the integrated model of human and machine computations along with its limitations and expressive power². We precisely define the semantics of the CyLog program, and identify the class of games implementable by CyLog. The latter is important because the games' class affects how the language can exploit human intelligence. Since formalizing the integrated machine and human computations is a challenge in itself, we believe that our paper is an important first step in discussing the integration of machine and human computations.

Our discussion assumes that workers are rational, which is not always true. However, it would be helpful to improve the code if we could find that the code performs poorly even with rational workers. Imposing various assumptions on worker models is an interesting future task.

Structure of This Paper. The remainder of this paper is structured as follows. Related work and the running example are presented in Sections 2 and 3, respectively. Section 4 overviews CyLog, a declarative language for crowdsourcing. Section 5 introduces the game aspect. Section 6 shows that CyLog with the game aspect allows us to write concise codes for complex crowdsourcing. Section 7 shows that we can prove some properties of the running example using the code. Section 8 shows the experimental results. Section 9 presents a formal discussion on the proposed model.

²This part is the main contribution of this paper as an extended version of [6].

2. Related Work

Many languages for data-centric crowdsourcing have been developed, most of which are SQL-like languages [5] [21] [25]. Other approaches to help develop crowdsourcing applications include toolkits (e.g., TurKit [20]) and abstractions for crowdsourcing [15][22]. However, existing languages have no explicit component to describe the incentive structure, and how workers behave is out of their scope.

Several crowdsourcing systems have introduced the notion of games. Verbosity [33] is a game-with-a-purpose system (GWAP) [32] that collects common-sense knowledge during gameplay. The ESP Game [31] collects tags for images, wherein each player is shown an image and guesses the tag another player would enter for that 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, the integration of human and machine computations has received much attention [8] [16]. This paper shows that a logic-based formalization can bridge the gap between human-only and machine-only computations in a unique way.

There are many kinds of systems that use rules, such as expert systems [4] and deductive databases [27]. The game aspect can live with some of the concepts embedded in these systems, and CyLog is not the only way to implement the game aspect. We intend to present CyLog as an example of practical language for achieving the integration of human and machine computations. Some rule-based frameworks deal with incomplete data [28]. They try to make the best use of incomplete or unsure data to answer queries. As we explain in Section 9, CyLog is different from such frameworks in that it uses sure data alone to evaluate rules' bodies.

The connections between the game theory and logic have also been extensively investigated [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 discusses various issues involving both algorithms and games such as the complexities of computing equilibrium of games [29]. We hope that results from the area are helpful in discussing computational complexity of programs involving human activities.

In Section 9, we prove that CyLog can implement games expressed by Turing machines that can interact with humans at any step of their execution. The result is interesting because it suggests a possible connection between CyLog and Wegner's interaction machines [34] that extend Turing machines with input and output actions to support dynamic interaction with external environments during the execution. Exploring the connection is interesting future work.

This paper proposes an implementation language for crowdsourcing. Applying the results of higher-level issues [11][7] to our language is an interesting challenge. Aspect oriented programming (AOP) was first introduced in [14]. Applying various results on AOP to our context, such as finding aspects (in

Figure 1: Overview of TweetPecker

Figure 2 shows the interface for entering values and extraction rules. The top section, 'Tweet Text', shows a tweet "It rains in London." and a form for entering values for 'weather' and 'place' attributes. The bottom section, 'Extraction Rule', shows a form for defining extraction rules with fields for 'Condition', 'Attribute', and 'Value'.

Figure 2: Interface for entering values (top) entering extraction rules (bottom)

our case, games) in the early stages of software development [35], is also our interesting future work.

3. Running Example: TweetPecker

As a running example, we explain TweetPecker [23], 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 $St(\mathbf{tw}, a_1, a_2, \dots, a_N)$ to store extracted values (Figure 1(1)) where $\mathbf{tw}$ is a mandatory attribute to store tweets, and a_i s are attributes to store values extracted from the tweet $\mathbf{tw}$. Then, TweetPecker shows workers each tweet one by one and asks them to enter values for attributes a_i s for the tweet. Each result is represented as a tuple (row) 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.

```

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(pid:p);
VE2: Output(tw, weather:value) <- Input(tw, attr:"weather", value, p:p1),
    Input(tw, attr:"weather", value, p:p2), p1!=p2;

```

Figure 3: Fragment of a CyLog program

(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 5 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 6.1).

(4) **Value-Rule-Entry with Incentives (VRE/I)** is the same as VRE except that workers receive scores according to their behaviors (Section 6.2).

4. CyLog

CyLog [23] is an executable abstraction for crowdsourcing. It is a rule-based language whose syntax is similar to Datalog. CyLog is adopted by Crowd4U [1], an open crowdsourcing platform being developed and operated by universities³. 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 5). 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* $\leftarrow$ *body*. Each *fact* or

³Atsuyuki Morishima and Shun Fukusumi were members of the Crowd4U operation team when this research was conducted.

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`⁴.

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 `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`)⁵. 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 5.

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

```
TweetOriginal(tw, loc) {
  ValidCity(cname:loc) {
```

⁴CyLog adopts the *named perspective* [2] 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.

⁵If all variables in the head are bounded to values in the rule body, CyLog asks workers whether the tuple should exist in the relation.

Figure 4: Payoff matrix for a game and its extensive form

```

    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)`⁶. Pre1 to Pre3 construct a set of tweets⁷. 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 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`.

5. 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 the description of 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 3) 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,

⁶For simplicity, we deal with only one attribute for extracted values. It is straightforward to deal with more than one attribute.

⁷We used Pre1 to Pre3 to explain facts and rules in CyLog. The validation step implemented by them is not essential for TweetPecker.

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 [30] in game theory.

Figure 4 (right) illustrates the same game in a tree style called the *extensive form* [30]. 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 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 choose⁸. 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

⁸The semantics of a CyLog program is precisely defined in Section 9.

```

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

```

Figure 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"]

Figure 6: Path table

games. For example, $\text{VEI}(\text{tw}, \text{attr})$ 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 $\text{Path}(\text{Order}, \text{Date}, \text{Player}, \text{Action})$, where each tuple records when and who took what action on the executed path. Figure 6 shows an example of the path table. Here, $[\text{"value"}, \text{value}]$ is a list 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 $\text{Payoff}[p1+=1, p2+=1]$ 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)).

Figure 7: VRE and VRE/I

6. Complex Crowdsourcing

6.1. 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):

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 6.1.2, the two actions are implemented by two independent CyLog rules with open predicates, which can be ready to fire at the same time. However, as explained in Section 4, the rules with open predicates are suspended until workers enter values. Therefore, the order of user’s taking actions determines the order of evaluating the 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 6.2.

6.1.1. 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

```

rules:
VRE1: Rules(rid,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;

```

Figure 8: The code for VRE

value. For example, extraction rule (“clear”, “weather”, “sunny”) means that if a tweet contains “clear”, the value of *weather* will be “sunny”.

6.1.2. 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.

6.2. 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).

6.2.1. 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.

6.2.2. 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 the **Extract** relation is a combination of **tw**, **attr** and **value**. As explained in Section 4, the evaluation priority of CyLog rules guarantees that the extraction rule entered earliest will be used for

```

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;
    }}

```

Figure 9: Game aspect of VRE/I

Figure 10: Game tree of VRE/I

extracting the value. Therefore, payoff is given to the worker who entered the first extraction rule that extracted `value` for `attr` of `tw`.

7. 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.

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 for which the extraction rule extracted a 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 workers⁹). A simple game-theoretic analysis proves the theorem holds. $\square$

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. $\square$

⁹The expected payoffs in Figure 10 is computed assuming that the probability that workers enter a correct value is 0.9.

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%

8. 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 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 comparison of this approach and other approaches such as training workers is an interesting issue but is out of the scope of this paper.

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. They were paid according to the scores they obtained; We paid each worker 200 JPY as a basic fee, and divided 3,000 JPY among the workers in proportion to their scores.

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,

Figure 11: Breakdown of agreed values into entered and selected values

$conf_i$ and sup_i are defined as follows:

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

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 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.

Figure 12 illustrates when workers entered extraction rules in VRE and VRE/I. Again, the X axis is the completion rate of extracting attribute values from all the tweets. In VRE, workers enter extraction rules at any time, while workers in VRE/I enter most extraction rules at the beginning stage. This is because 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.

Figure 12: Number of extraction rules entered at each period

9. Formal Model for Integration of Human and Machine Computations and its Limitations

Thus far, we have informally explained the ability of CyLog with the game aspect to integrate human and machine computations, deferring precise discussions on what constitutes this integration. This section answers two questions. The first question is how to define the integration of human and machine computations achieved by the framework. To answer the question, we develop a formal model that precisely defines the semantics of CyLog programs. The second question is the class of games that can be implemented by the language. As we have seen, the games a language can implement affect how much human intelligence we can exploit in the program. Later in this section, we show that (1) the class to which VRE/I belongs is strictly larger than that to which VE/I belongs, (2) a sufficient condition to implement games in the former class is that the language is Turing complete and allows interactions with humans at any execution step, and (3) CyLog can implement the former class.

This section proceeds as follows. We first explain the machine computation perspective of CyLog, which is important for discussing the other issues. We then explain how CyLog integrates human and machine computations and discuss the limitation of the approach. Finally, we identify the class of games that can be implemented by CyLog.

9.1. CyLog Rules without Open Predicates

CyLog statements (facts and rules) *fire* according to their logical dependencies: Every fact fires once¹⁰ without any conditions and a tuple is inserted into a relation. A rule fires if there exist tuples such that all atoms in the rule body are true. Atoms in the body are evaluated from left to right and variables are bound to attributes of tuples stored in the relation specified by each atom.

¹⁰We record what facts have already fired in P in the execution.

We call the result of binding variables to values a *valuation*. We call a rule with a valuation a *rule instance*. For example, a rule $R(x:1, y) \leftarrow P(y)$; with a valuation $y=10$ is a rule instance of the rule. Note that a rule always fires with a valuation (i.e., a rule instance fires), but we do not have to instantiate a fact, since it has no body atoms that bind variables to values. As we explained, the evaluation of any statement with an open predicate is suspended until a human enters values (i.e., gives a valuation) for the open predicate. Before discussing this issue, we explain the evaluation order of rules *without* open predicates.

Conflict resolution. When there are several rules that are ready to fire at the same time, we need to decide the first-evaluated rule. If the rules contain none of negation, update and deletion, the evaluation order does not matter [2]. However, if the language supports any of them, a special treatment is required. In CyLog, “ $R(x:1)/\text{delete};$ ” deletes the tuple $R(x:1)$, and “ $R(\underline{x}:1, y)/\text{update} \leftarrow P(y), \text{ not } Q(y);$ ” is an example of a rule involving negation and update. This states that if there exists a valuation of y (e.g., $y=10$) such that $P(y)$ exists but $Q(y)$ does not in the database, the tuple $R(\underline{x}:1, y:10)$ is generated and the existing tuple whose key x is 1 is replaced with that fact. Therefore, if more than one rule are ready to fire at a time, the evaluation order affects the results and a *conflict resolution* is required [10].

CyLog adopts simple principles for conflict resolution. First, the CyLog rules are prioritized. Logically, each fact or rule r is associated with a number p that represents its evaluation priority in an ascending order (figure 13)¹¹. If more than one rule are ready to fire at the same time, the rule with the highest (smallest) priority is evaluated. If there are multiple valuations for the same rule, we select the rule instance valued by the tuples appearing in the earliest rows of the body relations (we check tuples for body atoms from left to right to break the tie). If a fact or rule instance has the highest priority, we call it a *succeeding* fact or rule instance. This is a variation of the closed-loop hierarchical linear strategy [19]. Figure 13 shows a possible evaluation order of the shown rules.

Second, we accept non-monotonicity in the evaluation. The result of a rule r is valid since r is evaluated until the processor obtains an inconsistent result with it. For example, in the code shown in Figure 13, $T(x:1)$ holds in the period since rule 3 (with $x=1$) is evaluated until rule 6 is evaluated.

Dataflow among Rules. The evaluation order of CyLog rules implies that when we process a rule r for the first time, the available tuples for evaluating r are generated only by the higher-priority rules. Tuples that need to be generated by rules including lower-priority ones will arrive at r later and r will be incrementally evaluated with the newly arrived tuples.

Such dataflow among rules can be illustrated by a *rule precedence graph*. For example, Figure 14 graphs the rules in Figure 13. Intuitively, an arrow means that the result of a rule depends on that of another rule. In particular, a dotted

¹¹For simplicity, we consider only flat-style CyLog rules. The priority policy can be generalized to block-style rules but we omit the generalization here.

```

1. R(x:1);
2. U(x:2)
3. T(x) <- R(x), not U(x);
4. S(x,y)/open <- R(x);
5. R(x:2);
6. T(x:1)/delete;

```

Evaluation Order: 1, 2, 3 (x=1), 4 (x=1), 5, 3 (x=2), 4 (x=2), 6

Figure 13: Possible evaluation order of a CyLog code

Figure 14: Precedence graph of the rules in Figure 13

arrow means that tuples will be supplied to a rule after the first-time evaluation of the rule. The graph is drawn as follows.

- Each vertex R_q corresponds to the head predicate R of a rule whose priority number is q , i.e., vertices and rules have a one-to-one correspondence.
- Let R_q and R'_i be vertices of the graph. We draw an arrow from R'_i to R_q in the following cases.
 - R_q corresponds to the rule “ $R(u) \leftarrow \dots R'(u) \dots$ ” (or “ $R(u) \leftarrow \dots \text{not } R'(u) \dots$ ”) (e.g., $R_1 \rightarrow S_4$).
 - R_q corresponds to the rule “ $R(u)$ /update;” or “ $R(u)$ /delete;”, $R' = R$ and $i < q$ (e.g., $T_3 \rightarrow T_6$).

If the priority value q is larger (i.e., lower priority) than i , the arrow represents a *forward precedence* and is expressed as a solid arrow. If not, it represents a *backward precedence* and is expressed as a dotted arrow.

R_q is said to *depend on* R'_i if there is a direct or indirect (composite) dataflow from R'_i to R_q . For example, in Figure 14, T_6 depends on R_1 . Note that if there is no dependency between rules r_1 and r_2 (for example, Rule 3 (T_3) and Rule 4 (S_4)), they can be evaluated in parallel.

Note on Negation. The dataflow implies that the result of a negation (e.g., `not U(x)`) in r ’s body is affected by r ’s position in the list of rules, because the computed result is based on what does not exist *at the time of evaluating the rule*. In contrast, the negation semantics in declarative languages is often

defined by checking whether a tuple (fact) exists in the *final set of tuples*. In such a language, `not U(x)` with `x=10` becomes true only when the tuple `U(10)` does not exist in the final set of tuples. Negation in CyLog is compatible with the semantics when a CyLog rule with negation in its body is *data complete*. The evaluation of a rule r is data complete if all computations that affect the result of r have been completed before evaluating r . Whether the evaluation of r is data complete or not is easily determined from the data precedence graph: given an r with head R_q , if there is no vertex R'_i such that $q \leq i$ and R_q depends on R_i , the data are complete when evaluating r . For example, the evaluation of rule 6 for T_6 is data complete. The notion of data completeness is closely related to the stratified semantics of Datalog [2].

9.2. Integration of Human and Machine Computation

Let P be a CyLog program, C be a set of workers defined in P^{12} , and S be a function that maps C to their *strategies* which define the workers' behavior in games. S represents the strategies chosen by humans in C according to the game rules encoded in the game aspect of P . For example, let P be the code for VE/I shown in Figures 3 and 5. Then, $C = \{\text{Shun}, \text{Ken}\}$. If the players know that they are rewarded only when their values agree, S can map both players to the same strategy; namely, entering a value that others are likely to enter.

Let $\text{sch}(P)$ be the set of relation schemas defined by P , and K be the set of all tuples of an instance over $\text{sch}(P)$ where $K = K_{\text{sure}} \oplus K_{\text{open}}$. Here, K_{open} and K_{sure} are sets of *open tuples* (with open values) and *sure tuples* (with no open values), respectively. Assume that K is the set of tuples when we have just finished evaluating `Pre2` of the program for VE/I (Figure 3). Then, K_{sure} contains two sure tuples (one tuple for each of `TweetOriginal` and `ValidCity`). K_{open} is empty.

Immediate Logical Consequences. Given a program P and a set K of tuples, an immediate logical consequence is a tuple that can be immediately derived from the result of evaluating a statement in P on K . There are two types of immediate logical consequences; sure and open consequences. A tuple A is an *immediate sure consequence* of P on K if one of the followings holds. (1) $A \in K_{\text{sure}}$, (2) A is a succeeding fact in P , or (3) $A \leftarrow A_1, \dots, A_n$ is a succeeding rule instance in P and each A_i is in K_{sure} . Assuming that K is the set of sure and open tuples immediately after evaluating `Pre2`, an instance of `Pre3` generates `Tweet(tw:"IT rains in London")` as an immediate sure consequences of P on K . Similarly, a fact A is an *immediate open consequence* of P on K , if $A \in K_{\text{open}}$, A/open is a succeeding fact in P , or $A/\text{open} \leftarrow A_1, \dots, A_n$ is a succeeding rule instance in P and each A_i is in K_{sure} . For example, if K is the set of tuples immediately after evaluating `Pre5`, `VE1` generates `Input(tw:"It rains in London", attr:"Weather", value:open, p:"Shun")` as an immediate open consequence.

¹²Note that workers can be explicitly specified in open predicates (Section 4). If there is no worker specification in P , we assume that a worker is anybody accessible from the code.

Figure 15: Immediate Logical and Human Consequences

If a rule instance contains a negated atom `not` A_i in its body, it returns true if A_i is not in K_{sure} . This way, CyLog considers no open tuple in evaluating atoms and adopts the two-valued closed world assumption based on sure tuples.

Immediate Human Consequences. Given a program P that defines a set C of people, their strategies S and a set K of tuples, a tuple A is an immediate human consequence of C and S on K , if A is the result of valuating open values of a tuple in $K_{open}(\subseteq K)$ where the valuation is given by a human $c \in C$ following strategy $S(c)$. Note that the open tuple and its valuation are determined solely by C and S in the given games, not by any particular rule in P . If K_{open} includes `Input(tw: "It rains in London", attr: "Weather", value: open, p: "Shun")`, a possible immediate human consequence is `Input(tw: "It rains in London", attr: "Weather", value: "rainy", p: "Shun")`.

Integrated Consequences. Immediate integrated consequences are the union of all immediate logical and human consequences, excluding open tuples for which human valuations generate immediate human consequences.

Figure 15 illustrates the computation of integrated consequences. The inner circle in this figure represents the given $K(= K_{sure} \oplus K_{open}(\text{including } \Delta K'_2))$ where $\Delta K'_2$ denotes the open tuples originally in K_{open} , which will be valuated to generate immediate human consequences. The outer circle represents the set of immediate integrated consequences of P and S on K . Here, $K_{sure} \cup \Delta K'_1$ denotes the immediate logical sure consequences in which $\Delta K'_1$ are generated by the facts and rules in P . Similarly, $K_{open} \cup \Delta K'_{open}$ denotes the immediate logical open consequences. The set of *immediate integrated consequences* is then defined as $K' = K'_{sure} \oplus K'_{open}$ where $K'_{sure} = K_{sure} \cup \Delta K'_1 \cup \Delta K'_2$ (with valuations) and $K'_{open} = K'_{open} \cup K_{open} - \Delta K'_2$ (without valuations). Note that the CyLog rules determine $\Delta K'_1$ and $\Delta K'_{open}$ by logic, but humans determine $\Delta K'_2$ by their strategies in the given games. Therefore, in general, not all tuples in K_{open} are valuated. We define the immediate integrated consequence operator of P and S , denoted by $T_{P,S}$, such that $T_{P,S}(K)$ returns the set of all immediate integrated consequences of P and S on K .

Rational Behavior. Let $T_{P,S}^*(\phi)$ denote an infinite number of applications

of $T_{P,S}$ to the empty set (i.e., $K = \phi$). We call the sequence of results of each $T_{P,S}$ application as the *behavior* of P and S . If the sequence has a fixpoint K such that $T_{P,S}(K) = K$, we refer to K as the *conclusion* of P and S .

Since humans are involved in the evaluation, there are many possible strategies and corresponding code behaviors. In game theory, *rational* humans are those who adopt their best response strategies to maximize their own profits/utilities, and *solutions* are predictions on how the games will be played. For example, a solution can be represented by a Nash equilibrium [30], a combination of strategies in which no player deviates from her strategy when every player knows the strategies of all other players. If S is a solution of the implemented games played by rational humans, we call the behavior and the conclusion (if any) of P and S as a *rational behavior* and a *rational conclusion*, respectively.

Using game theory, we can discuss the rational behaviors of a CyLog program. For example, when analyzing TweetPecker, we used the solution concept to discuss the data quality and termination of the VRE/I program. The class of games that a language can implement is an important factor and will be discussed in Section 9.4. In general, a CyLog program can exhibit more than one rational behavior, as games can have more than one solution. Therefore, we define the semantics of a CyLog program as the set of its possible rational behaviors.

To summarize, our model for integrating human and machine computations operates as follows: (1) based on logical rules, the machine derives the sure and open tuples from existing sure tuples, (2) when the open tuples are valuated by humans, the rules with open predicates generate sure facts, and (3) the incentive structure embedded in the games exploits human intelligence to determine the order of the rule evaluation, and what values to be used for valuation when these cannot be determined by logic.

9.3. Discussion

Merits and Limitations. Despite its simplicity, the game concept approach to a rule-based language offers several merits for integrating machine and human computations. First, human computation is often asynchronous and the rule-based code imposes no unnecessary timing constraints. Second, complex human-machine interactions often require event-driven executions that rule-based languages can implement in a straightforward way. The model also maximizes parallel human-machine interactions, because every logically-derived open value at any time is always included in K_{open} . This property is desired in crowdsourcing settings, where a large number of people might join the computation in parallel. The major limitation of CyLog is that humans cannot add facts that are non-existent in K_{open} . This means that CyLog cannot exploit human intelligence beyond the programmer’s expectation.

Relationship between CyLog and Incomplete Databases. An open tuple becomes a sure tuple only if a human argues that it holds with a supplied valuation. We emphasize that CyLog does not use open tuples for logical inferences as the immediate logical consequence utilizes sure tuples only. Until the open tuples are valuated, we assume there are no such tuples and do not

consider any possible worlds suggested by the open tuples. In this way, CyLog fundamentally differs from databases that deal with incomplete information [28], which use the incomplete data as far as possible when answering queries.

9.4. Expressive Power for Exploiting Human Intelligence

The class of games that a program language can implement is important, because it defines how the system can exploit human intelligence. Here, we define two games' classes, $\mathcal{G}_N$ and $\mathcal{G}_*$ ¹³, and show that CyLog can implement games in the latter class $\mathcal{G}_*$.

In the following definitions, a *phase of interactions* contains a set of human-machine interactions in which the interactions are invisible to each other. For example, asking humans to assign tags to a presented picture without informing them of others' tag assignments constitutes a phase of interactions.

Definition 1. $\mathcal{G}_N$ is a class of games in which (1) $N(> 0)$ is known in advance; (2) each game has at most N -phases of interactions, each of which asks workers to enter data; (3) at each i -th phase workers are shown some information based on what was entered in the first to the $(i - 1)$ -th phases; and (4) payoffs are computed by a μ recursive function of the entered values in the first to the N -th phases. $\square$

For example, each VE/I game defined by the game aspect $\text{VE/I}(\text{tw}, \text{attr})$ in Figure 5 is in $\mathcal{G}_N$ ($N = 1$), because it comprises one phase of interactions and the payoffs can be computed from the entered values. Another example is a game that obtains an organization logo embodying the concept of the organization, which requires two phases of interactions. In the first phase, some of the workers are shown text explaining the concept and are asked to design logos that embody the concept. In the second phase, another set of workers is requested to vote for the logo that best embodies the concept. Workers in the first set receive payoffs depending on the vote count for their logos. Workers in the second set receive payoffs if their votes were in majority.

In contrast, $\mathcal{G}_*$ is the class of games in which we cannot bound the number of phases of interactions.

Definition 2. $\mathcal{G}_*$ is a class of games in which (1) each game executes a sequence of interactions with workers, with each step being generated by a μ recursive function of the entered values, (2) at each interaction workers are shown some information computed by a μ recursive function of the entered values, and (3) payoffs are computed by a μ recursive function of the entered values. $\square$

$\mathcal{G}_*$ is strictly larger than $\mathcal{G}_N$, in which the number of interaction phases is bounded by N . Intuitively, $\mathcal{G}_*$ contains indefinite-length sequential games (in game theory) and each interaction step in a class $\mathcal{G}_*$ game is generated by

¹³The definitions are the revised versions of the ones given in [9].

```

TuringMachine(id:1, st:s, head:0);
Tape(pos:head)/update <- TuringMachine(id, head);
TuringMachine(id, st, head), Tape(pos:head, sym),
  Rule(st, sym, new_st, new_sym, dir), new_pos = pos + dir {
    TuringMachine(id, st:new_st, head:new_pos)/update,
    Tape(pos, sym:new_sym)/update
  }

```

Figure 16: CyLog rules implementing a Turing machine

a μ recursive function of the entered values of past interactions. The game implemented in VRE/I is in $\mathcal{G}_*$ because the path is generated by a μ recursive function of the past interaction results, and the number of extraction rules entered by workers cannot be limited¹⁴. $\mathcal{G}_*$ can be expressed by Turing machines that can interact with humans at any step of its execution, as stated by the following theorem.

Theorem 3. *Turing machines that can interact with humans at any step of their execution can implement games in $\mathcal{G}_*$.*

Proof. Since Turing machines can compute μ recursive functions, a sufficient condition to be able to compute each step of the interaction sequence in games in $\mathcal{G}_*$ is to allow Turing machines to interact with humans at any step of its execution. This is also a sufficient condition to be able to compute the information to be presented to workers and the workers' payoffs. $\square$

Note that being Turing complete is not a sufficient condition for implementing $\mathcal{G}_*$, because a language can be Turing complete without any human interactions.

Theorem 4. *CyLog can implement any games in $\mathcal{G}_*$.*

Proof. We first prove that CyLog is Turing complete by presenting the CyLog rules that implements a Turing machine (Figure 16). Formally, a Turing machine consists of a quintuple $(K, \Sigma, \delta, s, H)$ where K is a finite set of states, Σ is an alphabet, $s \in K$ is the initial state, $H \subseteq K$ is the set of halting states, and δ is the transition function [18]. Intuitively, we need the following three components to implement a Turing machine.

1. Memory of the machine's inner state,
2. Head reading and writing information stored in the tape, and
3. An infinitely long tape.

¹⁴Note, however, that rational workers are expected to eventually stop entering extraction rules as discussed in Section 7.

In Figure 16, `TuringMachine(id, st, head)` implements components 1 and 2 of a Turing machine with the specified `id`. Here, `st` records the current state over domain K , and `head` stores the position of the head. `Tape(pos, sym)` implements component 3. Each tuple (p, s) of `Tape` states that symbol s (over domain Σ) is written at position p of the tape. `Rule(st, sym, new_st, new_sym, dir)` stores the rules to implement δ that read and write symbols on the tape and move the head.

Rule 1 initializes the Turing machine (to its initial state s). Rule 2 extends the tape when the head reaches a previously unvisited position. Rule 2 is necessary because Rule 3 requires that `Tape(pos, sym)` always exists. Rule 3 dictates the head movement and writes the symbols to the tape, following the rules stored in `Rule(st, sym, new_st, new_sym, dir)`. For an inner state `st` and a symbol `sym` at the current head position, Rule 3 writes `new_sym` at the head position, updates the inner state to `new_st`, and moves the head to `pos+dir`. We also need a rule that stops the machine when it reaches the halting states $H \subseteq K$. The rule is straightforward and omitted.

The third rule (implementing the state transition) can be rewritten to generate open values and receive human valuations of open values. Therefore, the code satisfies the sufficient condition to implement the sequence of interactions allowed in $\mathcal{G}^*$. $\square$

Interestingly, not all abstractions for crowdsourcing can implement games in $\mathcal{G}_*$. The classes of games that can be implemented by other abstractions are presented in [9].

10. 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 also provides a principled model of the fusion of human and machine computations. We showed experimental results with a real dataset for the running example. The results are consistent with those of the theoretical analysis, and showed that appropriately designed complex crowdsourcing applications obtain good results. In addition, we formally discussed the principled model for integrating human and machine computations using the game concept, and discussed its limitation and expressive power.

Acknowledgements

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 and Prof. Yuko Sakurai for giving valuable comments. 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.

- [1] Crowd4u. <http://crowd4u.org>.
- [2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [3] Alexander Artikis, Matthias Weidlich, François Schnitzler, Ioannis Boutsis, Thomas Liebig, Nico Piatkowski, Christian Bockermann, Katharina Morik, Vana Kalogeraki, Jakub Marecek, Avigdor Gal, Shie Mannor, Dimitrios Gunopulos, and Dermot Kinane. Heterogeneous stream processing and crowdsourcing for urban traffic management. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 712–723, 2014.
- [4] Daniel G. Bobrow, Sanjay Mittal, and Mark Stefik. Expert systems: Perils and promise. *Commun. ACM*, 29(9):880–894, 1986.
- [5] Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. Crowddb: answering queries with crowdsourcing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 61–72, 2011.
- [6] Shun Fukusumi, Atsuyuki Morishima, and Hiroyuki Kitagawa. Game aspect: An approach to separation of concerns in crowdsourced data management. In *Advanced Information Systems Engineering - 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8-12, 2015, Proceedings*, pages 3–19, 2015.
- [7] David Geiger, Michael Rosemann, Erwin Felt, and Martin Schader. Crowdsourcing information systems - definition, typology, and design. In *Proceedings of the International Conference on Information Systems, ICIS 2012, Orlando, Florida, USA, December 16-19, 2012*, 2012.
- [8] Sean Louis Goldberg, Daisy Zhe Wang, and Tim Kraska. CASTLE: crowd-assisted system for text labeling and extraction. In *Proceedings of the First AAAI Conference on Human Computation and Crowdsourcing, HCOMP 2013, November 7-9, 2013, Palm Springs, CA, USA*, 2013.
- [9] Kenji Gonnokami, Atsuyuki Morishima, and Hiroyuki Kitagawa. Condition-task-store: A declarative abstraction for microtask-based complex crowd-sourcing. In *Proceedings of the First VLDB Workshop on Databases and Crowdsourcing, DBCrowd 2013, Riva del Garda, Trento, Italy, August 26, 2013*, pages 20–25, 2013.
- [10] Anoop Gupta, Charles Forgy, Allen Newell, and Robert G. Wedig. Parallel algorithms and architectures for rule-based systems. In *Proceedings of the 13th Annual Symposium on Computer Architecture, Tokyo, Japan, June 1986*, pages 28–37, 1986.

- [11] Remigijus Gustas and Prima Gustiené. Conceptual modeling method for separation of concerns and integration of structure and behavior. *IJISMD*, 3(1):48–77, 2012.
- [12] Wilfrid Hodges. Logic and games. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2013 edition, 2013.
- [13] Shaili Jain and David C. Parkes. The role of game theory in human computation systems. In *Proceedings of the ACM SIGKDD Workshop on Human Computation, Paris, France, June 28, 2009*, pages 58–61, 2009.
- [14] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.
- [15] Aniket Kittur, Boris Smus, and Robert Kraut. Crowdforge: crowdsourcing complex work. In *Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Extended Abstracts Volume, Vancouver, BC, Canada, May 7-12, 2011*, pages 1801–1806, 2011.
- [16] Sarath Kumar Kondreddi, Peter Triantafillou, and Gerhard Weikum. Combining information extraction and human computing for crowdsourced knowledge acquisition. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 988–999, 2014.
- [17] Pat Langley and Herbert A. Simon. Applications of machine learning and rule induction. *Commun. ACM*, 38(11):54–64, 1995.
- [18] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation (2. ed.)*. Prentice Hall, 1998.
- [19] Antoni Ligeza. *Logical Foundations for Rule-Based Systems, 2nd Ed.*, volume 11 of *Studies in Computational Intelligence*. Springer, 2006.
- [20] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkkit: human computation algorithms on mechanical turk. In *Proceedings of the 23rd Annual ACM Symposium on User Interface Software and Technology, New York, NY, USA, October 3-6, 2010*, pages 57–66, 2010.
- [21] Adam Marcus, Eugene Wu, David R. Karger, Samuel Madden, and Robert C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.
- [22] Patrick Minder and Abraham Bernstein. Crowdlang: A programming language for the systematic exploration of human computation systems. In *Social Informatics - 4th International Conference, SocInfo 2012, Lausanne, Switzerland, December 5-7, 2012. Proceedings*, pages 124–137, 2012.

- [23] Atsuyuki Morishima, Norihide Shinagawa, Tomomi Mitsuishi, Hideto Aoki, and Shun Fikusumi. Cylog/crowd4u: A declarative platform for complex data-centric crowdsourcing. *PVLDB*, 5(12):1918–1921, 2012.
- [24] Michael Nebeling, Maximilian Speicher, Michael Grossniklaus, and Moira C. Norrie. Crowdsourced web site evaluation with crowdstudy. In *Web Engineering - 12th International Conference, ICWE 2012, Berlin, Germany, July 23-27, 2012. Proceedings*, pages 494–497, 2012.
- [25] Aditya G. Parameswaran, Hyunjung Park, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. Deco: declarative crowdsourcing. In *21st ACM International Conference on Information and Knowledge Management, CIKM’12, Maui, HI, USA, October 29 - November 02, 2012*, pages 1203–1212, 2012.
- [26] Hyunjung Park, Richard Pang, Aditya G. Parameswaran, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. An overview of the deco system: data model and query language; query processing and optimization. *SIGMOD Record*, 41(4):22–27, 2012.
- [27] Kotagiri Ramamohanarao and James Harland. An introduction to deductive database languages and systems. *VLDB J.*, 3(2):107–122, 1994.
- [28] Simon Razniewski and Werner Nutt. Completeness of queries over incomplete databases. *PVLDB*, 4(11):749–760, 2011.
- [29] Tim Roughgarden. Algorithmic game theory. *Commun. ACM*, 53(7):78–86, 2010.
- [30] Fernando Vega-Redondo. *Economics and Theory of Games*. Cambridge University Press, 2003.
- [31] Luis von Ahn and Laura Dabbish. ESP: labeling images with a computer game. In *Knowledge Collection from Volunteer Contributors, Papers from the 2005 AAAI Spring Symposium, Technical Report SS-05-03, Stanford, California, USA, March 21-23, 2005*, pages 91–98, 2005.
- [32] Luis von Ahn and Laura Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8):58–67, 2008.
- [33] Luis von Ahn, Mihir Kedia, and Manuel Blum. Verbosity: a game for collecting common-sense facts. In *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006, Montréal, Québec, Canada, April 22-27, 2006*, pages 75–78, 2006.
- [34] Peter Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5):80–91, May 1997.

- [35] Yijun Yu, Julio Cesar Sampaio do Prado Leite, and John Mylopoulos. From goals to aspects: Discovering aspects from requirements goal models. In *12th IEEE International Conference on Requirements Engineering (RE 2004)*, 6-10 September 2004, Kyoto, Japan, pages 38–47, 2004.