Condition-Task-Store: A Declarative Abstraction for Microtask-based Complex Crowdsourcing

Kenji Gonnokami
University of Tsukuba
s1320687@u.tsukuba.ac.jp

Atsuyuki Morishima
University of Tsukuba
mori@slis.tsukuba.ac.jp

Hiroyuki Kitagawa
University of Tsukuba
kitagawa@cs.tsukuba.ac.jp

ABSTRACT

Microtasks have been widely adopted by many crowdsourcing platforms as a unit for human computation. Recently, tools to support programmers to implement complex crowdsourcing applications with microtasks have been proposed. One approach is to provide a library of functions that can be called by programs written in imperative programming languages. Another approach is to allow SQL queries to invoke microtasks. The former approach provides large expressive power, while the latter allows declarative descriptions with limited expressive power. This paper proposes the Condition-Task-Store (CTS) abstraction, which is an alternative declarative approach to implement complex data-centric crowdsourcing with microtasks. The CTS abstraction is unique in that it has *all* the following features: (1) it naturally extends the task template adopted by many crowdsourcing platforms to define microtasks, (2) it allows declarative descriptions of crowdsourcing systems, and (3) it has large expressive power.

1. INTRODUCTION

As computer network technologies evolved, crowdsourcing became popular in many application domains. Software systems that take the crowdsourcing approach are called *crowdsourcing systems* [2].

Crowdsourcing systems are often constructed on *crowdsourcing platforms*, which provide fundamental functions for implementing crowdsourcing systems. For example, the Amazon Mechanical Turk (MTurk) [3] is a crowdsourcing platform that provides a market in which *workers* perform *microtasks* (called *HITs* in MTurk) with a small payment amount per task. Crowdsourcing platforms often provide APIs for *requesters* to register microtasks in the platforms.

Because there are frequent patterns appearing in programs for crowdsourcing systems, software tools have been developed to support the implementation of complex crowdsourcing systems. For example, Turkit [4] provides a library of functions to define and call tasks from general-purpose programming languages and introduces the crash-and-rerun programming model for minimizing the cost of re-running programs. Recently, the *declarative approach* to the development of crowdsourcing systems has emerged because declarative abstractions have an affinity toward crowdsourcing applications. Declarative descriptions do not impose unnecessary timing constraints, and we can adopt many well-known optimization techniques. For example, there are proposals that use SQL-like languages to describe data-centric crowdsourcing systems [7] [8] [13]. However, they provide limited expressive power (see Section 4).

In this paper, we offer the following two key contributions:

(1) Alternative approach to declarative crowdsourcing. We first introduce the Condition-Task-Store (CTS) abstraction, which is an alternative declarative approach for implementing complex data-centric crowdsourcing with microtasks. The CTS abstraction describes a crowdsourcing system as a set of *CTS rules*, each of which is a natural extension of task template adopted by many crowdsourcing platforms to define microtasks. Therefore, programmers who are familiar with task templates can easily write simple programs with the CTS abstraction.

The CTS abstraction is unique in that it has *all* the following features: (1) it naturally extends the task template, (2) it allows declarative descriptions of crowdsourcing systems, and (3) it has large expressive power.

(2) Novel criterion for the expressive power of languages. Next, we discuss the expressive power of the CTS abstraction. We introduce a novel criterion to measure the expressive power of programming languages for crowdsourcing; the criterion focuses on the class of interactions with humans we can implement with the language. The criterion is important for the following two reasons. First, complex crowdsourcing often requires various types of interactions with humans. For example, one of such interactions of crowdsourcing is the iterative collaboration [11], which is not necessarily supported by every existing framework. Second, the class of interactions is closely related to the class of games in game theory: because human behavior is affected by the incentives and rules defined by the game structure, the class represents the size of *mechanism design space*, i.e., the set of possible mechanisms we can implement to exploit the wisdom of the crowd. In fact, the change of game structure affects the quality of the data produced by crowdsourcing systems [5]. Our examples in Sections 3 and 4 also suggest how game structure is important in crowdsourcing. Then, we theoretically show that our CTS abstraction is not only Turing complete, but can also support a wide range of game structures.

The remainder of the paper is organized as follows. Section 2 explains related work. Section 3 introduces the CTS abstraction. Section 4 discusses its expressive power. Section 5 explains a prototype to support the software development using the CTS abstraction. Section 6 is the summary.

2. RELATED WORK

Many approaches to support the development of complex crowdsourcing systems have been proposed. They differ from one another in the abstraction they use to describe crowdsourcing systems.

(1) Imperative programming languages. TurKit [4] provides a function library for implementing crowdsourcing, which can be used via codes written in imperative programming languages. It supports the crash-and-rerun model to avoid re-performing costly operations.

(2) MapReduce-like abstraction. CrowdForge [14] is a MapReduce-like framework for describing complex tasks on MTurk. It models a crowdsourcing system as a set of tasks to implement partition, map, and reduce functions. As we discuss in Section 4.3, the expressive power of the CTS abstraction is larger than that of CrowdForge.

(3) Control/data flows. CrowdLang [11] is a language for describing crowdsourcing systems in terms of basic operators, data items, and control flow constructs. Currently, it seems that CrowdLang is used as a language for writing crowdsourcing systems at a high-abstraction level and that it does not provide the means to describe the details required to directly execute the code.

(4) Rule-based abstraction. The CTS abstraction is not the first rule-based abstraction. CyLog [9] is a Datalog-like, rule-based language for describing crowdsourcing systems. A weakness of CyLog is that it requires programmers to be familiar with programming by Horn clauses even for simple crowdsourcing. In contrast, the core component of the CTS abstraction is a task template. Therefore, although the CTS abstraction borrows concepts from logic-based languages, programmers can start with a set of simple task templates and then naturally proceed to more complex crowdsourcing.

(5) SQL-like abstraction. CrowdDB [8], Qurk [7], and Deco [13] use SQL to describe crowdsourcing systems. They propose novel query processing and optimization schemes and we believe that some of the proposed techniques can be applied to the CTS abstraction. As shown in Section 4.3, the expressive power of the CTS abstraction is larger.

To our knowledge, this paper is the first to investigate the CTS abstraction. The abstraction models crowdsourcing systems as a set of CTS rules, each of which is similar to a task template. Technically, a CTS rule can be implemented by combining two ECA rules [12]: one generates microtasks and the other stores results in the database (and can be implemented by using imperative languages). The CTS abstraction provides a higher-level, user-friendly abstraction designed for describing crowdsourcing systems, which has a well-defined semantics and proven large expressive power.

Our discussion on the expressive power is related to game theory. Recently, the literature on algorithmic game theory has addressed various aspects involving both algorithms and games, such as complexities of computing equilibrium of games [15]. To our knowledge, our paper is the first to discuss classes of games that can be implemented by abstractions for crowdsourcing.

3. THE CTS ABSTRACTION

```
<QuestionForm xmlns="http://mechanicalturk.
 amazonaws.com/AWSMechanicalTurkDataSchemas/
 2005-10-01/QuestionForm.xsd">
  <Question>
    <QuestionIdentifier>1</QuestionIdentifier>
    <QuestionContent>
       <Text>How many movies have you seen this month?</Text>
    </QuestionContent>
    <AnswerSpecification>
       <FreeTextAnswer/>
    </AnswerSpecification>
  </Question>
</QuestionForm>
```

Figure 1: Example of a task template

In this section, we first explain task templates. Then, we show examples to give an intuitive explanation of the CTS abstraction. Finally, we explain formal definitions.

3.1 Task Templates

The task template is a popular form for defining and registering microtasks into crowdsourcing platforms. Figure 1 shows an example of a task template in XML format for microtasks (HITs) of MTurk, which asks a worker to enter how many movies he or she watched in a month. The essential components of a task template are the question to be shown (`QuestionContent`) and the type of the values to be received by workers (`AnswerSpecification`). Task templates can contain *variables* (called placeholders) with which we can define many microtasks that are based on the same template but differ in the values bound to the variables.

Requesters first write task templates to define and insert microtasks into the *task pool*. Then, workers perform the tasks that exist in the task pool.

3.2 Overview of the CTS Abstraction

In the CTS abstraction, a crowdsourcing system is described by a set of *CTS rules*. We assume that there exists a relational database. CTS rules read and write data to and from the database.

A CTS rule is the fundamental building block of the CTS abstraction. We first give a simple example and next show another example that requires more than one CTS rule.

Example 1. A Simple Crowdsourcing System

We use the task shown in Figure 2 to ask workers to tag books. The details are as follows:

- The database has the `Book(bid, title, author)` relation to store book information.
- For each book stored in the `Book` relation, tags are given by three workers.
- The result is stored in the `Tag(bid, tag)` relation.
- Workers are paid 10 (cents or any currency) per task, if any other workers entered the same tag.

The crowdsourcing system can be described only by the CTS rule in Figure 3. A CTS rule consists of three parts: condition, task, and store. We explain each part below.

The condition part: We write the condition to generate and insert a task into the task pool. The condition specifies what tuples need to exist in the database for generating a task. For example, the condition in Figure 3 states that a task is generated for each tuple existing in the `Book` relation.

The task part: We write a task template that contains a question to be posed to workers. The question can contain

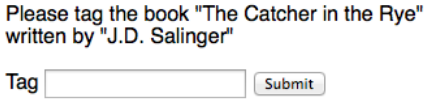

Figure 2: Example of a microtask

Condition	Book(bid, title, author)	
Task	Question	Please tag the book "$title" written by "$author"
	Generator	Entry(desc:"Tag", var:tag, type:text)
	Count	3
	Payoff	PayIf(count(Tag(bid, tag))>=2, 10)
Store	Tag(bid, tag)	

Figure 3: Example of a CTS rule

variables (such as `$author` and `$title`) bound to values in the condition (i.e., the variables are replaced with values each time the task is generated). The task part also specifies additional information, including the variables and its associated types, to store entered values.

Because there are frequently occurring patterns that appear in task template specifications, we provide *template generators* that allow users to describe task templates without specifying implementation details such as HTML tags. The task part in Figure 3 states that we use the `Entry` template generator with the following parameters: (1) the input field is labeled as "Tag," (2) the entered value is to be stored in the `tag` variable, and (3) the type of `tag` is `text`. Count is the number of tasks to be generated for the same value. Payoff describes how much is paid to workers per task. For example, `PayIf(count(Tag(bid, tag)) >= 2, 10)` states that 10 cents will be paid if there are other workers that entered the same tag to the same book.

The store part: We specify the relations and attributes in which the entered values will be stored. The store part in Figure 3 states that we use the `Tag` relation to store `bid` and the entered `tag`. □

As the example above suggests, a CTS rule is a natural extension of the widely-used task template. Because we can omit the condition and task parts, a CTS rule can be used to describe the following four types of processing.

1. Generate a task when the condition is satisfied, and store the result into the database.
2. If the condition part is omitted, generate a task with no condition, and store the result into the database.
3. If the task part is omitted, compute and store values into the database when the condition is satisfied.
4. If both of the condition and task parts are omitted, store values into the database with no condition.

Example 2. More Complex Crowdsourcing.

We consider a crowdsourcing system to rate restaurants, in which workers perform the following two types of tasks:

Task 1: Enter names of restaurants (Figure 4). A 10 cent payment is paid if the average rating by others is higher than 3.

Task 2: Enter an evaluation rating (1 to 5) for the given restaurant (Figure 5). Three workers perform this task for each restaurant, and they receive 10 cents per task.

We assume that the results of Task 1 are stored in `Restaurant(rname)`, and that those of Task 2 are stored in `Rating(rname, value)`. Then, Figure 6 shows CTS rules for Tasks 1 and 2.

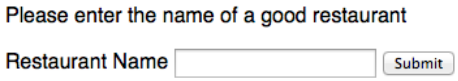

Figure 4: Example of a microtask: Task 1

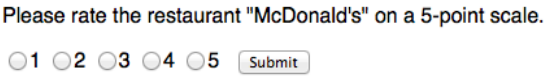

Figure 5: Example of a microtask: Task 2

Task1:

Condition		
Task	Question	Please enter the name of a good restaurant
	Generator	Entry(desc:"Restaurant Name", var: rname, type: text)
	Count	
	Payoff	PayIf(avg(Rating(rname, value))>3, 10)
Store	Restaurant(rname)	

Task2:

Condition	Restaurant(rname)	
Task	Question	Please rate the restaurant ''$rname'' on a 5-point scale.
	Generator	Choice(var: value, type: int, items: [1, 2, 3, 4, 5])
	Count	3
	Payoff	Pay(10)
Store	Rating(rname, value)	

Figure 6: CTS rules for Example 2.

The CTS rule for Task 1 has no condition: thus, the task is unconditionally generated. Unless the count number is specified, the number of generated tasks is determined as follows: (1) if the key of the predicate (`rname` of `Restaurant(rname)`) is bound to values by the condition, the task is generated only once for each case in which the condition is satisfied; (2) otherwise the same task is repeatedly generated everytime the task is performed and removed from the task pool.

The condition of the CTS rule for Task 2 states that it generates a task for each tuple stored in `Restaurant` relation. Thus, the task is generated for each restaurant entered in Task 1. Workers receive 10 cents for performing a task. □

Discussion. As the examples above suggest, the description is declarative, and each rule is invoked when its condition is satisfied. Compared to the code written in imperative programming languages, CTS rules naturally describe the parallel and asynchronous processing of computation involving human workers. On the other hand, the CTS abstraction is more expressive than the declarative query languages that do not support the transitive closure, because it essentially supports a loop with a dynamic condition check [1].

An important point is that the incentive structure plays a critical role to appropriately exploit the wisdom of crowd and (at least theoretically) ensure data quality. Because the incentive structure and rules involving multiple humans can be modeled as *games*, we can use game theory to discuss their behaviors. For example, with a simple game-theoretic analysis, the incentive structure of Example 1 guarantees that rational workers enter tags that others can easily come up with. Similarly, the incentive structure of Example 2 guarantees that rational workers for Task 1 enter the names of restaurants that are likely to receive high ratings.

3.3 Formal Definition

This section first defines the CTS rules and then explains the syntactic sugar for the concise description of rules.

3.3.1 Definition of CTS Rules

Definition 1. A program of the CTS abstraction is a set of CTS rules, each of which is modeled as a triple $R_i = (C_i, T_i, S_i)$, running over a relational database schema. Here, T_i is a task template, C_i is the condition to generate a task using T_i, and S_i is the description of how to store the result in the database. The database contains a pre-defined relation with the schema `Worker(pid, payoff)` to store information on workers and the accumulated values of the payoffs they received so far.

- C_i is a sequence $P_1(x_{11}, \ldots, x_{1n_1}), \ldots, P_m(x_{m1}, \ldots, x_{mn_m})$ of zero or more atoms. Some of the atoms can be arithmetic atoms, such as $x_{11} = 3$.
- T_i is either a triple $(q, \bar{x}, \bar{y})$ or *null*. Here, q is a question for workers, $\bar{x}$ is a sequence of variables bound by C_i, and $\bar{y}$ is a sequence of variables to store the results of performing the task.
- S_i is a sequence $Q_1(y_{11}, \ldots, y_{1n_1}), \ldots, Q_m(y_{m1}, \ldots, y_{mn_m})$ of one or more atoms, where each y_{jk} is any of a variable bound by C_i, a variable that appears in $\bar{y}$, or a constant. Each atom can be followed by either `/update` or `/delete`. □

3.3.2 Syntactic Sugar

We introduce the following variety of syntactic sugars to make the rule description concise.

(1) Attributes of atoms. The notation of atoms, which appear in the condition and store part of CTS rules, are essentially the same as those of Prolog and Datalog. A key difference is that they explicitly specify *attribute names* for their parameters. Each parameter is specified in the form *attribute:variable* or *attribute:constant*. For example, `Restaurant(name:x, zip:305)` is an atom.

There are cases in which parameters and attributes can be omitted in each atom, as described below.

- We can omit a parameter if the rule does not consume the value of the bounded variable. For example, `Restaurant(name:x)` is an atom that omits `zip`.
- We can omit an attribute name if the attribute name is the same as the variable name. For example, `Restaurant(name:name, zip:y)` can be represented as `Restaurant(name, zip:y)` because the attribute and the variable have the same name.

(2) Task part. Because we have frequently occuring patterns in the description, we introduce task-template generators and the following three fields for the task part.

- *Generator* is the name of a task-template generator, associated with its parameters. For example, `Entry` and `Choice` in Figure 6 are task-template generators. They generate actual task templates based on the given parameters and the sentence written in the Question field. These allow users to define task templates without specifying implementation details (such as HTML tags). The Crapid system (Section 5) implements various task-template generators.
- *Count* specifies the number of generated tasks. Let N be the number specified in the field. For every case in which the condition holds, the same task is generated N times and N tuples are inserted into the relation. This is implemented by adding the `count` attribute to the schema of the relation in the store part, and copying the CTS rule N times in the program.
- *Payoff* specifies a function to compute payoff values given to workers. If a function is specified, rules to update `Worker(pid, payoff)` are automatically generated and added to the set of rules. The Crapid system provides pre-defined payoff functions.

3.4 Evaluation Model and Semantics

Given a description d of the CTS abstraction, the evaluation of d on instance *ins* of the database is performed by evaluating each CTS rule in a bottom-up way on *ins*. More precisely,

- For each CTS rule $(C_i, T_i, S_i) \in d$, check if C_i is satisfied with *ins* in the following way: for each atom a_k in C_i (from left to right), check if there exists any tuple to bind variables in a_k to the values that are consistent with the values bound to the variables appeared in $a_0 \ldots a_{k-1}$.
- For every combination V of values that satisfies all the atoms in C_i, perform one of the following:
  - If $T_i \neq null$, replace variables in T_i with values in V and insert the task into the task pool.
  - If $T_i = null$, create new tuples using values in V for the atoms that appear in S_i. Then, perform one of the following: insert the tuples into *ins*, update *ins* with the tuples (when the atom is followed by `/update`), or delete the tuples from *ins* (when followed by `/delete`).
- If a worker completes the generated task, (1) create new tuples for the atoms that appear in S_i, using values in V and the entered values for the task, then (2) apply the insertion, delete, or update operation to *ins* with the new tuples.
- If *ins* is updated, check if there exist rules for which C_i is satisfied with the new *ins*. If such rules exist, process them. Terminate the process if we cannot find such a rule. If there are multiple rules that can be executed at the same time, the rule that appears earlier in the code is evaluated first.

For example, assume that we have the CTS rule shown in Figure 7. We first check if there exists a tuple that matches `Image(i, size, type:"photo")` in the database. Assume that the `Image` relation has tuple t = (`img98`, `100`, `photo`). Then, `i` and `size` are bound to `img98` and `100`, respectively. Next, check if there exists a tuple that matches `Large(size)` with `size=100`. If exists, we replace `$i` in the task template with `img98` and insert the task into the task pool to ask workers to choose the category for the image `img98`. If the task is performed, the result of performing the task is stored into `LargePhoto`, and the payoff attribute of the `Worker` relation is incremented by 1.

Given a set d of CTS rules, the semantics of d is defined as a set of *rational consequences* of d, in a similar way as the semantics of CyLog codes [9]. A rational consequence of d is a set of facts that are derived from the rules and the equilibrium [16] of the games, i.e., the states reached by rational workers.

Condition	`Image(i, size, type:"photo"), Large(size)`	
Task	Question	`Please choose a category of $i.`
	Generator	`Choice(var: category, type:string, items:[landscape, portrait, animal, food, others])`
	Count	`1`
	Payoff	`pay(1)`
Store	`LargePhoto(i, category)`	

Figure 7: Two atoms in the condition part

Rule 1

Condition	
Task	
Store	`TuringMachine(id:1, st:s, head:0)`

Rule 2

Condition	`TuringMachine(id, st, head), Tape(pos:head, sym), Rule(st, sym, new_st, new_sym, dir), new_pos = pos + dir`
Task	
Store	`TuringMachine(id, st:new_st, head:new_pos)/update, Tape(pos, sym:new_sym)/update`

Rule 3

Condition	`TuringMachine(id, head)`
Task	
Store	`Tape(pos:head)/update`

Figure 8: CTS rules implementing a Turing machine

4. EXPRESSIVE POWER

This section discusses the expressive power of the CTS abstraction. First, we show that the CTS abstraction is Turing complete. Then, we introduce a criterion to measure the expressive power of programming languages, which focuses on the class of games the language can implement. Finally, we compare the expressive power of the CTS abstraction with those of other frameworks.

4.1 Turing Completeness

Theorem 1. The CTS abstraction is Turing complete.

Proof Outline. Figure 8 is a set of CTS rules that implements any Turing machine. 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, $H \in K$ is the set of halting states, and δ is the transition function [6]. Intuitively, we need the following three components to implement a turing machine.

Element 1. Memory of the machine's inner state

Element 2. Head reading and writing information stored in the tape

Element 3. The long tape in infinitum.

In Figure 8, `TuringMachine(id, st, head)` implements Elements 1 and 2. Here, `st` records the current state whose domain is K, and `head` stores the position of the head. `Tape(pos, sym)` implements Element 3, in which each tuple (p, s) states that symbol s (whose domain is Σ) is written at position p of the tape. `Rule(st, sym, new_st, new_sym, dir)` stores the rules δ to read and write symbols on the tape and move the head.

Rule 1 initializes a Turing machine (the initial state is s). Rule 2 states how the head moves and writes symbols onto the tape according to the rules stored in `Rule(st, sym, new_st, new_sym, dir)`. It states that if the inner state is `st` and the symbol at the current position of the head is `sym`, write `new_sym` at the position, update the inner state by `new_st`, and move the head to `pos+dir`. Rule 3 extends the tape when the head reaches a position that the head never visited before. We need Rule 3 because Rule 2 always requires that `Tape(pos, sym)` exists. We also need a rule to stop the machine if it reaches the halting states $H \subseteq K$. The rule is obvious and omitted due to the space limitation.

Defining the CTS rules to implement a Turing machine proves that the CTS abstraction is Turing complete. □

4.2 Expressive Power in Terms of Games

This section proposes to use the game concept as a measure of the expressive power of programming languages for crowdsourcing, because the class of games that the language can implement affects the way in which the implemented system can exploit the power of the wisdom of crowd. First, we enumerate several classes of games and show the relationship among the classes.

Definition 2. $\mathcal{G}_1$ is a class of games that satisfy the following conditions:

1. Every input from a human is not affected by the inputs from others, and
2. The payoffs are computed by a primitive recursive function of the input values. □

An example of a game in $\mathcal{G}_1$ is one that asks humans to enter tags for a given image without telling them what tags are entered by other humans. Payoffs are defined for each combination of worker inputs. For example, workers receive payoffs when they enter the same tag. Games in $\mathcal{G}_1$ are called *simultaneous games* in game theory.

Definition 3. $\mathcal{G}_N$ is a class of programs in which (1) $N(>0)$ is known in advance; (2) each game has at most N-phases of interactions, each of which asks a worker 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 primitive recursive function of the entered values. □

Each game in $\mathcal{G}_N$ has at most N phases, each of which asks a human to enter data considering some information computed from data in the previous phases. For example, assume that we want to divide a set of cakes into two groups whose total prices are equivallent to each other. Then, a program that (1) asks one worker to divide the cakes into two groups at the first phase, then (2) asks another worker to choose one group, and finally (3) gives to each worker the total price of cakes in his group, belongs to $\mathcal{G}_N$ (with N=2). Note that the game guarantees that the prices of the two groups become the same if workers are rational; it exploits the power of human intelligence to compute how they can make two groups whose prices are equivalent to each other. From the definition, every game in $\mathcal{G}_1$ belongs to $\mathcal{G}_N$.

Definition 4. $\mathcal{G}_*$ is a class of programs in which each program (1) executes a sequence of interactions with workers, with the sequence being generated by a primitive recursive function, (2) shows workers at each interaction the information computed by a function whose parameters are taken from the results of past interactions, and (3) payoffs are computed by a primitive recursive function of the entered values. □

An example of a game in $\mathcal{G}_*$ is to ask workers to write a paragraph that explains a given keyword. Workers update

Table 1: Expressive power

Abstraction /Framework	Turing complete	Class of games
MTurk alone	N	$\subseteq \mathcal{G}_1$
CrowdForge	N	$\subseteq \mathcal{G}_N$
CrowdDB/Deco/Qurk	N	$\subseteq \mathcal{G}_N$
CTS Abstraction	Y	$\mathcal{G}_*$
Imperative programming languages with the MTurk API	Y	$\mathcal{G}_*$

the paragraph until the paragraph is satisfied by the majority of the crowd. When the paragraph is completed, every writer (worker) who contributed to the paragraph receives a payoff, which is computed by dividing a fixed total payment by the number of the contributors.

Theorem 2. $\mathcal{G}_N$ is a proper subset of $\mathcal{G}_*$.

Proof. For every g, $g \in \mathcal{G}_N \Rightarrow g \in \mathcal{G}_*$ and for any given i, there exists $g \in \mathcal{G}_*$ s.t. g has $i+1$ input phases and therefore $g \notin \mathcal{G}_N$. □

Theorem 3. Assume that we allow Turing machines to interact with humans at any step of its execution. Let M be the set of all such machines. M can implement any $g \in \mathcal{G}_*$.

Proof. The sequence of interactions with workers that can be generated by a primitive recursive function can be implemented by some $m \in M$. The information shown and the payoffs can be computed if m is a Turing machine. □

Note that being Turing complete is not a sufficient condition to be able to implement $\mathcal{G}_*$, because the power of interactions with humans does not matter for a language to be Turing complete. $\mathcal{G}_*$ contains indefinite-length sequential games (in game theory) that can be expressed with Turing machines that are able to interact with humans at any step of its execution.

From the definition of the CTS rules, the following holds.

Theorem 4. The CTS abstraction can implement any games in $\mathcal{G}_*$. □

4.3 Comparison of Expressive Powers

Table 1 compares the expressive powers of different abstractions and frameworks. Games implemented by manually registering HITs to MTurk are contained in $\mathcal{G}_1$, whereas code written in programming languages that uses MTurk APIs can implement games in $\mathcal{G}_*$. CrowdForge can implement a part of the games in $\mathcal{G}_N$, while its expressive power is not larger than $\mathcal{G}_N$, because it is not Turing complete. In particular, games implemented by a combination of partition, map, and reduce are contained in $\mathcal{G}_N$ with $N = 3$. Similarly, the class of games that CrowdDB, Qurk, and Deco can implement is not larger than $\mathcal{G}_N$.

5. PROTOTYPE SYSTEM

We implemented the Crapid system, a prototype system to develop crowdsourcing systems using the CTS abstraction. Crapid takes as input a set of CTS rules and outputs executable code. Crapid supports various task-template generators (i.e., Entry, Choice, Decision, and Comparisons as of May 2013) and payoff functions to help users easily define microtasks in CTS rules. Crapid provides a form-based user interface. When a user chooses a task-template generator, Crapid provides an appropriate set of selection boxes and drop-down menus to specify CTS rules. The output code is executable on Crowd4U [10], a crowdsourcing platform deployed at universities.

6. SUMMARY

This paper introduced the CTS abstraction, a declarative approach for implementing complex crowdsourcing with microtasks. The paper also introduced a novel criterion to measure the expressive power of programming languages for crowdsourcing by focusing on the class of games we can implement with the language. The class represents the size of mechanism design space, i.e., the set of possible mechanisms we can implement to exploit the wisdom of the crowd. The CTS abstraction is unique in that it has all the following features: (1) it naturally extends the task template adopted by many crowdsourcing platforms, (2) it allows declarative description, and (3) it has large expressive power.

Future work includes the development of various rewriting techniques for the CTS abstraction. For example, we plan to adapt various optimization techniques for crowdsourcing systems [7] [8] [13] into our context.

Acknowledgements. The authors are grateful to Prof. Shigeo Matsubara of Kyoto university for his helpful comments, and to the contributors of Crowd4U, whose names are partially listed at http://crowd4u.org. 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.

7. REFERENCES

[1] S. Abiteboul, R. Hull, V. Vianu: Foundations of Databases. Addison-Wesley 1995.
[2] A. Doan, R. Ramakrishnan, A. Y. Halevy. "Crowdsourcing systems on the World-Wide Web. Commun.ACM. 2011, vol. 54, no. 4, p. 86-96.
[3] Amazon Mechanical Turk, https://www.mturk.com/.
[4] G. Little, L. B. Chilton, M. Goldman, R. C. Miller. "Turkit: human computation algorithms on mechanical turk". Proc. UIST (2010), ACM, New York, 57-66.
[5] S. Jain, D. C. Parkes. "A game-theoretic analysis of the ESP game". ACM Trans. Economics and Comput. 1(1): 3 (2013)
[6] H. R. Lewis, C. H. Papadimitriou. "Elements of the theory of computation." Prentice-Hall, Englewood Cliffs, New Jersey, 1981
[7] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. "Human-powered sorts and joins". In VLDB, 2012.
[8] M. J. Franklin., D. Kossmann, T. Kraska, S. Ramesh, R. Xin. "CrowdDB: answering queries with crowdsourcing". SIGMOD Conference. 2011, p. 61-72.
[9] A. Morishima, N. Shinagawa, S. Mochizuki. "The Power of Integrated Abstraction for Data-centric Human/Machine Computations". VLDS2011, pp. 5-9.
[10] A. Morishima, N. Shinagawa, T. Mitsuishi, H. Aoki, S. Fukusumi. "CyLog/Crowd4U: A Declarative Platform for Complex Data-centric Crowdsourcing". PVLDB 5(12): 1918-1921 (2012).
[11] P. Minder, A. Bernstein. "CrowdLang: A Programming Language for the Systematic Exploration of Human Computation Systems". SocInfo 2012: 124-137
[12] N. W. Paton, O. Di'az. "Active Database Systems". ACM Comput. Surv. 31(1): 63-103 (1999)
[13] H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, J. Widom. "An overview of the deco system: data model and query language; query processing and optimization". SIGMOD Record 41(4): 22-27 (2012)
[14] A. Kittur, B. Smus, S. Khamkar, R. E. Kraut. "CrowdForge: crowdsourcing complex work". UIST 2011: 43-52
[15] T. Roughgarden. "Algorithmic game theory". Commun. ACM 53(7): 78-86 (2010)
[16] F. Vega-Redondo. Economics and Theory of Games, Cambridge University Press, 2003.