Expressing Algorithms As Concise As Possible via Computability Logic

Keehang Kwon
Faculty of Computer Engineering, DongA University
840 Hadan Saha, 604-714 Busan, Korea
khkwon@dau.ac.kr

Abstract

This paper proposes a new approach to defining and expressing algorithms: the notion of *task logical* algorithms. This notion allows the user to define an algorithm for a task T as a set of agents who can collectively perform T. This notion considerably simplifies the algorithm development process and can be seen as an integration of the sequential pseudocode and logical algorithms.

This observation requires some changes to algorithm development process. We propose a two-step approach: the first step is to define an algorithm for a task T via a set of agents that can collectively perform T. The second step is to translate these agents into (higher-order) computability logic.

Keywords : tasks, algorithm, agents, computability logic.

1 Introduction

Traditional acquaintance with algorithm languages relates to the pseudocodes, also known as imperative algorithms. Within this setting, algorithms are expressed as a sequence of instructions. Many algorithms in algorithm textbooks [11] have been written in pseudocodes. However, pseudocode is, in a sense, a low-level language because the user must specify the execution order. In particular, pseudocode is very awkward to use in expressing nondeterministic algorithms such as graph problems where execution orders are typically unknown beforehand.

Logical algorithm language is a high-level language in which the execution order can be omitted. Consequently, logical languages can express most deterministic/nondeterministic algorithms in a concise way. Traditional logical languages, however, suffer from weak expressibility because they are built around the notion of boolean logic (true/false) [3, 4]. It is possible to increase the expressibility of logical languages by employing a task/game logic called computability logic (CL) [6, 7], a powerful logic which is built around the notion of success/failure. The task logic offers many new, essential logical operators including parallel conjunction/disjunction, sequential conjunction/disjunction, choice conjunction/disjunction, *etc.*

This paper proposes to use CL as an algorithm language. The distinguishing feature of CL is that now each clause/agent is allowed to perform new, sophisticated tasks that have not been supported by previous logical languages. While CL is an excellent algorithm language, it is based on the first-order logic. We also consider its higher-order extension where first-order terms are replaced by higher-order terms. It is well-known that higher-order terms can describe objects of function types including programs and formulas. Higher-order terms have proven useful in many metalanguage applications such as theorem proving.

The remainder of this paper is structured as follows. We discuss a new way of defining algorithms in the next section. In Section 3, we present some examples. Section 4 concludes the paper.

2 Task Logical Algorithms

A *task logical* algorithm for a task T is of the form

$$c_1 : T_1, \ldots, c_n : T_n \longrightarrow d : T$$

where $c_i : T_i$ represents an agent c_i who can do task T_i. In the traditional developments of declarative algorithms, those T_is are limited to simple tasks such as computing recursive functions, relations or resources. Most complex tasks such as interactive ones are not permitted. In algorithm design, however, complex tasks are desirable quite often. Such examples include many OS processes, Web agents, *etc.*

To define the class of computable tasks, we need a specification language. An ideal language would support an optimal translation of the tasks. We argue that a reasonable, high-level translation of the tasks can be achieved via computability logic(CL)[5, 6]. An advantage of CL over other formalisms such as sequential pseudocode, linear logic[3], *etc*, is that it can optimally encode a number of essential tasks: nondeterminism, updates, *etc.* Hence the main advantage of CL over other formalisms is the minimum (linear) size of the encoding.

We consider here a higher-order version of CL. The logical language we consider in this paper is built based on a typed lambda calculus. Although types are strictly necessary, we will omit these here because their identity is not relevant in this paper. An atomic formula is $(p\ t_1 \ldots t_n)$ where p is a (predicate) variable or non-logical constant and each t_i is a lambda term.

The basic operator in CL is the reduction of the form $c : A \to B$. This expression means that the task B can be reduced to another task B. The expression $c : A \wedge B$ means that the agent c can perform two tasks A and B in parallel. The expression $!A$ means that the agent can perform the task A repeatedly. The expression $c : A \sqcap B$ means that the agent c can perform either task A or B, regardless of what the machine chooses. The expression $c : \bigsqcap x A(x)$ means that the agent c can perform the task A, regardless of what the machine chooses for x. The expression $c : A \sqcup B$ means that the agent c can choose and perform a true disjunct between A and B.

The expression $c : \bigsqcup x A(x)$ means that the agent can choose a right value for x so that it can perform the task A. We point the reader to [6, 7] to find out more about the whole calculus of CL.

3 Examples

The notion of task logical algorithms makes algorithms simpler and versatile compared to traditional approach. As an example, we present the factorial algorithm to help understand this notion. The factorial algorithm can be defined using an agent c whose tasks are described below in English:

(1) c can either claim that $fact(0,1)$ holds, or

(2) can replace $fact(X,Y)$ by $fact(X+1, XY+Y)$.

It is shown below that the above description can be translated into CL formulas. The following is a CL translation of the above algorithm, where the reusable action is preceded with !.

$c : (fact\ 0\ 1) \sqcap\ !\ \sqcap x \sqcap y\ ((fact\ x\ y)\ \rightarrow\ (fact\ x+1\ xy+y))$.

A task is typically given by a user in the form of a query relative to agents. Computation tries to solve the query with respect to the agent c. As an example, executing $agent\ c\ \longrightarrow\ \sqcap y \sqcup z fact(y,z)$ would involve the user choosing a value, say 5, for y. This results in the initial resource $fact(0,1)$ being transformed to $fact(1,1)$, then to $fact(2,2)$, and so on. It will finally produce the desired result $z = 120$ using the second conjucnt five times.

An example of interactive tasks is provided by the following agent t which has a lottery ticket. The ticket is represented as $0 \sqcup\ 1M$ which indicates that it has two possible values, nothing or one million dollars.

The following is a CL translation of the above algorithm.

$t : 0 \sqcup\ 1M$.

Now we want to execute t to obtain a final value. This interactive task is represented by the query t. Now executing the program $agent\ t\ \longrightarrow\ agent\ t$ would produce the following question asked by the agent in the task of $0 \sqcup\ 1M$ in the program: "how much is the final value?". The user's response would be zero dollars. This move brings the task down to $0\ \longrightarrow\ agent\ t$. Executing $0\ \longrightarrow\ agent\ t$ would require the machine to choose zero dollars in $0 \sqcup\ 1M$ for a success.

An example of parallel tasks is provided by the following two agents c and d working at a fastfood restaurant. The agent c waits for a customer to pay money(at least three dollars), and then generates a hamburger set consisting of a hamburger, a coke and a change. The agent d waits for a customer to pay money(at least four dollars), and then generates a fishburger set consisting of a fishburger, a coke and a change.

The following is a CL translation of the above algorithm.

$c :! \sqcap x(\geq (x,3)\ \rightarrow\ m(ham) \wedge m(coke) \wedge m(x-3))$.
$d :! \sqcap x(\geq (x,4)\ \rightarrow m(fi) \wedge m(coke) \wedge m(x-4))$.

Now we want to execute c and d in parallel to obtain a hamburger set and then a fishburger set by interactively paying money to c and d. This interactive task is represented by the query $c \wedge d$. Now executing the program $agent\ c, agent\ d \longrightarrow agent\ c \wedge agent\ d$ would produce the following question asked by the agent in the task of c: "how much do you want to pay me?". The user's response would be five dollars. This move brings the task down to $m(ham) \wedge m(coke) \wedge m(\$2)$ which would be a success. The task of d would proceed similarly.

As an example of higher-order algorithms, consider the interpreter for Horn clauses. It is described by G- and D-formulas given by the syntax rules below:

$$G ::= A \mid G\ and\ G \mid some\ x\ G$$

$$D ::= A \mid G\ imp\ A \mid all\ x\ D \mid D\ and\ D$$

In the rules above, A represents an atomic formula. A D-formula is called a Horn clause. The expression $some\ x\ G$ involves bindings. We represent such objects using lambda terms. For example, $all\ x\ p(x)$ is represented as $all\ \lambda x(p\ x)$.

In the algorithm to be considered, G-formulas will function as queries and D-formulas will constitute a program.

We will present an operational semantics for this language based on [10]. Note that execution alternates between two phases: the goal-reduction phase and the backchaining phase. Following Prolog's syntax, we assume that names beginning with uppercase letters are quantified by $\sqcap$.

Definition 1. Let G be a goal and let D be a program. Then the notion of executing $\langle D, G\rangle$ – $pv\ D\ G$ – is defined as follows:

(1) $bc\ D\ A\ A$. % This is a success.

(2) $pv\ D\ G_1 \rightarrow bc\ D\ (G_1\ imp\ A)\ A$).

(3) $bc\ D\ (D\ X)\ A \rightarrow bc\ D\ (all\ D)\ A$.

(4) $bc\ D\ D_1\ A \vee bc\ D\ D_2\ A \rightarrow bc\ D\ (D_1\ and\ D_2)\ A$.

(5) $atom\ A \wedge\ bc\ D\ D\ A \rightarrow pv\ D\ A$. % change to backchaining phase.

(6) $pv\ D\ G_1 \wedge pv\ D\ G_2 \rightarrow pv\ D\ (G_1\ and\ G_2)$.

(7) $pv\ D\ (G\ X) \rightarrow pv\ D\ (some\ G)$.

In the rules (3) and (7), the symbol X will be instantiated by a term. In this context, consider the query $pv\ (p\ a)\ (some\ \lambda x(p\ x))$. In solving this query, $pv\ (p\ a)\ (p\ a)$ will be formed and eventually solved.

The examples presented here have been of a simple nature. They are, however, sufficient for appreciating the attractiveness of the algorithm development process proposed here. We point the reader to [8, 9, 10] for more examples.

4 Conclusion

A proposal for designing algorithms is given. It is based on the view that an algorithm for a task T is a set of agents who can collectively perform the task. The advantage of our approach is that it simplifies the process of designing and writing algorithms for most problems.

Our ultimate interest is in a procedure for carrying out computations of the kind described above. Hence it is important to realize this CL interpreter in an efficient way, taking advantages of some techniques discussed in [1, 2, 4].

5 Acknowledgements

This paper was supported by Dong-A University Research Fund.

References

[1] M. Banbara. *Design and implementation of linear logic programming languages*. Ph.D. Dissertation, Kobe University, 2002.

[2] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *Proceedings of the 1996 Workshop on Extensions of Logic Programming*, LNAI 1050, pages 67 – 81.

[3] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[4] Joshus Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1994. Invited to a special issue of submission to the 1991 LICS conference.

[5] G. Japaridze. The logic of tasks. *Annals of Pure and Applied Logic*, 117:263–295, 2002.

[6] G. Japaridze. Introduction to computability logic. *Annals of Pure and Applied Logic*, 123:1–99, 2003.

[7] G. Japaridze. Sequential operators in computability logic. *Information and Computation*, vol.206, No.12, pp.1443-1475, 2008.

[8] D. Miller and G. Nadathur. 1987. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, S. Haridi, Ed. IEEE Computer Society Press, 379–388.

[9] D. Miller and G. Nadathur. 1988. λProlog version 2.7. Distributed in C-Prolog and Quintus Prolog source code.

[10] D. Miller and G. Nadathur. 2012. Programming with higher-order logic. Cambridge University Press.

[11] R. Neapolitan and K. Naimipour. *Foundations of Algorithms*. Heath, Amsterdam, 1997.