

Relative Algorithms: Agent-Oriented Declarative Algorithms

Keehang Kwon, Sungwoo Hur

Dong-A University Department of Computer Engineering
Busan, Republic of Korea
[The corresponding author]

Jungsoo Kim

Dong-A University Graduate school of Industry and Information
Busan, Republic of Korea
Email: khkwon@dau.ac.kr, swhur@dau.ac.kr, lusein@paran.com

Abstract- This paper proposes a new approach to defining algorithms: the notion of relative algorithms. This notion allows the programmer to define an algorithm (i.e., an agent) relative to a set of other algorithms. This notion considerably simplifies the algorithm development process and can be seen as an integration of the sequential pseudo code 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 computability logic enhanced with sequential operators.

Keywords- Agents, Algorithm, Computability logic, Tasks

I. INTRODUCTION

There have been two different approaches to defining algorithms, manifest in the notions of imperative algorithms and declarative algorithms[7]. The former notion corresponds to defining an algorithm as a sequence of instructions, whereas the latter notion defines an algorithm in the form of recursive functions or relations where its execution order being determined only at run time. At a pragmatic level, the first approach

has a drawback in that it cannot define nondeterministic tasks in an elegant fashion, while the second approach is not able to define complex tasks such as interactive tasks or sequential tasks. Our interest in this paper is in the situation where a regimen of declarative algorithms is adopted and, in particular, in the improvement in defining declarative algorithms.

Declarative algorithms has been based on the notion that each agent in them can compute functions, relations, or resources[3],[4]. Despite some popularity, this view of algorithms lacks devices for handling complex tasks. Lacking such devices as sequential tasks, dealing with complex tasks in this view is simply not possible and greatly reduces the applicability of the algorithm.

The disadvantages of imperative algorithms are the following:

a. Each statement can perform only limited tasks:

Imperative algorithms typically employ six statements: assignment statements, if-then-else statements, while-statements, etc. However, some important statements are missing from this set, as we shall see later. Especially reduction statements of the form $A \rightarrow B$ are missing.

b. They leads to lengthy codes:

Developers have to specify too much sequencing. For example, If developers wish

to do a matrix multiplication, they have to do n^3 multiplications. And for writing an ordinary program to do this, they have to specify the exact sequence which they are all to be done. Actually, it doesn't matter in what order processor do the multiplications so long as developers add them together in the right groups. Thus the ordinary sort of imperative language imposes much too much sequencing, which makes it very difficult to rearrange if you want to make things more efficient.

c. From the perspective of software engineering, software developments take longer than necessary and software testing is difficult and this makes them less productive.

The disadvantages of declarative algorithms are the following:

a. Sequential tasks are not permitted:

In declarative algorithms, programmers do not specify sequences of operations, but only definitions or equations specifying relation. So declarative algorithms have relatively poor performance on today's personal computer architecture, which is optimized for sequential processing.

b. Only limited applications are possible :

Only limited applications are possible. Many applications include games and OS. Unfortunately all these applications cannot be expressed in declarative algorithms.

c. Sharing is not permitted:

All the knowledge bases are simply a sequence of knowledges and more complex structures than a single sequence cannot be expressed in declarative algorithms. Hence, they degrade the efficiency. However, sharing can solve this efficiency problem.

This paper proposes a generalization of declarative algorithms: agent-oriented algorithms. The main concept is that an agent is allowed to perform complex tasks, not necessarily be a function or relation.

This paper also considers its effects on the algorithm development process. To be precise, we consider here algorithm design to be the process of finding a set of agents that can collectively perform the goal task. An attractive feature of this view is that it enhances the readability and modifiability of the algorithm for most problems. 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.

II. RELATIVE ALGORITHMS

Our interest is in a process for developing algorithms based on the observation described in the previous section. The traditional, declarative algorithm process models provide a useful structure for such a process, but some changes are needed. A declarative algorithm for a task T is of the form

$$C_1: T_1, \dots, C_n: T_n \rightarrow d: T$$

where $C_i: T_i$ represents an agent C_i who can do task T_i . In the developments of declarative algorithms, those T_i s are limited to simple tasks such as computing functions, relations or resources. Most complex tasks – sequential tasks or interactive ones – are not permitted. In algorithm design, however, complex tasks are desirable quite often. Such examples include the famous dining philosopher's problem, many OS processes, Web agents, etc. This is shown in Fig 1.

To define the class of computable tasks, we need a specification language. In choosing a language, there is an aspect that requires a special attention. An acceptable language should not expand the resulting description too much, rather support a

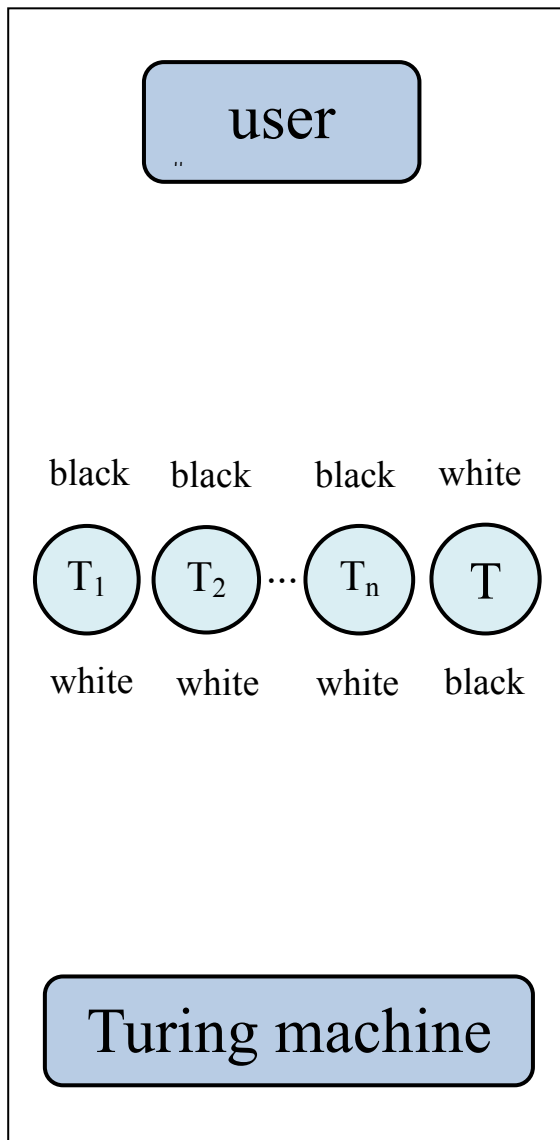


Fig. 1 Programming as a game

reasonable translation of the tasks. 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 attractive feature of CL over other formalisms such as Turing machines, sequential pseudo code, linear logic[3], etc, is that it can optimally encode a number of essential characteristics of tasks: nondeterminism, updates, etc. Hence the main advantage of CL over other formalisms is the minimum (linear) size of the encoding. The basic operator in CL is the reduction operator of the form $A \rightarrow B$. This expression means that the task B can be reduced to another task A. The expression $A \wedge B$

means an agent who can perform two tasks A and B in parallel.

The expression $!A$ means an agent who can perform the task A repeatedly. The expression $\Pi xA(x)$ means an agent who can perform the task A, regardless of what the environment chooses for x. The expression $\exists xA(x)$ means an agent who can choose a right value for x so that it can perform the task A. We point the reader to [6] to find out more about the whole calculus of CL.

Sequential tasks requires sequential operators. Due to the lack of them, applications of the declarative algorithms - including CL - have been limited, mostly to the database area and artificial intelligence. For the reason, we add two new sequential operators \cap and \cup . The expression $A \cap B$ means an agent who can perform both tasks A and B in sequence. Similarly, the expression $A \cup B$ means an agent who can perform at least one of two tasks A and B if it tries both tasks in sequence.

The advantages of relative algorithms are the following:

- a. Each agent can perform versatile tasks.
- b. Each algorithm in this approach becomes shorter. This requires far less code than the above two algorithms. That means fewer points of failure, less code to test, and a more productivity.
- c. Consequently, software development periods become shorter. And it will likely be paid off when developer modify source code in the future.
- d. Softwares are easier to read and test. It makes source code clear In all cases this can make code more readable. It makes designing and writing procedure simple.

In imperative algorithms, programmers using them must deal with the management of variables and assignment of values to them. The results of this are the difficulty in construction of programs. In relative algorithms, the programmer does not need to

be concerned with variables. Furthermore, they are easy to prove correct. These observations are summarized in Table I.

TABLE I

COMPARATIVE ANALYSIS OF IMPERATIVE, DECLARATIVE AND RELATIVE ALGORITHMS.

sub table 1

	Tasks an agent can perform
Imperative	Limited (no reduction tasks)
Declarative	Limited (no interactive tasks)
Relative	Unlimited

sub table 2

	Combined goal tasks
Imperative	Not allowed
Declarative	Allowed
Relative	Allowed

sub table 3

	Typical algorithm length
Imperative	Lengthy
Declarative	Concise
Relative	Concise

sub table 4

	Complex goal tasks
Imperative	Yes
Declarative	No (no sequential goals are allowed)
Relative	Yes

As discussed earlier, reduction tasks express conditional tasks. It means that an agent accomplish the task given a help from outside.

Suppose we have two instructions A and B. Combined tasks refer to a task that can be done using the combinations of A and B. This feature is useful for making programs short.

III. EXAMPLES

The notion of relative 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 two agents c_1 , c_2 whose tasks are described below in English:

- (1) The agent c_1 who can claim $\text{fact}(0,1)$ is true.
- (2) The agent c_2 who can compute $\text{fact}(X+1, XY+Y)$ using $\text{fact}(X,Y)$.

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.

$c_1 : \text{fact}(0,1)$.

$c_2 : ! \Pi x \Pi y (\text{fact}(x, y) \rightarrow \text{fact}(x+1,xy+y))$.

A task is typically given by a user in the form of a query relative to agents. Computation tries to solve the query. As an example, executing $c_1, c_2 \rightarrow \Pi z \text{fact}(5,z)$ would result in the initial resource $\text{fact}(0,1)$ being transformed to $\text{fact}(1,1)$, then to $\text{fact}(2,2)$, and so on. It will finally produce the desired result $z=120$ using the second agent five times. As another example, executing $c_1, c_2 \rightarrow \Pi z \Pi w (\text{fact}(3,z) \wedge \text{fact}(z,w))$ would result in processing $\text{fact}(3,z)$ in $\text{fact}(z,w)$ in parallel, producing $z=6$ and $w=720$.

An example of sequential tasks is provided by the following two agents c_1 and c_2 working at a fast-food restaurant. The agent c_1 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 c_2 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.

$$c_1: !\Pi x(\geq(x,3) \rightarrow m(h) \wedge m(c) \wedge m(x-3)).$$

$$c_2: !\Pi x(\geq(x,4) \rightarrow m(fi) \wedge m(c) \wedge m(x-4)).$$

Now we want to execute c_1 and c_2 to sequentially obtain a hamburger set and then a fishburger set by interactively paying money to c_1 and c_2 . This interactive (and sequential) task is represented by the query $c_1 \cap c_2$.

Now executing the program $c_1, c_2 \rightarrow c_1 \cap c_2$ would produce the following question asked by the agent in the task of c_1 : "how much do you want to pay me?". The user's response would be five dollars. This move brings the task down to $m(h) \wedge m(c) \wedge m(\$2)$ which would be a success. Then the task of c_2 would proceed similarly.

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 [6] for more examples.

IV. CONCLUSION

Programming languages are designed to tell a computer what to do. Different communities have developed different paradigms for expressing algorithms. The major programming paradigms are the following: imperative, object-oriented, functional and logical. All these paradigms require a computer to act as a universal Turing machine.

Our approach requires a computer to adopt a smart strategy compared to other paradigms. It means more work to a

computer and lesser work to a human. This means that the interpreter is hard to implement and if sharing is allowed, we have a long way to go.

A proposal for designing algorithms is given. It is based on the view that a piece of software is an agent simulated on a machine and an algorithm is a constructive definition of agents. 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 our 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].

ACKNOWLEDGEMENTS

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

REFERENCES

- [1] M.Banbara. "Design and implementation of linear logic programming languages". Ph.D. Dissertation, Kobe University, 2002.
- [2] Iliano Cervesto, 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, 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] R.Neapolitan and K. Naimipour. "Foundations of Algorithms". Heath, Amsterdam, 1997.