EPLAS: An Epistemic Programming Language for All Scientists

Isao Takahashi, Shinsuke Nara, Yuichi Goto, and Jingde Cheng

Department of Information and Computer Sciences, Saitama University,
255 Shimo-Okubo, Sakura-ku, Saitama-shi, Saitama, 338-8570, Japan
{isao, nara, gotoh, cheng}@aise.ics.saitama-u.ac.jp

Abstract. Epistemic Programming has been proposed as a new programming paradigm for scientists to program their epistemic processes in scientific discovery. As the first step to construct an epistemic programming environment, this paper proposes the first epistemic programming language, named 'EPLAS'. The paper analyzes the requirements of an epistemic programming language, presents the ideas to design EPLAS, shows the important features of EPLAS, and presents an interpreter implementation of EPLAS.

Keywords: Computer-aided scientific discovery, Epistemic process, Strong relevant logic, Scientific methodology.

1 Introduction

As a new programming paradigm, Cheng has proposed Epistemic Programming for scientists to program their epistemic processes in scientific discovery [3,4]. Conventional programming regards numeric values and/or character strings as the subject of computing, takes assignments as basic operations of computing, and regards algorithm as the subject of programming, but Epistemic Programming regards ***beliefs*** as the subject of computing, takes ***primary epistemic operations*** as basic operations of computing, and regards ***epistemic processes*** as the subject of programming [3,4].

Under the strong relevant logic model of epistemic processes proposed by Cheng, a belief is represented by a formula $A \in \mathrm{F}(\mathbf{EcQ})$ where **EcQ** is a predicate strong relevant logic [3,4] and F(**EcQ**) is the set of all well-formed formulas of **EcQ**. The three primary epistemic operations are ***epistemic deduction***, ***epistemic expansion***, and ***epistemic contraction***. Let $K \subseteq \mathrm{F}(\mathbf{EcQ})$ be a set of sentences to represent the explicitly known knowledge and current beliefs of an agent, and $\mathrm{T}_{\mathbf{EcQ}}(P)$ be a formal theory with premises P based on **EcQ**. For any $A \in \mathrm{T}_{\mathbf{EcQ}}(K) - K$ where $\mathrm{T}_{\mathbf{EcQ}}(K) \neq K$, an epistemic deduction of A from K, denoted by $K^{\mathrm{d}+A}$, by the agent is defined as $K^{\mathrm{d}+A} =_{\mathrm{df}} K \cup \{A\}$. For any $A \notin \mathrm{T}_{\mathbf{EcQ}}(K)$, an epistemic expansion of K by A, denoted by $K^{\mathrm{e}+A}$, by the agent is defined as $K^{\mathrm{e}+A} =_{\mathrm{df}} K \cup \{A\}$. For any $A \in K$, an epistemic contraction of K by A, denoted by K^{-A}, by the agent is defined as $K^{-A} =_{\mathrm{df}} K - \{A\}$. An epistemic process of an agent is a sequence K_0, o_1, K_1, o_2, K_2, ..., $K_{\mathrm{n}-1}$,

Y. Shi et al. (Eds.): ICCS 2007, Part I, LNCS 4487, pp. 406–413, 2007.

o_{n}, K_{n} where K_i ($\mathrm{n} \geq i \geq 0$) is an ***epistemic state***, and $o_{i+1}(\mathrm{n} > i \geq 0)$ is any of primary epistemic operations, and K_{i+1} is the result of applying o_{i+1} to K_i. In particular, K_0 is called the ***primary epistemic state*** of the epistemic process, K_{n} is called the ***terminal epistemic state*** of the epistemic process, respectively.

An ***epistemic program*** is a sequence of instructions such that for a primary epistemic state given as the initial input, an execution of the instructions produces an epistemic process where every primary epistemic operation corresponds to an instruction whose execution results in an epistemic state, in particular, the terminal epistemic state is also called the result of the execution of the program [3,4].

However, until now, there is no environment to perform Epistemic Programming and to run epistemic programs. We propose the first epistemic programming language, named 'EPLAS': an **E**pistemic **P**rogramming **L**anguage for **A**ll **S**cientists. In this paper, we analyze the requirements of an epistemic programming language at first, and then, present our design ideas for EPLAS and its important features. We also present an interpreter implementation of EPLAS.

2 Requirements

We define the requirements for an epistemic programming language and its implementation. We define R1 in order to write and execute epistemic programs.

R 1. *They should provide ways to represent beliefs and epistemic states as primary data, and epistemic operations as primary operations, and perform the operations.* Since, in Epistemic Programming, the subject of computing is a belief, basic operations of computing are epistemic operations, and the subject of the operations are epistemic states.

We also define R2, R3, and R4 in order to write and execute epistemic programs to help scientists with scientific discovery.

R 2. *They should represent and execute operations to perform deductive, inductive, and abductive reasoning, as primary operations.* Scientific reasoning is indispensable to any scientific discovery because any discovery must be previously unknown or unrecognized before the completion of discovery process and reasoning is the only way to draw new conclusions from some premises that are known facts or assumed hypothesis [3,4]. Therefore, reasoning is one of ways to get new belief when scientists perform epistemic expansion.

R 3. *They should represent and execute operations to help with dissolving contradictions as primary operations.* Scientists perform epistemic contraction in order to dissolve contradictions because beliefs may be inconsistent and incomplete.

R 4. *It should represent and execute operations to help with trial-and-error as primary operations.* Scientists do not always accurately know subjects of scientific discovery beforehand. Therefore, scientists must perform trial-and-error.

In a process of trial-and-error, scientists make many assumptions and test the assumptions by many different methods. It is a demanding work for scientists to make combinations of the assumptions and the methods. Furthermore, it is also demanding for scientists to test the combinations one at a time without omission.

3 EPLAS

EPLAS is designed as a typical procedural and strongly dynamic typed language. It has facilities to program control structures (they are *if-then statement*, *do-while statement*, and *foreach statement*) and procedures, and has a nested static-scope rule. With attribute grammar, we defined syntax and semantics of EPLAS and open EPLAS manual [1].

To satisfy R1, EPLAS should provide beliefs as a primary data type. For that purpose, EPLAS provides a way to represent beliefs as a primary data type. EPLAS also provides an operation to input a belief from standard input, which is denoted by '**input_belief**'. Conventional programming languages do not provide beliefs as a primary data type because their subjects of computing are lower-level data types. Then, to satisfy R1, EPLAS should provide epistemic states as a primary data type. Therefore, EPLAS provides sets of beliefs as set structured data type and assures that all epistemic states are numbered. EPLAS also provides an operation to get the i–th epistemic state, which is denoted by '**get_state**(i)'. Almost conventional programming languages do not provide a set structured data type and do not assure that all epistemic states are numbered. Furthermore, EPLAS should provide epistemic operations as primary operations to satisfy R1. Hence, EPLAS provides operations to perform epistemic deduction, epistemic expansion, and epistemic contraction by multiple beliefs as extensions as primary operations. An epistemic deduction is denoted by '**deduce**', and makes the current epistemic state K_i the next epistemic state $K_{i+1} = K_i \cup S$ for any $S \subseteq \mathrm{T}_{\mathbf{EcQ}}(K_i) - K_i$ where $K_i \subseteq \mathrm{F}(\mathbf{EcQ})$ and $\mathrm{T}_{\mathbf{EcQ}} \neq K_i$. An epistemic expansion of multiple beliefs S is denoted by '**expand**(S)', and makes the current epistemic state K_i the next epistemic state $K_{i+1} = K_i \cup S$ for any $S \nsubseteq \mathrm{T}_{\mathbf{EcQ}}(K_i)$ where $K_i \subseteq \mathrm{F}(\mathbf{EcQ})$. An epistemic contraction by multiple beliefs S is denoted by '**contract**(S)', and makes the current epistemic state K_i the next epistemic state $K_{i+1} = K_i - S$ for any $S \subset K_i$ where $K_i \subseteq \mathrm{F}(\mathbf{EcQ})$. Some conventional programming languages provide operations to perform epistemic expansion and epistemic contraction as set operations but any conventional programming languages do not provide an operation to perform epistemic deduction.

There are three forms of reasoning: deductive, inductive, and abductive reasoning. Therefore, an epistemic programming language should provide operations to perform reasoning by these three forms.

In order to satisfy R2, EPLAS should provide a way to represent various reasoning, at least, reasoning by the three forms. For that purpose, EPLAS provides inference rules as a primary data type. Inference rules are formulated with some schemata of well-formed formulas to reason by pattern matching,

and consist of at least one schemata of well-formed formula as premises and at least one schemata of well-formed formulas as conclusions [7]. Let K be premises including '$\{\mathrm{P0}(a0), \mathrm{P0}(a1), \mathrm{P1}(a0), \forall x0(\mathrm{P2}(x0) \rightarrow \mathrm{P1}(x0))\}$', ir_1 be an inference rule: '$\mathrm{P0}(x0)$, $\mathrm{P0}(x1) \vdash \forall x2\mathrm{P0}(x2)$', ir_2 be an inference rule: '$\mathrm{P0}(x0)$, $\mathrm{P1}(x0)$, $\mathrm{P0}(x1) \vdash \mathrm{P1}(x1)$', and ir_3 be an inference rule: '$\mathrm{P2}(x0) \rightarrow \mathrm{P1}(x1)$, $\mathrm{P1}(x1) \vdash \mathrm{P2}(x0)$'. ir_1 means an inductive generalization [5], ir_2 means an arguments from analogy [5], and ir_3 means an abductive reasoning [8]. '$\forall x2\mathrm{P0}(x2)$' is derived from K by ir_1, '$\mathrm{P1}(a1)$' is derived from K by ir_2, and '$\mathrm{P2}(a0)$' is derived from K by ir_3. EPLAS also provides an operation to input an inference rule from standard input, which is denoted by '**input_rule**'. Moreover, in order to satisfy R2, EPLAS should provide an operation to derive conclusions from beliefs in the current epistemic state by applying an inference rule to the beliefs and to perform epistemic expansion by the derived conclusions. A reasoning by an inference rule ir is denoted by '**reason**(ir)', and makes the current epistemic state K_i the next epistemic state $K_{i+1} = K_i \cup S$ where $K_i \subseteq \mathrm{F}(\mathbf{EcQ})$, $S \subset \mathrm{R}_{ir}(K_i)$, $S \neq \phi$, and $\mathrm{R}_{ir}(K_i)$ is a set of beliefs derived from K_i by an inference rule ir. Any conventional programming languages do not provide inference rules as a primary data type and a reasoning operation as a primary operation.

As operations to help with dissolving contradictions, at least, an epistemic programming language should provide an operation to judge whether two beliefs are conflicting or not. Then, it should provide operations to output a derivation tree of a belief and to get all beliefs in a derivation tree of a belief in order for scientists to investigate causes of contradictions. It should also provide an operation to perform epistemic contraction of beliefs derived from a belief in order for scientists to reject beliefs derived from a conflicting belief.

In order to satisfy R3, EPLAS should provide an operation to judge whether two beliefs are conflicting or not. For that purpose, EPLAS provides the operation as a primary operation, which is denoted by '`$$`'. The binary operator '`$$`' is to judge whether two beliefs are conflicting or not, and returns true if and only if one belief A is negation of the other belief B, or false if not so. Then, in order to satisfy R3, EPLAS should provide operations to output a derivation tree of a belief. Hence, EPLAS provides an operation to output a derivation tree of a belief A from standard output, which is denoted by '**see_tree**(A)'. EPLAS should also provide an operation to get all beliefs in a derivation tree of a belief. Therefore, EPLAS provides an operation to get beliefs in the derivation tree of a belief A, which is denoted by '**get_ancestors**(A)'. Furthermore, in order to satisfy R3, EPLAS should provide an operation to perform epistemic contractions of all beliefs derived by the specific beliefs. For that purpose, EPLAS provide the operation to perform an epistemic contractions of beliefs derived by the specific beliefs S, which is denoted by '**contract_derivation**(S)', and which makes the current epistemic state K_i the next epistemic state $K_{i+1} = K_i - (\mathrm{T}_{\mathbf{EcQ}}(K_i) - \mathrm{T}_{\mathbf{EcQ}}(K_i - S))$ where $K_i \subseteq \mathrm{F}(\mathbf{EcQ})$. Evidently, conventional programming languages do not provide these operations.

An epistemic programming language should provide operations to make combinations of beliefs and to test each combination in order to verify combinations

of assumptions, at least, as operations to help with trial-and-error. It also should provide operations to make permutations of procedures and to test each permutation in order for scientists to test many methods by various turns. Furthermore, it should provide an operation to make the current epistemic state change into a past epistemic state in order for scientists to test assumptions by various methods in same epistemic state.

To satisfy R4, EPLAS should provide operations to make combinations of beliefs and to test each combination. For that purpose, EPLAS provides sets of sets as a set-set structured type and set operations, e.g. , sum, difference, intersection, power, and direct product. Some conventional programming languages provide the structured data type and the set operations but almost conventional programming languages do not. Then, to satisfy R4, EPLAS should provide operations to make permutations of procedures and to test each permutation. Hence, EPLAS provide procedures as a primary data type and sequences as a seq structured type, and sequence operations, e.g. , appending to the bottom, dropping from the bottom. A procedure is a name of a *procedure with arguments*, and is similar to a function pointer in C languages. In order to satisfy R4, furthermore, EPLAS should provide an operation to change the current epistemic state into a past one identified by a number. EPLAS provides the operation, which is denoted by '**return_to**(n)', and makes the current epistemic state K_i the next epistemic state $K_{i+1} = K_n$ where K_n is n–th epistemic state.

4 An Interpreter Implementation of EPLAS

We show an interpreter implementation of EPLAS. We implemented the interpreter with Java (J2SE 6.0) in order for the interpreter to be available on various computer environments. We, however, implemented the interpreter by naive methods because the interpreter is a prototype as the first step to construct an epistemic programming environment. The interpreter consists of the analyzer section and the attribute evaluation section. The analyzer section analyzes a program in an input source file and makes a parse tree. It has been implemented with SableCC [6]. Accordingly to semantic rules in the attributes grammar of EPLAS, the attribute evaluation section evaluates attributes on a parse tree made by the analyzer section. The attribute evaluation section has the symbol table, the beliefs manager, the epistemic states manager, and the reasoner to evaluate attributes.

The symbol table manages declaration of variables, functions, and procedures, data types and structured types of variables. It also assure that EPLAS is a strongly dynamic typed language. We implemented it with a hash table by a popular method.

The beliefs manager manages all input and/or derived beliefs and all their derivations trees, and provides functions to perform **input_belief**, **see_tree**, and **get_ancestors**. We implemented the beliefs manager as follows. The beliefs manager has a set of tuples where a tuple consists of a belief and a derivation tree of the belief. When performing **input_rule**, the beliefs manager analyzes

Table 1. Vocabulary of A Language Producing Beliefs

Vocabulary	Symbols
Constants	`a0, a1, ..., ai, ...`
Variables	`x0, x1, ..., xi, ...`
Functions	`f0, f1, ..., fi, ...`
Predicates	`P0, P1, ..., Pi, ...`
Connectives	`=>(entailment), &(and), !(negation)`
Quantifiers	`@(forall), #(exists)`
Punctuation	(,), ,

an input string according to a belief form. The belief is formed by a language including vocabulary in Table 1 and the following Production Rules 1 and 2.

Production Rule 1. *Term*
(1) Any constant is a term and any variable is also a term.
(2) If `f` is a function and `t0, ..., tm` are terms then `f(t0, ..., tm)` is a term.
(3) Nothing else is a term.

Production Rule 2. *Formula*
(1) If `P` is a predicate and `t0, ..., tm` are terms then `P(t0, ..., tm)` is a formula.
(2) If `A` and `B` are formulas then (`A => B`), (`A & B`), and (`! A`) are formulas.
(3) If `A` is a formula and `x` is a variable then (`@xA`), (`#xA`) are formulas.
(4) Nothing else is a formula.

The beliefs manager also adds new tuple of an input belief and a tree which has only root node denoting the input belief into the set. When performing **see_tree**(A), the beliefs manager outputs a derivation tree of A with JTree. When performing **get_ancestors**(A), the beliefs manager collects beliefs in a derivation tree of A by scanning the derivation tree and returns the beliefs.

The epistemic states manager manages all epistemic states from the primary epistemic state to the terminal epistemic state, and provides functions to perform **expand**, **contract**, **contract_derivation**, **return_to**, **get_state**, and **get_id**. We implemented the epistemic states manager as follows. The epistemic states has a sequence of sets of beliefs where a set of beliefs is an epistemic state, and the sequence is variable-length. When performing **expand**(S), the epistemic states manager appends a sum of a set of the bottom of the sequence and S to the sequence. When performing **contract**(S), the epistemic states manager appends a difference of a set of the bottom of the sequence and S. When performing **contract_derivation**(S), the epistemic states manager appends a difference of a set of the bottom of the sequence and all beliefs in derivation trees of S to the sequence. When performing **return_to**(i), the epistemic states manager appends

i–th epistemic state to the sequence. When performing **get_id**, the epistemic states manager returns a number of elements of the sequence. When performing **get_state**(i), the epistemic states manager returns a set of beliefs of the i–th element of the sequence.

The reasoner provides functions to perform **input_rule** and **reason**. We implemented the epistemic states manager as follows. There is an automated forward deduction system for general purpose entailment calculus EnCal [2,7]. EnCal automatically deduces new conclusions from given premises by applying inference rules to the premises and deduced results. Therefore, the reasoner has been implemented as an interface to EnCal. When performing **input_rule**, the reasoner analyzes an input string according to an inference rule form. The inference rule form is "$SLogicalSchema_1$, $\cdots$, $SLogicalSchema_n$, $SLogicalSchema_{n+1}$." "$SLogicalSchema$" is formed by a language including vocabulary in Table 2 and the following Production Rules 1 and 3.

Table 2. Vocabulary of A Language Producing Semi Logical Schema

Vocabulary	Symbols
Constants	`a0, a1, ..., ti, ...`
Variables	`x0, x1, ..., xi, ...`
Functions	`f0, f1, ..., fi, ...`
Predicates	`P0, P1, ..., Pi, ...`
Predicate Variable	`X1, X1, ..., Xi, ...`
Formula Variable	`A0, A1, ..., Ai, ...`
Connectives	`=>(entailment), &(and), !(negation)`
Quantifiers	`@(forall), #(exists)`
Punctuation	(,), ,

Production Rule 3. *Semi logical Schema*
(1) Any formula variable is a semi logical formula.
(2) If `P` is a predicate or a predicate variable and `t0, ..., tm` are terms then `P(t0, ..., tm)` is a semi logical formula.
(3) If `A` and `B` are semi logical formulas then (`A => B`), (`A & B`), and (`! A`) are semi logical formulas.
(4) If `A` is a semi logical formula and `x` is a variable then (`@xA`), (`#xA`) are semi logical formulas.
(5) Nothing else is a semi logical formula.

When performing **reason**(ir), the reasoner translates ir into an inference rule as an EnCal form and current beliefs which the epistemic states manager returns into formulas as an EnCal form, inputs these data to EnCal and executes EnCal, and then, gets the formulas derived by ir, translates the formulas into an EPLAS form, and makes the beliefs manager registers the formulas.

5 Concluding Remarks

As the first step to construct an epistemic programming environment, we proposed the first epistemic programming language, named 'EPLAS', and its interpreter implementation. EPLAS provides ways for scientists to write epistemic programs to help scientists with reasoning, dissolving contradictions, and trial-and-error. We also presented an interpreter implementation of EPLAS. We have provided the first environment to perform Epistemic Programming and run epistemic programs. In future works, we would like to establish Epistemic Programming methodology to make scientific discovery become a 'science' and/or an 'engineering'.

Acknowledgments

We would like to thank referees for their valuable comments for improving the quality of this paper. The work presented in this paper was supported in part by The Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Exploratory Research No. 09878061, and Grant-in-Aid for Scientific Research (B) No. 11480079.

References

1. AISE Lab., Saitama University.: EPLAS Reference Manual. (2007)
2. Cheng, J.: Encal: An automated forward deduction system for general–purpose entailment calculus. In Terashima, N., Altman, E., eds.: Advanced IT Tools, IFIP World Conference on IT Tools, IFIP96 - 14th World Computer Congress. Chapman & Hall (1996) 507–517
3. Cheng, J.: Epistemic programming: What is it and why study it? Journal of Advanced Software Research **6**(2) (1999) 153–163
4. Cheng, J.: A strong relevant logic model of epistemic processes in scientific discovery. In Kawaguchi, E., Kangassalo, H., Jaakkola, H., Hamid, I.A., eds.: "Information Modelling and Knowledge Bases XI," Frontiers in Artificial Intelligence and Applications. Volume 61., IOS Press (2000) 136–159
5. Flach, P.A., Kakas, A.C.: Abductive and inductive reasoning: background and issues. In Flach, P.A., Kakas, A.C., eds.: Abduction and Induction: Essays on Their Relation and Integration. Kluwer Academic Publishers (2000)
6. Gagnon, E.M., Hendren, L.J.: Sablecc http://www.sablecc.org.
7. Nara, S., Omi, T., Goto, Y., Cheng, J.: A general-purpose forward deduction engine for modal logics. In Khosla, R., Howlett, R.J., Jain, L.C., eds.: Knowledge-Based Intelligent Information and Engineering Systems, 9th International Conference, KES2005, Melbourne, Australia, 14-16 September, 2005, Proceedings, Part II. Volume 3682., Springer-Verlag (2005) 739–745
8. Peirce, C.S.: Collected Papers of Charles Sanders Peirce. Harvard University Press (1958)