Programming Languages and Artificial General Intelligence

Vitaly Khudobakhshov[1,2](✉), Andrey Pitko[2], and Denis Zotov[2]

[1] St.-Petersburg State University, St. Petersburg, Russia
vitaly.khudobakhshov@gmail.com
[2] ITMO University, St. Petersburg, Russia

Abstract. Despite the fact that there are thousands of programming languages existing there is a huge controversy about what language is better to solve a particular problem. In this paper we discuss requirements for programming language with respect to AGI research. In this article new language will be presented. Unconventional features (e.g. probabilistic programming and partial evaluation) are discussed as important parts of language design and implementation. Besides, we consider possible applications to particular problems related to AGI. Language interpreter for Lisp-like probabilistic mixed paradigm programming language is implemented in Haskell.

1 Introduction

For many years researches tried to create programming languages for specific areas of research. In the history of AI there were many attempts to create language that would be the best for artificial intelligence. The two main examples are Lisp and Prolog. First one is particularly interesting, because some code can be considered as data in very natural way. Second one contains powerful inference engine based on Horn logic as part of the language. Since that time significant progress have been made in theory of programming languages and many brilliant languages like Haskell were created. Unfortunately, many of achievements in this field are not yet widely used neither artificial intelligence, nor mainstream software development. This paper is related to two advanced techniques: probabilistic programming and partial evaluation. Importance of this techniques will be briefly discussed in this paper. These ideas can be considered as unconventional and not widely used outside of particular areas of research. Incorporation of such techniques to programming language may have considerable impact on artificial general intelligence.

The next section is about core language design, programming paradigm and basic features like pattern matching. Choice between domain-specific embedded language and full-featured general purpose language is also discussed.

One of the main issue need to be discussed is application of probabilistic programming to AGI. Generative models can be very useful in knowledge representation, as well as some other aspects of cognitive architectures. Probabilistic programming is discussed in Section 3.

© Springer International Publishing Switzerland 2015
J. Bieger (Ed.): AGI 2015, LNAI 9205, pp. 293–300, 2015.
DOI: 10.1007/978-3-319-21365-1_30

Section 4 is focused on a deep relationship between theory of programming languages and artificial general intelligence.

Last sections contain some implementation notes and future work road map. Language interpreter and tools are implemented in Haskell. Therefore, many issues about implementation of mixed paradigm languages in pure functional language are discussed. Related programming languages like Church are also discussed.

2 Language Requirements and Design

In this section we discuss main choices and tradeoffs one faces during programming language design. Our goal is to create programming languages with best capabilities for artificial general intelligence. We started from the following:

1. Turing-completeness
2. General purpose
3. Ease of use for automatic program transformation, generation and search
4. Mixed-paradigm (the language must support functional and imperative style)
5. Based on existent language to effectively adopt user experience and legacy code with minimum changes
6. Easily extendible syntax
7. Simplicity

The language should be powerful enough to make it possible to develop AGI systems (e.g. cognitive architecture). In other hand the language should be good enough not only for human beings, but for programs which use other programs (probably itself) as data.

Last requirement is to push us toward Lisp language family because it has a very natural quote syntax.

Another problem we should start to discuss is typing. Languages with static typing is a good choice for enterprise software development because many errors can be found during compilation. Many modern languages like Haskell and Scala have very difficult type system and it makes programming very tricky in some cases. If we want to satisfy simplicity requirement, we should choose dynamic typing. Mixed-paradigm in our case supposes that language should not be pure.

Scheme and Church are good examples of programming languages with simple and extendible syntax. In real world applications some additional syntactic sugar may significantly improve usability of language (see Clojure for example).

One of the most controversial choice has been made between general purpose and domain-specific (embedded) language. DSL can be Turing-complete and may have many extensions, like probabilistic programming or metacomputations. On the other hand, general purpose language needs to have a parser, interactive interpreter, and IDE. The problem of language embedding is ambivalent because pros and limitations are the same things. One can use DSL in his or her own favorite language and provide very high level of extensibility. Nevertheless, embedded language obliges to use this particular general purpose language in which DSL is embedded. Presented language is implemented in Haskell as general purpose.

Presented language is based on Scheme language with some useful extensions. Bread and butter of modern functional programming is pattern matching. In Scheme and Clojure this functionality provided by extended library. In this language we incorporate some syntactic ideas from Haskell to provide pattern matching in core language. Symbol `:` used to match `cons` and underscore as wildcard symbol:

```
(define (count x lst)
  (match lst
    (() 0)
    ((x : ys) (+ 1 (count x ys)))
    ((_ : ys) (count x ys))))
```

In this example pattern with dynamic properties has been used. Second pattern contains variable `x` which is used as argument of function `count`. Which means that if first element of `lst` equals to `x`, then we will have a match. Moreover, repeated variables are allowed (in this case, expression will be evaluated to `2`):

```
(match '(2 3 2)
  ((a : b : a : ()) a)
  (_ 0))
```

Although prefix nature of Lisp-like languages is broken here, it is only made to improve usability of the language. Pattern matching is a good extension to make programs more readable and compact, but not directly applicable to AGI problems. In next two sections we introduce probabilistic programming and partial evaluation.

3 Probablistic Programming

According to [3], probabilistic programming languages unify technique of classical models of computation with the representation of uncertain knowledge. In spite of the fact that the idea of probabilistic programming is quite old (see references in [6]), only in last few years researchers in cognitive sciences and artificial intelligence started to apply this approach. Many concepts in cognitive studies ? such as concept learning, causal reasoning, social cognition, and language understanding ? can be modeled using language with probabilistic programming support [4].

As usual, we extend deterministic language of general purpose with random choice primitives. The main obstacle in using probabilistic programming in large projects is the efficient implementation of inference. In modern probabilistic languages used various techniques and algorithms are used to solve this problem, including partial filtering [3], and Metropolis-Hastings algorithm [5]. In many cases programs need to be transformed to special form (e.g. continuation of passing style in WebPPL [3]). But main problem is that these languages are not ready for production use. If one wants to use such technique in his or her own project, one needs to embed particular language or extend it. Church is general enough, but it is not easy to extend; WebPPL is easy to embed or extend, but it is just a subset of JavaScript. In recent paper [1] genetic programming and simulated annealing were successfully applied to implementing inference procedure.There are implementation difficulties for such algorithms because they involve programming traces. In the section concerning implementation specifics more details will be covered.

In spite of a mixed paradigm nature of presented language, probabilistic programming is now allowed only for pure functional subset as in cases of WebPPL and Church. It is clear that random function cannot be pure, but we share the idea that concept of purity can be generalized to concept of exchangeability [5]: if an expression is evaluated several times in the same environment, the distribution on return values is invariant to the order of evaluations. In this sense further softening of such a requirement needs more research and not all language constructions are allowed for probabilistic programs in our language. Therefore, we can not use `set!` function in probabilistic program, but some useful features such as memoization can be extended to stochastic case [5]. This approach can be seen as division to pure and monadic code in Haskell. It can be useful in designing programs, like cognitive architectures, which use wide range of programming techniques. All of this can be written in the same language, but for probabilistic part using only the subset can be enough.

This approach is closely related to DSL mentioned in previous section. One of the most interesting examples of application probabilistic DSL is presented in [12].

Here we do not show examples of probabilistic programs, because we tried to provide compatibility with Church programming language up to minor issues, such as pattern matching.

Programming language presented here is an effort to create open and extendable language with probabilistic programming capabilities. Our implementation is based on ideas described in [14].The main difference from other implementations like Church and WebPPL is that inference algorithm is implemented in host language. Moreover no additional program transformation is needed.

4 Why Partial Evaluation Matters?

In this section one connection between programming languages in general and artificial intelligence will be discussed. In papers [9,13] possible application of partial evaluation was introduced. One should mention that there were some attempts long before this papers to apply partial evaluation to artificial intelligence (see for example [8]). Here new approach to understanding relations between two fields will be presented and discussed.

Lets start from general idea proposed by Futamura [2]. Let we have program `p` with two or more arguments written in language `S`, such that `p(x, y) = d`.

Here, `d` is a result of program execution. Suppose one have a program `spec` which can be applied to two arguments a program and the first argument and produce residual program of one argument `spec(p, x0) = p'` specialized for specified argument `x0`. Residual program `p'` satisfied an equation `p'(y) = p(x0, y) = d` for every `y`. But `p'` has possible optimizations according to knowledge of particular value `x0` and therefore work much faster.

This approach is very useful for automatic compiler construction. Suppose we have an interpreter of (source) language `S` written in (target) language `T` defined by `int(p, args) = d` (for more formal description see book [7]). One can apply specializer `spec` to interpreter `int` with respect to program `p`. It is easy to check that this will be the result of compilation from `S` to `T`.

In the context of artificial general intelligence this makes a connection between AGI and classical AI [9]. Here we need some philosophical remarks. Almost everybody knows a very famous proposition about general intelligence and specialization:

> *A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects.*
> *– Robert A. Heinlein*

It sounds reasonable, but in reality, the situation is different. Nobody asks painter to solve equations in mathematical physics. Moreover, we need to be precise and fast. If one needs to do accounting, then he or she definitely will use calculator to make job done. In this sense, ability to specialize by making tools is a crucial ability of general intelligence.

Is research in artificial general intelligence a replacement of good old-fashioned artificial intelligence? Suppose to be not. Imagine for a second that we have an AGI program which can solve almost all problems, but very slow. If we need to have effective solution of one particular problem, we have to develop optimized solution for the problem. But if we have ability to specialize our general program, we do not need to solve certain problem again anymore.

Is it possible to view AGI problems in terms of programming language theory and partial evaluation? Lets restrict ourselves to quasi-general example: general game playing. This example can be easily extended to AGI with some additional assumptions.

General game player is a program that must be able to play arbitrary logic game (sometimes only full information games considered) for one or more players. It can be either a 15-puzzle, or chess, or another game. The main point is that player gets the game rules seconds before the game starts. Handlers for five requests must be implemented: `info`, `start`, `play`, `stop` and `abort`. Function `start` receives the game rules described as open logic program written in Game Description Language [11]. After that player is involved into a request-response cycle with game server and `play` handler makes choices and realizes strategy of the game.

This interaction can be seen as classical Read–Evaluate–Print Loop of interactive interpreter. In such a way one can apply partial evaluation principles to artificial general intelligence. In the case of general game playing we will deduce specialized program which can play certain game by partially evaluating general program according to game rules.

Many players use advanced techniques to optimize program for particular games up to code generation and compilation [10]. We believe that it can be done by partial evaluation. It is clear that partial evaluation can not be very useful in search and do not provide heuristics for search optimization. It is proven that in many cases only linear speedup is possible [7]. But manipulating with GDL for computing legal moves and state has huge overhead and it can be removed by specialization.

Applying the idea to more general case including learning is also possible, independently of knowledge representation. In the case of procedural or symbolic representation, it is pretty straightforward. Possible applications of partial evaluation to neural networks are described in [7].

5 Implementation Issues

This section is about implementation details of the project. Besides the decision to implement general purpose language, choosing of implementation language is always coupled with some trade-offs. In our case, it was speed, development difficulty and extensibility. Only two candidates will be considered OCaml and Haskell. OCaml is good for catching imperative programming with full power of functional language, including pattern matching and algebraic data types. Haskell provides a more compact code with very good support of external libraries via foreign function interface, but it has some drawbacks connected with imperative issues, such as monads, lifting, and error handling. Choosing Haskell as implementation language is probably controversial in this case, but compiler quality and larger community were conclusive issues during the process of the decision making.

Language tools consist of following parts: interpreter, partial evaluator, and probabilistic programming support including tracer. All parts share some code according to language specification.

Interpreter uses `Parsec` library and support REPL mode. Double precision floating-point and arbitrary precision integers are supported. Strings are also supported as built-in type.

The crucial aspect of probablistic programming langauge is implementation of probablisic inference algorithm. As in many other probablistic languages Metropolis-Hastings is one of the most important sampling strategies. The implementation is based on a method carefully described in [14]. There are different ways to implement ERPs (elementary random primitives) - basic blocks of probablistic programs. To keep things simple we just maintain any key-value stucture for every random function where value is a tuple consisting of `likelihood`, `sample` and `proposal_kernel` functions for particular ERP.

To implement MCMC inference (in particular Metropolis-Hastings) one need to maintain a trace of the program. Trace consists of chunks - memoized random values:

```
data Chunk = Chunk { value::Value
                   , name::String
                   , erp::ERP
                   , args::[Value] }
```

where `value` is generated value wrapped up into language primitive, `name` is unique call address (see [14] for more information about structural naming strategy), `args` are parameters which used for generation.

Partial evaluation is implemented in Haskell. We use code annotation to describe static-dynamic division which means that original AST (abstract syntax tree) is transformed to annotated one before specialization [7].

The resulting language inherits some parts of Haskell semantics. It does not support lazy evaluation of infinite data structures, but it has some issues, for instance, normal evaluation order instead of applicative one. We do not force interpreter to evaluate arguments before passing. But final decision will be made later.

6 Conclusion and Future Work

Despite early stage of the work, it needs to be mentioned that it is the first attempt to create language with build-in support of both partial evaluation and probabilistic programming. At the level of intuition specialization and probabilistic programming are somehow connected and can be used effectively together. This project joins efforts to research in related fields.

Behind this work there is an idea to create cognitive architecture based on concepts mentioned above. We believe that probabilistic programming with partial evaluation may be effectively applied to AGI problems.

Many ideas of probabilistic programming will be already successfully applied to AGI problems [1] and computer vision in the context of AGI [12]. We are planning to incorporate this ideas to our language.

In this stage of the project probablistic programming and partial evaluation used independently and relationship between them is not very clear. Definitely inference algorithm can be considered as interpretation (in fact interpreter and MCMC query function use large amount of code with very small differences). In other hand it may be impractical or technically difficult to apply such kind of program transformation to inference algorithms. This is a important part of our future work. Moreover real application of this techniques to AGI is still challenging. In the next stage we are planning to create proof-of-concept intelligent software (e.g. cognitive architecture) which will extensively use probablistic programming and partial evaluation.

Acknowledgments. We would like to thank Alexey Potapov, Vita Batischeva and many others for very useful discussion which inspired us to make this work.

This work was supported by Ministry of Education and Science of the Russian Federation.

References

1. Batischeva, V., Potapov, A.: Genetic programming on program traces as an inference engine for probabilistic. In: These AGI-15 Proceedings (to appear)
2. Futamura, Y.: Partial evaluation of computation process an approach to a compiler-compiler. Systems, Computers, Controls **2**, 45–50 (1971)
3. Goodman, N.D., Stuhlmüller, A.: The design and implementation of probabilistic programming languages (retrieved on 2015/3/30). http://dippl.org
4. Goodman, N.D., Tenenbaum, J.B.: Probabilistic models of cognition (retrieved on 2015/3/30). http://probmods.org
5. Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., Tarlow, D.: Church: a language for generative models. In: Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI), pp. 220–229 (2008)
6. Jones, C., Plotkin, G.D.: A probablistic powerdomain of evaluations. In: Proceedings of Fourth Annual Symposium on Logic in Computer Science, pp. 186–195. IEEE Computer Society Press (1989)
7. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1994)
8. Kahn, K.: Partial evaluation, programming methodology, and artificial intelligence. AI Magazine **5**, 53–57 (1984)
9. Khudobakhshov, V.: Metacomputations and program-based knowledge representation. In: Kühnberger, K.-U., Rudolph, S., Wang, P. (eds.) AGI 2013. LNCS, vol. 7999, pp. 70–77. Springer, Heidelberg (2013)
10. Kowalski, J., Szykuła, M.: Game description language compiler construction. In: Cranefield, S., Nayak, A. (eds.) AI 2013. LNCS, vol. 8272, pp. 234–245. Springer, Heidelberg (2013)
11. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: game description language specification. Tech. rep., Stanford Logic Group Computer Science Department Stanford University, Technical Report LG-2006-01 (2008)
12. Potapov, A., Batischeva, V., Rodionov, S.: Optimization framework with minimum description length principle for probabilistic programming. In: These AGI-15 Proceedings (to appear)
13. Potapov, A., Rodionov, S.: Making universal induction efficient by specialization. In: Goertzel, B., Orseau, L., Snaider, J. (eds.) AGI 2014. LNCS, vol. 8598, pp. 133–142. Springer, Heidelberg (2014)
14. Wingate, D., Stuhlmüller, A., Goodman, N.D.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Proc. of the 14th Artificial Intelligence and Statistics (2011)