Metacomputations and Program-Based Knowledge Representation

Vitaly Khudobakhshov

St.-Petersburg State University, St.-Petersburg, Russia
vitaly.khudobakhshov@gmail.com

Abstract. Computer programs are a very attractive way to represent knowledge about the world. A program is more than just objects and relations. It naturally provides information about evolution of a system in time. Therefore, programs can be considered the most universal data structures. The main problem with such representation is that it is much more difficult to deal with programs than with usual data structures. Metacomputations are a powerful tool for program analysis and transformations. This paper describes artificial general intelligence from metacomputational point of view. It shows that many methods of metacomputations e.g. supercompilation and Futamura projections can be applied to AGI problems.

1 Introduction

Let us start from a brief overview of the subject, as it can lead us to the roots of ideas behind this paper. Metasystem Transition Theory [1] was developed in 70s by V. Turchin[1]. This theory evolved into the theory of metacomputations. It describes properties of programs which manipulate other programs as data. Such programs are called metaprograms. The former theory describes basic rules of evolution in terms of system-metasystem transitions in which the next step sets up every system as an object of manipulation of a system of the next level, i.e. a metasystem, and so on. In these terms, metaprograms manipulate other programs and a metaprogram can be applied to itself. This leads us to the idea of evolution in the world of programs. Program manipulations include analysis and transformations. The crucial idea that was pointed out by Turchin is that metacomputations can be applied to any science if it has a formal linguistic description [2].

In this paper, the ideas of metacomputations are applied to artificial general intelligence by means of creating a suitable formal description.

It is easy to understand that static program analysis is difficult due to the halting problem. Therefore, metacomputations are based on process-oriented approaches: to analyze and transform a program p, the metaprogram M must run it somehow and observe states of the process of execution of p. More precisely,

[1] Valentin Turchin is the author of Refal programming language, and he developed the methods of metacomputations for Refal.

M runs p with some data d or the whole set or a class of data $\mathcal{C}$. A language of implementation of programs should also be fixed. Let us call it the reference language R. If all programs including metaprograms are written in the language R, then metaprograms can be applied to themselves. One of the most challenging tasks in artificial intelligence is to find a suitable knowledge representation. Data structures determine algorithms we are using to work with them. Usual data structures, e.g. lists, trees and graphs are well understood, and there are a lot of effective algorithms to work with the corresponding structures. Programs are more flexible and can represent notion of time in a very natural way, because we used it for modeling processes in the world. However, we need sophisticated methods for program transformation to work with such representation. In the following sections, these methods will be examined and their possible interpretation will be discussed in the context of artificial general intelligence.

2 Specialization and Futamura Projections

In this section, one of the most fundamental results in metacomputations will be discussed. Is it possible to transform one program into another with completely different but desirable properties? Y. Futamura presented the first nontrivial example of such a transformation in his famous paper [3]. He described a theoretical possibility to create a compiler directly from an interpreter using a metaprogram called specializer.

For the sake of clarity of explanation, we will sometimes deviate from rigor in the mathematical sense. Lets define a program p in a language L. We will write $p \in L$ or just p_L in this case. We assume that r is a result of execution of the program p_L with data $d \in D$, where D is a set of all possible data and we will write it as $r = p_L(d)$. As stated above, the reference language R is fixed, and in order to rewrite the last equation in terms of the reference language, we need to introduce an interpreter of the language L written in R. We will denote it as $intL \in R$. In this case, the equation will be as follows:

$$p_L(d) = intL(p_L, d). \tag{1}$$

Equivalence in (1) means that the left and right hand sides are defined in the same domain. The specializer $spec_R$ is defined as a program with the following behavior:

$$p(x, y) = spec_R(p, x)(y) \tag{2}$$

for all $x, y \in D$ and all $p \in R$. Here $spec_R$ makes a deep transformation which makes $p|_x = spec_R(p, x)$ efficient. Therefore, we can say that $p(x, y)$ was partially evaluated to $p|_x$ and $p(x, y) = p|_x(y)$ for any $y \in D$. Let us apply the specializer to the interpreter defined in (1) as follows:

$$intL(p_L, d) = spec_R(intL, p_L)(d). \tag{3}$$

This is the first Futamura projection. The value of $spec_R(intL, p_L)$ is a program in R language and it should be interpreted as a result of compilation of the program p_L from L to R. Since $spec_R$ is just a two-argument program one can immediately apply the rule (2) to $spec_R$ itself and get the second projection

$$spec_R(intL, p_L)(d) = spec_R(spec_R, intL)(p_L)(d). \tag{4}$$

Now, the right hand side of the last equation gives us the compiler from L to R. Moreover it is possible to apply the specializer once again because we have the same program $spec_R$ in the right hand side, but with another two arguments. It leads us to the third projection:

$$spec_R(spec_R, intL)(p_L)(d) = spec_R(spec_R, spec_R)(intL)(p_L)(d). \tag{5}$$

The result of the $spec_R(spec_R, spec_R)$ execution is a program which for every interpreter $intL$ of the language L (obviously, it can be any interpreter of any language) generate a compiler from L to R.

Is it possible to use this technique outside the interpreter-compiler example? Consider a program $pred \in R$ with two arguments e and w, where e is a empirical knowledge about the world and $w \in W$ is a world state. Empirical knowledge can be a table-like structure which maps world states to itself. Program $pred$ is a program which predicts the next world state according to experience e and the current world state w.

Let us try to apply Futamura projections to such a system. After applying specializer for the first time, we will have the following result:

$$pred(e, w) = spec_R(pred, e)(w). \tag{6}$$

One can find a suitable meaning for this equation. $spec_R(pred, e)$ is a program in language R and it can be treated as a world model based on experience during the prediction process. The next projection leads us to a model generator

$$spec_R(spec_R, pred)(e)(w), \tag{7}$$

which has semantics similar to the compiler in the second Futamura projection. In fact, the model generator for any experience e returns the required model in R language.

After applying the specializer to (7), one will have the same result as in the compiler case (5). However, this time it can be considered a generator of generators. If $spec_R(spec_R, spec_R)$ is applied to $pred$, then it will be a program which generates a model generator for the specified predictor $pred$.

3 Another Application of Specializer

This section describes another application of metacomputations. Let us describe the world as a sequence of states W. As in the previous section, $w \in W$ describes the current state of the world. For example, it can be realized as k-dimensional vector for some k. One can imagine that evolution of the world is specified by the function $n(w)$ that returns the next state of the world for the given one, so we will have a chain of states $n(w_0) = w_1$, $n(w_1) = w_2$ and so on[2]. We know nothing about the internal structure of the function n, it is completely imaginary and is used only to describe the sequential nature of W.

Let us consider a model m of the world W. This model is a program in some language L. The model may be good or bad depending on how precisely it represents the world W. Predictions of states of the world can be described by the following equation: $m(w) = \bar{w}$, where $\bar{w}$ is the predicted next state.

Let $imp \in R$ will be a program which improves the specified model m according to a new (actual) world state w' by the following rule:

$$imp(m, w, w') = \begin{cases} m, & m(w) = w' \\ m' : m'(w) = w', & m(w) \neq w' \end{cases}, \tag{8}$$

where w is defined by an equation $n(w) = w'$, i.e. it is the previous world state. One can improve the model of the world by an iterative application of the program imp to every new state. Improvement process is assumed to be continuous, i.e. $m(w_0) = m'(w_0) = w$, where w_0 is a state before w. Before the examination of properties of the program imp, let us define its place in the whole intelligent system. The program can be considered a desirable component of the system. However, the system must have other components.

Obviously, the program provides only one way interaction. In this case, only the world can affect the system. If we want to achieve practical success, we need to add a component that can modify the world state. Intentions of the system are out of scope of this paper, but one thing should be mentioned. After intention is stated, the system must be able to achieve the desirable world state. To do so, one must require the language L to have an inverse semantics [4]. This means the program m will be invertible, i.e. m^{-1} exists in some sense. This property leads us to some special cases including the situation in which desirable goals can not be achieved. It happens if there is no $x \in W$ for the given w, such that $m(x) = w$, or a solution can not be found in reasonable time due to computational complexity. The inversion problem will be discussed in the next section. On the other hand, if m is invertible, then one can have a history chain started from given w

$$m^{-1} \circ m^{-1} \circ \ldots \circ m^{-1}(w), \tag{9}$$

as opposed to the prediction chain defined by the composition of m.

Iterative application of metaprogram imp to its arguments can be considered as a reasoning about the world.

Futamura projections do not have special meaning in this case, but one can suggest possible generalization of equation (8) as follows:

$$imp(y, x, x') = \begin{cases} y, & y(x) = x' \\ y' : y'(x) = x', & y(x) \neq x' \end{cases}. \tag{10}$$

[2] To be precise, $n(w)$ returns the next *observable* state, because the world usually evolves much faster than any system can react.

In this equation, y, x and x' are programs and y also can be considered a mataprogram, because it transforms x somehow. In this case, the sign "=" implies equivalence in the most general sense.

The next step includes description of world states in terms of programs, i.e. we would like to consider w as a program. Unification of the programming language to the reference one, i.e. $L = R$ is also a very important assumption. These requirements allow to get (8) from (10) by substitution $y \leftarrow m$, $x \leftarrow w$ and $x' \leftarrow w'$, moreover we can use projections for partial self-application.

Specialization can be done for the arbitrary argument and one can get the following projection by applying the program s twice: the first time for the second argument w and the second time for third argument w' (or vice versa):

$$imp|_{w,w'}(m) = spec_R(spec_R(imp, w), w')(m). \tag{11}$$

It allows to apply the program imp to $imp|_{w,w'}$ for some m_1 and m_2:

$$imp' = imp(imp|_{w,w'}, m_1, m_2). \tag{12}$$

According to (10), it can be treated as an improvement of improvement mechanism with a fixed world state. In this case, imp' is a one-argument program, and it returns a new model for the old one. The interpretation of the equation (12) is clear. Consider a situation in which there are two intelligent systems of the same type defined by the equation (10) and these systems share the same world[3]. Let us call these systems $\mathcal{S}$ and $\mathcal{T}$. According to our assumption, both systems are tuples $(imp_{\mathcal{S}}, m_{\mathcal{S}}, \ldots)$ and $(imp_{\mathcal{T}}, m_{\mathcal{T}}, \ldots)$ respectively, where dots mean that there are some other components of these systems we are not interested in[4].

Suppose that we need to provide the knowledge transfer from $\mathcal{T}$ to $\mathcal{S}$. One can think that $\mathcal{T}$ is a teacher and $\mathcal{S}$ is a student. Only two methods are available for the transfer. One can use $m_{\mathcal{T}}$ as a bootstrap for $\mathcal{S}$. In this case, we will have an improved world model provided by the following procedure: $imp_{\mathcal{S}}(m_{\mathcal{T}}, w, w')$. Another method of knowledge transfer from one system to another is to describe the improvement procedure based on examples, i.e. $imp_{\mathcal{S}}|_{w,w'}$ can be treated as the improvement procedure. Using (12), the system $\mathcal{T}$ can improve $imp_{\mathcal{S}}|_{w,w'}$ by

$$imp'_{\mathcal{S}} = imp_{\mathcal{T}}(imp_{\mathcal{S}}|_{w,w'}, m_{\mathcal{S}}, m_{\mathcal{T}}). \tag{13}$$

Obviously, $imp'_{\mathcal{S}}$ is not useful if the world state has changed. Therefore, after application of this metaprogram to some model m we have to use the old $imp_{\mathcal{S}}$ metaprogram to provide continuous evolution of the model according to the changing world state. We can also describe the result as follows: *there is no way to improve the improvement process in a global context in terms of the improvement process.*

In this section, we have deduced useful properties of intelligent systems of the given type using metacomputations.

[3] We also assume that they are somehow separated from the world relative to each other.

[4] One can require for these components that they can not change an internal structure of *imp*.

4 Supercompilation and Inversion

Supercompilation is a very important technique in metacomputations due to the huge practical interest. Starting from the early works, it was the most claiming metacomputational technique. It is on the frontier of computer science today. Despite of many works published and some practical supercompilers developed, this technique is not widely used. A detailed discussion is out of purpose of this paper, but one important result for practical application to artificial general intelligence will be briefly discussed below.

The term supercompilation was proposed by Turchin. He described this term as follows[5]:

> A program is seen as a machine. To make sense of it, one must observe its operation. So a supercompiler does not transform the program by steps; it controls and observes (SUPERvises) the machine, let us call it M_1, which is represented by the program. In observing the operation of M_1, the supercompiler COMPILES a program which describes the activities of M_1, but it makes shortcuts and whatever clever tricks it knows, in order to produce the same effect as M_1, but faster. the goal of the supercompiler is to make the definition of this program (machine) M_2, self-sufficient. When this is achieved, it outputs M_2, and simply throws away the (unchanged) machine M_1. A supercompiler would run M_1 in a general form, with unknown values of variables, and create a graph of states and transitions between possible configurations of the computing system. . . in terms of which the behavior of the system can be expressed. Thus the new program becomes a self-sufficient model of the old one.

The process described can be considered as an equivalent program transformation. The main goal of such transformations is to make the program more efficient. Methods of the transformation are well known [6].

One of the most interesting effects of supercompilation is that it can transform ineffective program to an effective one. A well known example of such an improvement is the KMP-test. For the given naive string matching program, a supercompiler transforms it to an effective Knuth-Morris-Pratt matching program. This property makes supercompiler a desirable component of an intelligent system with the program-based knowledge representation. Moreover supercompilation with specialization gives us a very important relationship between artificial general intelligence and narrow intelligence:

$$nAI = sc(spec(AGI, pd)), \tag{14}$$

where *spec* is a specializer, *sc* is a supercompiler. In the equation narrow artificial intelligence can be obtained by supercompilation of specialized AGI system for given problem description *pd*. Of course we cannot achieve more than AGI can do but we can get optimized solution of the problem by removing all unnecessary computational steps. In theory it is a useful tool to compare general intelligent systems because it reduces these systems to unified basis (it is clear if we compare systems written in the same language).

In the previous section, the model inversion was used to obtain particular state changes that the system must undergo to achieve goals. Turchin proposed Universal Resolving Algorithm (URA) to solve the problem [7]. A specializer allows to obtain a program invertor and an invertor generator using URA [8,4] in the same manner as described above for the model generator and for the Futamura case.

To apply inversion to our problem, we should supply the language, in which models are described, with inverse semantics. Examples of such languages can be found in [4].

5 Conclusion

Application of metacomputations to artificial intelligence is not quite new. Kahn in [9] discussed possible applications of partial evaluation to AI. Possibility of using specialization to wide class of programs with interpretator-like behavior was mentioned in [10].

In this paper, metacomputations were examined in the context of artificial general intelligence. Metacomputations were applied to program-based knowledge representation to get a new description of AGI problems. The fundamental Futamura result was extended to intelligent systems with predictors. As a model can be explicitly constructed from the predictor, it can be used as a bootstrap for the system (8).

After the generalization of the system (8) to the case (10), some important limitations of self-application for such systems were discovered. From the philosophical point of view, these limitations are very natural and can be described by the following words of Pythagoras: *“You cannot explain everything to everyone”*. In this case program *imp* can be treated as a talent which cannot be overcome. But if we want to improve *imp* program we need to have a higher level metaprogram which will provide an evolution of *imp*. Therefore, it is sufficient to have *imp* program in the form of (8) for practical purposes.

Due to technical difficulties one can confront with during construction and using supercompiler, equation (14) can be written in the weaker form

$$nAI = spec(AGI, pd), \tag{15}$$

but it may be sufficient for practical purposes. Furthermore in the case of limited resources nAI can be considerably stronger than its general counterpart. In time bounded conditions inequality

$$t_{spec}(AGI, pd) + t_{nAI} < t_{AGI}(pd) \tag{16}$$

gives us a speedup and therefore a better result. Inequality (16) is a variation of inequality given in [10].

This work should be considered only the first step of the research towards creating the real world intelligent systems based on the metacomputational approach.

References

1. Turchin, V.F.: The Phenomenon of Science. Columbia University Press, New York (1977)
2. Glück, R., Klimov, A.: Metacomputation as a Tool for Formal Linguistic Modeling. In: Cybernetics and Systems 1994, pp. 1563–1570. World Scientific (1994)
3. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls 2(5), 45–50 (1971)
4. Romanenko, A.Y.: Inversion and metacomputation. In: Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 12–22. ACM Press (1991)
5. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(3), 292–325 (1986)
6. Sørensen, M.H.: Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation. Master's thesis, Københavns Universitet, Datalogisk Institut (1994)
7. Turchin, V.F.: Equivalent Transformations of Recursive Functions defined in REFAL. In: Teoria Yasykov i Methody Postroenia System Programirowania. Trudy Symp. Kiev-Alushta, pp. 31–42 (1972) (in Russian)
8. Abramov, S.M.: Metacomputation and Logic Programming. In: Semiotic Aspects of Formalization of the Intellectual Activity. All-union School-workshow 'Borjomi 1988', Moscow (1988) (in Russian)
9. Kahn, K.: Partial Evaluation, Programming Methodology, and Artificial Intelligence. AI Magazine 5(1), 53–57 (1984)
10. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1994)