A Representation of Programs for Learning and Reasoning

Moshe Looks
Google, Inc.
madscience@google.com

Ben Goertzel
Novamente LLC
ben@novamente.net

Abstract

Traditional machine learning systems work with relatively flat, uniform data representations, such as feature vectors, time-series, and context-free grammars. However, reality often presents us with data which are best understood in terms of relations, types, hierarchies, and complex functional forms. One possible representational scheme for coping with this sort of complexity is computer programs. This immediately raises the question of how programs are to be best represented. We propose an answer in the context of ongoing work towards artificial general intelligence.

Background and Motivation

First, what do we mean by programs? The essence of programmatic representations is that they are well-specified, compact, combinatorial, and hierarchical. *Well-specified:* unlike sentences in natural language, programs are unambiguous; two distinct programs can be precisely equivalent. *Compact:* programs allow us to compress data on the basis of their regularities. Accordingly, for the purposes of this paper, we do not consider overly constrained representations such as the well-known conjunctive and disjunctive normal forms for Boolean formulae to be programmatic. Although they can express any Boolean function (data), they dramatically limit the range of data that can be expressed compactly, in comparison to unrestricted Boolean formulae (Weg87; Hol90). *Combinatorial:* programs can access the results of running other programs (e.g. via function application), as well as delete, duplicate, and rearrange these results (e.g., via variables or combinators). *Hierarchical:* programs have an intrinsic hierarchical organization, and may be decomposed into subprograms.

Baum has advanced a theory "under which one understands a problem when one has mental programs that can solve it and many naturally occurring variations." (Bau06). Accordingly, one of the primary goals of artificial general intelligence will be system that can represent, learn, and reason about such programs (Bau06; Bau04). Furthermore, integrative AGI systems such as Novamente (LGP04) may contain subsystems operating on programmatic representations. Would-be AGI systems with no direct support for programmatic representation will clearly need to represent procedures and procedural abstractions *somehow*. Alternative representations such as recurrent neural networks have serious downsides, however, including opacity and inefficiency.

It is worth noting that the problem of how to represent programs for an AGI system dissolves in the limiting case of unbounded computational resources. The solution is algorithmic probability theory (Sol64) extended recently to the case of sequential decision theory (Hut05). The latter work defines the universal algorithmic agent AIXI, which in effect simulates all possible programs that in agreement with the agent's set of observations. While AIXI is uncomputable, the related agent AIXI*tl* may be computed, and is superior to any other agent bounded by time t and space l (Hut05). The choice of a representational language for programs[1] is of no consequence, as it will merely introduce a bias that will disappear within a constant number of time steps.[2]

The contribution of this paper may be seen as providing practical techniques for approximating the ideal provided by algorithmic probability, based on what Pei Wang has termed the *assumption of insufficient knowledge and resources* (Wan06). Given this assumption, how programs are represented is of paramount importance, as we shall see in the next section. What exactly is meant by a representation is also addressed. The third section of the paper delves into effective techniques for representing programs. The fourth and final section concludes and suggests future work.

Representational Challenges

Despite the advantages outlined in the previous section, there are a number of challenges in working with programmatic representations:

[1] As well as a language for proofs in the case of AIXI*tl*.

[2] In fact, the universal distribution converges very quickly (Sol64).

- **Open-endedness** – in contrast to other knowledge representations current in machine learning, programs may vary in size and "shape", and there is no obvious problem-independent upper bound on program size. This makes it difficult represent programs as points in a fixed-dimensional space, or learn programs with algorithms that assume such a space.
- **Over-representation** – often, syntactically distinct programs will be semantically identical (i.e. represent the same underlying behavior or functional mapping). Lacking prior knowledge, many algorithms will inefficiently sample semantically identical programs repeatedly (Loo07b; GBK04).
- **Chaotic Execution** – programs that are very similar, syntactically, may be very different, semantically. The presents difficult for many heuristic search algorithms, which require syntactic and semantic distance to be correlated (Loo07c; TVCC05).
- **High resource-variance** – programs in the same space may vary greatly in the space and time they require to execute.

Based on these concerns, it is no surprise that search over program spaces quickly succumbs to combinatorial explosion, and that heuristic search methods are sometimes no better than random sampling (LP02). Regarding the difficulties caused by over-representation and high resource-variance, one may of course raise the objection that determinations of e.g. programmatic equivalence for the former, and e.g. halting behavior for the latter, are uncomputable. Given the assumption of insufficient knowledge and resources however, these particular concerns dissolve into the larger issue of computational intractability and the need for efficient heuristics. Determining the equivalence of two Boolean formulae over 500 variables by computing and comparing their truth tables is trivial from a computability standpoint, but, in the words of Leonid Levin, "only math nerds would call 2^{500} finite" (Lev94). Similarly, a program that never terminates is a special case of a program that runs too slowly to be of interest to us.

Now, in advocating that these challenges be addressed through "better representations", it must be clear that we do not mean merely trading one Turing-complete programming language for another; in the end it will all come to the same. Rather, as we have argued previously (Loo06; Loo07a; Loo07c), we claim that to tractably learn and reason about programs requires us to have prior knowledge of programming language semantics. The mechanism whereby programs are executed is known *a priori*, and remains constant across many problems. We have proposed, by means of exploiting this knowledge, that programs be represented in normal forms that preserve their hierarchical structure, and heuristically simplified based by reduction rules. Accordingly, one formally equivalent programming language may be preferred over another by virtue of making these reductions and transformations more explicit and concise to describe and to implement.

Normal forms have been proposed and experimentally validated for Boolean formulae, and for a number of toy problems, such as the artificial ant on the Santa Fe trail (Loo06). This paper has two main innovations. Firstly, the system of normal forms is extended to encompass a full programming language. Secondly, a more extended taxonomy of programmatic transformations beyond simplification are proposed to aid in learning and reasoning about programs, situated in a unified framework.

Tractable Representations

In order to distinguish between the various normal forms we introduce below, a simple type system is introduced. This is necessary in order to convey the minimal information needed to correctly apply the basic functions in our canonical forms. Various systems and applications may of course augment these with additional type information, up to an including arbitrary the satisfaction of arbitrary predicates (e.g. a type for prime numbers). This can be overlaid on top of our minimalist system to convey additional bias in selecting which transformations to apply, introducing constraints as necessary. For instance, a call to a function expecting a prime number, called with a potentially composite argument, may be wrapped in a conditional testing the argument's primality. A similar technique is used in the normal form for functions to deal with arguments that are (possibly empty) lists.

Normal Forms

Normal forms are provided for *Boolean* and *number* primitive types, and the following parametrized types:

- list types, $list_T$, where T is a type with a normal form,
- tuple types, $tuple_{T_1,T_2,\ldots T_N}$, where all T_i are types with normal forms, and N is a positive natural number,
- enum types, $\{s_1, s_2, \ldots s_N\}$, where N is a positive number and all s_i are unique identifiers,
- function types $T_1, T_2, \ldots T_N \rightarrow O$, where O and all T_i are types with normal forms,
- action result types.

A list of type $list_T$ is an ordered sequence consisting any number of elements, all of which must have type T. A tuple of type $tuple_{T_1,T_2,\ldots T_N}$ is an ordered sequence of exactly N elements, where every ith element is of type T_i. An enum of type $\{s_1, s_2, \ldots s_N\}$ is some element s_i from the set. Action result types concern side-effectful interaction with some world external to the system (but perhaps simulated, of course), and will be described in detail in their subsection below. Other types may certainly be added at a later date, but we believe that those listed above provide sufficient expressive power to conveniently encompass a wide range of programs, and serve as a compelling proof of concept.

The normal form for some type T is a set of elementary functions with codomain T, a set of constants of type T, and a tree grammar, G. Internal nodes for expressions described by G are elementary functions, and leaves are the symbols of the form U_{var} or $U_{constant}$, where U is some type with a normal form (often $U = T$).

Sentences in a normal form grammar may be transformed into normal form expressions. The set of expressions that may be generated is a function of a set of bound variables and a set of external functions that must be provided (both bound variables and external functions are typed). The transformation is as follows:

- leaves labeled $T_{constant}$ are replaced with constants of type T,
- leaves labeled T_{var} are replaced with either bound variables matching type T, or expressions of the form $f(expr_1, expr_2, \ldots expr_M)$, where f is an external function of type $T_1, T_2, \ldots T_M \rightarrow T$, and each $expr_i$ is a normal form expression of type T_i (given the available bound variables and external functions).

Boolean Normal Form

The elementary functions for Boolean normal form are *and*, *or*, and *not*. The constants are $\{true, false\}$. The grammar is:

```
bool_root = or_form  | and_form
          | literal  | bool_constant
literal   = bool_var | not( bool_var )
or_form   = or( {and_form | literal}{2,} )
and_form  = and( {or_form | literal}{2,} ) .
```

The construct `foo{x,}` refers to `x` or more matches of `foo` (e.g. `{or_form | literal}{2,}` is two or more items in sequences where each item is either an `or_form` or a `literal`).

Number Normal Form

The elementary functions for number normal form are *times* and *plus*. The constants are some specified subset of the rationals (e.g. those with corresponding IEEE single-precision floating-point representations). The normal form is as follows:

```
num_root   = times_form   | plus_form
           | num_constant | num_var
times_form = times( {num_constant |
                     plus_form}
                    plus_form{1,} )
           | num_var
plus_form  = plus( {num_constant  |
                    times_form}
                   times_form{1,} )
           | num_var .
```

List Normal Form

For list types $list_T$, the elementary functions are *list* (an n-ary list constructor) and *append*. The only constant is the empty list (*nil*). The normal form is as follows:

```
list_T_root = append_form | list_form
            | list_T_var  | list_T_constant
append_form = append( {list_form |
                       list_T_var}{2,} )
list_form   = list( T_root{1,} ) .
```

Tuple Normal Form

For tuple types $tuple_{T_1,T_2,\ldots T_N}$, the only elementary function is the tuple constructor (*tuple*). The constants are T_1`_constant`$\times T_2$`_constant`$\times \cdots \times T_N$`_constant`. The normal form consists simply of either a constant, a var, or `tuple(` T_1`_root` T_2`_root` $\ldots T_N$`_root )`.

Enum Normal Form

Enums are atomic tokens with no internal structure - accordingly, there are no elementary functions. The constants for the enum $\{s_1, s_2, \ldots s_N\}$ are the s_is. The normal form is either a constant or a var.

Function Normal Form

The normal form for a function of type $T_1, T_2, \ldots T_N \rightarrow O$ is a lambda-expression of arity N whose body is of type O. The list of variable names for the lambda-expression is not a "proper" argument - it does not have a normal form of its own. Assuming that none of the T_is is a list type, the body of the lambda-expression is simply in the normal form for type O (with the possibility of the lambda-expressions arguments appearing with their appropriate types). If one or more T_is is a list type, then the body is a call to the *split* function (described below), whose arguments are all in normal form.

Split is a family of functions with type signatures

$$(T_1, list_{T_1}, T_2, list_{T_2}, \ldots T_k, list_{T_k} \rightarrow O),$$
$$tuple_{list_{T_1},O}, tuple_{list_{T_2},O}, \ldots tuple_{list_{T_k},O} \rightarrow O.$$

When a call $split(f, tuple(l_1, o_1), tuple(l_2, o_2), \ldots tuple(l_k, o_k))$ is made, the list arguments $l_1, l_2, \ldots l_k$ are examined sequentially. If some l_i is found that is empty, then the result is the corresponding value o_i. If all l_i are nonempty, we can deconstruct each of them $x_i : xs_i$, where x_i is the first element of the list and xs_i is the remainder of the list. The result is then $f(x_1, xs_1, x_2, xs_2, \ldots x_k, xs_k)$. The *split* function thus acts as an implicit case statement to deconstruct lists if they are nonempty, and return some other value if they are empty.

Action Result Normal Form

An action result type *act*, corresponds to the result of taking an action in some world. Every action result type has a corresponding world type, *world*. Associated with action results and word are two special sorts of functions.

- *Perceptions* - functions that take a *world* as their first argument and normal (i.e. non-world and non-action-result) types as their remaining arguments, and return normal types. Unlike normal functions the result of evaluating a perception call may be different at different times.
- *Actions* - functions that take a *world* as their first argument and normal types as their remaining arguments, and return action results (of the type associated with the type of their world argument). As with perceptions, the result of evaluating an action call may be different at different times. Furthermore, actions may have *side effects* in the associated world that they are called in. Thus, unlike any other sort of function, actions *must* be evaluated, even if their return values are ignored.

Other sorts of functions acting on worlds (e.g. ones that take multiple worlds as arguments) are disallowed.

Note that an action result expression cannot appear nested inside an expression of any other type. Consequently, there is no way to convert e.g. an action result to a Boolean, although conversion in the opposite direction is permitted. This is required because mathematical operations in our language have classical mathematical semantics; *x and y* must equal *y and x*, which will not generally be the case if x or y can have side-effects. Instead, there are special sequential versions of logical functions which may be used instead.

The elementary functions for action result types are and_{seq} (sequential and, equivalent to *C*'s short-circuiting `&&`), or_{seq} (sequential or, equivalent to *C*'s short-circuiting `||`), and $fails$ (negates success to failure and vice versa). The constants may vary from type to type but must at least contain *success*, indicating absolute success in execution. The normal form is as follows:

```
act_root    = orseq_form | andseq_form
            | seqlit
seqlit      = act | fails( act )
act         = act_constant | act_var
orseq_form  = orseq( {andseq_form |
                      seqlit}{2,} )
andseq_form = andseq( {orseq_form
                     | seqlit}{2,} ) .
```

Note that a $do(arg_1, arg_2, \ldots arg_N)$ statement (known as *progn* in Lisp), which evaluates its arguments sequentially regardless of success or failure, is equivalent to $and_{seq}(or_{seq}(arg_1, success), or_{seq}(arg_2, success), \ldots\ or_{seq}(arg_N, success))$.

Program Transformations

A program transformation is any type-preserving mapping from expressions to expressions. Transformations may be semantics preserving, or may potentially alter semantics. In the context of program evolution there is an intermediate category of fitness preserving transformations that may or may not alter semantics. In general the only way that fitness preserving transformations will be uncovered is by scoring programs that have had their semantics potentially transformed to determine their fitness.

Reductions

Reductions are semantics preserving transformations that do not increase some size measure (typically number of discrete symbols) of expressions, and are idempotent. A set of *canonical reductions* is defined for every type that has a normal form. For example, $and(x, x, y) \rightarrow x$ is a reduction for Boolean expressions. For numerical functions, the simplifier in a computer algebra system may be used. The full list of reductions is omitted in this paper for brevity - see (Loo06; Loo08) for details. An expression is *reduced* if it maps to itself under all canonical reductions for its type, and all of its children are reduced.

Another important set of reductions are the *compressive abstractions*, which reduce or keep constant the size of expressions by introducing subfunctions. Consider the expression

```
list( times( plus( a, p, q ) r ),
      times( plus( b, p, q ) r ),
      times( plus( c, p, q ) r ) )
```

which contains 19 symbols. By introducing the subfunction

```
f( x ) = times( plus( x, p, q ) r ),
```

and transforming the expression to

```
list( f( a ), f( b ), f( c ) ),
```

the total number of symbols is reduced to 15. One can quite naturally generalize this notion to consider compressive abstractions across a set of programs, as in the PLEASURE approach (Goe08). Compressive abstractions unfortunately appear to be rather expensive to uncover, although perhaps not prohibitively so (the computation is easily parallelized, for instance).

Neutral Transformations

Semantics preserving transformations that are not reductions are not useful on their own - they can only have value when followed by additional transformations in some other class. They are thus more speculative than reductions, and more costly to consider. I will refer to these as *strongly neutral transformations*. Using strongly neutral transformations in evolutionary search was first proposed by Roland Olsson (Ols95)).

- **Abstraction** - given an expression E containing non-overlapping subexpressions E_1, E_2, $\ldots E_N$, let E' be E with all E_i replaced with the unbound variables v_i $(1 \leq i \leq N)$. Define the function $f(v_1, v_2, \ldots v_3) = E'$, and replace E with $f(E_1, E_2, \ldots E_N)$.

  Abstraction is a distinct from compressive abstraction in that only a single call to the synthesized function f is introduced, whereas in compressive abstraction there will be at least two (so that the number of symbols will not be increased).

- **Inverse abstraction** - replace a call to a user-defined function with the body of the function, with arguments instantiated (note that this can also be used to partially invert a compressive abstraction).
- **Distribution** - let E be a call to some function f, and let E' be a subexpression of E's ith argument that is a call to some function g, such that f is distributive over g's arguments, or a subset thereof. We shall refer to the actual arguments to g in these positions in E' as $x_1, x_2, \ldots x_n$. Now, consider the function $D(F)$ that is obtained by evaluating E with its ith argument (the one containing E') replaced with the expressions F. Distribution is then defined as replacing E with E' after replacing each x_j $(1 \leq j \leq n)$ with $D(x_j)$.

  An example should be much easier to follow. Consider the expression

```
plus( x, times( y, ifThenElse( cond,
                               a, b ) ) ) .
```

  Since both *plus* and *times* are distributive over the result branches of $ifThenElse$, there are two possible distribution transformations, leading to the expressions

```
ifThenElse( cond,
            plus( x, times( y, a ) ),
            plus( x, times( y, b ) ) )
```

  and

```
plus( x ( ifThenElse( cond,
                      times( y, a ),
                      times( y, b ) ) ) ).
```

- **Inverse distribution** - the opposite of distribution. This is *nearly* a reduction; the exceptions are expressions such as $f(g(x))$, where f and g are mutually distributive.
- **Arity broadening** - given a function f, modify it to take an additional argument of some type. All calls to f must be correspondingly broadened to pass it an additional argument of the appropriate type.
- **List broadening**[3] - given a function f with some ith argument, x, of type T, modify f such that x is of type $list_T$ instead. This necessitates splitting on x into e.g. $y : ys$, and replacing all references to x in the definition of f with references to y. Furthermore, all calls to f with $x = x'$ must have x' replaced with $list(x')$.
- **Conditional insertion** - an expression x is replaced by $ifThenElse(true, x, y)$, where y is some expression of the same type of x.

As a technical note, action result expressions, which may cause side-effects, complicate neutral transformations somewhat. Specifically, abstractions and compressive abstractions must take their arguments lazily (i.e. not evaluate them before the function call itself is evaluated), in order for neutrality to be obtained. Furthermore, distribution and inverse distribution may only be applied when f has no side-effects that will vary (e.g. be duplicated or halved) in the new expression, or affect the nested computation (e.g. change the result of a conditional). Another way to think about this issue is to consider the action result type as a lazy domain-specific language embedded within a pure functional language (where evaluation order is unspecified). An emperical study of the tradeoffs involved lazy vs. eager function abstraction for program evolution may be found in (Spe96).

The number of possible neutral transformation that may be performed on any given program grows very quickly with program size - exact calculations may be found in (Ols95). Furthermore, synthesis of complex programs and abstractions does not seem to be possible without them. Accordingly, a key hypothesis of any approach to AGI requiring significant feats of program synthesis, without assuming the currently infeasible computational capacities required to brute-force the problem, must be that the *inductive bias* necessary to select promising neutral transformation can be learned and/or preprogrammed.

Non-Neutral Transformations

Non-neutral transformations are the general class defined by removal, replacement, and insertion of subexpressions, acting on expressions in normal form, and preserving the normal form property. Clearly these transformations are sufficient to convert any normal form expression into any other. What is desired is a subclass of the non-neutral transformations that is combinatorially complete, where each individual transformation is nonetheless a semantically small step.

For Boolean expressions, the full set of transformations is given in (Loo06). For numerical expressions, the transcendental functions sin, log, and e^x, not present in the normal form are used to construct transformations. These obviate the need for division (since $a/b = e^{log(a)-log(b)}$), and of course $a - b = a + -1 * b$. For lists, transformations are based on insertion of new leaves (e.g. to append function calls), and "deepening" of the normal form by insertion of subclauses (see (Loo06) for details). For tuples, it is simply the union of the transformations of all the subtypes (of course for other mixed-type expressions the union of the non-neutral transformations for all types must be considered as well. For enum types the transformations are simply replacing one symbol with another. For function types, the transformations are based on function composition. For action result types, actions are inserted/removed/altered, analogous to the insertion/removal/alteration of Boolean literals in the Boolean type.

We propose an additional class of non-neutral transformations across type based on the marvelous $fold$ function:

[3] Analogous tuple-broadening transformations may be defined as well, but are omitted for brevity.

$$fold(f, v, l) =$$
$$ifTheElse(empty(l), v,$$
$$f(first(l), fold(f, v, rest(l))))$$

The fold function allows us to express a wide variety of iterative constructs, while guaranteeing termination, and biasing us towards low computational complexity. In fact, fold allows us to represent exactly the primitive recursive functions (Hut99).

Even considering only this reduced space of possible transformations, in many cases we will find that there are still to many possible programs "nearby" some target to effectively consider all of them. For example many probabilistic model-building algorithms, such as learning the structure of a Bayesian network from data, may require time cubic in the number of variables (in this context each independent non-neutral transformation can correspond to a variable). Especially as the size of the programs we wish to learn grows, and as the number of typologically matching functions increases, there will be simply too many variables to consider each one intensively, let alone apply a quadratic-time algorithm.

To alleviate this scaling difficult, we propose two techniques. The first is to consider each potential variable (i.e. independent non-neutral transformations) in order to heuristically determine its usefulness in expressing constructive semantic variation. For example, a Boolean transformation that collapses the overall expression into a tautology is assumed to be not very useful.̃footnoteThis is a heuristic because such a transformation *might* turn out to be useful when applied in conjunction with other transformations. The second technique is heuristic coupling rules that allow us to calculate, for a pair of transformations, the expected utility or disutility of applying them in conjunction.

Conclusions

In this paper, the system of normal forms begun in (Loo06) has been extended to encompass a full programming language. An extended taxonomy of programmatic transformations has been proposed to aid in learning and reasoning about programs. In the future, we would like to experimentally validate that these normal forms and heuristic transformations *do* in fact increase the syntactic-semantic correlation in program spaces, as has been shown so far only in the Boolean case. We would also like to incorporate these normal forms and transformation into a program evolution system, such as meta-optimizing semantic evolutionary search (Loo07a), and apply them as constraints on probabilistic inference on programs.

References

E. B. Baum. *What is Thought?* MIT Press, 2004.

E. Baum. A working hypothesis for general intelligence. In *Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms*, 2006.

S. Gustafson, E. K. Burke, and G. Kendall. Sampling of unique structures and behaviours in genetic programming. In *European Conference on Genetic Programming*, 2004.

B. Goertzel. The pleasure algorithm. *groups.google.com/group/opencog/files*, 2008.

C. Holman. *Elements of an Expert System for Determining the Satisfiability of General Boolean Expressions.* PhD thesis, Northwestern University, 1990.

G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 1999.

M. Hutter. Universal algorithmic intelligence: A mathematical top-down approach. In B. Goertzel and C. Pennachin, editors, *Artificial General Intelligence.* Springer-Verlag, 2005.

L. Levin. Randomness and nondeterminism. In *The International Congress of Mathematicians*, 1994.

M. Looks, B. Goertzel, and C. Pennachin. Novamente: An integrative architecture for artificial general intelligence. In *AAAI Fall Symposium Series*, 2004.

M. Looks. *Competent Program Evolution.* PhD thesis, Washington University in St. Louis, 2006.

M. Looks. Meta-optimizing semantic evolutionary search. In *Genetic and evolutionary computation conference*, 2007.

M. Looks. On the behavioral diversity of random programs. In *Genetic and evolutionary computation conference*, 2007.

M. Looks. Scalable estimation-of-distribution program evolution. In *Genetic and evolutionary computation conference*, 2007.

M. Looks. Moses wiki. *code.google.com/p/moses/wiki*, 2008.

W. B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

J. R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 1995.

R. Solomonoff. A formal theory of inductive inference. *Information and Control*, 1964.

L. Spector. Simultaneous evolution of programs and their control structures. In *Advances in Genetic Programming 2.* MIT Press, 1996.

M. Tomassini, L. Vanneschi, P. Collard, and M. Clergue. A study of fitness distance correlation as a difficulty measure in genetic programming. *Evolutionary Computation*, 2005.

P. Wang. *Rigid Flexibility: The Logic of Intelligence.* Springer, 2006.

I. Wegener. *The Complexity of Boolean Functions.* John Wiley and Sons, 1987.