# Program Representation for General Intelligence

**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

What are 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, compared to unrestricted Boolean formulae. *Combinatorial:* programs 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 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 is systems 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*. Alternatives such as recurrent neural networks have serious downsides, however, including opacity and inefficiency.

Note 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 are 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 is 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 is substantiated the next two sections, where we give a conceptual formulation of what we mean by *tractable program representations*, and introduce tools for formalizing tractability. The fourth section of the paper proposes an approach for tractably representing programs. The fifth 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:

- **Open-endedness** – in contrast to other knowledge representations current in machine learning, programs vary in size and "shape", and there is no obvious problem-independent upper bound on program size. This makes it difficult to represent programs as points in a fixed-dimensional space, or to 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).

[1] As well as a language for proofs in the case of AIXI$tl$.
[2] The universal distribution converges quickly (Sol64).

Lacking prior knowledge, many algorithms will inefficiently sample semantically identical programs repeatedly (GBK04; Loo07b).

- **Chaotic Execution** – programs that are very similar, syntactically, may be very different, semantically. This presents difficulties for many heuristic search algorithms, which require syntactic and semantic distance to be correlated (TVCC05; Loo07c).
- **High resource-variance** – programs in the same space 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 object 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 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.

In advocating that these challenges be addressed through "better representations", we do not mean merely trading one Turing-complete programming language for another; in the end it will all come to the same. Rather, 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 on 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.

## What Makes a Representation Tractable?

Creating a comprehensive formalization of the notion of a *tractable program representation* would constitute a significant achievement; and we will not fulfill that summons here. We will, however, take a step in that direction by enunciating a set of positive principles for tractable program representations, corresponding closely to the list of representational challenges above. While the discussion in this section is essentially conceptual rather than formal, we will use a bit of notation to ensure clarity of expression; $S$ to denote a space of programmatic functions of the same type (e.g. all pure *Lisp* $\lambda$-expressions mapping from lists to numbers), and $B$ to denote a metric space of *behaviors*.

In the case of a deterministic, side-effect-free program, execution maps from programs in $S$ to points in $B$, which will have separate dimensions for function outputs across various inputs of interest, as well as dimensions corresponding to the time and space costs of executing the program. In the case of a program that interacts with an external environment, or is intrinsically nondeterministic, execution will map from $S$ to probability distributions over points in $B$, which will contain additional dimensions for any side-effects of interest that programs in $S$ might have. Note the distinction between *syntactic distance*, measured as e.g. tree-edit distance between programs in $S$, and *semantic distance*, measured between programs' corresponding points in or probability distributions over $B$. We assume that semantic distance accurately quantifies our preferences in terms of a weighting on the dimensions of $B$; i.e., if variation along some axis is of great interest, our metric for semantic distance should reflect this.

Let $\mathcal{P}$ be a probability distribution over $B$ that describes our knowledge of what sorts of problems we expect to encounter, and let $R(n) \subseteq S$ be the set of all of the programs in our representation with (syntactic) size no greater than $n$. We will say that "$R(n)$ $d$-covers the pair $(B, \mathcal{P})$ to extent $p$" if $p$ is the probability that, for a random behavior $b \in B$ chosen according to $\mathcal{P}$, there is some program in $R$ whose behavior is within semantic distance $d$ of $b$. Then, some among the various properties of tractability that seem important based on the above discussion are as follows:

- for fixed $d$, $p$ quickly goes to 1 as $n$ increases,
- for fixed $p$, $d$ quickly goes to 0 as $n$ increases,
- for fixed $d$ and $p$, the minimal $n$ needed for $R(n)$ to $d$-cover $(B, \mathcal{P})$ to extent $p$ should be as small as possible,
- ceteris paribus, syntactic and semantic distance (measured according to $\mathcal{P}$) are highly correlated.

Since execution time and memory usage measures may be incorporated into the definition of program behavior, minimizing chaotic execution and managing resource variance emerges conceptually here as a subcase of maximizing correlation between syntactic and semantic distance. Minimizing over-representation follows from the desire for small $n$: roughly speaking the less over-representation there is, the smaller average program size can be achieved.

In some cases one can empirically demonstrate the tractability of representations without any special assumptions about $\mathcal{P}$: for example in prior work we have shown that adoption of an appropriate hierarchical normal form can generically increase correlation between syntactic and semantic distance in the space of Boolean functions (Loo06; Loo07c). In this case we may say that we have a *generically tractable* representation. However, to achieve tractable representation of more complex programs, some fairly strong assumptions about $\mathcal{P}$ will be necessary. This should not be philosophically disturbing, since it's clear that human intelligence has evolved in a manner strongly conditioned by certain classes of environments; and similarly, what we need to do to create a viable program representation system for pragmatic AGI usage is to achieve tractability relative to the

distribution $\mathcal{P}$ corresponding to the actual problems the AGI is going to need to solve. Formalizing the distributions $\mathcal{P}$ of real-world interest is a difficult problem, and one we will not address here. However, we hypothesize that the representations presented in the following section may be tractable to a significant extent irrespective[3] of $\mathcal{P}$, and even more powerfully tractable with respect to this as-yet unformalized distribution. As weak evidence in favor of this hypothesis, we note that many of the representations presented have proved useful so far in various narrow problem-solving situations.

## (Postulated) Tractable Representations

We use a simple type system to distinguish between the various normal forms introduced below. This is necessary 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 and including 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, and 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 list arguments that may be empty.

### Normal Forms

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

- list types, $list_T$, where $T$ is any type,
- tuple types, $tuple_{T_1,T_2,\ldots T_N}$, where all $T_i$ are types, 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,
- action result types.

A list of type $list_T$ is an ordered sequence of 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 $i$th 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 a type $T$ is a set of elementary functions with codomain $T$, a set of constants of type $T$, and a tree grammar. Internal nodes for expressions described by the grammar are elementary functions, and leaves are either $U_{var}$ or $U_{constant}$, where $U$ is some type (often $U = T$).

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

- $T_{constant}$ leaves are replaced with constants of type $T$,
- $T_{var}$ leaves 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 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. `{x | y}{2,}` is two or more items in sequences where each item is either an `x` or a `y`).

**Number Normal Form** The elementary functions are $times$ and $plus$. The constants are some subset of the rationals (e.g. those with IEEE single-precision floating-point representations). The grammar is:

```
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 grammar is:

```
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 is 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 $s_i$s. The normal form is either a constant or a var.

[3] Technically, with only weak biases that prefer smaller and faster programs with hierarchical decompositions.

**Function Normal Form** For $T_1, T_2, \ldots T_N \rightarrow O$, the normal form 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 $T_i$s 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 $T_i$s are list types, then the body is a call to the *split* function, with all arguments in normal form.

*Split* is a family of functions with type signatures

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

To evaluate $split(f,\ tuple(l_1, o_1),\ tuple(l_2, o_2),\ \ldots\ tuple(l_k, o_k))$, 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 deconstruct each of them into $x_i\ :\ xs_i$, where $x_i$ is the first element of the list and $xs_i$ is the rest. 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 only if they are nonempty.

**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 worlds are two special sorts of functions.

- **Perceptions** - functions that take a $world$ as their first argument and regular (non-world and non-action-result) types as their remaining arguments, and return regular types. Unlike other function types, the result of evaluating a perception call may be different at different times.
- **Actions** - functions that take a $world$ as their first argument and regular 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 $success$ and $failure$, indicating absolute success/failure in execution.[4] 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,} ) .
```

## Program Transformations

A program transformation is any type-preserving mapping from expressions to expressions. Transformations may be guaranteed to preserve semantics. When doing program evolution there is an intermediate category of fitness preserving transformations that may 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** These are semantics preserving transformations that do not increase some size measure (typically number of symbols), and are idempotent. For example, $and(x, x, y) \rightarrow and(x, y)$ is a reduction for the $Boolean$ type. A set of *canonical reductions* is defined for every type with a normal form. For the $number$ type, the simplifier in a computer algebra system may be used. The full list of reductions is omitted in this paper for brevity. An expression is *reduced* if it maps to itself under all canonical reductions for its type, and all of its subexpressions are reduced.

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

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

which contains 19 symbols. Transforming this to

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

reduces the total number of symbols to 15. One can generalize this notion to consider compressive abstractions across a set of programs. Compressive abstractions appear to be rather expensive to uncover, although perhaps not prohibitively so (the computation is easily parallelized).

**Neutral Transformations** Semantics preserving transformations that are not reductions are not useful on their own - they can only have value when followed by transformations from some other class. This class of transformations is thus more speculative than reductions, and more costly to consider - cf. (Ols95).

- **Abstraction** - given an expression $E$ containing non-overlapping subexpressions $E_1$, $E_2$, $\ldots E_N$, let $E'$ be $E$

[4] 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 $and_{seq}(or_{seq}(arg_1, success), or_{seq}(arg_2, success), \ldots\ or_{seq}(arg_N, success))$.

with all $E_i$ replaced by the unbound variables $v_i$. 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 distinct from compressive abstraction because only a single call to the new function $f$ is introduced.[5]

- **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 $i$th 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, let $D(F)$ by the function that is obtained by evaluating $E$ with its $i$th argument (the one containing $E'$) replaced with the expression $F$. Distribution is replacing $E$ with $E'$, and then replacing each $x_j$ $(1 \leq j \leq n)$ with $D(x_j)$. For example, consider

```
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, giving the expressions

```
ifThenElse( cond,
            plus( x, times( y, a ) ),
            plus( x, times( y, b ) ) ),
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**[6] - given a function $f$ with some $i$th argument $x$ of type $T$, modify $f$ to instead take an argument $y$ of type $list_T$, which gets split into $x : xs$. All calls to $f$ with $i$th argument $x'$ must be replaced by corresponding calls with $i$th argument $list(x')$.
- **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. Specifically, abstractions and compressive abstractions must take their arguments lazily (i.e. not evaluate them before the function call itself is evaluated), in order to be neutral. 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). Spector has performed an empirical study of the tradeoffs in lazy vs. eager function abstraction for program evolution (Spe96).

The number of neutral transformation applicable to any given program grows quickly with program size.[7] Furthermore, synthesis of complex programs and abstractions does not seem to be possible without them. Thus, a key hypothesis of any approach to AGI requiring significant program synthesis, without assuming the currently infeasible computational capacities required to brute-force the problem, is that the inductive bias to select promising neutral transformations can be learned and/or programmed. Referring back to the initial discussion of what constitutes a tractable representation, we speculate that perhaps, whereas well-chosen reductions are valuable for generically increasing program representation tractability, well-chosen neutral transformations will be valuable for increasing program representation tractability relative to distributions $\mathcal{P}$ to which the transformations have some (possibly subtle) relationship.

**Non-Neutral Transformations** Non-neutral transformations may encompass 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 subset of these transformations that is combinatorially complete, where each individual transformation is nonetheless a semantically small step.

The full set of transformations for Boolean expressions is given in (Loo06). For numerical expressions, the transcendental functions $sin$, $log$, and $e^x$ are used to construct transformations. These obviate the need for division ($a/b = e^{log(a)-log(b)}$), and subtraction ($a - b = a + -1 * b$). 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, we take the union of the transformations of all the subtypes. For other mixed-type expressions the union of the non-neutral transformations for all types must be considered as well. For enum types the only transformation is replacing one symbol with another. For function types, the transformations are based on function composition. For action result types, actions are inserted/removed/altered, akin to the treatment of Boolean literals for the Boolean type.

We propose an additional set of non-neutral transformations based on the marvelous *fold* function:

$$\begin{aligned} &\textit{fold}(f, v, l) = \\ &ifThenElse(\textit{empty}(l), v, \\ &\qquad f(\textit{first}(l), \textit{fold}(f, v, rest(l)))) \ . \end{aligned}$$

With *fold* we can express a wide variety of iterative con-

[5] In compressive abstraction there must be at least two calls in order to avoid increasing the number of symbols.

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

[7] Exact calculations are given by Olsson (Ols95).

structs, with guaranteed termination and a bias 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 there are still too 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, can 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 cubic-time algorithm.

To alleviate this scaling difficulty, we propose three techniques. The first is to consider each potential variable (i.e. independent non-neutral transformation) 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 useless.[8] The second is heuristic coupling rules that allow us to calculate, for a pair of transformations, the expected utility of applying them in conjunction. Finally, while *fold* is powerful, it may need to be augmented by other methods in order to provide tractable representation of complex programs that would normally be written using numerous variables with diverse scopes. One approach that we have explored involves application of Sinot's ideas about *director strings as combinators* (SMI03). In this approach, special program tree nodes are labeled with director strings, and special algebraic operators interrelate these strings. One then achieves the representational efficiency of local variables with diverse scopes, without needing to do any actual variable management. Reductions and (non-)neutral transformation rules related to broadening and reducing variable scope may then be defined using the director string algebra.

## Conclusions

In this paper, we have articulated general conceptual requirements that should be fulfilled by a program representation scheme if it is to be considered tractable, either generically or with respect to particular probabilistic assumptions about the environments and tasks on which programs will be evaluated. With the intention of addressing these requirements, 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 will 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 explore the extent to which even stronger correlation, and additional tractability properties, can be observed when realistic probabilistic constraints on "natural" environments and task spaces are imposed. Finally, we intend to incorporate these normal forms and transformations into a program evolution system, such as meta-optimizing semantic evolutionary search (Loo07a), and apply them as constraints on probabilistic inference on programs.

[8] This is heuristic because such a transformation might be useful together with other transformations.

## References

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

E. B. 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.

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.

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.

F. R. Sinot, Fernández M., and Mackie I. Efficient reductions with director strings. In *Rewriting Techniques and Applications*, 2003.

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.