Final Report on SGER For:
“A Cognitively Informed Approach to Automatic Programming”*

PI and co-PI: Selmer Bringsjord & Konstantine Arkoudas
Primary GRA: Jinrong Li
URA: Dylan Thies
Supporting GRAs: Andrew Shilliday, Micah Clark, Joshua Taylor
Department of Cognitive Science
Department of Computer Science
Rensselaer Polytechnic Institute (RPI)
Troy NY 12180 USA
selmer@rpi.edu • arkouk@rpi.edu

December 9, 2008

Contents

1 Introduction 1

2 A Survey of the Field of Automatic Programming 2
   2.1 Inductive techniques ................................................. 3
      2.1.1 Recurrence Detection ........................................... 3
      2.1.2 Genetic Programming ............................................ 5
      2.1.3 Inductive Logic Programming ................................. 6
   2.2 Deductive synthesis .................................................. 7

3 Symposia on AP and Automated Discovery 12
   3.1 Automatic Programming Track at NA-CAP 2008 ................. 12
   3.2 Automated Discovery Symposium under AAAI in Washington DC ........................................ 13

4 Investigating Human Creativity in Computer Programming 14

5 Consolidation 16

6 Toward Infusing AP With Elements of Human Creativity 17

References 17

*We are indebted to NSF for allowing us to take the first, promising steps toward our vision of galvanizing automatic programming by (in short) turning in earnest to human creativity for guidance.
1 Introduction

Put simply, automatic programming (AP) is the field devoted to writing computer programs smart enough to write significant computer programs. By and large, AP has not made any impressive strides over the last 30 years. To be sure, the technical problems in this field are exceedingly complex. But in our view there is another significant reason for the meager progress: Approaches to AP have not paid sufficient attention to human creativity and discovery.

With support from NSF’s CreativeIT program, we have initiated a research project that will change this state of affairs. This project has included five efforts:

1. Comprehensive review, analysis, and evaluation of the history of AP.
2. Workshops designed to explore new ideas regarding AP-relevant creativity and discovery, in both the human and machine spheres.
3. An investigation of human creativity in computer programming and related areas, with an eye to developing a human-centric approach to AP.
4. Consolidation of the present state of the art in automatic programming in one new system, with a special emphasis on deductive techniques.
5. Exploration of infusing traditional approaches to AP with elements inspired by the study of human creativity and reasoning.

In what follows, section by section, we summarize our efforts along these five lines. That is, in brief, we define automatic programming, review its history, summarize the workshops we have held, describe our experimental framework for investigating human creativity in programming and intimately related areas, briefly describe our attempt to make progress on AP by reenergizing the traditional deductive approach to AP with innovations inspired by how (suitably trained) humans reason, and offer a few remarks about our under-development theory of human computer programming as a creative process.
2 A Survey of the Field of Automatic Programming

The aims of automatic programming (AP) have fluctuated considerably over the decades, as has been pointed out by Rich & Waters (1988) and others. In the 1950s, the mechanical compilation of Fortran programs into machine code was viewed as “automatic programming.” In the 1960s, a much more ambitious goal was set for the field—the black-box version of AP, whereby a non-executable description of the desired relationship between input and output is provided to the synthesis software, and the latter then emits an executable computer program that realizes the given specification.

There are several degrees of freedom in the above scheme. First, there are several choices concerning the medium in which we express our description:

1. **Natural language**: Ideally, we would be able to simply tell the machine in a natural language such as English what we want the program to do. In practice, this is not feasible currently. One could of course restrict the input language to, say, a controlled subset of English, but such subsets are in fact formally defined.

2. **Formal specification**: The description could be expressed in a high-level declarative formal system, such as first-order logic. It would be assumed that the formal specification is sound and complete with respect to our informal requirements.

3. **Input-output examples**: The “description” could be illustrated by way of input/output pairs. This would be necessarily incomplete, as there will always be infinitely many programs that cover any presented finite set of examples.

Most of the work on AP so far has adopted one of the latter two approaches. Typically, inductive techniques synthesize programs from input-output examples, whereas deductive techniques synthesize programs from formal specifications. In the following two subsections we review the basics of these two approaches to AP.

To make things concrete, we will be using the following example throughout: Synthesize a computable definition for the *last* function, which takes a non-empty list $L$ as input and returns the last element of $L$. We write $\text{nil}$ for the empty list and $\text{cons}(x, L)$ for the list obtained by prepending (“consing”) the element $x$ in front of the list $L$. Thus, the two-element list
with 5 as its first element and 8 as its second is written as

\[ \text{cons}(5, \text{cons}(8, \text{nil})). \]  

(1)

This is the long-hand notation for lists. A more succinct way of writing a list is simply to enumerate its elements, separated by commas and enclosed within square brackets. Thus, e.g., \([5, 8]\) denotes the same list as \([1]\).

### 2.1 Inductive techniques

The following represent three of the most prominent threads of research in the inductive camp of AP:

1. Recurrence detection
2. Genetic programming
3. Inductive logic programming

We briefly survey these three areas in the following three sub-sections:

#### 2.1.1 Recurrence Detection

The seminal work in this area was carried out by Summers (1977). It is one of the most psychologically plausible approaches in the field. The key idea can be summarized as follows (using the last example):

- The procedure begins by analyzing a number of input-output examples:
  
  \[
  \begin{align*}
  \text{[5]} & \rightarrow 5 \\
  \text{[8,2]} & \rightarrow 2 \\
  \text{[7,5,1]} & \rightarrow 1 \\
  \text{[2,13,6,4]} & \rightarrow 4 
  \end{align*}
  \]

- The method then tries to abstract a finite program (set of rewrite rules) that covers all and only the given pairs:
  
  \[
  \begin{align*}
  \text{cons}(x, \text{nil}) & \rightarrow x \\
  \text{cons}(x, \text{cons}(y, \text{nil})) & \rightarrow y \\
  \text{cons}(x, \text{cons}(y, \text{cons}(z, \text{nil})))) & \rightarrow z \\
  \text{cons}(x, \text{cons}(y, \text{cons}(z, \text{cons}(w, \text{nil})))) & \rightarrow w 
  \end{align*}
  \]

This is done essentially by replacing constants with arbitrary variables.
• The rewrite rules are then expressed using selectors \( (hd \text{ and } tl, \text{i.e., car and cdr})\):

\[
\begin{align*}
  l &= cons(x, nil) \to f_1(l) = hd(l) \\
  l &= cons(x, cons(y, nil)) \to f_2(l) = hd(tl(l)) \\
  l &= cons(x, cons(y, cons(z, nil))) \to f_3(l) = hd(tl(tl(l))) \\
  l &= cons(x, cons(y, cons(z, cons(w, nil)))) \to f_4(l) = hd(tl(tl(tl(l))))
\end{align*}
\]

• Finally, the procedure detects regularities in the finite program:

\[
\begin{align*}
  l &= cons(x, nil) \to f_1(l) = hd(l) \\
  l &= cons(x, cons(y, nil)) \to f_2(l) = f_1(tl(l)) \\
  l &= cons(x, cons(y, cons(z, nil))) \to f_3(l) = f_2(tl(l)) \\
  l &= cons(x, cons(y, cons(z, cons(w, nil)))) \to f_4(l) = f_3(tl(l))
\end{align*}
\]

The final induced definition is the following:

\[
\begin{align*}
  l &= cons(x, nil) \to last(l) = x \\
  l &= cons(x, cons(y, l')) \to last(l) = last(cons(y, l'))
\end{align*}
\]

or in more standard (ML-like) notation:

\[
\begin{align*}
  last([x]) &= x \\
  last(x::y::rest) &= last(y::rest)
\end{align*}
\]

The recurrence detection is done purely syntactically, by pattern matching.

The class of recursive functions that can be induced by this technique is rather restricted, for instance, recursive calls cannot be nested, and mutual recursion cannot be accommodated either.

Summer’s ideas have been extended, most notably in the 1980s by Kordatoff, Franová & Partridge (1989) and Wysotzki (1986), who augmented the basic scheme outlined above. Similar techniques have been used for “programming-by-demonstration” systems such as Tinker (Lieberman 1993). Kitzelmann, Schmid & Kaelbling (2006) and others are continuing this line of work, but results so far have been limited.
2.1.2 Genetic Programming

Genetic programming (GP) (Koza 1992) was discovered in the 1980s (although the general idea of genetic algorithms goes back to the 1950s). The main idea of GP is the following:

- Start with a population (say $10^4$) of random computer programs. Typically programs are purely functional, often expressed in pure LISP, and represented as ASTs (abstract syntax trees).
- Assign a fitness value to each program.
- Create a new population by performing “genetic operations” on selected programs, most notably crossover and mutation.

This loop is continued until some program in the current population achieves a satisfactory fitness, or until a maximum number of iterations has been made. An important point is that the programs to be operated on are selected with probability proportional to their fitness.

The most common genetic operations are the following: Usual operations:

- **Mutation** (on one program only): Randomly alter part of a program’s structure.
- **Crossover** (on two programs): Randomly shuffle two parts of the two programs.
- **Reproduction** (on one program): Simply carry over a program unchanged into the new population.

Crossover is the most important of these three operations, and performed most frequently (with the greatest probability).

GP is generally well-suited for optimization and control problems, and for games. Unfortunately, it is too computationally intensive. Evaluating the fitness of programs entails evaluating the programs themselves, which can be very time-consuming. ADATE (Olsson 1998), for instance, one of the most prominent AP systems based based on genetic programming, takes 6.5 days to evolve a program for list intersection. Like the other approaches, genetic AP techniques have not scaled to realistic programs. In addition, and in contrast to inductive logic programming (see next section), genetic programming is very weak on understanding and explanation. Typically
the generated programs are horribly convoluted spaghetti code. (Although one can mitigate that to a certain extent via simplification, and by making program structure and succinctness part of the fitness function.) As a result, genetic programming the least cognitively plausible of all well-known methodologies for AP.

2.1.3 Inductive Logic Programming

Inductive logic programming (ILP) (Muggleton 1992) synthesizes logic (rather than functional) programs. The input to the synthesis process consists of:

1. A background theory $B$, say some facts about lists, e.g.,
   \[ \text{app} (\text{cons}(x,L_1),L_2,\text{cons}(x,L_3)) \Leftarrow \text{app}(L_1,L_2,L_3) \]

2. A set of positive examples $E^+$ (almost always atoms):
   \[ \text{last}([8],8), \text{last}([5,2],2), \ldots \]

3. A set of negative examples $E^-$:
   \[ \neg \text{last}([5,2],5), \ldots \]

The following requirements are imposed\(^1\)

1. $\forall e^- \in E^- . B \not\models e^-$
2. $\neg \forall e^+ \in E^+. B \models e^+$

The output is a hypothesis $h$ such that:

1. $\forall e^+ \in E^+. B \land h \models e^+$
2. $\forall e^- \in E^- . B \land h \not\models e^-$

Of course the conjunction of all the positive examples is a trivial solution, but what we are really after is predictive power—the generated hypothesis should do well on \textit{unseen} data.

The basic algorithm of ILP is to start with a very specific hypothesis and keep generalizing; or, alternatively, to start with a very general hypothesis and keep specializing. Various combinations are also possible.

Many successful ILP systems view induction as the inverse of deduction, and form hypotheses by inverting deductive inference rules. A typical inference rule is absorption:

\(^1\)We write $\Phi \models p$ to indicate that the set of formulas $\Phi$ logically implies the formula $p$.\nil
\[ A \Rightarrow q \quad A, B \Rightarrow p \quad [\text{Absorption}] \]

The conclusion here logically entails the premises.

While ILP has been successful in data mining, in automatic programming the results have been underwhelming. There have been no remarkable programs generated beyond the usual toy examples (list reversal, etc.). In addition, the generated programs are often quite inefficient. In fact, the method itself is inefficient for recursive programs, since testing examples requires running arbitrary code.

### 2.2 Deductive synthesis

In deductive program synthesis, the input is a formal specification of the desired relationship between the input and output, expressed either in first- or higher-order logic, or else in a very high-level specification language; and the output is a program, typically in a functional language, such as the purely functional subset of Lisp, that is guaranteed to terminate and to satisfy the specified relationship. The guarantees are in the form of formal proofs.

Much of the work in this field has been carried out in the context of constructive logic, whereby the program is extracted from a constructive proof asserting the existence of a suitable output (i.e., an output that meets the specification). Nevertheless, it is not strictly necessary to use constructive logic per se, and indeed some of the most seminal work in this vein took place in the backdrop of classical first-order logic. At any rate, the main idea in either case is the same: Given the formal specification and a background theory that axiomatizes the relevant domain (e.g., a theory of lists or trees), we attempt to construct a proof that the desired function satisfies the given specification.

More precisely, let \( S \) denote the given specification:

\[
\forall x : I, y : O . \ S(x, y)
\]  

(2)

where \( x \) and \( y \) range over the input and output domains, respectively, \( I \) and \( O \). In the interest of flexibility, we do not require the specification to be functional. That is, for any given input \( x \), there may be zero, one, or multiple outputs \( y \) that bear the desired relationship to \( x \). Oftentimes the specification \( S(x, y) \) is of the form

\[
\text{Pre}(x) \Rightarrow \text{Post}(x, y),
\]  

(3)
asserting that if the input $x$ satisfies a certain precondition, then the output $y$ is related to $x$ in accordance with some desired postcondition.

The goal is to synthesize a computable definition of a function $f : I \rightarrow O$

for which the following holds:

$$\forall x : I . S(x, f(x)).$$  \hspace{1cm} (4)

In particular, when the specification $S$ is of the form (3), the desired condition can be equivalently rewritten as follows:

$$\forall x : I . \text{Pre}(x) \Rightarrow \text{Post}(x, f(x)).$$  \hspace{1cm} (5)

Let us illustrate with an example. Suppose we want to synthesize a function that returns the last element of a non-empty list. The background theory here includes the inductive definition of lists, whereby every list is either the empty list $\text{nil}$ or the result of prepending an element $x$ to a list $L$ by applying the $\text{cons}$ function to $x$ and $L$:

$$\forall L : \text{List} . L = \text{nil} \lor \exists h, T . L = \text{cons}(h, T)$$  \hspace{1cm} (6)

with the following two stipulations:

$$\forall x, L . \text{nil} \neq \text{cons}(x, L)$$  \hspace{1cm} (7)

and

$$\forall x_1, L_1, x_2, L_2 . \text{cons}(x_1, L_1) = \text{cons}(x_2, L_2) \Rightarrow x_1 = x_2 \land L_1 = L_2.$$  \hspace{1cm} (8)

Axiom (7) states that $\text{nil}$ is distinct from any list of the form $\text{cons}(x, L)$; while axiom (8) states that a necessary condition for two lists of the form $\text{cons}(x_1, L_1)$ and $\text{cons}(x_2, L_2)$ to be identical is that $x_1 = x_2$ and $L_1 = L_2$. The background theory also includes definitions of various useful functions on lists, such as the append function:

$$\text{app}(((), \text{nil}), L) = L;$$  \hspace{1cm} (9)

$$\text{app}(((), \text{cons})(x, L_1), L_2) = \text{cons}(x, \text{app}(((), L_1), L_2));$$  \hspace{1cm} (10)

as well as various useful results in the theory of lists, such as the following:

**Lemma 1:** For all $L_1, L_2, x$, and $y$, $\text{app}(L_1, [x]) = \text{app}(L_2, [y]) \Rightarrow x = y.$
Lemma 2: \( \text{app}((, L_1), \text{cons}(x, L_2)) \neq \text{nil} \).

We write \([x]\) as an abbreviation for the single-element list \(\text{cons}(x, \text{nil})\).

The specification \(S(L, x)\) relating the input \(L\) and the output \(x\) is of the pre- and post-condition form (5), namely:

\[
\forall L, x . \ L \neq \text{nil} \Rightarrow \exists L'. \ L = \text{app}((, L'), [x]).
\]

This simply says that if the input \(L\) is non-empty then the output \(x\) must be such that, for some \(L'\), \(L\) is the result of appending \(L'\) to the one-element list \([x]\). This is just a way of using existential quantification in a non-constructive way to express the condition that \(x\) is the last element of \(L\). The task now is to synthesize a function \(\text{last}\) that returns the last element of a non-empty list in accordance with this specification. That is, we must be able to prove the following version of (5):

\[
\forall L . \ L \neq \text{nil} \Rightarrow \exists L'. \ L = \text{app}((, L'), [\text{last}(L)])]. \tag{11}
\]

Synthesis in this context works by going forward and trying to prove the goal (11) by induction on the structure of the input \(L\). The obvious question is how we can prove this when we don’t have a definition for \(\text{last}\) yet. Nevertheless, pushing ahead with the proof in a backward manner will eventually “solve” various constraints for \(\text{last}\), and the solution will ultimately result in a full definition for \(\text{last}\). At that point, the steps can be reversed and the full inductive proof can go through successfully. This means that at the end of the synthesis process we will also have a full working correctness proof for the constructed program, which is of course a significant advantage of this method.

To see how the process might work more concretely, consider an inductive proof of (11) in more detail. The base case, when \(L = \text{nil}\) goes through vacuously, since the antecedent becomes \(\text{nil} \neq \text{nil}\), i.e., \(\text{false}\). Consider now the inductive step, when \(L\) is of the form \(\text{cons}(h, T)\) for some \(h\) and \(T\). The goal (11) then becomes

\[
\text{cons}(h, T) \neq \text{nil} \Rightarrow \exists L'. \ \text{cons}(h, T) = \text{app}((, L'), [\text{last}((\text{cons}(h, T))])].
\]

Since the antecedent \(\text{cons}(h, T) \neq \text{nil}\) is clearly true (by virtue of (7)), the above goal becomes reduced to

\[
\exists L'. \ \text{cons}(h, T) = \text{app}((, L'), [\text{last}((\text{cons}(h, T))])]. \tag{12}
\]

By (6), this is reduced to the following disjunction:
\[
\text{cons}(h, T) = \text{app}((, \text{nil}), [\text{last}(\text{cons}(h, T))]) \lor \\
\exists h_1, T_1 . \text{cons}(h, T) = \text{app}((, \text{cons})(h_1, T_1), [\text{last}(\text{cons}(h, T))])
\]

or, using some abbreviations,

\[\text{case}_1 \lor \text{case}_2,\] (13)

where

\[\text{case}_1 \equiv \text{cons}(h, T) = \text{app}((, \text{nil}), [\text{last}(\text{cons}(h, T))])\] (14)

and

\[\text{case}_2 \equiv \exists h_1, T_1 . \text{cons}(h, T) = \text{app}((, \text{cons})(h_1, T_1), [\text{last}(\text{cons}(h, T))]).\] (15)

Now by applying the definition of \text{app} in (14), \text{case}_1 becomes reduced to:

\[\text{cons}(h, T) = \text{cons}(\text{last}(\text{cons}(h, T)), \text{nil}),\] (16)

which, by (8), is equivalent to the following:

\[T = \text{nil} \land \text{last}(\text{cons}(h, T)) = h,\] (17)

Therefore:

\[\text{case}_1 \iff T = \text{nil} \land \text{last}(\text{cons}(h, T)) = h,\] (18)

or, using our abbreviation,

\[\text{case}_1 \iff \text{last}([h]) = h.\] (19)

Consider now \text{case}_2. By applying the definition of append, we get

\[\text{case}_2 \iff \exists h_1, T_1 . \text{cons}(h, T) = \text{cons}(h_1, \text{app}((, T)_1, \text{last}(\text{cons}(h, T)))),\] (20)

which, by (8) becomes

\[\text{case}_2 \iff \exists h_1, T_1 . h = h_1 \land T = \text{app}((, T)_1, [\text{last}(\text{cons}(h, T))]).\] (21)

By lemma 2 the above becomes:

\[\text{case}_2 \iff T \neq \text{nil} \land \exists h_1, T_1 . h = h_1 \land T = \text{app}((, T)_1, [\text{last}(\text{cons}(h, T))]).\] (22)

But now the inductive hypothesis applies. Recall that the inductive hypothesis implies:

\[T \neq \text{nil} \Rightarrow [\exists L' . T = \text{app}((, L)', [\text{last}(T)])].\] (23)
Since $T$ is in fact non-empty in (21), we have:

\[
\text{case}_2 \iff T \neq \text{nil} \land \\
\text{[}\exists h_1, T_1 . \, h = h_1 \land T = \text{app)((, T)_1, [\text{last}(\text{cons}(h, T)))]\text{]} \land (24) \\
\text{[}\exists L' . \, T = \text{app}((, L'), [\text{last}(T)])\text{].}
\]

Merging the existential quantifiers gives:

\[
\text{case}_2 \iff T \neq \text{nil} \land \\
\text{[}\exists h_1, T_1, L' . \, h = h_1 \land T = \text{app}((, T)_1, [\text{last}(\text{cons}(h, T))])\text{]} \land (25) \\
\text{[}\exists L' . \, T = \text{app}((, L'), [\text{last}(T)])\text{]},
\]

which, by lemma 1 becomes:

\[
\text{case}_2 \iff T \neq \text{nil} \land \\
\text{[}\exists h_1, T_1, L' . \, h = h_1 \land T = \text{app}((, T)_1, [\text{last}(\text{cons}(h, T))])\text{]} \land (26) \\
\text{[}\exists L' . \, T = \text{app}((, L'), [\text{last}(T)])\text{]} \land \text{last}(\text{cons}(h, T)) = \text{last}(T).
\]

The last identity can be moved outside of the existential quantifiers, resulting in:

\[
\text{case}_2 \iff T \neq \text{nil} \land \\
\text{[}\exists h_1, T_1, L' . \, h = h_1 \land T = \text{app}((, T)_1, [\text{last}(\text{cons}(h, T))])\text{]} \land (27) \\
\text{[}\exists L' . \, T = \text{app}((, L'), [\text{last}(T)])\text{]},
\]

or, equivalently,

\[
T \neq \text{nil} \land \text{last}(\text{cons}(h, T)) = \text{last}(T) \land C
\]

where

\[
C = \exists h_1, T_1, L' . \, h = h_1 \land T = \text{app}((, T)_1, [\text{last}(\text{cons}(h, T))]) \land (29) \\
\text{[}\exists L' . \, T = \text{app}((, L'), [\text{last}(T)])\text{].}
\]

Therefore, the entire proof breaks down to the following disjunction, which now contains an explicit definition of $\text{last}$:

\[
\text{[}T = \text{nil} \land \text{last}(\text{cons}(h, T)) = h\text{]} \lor [T \neq \text{nil} \land \text{last}(\text{cons}(h, T)) = \text{last}(T) \land C].
\]

In a more readable format, the definition extracted from the above can be stated as follows:

\[
\forall h, T . \, T = \text{nil} \implies \text{last}(\text{cons}(h, T)) = h; \quad (30) \\
\forall h, T . \, T \neq \text{nil} \implies \text{last}(\text{cons}(h, T)) = \text{last}(T). \quad (31)
\]

There are several drawbacks to deductive approach:
The approach requires the user to submit a formal specification of the relationship between the inputs and desired outputs. But writing such specifications can often be just as challenging as writing a program to compute the desired function.

The approach depends crucially on the state of the art in theorem proving. Unfortunately, theorem proving is an extremely challenging problem, and while there has been some progress in the field, current capabilities fall well short of what would be required for automated deductive synthesis of realistic programs.

Typically, the generated programs are purely functional and often quite inefficient. In principle, more efficient versions could then be successively obtained by applying suitable program transformations, but in practice this prospect also runs up against the limited capabilities of theorem-proving systems.

Nevertheless, the approach could prove feasible in the setting of interactive theorem proving, provided that the amount of required human guidance could be kept at a minimum. Moreover, a deductive synthesis module might be a useful (perhaps indispensable) component of a larger synthesis system that combines inductive and deductive techniques.

3 Symposia on AP and Automated Discovery

3.1 Automatic Programming Track at NA-CAP 2008

The North American Computing and Philosophy Conference was held July 10–12, 2008 at Indiana University, in Bloomington, Indiana. On the top level of the conference web site, one can find a synopsis of our “Special Session on Automatic Programming and Human Creativity,” which proved to be very productive. The turnout was excellent, with nary a seat to be had, and attendees seemed to find the session stimulating, informative, and challenging (challenging because a number of those in attendance were unaware of the acute mathematical difficulty of automatic programming, and the history of the field, which clearly confirms this level of difficulty from a pragmatic perspective). With respect to the fully funded research program

http://www.ia-cap.org/na-cap08/index.htm
on automatic programming that we hope our SGER springboards us into, because of exposure to, and detailed discussion about, Douglas Hofstadter’s approach to human and machine creativity, through the presentation

- “Creativity vs Classical Computation,” Eric Nichols, Center for Research on Concepts and Cognition, Indiana University, and Alexandre Linhares, Getulio Vargas Foundation. (Nichols presented.)

our sense is that not only inductive and abduction, but also analogy, will need to be, in some form, mechanized, and given to a system able to automatically generate programs upon receiving a description of a function as input. (In our original proposal, we expressed our belief that abduction is a key form of reasoning that agents engage in in order to devise successful computer programs.) As a result of the exposure to Hofstadterian work carried out by Nichols and Linhares, we extended an invitation to the pair to participate in a second venue, to which we now turn.

3.2 Automated Discovery Symposium under AAAI in Washington DC

With considerable assistance from PhD student Andrew Shilliday, Bringsjord submitted a proposal to the Association for the Advancement of AI (AAAI) to run a symposium in AAAI’s annual Fall Symposium series. The proposal was successful, and the symposium, information about which can be obtained at


was held in Washington DC, over three days (November 7–9, 2008). Our SGER grant partially supported the travel expenses of Simon Colton, who gave a keynote talk on the last day regarding what he calls “joined up” reasoning. The basic idea driving joined up reasoning is to strive for systems that integrate specific machine capabilities in order to make a composite system with more power than any of its parts. We are of the view that when humans creatively solve programming challenges they draw upon a broad, heterogeneous array of representation and reasoning schemes, and we find the research of Simon Colton highly suggestive. We expect that our attack on automatic programming will make use of a number of different problem-solving strategies and approaches at once, and look forward to interacting with, and possibly collaborating with, Colton.

There were a number of other fascinating presentations; readers can be view slides on the web site in question. (E.g., Doug Lenat gave a masterful
overview of the history of automated discovery systems in AI.) One of these presentations was given by Arkoudas, who claimed that a key aspect of discovery and creativity in science is what can be called *concept innovation*. Heretofore, discovery systems have been based on *pre-established* concepts. We expect to confirm, empirically, that human programmers of the first rank develop new concepts as they compose code. Figuring out how an AP system can be engineered to creative new concepts from scratch will no doubt be a specific challenge that we shall need to surmount.

4 Investigating Human Creativity in Computer Programming

Our investigation of human creativity in computer programming—carried out, as planned, in the context of education and computer programming—is based on an experimental framework that we now describe briefly. During the course of the SGER we designed this framework after much background reading and study[^1] and have been refining it with help from sample subjects and domain experts. Starting in January, the framework will be used to run numerous subjects, in work overseen by GRA Li and URA Thies. The category of subjects we are most interested in are those who are mature programmers, but have not used Common Lisp or Scheme, and have not solved programming challenges of the sort that constitute the key stimuli in our experiments. It is this class of subjects that we expect will manifest impressive creativity when writing code.

It is important to know, first, that our framework premeditatedly presents problems to students that are more difficult than problems used previously in the teaching of programming by intelligent tutoring systems, and for that matter in the teaching of programming with help from courseware and educational programming languages (e.g., Logo). One of our specific methodological assumptions is that empirical work will provide no insights into how human creativity manages to lead to computer programs beyond what can be produced by those programs produced in AP, unless the level of difficulty of problems given to humans *exceeds what present-day machines are capable of*.

[^1]: Our framework stands on a neo-Piagetian theoretical foundation espoused by Bringsjord (e.g., see Bringsjord, Bringsjord & Noel 1998, Rinella, Bringsjord & Yang 2001), uses verbal protocol methodology (Ericsson & Simon 1984), relies for background reference on a Lisp textbook long-used in Bringsjord’s pedagogy (i.e., Shapiro 1992), takes account of ACT-R-based intelligent tutoring systems developed for teaching Lisp (e.g., see Anderson, Corbett, Koedinger & Pelletier 1995), and reflects a view of computer programming as reasoning, a view adumbrated in (Bringsjord & Li forthcoming).
of handling. This assumption is one we followed in previous, related work (e.g., see the harder problems used in the pre- and post-tests in Rinella et al. 2001).

The programming languages used in our framework include a purely functional fragment of Common Lisp, and Scheme. These are the dominant languages long-used in our lab for building intelligent software.

Our experiments use a uniform specification scheme $S$ for what the program to be devised is to compute. We now sketch the scheme. The scheme consists of an announcement of the following items:

- a background alphabet $A$;
- a grammar $G$ for specifying some set $\Gamma \subseteq A^*$;
- a definition of some function $f$ from a domain $D \subseteq \Gamma$ (or the subset of $\Gamma \times \ldots \times \Gamma$) to some range $R$ that may likewise be a subset of $\Gamma$, or a newly introduced set.

As a simple example, consider the natural numbers $\mathbb{N} = \{0, 1, \ldots, n, \ldots\}$ (which can of course be constructed from an alphabet consisting of the digits), and consider the binary function $e$ which maps natural numbers to $\{0, 1\}$, depending upon whether the given number is even (which should result in 1) or odd (which should result in 0). As an example that is in line with the level of difficulty of those problems meant to produce the sort of human creativity we are capturing and (at least to a degree) mechanizing, consider an alphabet composed of $n$ propositional variables, a grammar that matches the standard one for the propositional (= sentential) calculus, and a function $d$ that maps pairs of formulas in the propositional calculus to $\{0, 1\}$, depending upon whether the second formula is derivable from the first by deductive inference rules in this calculus (0 for “No” and 1 for “Yes”). For example,

$$d(p_1 \land p_2, p_1 \lor p_2) = 1;$$
$$d((p_1 \to p_2) \land \neg p_2, \neg p_1) = 1.$$  

The challenge here, of course, would be to write a predicate `derivable?` that implements $d$. Another example is to challenge the subject to devise an algorithm, and implement it, in order to compute all-inclusive rules of inference (e.g., condensed detachment), given that they are presented in our scheme.

Please note that, under time constraints, and without prior exposure to the languages available, this is a difficult programming challenge, and it is rather extraordinary that some humans, on the spot, can rise to the occasion. How they manage to do so is a mystery we will solve.
The basic structure of the experiment is as follows. Subjects complete basic survey and demographic information. After this, they are presented with a recorded video that functions as a tutorial introducing them to three straightforward instantiations $S_1, S_2, S_3$ of the above specification scheme, and to the creation of code that successfully implements the functions that have been specified in each scheme. They are also given two short documents: a formal description of the language they will be using, and a chart with quick reference to the built-in functions and commands that are covered in the video tutorial. They are also told that someone will be on hand to answer any questions they might have during their session—including questions they might have about the particular coding environment they are using.\textsuperscript{4} Subjects are then presented with a programming challenge (as described above), and they attempt to solve it. All of their keystrokes are recorded, and a video recording of their face and their use of scrap paper is made. Subjects have one hour to complete their work. After they are finished, retrospective verbal protocol analysis is carried out for an additional fifteen minutes; this analysis is itself recorded.

5 Consolidation

We would like to build an AP system that consolidates the present state of the art, and advances it further in two ways:

1. It will integrate deductive and inductive techniques. There is a great deal of synergistic potential between the two approaches, and it is remarkable that so far such synergy has remained completely uncharted territory. We have experience in implementing both types of systems, and we have good reason to believe that combining disparate approaches is likely to pay significant dividends in this particular area.

2. All of the prominent deductive AP systems are based on interactive theorem-proving technology from the 1970s and 1980s. There has been a great deal of progress in theorem proving since then, both in interactive and in automated theorem proving. We intend to leverage that progress, as we have in much of our previous work, and we expect that this will greatly increase the capabilities of deductive program synthesis. In fact we have already implemented a first version of a deductive-synthesis system in Athena (Arkoudas n.d.), and prelimi-

\textsuperscript{4}That environment is \textit{LispWorks}. 

16
nary results are very encouraging. (The detailed synthesis of the last function presented earlier was obtained from that implementation.)

In addition, as we explain in the next section, our work will make a point of taking the data on human programming creativity seriously, in order to formulate a new, rigorous theory of human computer programming as a creative process. Looking to the human sphere in this manner is a notable break from tradition, unfortunately.

6 Toward Infusing AP With Elements of Human Creativity

We are still refining our new theory of computer programming as a creative process (with the goal of infusing AP with this theory, of course), and hope to unveil the first version of it at the upcoming January 2009 CreativeIT meeting in Washington DC. Our theory puts a premium on the use of non-deductive forms of reasoning, especially abductive, inductive, and analogical. From what we can tell thus far, while creative human programmers do reason and problem-solve in a deductive manner, these other three modes of reasoning are at least as important. From where we stand at present, given all that we have learned through our SGER grant, which is knowledge built atop what we knew about mechanized creativity in other domains before inaugurating our attack on AP (e.g., the literary domain: Bringsjord & Ferrucci 2000), one of the key marks of human creativity is that it consists in the harnessing of a heterogeneous but smoothly integrated array of techniques that, hitherto, have been either left aside in AP, or studied each unto themselves in the AP-related fields of AI and cognitive science.

References


URL: http://kryten.mm.rpi.edu/Selmer_clear_prog_Dec17.pdf


