Motivating Application: Repetitive Text Editing

I like programs, but I wish programs would just program themselves since I don’t like programming.

I like <i>programs</i>, but I wish <i>programs</i> would just <i>program</i> themselves since I don’t like <i>programming</i>.
Motivating Application: Repetitive Text Editing

I like programs, but I wish programs would just program themselves since I don’t like programming.

⇒

I like <i>programs</i>, but I wish <i>programs</i> would just <i>program</i> themselves since I don’t like <i>programming</i>.

Goal: Programming by Demonstration

If the user demonstrates italicizing the first occurrence, can we generalize to the remaining?
Motivating Application: Repetitive Text Editing

I like programs, but I wish programs would just program themselves since I don’t like programming.

⇒

I like <i>programs</i>, but I wish <i>programs</i> would just <i>program</i> themselves since I don’t like <i>programming</i>.

Goal: Programming by Demonstration

If the user demonstrates italicizing the first occurrence, can we generalize to the remaining?

Solution: represent task by a program to be learned

1. Move to next occurrence of word with prefix program
2. Insert <i>
3. Move to end of word
4. Insert </i>
Motivating Application: Repetitive Text Editing

I like programs, but I wish programs would just program themselves since I don’t like programming. ⇒ I like <i>programs</i>, but I wish <i>programs</i> would just <i>program</i> themselves since I don’t like <i>programming</i>.

Goal: Programming by Demonstration

If the user demonstrates italicizing the first occurrence, can we generalize to the remaining?

Solution: represent task by a program to be learned

1. Move to next occurrence of word with prefix program
2. Insert <i>
3. Move to end of word
4. Insert </i>

Challenge: learn from very few examples
General Setup

Goal:

\((X_1, Y_1)\)

\[\ldots\]\n
\((X_n, Y_n)\)

Training data
General Setup

Goal:

\[(X_1, Y_1) \quad \cdots \quad (X_n, Y_n) \rightarrow Z \text{ such that } (Z \cdot X_j) = Y_j\]

Training data \hspace{1cm} Consistent program
General Setup

Goal:

\[(X_1, Y_1) \quad \ldots \quad (X_n, Y_n) \implies Z \text{ such that } (Z \ X_j) = Y_j\]

Training data Consistent program

Challenge:

When \( n \) small, many programs consistent with training data.

I like <i>programs</i>, but I wish programs would just program themselves since I don’t like programming.
General Setup

Goal:

\[(X_1, Y_1) \quad \cdots \quad (X_n, Y_n) \implies Z\text{ such that } (Z \ X_j) = Y_j\]

Training data \hspace{1cm} Consistent program

Challenge:

When \( n \) small, many programs consistent with training data.

I like \(<i>\text{programs}</i>\), but I wish programs would just program themselves since I don’t like programming.

Which program to choose?
Key Intuition

One task:

Want to choose a program which is simple (Occam’s razor).

Examples $\implies Z$
Key Intuition

One task:

Want to choose a program which is simple (Occam’s razor).

Examples $\implies Z$

What’s the right complexity metric (prior)?
Key Intuition

One task:

Want to choose a program which is simple (Occam’s razor).

Examples $\Rightarrow Z$

What’s the right complexity metric (prior)? No general answer.
Key Intuition

One task:
Want to choose a program which is simple (Occam’s razor).

Examples $\implies Z$

What’s the right complexity metric (prior)? No general answer.

Multiple tasks:
Task 1 examples $\implies Z_1$

\[ \vdots \]

Task $K$ examples $\implies Z_K$
Key Intuition

One task:
Want to choose a program which is simple (Occam’s razor).

Examples $\implies Z$

What’s the right complexity metric (prior)? No general answer.

Multiple tasks:

Task 1 examples $\implies Z_1$

$\ldots$

Task $K$ examples $\implies Z_K$

Find programs that share common subprograms.
Key Intuition

One task:

Want to choose a program which is simple (Occam’s razor).

Examples $\implies Z$

What’s the right complexity metric (prior)? No general answer.

Multiple tasks:

Task 1 examples $\implies Z_1$

$\cdots$

Task $K$ examples $\implies Z_K$

Find programs that share common subprograms.

• Programs do tend to share common components.
Key Intuition

One task:

Want to choose a program which is simple (Occam’s razor).

Examples \(\implies Z\)

What’s the right complexity metric (prior)? No general answer.

Multiple tasks:

Task 1 examples \(\implies Z_1\)

\[\ldots\]

Task \(K\) examples \(\implies Z_K\)

Find programs that share common subprograms.

- Programs do tend to share common components.
- Penalize joint complexity of all \(K\) programs.
Outline of Proposed Solution

Program representation: What are subprograms?

Combinatory logic

\[
\begin{align*}
C & \\
B & \\
S & \\
\ast & I
\end{align*}
\]
Outline of Proposed Solution

Program representation: What are subprograms?

Combinatory logic

Probabilistic model: Which programs are favorable?

Nonparametric Bayes
Outline of Proposed Solution

Program representation: What are subprograms?

Combinatory logic

Probabilistic model: Which programs are favorable?

Nonparametric Bayes

Statistical inference: How do we search for good programs?

MCMC
Outline of Proposed Solution

Program representation: What are subprograms?

Combinatory logic

Probabilistic model: Which programs are favorable?

Nonparametric Bayes

Statistical inference: How do we search for good programs?

MCMC
Representation: What Language?

Goal: allow sharing of subprograms
Representation: What Language?

Goal: allow sharing of subprograms

Our language:

Combinatory logic [Schönfinkel, 1924]
Representation: What Language?

Goal: allow sharing of subprograms

Our language:

  Combinatory logic [Schönfinkel, 1924]
  + higher-order combinators (new)
  + routing intuition, visual representation (new)
Representation: What Language?

Goal: allow sharing of subprograms

Our language:
Combinatory logic [Schönfinkel, 1924]
+ higher-order combinators (new)
+ routing intuition, visual representation (new)

Properties: no mutation, no variables ⇒ simple semantics
Representation: What Language?

Goal: allow sharing of subprograms

Our language:
Combinatory logic [Schönfinkel, 1924]
+ higher-order combinators (new)
+ routing intuition, visual representation (new)

Properties: no mutation, no variables ⇒ simple semantics

Result:
• Programs are trees
• Subprograms are subtrees
Programs with No Arguments

Example: compute $\text{min}(3, 4)$
Programs with No Arguments

Example: compute \( \text{min}(3, 4) \)

\[
(\text{if} \ (\lt \ 3 \ 4) \ 3 \ 4)
\]
Programs with No Arguments

Example: compute $\min(3, 4)$

$$\text{if } (< \ 3 \ 4) \ 3 \ 4$$
Programs with No Arguments

Example: compute $\text{min}(3, 4)$

$$(\text{if } (< 3 4) 3 4)$$

General:

\[ x, y \Rightarrow \text{result of applying function } x \text{ to argument } y \]
Programs with No Arguments

Example: compute \( \min(3, 4) \)

\[
\text{if } (< 3 \ 4) \ 3 \ 4
\]

General:

\[
 x \ y \Rightarrow \text{result of applying function } x \text{ to argument } y
\]

Arguments are curried
Programs with No Arguments

Example: compute $\min(3, 4)$

$\text{(if } (< 3 4) \ 3 \ 4 \text{)}$  

(if true 3 4)

General:

$x \ y \Rightarrow$ result of applying function $x$ to argument $y$

Arguments are curried
Programs with No Arguments

Example: compute \(\min(3, 4)\)

\[
\begin{align*}
& (\text{if } (\ < \ 3 \ 4) \ 3 \ 4) \\
& \quad \Rightarrow \\
& \quad (\text{if true} \ 3 \ 4) \\
& \quad \Rightarrow \ 3
\end{align*}
\]

General:

\[
\begin{array}{c}
x \\
\downarrow \\
\text{if} \\
\downarrow \\
< \\
\downarrow \\
3 \\
\downarrow \\
\ldots \\
\Rightarrow \\
\text{result of applying function } x \text{ to argument } y
\end{array}
\]

Arguments are curried
Programs with One Argument

Example:  \( x \mapsto x^2 + 1 \)
Programs with One Argument

Example: $x \mapsto x^2 + 1$

\[ \lambda x . \ 1 \]

Lambda calculus
Programs with One Argument

Example: $x \mapsto x^2 + 1$

\[
\lambda x . +
\]

\[
\ast x,
\]

\[
1
\]

Lambda calculus

Combinatory logic

\[
B
\]

\[
S
\]

\[
I
\]
Programs with One Argument

Example: $x \mapsto x^2 + 1$

\[
\lambda x . + \quad \begin{array}{c}
\ast \quad x \quad x \quad 1 \\
\end{array}
\]

Lambda calculus

Combinatory logic

Intuition:

Combinators \{B, C, S, I\} encode placement of arguments
Programs with One Argument

Example: \( x \mapsto x^2 + 1 \)

\[ \lambda x . + \]

\[ * x \]

\[ x + 1 \]

\[ * I \]

\[ B I S B 1 C \]

Lambda calculus

Combinatory logic

Intuition:
Combinators \{B, C, S, I\} encode placement of arguments

Semantics:
\[ x y r \Leftrightarrow (r x y) \]

\[ r \in \{B, C, S, I\} \]
Programs with One Argument

Example: $x \mapsto x^2 + 1$

\[ \lambda x . + \ast x x + \ast I B I S B 1 C \]

Lambda calculus Combinatory logic

Intuition:
Combinators \{B, C, S, I\} encode placement of arguments

Semantics:

\[ (r x y) \iff (r x y) \]

Rules:

\[ (B f g x) = (f (g x)) \]

...
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5

```
+  
  |    
  +   
  |    
B   S   B   I   B   I   *   I
  |    |    |  |    |    |  
  1   5   1   5   1   5   1
```

```
C 5
```

```
C z ⇔
x y
```

route left
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5

 route left

$+$ $*$ $\rightarrow$

$B$ $S$ $B$ $I$ $B$ $I$
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5

```
+  
  |  
  S  
  |  
  B  I  
  |  
  * I  
  
1
```

```
C  
  /  
 z  
  /  
 x  y
route left

x y  
  
B  
  /  
 z  
  /  
 x y
route right
```
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5

- Route left
- Route right
- Route left and right
Programs with One Argument

Example: Apply \( x \mapsto x^2 + 1 \) to 5

\[
\begin{align*}
\mathbf{B} & \quad \mathbf{5} & \quad \mathbf{I} & \quad \mathbf{5} \\
* & \quad \mathbf{I} & \quad 1 & \\
+ & & & \\
\end{align*}
\]

\[
\begin{align*}
\mathbf{C} & \quad z & \quad \leftrightarrow & \quad x & \quad z \\
x & \quad y & & & \quad y \\
\end{align*}
\]

route left

\[
\begin{align*}
\mathbf{B} & \quad z & \quad \leftrightarrow & \quad x & \quad y & \quad z \\
x & \quad y & & & & \quad z \\
\end{align*}
\]

route right

\[
\begin{align*}
\mathbf{S} & \quad z & \quad \leftrightarrow & \quad x & \quad z & \quad y & \quad z \\
x & \quad y & & & & \quad \\
\end{align*}
\]

route left and right
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5

Route left: $\begin{array}{c}
\text{route left}
\end{array}$

Route right: $\begin{array}{c}
\text{route right}
\end{array}$
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5

route left

route right

route left and right

stop
Programs with One Argument

Example: Apply $x \mapsto x^2 + 1$ to 5

\[
\begin{align*}
\text{C} & \quad \iff \\
x & \quad y \\
\text{B} & \quad \iff \\
x & \quad y & z \\
\text{S} & \quad \iff \\
x & \quad y \\
\text{I} & \quad \iff \\
x
\end{align*}
\]

route left

route right

route left and right

stop
Programs with Multiple Arguments

Example: $\ (x, y) \mapsto \min(x, y) \ $
Programs with Multiple Arguments

Example: \((x, y) \mapsto \min(x, y)\)

Classical: first-order combinators \(\{B, C, S, I\}\)

Complete basis, so can implement \(\min\), but cumbersome
Programs with Multiple Arguments

Example: \((x, y) \mapsto \min(x, y)\)

Classical: first-order combinators \(\{B, C, S, I\}\)

Complete basis, so can implement \(\min\), but cumbersome

New: higher-order combinators \(\{B, C, S, I\}^*\)

Infinite basis, but resulting programs are more intuitive

e.g., \(CS\) routes 1st arg. left, 2nd arg. left and right
Programs with Multiple Arguments

Example: \((x, y) \mapsto \min(x, y)\)

Classical: first-order combinators \(\{B, C, S, I\}\)

Complete basis, so can implement \(\min\), but cumbersome

New: higher-order combinators \(\{B, C, S, I\}^*\)

Infinite basis, but resulting programs are more intuitive
e.g., \(\text{CS}\) routes 1st arg. left, 2nd arg. left and right

\[
\text{if } < \\
\text{BB } I \\
\text{SC } I \\
\text{CS}
\]
Programs with Multiple Arguments

Example: \((x, y) \mapsto \min(x, y)\)

Classical: first-order combinators \(\{B, C, S, I\}\)

Complete basis, so can implement \(\min\), but cumbersome

New: higher-order combinators \(\{B, C, S, I\}\)

Infinite basis, but resulting programs are more intuitive

e.g., CS routes 1st arg. left, 2nd arg. left and right

if <

BB I
SC I
CS 3 4
Programs with Multiple Arguments

Example: \((x, y) \mapsto \min(x, y)\)

Classical: first-order combinators \(\{B, C, S, I\}\)

Complete basis, so can implement \(\min\), but cumbersome

New: higher-order combinators \(\{B, C, S, I\}\)^*

Infinite basis, but resulting programs are more intuitive

e.g., \(CS\) routes 1st arg. left, 2nd arg. left and right

if \(<\)

\[
\begin{array}{c}
\text{BB} \\
\text{I} \\
\text{SC} \\
\text{I} \\
\text{CS} \\
\text{3} \\
\text{4}
\end{array}
\]
Programs with Multiple Arguments

Example: \((x, y) \mapsto \min(x, y)\)

Classical: first-order combinators \(\{B, C, S, I\}\)

Complete basis, so can implement \(\min\), but cumbersome

New: higher-order combinators \(\{B, C, S, I\}\)∗

Infinite basis, but resulting programs are more intuitive

\(\) e.g., \(CS\) routes 1st arg. left, 2nd arg. left and right

\(\) if

\(\) 3

\(\) 3

\(\) 4

\(\) \(B\) 3

\(\) \(C\) \(I\)

\(\) \(S\)
Programs with Multiple Arguments

Example: \((x, y) \mapsto \min(x, y)\)

Classical: first-order combinators \{B, C, S, I\}

Complete basis, so can implement \(\min\), but cumbersome

New: higher-order combinators \{B, C, S, I\}*

Infinite basis, but resulting programs are more intuitive

e.g., \(CS\) routes 1st arg. left, 2nd arg. left and right

\[
\text{if } \begin{array}{c}
< 3 \\
B 3 \\
C I \\
S 4 \\
11
\end{array}
\]
Programs with Multiple Arguments

Example: \((x, y) \mapsto \min(x, y)\)

Classical: first-order combinators \(\{B, C, S, I\}\)

Complete basis, so can implement \(\min\), but cumbersome

New: higher-order combinators \(\{B, C, S, I\}\)^*

Infinite basis, but resulting programs are more intuitive
e.g., \(CS\) routes 1st arg. left, 2nd arg. left and right

```
  if
   < 3
     3
     4
     4
```

^*
Using Combinators for Refactoring

\[
\text{min}
\]

\[
\begin{array}{c}
\text{CS} \\
\text{SC} \\
\text{BB} \\
\text{if} <
\end{array}
\]

\[
\text{max}
\]

\[
\begin{array}{c}
\text{CS} \\
\text{SC} \\
\text{BB} \\
\text{if} >
\end{array}
\]
Using Combinators for Refactoring

\[
\begin{align*}
\text{min} & : \quad \text{CS} \\
& \quad \text{SC} \quad \text{I} \\
& \quad \text{BB} \quad \text{I} \quad \text{if} < \quad \text{BB} \quad \text{I} \\
\text{max} & : \quad \text{CS} \\
& \quad \text{SC} \quad \text{I} \\
& \quad \text{BB} \quad \text{I} \quad \text{if} > \quad \text{BB} \quad \text{I}
\end{align*}
\]

No significant sharing of subtrees (subprograms)
Using Combinators for Refactoring

\[
\begin{align*}
&\text{min} & \text{max} \\
&\quad CS & \quad CS \\
&\quad \quad SC & \quad \quad SC \\
&\quad \quad \quad BB & \quad \quad \quad BB \\
&\quad \quad \quad \quad \text{if} & \quad \quad \quad \text{if} \\
&\quad \quad \quad \quad \quad < & \quad \quad \quad \quad \quad >
\end{align*}
\]

No significant sharing of subtrees (subprograms)

Refactored:
Using Combinators for Refactoring

\[
\begin{align*}
\text{min} & : \text{CS} \\
& : \text{SC} \\
& : \text{BB} \\
& : \text{I} \\
\text{if} & : < \\
\text{max} & : \text{CS} \\
& : \text{SC} \\
& : \text{BB} \\
& : \text{I} \\
\text{if} & : >
\end{align*}
\]

No significant sharing of subtrees (subprograms)

Refactored:

\[
\begin{align*}
\text{CCS} & : \text{CSC} \\
& : \text{BBB} \\
& : \text{I} \\
\text{if} & : I \\
\text{CCS} & : \text{CSC} \\
& : \text{BBB} \\
& : \text{I} \\
\text{if} & : I
\end{align*}
\]

Fruitful sharing of subtrees (subprograms)
Summary

Introduced new combinatory logic basis (intuition: routing)
Summary

Introduced new combinatory logic basis (intuition: routing)

Purpose of these combinators:

• Represent multi-argument functions
• Allow refactoring to expose common substructures
Summary

Introduced new combinatory logic basis (intuition: routing)

Purpose of these combinators:
- Represent multi-argument functions
- Allow refactoring to expose common substructures

Achieved uniformity: Every subtree is a subprogram
Outline of Proposed Solution

Program representation: What are subprograms?

Combinatory logic

Probabilistic model: Which programs are favorable?

Nonparametric Bayes

Statistical inference: How do we search for good programs?

MCMC
Probabilistic Context-Free Grammars

\texttt{GenIndep}(t): [returns a combinator of type \( t \)]
Probabilistic Context-Free Grammars

\texttt{GenIndep}(t): [returns a combinator of type } t] \quad \text{With probability } \lambda_0:
Probabilistic Context-Free Grammars

\texttt{GenIndep}(t): \texttt{[returns a combinator of type } t]\\
With probability \( \lambda_0 \):
\hspace{1cm} \text{Return a random primitive combinator (e.g., +, 3, I)}
Probabilistic Context-Free Grammars

\texttt{GenIndep}(t): \text{[returns a combinator of type } t]\]
With probability \( \lambda_0 \):
Return a random primitive combinator (e.g., +, 3, I)
Else:
Choose a type \( s \)
\( x \leftarrow \texttt{GenIndep}(s \rightarrow t) \)
Probabilistic Context-Free Grammars

\texttt{GenIndep}(t): [returns a combinator of type } t] \\
\text{With probability } \lambda_0: \\
\quad \text{Return a random primitive combinator (e.g., } +, 3, \mathbf{I}) \\
\text{Else:} \\
\quad \text{Choose a type } s \\
\quad x \leftarrow \texttt{GenIndep}(s \rightarrow t) \\
\quad y \leftarrow \texttt{GenIndep}(s)
Probabilistic Context-Free Grammars

\textbf{\texttt{GenIndep}}(t): [returns a combinator of type \texttt{t}]
With probability \( \lambda_0 \):
\hspace{1em} Return a random primitive combinator (e.g., +, 3, \texttt{I})
Else:
\hspace{1em} Choose a type \texttt{s}
\hspace{1.5em} \texttt{x} \leftarrow \texttt{GenIndep}(\texttt{s} \to \texttt{t})
\hspace{1.5em} \texttt{y} \leftarrow \texttt{GenIndep}(\texttt{s})
\hspace{1.5em} return \((\texttt{x}, \texttt{y})\)
Probabilistic Context-Free Grammars

\texttt{GenIndep}(t): \text{[returns a combinator of type } t]\]

With probability \( \lambda_0 \):

Return a random primitive combinator (e.g., +, 3, \textbf{I})

Else:

Choose a type \( s \)

\( x \leftarrow \texttt{GenIndep}(s \rightarrow t) \)

\( y \leftarrow \texttt{GenIndep}(s) \)

return \((x, y)\)

Example:

\[ \texttt{GenIndep}(\text{int} \rightarrow \text{int}) \quad \Rightarrow \quad + 1 \]
Probabilistic Context-Free Grammars

\textbf{GenIndep}(t): [returns a combinator of type } t \text{]

With probability } \lambda_0:\
\text{Return a random primitive combinator (e.g., } +, 3, 1\text{)}

Else:
\text{Choose a type } s
\text{ } x ← \text{GenIndep}(s → t)
\text{ } y ← \text{GenIndep}(s)
\text{return } (x, y)

Example:
\text{GenIndep}(\text{int} → \text{int}) = \implies \begin{array}{c}
* \\
- \\
3 \\
1
\end{array}
Probabilistic Context-Free Grammars

\texttt{GenIndep}(t): [returns a combinator of type \( t \)]

With probability \( \lambda_0 \):

Return a random primitive combinator (e.g., +, 3, I)

Else:

Choose a type \( s \)

\( x \leftarrow \text{GenIndep}(s \rightarrow t) \)

\( y \leftarrow \text{GenIndep}(s) \)

return \( (x, y) \)

Example:

\( \text{GenIndep}(\text{int} \rightarrow \text{int}) \Rightarrow * - 3 \ 1 \)

Problem: No encouragement to share subprograms
Adaptor Grammars [Johnson, 2007]

$C_t \leftarrow []$ for each type $t$ [cached list of combinators]
Adaptor Grammars [Johnson, 2007]

$C_t \leftarrow []$ for each type $t$ [cached list of combinators]
(notation: return$^*$ $c$ adds $c$ to $C_t$ and returns $c$)
Adaptor Grammars [Johnson, 2007]

$C_t \leftarrow []$ for each type $t$ [cached list of combinators]
  (notation: return* $c$ adds $c$ to $C_t$ and returns $c$)

**GenCache**($t$): [returns a combinator of type $t$]
  With probability $\frac{\alpha_0 + N_t d}{\alpha_0 + |C_t|}$:
Adaptor Grammars [Johnson, 2007]

\[ C_t \leftarrow \text{[cached list of combinators]} \]

(notation: return\(^\ast\) \(c\) adds \(c\) to \(C_t\) and returns \(c\))

**GENCACHE**(\(t\)): [returns a combinator of type \(t\)]

With probability \(\frac{\alpha_0 + N_t d}{\alpha_0 + |C_t|}\):

With probability \(\lambda_0\):

Return\(^\ast\) a random primitive combinator (e.g., +, 3, I)

Else:

Choose a type \(s\)

\(x \leftarrow \text{GENCACHE}(s \rightarrow t)\)

\(y \leftarrow \text{GENCACHE}(s)\)

Return\(^\ast\) \((x, y)\)

Else:
Adaptor Grammars [Johnson, 2007]

$C_t \leftarrow []$ for each type $t$ [cached list of combinators]
  (notation: return $^*c$ adds $c$ to $C_t$ and returns $c$)

$\text{GenCache}(t)$: [returns a combinator of type $t$]

With probability $\frac{\alpha_0 + N_t d}{\alpha_0 + |C_t|}$:
  With probability $\lambda_0$:
    Return $^*a$ random primitive combinator (e.g., +, 3, I)
  Else:
    Choose a type $s$
    $x \leftarrow \text{GenCache}(s \rightarrow t)$
    $y \leftarrow \text{GenCache}(s)$
    Return $^*(x, y)$
Else:
  Return $^*z \in C_t$ with probability $\frac{M_z - d}{|C_t| - N_t d}$
Adaptor Grammars [Johnson, 2007]

\[ C_t \leftarrow [] \text{ for each type } t \text{ [cached list of combinators]} \]
(notation: return* \( c \) adds \( c \) to \( C_t \) and returns \( c \))

**GenCache**(\( t \)): [returns a combinator of type \( t \)]

With probability \( \frac{\alpha_0 + N_t d}{\alpha_0 + |C_t|} \):

With probability \( \lambda_0 \):

Return* a random primitive combinator (e.g., +, 3, I)

Else:

Choose a type \( s \)

\( x \leftarrow \text{GenCache}(s \rightarrow t) \)

\( y \leftarrow \text{GenCache}(s) \)

Return* \( (x, y) \)

Else:

Return* \( z \in C_t \) with probability \( \frac{M_z - d}{|C_t| - N_t d} \)

**Interpretation of cache** \( C_t \): library of generally useful (unnamed) subroutines which are reused.
Outline of Proposed Solution

Program representation: What are subprograms?

Combinatory logic

Probabilistic model: Which programs are favorable?

Nonparametric Bayes

Statistical inference: How do we search for good programs?

MCMC
Inference via MCMC

User provides tree structure that encodes set of programs $U$
Objective: sample from posterior given program in $U$
Inference via MCMC

User provides tree structure that encodes set of programs $U$
Objective: sample from posterior given program in $U$

Use Metropolis-Hastings
Proposal: sample a random program transformation
Inference via MCMC

User provides tree structure that encodes set of programs $U$
Objective: sample from posterior given program in $U$

Use Metropolis-Hastings
  Proposal: sample a random program transformation

Program transformations maintain invariant that program is correct (likelihood is 1)
Inference via MCMC

User provides tree structure that encodes set of programs $U$.
Objective: sample from posterior given program in $U$.

Use Metropolis-Hastings.

Proposal: sample a random program transformation.

Program transformations maintain invariant that program is correct (likelihood is 1).

Two types of transformations:
1. Switching
2. Refactoring.
Program transformations (MCMC moves)

**Switching**: Change content, preserve empirical semantics
Program transformations (MCMC moves)

Switching: Change content, preserve empirical semantics

Data: \{(2, 8)\}
Program transformations (MCMC moves)

Switching: Change content, preserve empirical semantics

Data: \{(2, 8)\} ∗
+ 2
S

\[x \mapsto x(x + 2)\]
Program transformations (MCMC moves)

Switching: Change content, preserve empirical semantics

Data: \{(2, 8)\} ∗

+ 2

S

\[x \mapsto x(x + 2)\]

⇔

∗

I

S

\[x \mapsto x^3\]
Program transformations (MCMC moves)

Switching: Change content, preserve empirical semantics

Data: \{(2, 8)\} * + 2 S [x \mapsto x(x + 2)] \iff * I S [x \mapsto x^3]

Purpose: change generalization
Program transformations (MCMC moves)

Switching: Change content, preserve empirical semantics

Data: \{(2, 8)\} ∗
+ ... 2)
⇔
∗
∗ I
S
S
[x ↦→ x]
⇔
[x ↦→ x]

Purpose: change generalization

Refactoring: Change form, preserve total semantics
Program transformations (MCMC moves)

Switching: Change content, preserve empirical semantics

Data: \{(2, 8)\} ∗

\[ x \mapsto x(x + 2) \]

Purpose: change generalization

Refactoring: Change form, preserve total semantics

\[ x \mapsto x^3 \]
Program transformations (MCMC moves)

Switching: Change content, preserve empirical semantics

Data: \{(2, 8)\} 
\[
S \ast + 2 \quad \iff \quad S \ast \quad I
\]
\[
[x \mapsto x(x + 2)] \quad \iff \quad [x \mapsto x^3]
\]

Purpose: change generalization

Refactoring: Change form, preserve total semantics

\[
S \ast + 2 \quad \iff \quad BS \ast + 2
\]
\[
[x \mapsto x(x + 2)] \quad \iff \quad [x \mapsto x(x + 2)]
\]
Program transformations (MCMC moves)

**Switching**: Change content, preserve empirical semantics

Data: \( \{(2, 8)\} \)

\[
S \quad \ast \quad + \quad 2 \\
\Rightarrow \\
S \quad \ast \\
\]

\[
x \mapsto x(x + 2)
\]

\[
S \quad \ast \\
\Rightarrow \\
BS \quad 2
\]

\[
x \mapsto x^3
\]

Purpose: change generalization

**Refactoring**: Change form, preserve total semantics

\[
S \quad \ast \quad + \quad 2 \\
\Rightarrow \\
BS \quad 2
\]

\[
x \mapsto x(x + 2)
\]

Purpose: expose different subprograms for sharing
Text Editing Experiments

Setup:

Dataset of [Lau et al., 2003]

\( K = 24 \) tasks

Each task: train on 2–5 examples, test on \( \approx 13 \) examples

10 random trials
Text Editing Experiments

Setup:

Dataset of [Lau et al., 2003]

\( K = 24 \) tasks

Each task: train on 2–5 examples, test on \( \approx 13 \) examples

10 random trials

Example task:

Cardinals 5, Pirates 2.

\[\Downarrow\]

GameScore[ winner 'Cardinals'; loser 'Pirates'; scores [5, 2]].
Experimental Results

- Uniform prior
- Independent prior
- Joint prior
Experimental Results

![Bar Graph]

- **error**
- **Uniform prior**
- **Independent prior**
- **Joint prior**

<table>
<thead>
<tr>
<th></th>
<th>Uniform prior</th>
<th>Independent prior</th>
<th>Joint prior</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Experimental Results

Observations:
- Independent prior is even worse than uniform prior
Experimental Results

Observations:

- Independent prior is even worse than uniform prior
- Joint prior (multi-task learning) is effective
Summary

\[ X \Rightarrow \square \Rightarrow Y \]
Summary

\[ X \Rightarrow \text{program} \Rightarrow Y \]
Summary

\[ X \Rightarrow \text{program} \Rightarrow Y \]

**Key challenge:** learn programs from few examples
Summary

\[ X \Rightarrow \text{program} \Rightarrow Y \]

**Key challenge:** learn programs from few examples

**Main idea:** share subprograms across multiple tasks
Summary

\[ X \Rightarrow \text{program} \Rightarrow Y \]

Key challenge: learn programs from few examples

Main idea: share subprograms across multiple tasks

Tools:
- Combinatory logic: expose subprograms to be shared
- Adaptor grammars: encourage sharing of subprograms
- Metropolis-Hastings: proposals are program transformations