Inductive Program Synthesis
An Introduction

Martin Hofmann
Cognitive Systems Group
University of Bamberg, Germany

8th September 2009
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Motivations</td>
</tr>
<tr>
<td>2</td>
<td>Inductive Program Synthesis</td>
</tr>
<tr>
<td>3</td>
<td>Approaches</td>
</tr>
<tr>
<td>4</td>
<td>IGOR II Algorithm</td>
</tr>
</tbody>
</table>
Contents

1 Motivations

2 Inductive Program Synthesis

3 Approaches

4 IGOR II Algorithm
“We’re all lazy and trying to cheat — and let the computer write some piece of code.”

Lennart Augustsson
Ways to describe a problem:

- Natural language.
- Input/Output examples or exemplary computation traces
  \(\rightarrow\) \textit{inductive program synthesis}.
- Complete and formal specifications
  \(\rightarrow\) \textit{deductive program synthesis}.

Automatically generating whole software systems is too ambitious.

\textit{Semi-automatical construction of modules, functions, or other pieces of programs.}
Ways to describe a problem:

- Natural language.
- Input/Output examples or examplary computation traces
  \(\leadsto\) *inductive program synthesis*.
- Complete and formal specifications
  \(\leadsto\) *deductive program synthesis*.

Automatically generating whole software systems is too ambitious.

*Semi-automatical construction of modules, functions, or other pieces of programs.*
Ways to describe a problem:

- Natural language.
- Input/Output examples or examplary computation traces
  \(\leadsto\) *inductive program synthesis*.
- Complete and formal specifications
  \(\leadsto\) *deductive program synthesis*.

Automatically generating whole software systems is too ambitious.

*Semi*-automatical construction of modules, functions, or other *pieces of programs.*
Contents

1 Motivations

2 Inductive Program Synthesis

3 Approaches

4 IGOR II Algorithm
IPS is the automatic construction of (recursive) programs from incomplete specifications, e.g. input/output (I/O) examples.

**Example:** \texttt{last}

I/O examples

\begin{align*}
\texttt{last} \ [a] &= a \\
\texttt{last} \ [a,b] &= b \\
\texttt{last} \ [a,b,c] &= c \\
\texttt{last} \ [a,b,c,d] &= d
\end{align*}

induced functional program

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

**Note on list syntax**

\begin{itemize}
\item syntactic sugar
\begin{align*}
[1,2,3,4]
\end{align*}
\item desugared infix
\begin{align*}
(1:2:3:4:[])
\end{align*}
\item desugared prefix
\begin{align*}
( ( : ) 1 \\
( ( : ) 2 \\
( ( : ) 3 \\
( ( : ) 4 \\
[] )))
\end{align*}
\end{itemize}
Inductive Program Synthesis (IPS)

IPS is the automatic construction of (recursive) programs from incomplete specifications, e.g. input/output (I/O) examples.

**Example:** `last`

I/O examples

\[
\begin{align*}
\text{last } [a] &= a \\
\text{last } [a,b] &= b \\
\text{last } [a,b,c] &= c \\
\text{last } [a,b,c,d] &= d
\end{align*}
\]

induced functional program

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

**Note on list syntax**

- syntactic sugar
  \[ [1,2,3,4] \]
- desugared infix
  \[(1:2:3:4:[])\]
- desugared prefix
  \[(::) 1
      (::) 2
      (::) 3
      (::) 4
      []]]]]]]
}
Inductive Program Synthesis (IPS)

IPS is search in a class of programs.

Program class determined by:

Primitives
- Basic primitives, usually data constructors
- Background Knowledge, additional, user-provided, problem-dependent primitives
- Subfunctions, automatically constructed, auxiliary primitives

Restriction Bias
Syntactic rules/constraints to a declarative programming language

Result determined by:

Preference (or search) bias
Criteria to choose among syntactically different result programs
Inductive Program Synthesis (IPS)

IPS is search in a class of programs.

Program class determined by:

**Primitives**
- Basic primitives, usually data constructors
- **Background Knowledge**, additional, user-provided, problem-dependent primitives
- **Subfunctions**, automatically constructed, auxiliary primitives

**Restriction Bias**
- Syntactic rules/constraints to a declarative programming language

Result determined by:

**Preference (or search) bias**
- Criteria to choose among syntactically different result programs
Inductive Program Synthesis (IPS)

IPS is search in a class of programs.

Program class determined by:

Primitives
- Basic primitives, usually data constructors
- Background Knowledge, additional, user-provided, problem-dependent primitives
- Subfunctions, automatically constructed, auxiliary primitives

Restriction Bias
Syntactic rules/constraints to a declarative programming language

Result determined by:

Preference (or search) bias
Criteria to choose among syntactically different result programs
**Example:** reverse

**I/O examples**

- reverse \( [\, ] \) = \( [\, ] \)
- reverse \( [a, b] \) = \( [b, a] \)
- reverse \( [a] \) = \( [a] \)
- reverse \( [a, b, c] \) = \( [c, b, a] \)

**induced functional program**

- reverse \( [\, ] \) = \( [\, ] \)
- reverse \( (x:xs) \) = last \( (x:xs) \) : reverse \( init (x:xs) \)

**automatically induced auxiliary functions (renamed)**

- last \( [x] \) = \( x \)
- init \( [a] \) = \( [\, ] \)
- last \( (x:xs) \) = last \( xs \)
- init \( (x:xs) \) = \( x:(init xs) \)
Contents

1  Motivations

2  Inductive Program Synthesis

3  Approaches

4  IGOR II Algorithm
Different Approaches

- **analytical**
  - functional
  - logic
    - IGOR I, IGOR II, THESYS
    - DIALOGS, DIALOGS-II, GOLEM

- **generate & test**
  - systematical
    - MAGIC-HASKELLER
  - evolutionary
    - ADATE
    - FFOIL, PROGOL
Different Approaches

analytical  generate & test
systematical  evolutionary

functional  IGOR I, IGOR II, THESYS
logic       DIALOGS, DIALOGS-II, GOLEM

In recursive functions, the output for one input is computed by means of outputs for “smaller” inputs.

This implies regularities between I/O-examples

Fold regularities into a recursive definition
Different Approaches

<table>
<thead>
<tr>
<th>analytical</th>
<th>generate &amp; test</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGOR I, IGOR II, THESYS</td>
<td></td>
</tr>
<tr>
<td>DIALOGS, DIALOGS-II, GOLEM</td>
<td></td>
</tr>
<tr>
<td>systematical</td>
<td>evolutionary</td>
</tr>
<tr>
<td>MAGIC-HASKELL</td>
<td></td>
</tr>
<tr>
<td>ADATE</td>
<td></td>
</tr>
</tbody>
</table>

Example: `last` (variables renamed)

| `last (d:[])` | `d` |
| `last (c:d:[])` | `d` |
| `last (b:c:d:[])` | `d` |
| `last (a:b:c:d:[])` | `d` |
| `last [x]` | `x` |
| `last (x:xs)` | `last xs` |
Different Approaches

- **Analytical**
  - Functional
    - IGOR I, IGOR II, THESYS
  - Logic
    - DIALOGS, DIALOGS-II, GOLEM

- **Generate & Test**
  - Systematically
    - MAGIC-HASKELL
  - Evolutionary
    - ADATE

**Generate & Test: systematic**
- systemically enumerate all correct programs
- search base restrictions (type information, libraries, etc.)
- I/Os are only used for testing
# Different Approaches

<table>
<thead>
<tr>
<th>Analytical</th>
<th>Generate &amp; Test</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional</strong></td>
<td><strong>Systematically</strong></td>
</tr>
<tr>
<td>IGOR I, IGOR II, THESYS</td>
<td>MAGIC-HASKELLER</td>
</tr>
<tr>
<td><strong>Logic</strong></td>
<td><strong>Evolutionary</strong></td>
</tr>
<tr>
<td>DIALOGS, DIALOGS-II, GOLEM</td>
<td>ADATE</td>
</tr>
</tbody>
</table>

## Generate & Test: Evolutionary

- Programs are individuals of a population
- *Genetic operators* change individual (cross over, mutation, etc.)
- *Survival of the fittest* w.r.t. run time, size, etc.
- I/Os are only used for testing
Contents

1 Motivations

2 Inductive Program Synthesis

3 Approaches

4 IGOR II Algorithm
Basic Idea

• **IGOR II (Kitzelmann):**
  - inspired by Summer’s THESYS, sequel to IGOR I
  - detect recursion by finding regularities in I/Os
  - integrated *search*
  - reimplemented in HASKELL

• **Strengths:**
  - **termination** by construction
  - arbitrary *user defined data types*
  - arbitrary *background knowledge* useable
  - **subfunctions**
  - **complex call relations** (tree recursion, nested recursion)
  - I/Os with *variables*
  - simultaneously induce multiple (mutual) recursive target functions
Basic Idea

IGOR II (Kitzelmann):
- inspired by Summer’s THESYS, sequel to IGOR I
- detect recursion by finding regularities in I/Os
- integrated search
- reimplemention in HASKELL

Strengths:
- **termination** by construction
- arbitrary user defined data types
- arbitrary background knowledge useable
- subfunctions
- complex call relations (tree recursion, nested recursion)
- I/Os with variables
- simultaneously induce multiple (mutual) recursive target functions
Basic Idea

- **IGOR II (Kitzelmann):**
  - inspired by Summer’s THESYS, sequel to IGOR I
  - detect recursion by finding regularities in I/Os
  - integrated *search*
  - reimplementaton in HASKELL

- **Strengths:**
  - **termination** by construction
  - arbitrary **user defined data types**
  - arbitrary **background knowledge** useable
  - **subfunctions**
  - **complex call relations** (tree recursion, nested recursion)
  - I/Os with **variables**
  - simultaneously induce multiple (mutual) recursive target functions
Basic Idea

IGOR II (Kitzelmann):
- inspired by Summer’s THESYS, sequel to IGOR I
- detect recursion by finding regularities in I/Os
- integrated search
- reimplemented in HASKELL

Strengths:
- **termination** by construction
- arbitrary **user defined data types**
- arbitrary **background knowledge** useable
- subfunctions
- complex call relations (tree recursion, nested recursion)
- I/Os with variables
- simultaneously induce multiple (mutual) recursive target functions
Basic Idea

- IGOR II (Kitzelmann):
  - inspired by Summer’s THESYS, sequel to IGOR I
  - detect recursion by finding regularities in I/Os
  - integrated search
  - reimplemention in HASKELL

- Strengths:
  - termination by construction
  - arbitrary user defined data types
  - arbitrary background knowledge useable
  - subfunctions
  - complex call relations (tree recursion, nested recursion)
  - I/Os with variables
  - simultaneously induce multiple (mutual) recursive target functions
Basic Idea

- **IGOR II (Kitzelmann):**
  - inspired by Summer’s THESYS, sequel to IGOR I
  - detect recursion by finding regularities in I/Os
  - integrated *search*
  - reimplement in HASKELL

- **Strengths:**
  - **termination** by construction
  - arbitrary **user defined data types**
  - arbitrary **background knowledge** useable
  - **subfunctions**
  - **complex call relations** (tree recursion, nested recursion)
  - I/Os with **variables**
  - simultaneously induce multiple (mutual) recursive target functions
Basic Idea

IGOR II (Kitzelmann):
- inspired by Summer’s THESYS, sequel to IGOR I
- detect recursion by finding regularities in I/Os
- integrated search
- reimplement in HASKELL

Strengths:
- termination by construction
- arbitrary user defined data types
- arbitrary background knowledge useable
- subfunctions
- complex call relations (tree recursion, nested recursion)
- I/Os with variables
- simultaneously induce multiple (mutual) recursive target functions
Basic Idea

IGOR II (Kitzelmann):
- inspired by Summer’s THESYS, sequel to IGOR I
- detect recursion by finding regularities in I/Os
- integrated search
- reimplemention in HASKELL

Strengths:
- **termination** by construction
- arbitrary user defined data types
- arbitrary background knowledge useable
- subfunctions
- complex call relations (tree recursion, nested recursion)
- I/Os with variables
- simultaneously induce multiple (mutual) recursive target functions
Basic Idea

- **IGOR II (Kitzelmann):**
  - inspired by Summer’s THESYS, sequel to IGOR I
  - detect recursion by finding regularities in I/Os
  - integrated *search*
  - reimplemention in HASKELL

- **Strengths:**
  - *termination* by construction
  - arbitrary *user defined data types*
  - arbitrary *background knowledge* useable
  - *subfunctions*
  - *complex call relations* (tree recursion, nested recursion)
  - I/Os with *variables*
  - simultaneously induce multiple (mutual) recursive target functions
**Input**

- **data type definition, e.g.:**

```
data List a = [] | a : (List a)
```

- the first \( k \) non-recursive equations for target function and background knowledge, e.g.:
  - **target function:**
    ```
    reverse :: (List a) → (List a) 
    reverse [] = []
    reverse (a::[]) = (a::[])
    reverse (a:b::[]) = (b:a::[])
    ```
  - **background knowledge (optional):**
    ```
    snoc :: (List a) → a → (List a) 
    snoc [] x = (x::[])
    snoc (x::[]) y = (x:y::[])
    ```
**Input**

- data type definition, e.g.:
  
  ```
  data List a = [] | a:(List a)
  ```

- the first $k$ non-recursive equations for target function and background knowledge, e.g.:
  
  - target function:
    
    ```
    reverse :: (List a) → (List a)
    reverse [] = []
    reverse (a:[]) = (a:[])
    reverse (a:b:[]) = (b:a:[])
    ```

  - background knowledge (optional):
    
    ```
    snoc :: (List a) → a → (List a)
    snoc [] x = (x:[])
    snoc (x:[]) y = (x:y:[])
    ```
Set of equations modelling the given I/Os

restriction and preference bias

- case distinction by *pattern matching*.
- minimal number of cases is desired
- syntactic constraints, patterns are not allowed to unify

**reverse - solution**

\[
\begin{align*}
\text{reverse} \quad & \quad [] = [] \\
\text{reverse} \quad & \quad (x:xs) = \text{snoc} (\text{reverse} \ x) \ x
\end{align*}
\]
For a given (sub-)set of equations one equations is searched covering all. First hypothesis through anti-unification of examples:

Example equations:

\[
\begin{align*}
\text{reverse } (a:[]) &= (a:[]) \\
\text{reverse } (a:b:[]) &= (b:a:[]) \\
\end{align*}
\]

Initial Hypothesis:

\[
\begin{align*}
\text{reverse } (x:xs) &= (y:ys) \\
\end{align*}
\]
**Process Initial Hypothesis**

**Initial Hypothesis:**

\[
\text{reverse } (x:xs) = (y:ys)
\]

**Problem:**

unbound variables on the right-hand side (rhs) \( y, ys \)

Three (+1) solutions:

1. **Partitioning** of examples
   \( \mapsto \) set of equations, case distinctions.

2. Replace rhs by **program call** (recursive or background knowledge).

3. Replace subterm with unbound variable by a **subfunction**.

4. (Introduce program schema as **higher-order function**)

M. Hofmann (CogSys) Inductive Program Synthesis 08/09/09 21 / 40
Initial Hypothesis:

\[ \text{reverse} \ (x:xs) = (y:ys) \]

Problem:

unbound variables on the right-hand side (rhs) \( y, ys \)

Three (+1) solutions:

1. **Partitioning** of examples
   \( \leadsto \) set of equations, case distinctions.

2. Replace rhs by **program call** (recursive or background knowledge).

3. Replace subterm with unbound variable by a **subfunction**.

4. (Introduce program schema as **higher-order function**)
Partitioning of examples, case distinctions

- anti-unified inputs differ at least at one position w.r.t. constructor
- partitioning of examples w.r.t. constructor at this position

**Example set:**

1. reverse `[]` = `[]`
2. reverse `(a:[])` = `(a:[])`
3. reverse `(a:b:[])` = `(b:a:[])`

**Anti-unified term:**

reverse `x` = `y`

At root position the constructors `[]` and `(:)` occur, resulting partitions:

{1} {2,3}

reverse `[]` = `[]` reverse `(x:xs)` = `(y:ys)`
Program Call

```
reverse [a, b] = [b, a]    snoc [x] y = [x, y]

\{ x ← b, y ← a \}

reverse [a, b] = snoc ??
```

- If one output matches an output of another function \( f \), it can be replaced by a program call to \( f \).
- Constructing the arguments of the call is a new induction problem.
- I/O examples for the new problem are abduced:
  - Inputs stay the same.
  - New outputs are the substituted *inputs* of the matched output.
Program call – Example

Example equations:

\( \text{reverse} \ [a,b] = [b,a] \)

Background Knowledge:

\( \text{snoc} \ [x] y = [x,y] \)

\([b,a]\) matches \([x,y]\) with substitution

\[ \{ x \leftarrow b, y \leftarrow a \} \]

output \([b,a]\) can be replaced:

\( \text{reverse} \ [a,b] = \text{snoc} (\text{fun1} \ [a,b]) (\text{fun2} \ [a,b]) \)

Abduced examples:

\( \text{fun1} \ [a,b] = [b] \)
\( \text{fun2} \ [a,b] = a \)
Subfunctions

Example equations:

\[
\text{reverse } [a] = [a] \\
\text{reverse } [a,b] = [b,a]
\]

Initial Hypothesis:

\[
\text{reverse } (x:xs) = (y:ys)
\]

- Each subterm on the rhs with an unbound variable is replaced by a call to a subfunction.
- Constructing the subfunction is a new induction problem.
- I/Os for the new subfunction will be abduced:
  - Inputs remain
  - New outputs are the corresponding subterms of old outputs.
Subfunctions – Example

Example equations:

\[
\begin{align*}
\text{reverse } [a] &= [a] \\
\text{reverse } [a, b] &= [b, a]
\end{align*}
\]

Initial hypothesis:

\[
\text{reverse } (x:xs) = (y:ys)
\]

replace right-hand side

\[
\text{reverse } (x:xs) = \text{fun1 } (x:xs) : \text{fun2 } (x:xs)
\]

Abduced I/Os for new subfunctions

\[
\begin{align*}
\text{fun1 } [a] &= a \\
\text{fun1 } [a, b] &= b \\
\text{fun2 } [a] &= [] \\
\text{fun2 } [a, b] &= [a]
\end{align*}
\]
Higher-Order Schemes

Exploit Universal Properties of higher-order functions

Fold - a primitive recursion schema

\[
\text{fold} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow ([a] \rightarrow b)
\]

\[
\text{fold } f \ v \ [] = v
\]

\[
\text{fold } f \ v \ (x:xs) = f \ x \ (\text{fold } f \ v \ xs)
\]

Fold - Universal Property

\[
g [] = v
\]

\[
g (x:xs) = f \ x \ (g \ xs) \iff g = \text{fold } f \ v
\]

enables \textit{map - filter - reduce} pattern
Higher-Order Schemes

Exploit Universal Properties of higher-order functions

Fold - a primitive recursion schema

fold :: (a → b → b) → b → ([a] → b)
fold f v [] = v
fold f v (x:xs) = f x (fold f v xs)

Fold - Universal Property

g [] = v
g (x:xs) = f x (g xs) ⇐⇒ g = fold f v

enables map - filter - reduce pattern
Higher-Order Schemes

Exploit Universal Properties of higher-order functions

Fold - a primitive recursion schema

\[
\text{fold} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow ([a] \rightarrow b) \\
\text{fold } f \text{ } v \text{ } [] = v \\
\text{fold } f \text{ } v \text{ } (x:xs) = f \text{ } x \text{ } (\text{fold } f \text{ } v \text{ } xs)
\]

Fold - Universal Property

\[
\text{g } [] = v \\
\text{g } (x:xs) = f \text{ } x \text{ } (\text{g } xs) \iff g = \text{fold } f \text{ } v
\]

enables \textit{map} - \textit{filter} - \textit{reduce} pattern
Fold Examples

map

\[
\text{map} :: (a \rightarrow b) \rightarrow ([a] \rightarrow [b])
\]

\[
\text{map f} = \text{fold} (\lambda x \, xs \rightarrow f \, x : \, xs) \, []
\]

filter

\[
\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow ([a] \rightarrow [a])
\]

\[
\text{filter p} = \text{fold} (\lambda x \, xs \rightarrow \text{if} \, p \, x \, \text{then} \, x : \, xs \, \text{else} \, xs) \, []
\]

reduce

\[
\text{length} :: [a] \rightarrow \text{Int}
\]

\[
\text{length} = \text{fold} (\lambda x \, n \rightarrow 1 + n) \, 0
\]

\[
\text{reverse} :: [a] \rightarrow [a]
\]

\[
\text{reverse} = \text{fold} (\lambda x \, xs \rightarrow xs ++ [x]) \, []
\]
Higher-Order Schemes

Example equations:

\[
\begin{align*}
\text{reverse} & \quad [] = [] \\
\text{reverse} & \quad (a : []) = (a : []) \\
\text{reverse} & \quad (a : b : []) = (b : a : []) \\
\text{reverse} & \quad (a : b : c : []) = (c : b : a : [])
\end{align*}
\]

Universal Property:

\[
\begin{align*}
g & \quad [] = v \\
g & \quad (x : xs) = f \ x \ (g \ xs)
\end{align*}
\]

⇒ Try to satisfy the universal property with given I/O examples.

- if \( g \) is defined for \([\ ]\), its rhs is \( v \)
- abduce I/Os for \( f \), s.t. \( f \ x \ xs = g \ (x : xs) \) for all I/Os of \( g \) with pattern \((x : xs)\)
- solving \( f \) is a new induction problem
Fold – Example

Example Equations:

\[
\text{reverse } [] = [] \\
\text{reverse } (a:[]) = (a:[]) \\
\text{reverse } (a:b:[]) = (b:a:[]) \\
\text{reverse } (a:b:c:[]) = (c:b:a:[]) \\
\]

Check Universal Property:

\[
\text{reverse } [] = v \\
\text{reverse } (x:xs) = f \ x \ (\text{reverse } \ xs) \\
\Rightarrow v = [] \\
f = \text{fun3}
\]

Abduce I/O for subfunction:

\[
\text{fun3 } a \ [ ] = [a] \\
\text{fun3 } a \ [b] = [b,a] \\
\text{fun3 } a \ [c,b] = [c,b,a]
\]

Reformulate problem:

\[
\text{reverse } x = \text{fold } \text{fun3 } [ ] \ x
\]
What is possible today?

Automatically induce and use subprograms

```plaintext
reverse

reverse [] = []
reverse (x:xs) = last (x:xs) : reverse (init (x:xs))

last [x] = x
last (x:xs) = last xs

init [a] = []
init (x:xs) = x:(init xs)
```
What is possible today?

Complex case distinctions

\[
\begin{align*}
\text{sum} \ [ ] &= 0 \\
\text{sum} \ (Z:x) &= \text{sum} \ x \\
\text{sum} \ (S(x):xs) &= S \left( \text{sum} \ (x:xs) \right)
\end{align*}
\]
What is possible today?

Recursion over multiple arguments

\[\begin{align*}
  Z \leq x &= \text{True} \\
  (S \ x) \leq Z &= \text{False} \\
  (S \ x) \leq (S \ y) &= x \leq y
\end{align*}\]
What is possible today?

Mutual recursive functions

odd/even

odd  Z   = False
odd  (S x) = even  x

even  Z   = True
even  (S x) = odd  x
What is possible today?

Complex call relationships

\[
\begin{align*}
\text{qsort} & \quad [] = [] \\
\text{qsort} \ (x:xs) &= \text{qsort} \ ((\text{allLEQ} \ xs) \ ++ \ [x] \ ++ \ (\text{allGT} \ xs))
\end{align*}
\]
Immanent Problems

Intention

- underspecified problem due to incomplete specification
- non-intended functions as solutions of IP algorithm

Scalability

- “interesting” problems are (syntactically) highly complex
- exponential search problem
Problem: Intention

- underspecified problem due to incomplete specification
- \( \sim \) non-intended functions as solutions of IP algorithm

Open question: Does an additional criteria \( C \) exists which completely specifies the target function \( T \), s.t. \( E, B, C \models T \)

Pragmatic solution: Preference Bias, e.g. “choose syntactically smallest program

But: How are intention and preference bias connected?

Second ’But’: Exponential complexity...
Problem: Intention

- underspecified problem due to incomplete specification
- \( \sim \) non-intended functions as solutions of IP algorithm
- Open question: Does an additional criteria \( C \) exists which completely specifies the target function \( T \), s.t. \( E, B, C \models T \)
- Pragmatic solution: Preference Bias, e.g. “choose syntactically smallest program
- But: How are intention and preference bias connected?
- Second ‘But’: Exponential complexity…
Problem: Intention

- underspecified problem due to incomplete specification
- non-intended functions as solutions of IP algorithm
- Open question: Does an additional criteria $C$ exists which completely specifies the target function $T$, s.t. $E, B, C \models T$
- Pragmatic solution: Preference Bias, e.g. “choose syntactically smallest program
- But: How are intention and preference bias connected?
- Second 'But': Exponential complexity…
Problem: Intention

- underspecified problem due to incomplete specification
- ~ non-intended functions as solutions of IP algorithm
- Open question: Does an additional criteria $C$ exists which completely specifies the target function $T$, s.t. $E, B, C |\models T$
- Pragmatic solution: Preference Bias, e.g. “choose syntactically smallest program
- But: How are intention and preference bias connected?
- Second ’But’: Exponential complexity…
Problem: Exponential Complexity

- Finding w.r.t. preference bias best solution is $O(k^n)$ with state-of-the-art IP algorithms

- Open question: Is this necessary? Is IP-problem with w.r.t. preference bias optimal solution NP-complete?

- If so, would sub-optimal solutions be sufficient? Are there suitable heuristics?

- And: If there is an additional general correctness criteria, what about complexity?
Problem: Exponential Complexity

- Finding w.r.t. preference bias best solution is $O(k^n)$ with state-of-the-art IP algorithms
- Open question: Is this necessary? Is IP-problem with w.r.t. preference bias optimal solution NP-complete?
- If so, would sub-optimal solutions be sufficient? Are there suitable heuristics?
- And: If there is an additional general correctness criteria, what about complexity?
Problem: Exponential Complexity

- Finding w.r.t. preference bias **best** solution is $O(k^n)$ with state-of-the-art IP algorithms
- Open question: Is this necessary? Is IP-problem with w.r.t. preference bias optimal solution NP-complete?
- If so, would sub-optimal solutions be sufficient? Are there suitable heuristics?
- And: If there is an additional general correctness criteria, what about complexity?
Finding w.r.t. preference bias *best* solution is $O(k^n)$ with state-of-the-art IP algorithms.

Open question: Is this necessary? Is IP-problem with w.r.t. preference bias optimal solution NP-complete?

If so, would sub-optimal solutions be sufficient? Are there suitable heuristics?

And: If there is an additional general correctness criteria, what about complexity?
Introduction to IP
Systems’ overview
Repository with benchmark problems
IP related publications
Mailing list

http://www.inductive-programming.org
Thank you very much for your attention!
Questions?
Questions?
Questions?
Questions?
Questions?
Questions?
Questions?
Questions?