Inductive Programming
A Unifying Framework for Analysis and Evaluation of Inductive Programming Systems

Hofmann, Kitzelmann, Schmid

Cognitive Systems Group
University of Bamberg

AGI 2009
Inductive Program Synthesis (IP)

Inductive Program Synthesis (IP) researches the automatic construction of (recursive) programs from *incomplete* specifications, i.e. input/output examples (I/O examples)

Example (reverse)

I/O-examples:

- `reverse [] = []`
- `reverse [a] = [a]`
- `reverse [a,b] = [b,a]`
- `reverse [a,b,c] = [c,b,a]`

Induced functional program:

- `reverse [] = []`
- `reverse (x:xs) = (reverse xs) ++ [x]`
Key Concepts

Preference Bias criteria to choose among (semantically different!) candidate solutions, i.e. syntactic size, number of case distinctions, runtime (search strategy).

Restriction Bias Restricts the inducable class of problems, through syntactic constraints, i.e. linear recursion as sole kind of recursion (hypothesize language).

Background Knowledge already implemented functions, which can by used for synthesis, i.e. append and partition for quicksort.

Sub Functions Functions neither defined as target functions nor in the background knowledge, but automatically introduced as auxiliary functions by the IP algorithm.
Key Concepts

Preference Bias criteria to choose among (semantically different!) candidate solutions, i.e. syntactic size, number of case distinctions, runtime (search strategy).

Restriction Bias Restricts the inducable class of problems, through syntactic constraints, i.e. linear recursion as sole kind of recursion (hypotheses language).

Background Knowledge already implemented functions, which can by used for synthesis, i.e. append and partition for quicksort.

Sub Functions Functions neither defined as target functions nor in the background knowledge, but automatically introduced as auxiliary functions by the IP algorithm.
Key Concepts

Preference Bias criteria to choose among (semantically different!) candidate solutions, i.e. syntactic size, number of case distinctions, runtime (search strategy).

Restriction Bias Restricts the inducable class of problems, through syntactic constraints, i.e. linear recursion as sole kind of recursion (hypotheses language).

Background Knowledge already implemented functions, which can by used for synthesis, i.e. append and partition for quicksort.

Sub Functions Functions neither defined as target functions nor in the background knowledge, but automatically introduced as auxiliary functions by the IP algorithm.
Key Concepts

Preference Bias **criteria** to choose among (semantically different!) candidate solutions, i.e. syntactic size, number of case distinctions, runtime (search strategy).

Restriction Bias **Restricts** the inducable class of problems, through syntactic constraints, i.e. linear recursion as sole kind of recursion (hypothese language)

Background Knowledge already **implemented functions**, which can by used for synthesis, i.e. **append** and **partition** for **quicksort**

Sub Functions Functions **neither** defined as **target functions** nor in the **background knowledge**, but automatically introduced as auxiliary functions by the IP algorithm
Different Approaches

<table>
<thead>
<tr>
<th>analytic</th>
<th>generate &amp; test</th>
</tr>
</thead>
<tbody>
<tr>
<td>logic</td>
<td>systematic</td>
</tr>
<tr>
<td></td>
<td>evolutionary</td>
</tr>
<tr>
<td>DIALOGS-II</td>
<td>FOIL/FFOIL,</td>
</tr>
<tr>
<td></td>
<td>GOLEM</td>
</tr>
<tr>
<td>functional</td>
<td>THESYS,</td>
</tr>
<tr>
<td></td>
<td>MAGIC-</td>
</tr>
<tr>
<td></td>
<td>ADATE</td>
</tr>
<tr>
<td></td>
<td>HASKELLER</td>
</tr>
<tr>
<td>IGOR I,</td>
<td></td>
</tr>
<tr>
<td>IGOR II</td>
<td></td>
</tr>
</tbody>
</table>

**Inductive Logic Programming (ILP)**

**Inductive Functional Programming (IFP)**
### Different Approaches

<table>
<thead>
<tr>
<th>Logic</th>
<th>Analytic</th>
<th>Generate &amp; Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIALOGS-II</td>
<td>FOIL/FFOIL, GOLEM</td>
<td></td>
</tr>
<tr>
<td>functional</td>
<td>THESYS, IGOR I, IGOR II</td>
<td>MAGIC-HASKELL, ADATE</td>
</tr>
</tbody>
</table>

**Inductive Logic Programming (ILP)**

- ILP is machine learning with representation and inference based on *Computational Logic* (PROLOG).
- IP as special case of ILP.

**Inductive Functional Programming (IFP)**
## Different Approaches

<table>
<thead>
<tr>
<th>Logic</th>
<th>analytic</th>
<th>generate &amp; test</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIALOGS-II</td>
<td>systematic</td>
<td>evolutionary</td>
</tr>
<tr>
<td>FOIL/F FOIL, GOLEM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>functional</td>
<td>THESYS, IGOR I, IGOR II</td>
<td>MAGIC-HASKELLER, ADATE</td>
</tr>
</tbody>
</table>

### Inductive Logic Programming (ILP)

### Inductive Functional Programming (IFP)
- Based *Term Rewriting or Combinatory Logic / \(\lambda\)-calculus*
- primary objective is program learning
## Different Approaches

<table>
<thead>
<tr>
<th>analytic</th>
<th>generate &amp; test</th>
</tr>
</thead>
<tbody>
<tr>
<td>logic</td>
<td>systematic</td>
</tr>
<tr>
<td></td>
<td>evolutionary</td>
</tr>
<tr>
<td></td>
<td>FOIL/FFOIL,</td>
</tr>
<tr>
<td></td>
<td>GOLEM</td>
</tr>
<tr>
<td>functional</td>
<td>MAGIC-</td>
</tr>
<tr>
<td></td>
<td>HASKELLER</td>
</tr>
<tr>
<td></td>
<td>ADATE</td>
</tr>
</tbody>
</table>

### Analytic

- DIALOGS-II
- THESYS,
  - IGOR I,
  - IGOR II

### Generate & Test

- FOIL/FFOIL
- GOLEM
- MAGIC-HASKELLER
- ADATE
Different Approaches

*analytic*  

**logic**  
DIALOGS-II

**functional**  
THESYS, IGOR I, IGOR II

*generate & test*  

**systematic**  
FOIL/FFOIL, GOLEM

**evolutionary**  
MAGIC-HASKELL, ADATE

---

**Analytic**
- different inputs are "sub problems" of each other
- so their output is included in other outputs as subterms
- analyze I/Os and fold regularities into a recursive definition

---

**Generate & Test**
## Different Approaches

<table>
<thead>
<tr>
<th><strong>logic</strong></th>
<th><strong>functional</strong></th>
<th><strong>generate &amp; test</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>DIALOGS-II</td>
<td>THESYS, IGOR I, IGOR II</td>
<td>FOIL/FFOIL, GOLEM MAGIC-HASKELL, ADATE</td>
</tr>
</tbody>
</table>

### Analytic

**Generate & Test (1): systematic**

- **enumerate** all correct programs systematically
- constraints limit search space (type information, library, modes)
- I/Os are only used as **filter**
Different Approaches

<table>
<thead>
<tr>
<th>Logic</th>
<th>Analytic</th>
<th>Generate &amp; Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIALOGS-II</td>
<td></td>
<td>FOIL/FFOIL, GOLEM</td>
</tr>
<tr>
<td>functional</td>
<td></td>
<td>MAGIC-HASKELL, ADATE</td>
</tr>
</tbody>
</table>

Analytic

Generate & Test (2): evolutionary heuristic

- use **genetic operators** to traverse search space
- **fitness function** maps programs to numeric space
- evaluated program attributes are e.g. runtime, program size, etc.
Different Approaches

<table>
<thead>
<tr>
<th>analytic</th>
<th>generate &amp; test</th>
</tr>
</thead>
<tbody>
<tr>
<td>logic</td>
<td>systematic</td>
</tr>
<tr>
<td>DIALOGS-II</td>
<td>FOIL/FFOIL,</td>
</tr>
<tr>
<td></td>
<td>GOLEM</td>
</tr>
<tr>
<td>functional</td>
<td>evolutionary</td>
</tr>
<tr>
<td>THESYS,</td>
<td>MAGIC-</td>
</tr>
<tr>
<td>IGOR I,</td>
<td>ADATE</td>
</tr>
<tr>
<td>IGOR II</td>
<td>HASKELLER</td>
</tr>
</tbody>
</table>

large diversity of underlying theoretical concepts and requirements
⇒ hard to compare and evaluate
Need for Unifying Framework

Provide system independent syntax and operational semantics

Benefits

+ consistent representation of different target languages
+ gives a unifying (“normalised”) perspective on IP systems
+ helps identifying system specific strength and weaknesses
+ provide a transparent evaluation and comparison of IP systems
+ basis for a general IP algorithm
+ means for an abstract problem definition language (IP Problem Definition Language)

Conditional Constructor (Rewrite) Systems (CCS)
# Need for Unifying Framework

Provide system independent syntax and operational semantics

## Benefits

+ **consistent representation** of different target languages
+ **gives a unifying ("normalised") perspective** on IP systems
+ helps **identifying** system specific strength and weaknesses
+ provide a **transparent evaluation and comparison** of IP systems
+ basis for a **general IP algorithm**
+ means for an abstract problem definition language
  (IP Problem Definition Language)

## Conditional Constructor (Rewrite) Systems (CCS)
Need for Unifying Framework

Provide system independent syntax and operational semantics

Benefits

+ consistent representation of different target languages
+ gives a unifying (“normalised”) perspective on IP systems
+ helps identifying system specific strength and weaknesses
+ provide a transparent evaluation and comparison of IP systems
+ basis for a general IP algorithm
+ means for an abstract problem definition language (IP Problem Definition Language)

Conditional Constructor (Rewrite) Systems (CCS)
Need for Unifying Framework

Provide system independent syntax and operational semantics

Benefits

+ consistent representation of different target languages
+ gives a unifying ("normalised") perspective on IP systems
+ helps identifying system specific strength and weaknesses
+ provide a transparent evaluation and comparison of IP systems
+ basis for a general IP algorithm
+ means for an abstract problem definition language (IP Problem Definition Language)
Need for Unifying Framework

Provide system independent syntax and operational semantics

Benefits

+ consistent **representation** of different target languages
+ gives a **unifying** ("normalised") perspective on IP systems
+ helps **identifying** system specific **strength and weaknesses**
+ provide a **transparent evaluation and comparison** of IP systems
+ basis for a **general IP algorithm**
+ means for an abstract problem definition language (IP Problem Definition Language)

Conditional Constructor (Rewrite) Systems (CCS)
Need for Unifying Framework

Provide system independent syntax and operational semantics

Benefits
+ consistent representation of different target languages
+ gives a unifying ("normalised") perspective on IP systems
+ helps identifying system specific strength and weaknesses
+ provide a transparent evaluation and comparison of IP systems
+ basis for a general IP algorithm
+ means for an abstract problem definition language (IP Problem Definition Language)

Conditional Constructor (Rewrite) Systems (CCS)
Need for Unifying Framework

Provide system independent syntax and operational semantics

Benefits

- consistent representation of different target languages
- gives a unifying ("normalised") perspective on IP systems
- helps identifying system specific strength and weaknesses
- provide a transparent evaluation and comparison of IP systems
- basis for a general IP algorithm
- means for an abstract problem definition language (IP Problem Definition Language)

Conditional Constructor (Rewrite) Systems (CCS)
The paper at one glance

<table>
<thead>
<tr>
<th></th>
<th>$C$</th>
<th>$F_T$</th>
<th>$F_B$</th>
<th>$F_I$</th>
<th>$E^+$</th>
<th>$E^-$</th>
<th>$BK$</th>
<th>$\chi_2$</th>
<th>search strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADATE</td>
<td>●</td>
<td>{·}</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>global search, g ’n t</td>
</tr>
<tr>
<td>FLIP</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>∅</td>
<td>○</td>
<td>○</td>
<td>○,∅</td>
<td>●</td>
<td>sequential covering</td>
</tr>
<tr>
<td>FFOIL</td>
<td>$c$</td>
<td>●</td>
<td>⊃</td>
<td>∅</td>
<td>○</td>
<td>○</td>
<td>○,∅</td>
<td>○</td>
<td>sequential covering</td>
</tr>
<tr>
<td>GOLEM</td>
<td>●</td>
<td>{·}</td>
<td>●</td>
<td>∅</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>sequential covering</td>
</tr>
<tr>
<td>IGOR I</td>
<td>●</td>
<td>{·}</td>
<td>∅</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>2-step, global search</td>
</tr>
<tr>
<td>IGOR II</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>∅</td>
<td>○</td>
<td>○</td>
<td>global search</td>
</tr>
<tr>
<td>MAGH.</td>
<td>●</td>
<td>{·}</td>
<td>●</td>
<td>∅</td>
<td>●</td>
<td>●</td>
<td>⊃</td>
<td>○</td>
<td>breadth first, g ’n t</td>
</tr>
</tbody>
</table>

- ● unrestricted / conditional rules
- {·} singleton set
- $c$ constants
- ○ restricted / unconditional rules
- ∅ empty set
- ⊃ built in predicates
## Empirical Results

<table>
<thead>
<tr>
<th>isort</th>
<th>reverse</th>
<th>weave</th>
<th>shiftr</th>
<th>mult/add</th>
<th>allodds</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADATE</strong></td>
<td>70.0</td>
<td>78.0</td>
<td>80.0</td>
<td>18.81</td>
<td>—</td>
</tr>
<tr>
<td><strong>FLIP</strong></td>
<td>×</td>
<td>—</td>
<td>134.24⊥</td>
<td>448.55⊥</td>
<td>×</td>
</tr>
<tr>
<td><strong>FFOIL</strong></td>
<td>×</td>
<td>—</td>
<td>0.4⊥</td>
<td>&lt; 0.1⊥</td>
<td>8.1⊥</td>
</tr>
<tr>
<td><strong>GOLEM</strong></td>
<td>0.714</td>
<td>—</td>
<td>0.66⊥</td>
<td>0.298</td>
<td>—</td>
</tr>
<tr>
<td><strong>IGOR II</strong></td>
<td>0.105</td>
<td>0.103</td>
<td>0.200</td>
<td>0.127</td>
<td>⊙</td>
</tr>
<tr>
<td><strong>MAGH.</strong></td>
<td>0.01</td>
<td>0.08</td>
<td>⊙</td>
<td>157.32</td>
<td>—</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lasts</th>
<th>last</th>
<th>member</th>
<th>odd/even</th>
<th>mult/last</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADATE</strong></td>
<td>822.0</td>
<td>0.2</td>
<td>2.0</td>
<td>—</td>
</tr>
<tr>
<td><strong>FLIP</strong></td>
<td>×</td>
<td>0.020</td>
<td>17.868</td>
<td>0.130</td>
</tr>
<tr>
<td><strong>FFOIL</strong></td>
<td>0.7⊥</td>
<td>0.1</td>
<td>0.1⊥</td>
<td>&lt; 0.1⊥</td>
</tr>
<tr>
<td><strong>GOLEM</strong></td>
<td>1.062</td>
<td>&lt; 0.001</td>
<td>0.033</td>
<td>—</td>
</tr>
<tr>
<td><strong>IGOR II</strong></td>
<td>5.695</td>
<td>0.007</td>
<td>0.152</td>
<td>0.019</td>
</tr>
<tr>
<td><strong>MAGH.</strong></td>
<td>19.43</td>
<td>0.01</td>
<td>⊙</td>
<td>—</td>
</tr>
</tbody>
</table>

— not tested × stack overflow ⊙ timeout ⊥ wrong
all runtimes in seconds
Our Project

Effiziente Algorithmen zur inductiven Programmsynthese

Content

The Project

Efficient Algorithms for Inductive Program Synthesis

Inductive program synthesis addresses the problem of constructing recursive programs from incomplete specifications, typically input/output examples. Goal of this proposal is the advancement of classical (Summers like) approaches for example-driven, analytical synthesis of functional programs. In contrast to approaches of inductive logic programming and evolutionary computation, which are mainly search-based, analytical approaches have the advantage that synthesis effort is considerably lower.

However, current approaches typically are restricted to structural problems (such as

http://www.cogsys.wiai.uni-bamberg.de/effalip/

- Publications
- Downloads
- Links
inductive-programming.org

http://www.inductive-programming.org

- Introduction to IP
- Systems’ overview
- Repository with benchmark problems
- IP related publications
- Mailing list
- ...
Thank you
for your attention!
Questions?
CCS in a nutshell

- given a set of function symbols $\Sigma$ and a set of variables $\mathcal{X}$
- terms over $\Sigma$ and $\mathcal{X}$ denoted as $\mathcal{T}_\Sigma(\mathcal{X})$
- constructors $\mathcal{C}$ and defined function symbols $\mathcal{F}$
  $\Sigma = \mathcal{F} \cup \mathcal{C}$, $\mathcal{F} \cap \mathcal{C} = \emptyset$
- programs are sets of rewrite rules $lhs \rightarrow rhs$
- $lhs$ is of the form $F(p_1, \ldots, p_n)$ with $F \in \mathcal{F}$ and $p_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$
- conditional rewrite rules $lhs \rightarrow rhs \leftarrow cond$ where
  $cond \equiv \{ v_1 = u_1, \ldots, v_n = u_n \}$ and $v_i, u_i \in \mathcal{T}_\Sigma(\mathcal{X})$
- rewriting binds free variables in $v_i$, modelling variable declaration, let- and case-expressions
- higher-order context with $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ and abstraction operator $[\cdot]$
CCS in a nutshell

- given a set of function symbols $\Sigma$ and a set of variables $\mathcal{X}$
- terms over $\Sigma$ and $\mathcal{X}$ denoted as $\mathcal{T}_\Sigma(\mathcal{X})$
- constructors $\mathcal{C}$ and defined function symbols $\mathcal{F}$
  \[ \Sigma = \mathcal{F} \cup \mathcal{C}, \mathcal{F} \cap \mathcal{C} = \emptyset \]
- programs are sets of rewrite rules $lhs \rightarrow rhs$
- $lhs$ is of the form $F(p_1, \ldots, p_n)$ with $F \in \mathcal{F}$ and $p_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$
- conditional rewrite rules $lhs \rightarrow rhs \leftarrow cond$ where
  \[ cond \equiv \{v_1 = u_1, \ldots, v_n = u_n\} \text{ and } v_i, u_i \in \mathcal{T}_\Sigma(\mathcal{X}) \]
- rewriting binds free variables in $v_i$, modelling variable declaration, let- and case-expressions
- higher-order context with $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ and abstraction operator $[\_]_-$
CCS in a nutshell

- given a set of function symbols $\Sigma$ and a set of variables $\mathcal{X}$
- terms over $\Sigma$ and $\mathcal{X}$ denoted as $\mathcal{T}_\Sigma(\mathcal{X})$
- constructors $C$ and defined function symbols $\mathcal{F}$
  \[ \Sigma = \mathcal{F} \cup C, \mathcal{F} \cap C = \emptyset \]
- programs are sets of rewrite rules $lhs \rightarrow rhs$
  - $lhs$ is of the form $F(p_1, \ldots, p_n)$ with $F \in \mathcal{F}$ and $p_i \in \mathcal{T}_C(\mathcal{X})$
  - conditional rewrite rules $lhs \rightarrow rhs \leftarrow cond$ where
    \[ cond \equiv \{ v_1 = u_1, \ldots, v_n = u_n \} \]
    and $v_i, u_i \in \mathcal{T}_\Sigma(\mathcal{X})$
  - rewriting binds free variables in $v_i$, modelling variable declaration, let- and case-expressions
- higher-order context with $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ and abstraction operator $[\cdot]$
CCS in a nutshell

- given a set of function symbols $\Sigma$ and a set of variables $\mathcal{X}$
- terms over $\Sigma$ and $\mathcal{X}$ denoted as $\mathcal{T}_\Sigma(\mathcal{X})$
- constructors $\mathcal{C}$ and defined function symbols $\mathcal{F}$
  \[ \Sigma = \mathcal{F} \cup \mathcal{C}, \mathcal{F} \cap \mathcal{C} = \emptyset \]
- programs are sets of rewrite rules $lhs \rightarrow rhs$
- $lhs$ is of the form $F(p_1, \ldots, p_n)$ with $F \in \mathcal{F}$ and $p_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$
- conditional rewrite rules $lhs \rightarrow rhs \leftarrow cond$ where
  \[ cond \equiv \{ v_1 = u_1, \ldots, v_n = u_n \} \] and $v_i, u_i \in \mathcal{T}_\Sigma(\mathcal{X})$
- rewriting binds free variables in $v_i$, modelling variable declaration, $let$- and $case$-expressions
- higher-order context with $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ and abstraction operator $[\cdot]_\cdot$
CCS in a nutshell

- given a set of function symbols $\Sigma$ and a set of variables $\mathcal{X}$
- terms over $\Sigma$ and $\mathcal{X}$ denoted as $T_{\Sigma}(\mathcal{X})$
- constructors $\mathcal{C}$ and defined function symbols $\mathcal{F}$
  $\Sigma = \mathcal{F} \cup \mathcal{C}, \mathcal{F} \cap \mathcal{C} = \emptyset$
- programs are sets of rewrite rules $lhs \rightarrow rhs$
- $lhs$ is of the form $F(p_1, \ldots, p_n)$ with $F \in \mathcal{F}$ and $p_i \in T_{\mathcal{C}}(\mathcal{X})$
- conditional rewrite rules $lhs \rightarrow rhs \leftarrow cond$ where
  $cond \equiv \{ v_1 = u_1, \ldots, v_n = u_n \}$ and $v_i, u_i \in T_{\Sigma}(\mathcal{X})$
- rewriting binds free variables in $v_i$, modelling variable declaration, let- and case-expressions
- higher-order context with $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ and abstraction operator $[\cdot]$
CCS in a nutshell

- given a set of function symbols $\Sigma$ and a set of variables $\mathcal{X}$
- terms over $\Sigma$ and $\mathcal{X}$ denoted as $\mathcal{T}_\Sigma(\mathcal{X})$
- constructors $\mathcal{C}$ and defined function symbols $\mathcal{F}$
  \[ \Sigma = \mathcal{F} \cup \mathcal{C}, \mathcal{F} \cap \mathcal{C} = \emptyset \]
- programs are sets of rewrite rules $\text{lhs} \rightarrow \text{rhs}$
- $\text{lhs}$ is of the form $F(p_1, \ldots, p_n)$ with $F \in \mathcal{F}$ and $p_i \in \mathcal{T}_\mathcal{C}(\mathcal{X})$
- conditional rewrite rules $\text{lhs} \rightarrow \text{rhs} \leftarrow \text{cond}$ where
  \[ \text{cond} \equiv \{ v_1 = u_1, \ldots, v_n = u_n \} \] and $v_i, u_i \in \mathcal{T}_\Sigma(\mathcal{X})$
- rewriting binds free variables in $v_i$, modelling variable declaration, $\text{let}$- and $\text{case}$-expressions
- higher-order context with $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$ and abstraction operator $[-]$
Target Languages in the CCS Framework

**CCS**

```plaintext
multlast([]) -> []
multlast([A]) -> [A]
multlast([A,B|C]) -> [D,D|E]
<= [D|E] = multlast([B|C])
```

**Haskell**

```plaintext
multlast [] = []
multlast [A] = [A]
multlast [A,B|C] = 
    let [D|E] = multlast([B|C]) in [D,D|E]
```

**Prolog**

```plaintext
multlast([], []).
multlast([A], [A]).
multlast([A,B|C], [D,D|E]) :-
    multlast([B|C], [D|E]).
```
The IP task in CCS

function symbols $\mathcal{F} = \mathcal{F}_T \cup \mathcal{F}_B \cup \mathcal{F}_I$

user defined rules $R = E^+ \cup E^- \cup BK$

restriction bias ($lhs, rhs, u, v \subseteq \mathcal{T}_{\Sigma}(X)$)

preference bias ($\preceq$)

IP Task
The IP task in CCS

function symbols $\mathcal{F} = \mathcal{F}_T \cup \mathcal{F}_B \cup \mathcal{F}_I$

- $\mathcal{F}_T$ function symbols of target functions
- $\mathcal{F}_B$ function symbols of background knowledge
- $\mathcal{F}_I$ pool of function symbols for inventing sub functions

user defined rules $R = E^+ \cup E^- \cup BK$

restriction bias ($lhs, rhs, u, v \subseteq \mathcal{T}_\Sigma(\mathcal{X})$)

preference bias ($\preceq$)

IP Task
The IP task in CCS

function symbols $\mathcal{F} = \mathcal{F}_T \cup \mathcal{F}_B \cup \mathcal{F}_I$

user defined rules $R = E^+ \cup E^- \cup BK$

- $E^+$ positive evidence $F(t_1, \ldots, t_n) \rightarrow r$
- $E^-$ negative evidence as inequalities $F(t_1, \ldots, t_n) \rightarrow r$
- $BK$ background knowledge
  
  $F(t_1, \ldots, t_n) \rightarrow r \iff \{v_1 = u_1, \ldots, v_n = u_n\}$

restriction bias ($lhs, rhs, u, v \subseteq T_\Sigma(\mathcal{X})$)

preference bias ($\preceq$)

IP Task
function symbols $\mathcal{F} = \mathcal{F}_T \cup \mathcal{F}_B \cup \mathcal{F}_I$

user defined rules $R = E^+ \cup E^- \cup BK$

restriction bias ($lhs, rhs, u, v \subseteq \mathcal{T}_\Sigma(X)$)
Allow only a subset of $\mathcal{T}_\Sigma(X)$ for lhss, rhss, and conditions

preference bias ($\preceq$)

IP Task
The IP task in CCS

function symbols $\mathcal{F} = \mathcal{F}_T \cup \mathcal{F}_B \cup \mathcal{F}_I$

user defined rules $R = E^+ \cup E^- \cup BK$

restriction bias ($\text{lhs, rhs, } u, v \subseteq \mathcal{T}_\Sigma(\mathcal{X})$)

preference bias ($\preceq$)
Partial ordering on terms, lhss, rhss, conditions, rules, and programs

IP Task
The IP task in CCS

function symbols \( \mathcal{F} = \mathcal{F}_T \cup \mathcal{F}_B \cup \mathcal{F}_I \)

user defined rules \( R = E^+ \cup E^- \cup BK \)

restriction bias (\( lhs, rhs, u, v \subseteq \mathcal{T}_\Sigma(\mathcal{X}) \))

preference bias (\( \preceq \))

IP Task

Find a set of rules \( R_T \) s.t.

\[
R_T \cup BK \models E^+ \\
R_T \cup BK \not\models E^-
\]

and \( R_T \) is optimal w.r.t. restriction and preference bias.
Higher-Order Rewriting

\[
\begin{align*}
\text{map}(\{u\}Z(u), \text{nil}) & \rightarrow \text{nil} \\
\text{map}(\{u\}Z(u), \text{cons}(X, Y)) & \rightarrow \text{cons}(Z(X), \text{map}(\{u\}Z(u), Y))
\end{align*}
\]

more Terese p 612
\( C \) unrestricted
\( \mathcal{F}_T \) singleton
\( \mathcal{F}_B \) unrestricted
\( \mathcal{F}_I \) \( \emptyset \)
\( E^+ \) unrestricted
\( E^- \) unrestricted
\( BK \) unrestricted
\( X_2 \) \( \emptyset \)

**Restr. Bias** subset of SML

**Pref. Bias** user defined fitness function

**Search Str.** global search, generate and test
FLIP

\[ C \text{ unrestricted} \]
\[ F_T \text{ unrestricted} \]
\[ F_B \text{ unrestricted} \]
\[ F_I \emptyset \]
\[ E^+ \text{ unconditional} \]
\[ E^- \text{ unconditional (may be empty)} \]
\[ BK \text{ unrestricted} \]
\[ \chi_2 \emptyset \]

**Restr. Bias** \( lhs \) is a consistent (w.r.t. evidence) but restricted (no new variables on \( rhs \)) least general generalisation of two positive examples \( rhs \) is derived via inverse narrowing from two \( lhss \)

**Pref. Bias** minimum description length and coverage

**Search Str.** heuristic search with sequential covering
\( \mathcal{C} \) constants, including \( \{true, false\} \)

\( \mathcal{F}_T \) singleton

\( \mathcal{F}_B \cup \{=, \neq, <, \leq, >, \geq, \neg\} \)

\( \mathcal{F}_I \emptyset \)

\( E^+ \) unconditional

\( E^- \) unconditional (may be empty)

\( BK \) unconditional

\( \mathcal{X}_2 \emptyset \)

**restr. bias** \( l, v \in \{F(i_1, \ldots, i_n)|i_i \in \mathcal{X}_1, F \in \mathcal{F}\} \)

\( r, u \in \mathcal{T}_\Sigma(\mathcal{X}) \)

**pref. bias** *foil gain*

**search str.** sequential covering
\[ \mathcal{C} \cup \{ \text{true, false} \} \]
\[ \mathcal{F}_T \text{ singleton} \]
\[ \mathcal{F}_B \text{ unrestricted} \]
\[ \mathcal{F}_I \emptyset \]
\[ E^+ \text{ unconditional} \]
\[ E^- \text{ unconditional} \]
\[ BK \text{ unrestricted} \]
\[ \mathcal{X}_2 \emptyset \]

restr. bias \[ l, v \in \{ F(i_1, \ldots, i_n) | i_i \in \mathcal{T}_\Sigma(\mathcal{X}), F \in \mathcal{F} \} \]
\[ r, u \in \mathcal{T}_\Sigma(\mathcal{X}) \]

pref. bias clause with highest coverage in a lattice of least general generalisations relative to \( BK \) of randomly picked examples

search str. sequential covering
IGOR II

\( C \) unrestricted
\( \mathcal{F}_T \) unrestricted
\( \mathcal{F}_B \) unrestricted
\( \mathcal{F}_I \) domain of invented function equals domain of calling function (no variable invention)
\( E^+ \) unconditional
\( E^- \) \( \emptyset \)
\( BK \) unconditional
\( \chi_2 \) \( \emptyset \)

restr. bias non-overlapping lhss, \( rhs = F(\ldots) \), \( F \not\in \mathcal{F}_I \), conditions model only let-expressions

pref. bias fewer case distinctions, most specific patterns, fewer recursive calls or calls to \( BK \)

search str. best first
$\mathcal{C}$ unrestricted
$\mathcal{F}_T$ singleton
$\mathcal{F}_B$ unrestricted
$\mathcal{F}_I$ $\emptyset$
$E^+$ unrestricted
$E^-$ unrestricted
$BK$ unrestricted
$\chi_2$ only via paramorphisms from $BK$
restr. bias type constraints, composition of functions from $BK$
pref. bias smallest w.r.t. $BK$
search str. breadth first, generate and test