Inductive Synthesis of Functional Programs
Generalizing Initial Terms to Recursive Program Schemes

Emanuel Kitzelmann

Department of Information Systems and Applied Computer Science
University of Bamberg

Tuesdays Afternoon Club, Summer Term 2005
Outline

1. Introduction

2. Generalizing Initial Terms to Recursive Program Schemes
   - Recursive Program Schemes
   - Restrictions and the Generalization Problem
   - The Generalization Algorithm

3. Generation of Initial Terms from Examples
   - Overview
   - Generation of Initial Terms Based on Datatype Knowledge
What is Inductive Functional Program Synthesis?

Inductive Functional Program Synthesis

Program Synthesis

- Automatic generation of programs or algorithms from specifications.
- Classification along two dimensions:
  - deductive vs inductive synthesis,
  - fully automatic vs assisting systems
What is Inductive Functional Program Synthesis?

Inductive Functional Program Synthesis

Deductive Program Synthesis

Starts with a complete, formal specification. E.g. Manna & Waldinger: Synthesized program is result of a constructive proof of a theorem.

Example

\[ \text{last}(l) \gets \text{find } z \text{ such that for some } y: \text{ if } l \neq [] \text{ then } l = y \circ [z] \]

becomes the theorem: \( \forall l \exists z. (\exists y. l \neq [] \Rightarrow l = y \circ [z]) \)
What is Inductive Functional Program Synthesis?

Inductive Functional Program Synthesis

Inductive Program Synthesis

- Synthesis of programs from *incomplete* specifications, mostly in terms of a set of input/output examples.
- Different approaches:
  - Genetic Programming
  - Inductive Logic Programming (ILP)
  - Functional Program Synthesis

Example

\{ [a] \mapsto a,  [a, b] \mapsto b,  [a, b, c] \mapsto c,  [a, b, c, d] \mapsto d \}
What is Inductive Functional Program Synthesis?

Inductive Functional Program Synthesis

- Synthesis of functional programs from a finite set of input/output examples.
- Proceeds in two steps:
  1. Calculation of an *initial term*, which computes the given outputs from the respective inputs. Depends on domain knowledge.
  2. Detection of syntactical regularities which are generalized to recursive equations. Purely syntactical driven.
- In this talk we consider an algorithm for solving the second step.
Two Synthesis Steps

I/O-examples → Initial Term → RPS

1. Explanation

- $[a] \mapsto a$
- $[a, b] \mapsto b$
- $[a, b, c] \mapsto c$
- $[a, b, c, d] \mapsto d$

2. Generalization

- $\text{hd}($
  - if($\text{empty}(\text{tl}(x))$, $x$,
  - if($\text{empty}(\text{tl}(\text{tl}(x)))$, $\text{tl}(x)$,
    - if($\text{empty}(\text{tl}(\text{tl}(\text{tl}(x))))$, $\text{tl}(\text{tl}(x))$,
      - $\Omega))))
  - $\text{last}(x) = \text{hd}(\text{last}^\prime(x))$
  - $\text{last}^\prime(x) =$
    - if($\text{empty}(\text{tl}(x))$, $x$,
      - $\text{last}^\prime(\text{tl}(x)))$)
Two Synthesis Steps

I/O-examples $\rightarrow$ Initial Term $\rightarrow$ RPS

1. Explanation

- $[a] \mapsto a$
- $[a, b] \mapsto b$
- $[a, b, c] \mapsto c$
- $[a, b, c, d] \mapsto d$

2. Generalization

- $\text{hd}($
  - if(empty(tl(x)), x,
    - if(empty(tl(tl(x))), tl(x),
      - if(empty(tl(tl(tl(x)))), tl(tl(tl(x))), \Omega))))

- $\text{last}(x) = \text{hd}(\text{last}'(x))$

- $\text{last}'(x) =$
  - if(empty(tl(x)), x,
    - last'(tl(x)))$
Two Synthesis Steps

I/O-examples

1. Explanation
   → Initial Term

2. Generalization
   → RPS

<table>
<thead>
<tr>
<th>I/O-examples</th>
<th>Explanation</th>
<th>Initial Term</th>
<th>Generalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a] ↦ a,</td>
<td>(\text{hd}(\text{if}(\text{empty}(\text{tl}(x)), x, \text{if}(\text{empty}(\text{tl}(\text{tl}(x))), \text{tl}(x), \text{hd}(\Omega)))))</td>
<td>last(x) = (\text{hd}(\text{last}'(x)))</td>
<td></td>
</tr>
<tr>
<td>[a, b] ↦ b,</td>
<td>\text{last}'(x) = \text{if}(\text{empty}(\text{tl}(x)), x, \text{last}'(\text{tl}(x))))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[a, b, c] ↦ c,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[a, b, c, d] ↦ d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Outline

1 Introduction

2 Generalizing Initial Terms to Recursive Program Schemes
   - Recursive Program Schemes
   - Restrictions and the Generalization Problem
   - The Generalization Algorithm

3 Generation of Initial Terms from Examples
   - Overview
   - Generation of Initial Terms Based on Datatype Knowledge
Recursive Program Schemes

Definition

Given

- a signature $\Sigma$,
- a set of function variables $\Phi = \{ G_1, \ldots, G_n \}$ with arity $\alpha(G_i) > 0$ for all $i \in [1; n]$, and
- a set of variables $X$ with $\Sigma \cap \Phi \cap X = \emptyset$,

then a recursive program scheme (RPS) $S$ is a pair $S = (G, m)$ where

1. $G$ is a set of $n$ equations
   \[ \{ G_1(x_1, \ldots, x_{\alpha(G_1)}) = t_1, \ldots, G_n(x_1, \ldots, x_{\alpha(G_n)}) = t_n \}, \] where the $t_i$ are terms with respect to the signature $\Sigma \cup \Phi$ and variables $X$ and

2. $m \in [1; n]$ is a number indicating the main equation.
The examplary \textit{lasts}-RPS is given by:

\begin{itemize}
  \item \( \Sigma = \{ \text{if, empty, cons, hd, tl, \texttt{[]}} \} \),
  \item \( \Phi = \{ G_1, G_2 \}, \ G_1 = \text{lasts}, \ G_2 = \text{last}, \ \alpha(\text{lasts}) = \alpha(\text{last}) = 1 \),
  \item \( X = \{ x \} \),
  \item \( G = \{ \text{lasts}(x) = \text{if}(\text{empty}(x), \texttt{[]}), \ \text{cons}(\text{hd}(\text{last}(\text{hd}(x))), \text{lasts}(\text{tl}(x)))), \ \text{last}(x) = \text{if}(\text{empty}(\text{tl}(x)), x, \text{last}(\text{tl}(x))) \} \),
  \item \( m = 1. \)
\end{itemize}
Definition

Each equation $G_i(x_1, \ldots, x_{\alpha(G_i)}) = t_i$ of an RPS implies a rule $G_i(x_1, \ldots, x_{\alpha(G_i)}) \rightarrow t_i$ of the implied TRS. Each recursive equation with head $G_j(x_1, \ldots, x_{\alpha(G_j)})$ implies the additional rule $G_j(x_1, \ldots, x_{\alpha(G_j)}) \rightarrow \Omega$.

Example

The lasts-RPS implies the TRS which containing exactly the rules

\[
\{ \text{lasts}(x) \rightarrow \text{if}(\text{empty}(x), [], \text{cons}(\text{hd(last(hd(x)))), \text{lasts}(\text{tl(x))))), \\
\text{last}(x) \rightarrow \text{if}(\text{empty}(\text{tl(x)}), x, \text{last}(\text{tl(x)))), \\
\text{lasts}(x) \rightarrow \Omega, \quad \text{last}(x) \rightarrow \Omega \}
\]
Free Interpretation and Language of an RPS

Definition

The standard interpretation of an RPS, called *free interpretation*, is defined as the supremum of the set of all terms which can be derived by the implied TRS from the head of the main equation.

Definition

The set of all *ground* terms *without function variables*, which can be derived by the *instantiated* head of the main equation regarding some instantiation $\beta : X \rightarrow T_\Sigma$ is called *language* of an RPS relative to $\beta$.

We say that an RPS *explains* the elements of its languages.
Suppose the RPS only containing the last equation and the instantiation $\beta : X \rightarrow T_{\Sigma} : x \mapsto [a, b, c]$:

\[\text{last}([a, b, c]) \rightarrow_{R}\]

\[\text{if}(\text{empty}(\text{tl}([a, b, c])), [a, b, c], \text{last}(\text{tl}([a, b, c]))) \rightarrow_{R}\]

\[\text{if}(\text{empty}(\text{tl}([a, b, c])), [a, b, c],\]
\[\qquad \text{if}(\text{empty}(\text{tl}(\text{tl}([a, b, c]))), \text{tl}([a, b, c]), \]
\[\qquad\qquad \text{last}(\text{tl}(\text{tl}([a, b, c]))) \rightarrow_{R}\]

\[\text{if}(\text{empty}(\text{tl}([a, b, c])), [a, b, c],\]
\[\qquad \text{if}(\text{empty}(\text{tl}([a, b, c])), \text{tl}([a, b, c]),\]
\[\qquad\qquad \text{if}(\text{empty}(\text{tl}(\text{tl}([a, b, c]))), \text{tl}(\text{tl}([a, b, c])), \Omega))))\]
Concepts in RPSs and Unfoldings

- Positions in equation bodies, where a recursive call appears, are named *recursion positions*.

- The argument terms in recursive calls imply variable substitutions $\sigma_r$ and are named *substitution terms*.

- Positions in unfoldings at which unfolding steps occurred are named *unfolding positions*. They are concatenations of recursion positions.

- Variable instantiations occurring while unfolding are caused by the initial instantiation and the substitution terms. It holds:
  \[ \begin{align*}
  1 & \quad \beta_\epsilon = \beta \\
  2 & \quad \beta_{ur} = \sigma_r \beta_u, \text{ for recursion position } r \text{ and unfolding position } u
  \end{align*} \]

- Initial terms are considered as (incomplete) unfoldings, i.e., elements of the language, of the to be induced RPS.
Concepts in RPSs and Unfoldings

- Positions in equation bodies, where a recursive call appears, are named *recursion positions*.

- The argument terms in recursive calls imply variable substitutions \( \sigma_r \) and are named *substitution terms*.

- Positions in unfoldings at which unfolding steps occurred are named *unfolding positions*. They are concatenations of recursion positions.

- Variable instantiations occurring while unfolding are caused by the initial instantiation and the substitution terms. It holds:
  1. \( \beta_\epsilon = \beta \)
  2. \( \beta_{ur} = \sigma_r \beta_u \), for recursion position \( r \) and unfolding position \( u \)

- Initial terms are considered as (incomplete) unfoldings, i.e. elements of the language, of the to be induced RPS.
Concepts in RPSs and Unfoldings

- Positions in equation bodies, where a recursive call appears, are named *recursion positions*.
- The argument terms in recursive calls imply variable substitutions $\sigma_r$ and are named *substitution terms*.
- Positions in unfoldings at which unfolding steps occurred are named *unfolding positions*. They are concatenations of recursion positions.
- Variable instantiations occurring while unfolding are caused by the initial instantiation and the substitution terms. It holds:
  1. $\beta_\epsilon = \beta$
  2. $\beta_{ur} = \sigma_r \beta_u$, for recursion position $r$ and unfolding position $u$
- Initial terms are considered as (incomplete) unfoldings, i.e. elements of the language, of the to be induced RPS.

Emanuel Kitzelmann
Inductive Synthesis of Functional Programs
Concepts in RPSs and Unfoldings

- Positions in equation bodies, where a recursive call appears, are named *recursion positions*.
- The argument terms in recursive calls imply variable substitutions $\sigma_r$ and are named *substitution terms*.
- Positions in unfoldings at which unfolding steps occurred are named *unfolding positions*. They are concatenations of recursion positions.
- Variable instantiations occurring while unfolding are caused by the initial instantiation and the substitution terms. It holds:
  1. $\beta_\epsilon = \beta$
  2. $\beta_{ur} = \sigma_r \beta_u$, for recursion position $r$ and unfolding position $u$
- Initial terms are considered as (incomplete) unfoldings, i.e. elements of the language, of the to be induced RPS.
Expressions are considered as (incomplete) unfoldings, i.e. elements of the language, of the to be induced RPS.
Outline

1. Introduction

2. Generalizing Initial Terms to Recursive Program Schemes
   - Recursive Program Schemes
   - Restrictions and the Generalization Problem
   - The Generalization Algorithm

3. Generation of Initial Terms from Examples
   - Overview
   - Generation of Initial Terms Based on Datatype Knowledge
Restrictions

Inferable RPSs are restricted regarding their recursive form:

1. No mutual recursive equations, e.g. $G_i$ calls $G_j$ and $G_j$ calls $G_i$.
2. No nested recursive calls, e.g. $G_i(x) = \ldots G_j(G_i(x)) \ldots$. 
Generalization Criteria

Which RPS is the right one?
An inferred RPS has to explain the initial terms

1 \textit{recurrently}, i.e. at least one initial term is derived by applying each RPS-rule at least twice, and

2 \textit{substitution uniquely}, i.e. \textit{no} substitution term can be changed, such that the RPS still explains the initial terms.

Recurrence follows from substitution uniqueness (but not vice versa).

Hypothesis
Two RPSs, both explaining some initial trees substitution uniquely, are equivalent regarding their free interpretation.
Generalization Criteria

Which RPS is the right one?
An inferred RPS has to explain the initial terms

1. *recurrently*, i.e. at least one initial term is derived by applying each RPS-rule at least twice, and

2. *substitution uniquely*, i.e. *no* substitution term can be changed, such that the RPS still explains the initial terms.

Recurrence follows from substitution uniqueness (but not vice versa).

Hypothesis

Two RPSs, both explaining some initial trees substitution uniquely, are equivalent regarding their free interpretation.
Generalization Criteria

Which RPS is the right one?
An inferred RPS has to explain the initial terms

1. *recurrently*, i.e. at least one initial term is derived by applying each RPS-rule at least twice, and

2. *substitution uniquely*, i.e. *no* substitution term can be changed, such that the RPS still explains the initial terms.

Recurrence follows from substitution uniqueness (but not vice versa).

Hypothesis

Two RPSs, both explaining some initial trees substitution uniquely, are equivalent regarding their free interpretation.
The task is to come up with a recursive program scheme that substitution uniquely explains a set of given initial terms.
Outline

1. Introduction

2. Generalizing Initial Terms to Recursive Program Schemes
   - Recursive Program Schemes
   - Restrictions and the Generalization Problem
   - The Generalization Algorithm

3. Generation of Initial Terms from Examples
   - Overview
   - Generation of Initial Terms Based on Datatype Knowledge
Example I/O-examples, Initial Term, and RPS

\[
\begin{align*}
[] & \mapsto [], \\
[[a]] & \mapsto [a], \quad [[a, b]] \mapsto [b], \quad [[a, b, c]] \mapsto [c], \quad [[a, b, c, d]] \mapsto [d], \\
[[a], [b]] & \mapsto [a, b], \quad [[a], [b, c]] \mapsto [a, c], \\
[[a, b], [c], [d]] & \mapsto [b, c, d], \quad [[a, b], [c, d], [e, f]] \mapsto [b, d, f], \\
[[a], [b], [c], [d]] & \mapsto [a, b, c, d]
\end{align*}
\]

\[
\text{specify}
\]

\[
\text{lasts}(x) = \text{if}(\text{empty}(x), [],
\quad \text{cons}(\text{hd}(\text{last}(\text{hd}(x))), \text{lasts}(\text{tl}(x)))),
\quad \text{last}(x) = \text{if}(\text{empty}(\text{tl}(x)), x, \text{last}(\text{tl}(x)))
\]
Exemplary I/O-examples, Initial Term, and RPS
The Generalization Algorithm

Each recursive equation is induced in three steps which are organized in a divide-and-conquer pattern:

- **Divide-phase:** 1. Search for a segmentation, i.e. determining recursion and subscheme positions for each equation.
- **Conquer-phase:** 2. Calculation of equation bodies followed by 3. calculation of substitution terms.

- Only segmentation utilizes search.
- Segmentation can be seen as search through a hypothesis space.
- Correspondingly, calculation of bodies and substitution terms can be seen as constructive goal test. Success results in a completed RPS, failure initiates a backtrack to segmentation.
The Generalization Algorithm

Each recursive equation is induced in three steps which are organized in a divide-and-conquer pattern:

- Divide-phase: 1. Search for a segmentation, i.e. determining recursion and subscheme positions for each equation.
- Conquer-phase: 2. Calculation of equation bodies followed by 3. calculation of substitution terms.
- Only segmentation utilizes search.
- Segmentation can be seen as search through a hypothesis space.
- Correspondingly, calculation of bodies and substitution terms can be seen as constructive goal test. Success results in a completed RPS, failure initiates a backtrack to segmentation.
1. Step: Segmentation

- Only positions on paths leading to some $\Omega$ come into question as recursion positions (substantial narrowing of search space).
- All $\Omega$s not explained by recursion positions imply *subscheme positions*, i.e. calls of *further* recursive equations.
- Recursion and subscheme positions determine an *equation body skeleton*, which must be *equal* on each unfolding position in each initial term.
- (Greedy) search strategy for recursion positions is:
  1. Explanation of as much as possible $\Omega$s by recursion positions.
  2. As small as possible recursion positions (i.e. as small as possible equation bodies).
1. Step: Segmentation

- Only positions on pathes leading to some $\Omega$ come into question as recursion positions (substantial narrowing of search space).
- All $\Omega$s not explained by recursion positions imply *subscheme positions*, i.e. calls of *further* recursive equations.
- Recursion and subscheme positions determine an *equation body skeleton*, which must be *equal* on each unfolding position in each initial term.

  (Greedy) search strategy for recursion positions is:
  1. Explanation of as much as possible $\Omega$s by recursion positions.
  2. As small as possible recursion positions (i.e. as small as possible equation bodies).
1. Step: Segmentation

- Only positions on paths leading to some \( \Omega \) come into question as recursion positions (substantial narrowing of search space).
- All \( \Omega \)s not explained by recursion positions imply subscheme positions, i.e. calls of further recursive equations.
- Recursion and subscheme positions determine an equation body skeleton, which must be equal on each unfolding position in each initial term.
- (Greedy) search strategy for recursion positions is:
  1. Explanation of as much as possible \( \Omega \)s by recursion positions.
  2. As small as possible recursion positions (i.e. as small as possible equation bodies).
Segmentation Example
Segmentation Example

[Diagram of a tree data structure with nodes labeled with 'if', 'cons', 'hd', and 'Omega', along with variables such as 'x1', 'x2', 'x3', etc.]
Segmentation Example

Recursion position: 32
Segmentation Example

Recursion position: 32
Subscheme position: 31
Segmentation Example
2. Step: Calculation of Equation Bodies

- Initial terms can be explained by one recursive equation.
- All initial terms are splitted at unfolding positions into segments: $\{t|_u[R \leftarrow G] \mid u \in U \cap \text{pos}(t), R \subset t|_u\}$
- Segments are instantiations of the (incompl.) equation body.
- Positions which are equally labeled in each segment are assumed to belong to this equation body.
- Positions which are differently labeled in at least two segments can only belong to (different) variable instantiations.

**Definition**
The (incomplete) equation body is defined as the most specific maximal pattern of all segments.
2. Step: Calculation of Equation Bodies

- Initial terms can be explained by one recursive equation.
- All initial terms are splitted at unfolding positions into segments: \( \{ t |_u [R \leftarrow G] \mid u \in U \cap \text{pos}(t), R \subset t |_u \} \)
- Segments are instantiations of the (incompl.) equation body.
- Positions which are equally labeled in each segment are assumed to belong to this equation body.
- Positions which are differently labeled in at least two segments can only belong to (different) variable instantiations.

**Definition**

The (incomplete) equation body is defined as the most specific maximal pattern of all segments.
2. Step: Calculation of Equation Bodies

- Initial terms can be explained by *one* recursive equation.
- All initial terms are splitted at unfolding positions into *segments*: \[ \{ t_u[R ← G] \mid u ∈ U ∩ \text{pos}(t), R ⊂ t_u \} \]
- Segments are instantiations of the (incompl.) equation body.
- Positions which are equally labeled in each segment are assumed to belong to this equation body.
- Positions which are differently labeled in at least two segments can only belong to (different) variable instantiations.

**Definition**

The (incomplete) equation body is defined as the most specific maximal pattern of all segments.
2. Step: Calculation of Equation Bodies

- Initial terms can be explained by one recursive equation.
- All initial terms are splitted at unfolding positions into segments: \( \{ t_u[R \leftarrow G] \mid u \in U \cap \text{pos}(t), R \subset t_u \} \)
- Segments are instantiations of the (incompl.) equation body.
- Positions which are equally labeled in each segment are assumed to belong to this equation body.
- Positions which are differently labeled in at least two segments can only belong to (different) variable instantiations.

**Definition**

The (incomplete) equation body is defined as the most specific maximal pattern of all segments.
Calculation of Bodies – Example
Calculation of Bodies – Example

\[ G_2(x) = \text{if}(\text{empty}(\text{tl}(x)), x, G_2(\text{tl}(x))) \]
Calculation of Bodies – Example
Calculation of Bodies – Example
Calculation of Bodies – Example

(Incomplete) body of equation $G_1$:

\[
\text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1))
\]
3. Step: Calculation of Substitution Terms

- As result of antiunifying the segments for determining the body, variables and their instantiations in unfoldings are given.
- Recall the inductive structure of instantiations in unfoldings:
  1. $\beta_\epsilon = \beta$
  2. $\beta_{ur} = \sigma_r \beta_u$, for recursion position $r$ and unfolding position $u$

By means of this characterization we can calculate the substitution terms $\sigma_r$ based on the instantiations in unfoldings $\beta_u$.

- Special case: *Hidden variables*. That are variables not occurring in an (incomplete) equation body, but only in substitution terms. Thus, neither hidden variables, nor their instantiations in unfoldings are given as result from antiunifying the segments.
3. Step: Calculation of Substitution Terms

- As result of antiunifying the segments for determining the body, variables and their instantiations in unfoldings are given.
- Recall the inductive structure of instantiations in unfoldings:
  1. \( \beta_\epsilon = \beta \)
  2. \( \beta_{ur} = \sigma_r \beta_u \), for recursion position \( r \) and unfolding position \( u \)

By means of this characterization we can calculate the substitution terms \( \sigma_r \) based on the instantiations in unfoldings \( \beta_u \).

- Special case: Hidden variables. That are variables not occurring in an (incomplete) equation body, but only in substitution terms. Thus, neither hidden variables, nor their instantiations in unfoldings are given as result from antiunifying the segments.
3. Step: Calculation of Substitution Terms

- As result of antiunifying the segments for determining the body, variables and their instantiations in unfoldings are given.
- Recall the inductive structure of instantiations in unfoldings:
  1. $\beta_\epsilon = \beta$
  2. $\beta_{ur} = \sigma_r \beta_u$, for recursion position $r$ and unfolding position $u$

By means of this characterization we can calculate the substitution terms $\sigma_r$ based on the instantiations in unfoldings $\beta_u$.

- Special case: Hidden variables. That are variables not occurring in an (incomplete) equation body, but only in substitution terms. Thus, neither hidden variables, nor their instantiations in unfoldings are given as result from antiunifying the segments.
Calculation of Substitution Terms continues

\( r \) denotes recursion, \( u \) denotes unfolding positions:

From \( \beta_\epsilon = \beta, \beta_{ur} = \sigma_r \beta_u \) follows:

**Lemma**

1. If \( (x_i \sigma_r)|_v = x_j \) then for all \( u \in U \) hold \( (x_i \beta_{ur})|_v = x_j \beta_u \).
2. If \( (x_i \sigma_r)|_v = f((x_i \sigma_r)|_{v_1}, \ldots, (x_i \sigma_r)|_{v_n}), f \in \Sigma, \alpha(f) = n \) then for all \( u \in U \) hold \( \text{node}(x_i \beta_{ur}, v) = f \).

- Reading the implications in reverse direction yields a basic algorithm for calculating the substitution terms.
- If none of the two conclusions is given, a hidden variable \( x_j \) is hypothesized. Reading implication one in regular direction yields the unfolding instantiations of the hidden variable.
- For each inferred substitution term, substitution uniqueness have to be checked.
Calculation of Substitution Terms continues

\( r \) denotes recursion, \( u \) denotes unfolding positions:
From \( \beta_\epsilon = \beta, \beta_{ur} = \sigma_r \beta_u \) follows:

**Lemma**

1. If \((x_i \sigma_r)|_v = x_j\) then for all \(u \in U\) hold \((x_i \beta_{ur})|_v = x_j \beta_u\).
2. If \((x_i \sigma_r)|_v = f((x_i \sigma_r)|_{v1}, \ldots, (x_i \sigma_r)|_{vn})\), \(f \in \Sigma, \alpha(f) = n\) then for all \(u \in U\) hold \(\text{node}(x_i \beta_{ur}, v) = f\).

- Reading the implications in reverse direction yields a basic algorithm for calculating the substitution terms.
- If none of the two conclusions is given, a hidden variable \(x_j\) is hypothesized. Reading implication one in regular direction yields the unfolding instantiations of the hidden variable.
- For each inferred substitution term, substitution uniqueness have to be checked.
Calculation of Substitution Terms – Example

- Only recursion position: 32, only variable: \( x \), unfolding instantiations: \( x\beta_\epsilon = x \), \( x\beta_{32} = \text{tl}(x) \), \( x\beta_{3232} = \text{tl}(\text{tl}(x)) \), searched for: \( x\sigma_{32} \)

- Starting with position \( \epsilon \):
  1. There is no \( u \), such that holds: \( x\beta_{u32} = x\beta_u \)
  2. Yet for all \( u \) hold: \( \text{node}(x\beta_{u32}, \epsilon) = \text{tl} \)

Thus \( x\sigma_{32} = \text{tl}((x\sigma_{32})|_1) \)

- It remains position 1:
  1. For all \( u \) hold: \( (x\beta_{u32})|_1 = x\beta_u \)

Thus \( (x\sigma_{32})|_1 = x \), no remaining positions.

- Calculated substitution term: \( x\sigma_{32} = \text{tl}(x) \)

- \( \sigma_{32} \) is substitution unique.
Calculation of Substitution Terms – Example

- Only recursion position: 32, only variable: $x$, unfolding instantiations: $x\beta_\epsilon = x$, $x\beta_{32} = \text{tl}(x)$, $x\beta_{3232} = \text{tl}(\text{tl}(x))$, searched for: $x\sigma_{32}$

- Starting with position $\epsilon$:
  1. There is no $u$, such that holds: $x\beta_{u32} = x\beta_u$
  2. Yet for all $u$ hold: $\text{node}(x\beta_{u32}, \epsilon) = \text{tl}$

Thus $x\sigma_{32} = \text{tl}((x\sigma_{32})|_1)$

- It remains position 1:
  1. For all $u$ hold: $(x\beta_{u32})|_1 = x\beta_u$

Thus $(x\sigma_{32})|_1 = x$, no remaining positions.

- Calculated substitution term: $x\sigma_{32} = \text{tl}(x)$

- $\sigma_{32}$ is substitution unique.
Calculation of Substitution Terms – Example

- Only recursion position: 32, only variable: \( x \), unfolding instantiations: \( x\beta_\epsilon = x \), \( x\beta_{32} = \text{tl}(x) \), \( x\beta_{3232} = \text{tl}(\text{tl}(x)) \), searched for: \( x\sigma_{32} \)

- Starting with position \( \epsilon \):
  1. There is no \( u \), such that holds: \( x\beta_{u32} = x\beta_u \)
  2. Yet for all \( u \) hold: \( \text{node}(x\beta_{u32}, \epsilon) = \text{tl} \)

  Thus \( x\sigma_{32} = \text{tl}((x\sigma_{32})|_1) \)

- It remains position 1:
  1. For all \( u \) hold: \( (x\beta_{u32})|_1 = x\beta_u \)

  Thus \( (x\sigma_{32})|_1 = x \), no remaining positions.

- Calculated substitution term: \( x\sigma_{32} = \text{tl}(x) \)

- \( \sigma_{32} \) is substitution unique.
Calculation of Substitution Terms – Example

- Only recursion position: 32, only variable: $x$, unfolding instantiations: $x\beta_{\epsilon} = x$, $x\beta_{32} = \text{tl}(x)$, $x\beta_{3232} = \text{tl}(\text{tl}(x))$, searched for: $x\sigma_{32}$
- Starting with position $\epsilon$:
  1. There is no $u$, such that holds: $x\beta_{u32} = x\beta_u$
  2. Yet for all $u$ hold: $\text{node}(x\beta_{u32}, \epsilon) = \text{tl}$

Thus $x\sigma_{32} = \text{tl}((x\sigma_{32})|_1)$

- It remains position 1:
  1. For all $u$ hold: $(x\beta_{u32})|_1 = x\beta_u$

Thus $(x\sigma_{32})|_1 = x$, no remaining positions.

- Calculated substitution term: $x\sigma_{32} = \text{tl}(x)$
- $\sigma_{32}$ is substitution unique.
Putting All Parts Together

- Found recursion position: 32, resulting subscheme position: 33
- Infered equation $G_2$: $G_2(x) = \text{if}(\text{empty}(\text{tl}(x)), x, G_2(\text{tl}(x)))$
- Incomplete body of $G_1$:
  \[ \text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1)) \]
- Infered substitution term for $G_1$: $x \sigma_{32} = \text{tl}(x)$
- Infered equation $G_1$:
  $G_1(x) = \text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1(\text{tl}(x))))$
- Both equations together are the intended \textit{lasts}-RPS.
Putting All Parts Together

- Found recursion position: 32, resulting subscheme position: 33
- Infered equation $G_2$: $G_2(x) = \text{if}(\text{empty}(\text{tl}(x)), x, G_2(\text{tl}(x)))$
- Incomplete body of $G_1$:
  
  $\text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1))$

- Infered substitution term for $G_1$: $x \sigma_{32} = \text{tl}(x)$
- Infered equation $G_1$:
  
  $G_1(x) = \text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1(\text{tl}(x))))$

- Both equations together are the intended lasts-RPS.
Putting All Parts Together

- Found recursion position: 32, resulting subscheme position: 33
- Inferred equation $G_2$: $G_2(x) = \text{if}(\text{empty}(\text{tl}(x)), x, G_2(\text{tl}(x)))$
- Incomplete body of $G_1$:
  $\text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1))$
- Inferred substitution term for $G_1$: $x \sigma_{32} = \text{tl}(x)$
- Inferred equation $G_1$:
  $G_1(x) = \text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1(\text{tl}(x))))$
- Both equations together are the intended \textit{lasts}-RPS.
Putting All Parts Together

- Found recursion position: 32, resulting subscheme position: 33
- Infered equation $G_2$: $G_2(x) = \text{if}(\text{empty}(tl(x)), x, G_2(tl(x)))$
- Incomplete body of $G_1$:
  \[
  \text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1))
  \]
- Infered substitution term for $G_1$: $x \sigma_{32} = tl(x)$
- Infered equation $G_1$:
  \[
  G_1(x) = \text{if}(\text{empty}(x), [], \text{cons}(\text{hd}(G_2(\text{hd}(x))), G_1(tl(x))))
  \]
- Both equations together are the intended \textit{lasts}-RPS.
Putting All Parts Together

- Found recursion position: 32, resulting subscheme position: 33
- Infered equation $G_2$: $G_2(x) = if\(empty(tl(x)), x, G_2(tl(x))\)$
- Incomplete body of $G_1$:
  $if\(empty(x), [], cons(hd(G_2(hd(x))), G_1)\)$
- Infered substitution term for $G_1$: $x\sigma_{32} = tl(x)$
- Infered equation $G_1$:
  $G_1(x) = if\(empty(x), [], cons(hd(G_2(hd(x))), G_1(tl(x)))\)$
- Both equations together are the intended $lasts$-RPS.
Putting All Parts Together

- Found recursion position: 32, resulting subscheme position: 33
- Inferred equation \(G_2\): \(G_2(x) = \text{if}(\text{empty}(\text{tl}(x)), x, G_2(\text{tl}(x)))\)
- Incomplete body of \(G_1\):
  \[\text{if}(\text{empty}(x), [], \text{cons} (\text{hd}(G_2(\text{hd}(x))), G_1))\]
- Inferred substitution term for \(G_1\): \(x\sigma_{32} = \text{tl}(x)\)
- Inferred equation \(G_1\):
  \(G_1(x) = \text{if}(\text{empty}(x), [], \text{cons} (\text{hd}(G_2(\text{hd}(x))), G_1(\text{tl}(x))))\)
- Both equations together are the intended \textit{lasts}-RPS.
Outline

1. Introduction

2. Generalizing Initial Terms to Recursive Program Schemes
   - Recursive Program Schemes
   - Restrictions and the Generalization Problem
   - The Generalization Algorithm

3. Generation of Initial Terms from Examples
   - Overview
   - Generation of Initial Terms Based on Datatype Knowledge
Generation of Initial Terms from Examples

- Depends on domain knowledge.
- Different approaches:
  - Explanation based on knowledge of inductive datatypes
  - AI planning
  - Genetic Programming
- We shortly introduce the first approach.
Outline

1. Introduction

2. Generalizing Initial Terms to Recursive Program Schemes
   - Recursive Program Schemes
   - Restrictions and the Generalization Problem
   - The Generalization Algorithm

3. Generation of Initial Terms from Examples
   - Overview
   - Generation of Initial Terms Based on Datatype Knowledge
Exemplary datatype *List*: $$\text{List } \alpha = [\ ] \mid \text{cons}(\alpha, \text{List } \alpha)$$

- Used functions: *cons* and *[]* as constructors, *hd* and *tl* as functions for selecting the head and the tail of a list respectively, *empty* as predicate function for testing, whether a list is empty, and *if* as 3-ary conditional.

- For each example output a trace is calculated, which express the output in terms of the input.

- Subsequently the traces are integrated in an initial term.
Calculation of Traces

Two steps:
1. Calculation of all subexpressions of the input using \(hd\) and \(tl\).
2. Construction of the trace from the subexpressions.

Two conditions:
1. Each atom in the output is contained in the input.
2. Each atom occurs exactly once in the input.

I/O pair: \([a, b]] \rightarrow [b]\)

- All subexpressions of \([a, b]]: \(x = [a, b], \ hd(x) = [a, b], \ tl(x) = [], \ hd(hd(x)) = a, \ tl(hd(x)) = [b], \ hd(tl(hd(x))) = b, \ tl(tl(hd(x))) = []\)
- Trace: \(\text{cons}(hd(tl(hd(x))), [])\)
Integration of Traces into an Initial Term

- Inputs have to be (partially) ordered.
- If all traces have the same root, this root becomes the root of the initial term. Then recursively all subterms are constructed from the inputs and respective subtraces.
- If roots differ,
  1. calculate a predicate term for discriminating the smallest input from the others,
  2. construct an *if*-term with the *if*-symbol as root, the predicate term as first subterm, the trace for the smallest input as second subterm, and recursively calculate the third subterm from the remaining inputs and the remaining traces.
- If roots differ and only two input/trace pairs remain, then construct an *if*-term with $\Omega$ as third subtree.
Integration of Traces into an Initial Term

- Inputs have to be (partially) ordered.
- If all traces have the same root, this root becomes the root of the initial term. Then recursively all subterms are constructed from the inputs and respective subtraces.
- If roots differ,
  1. calculate a predicate term for discriminating the smallest input from the others,
  2. construct an *if*-term with the *if*-symbol as root, the predicate term as first subterm, the trace for the smallest input as second subterm, and recursively calculate the third subterm from the remaining inputs and the remaining traces.
- If roots differ and only two input/trace pairs remain, then construct an *if*-term with Ω as third subtree.
Integration of Traces into an Initial Term

- Inputs have to be (partially) ordered.
- If all traces have the same root, this root becomes the root of the initial term. Then recursively all subterms are constructed from the inputs and respective subtraces.
- If roots differ,
  1. calculate a predicate term for discriminating the smallest input from the others,
  2. construct an if-term with the if-symbol as root, the predicate term as first subterm, the trace for the smallest input as second subterm, and recursively calculate the third subterm from the remaining inputs and the remaining traces.
- If roots differ and only two input/trace pairs remain, then construct an if-term with Ω as third subtree.
Inductive synthesis of functional programs is automatic generation of functional programs from input/output examples of the intended program.

Synthesis proceeds in two steps:

1. Calculation of an initial term, which is considered as unfolding of the searched for set of recursive equations.
2. Generalization to a set of recursive equations.

The second step is based on detection of recurrent patterns in the initial term.

Outlook

- Investigation and extension of the first step
- Overcome the restrictions of the second step
Summary

Inductive synthesis of functional programs is automatic generation of functional programs from input/output examples of the intended program.

Synthesis proceeds in two steps:

1. Calculation of an initial term, which is considered as unfolding of the searched for set of recursive equations.
2. Generalization to a set of recursive equations.

The second step is based on detection of recurrent patterns in the initial term.

Outlook

- Investigation and extension of the first step
- Overcome the restrictions of the second step
Summary

- Inductive synthesis of functional programs is automatic generation of functional programs from input/output examples of the intended program.

- Synthesis proceeds in two steps:
  1. Calculation of an initial term, which is considered as unfolding of the searched for set of recursive equations.
  2. Generalization to a set of recursive equations.

- The second step is based on detection of recurrent patterns in the initial term.

Outlook

- Investigation and extension of the first step
- Overcome the restrictions of the second step
Inductive synthesis of functional programs is automatic generation of functional programs from input/output examples of the intended program.

Synthesis proceeds in two steps:

1. Calculation of an initial term, which is considered as unfolding of the searched for set of recursive equations.
2. Generalization to a set of recursive equations.

The second step is based on detection of recurrent patterns in the initial term.

Outlook

- Investigation and extension of the first step
- Overcome the restrictions of the second step