Induktive Programmsynthese

Einführung Higher-Order Konzepte durch Kombination analytischer und schemabasierter Methoden

Martin Hofmann

Gruppe Kognitive Systeme
Fakultät für Wirtschafts- und Angewandte Informatik
Universität Bamberg

Berlin 2008

Gliederung

Gliederung

Induktive Programmsynthese (IP)

Induktive Programmsynthese (IP) erforscht das automatische Erzeugen (rekursiver) Programme *unvollständigen* Spezifikationen, z.B. aus Input/Output-Beispielen (I/O-Beispiele).

Example (*Reverse*)

I/O-Beispiele:

$$\mathit{Reverse}([]) = [],\ \mathit{Reverse}([a]) = [a],\ \mathit{Reverse}([a,b]) = [b,a], \ldots$$

Induziertes funktionales Pogramm:

$$\mathit{Reverse}([]) = []$$
$$\mathit{Reverse}(\mathrm{cons}(X, Xs)) = \mathit{Reverse}(Xs) \mathbin{@} \mathrm{cons}(X, [])$$

Immanente Probleme

Intendiertheit Erfüllen der *unvollständigen* Spezifikation als *einziges* Korrektheitskriterium des IP-Algorithmus' lässt nicht intendierte Funktionen als Lösung zu.

Skalierbarkeit Ab einer gewissen (syntaktischen) Komplexität der möglichen Programme ist Inductive Programming ein *Suchproblem* (exponentielle Komplexität).

Begriffe

Preference Bias **Kriterium** zum Auswählen eines der (**semantisch** verschiedenen!) möglichen Lösungsprogramme, z.B. syntaktische Größe, Anzahl der Fallunterscheidungen, Laufzeit.

Restriction Bias **Einschränkung** der induzierbaren Programmklasse durch **syntaktische** Constraints, z.B. lineare Rekursion als einzige Rekursionsform.

Hintergrundwissen Bereits **implementierte Funktionen**, die aufgerufen werden dürfen, z.B. *append* und *partition* für Quicksort.

Unterprogramme Funktionen, die **weder** als **Zielfunktion** spezifiziert **noch** im **Hintergrundwissen** vorhanden sind, sondern vom IP-Algorithmus selbst als Hilfsfunktion eingeführt werden.

Begriffe

Preference Bias **Kriterium** zum Auswählen eines der (**semantisch** verschiedenen!) möglichen Lösungsprogramme, z.B. syntaktische Größe, Anzahl der Fallunterscheidungen, Laufzeit.

Restriction Bias **Einschränkung** der induzierbaren Programmklasse durch **syntaktische** Constraints, z.B. lineare Rekursion als einzige Rekursionsform.

Hintergrundwissen Bereits **implementierte Funktionen**, die aufgerufen werden dürfen, z.B. *append* und *partition* für Quicksort.

Unterprogramme Funktionen, die **weder** als **Zielfunktion** spezifiziert **noch** im **Hintergrundwissen** vorhanden sind, sondern vom IP-Algorithmus selbst als Hilfsfunktion eingeführt werden.

Begriffe

Preference Bias **Kriterium** zum Auswählen eines der (**semantisch** verschiedenen!) möglichen Lösungsprogramme, z.B. syntaktische Größe, Anzahl der Fallunterscheidungen, Laufzeit.

Restriction Bias **Einschränkung** der induzierbaren Programmklasse durch **syntaktische** Constraints, z.B. lineare Rekursion als einzige Rekursionsform.

Hintergrundwissen Bereits **implementierte Funktionen**, die aufgerufen werden dürfen, z.B. *append* und *partition* für Quicksort.

Unterprogramme Funktionen, die **weder** als **Zielfunktion** spezifiziert **noch** im **Hintergrundwissen** vorhanden sind, sondern vom IP-Algorithmus selbst als Hilfsfunktion eingeführt werden.

Begriffe

Preference Bias **Kriterium** zum Auswählen eines der (**semantisch** verschiedenen!) möglichen Lösungsprogramme, z.B. syntaktische Größe, Anzahl der Fallunterscheidungen, Laufzeit.

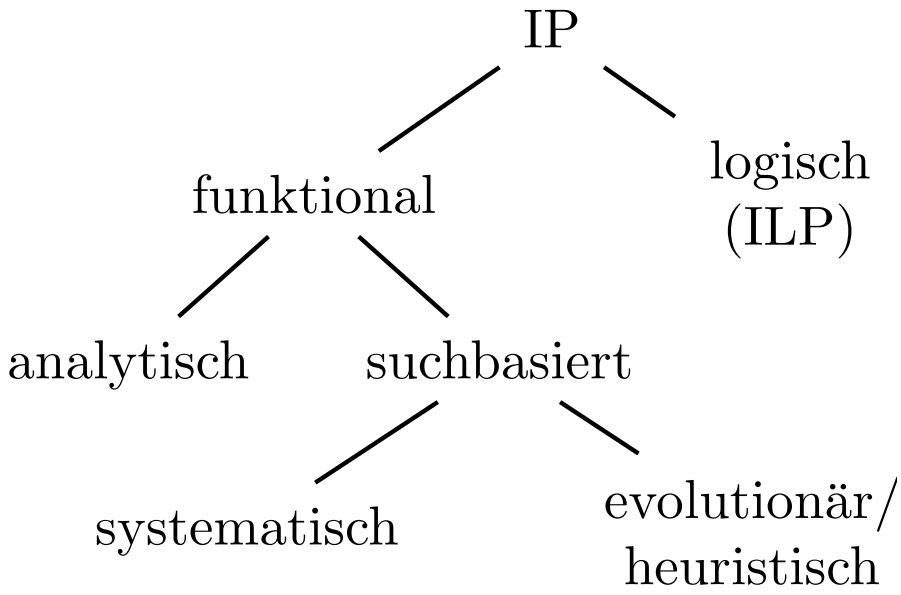
Restriction Bias **Einschränkung** der induzierbaren Programmklasse durch **syntaktische** Constraints, z.B. lineare Rekursion als einzige Rekursionsform.

Hintergrundwissen Bereits **implementierte Funktionen**, die aufgerufen werden dürfen, z.B. *append* und *partition* für Quicksort.

Unterprogramme Funktionen, die **weder** als **Zielfunktion** spezifiziert **noch** im **Hintergrundwissen** vorhanden sind, sondern vom IP-Algorithmus selbst als Hilfsfunktion eingeführt werden.

Gliederung

Verschiedene Ansätze



Analytisch

Suchbasiert

Induktives Logisches Programmieren (ILP)

Verschiedene Ansätze

Analytisch

- Beobachtung: I/O-Beispiele rekursiv definierter Funktionen weisen einen regelmäßigen Zusammenhang auf, da verschiedene Inputs “Unterprobleme” voneinander sind und daher Outputs andere Outputs als Unterterme enthalten.
- Diese Beziehung wird ermittelt und daraus die rekursive Definition abgeleitet.
- Systeme: Summers’ THESYS, IGOR I (Schmid, Mühlpfordt, Kitzelmann)

Suchbasiert

Induktives Logisches Programmieren (ILP)

Verschiedene Ansätze

Analytisch

Suchbasiert (1): evolutionär heuristisch

- I/O-Beispiele (oder allgemeinere Spezifikationen) und gewünschte Programmeigenschaften dienen als Fitness-Funktion evolutionär generierter Programm-Individuen.
- Bewertete Programmeigenschaften sind z.B. Laufzeit oder Programmgröße
- Systeme: ADATE (Roland Olsson)

Induktives Logisches Programmieren (ILP)

Verschiedene Ansätze

Analytisch

Suchbasiert (2): systematisch

- systematisches Aufzählen aller korrekten Programme
- gewöhnlich mit zusätzlichen Constraints um Suchraum zu verringern (Typinformation, Bibliothek)
- I/O Beispiele dienen als Filter
- Systeme: MAGICHASKELLER (Katayama)

Induktives Logisches Programmieren (ILP)

Verschiedene Ansätze

Analytisch

Suchbasiert

Induktives Logisches Programmieren (ILP)

- ILP ist maschinelles Lernen mit Repräsentation und Induktionstechniken basierend auf *Computational Logic* (PROLOG).
- IP als Spezialfall von ILP.
- Systeme: FOIL/FFOIL (Quinlan), GOLEM (Muggleton), DIALOGS (Flener)

Was ist heute möglich?

Automatisch benötigte Unterprogramme finden und einbinden.

Example (*Reverse*)

$$Reverse([]) = []$$
$$Reverse([X|Xs]) = [\ Last([X|Xs]) \mid Reverse(Init([X|Xs]))\ ]$$

$$Last([X]) = X$$
$$Last([X, Y|Xs]) = Last([Y|Xs])$$

$$Init([X]) = []$$
$$Init([X, Y|Xs]) = [\ X \mid Init([Y|Xs])\ ]$$

Was ist heute möglich?

Komplexe Fallunterscheidungen.

Example (*Sum*)

$$Sum([]) = 0$$
$$Sum([0|Xs]) = Sum(Xs)$$
$$Sum([s(X)|Xs]) = s(Sum([X|Xs]))$$

Was ist heute möglich?

Rekursion über mehrere Parameter.

Example ($\leq$)

$$0 \leq Y = t$$
$$s(X) \leq 0 = f$$
$$s(X) \leq s(Y) = X \leq Y$$

Was ist heute möglich?

Wechselseitig abhängige Funktionen.

Example (*Odd*/*Even*)

$$Odd(0) = f$$
$$Odd(s(X)) = Even(X)$$

$$Even(0) = t$$
$$Even(s(X)) = Odd(X)$$

Was ist heute möglich?

Komplexe Funktionsaufruf-Relationen.

Example (*QuickSort*)

$$\mathit{QuickSort}([]) = []$$
$$\mathit{QuickSort}([X|Xs]) =$$
$$\mathit{QuickSort}(\mathrm{part}_{\leq}([X|Xs])) \mathbin{@} [X] \mathbin{@} \mathit{QuickSort}(\mathrm{part}_{>}([X|Xs]))$$

Was ist heute möglich?

Bei evolutionären Ansätzen (ADATE) theoretisch alles!

- Optimierung von Graphikalgorithmen
- Fahrzeugsteuerung über ANN
- komplexe numerische Berechnungen
- ...

Speicher und Zeitproblem !!!

Empirischer Systemvergleich

	lasts	*last*	*member*	*odd/even*	*multlast*	*isort*	*reverse*	*weave*	*shiftr*	*mult/add*	*allodds*
ADATE	822.0	0.2	2.0	—	4.3	70.0	78.0	80.0	18.81	—	214.87
FLIP	×	0.020	17.868	0.130	$448.90^{\perp}$	×	—	$134.24^{\perp}$	$448.55^{\perp}$	×	×
FFOIL	$0.7^{\perp}$	0.1	$0.1^{\perp}$	$< 0.1^{\perp}$	< 0.1	×	—	$0.4^{\perp}$	$< 0.1^{\perp}$	$8.1^{\perp}$	$0.1^{\perp}$
GOLEM	1.062	< 0.001	0.033	—	< 0.001	0.714	—	$0.66^{\perp}$	0.298	—	$0.016^{\perp}$
IGOR II	5.695	0.007	0.152	0.019	0.023	0.105	0.103	0.200	0.127	⊙	⊙
MAGH.	19.43	0.01	⊙	—	0.30	0.01	0.08	⊙	157.32	—	×

— not tested × stack overflow ⊙ timeout $\perp$ wrong
all runtimes in seconds

Gliederung

Mögliche Anwendungsfelder I: End-user Programming

Zwei Beispiele:

- Generieren von XSL Transformationen aus XML-Beispielen.
- Generieren von komplexen Suchen/Ersetzen-Regeln aufgrund einiger Beispielinstanzen.

Probleme:

- Problemspezifische Aufbereitung der Benutzeraktionen zu geeigneten Beispielen.
- Dadurch i.d.R. starke Vorannahmen nötig, die die Problemklasse stark einschränken.
- Resultierende Problemklasse reizt IP nicht mehr aus.

Mögliche Anwendungsfelder I: End-user Programming

Zwei Beispiele:

- Generieren von XSL Transformationen aus XML-Beispielen.
- Generieren von komplexen Suchen/Ersetzen-Regeln aufgrund einiger Beispielinstanzen.

Probleme:

- Problemspezifische Aufbereitung der Benutzeraktionen zu geeigneten Beispielen.
- Dadurch i.d.R. starke Vorannahmen nötig, die die Problemklasse stark einschränken.
- Resultierende Problemklasse reizt IP nicht mehr aus.

Mögliche Anwendungsfelder I: End-user Programming

Zwei Beispiele:

- Generieren von XSL Transformationen aus XML-Beispielen.
- Generieren von komplexen Suchen/Ersetzen-Regeln aufgrund einiger Beispielinstanzen.

Probleme:

- Problemspezifische Aufbereitung der Benutzeraktionen zu geeigneten Beispielen.
- Dadurch i.d.R. starke Vorannahmen nötig, die die Problemklasse stark einschränken.
- Resultierende Problemklasse reizt IP nicht mehr aus.

Mögliche Anwendungsfelder I: End-user Programming

Zwei Beispiele:

- Generieren von XSL Transformationen aus XML-Beispielen.
- Generieren von komplexen Suchen/Ersetzen-Regeln aufgrund einiger Beispielinstanzen.

Probleme:

- Problemspezifische Aufbereitung der Benutzeraktionen zu geeigneten Beispielen.
- Dadurch i.d.R. starke Vorannahmen nötig, die die Problemklasse stark einschränken.
- Resultierende Problemklasse reizt IP nicht mehr aus.

Mögliche Anwendungsfelder II: Softwaretechnik

“Natürliche” Anwendung: Alternative/zusätzliche Art des Programmierens und der Algorithmenentwicklung.
Probleme: Intendiertheit und Skalierbarkeit.

Softwareengineering – Designphase

Sukzessive, immer detailiertere Spezifikation von Komponenten/ Klassen/ Methoden in der Designphase des Softwareprozesses, I/Os und Testfälle sollen nicht nur als “Programmierernotiz” dienen, sondern konkret zur Programmsynthese.

Testen/Debuggen

Testfälle und evtl. fehlerhaftes Programm als Input für IP-System welches Fehler feststellt, lokalisiert *und ausbessert*.

Mögliche Anwendungsfelder II: Softwaretechnik

“Natürliche” Anwendung: Alternative/zusätzliche Art des Programmierens und der Algorithmenentwicklung.
Probleme: Intendiertheit und Skalierbarkeit.

Softwareengineering – Designphase

Sukzessive, immer detailiertere Spezifikation von Komponenten/ Klassen/ Methoden in der Designphase des Softwareprozesses, I/Os und Testfälle sollen nicht nur als “Programmierernotiz” dienen, sondern konkret zur Programmsynthese.

Testen/Debuggen

Testfälle und evtl. fehlerhaftes Programm als Input für IP-System welches Fehler feststellt, lokalisiert *und ausbessert*.

Mögliche Anwendungsfelder II: Softwaretechnik

“Natürliche” Anwendung: Alternative/zusätzliche Art des Programmierens und der Algorithmenentwicklung.
Probleme: Intendiertheit und Skalierbarkeit.

Softwareengineering – Designphase

Sukzessive, immer detailiertere Spezifikation von Komponenten/ Klassen/ Methoden in der Designphase des Softwareprozesses, I/Os und Testfälle sollen nicht nur als “Programmierernotiz” dienen, sondern konkret zur Programmsynthese.

Testen/Debuggen

Testfälle und evtl. fehlerhaftes Programm als Input für IP-System welches Fehler feststellt, lokalisiert *und ausbessert*.

Gliederung

Grundlegende Idee

- IGOR II (Kitzelmann):
  - inspiriert von Summer's THESYS, Weiterentwicklung von IGOR I
  - Rekursion aufgrund von Regularitäten in den Beispielen entdecken
  - integrierte *Suche*.
- Stärken:
  - **Terminierung** per Konstruktion
  - beliebige **nutzerdefinierte Datentypen**
  - beliebiges **Hintergrundwissen** nutzbar
  - **Unterprogramme**
  - **komplexere Aufruf-Beziehungen** (Baumrekursion, Vernestung)
  - Beispielgleichungen mit **Variablen**
  - parallele Induktion mehrerer (abhängiger) Zielfunktionen

Input

- Datentyp Definition, z.B.

$$\text{List} = [] \mid \text{cons}(\text{Elt}, \text{List})$$

- die ersten k nicht-rekursiven Gleichungen für Zielfunktion und Hintergrundwissen, z.B:
  - Zielfunktion:

    $\mathit{Reverse} : \text{List} \rightarrow \text{List}$

    $\mathit{Reverse}([]) = [],\ \mathit{Reverse}([X]) = [X],\ \mathit{Reverse}([X, Y]) = [Y, X],\ \ldots$

  - Hintergrundwissen:

    $\text{snoc} : \text{List Elt} \rightarrow \text{List}$

    $\text{snoc}([], X) = [X],\ \text{snoc}([X], Y) = [X, Y],\ \ldots$

Input

- Datentyp Definition, z.B.

$$\text{List} = [] \mid \text{cons}(\text{Elt}, \text{List})$$

- die ersten *k* nicht-rekursiven Gleichungen für Zielfunktion und Hintergrundwissen, z.B:
  - Zielfunktion:

    $\mathit{Reverse} : \text{List} \rightarrow \text{List}$

    $\mathit{Reverse}([]) = [],\ \mathit{Reverse}([X]) = [X],\ \mathit{Reverse}([X, Y]) = [Y, X],\ \ldots$

  - Hintergrundwissen:

    $\text{snoc} : \text{List Elt} \rightarrow \text{List}$

    $\text{snoc}([], X) = [X],\ \text{snoc}([X], Y) = [X, Y],\ \ldots$

Output

Menge an Gleichungen die die gegebenen Beispielgleichungen modellieren.

- Fallunterscheidung duch *Pattern Matching*.
- Preference Bias: Minimale Anzahl an Fallunterscheidungen.
- Einige syntaktische Constraints, u.a. dürfen die Patterns paarweise nicht unifizieren.

Reverse-Beispiel:

$$\mathit{Reverse}([]) = []$$
$$\mathit{Reverse}([X|Xs]) = \mathrm{snoc}(\mathit{Reverse}(Xs), X)$$

Gliederung

Initiale Hypothese

Für eine gegebene (Unter-)Menge Beispielgleichungen wird versucht, *eine* Gleichung zu finden. Erste Hypothese durch **Antiunifikation** der Beispielgleichungen:

Example

Beispielgleichungen:

$$Reverse([B]) = [B], \quad Reverse([A, B]) = [B, A]$$

Initiale Hypothese:

$$Reverse([X|Xs]) = [Y|Ys]$$

Initiale Hypothese weiterverarbeiten

Example

Initiale Hypothese:

$$Reverse([X|Xs]) = [Y|Ys]$$

Problem: Ungebundene Variablen Y, Ys in der rechten Seite.

Drei Lösungsmöglichkeiten:

1. **Partitionierung** der Beispielgleichungen
   $\leadsto$ *Menge* von Gleichungen, Fallunterscheidung.
2. Ersetzen der rechten Seite durch **Programmaufruf** (rekursiv oder Hintergrundwissen).
3. Ersetzen der Unterterme mit ungebundenen Variablen durch **Unterprogramme**.

Initiale Hypothese weiterverarbeiten

Example

Initiale Hypothese:

$$Reverse([X|Xs]) = [Y|Ys]$$

Problem: Ungebundene Variablen Y, Ys in der rechten Seite.

Drei Lösungsmöglichkeiten:

1. **Partitionierung** der Beispielgleichungen
   $\leadsto$ *Menge* von Gleichungen, Fallunterscheidung.
2. Ersetzen der rechten Seite durch **Programmaufruf** (rekursiv oder Hintergrundwissen).
3. Ersetzen der Unterterme mit ungebundenen Variablen durch **Unterprogramme**.

Partitionierung der Beispiele, Fallunterscheidung

- Antiunifizierte Inputs unterscheiden sich an wenigstens einer Position hinsichtlich des Konstruktors.
- Partitionierung der Beispiele anhand des Konstruktors an dieser Position

Example

Beispielmenge:

$$\{1.\ \mathit{Reverse}([]) = [],\ 2.\ \mathit{Reverse}([B]) = [B],\ 3.\ \mathit{Reverse}([A, B]) = [B, A]\}$$

An Wurzelposition kommen die Konstruktoren [] und cons vor, resultierende Partition:

$$\{1\}, \{2, 3\}$$

Programmaufruf

Example

Initiale Hypothese:

$$Reverse([X, Y]) = [Y, X]$$

- Falls ein Output einen anderen Output matcht, kann er durch einen Programmaufruf ersetzt werden.
- Konstruktion des Arguments des Programmaufrufs wird als neues Induktionsproblem behandelt.
- Beispielmenge für neues Problem wird abduziert:
  - Inputs bleiben dieselben.
  - Neue Outputs werden die substituierten *Inputs* der gematchten Outputs.

Programmaufruf, Beispiel

Betrachtetes Beispiel:

$$Reverse([A, B]) = [B, A]$$

Hintergrundwissen:

$$\mathrm{snoc}([X], Y) = [X, Y]$$

$[B, A]$ matcht $[X, Y]$ mit

$$\{X \leftarrow B, Y \leftarrow A\}$$

Output $[B, A]$ kann ersetzt werden:

$$Reverse([A, B]) = \mathrm{snoc}(\mathrm{Sub}_1([A, B]), \mathrm{Sub}_2([A, B]))$$

Abduzierte Beispiele:

$$\mathrm{Sub}_1([A, B]) = [B]$$
$$\mathrm{Sub}_2([A, B]) = A$$

Unterprogramme

Beispiele: $Reverse([B]) = [B], \quad Reverse([A, B]) = [B, A]$
Initiale Hypothese: $Reverse([X|Xs]) = [Y|Ys]$

- Jeder Unterterm der rechten Seite mit einer ungebundenen Variable wird durch den Aufruf eines neuen Unterprogramms ersetzt.
- Konstruktion der Unterprogramme als neue Induktionsprobleme.
- Beispiele für jedes neue Unterprogramm werden abduziert:
  - Inputs bleiben dieselben.
  - Neue Outputs werden die entsprechenden Unterterme der alten Outputs.

Unterprogramme, Beispiel

Beispielgleichungen:

$$Reverse([B]) = [B], \quad Reverse([A, B]) = [B, A]$$

Initiale Hypothese:

$$Reverse([X|Xs]) = [Y|Ys]$$

Rechte Seite wird ersetzt:

$$Reverse([X|Xs]) = [Sub_1([X|Xs])|Sub_2([X|Xs])]$$

Abduzierte Beispiele für die neuen Unterprogramme:

$$Sub_1([B]) = B$$
$$Sub_1([A, B]) = B$$

$$Sub_2([B]) = []$$
$$Sub_2([A, B]) = [A]$$

Gliederung

Pläne

- IGOR II wurde von Emanuel Kitzelmann entwickelt und ist ein wesentlicher Teil seiner Dissertation
- ich bin seit Oktober Doktorand und möchte IGOR II weiterentwickeln

Arbeitspakete:

1. Generischer Spezifikationsgenerator
   - für empirischen Systemvergleich
   - generische Aufzählen von Datentypen und I/Os für Funktionen
   - implementiert mit Template Haskell
2. IGOR II: MAUDE→ HASKELL
3. Erweiterung von IGOR II

Pläne

- IGOR II wurde von Emanuel Kitzelmann entwickelt und ist ein wesentlicher Teil seiner Dissertation
- ich bin seit Oktober Doktorand und möchte IGOR II weiterentwickeln

Arbeitspakete:

1. Generischer Spezifikationsgenerator
2. IGOR II: MAUDE→ HASKELL
   - Erhöhung der Sichtbarkeit
   - Higher-Order Kontext
   - Problem(?): Reflexion in MAUDE zur Laufzeit, in HASKELL zur Compilezeit
3. Erweiterung von IGOR II

Pläne

- IGOR II wurde von Emanuel Kitzelmann entwickelt und ist ein wesentlicher Teil seiner Dissertation
- ich bin seit Oktober Doktorand und möchte IGOR II weiterentwickeln

Arbeitspakete:

1. Generischer Spezifikationsgenerator
2. IGOR II: MAUDE→ HASKELL
3. Erweiterung von IGOR II
   - *let* Ausdrücke einführen
   - Unterprogramme an Wurzelfunktion
   - Higher-Order Funktionen als Schemata
   - Hintergrundwissen aus Instanzinformation nutzen
   - Lücken in Beispielen zulassen
   - zusätzliche Variablen einführen

Gliederung

Mögliche Umsetzung

- let-Ausdrücke

Example (*Multlast* naiv)

$$Multlast([]) = []$$
$$Multlast([A]) = [A]$$
$$Multlast([A, B|C]) =$$
$$\quad [Last([B|C])|Multlast([B|C])]$$

Example (*Multlast* mit *let*)

$$Multlast([]) = []$$
$$Multlast([A]) = [A]$$
$$Multlast([A, B|C]) =$$
$$\quad let\ [D|E] = Multlast([B|C])$$
$$\quad in\ [D, D|E]$$

$\rightarrow$ Ergebnis des einen Aufrufs ist im anderen enthalten

Mögliche Umsetzung

- Unterprogramme an der Wurzelposition

Aber wie soll das gehen?!

$$Sort([]) = [], \quad Sort([1]) = [1], \quad Sort([2]) = [2]$$
$$Sort([1,2]) = [1,2], \quad Sort([2,1]) = [1,2], \quad \ldots$$

$$Sort([]) = []$$
$$Sort([X|Xs]) = Insert(X, Sort(Xs))$$

- Woher I/Os für *Insert*???
- könnten Higher-Order Funktionen teilweise weiterhelfen????

$$Sort(X) = Foldr(Insert, [], X)$$

Mögliche Umsetzung

- Higher-Order Funktionen als Schemata

Nutzen von Domänenwissen

- *Map*, *Filter*, *Reduce*, *Fold*, . . .[a]
- Induktion + Abduktion,
- Explanation Based Learning, Schema als Domain Theory

[a] map, reduce, filter äquivalent zu while-Programmen?

Explizit?

- ILP System DIALOGS verwendet problemspezifische Schemata
- vom Nutzer vorgegeben (divide-and-conquer,. . .)

→ “fit data to schema”

Mögliche Umsetzung

- Higher-Order Funktionen als Schemata

Nutzen von Domänenwissen

- *Map*, *Filter*, *Reduce*, *Fold*, . . .[a]
- Induktion + Abduktion,
- Explanation Based Learning, Schema als Domain Theory

[a] map, reduce, filter äquivalent zu while-Programmen?

Explizit?

- ILP System DIALOGS verwendet problemspezifische Schemata
- vom Nutzer vorgegeben (divide-and-conquer,. . .)

→ “fit data to schema”

Mögliche Umsetzung

- Higher-Order Funktionen als Schemata

Implizit ist besser!

- Ist es möglich “HO-Verdacht” zu erkennen?

$$Fun([]) = [], \quad Fun([A]) = [S(A)], \quad Fun([A, B]) = [S(A), S(B)]$$

$$Fun(X) = Map(Succ, X)$$

- Typ und Instanzinformation nutzen ($[\alpha] \rightarrow [\beta]$, Functor, Foldable)
- Ähnlichkeit auf Termen, ausgehend von ‘Prototyp’
- higher order narrowing ?
  $\rightarrow$“fit schema to data”

Mögliche Umsetzung

- Hintergrundwissen aus Instanzinformation nutzen

Example (instance Eq/Ord Int)

- über Reflexion alle abgeleiteten Typklassen erhalten
  $\text{Int} \rightarrow \text{Eq}, \text{Ord}, \text{Num}...$
- Instanzfunktionen als spezifischen Hintergrundwissen ($=$,$<$,$\leq$,$+$,$-$,...)

- Lücken in Beispielen zulassen
  - Lücken mit *-Output ergänzen
  - Inputs für Lücken generieren und den Nutzer fragen
- zusätzliche Variablen einführen

$$\begin{aligned} reverse(X) &= rev(X, []) \\ rev([], X) &= X \\ rev([X|Xs], Y) &= rev(Xs, [X|Y]) \end{aligned}$$

  - Konstanter Term für initialen Aufruf $rev(X, [])$ ist in allen I/Os enthalten

Mögliche Umsetzung

- Hintergrundwissen aus Instanzinformation nutzen

Example (instance Eq/Ord Int)

- über Reflexion alle abgeleiteten Typklassen erhalten
  $\text{Int} \rightarrow \text{Eq}, \text{Ord}, \text{Num}...$
- Instanzfunktionen als spezifischen Hintergrundwissen ($=$,$<$,$\leq$,$+$,$-$,...)

- Lücken in Beispielen zulassen
  - Lücken mit *-Output ergänzen
  - Inputs für Lücken generieren und den Nutzer fragen
- zusätzliche Variablen einführen

$$\begin{aligned} reverse(X) &= rev(X, []) \\ rev([], X) &= X \\ rev([X|Xs], Y) &= rev(Xs, [X|Y]) \end{aligned}$$

  - Konstanter Term für initialen Aufruf $rev(X, [])$ ist in allen I/Os enthalten

Mögliche Umsetzung

- Hintergrundwissen aus Instanzinformation nutzen

Example (instance Eq/Ord Int)

- über Reflexion alle abgeleiteten Typklassen erhalten
  $\text{Int} \rightarrow \text{Eq}, \text{Ord}, \text{Num}...$
- Instanzfunktionen als spezifischen Hintergrundwissen $(=,<,\leq,+,-,\ldots)$

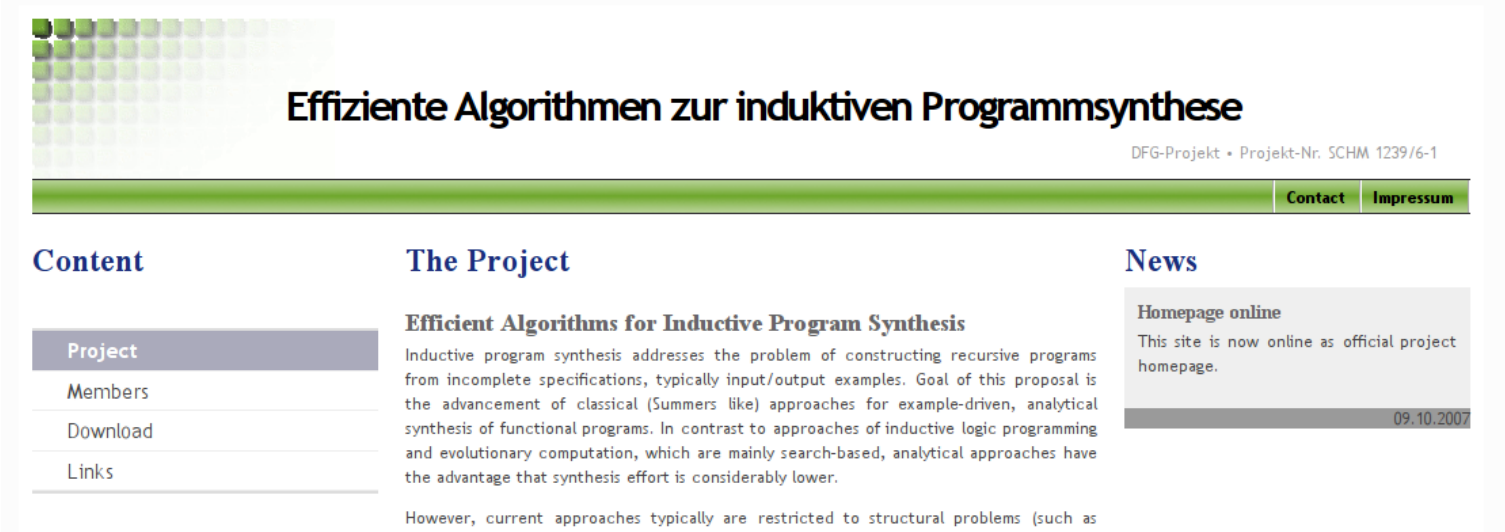
- Lücken in Beispielen zulassen
  - Lücken mit *-Output ergänzen
  - Inputs für Lücken generieren und den Nutzer fragen
- zusätzliche Variablen einführen

$$\begin{aligned} \mathit{reverse}(X) &= \mathit{rev}(X, []) \\ \mathit{rev}([], X) &= X \\ \mathit{rev}([X|Xs], Y) &= \mathit{rev}(Xs, [X|Y] \end{aligned}$$

  - Konstanter Term für initialen Aufruf $\mathit{rev}(X, [])$ ist in allen I/Os enthalten

Gliederung

Unser Projekt



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

- Publikationen
- Downloads
- Links

inductive-programming.org

http://www.inductive-programming.org

(im Aufbau)

- Einführung in IP
- Vorstellung verschiedener IP Systeme
- Repository mit Benchmarkproblemen
- Publikationen mit Schwerpunkt IP
- Mailing List
- . . .

Herzlichen Dank!

Fragen?
Fragen?
Fragen?
Fragen?
Fragen?
Fragen?
Fragen?
Fragen?
Fragen!

MAUDE VS. HASKELL

MAUDE

- interpretiert
- niedrige Abstraktion
- sehr beschränkter HO-Kontext
- Introspektion zur Laufzeit
- umfangreiche Reflexion
- Maude-Sytanx als abstrakter Datentyp mit Infixkonstruktoren in Meta-Maude

HASKELL

- compiliert
- hohe Abstraktion
- umfangreicher HO-Kontext
- Introspektion zur Compilezeit (*compile-time meta/macro programming*)
- Reflexionen nicht vollständig implementiert (type $\rightarrow$ class instances)
- umständliche Q-Monade

Das Inductive Functional Programming Problem formal

Gegeben

- eine Menge syntaktisch korrekter Programme $\mathcal{P}$,
- eine Menge Beispielgleichungen E,
- eine weitere Menge Gleichungen B (Hintergrundwissen),

finde ein Programm (eine Menge Gleichungen) $P \in \mathcal{P}$ so dass

- $P, B \models E$ und
- P über E generalisiert.

Problem: Intendiertheit

- Erinnerung: Erfüllen der *unvollständigen* Spezifikation als *einzige* Korrektheitsbedingung lässt nicht intendierte Funktionen als Lösung zu.
- Offene Frage: Gibt es sinnvolles zusätzliches Kriterium K, so dass die Zielfunktion *vollständig* spezifiziert ist? (So dass also das "induzierte" Programm deduktiv-logisch folgt: $E, B, K \models P$)
- Bisheriger praktischer Behelf: Preference Bias, z.B. "wähle das syntaktisch kleinste Lösungsprogramm".
- Aber (offene Frage): Wie hängen Intention und Preference Biases zusammen?
- Zweites "aber": Exponentielle Komplexität...

Problem: Intendiertheit

- Erinnerung: Erfüllen der *unvollständigen* Spezifikation als *einzige* Korrektheitsbedingung lässt nicht intendierte Funktionen als Lösung zu.
- Offene Frage: Gibt es sinnvolles zusätzliches Kriterium K, so dass die Zielfunktion *vollständig* spezifiziert ist? (So dass also das "induzierte" Programm deduktiv-logisch folgt: $E, B, K \models P$)
- Bisheriger praktischer Behelf: Preference Bias, z.B. "wähle das syntaktisch kleinste Lösungsprogramm".
- Aber (offene Frage): Wie hängen Intention und Preference Biases zusammen?
- Zweites "aber": Exponentielle Komplexität...

Problem: Intendiertheit

- Erinnerung: Erfüllen der *unvollständigen* Spezifikation als *einzige* Korrektheitsbedingung lässt nicht intendierte Funktionen als Lösung zu.
- Offene Frage: Gibt es sinnvolles zusätzliches Kriterium K, so dass die Zielfunktion *vollständig* spezifiziert ist? (So dass also das "induzierte" Programm deduktiv-logisch folgt: $E, B, K \models P$)
- Bisheriger praktischer Behelf: Preference Bias, z.B. "wähle das syntaktisch kleinste Lösungsprogramm".
- Aber (offene Frage): Wie hängen Intention und Preference Biases zusammen?
- Zweites "aber": Exponentielle Komplexität...

Problem: Intendiertheit

- Erinnerung: Erfüllen der *unvollständigen* Spezifikation als *einzige* Korrektheitsbedingung lässt nicht intendierte Funktionen als Lösung zu.
- Offene Frage: Gibt es sinnvolles zusätzliches Kriterium K, so dass die Zielfunktion *vollständig* spezifiziert ist? (So dass also das "induzierte" Programm deduktiv-logisch folgt: $E, B, K \models P$)
- Bisheriger praktischer Behelf: Preference Bias, z.B. "wähle das syntaktisch kleinste Lösungsprogramm".
- Aber (offene Frage): Wie hängen Intention und Preference Biases zusammen?
- Zweites "aber": Exponentielle Komplexität...

Problem: Exponentielle Komplexität

- Finden der hinsichtlich des Preference Bias *besten* Lösung erfordert exponentiellen Aufwand, zumindest mit heutigen IP-Algorithmen.
- Offene Frage: Ist das notwendig? Ist das IP-Problem mit gemäß Preference Bias exakter Lösung NP-vollständig?
- Falls ja: Tun es auch nicht-beste Lösungen? Gibt es brauchbare Heuristiken?
- Und: Falls es ein zusätzliches allgemeines Korrektheitskriterium gibt, wie sieht es dann mit der Komplexität aus?

Problem: Exponentielle Komplexität

- Finden der hinsichtlich des Preference Bias *besten* Lösung erfordert exponentiellen Aufwand, zumindest mit heutigen IP-Algorithmen.
- Offene Frage: Ist das notwendig? Ist das IP-Problem mit gemäß Preference Bias exakter Lösung NP-vollständig?
- Falls ja: Tun es auch nicht-beste Lösungen? Gibt es brauchbare Heuristiken?
- Und: Falls es ein zusätzliches allgemeines Korrektheitskriterium gibt, wie sieht es dann mit der Komplexität aus?

Problem: Exponentielle Komplexität

- Finden der hinsichtlich des Preference Bias *besten* Lösung erfordert exponentiellen Aufwand, zumindest mit heutigen IP-Algorithmen.
- Offene Frage: Ist das notwendig? Ist das IP-Problem mit gemäß Preference Bias exakter Lösung NP-vollständig?
- Falls ja: Tun es auch nicht-beste Lösungen? Gibt es brauchbare Heuristiken?
- Und: Falls es ein zusätzliches allgemeines Korrektheitskriterium gibt, wie sieht es dann mit der Komplexität aus?

Problem: Exponentielle Komplexität

- Finden der hinsichtlich des Preference Bias *besten* Lösung erfordert exponentiellen Aufwand, zumindest mit heutigen IP-Algorithmen.
- Offene Frage: Ist das notwendig? Ist das IP-Problem mit gemäß Preference Bias exakter Lösung NP-vollständig?
- Falls ja: Tun es auch nicht-beste Lösungen? Gibt es brauchbare Heuristiken?
- Und: Falls es ein zusätzliches allgemeines Korrektheitskriterium gibt, wie sieht es dann mit der Komplexität aus?