

Inductive Programming

Ein Überblick und der IGOR II-Algorithmus

Emanuel Kitzelmann

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

Gruppe Prof. Krieg-Brückner, Bremen 2008

Gliederung

1 Inductive Programming

- Überblick

2 Verwandte Forschungsfelder

- Verwandte Forschungsfelder
- Mögliche Anwendungsfelder

3 IGOR II

- Überblick
- Algorithmus
- Empirie

4 Offene Fragen

- Offene Fragen

Gliederung

1 Inductive Programming

- Überblick

2 Verwandte Forschungsfelder

- Verwandte Forschungsfelder
- Mögliche Anwendungsfelder

3 IGOR II

- Überblick
- Algorithmus
- Empirie

4 Offene Fragen

- Offene Fragen

Inductive Programming (IP)

Inductive Programming (IP) erforscht das automatische Konstruieren von Algorithmen und Programmen aus *unvollständigen* Spezifikationen, z.B. aus Input/Output-Beispielen (I/O-Beispiele).

Example (*Reverse*)

I/O-Beispiele:

$$\text{Reverse}([]) = [], \text{Reverse}([a]) = [a], \text{Reverse}([a, b]) = [b, a], \dots$$

Induziertes funktionales Programm:

$$\text{Reverse}([]) = []$$

$$\text{Reverse}(\text{cons}(X, Xs)) = \text{Reverse}(Xs) @ \text{cons}(X, [])$$

Elementares Charakteristikum: Es erfüllen verschiedene auch *semantisch* verschiedene Programme die Spezifikation.

Verschiedene Ansätze

Funktional analytisch

- Es werden funktionale Programme induziert.
- 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 (Schmied, Mühlpfordt, Kitzelmann)

Evolutionär heuristisch

Induktives Logisches Programmieren (ILP)

Verschiedene Ansätze

Funktional analytisch

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

Funktional analytisch

Evolutionär heuristisch

Induktives Logisches Programmieren (ILP)

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

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).

Was ist heute möglich?

Automatisch benötigte Unterprogramme finden und einbinden.

Example (*Reverse*)

$$\text{Reverse}([]) = []$$

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

$$\text{Last}([X]) = X$$

$$\text{Last}([X, Y|Xs]) = \text{Last}([Y|Xs])$$

$$\text{Init}([X]) = []$$

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

Was ist heute möglich?

Komplexe Fallunterscheidungen.

Example (*Sum*)

$$\textit{Sum}([]) = 0$$

$$\textit{Sum}([0|Xs]) = \textit{Sum}(Xs)$$

$$\textit{Sum}([s(X)|Xs]) = s(\textit{Sum}([X|Xs]))$$

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*)

$$\text{Odd}(0) = f$$

$$\text{Odd}(s(X)) = \text{Even}(X)$$

$$\text{Even}(0) = t$$

$$\text{Even}(s(X)) = \text{Odd}(X)$$

Was ist heute möglich?

Komplexe Funktionsaufruf-Relationen.

Example (*QuickSort*)

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

Begriffe

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

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

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

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

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

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

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

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

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

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

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

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

- 1 Inductive Programming
 - Überblick
- 2 Verwandte Forschungsfelder
 - Verwandte Forschungsfelder
 - Mögliche Anwendungsfelder
- 3 IGOR II
 - Überblick
 - Algorithmus
 - Empirie
- 4 Offene Fragen
 - Offene Fragen

Verwandte Forschungsfelder

- **Algorithmik**
- Softwaretechnik
- Deduktive Programmsynthese
 - ▶ Konstruktion von Programmen aus vollständigen Spezifikationen
- Maschinelles Lernen
 - ▶ Induzieren von intensionalen Konzeptbeschreibungen aus Beispielen und Gegenbeispielen
- Inductive Inference und Computational Learning Theory
 - ▶ Welche Konzepte sind unter welchen Bedingungen lernbar?
- End-user Programming

Verwandte Forschungsfelder

- **Algorithmik**
- **Softwaretechnik**
- Deduktive Programmsynthese
 - ▶ Konstruktion von Programmen aus vollständigen Spezifikationen
- Maschinelles Lernen
 - ▶ Induzieren von intensionalen Konzeptbeschreibungen aus Beispielen und Gegenbeispielen
- Inductive Inference und Computational Learning Theory
 - ▶ Welche Konzepte sind unter welchen Bedingungen lernbar?
- End-user Programming

Verwandte Forschungsfelder

- Algorithmik
- Softwaretechnik
- Deduktive Programmsynthese
 - ▶ Konstruktion von Programmen aus vollständigen Spezifikationen
- Maschinelles Lernen
 - ▶ Induzieren von intensionalen Konzeptbeschreibungen aus Beispielen und Gegenbeispielen
- Inductive Inference und Computational Learning Theory
 - ▶ Welche Konzepte sind unter welchen Bedingungen lernbar?
- End-user Programming

Verwandte Forschungsfelder

- Algorithmik
- Softwaretechnik
- Deduktive Programmsynthese
 - ▶ Konstruktion von Programmen aus vollständigen Spezifikationen
- Maschinelles Lernen
 - ▶ Induzieren von intensionalen Konzeptbeschreibungen aus Beispielen und Gegenbeispielen
- Inductive Inference und Computational Learning Theory
 - ▶ Welche Konzepte sind unter welchen Bedingungen lernbar?
- End-user Programming

Verwandte Forschungsfelder

- Algorithmik
- Softwaretechnik
- Deduktive Programmsynthese
 - ▶ Konstruktion von Programmen aus vollständigen Spezifikationen
- Maschinelles Lernen
 - ▶ Induzieren von intensionalen Konzeptbeschreibungen aus Beispielen und Gegenbeispielen
- Inductive Inference und Computational Learning Theory
 - ▶ Welche Konzepte sind unter welchen Bedingungen lernbar?
- End-user Programming

Verwandte Forschungsfelder

- Algorithmik
- Softwaretechnik
- Deduktive Programmsynthese
 - ▶ Konstruktion von Programmen aus vollständigen Spezifikationen
- Maschinelles Lernen
 - ▶ Induzieren von intensionalen Konzeptbeschreibungen aus Beispielen und Gegenbeispielen
- Inductive Inference und Computational Learning Theory
 - ▶ Welche Konzepte sind unter welchen Bedingungen lernbar?
- End-user Programming

Abstrakte Gemeinsamkeit

Lernen einer Funktion aus Beispielen bzw. unvollständigem Wissen.

Differenzen:

- Art der Funktion: ML: nicht-rekursiv, diskreter oder numerischer Output; IP: rekursiv, rekursiv konstruierte Outputs.
- Art der Trainingsdaten: ML: Große Menge an i.d.R. verrauschten, zufällig gewählten (Gegen-)Beispielen; IP: Kleine Menge an korrekten, ausgesuchten Beispielen, evtl. zusätzliches Wissen.
- Anspruch an die gelernte Funktion: ML: Approximation; IP: i.d.R. exakte Identifikation der Zielfunktion.

Gliederung

- 1 Inductive Programming
 - Überblick
- 2 Verwandte Forschungsfelder
 - Verwandte Forschungsfelder
 - **Mögliche Anwendungsfelder**
- 3 IGOR II
 - Überblick
 - Algorithmus
 - Empirie
- 4 Offene Fragen
 - Offene Fragen

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.

Algebraische Spezifikation

Algebraische Spezifikationen als Verallgemeinerung funktionaler Programme betrachten, (möglicherweise) mit ähnlichen Techniken aus Beispielgleichungen induzierbar.

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.

Algebraische Spezifikation

Algebraische Spezifikationen als Verallgemeinerung funktionaler Programme betrachten, (möglicherweise) mit ähnlichen Techniken aus Beispielgleichungen induzierbar.

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.

Algebraische Spezifikation

Algebraische Spezifikationen als Verallgemeinerung funktionaler Programme betrachten, (möglicherweise) mit ähnlichen Techniken aus Beispielgleichungen induzierbar.

Testen/Debuggen

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

Gliederung

- 1 Inductive Programming
 - Überblick
- 2 Verwandte Forschungsfelder
 - Verwandte Forschungsfelder
 - Mögliche Anwendungsfelder
- 3 **IGOR II**
 - **Überblick**
 - Algorithmus
 - Empirie
- 4 Offene Fragen
 - Offene Fragen

Entwicklung I

- IGOR I und II basieren auf 2-schrittigem Ansatz von Summers:

- ① **Prädikate** zum Unterscheiden der Inputs und **Traces** zum Berechnen der Outputs aus den Inputs werden *berechnet*.
- ② Zwischen Traces und Prädikaten wird **Rekurrenzbeziehung** ermittelt, die als Basis der rekursiven Funktionsdefinition dient.

Nur ein linear-rekursiver Aufruf möglich, keine Unterprogramme.

- Schmid und Mühlpfordt generalisieren 2. Teil auf **Terme**:
 - ▶ Gegebener Term wird als *ktes* Element in Kleene-Kette einer unbekannten rekursiven Gleichungsmenge betrachtet.
 - ▶ Dieser wird nach Rekurrenzen durchsucht und darauf basierend in eine entsprechende Gleichungsmenge **gefaltet**. *Induktion als Umkehrung des **Entfaltens** einer Funktion.*
- Kitzelmann entwickelt Algorithmus zum Berechnen eines solchen **initialen Terms** aus I/O-Beispielen. Zusammen: IGOR I.
- IGOR I findet benötigte Unterprogramme und geht über lineare Rekursion hinaus, vernetzte Funktionsaufrufe aber nicht möglich.

Entwicklung I

- IGOR I und II basieren auf 2-schrittigem Ansatz von Summers:

- ① **Prädikate** zum Unterscheiden der Inputs und **Traces** zum Berechnen der Outputs aus den Inputs werden *berechnet*.
- ② Zwischen Traces und Prädikaten wird **Rekurrenzbeziehung** ermittelt, die als Basis der rekursiven Funktionsdefinition dient.

Nur ein linear-rekursiver Aufruf möglich, keine Unterprogramme.

- Schmid und Mühlpfordt generalisieren 2. Teil auf **Terme**:
 - ▶ Gegebener Term wird als *ktes* Element in Kleene-Kette einer unbekannten rekursiven Gleichungsmenge betrachtet.
 - ▶ Dieser wird nach Rekurrenzen durchsucht und darauf basierend in eine entsprechende Gleichungsmenge **gefaltet**. *Induktion als Umkehrung des **Entfaltens** einer Funktion.*
- Kitzelmann entwickelt Algorithmus zum Berechnen eines solchen **initialen Terms** aus I/O-Beispielen. Zusammen: IGOR I.
- IGOR I findet benötigte Unterprogramme und geht über lineare Rekursion hinaus, vernetzte Funktionsaufrufe aber nicht möglich.

Entwicklung I

- IGOR I und II basieren auf 2-schrittigem Ansatz von Summers:

- ① **Prädikate** zum Unterscheiden der Inputs und **Traces** zum Berechnen der Outputs aus den Inputs werden *berechnet*.
- ② Zwischen Traces und Prädikaten wird **Rekurrenzbeziehung** ermittelt, die als Basis der rekursiven Funktionsdefinition dient.

Nur ein linear-rekursiver Aufruf möglich, keine Unterprogramme.

- Schmid und Mühlpfordt generalisieren 2. Teil auf **Terme**:
 - ▶ Gegebener Term wird als *ktes* Element in Kleene-Kette einer unbekannten rekursiven Gleichungsmenge betrachtet.
 - ▶ Dieser wird nach Rekurrenzen durchsucht und darauf basierend in eine entsprechende Gleichungsmenge **gefaltet**. *Induktion als Umkehrung des **Entfaltens** einer Funktion.*
- Kitzelmann entwickelt Algorithmus zum Berechnen eines solchen **initialen Terms** aus I/O-Beispielen. Zusammen: IGOR I.
- IGOR I findet benötigte Unterprogramme und geht über lineare Rekursion hinaus, vernestete Funktionsaufrufe aber nicht möglich.

Entwicklung I

- IGOR I und II basieren auf 2-schrittigem Ansatz von Summers:

- ① **Prädikate** zum Unterscheiden der Inputs und **Traces** zum Berechnen der Outputs aus den Inputs werden *berechnet*.
- ② Zwischen Traces und Prädikaten wird **Rekurrenzbeziehung** ermittelt, die als Basis der rekursiven Funktionsdefinition dient.

Nur ein linear-rekursiver Aufruf möglich, keine Unterprogramme.

- Schmid und Mühlpfordt generalisieren 2. Teil auf **Terme**:
 - ▶ Gegebener Term wird als *ktes* Element in Kleene-Kette einer unbekannten rekursiven Gleichungsmenge betrachtet.
 - ▶ Dieser wird nach Rekurrenzen durchsucht und darauf basierend in eine entsprechende Gleichungsmenge **gefaltet**. *Induktion als Umkehrung des **Entfaltens** einer Funktion.*
- Kitzelmann entwickelt Algorithmus zum Berechnen eines solchen **initialen Terms** aus I/O-Beispielen. Zusammen: IGOR I.
- IGOR I findet benötigte Unterprogramme und geht über lineare Rekursion hinaus, vernetzte Funktionsaufrufe aber nicht möglich.

- Grundidee bei Summers und IGOR I ist, *Suche* weitestgehend zu *vermeiden*.
- Deshalb starke syntaktische Constraints (nur lineare Rekursion, keine Verneftung) und kein Hintergrundwissen nutzbar.
- Erweiterung nur auf Basis von Suche möglich. Das führt zu Redundanzen beim 2-schrittigen Ansatz.

Entwicklung II

- IGOR II (Kitzelmann):

- ▶ Grundlegende Idee, Rekursion aufgrund von Regularitäten in den Beispielen zu finden, bleibt bestehen.
- ▶ 2-schrittiger Ansatz wird verworfen, Beispiele werden direkt verglichen, **eine integrierte Suche**.
- ▶ Weitere Vereinfachung und Transparenz durch Ausnutzen von **Pattern Matching** (konnte LISP noch nicht!)
- ▶ Beliebige nutzerdefinierte Datentypen.
- ▶ **Hintergrundwissen** nutzbar,
- ▶ komplexere Aufruf-Beziehungen (Baumrekursion, Vernetzung) möglich,
- ▶ Beispielgleichungen können Variablen enthalten,
- ▶ mehrere (abhängige) Zielfunktionen (*Even/Odd*) können parallel induziert werden.

- IGOR II ist mein Algorithmus und System, aktueller Stand der Dinge und ein wesentlicher Teil meiner Doktorarbeit.

Entwicklung II

- IGOR II (Kitzelmann):

- ▶ Grundlegende Idee, Rekursion aufgrund von Regularitäten in den Beispielen zu finden, bleibt bestehen.
- ▶ 2-schrittiger Ansatz wird verworfen, Beispiele werden direkt verglichen, **eine integrierte Suche**.
- ▶ Weitere Vereinfachung und Transparenz durch Ausnutzen von **Pattern Matching** (konnte LISP noch nicht!)
- ▶ Beliebige nutzerdefinierte Datentypen.
- ▶ **Hintergrundwissen** nutzbar,
- ▶ komplexere Aufruf-Beziehungen (Baumrekursion, Vernetzung) möglich,
- ▶ Beispielgleichungen können Variablen enthalten,
- ▶ mehrere (abhängige) Zielfunktionen (*Even/Odd*) können parallel induziert werden.

- IGOR II ist mein Algorithmus und System, aktueller Stand der Dinge und ein wesentlicher Teil meiner Doktorarbeit.

- Datentyp Definition, z.B.

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

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

- ▶ Zielfunktion:

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

$$\text{Reverse}([]) = [], \text{Reverse}([X]) = [X], \text{Reverse}([X, Y]) = [Y, X], \dots$$

- ▶ Hintergrundwissen:

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

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

- Datentyp Definition, z.B.

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

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

- ▶ Zielfunktion:

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

$$\text{Reverse}([]) = [], \text{Reverse}([X]) = [X], \text{Reverse}([X, Y]) = [Y, X], \dots$$

- ▶ Hintergrundwissen:

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

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

Menge an Gleichungen die die gegebenen Beispielgleichungen modellieren.

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

Reverse-Beispiel:

$$\text{Reverse}([]) = []$$

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

Gliederung

- 1 Inductive Programming
 - Überblick
- 2 Verwandte Forschungsfelder
 - Verwandte Forschungsfelder
 - Mögliche Anwendungsfelder
- 3 **IGOR II**
 - Überblick
 - **Algorithmus**
 - Empirie
- 4 Offene Fragen
 - Offene Fragen

Initiale Hypothese

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

Example

Beispielgleichungen:

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

Initiale Hypothese:

$$\text{Reverse}([X|Xs]) = [Y|Ys]$$

Initiale Hypothese weiterverarbeiten

Example

Initiale Hypothese:

$$\text{Reverse}([X|Xs]) = [Y|Ys]$$

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

Drei Lösungsmöglichkeiten:

- 1 **Partitionierung** der Beispielformeln
 $\rightsquigarrow$ 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:

$$\text{Reverse}([X|Xs]) = [Y|Ys]$$

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

Drei Lösungsmöglichkeiten:

- 1 **Partitionierung** der Beispielformeln
 $\rightsquigarrow$ 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:

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

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

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

Programmaufruf

- 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:

$$\text{Reverse}([A, B]) = [B, A]$$

Hintergrundwissen:

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

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

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

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

$$\text{Reverse}([A, B]) = \text{snoc}(\text{Sub1}([A, B]), \text{Sub2}([A, B]))$$

Abduziertes Beispiel:

$$\text{Sub1}([A, B]) = [B]$$

Unterprogramme

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

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:

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

Initiale Hypothese:

$$(\text{Reverse}([X|Xs]) = [Y|Ys])$$

Rechte Seite wird ersetzt:

$$\text{Reverse}([X|Xs]) = [\text{Sub}_1([X|Xs])|\text{Sub}_2([X|Xs])]$$

Abduzierte Beispiele für die neuen Unterprogramme:

$$\text{Sub}_1([B]) = B$$

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

$$\text{Sub}_2([B]) = []$$

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

Gliederung

- 1 Inductive Programming
 - Überblick
- 2 Verwandte Forschungsfelder
 - Verwandte Forschungsfelder
 - Mögliche Anwendungsfelder
- 3 IGOR II
 - Überblick
 - Algorithmus
 - **Empirie**
- 4 Offene Fragen
 - Offene Fragen

- *Length*, *Last*, *Reverse*, *Member*, *Take* (behält n Elemente und löscht die restlichen), *Insertion Sort* (mit *Insert* als Hintergrundwissen) *Even/Odd*, *Add*, *Mirror* (spiegelt einen Binärbaum) werden in Millisekunden aus ≤ 6 Beispielen induziert, (*Member* 13 Beispiele).
- *Reverse* in zwei Varianten
 - 1 ohne Hintergrundwissen; Unterprogramme *Last* und *Init* werden automatisch gefunden,
 - 2 mit `snoc` als Hintergrundwissen.
- *Quicksort* mit *Append* und Partitionierungsfunktionen in 63 Sekunden.

Gliederung

1 Inductive Programming

- Überblick

2 Verwandte Forschungsfelder

- Verwandte Forschungsfelder
- Mögliche Anwendungsfelder

3 IGOR II

- Überblick
- Algorithmus
- Empirie

4 Offene Fragen

- Offene Fragen

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 *einzig* 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?