Ein kombinierter analytischer und suchbasierter Ansatz zur induktiven Synthese funktionaler Programme

Emanuel Kitzelmann*

emanuel@icsi.berkeley.edu

Abstract: Induktive Programmsynthese beschäftigt sich mit der automatisierten Konstruktion von Computer-Programmen auf Basis von unvollständigen Spezifikationen wie z.B. Eingabe/Ausgabe-Beispielen. Es lassen sich zwei komplementäre Ansätze unterscheiden: Im effizienten aber eingeschränkten *analytischen* Ansatz wird eine rekursive Funktionsdefinition generiert, indem rekurrente Strukturen zwischen den einzelnen Beispielen gefunden und generalisiert werden. Im mächtigeren aber ineffizienten *erzeuge-und-teste* Ansatz werden unabhängig von den bereitgestellten Beispielen solange Programme einer Klasse generiert, bis ein Programm gefunden wurde das alle Beispiele korrekt berechnet. Hauptbeitrag dieser Arbeit ist der neue Algorithmus IGOR2 zur induktiven Synthese funktionaler Programme, der den analytischen Ansatz generalisiert und mit Suche in einem Programmraum kombiniert, um einen guten Kompromiss zwischen Expressivität und Effizienz zu erreichen. IGOR2 ist terminierend und garantiert Korrektheit synthetisierter Programme bzgl. gegebener Beispiele. Experimente zeigen, dass IGOR2 nicht-triviale Programme in verschiedenen Domänen induzieren kann und meist effizienter ist als andere vergleichbare Systeme.

1 Einleitung und Motivation

Induktive Programmsynthese oder *Induktives Programmieren (IP)* bezeichnet das automatisierte Generieren von Programmen aus unvollständigen Spezifikationen wie z.B. Eingabe/Ausgabe-(E/A-)Beispielen. Seit kurzem wird IP in steigendem Maße praktisch angewandt und hat das Potential, verschiedene Anwendungsgebiete stark zu beeinflussen. Einige Beispiele: Universelle oder Mehrzweck-Computer in verschiedenen Formen – Desktops, Smartphones, Netbooks etc. – erobern in steigendem Maße den Alltag von vielen Menschen. Für optimale Produktivität muss ihre Software auf intuitive Weise individuell anpassbar und in ihrer Funktionalität erweiterbar sein. Endnutzer haben üblicherweise jedoch weder Programmier- noch Spezifikations-Kenntnisse, sondern drücken ihre Erwartungen intuitiv durch *Beispiele* aus. In [Gul11] wird ein System zur *Endnutzer-Programmierung mittels IP* beschrieben, das in Echtzeit auf Basis von wenigen Beispielen leistungsfähige String-Transformationen lernt und als interaktives Add-in für die Tabellenkalkulation MS Excel realisiert ist. In [SK11] zeigen Ute Schmid und ich, wie IP und speziell der im Folgenden vorgestellte Algorithmus IGOR2 in der *Kognitiven Modellierung* verwendet werden kann, um das Lernen von produktivem Problemlöse-Wissen in Form rekursiver Regeln

*Der Autor wurde während des Verfassens dieses Artikels im Rahmen des DAAD-FIT-Programms gefördert.

zu modellieren. Dies ist eine wichtige Voraussetzung für zukünftige intelligente Agenten, die sich autonom an neue Situationen und Problemstellungen anpassen können müssen. Für *domäne-konfigurierbare Planungssysteme* (z.B. [NAI+03]), die mit Domänewissen in Form von *Hierarchischen Task-Netzwerken (HTNs)* – eine Form nicht-deterministischer Programme – ausgestattet werden und in praktischen Anwendungen auf Grund höherer Effizienz gegenüber domäne-unabhängigen Planern bevorzugt werden, muss das Domänewissen bislang zeitaufwendig und kostspielig von Hand erstellt werden. In [NLK06] wird eine Technik beschrieben, wie solches programmförmiges Domänewissen automatisch generiert werden kann. Das evolutionäre IP-System ADATE "erfindet" und optimiert Algorithmen [Ols95]. In der testgetriebenen *Software-Entwicklung* schließlich, wo Testfälle, oft in Form von E/A-Beispielen, der Ausgangspunkt der Implementierung sind, könnte IP eingesetzt werden, um automatisch passenden prototypischen Code zu erzeugen.

Wir befassen uns im Folgenden mit der induktiven Programmierung rekursiver funktionaler Programme. Hier lassen sich bisherige Systeme gemäß zweier Ansätze klassifizieren: Im *analytischen* Ansatz [Sum77, KS06] werden syntaktische Regularitäten zwischen gegebenen Beispielen bestimmt und dann zu einer rekursiven Funktionsdefinition verallgemeinert. Dies ist effizient, setzt aber (i) stark eingeschränkte Formen synthetisierbarer Programme sowie (ii) Beispielmengen, die bis zu einer bestimmten Komplexität vollständig sind voraus. Im *erzeuge-und-teste (E&T)* Ansatz [Ols95, Kat07] werden hingegen wiederholt Programme einer definierten Klasse unabhängig von den gegebenen Beispielen generiert und dann gegen die Beispiele getestet, bis ein korrektes Programm gefunden wurde. Dieser Ansatz überwindet die starken Restriktionen des analytischen Ansatzes, allerdings zu dem Preis einer kombinatorischen Suche in riesigen Programmräumen.[1]

Der im Folgenden beschriebene Algorithmus IGOR2 [Kit10] kombiniert beide Ansätze, indem einerseits in einem relativ unbeschränkten Programmraum *gesucht* wird, die Programmkandidaten allerdings nicht unabhängig von den gegebenen Beispielen, sondern *analytisch* generiert werden, wodurch viele mit den Beispielen inkompatible Programme gar nicht konstruiert und betrachtet werden. IGOR2 kann Hintergrundwissen in Form von vordefinierten (Hilfs-)Funktionen verwenden und automatisch neue rekursive Hilfsfunktionen einführen. Es werden komplexe rekursive Muster wie das der Ackermann-Funktion oder der wechselseitig-rekursiven Definition von odd/even gefunden. Experimente zeigen, dass IGOR2 nicht-triviale Programme in unterschiedlichen Domänen effizient synthetisieren kann. IGOR2 synthetisiert Programme, die über die Möglichkeiten bisheriger analytischer Systeme hinaus gehen und dies oft effizienter als E&T-Systeme.

2 Der IGOR2-Algorithmus

Funktionen, die synthetisiert werden sollen, nennen wir *Zielfunktionen*, während Funktionen, die bereits implementiert sind und als Hilfsfunktionen verwendet werden können, *Hintergrundfunktionen* genannt werden.

[1] In Kapitel 3 meiner Dissertation [Kit10] sind diese zwei Ansätze und ihre Methoden sowie auch induktive *logische* Programmierung (ILP) ausführlich dargestellt und im Zusammenhang diskutiert.

2.1 Konstruktorsysteme als Objektsprache

Die kurz eingeführten Termersetzungs-Konzepte sind detailliert z.B. in [BN99] behandelt.

Spezifikationen von Ziel- und Hintergrundfunktionen sowie synthetisierte Zielfunktionen werden als *orthogonale Konstruktor-Termersetzungs-Systeme* (*Konstruktor-Systeme, KSe*) repräsentiert. Ein KS ist eine Menge von Ersetzungsregeln über einer algebraischen Signatur (Funktionssymbole) und Variablen, wobei die Signatur in *definierte Funktionen* $\mathcal{D}$ und *Konstruktoren* $\mathcal{C}$ partitioniert ist und die Regeln die Form $f(p_1, \ldots, p_n) \to t$ haben; dabei ist $f \in \mathcal{D}$ eine definierte Funktion, die p_i sind *Konstruktorterme*, bestehen also nur aus Konstruktoren und Variablen, und Variablen der rechten Seite t (Körper) sind auch in der linken Seite $f(p_1, \ldots, p_n)$ (Kopf) enthalten ($Var(t) \subseteq Var(f(p_1, \ldots, p_n))$). Die p_i heißen *Muster*. Wir schreiben Term-Sequenzen wie $p_1, \ldots, p_n$ auch kurz als $\boldsymbol{p}$.

Ein KS ist *orthogonal*, wenn jeder Kopf jede Variable maximal einmal enthält und keine zwei Köpfe unifizieren. Zwei Terme t_1, t_2 unifizieren, wenn es eine Substitution (von Variablen durch Terme) σ gibt, so dass $t_1\sigma = t_2\sigma$ ist. Orthogonale KSe sind eine einfache Form funktionaler Programme (ohne Funktionen höherer Ordnung). Ein Eingabeterm wird ausgewertet, indem wiederholt Subterme mit Funktionsköpfen gematcht werden, was jeweils zu einer Substitution σ führt, und die Subterme durch die entsprechenden Körper, substituiert mit σ, ersetzt werden. Orthogonalität garantiert *eindeutige Normalformen*. Die nicht-unifizierenden Muster in den Regelköpfen einer definierten Funktion fungieren zum einen als *Bedingungen* für die Anwendung der entsprechenden Regeln und zum anderen dekomponieren sie einen gematchten Term in Subterme, gebunden an die Mustervariablen. Dieses Konzept heißt *Musterabgleich*.

2.2 Syntheseproblem, Suchstrategie und Induktives Bias

Spezifikationen von Ziel- und Hintergrundfunktionen sind orthogonale KSe mit der Restriktion, dass die Körper Konstruktorterme sind (keine Funktionsaufrufe enthalten). Spezifikationsregeln *ohne* Variablen bezeichnen E/A-Beispiele, während eine Spezifikationsregel *mit* Variablen die *Menge* aller ihrer Instanzen als E/A-Beispiele repräsentiert. Wir nennen beide Sorten Spezifikationsregeln schlicht (E/A-)Beispiele oder Beispielregeln und deren linke bzw. rechte Seiten (Beispiel-)Eingaben bzw. Ausgaben.

Definition 1 (Syntheseproblem)**.** Gegeben zwei Spezifikationen Φ und B mit disjunkten definierten Funktionen, $\mathcal{D}_\Phi \cap \mathcal{D}_B = \emptyset$, genannt *Zielfunktionen* und *Hintergrundfunktionen*, finde ein KS P mit definierten Funktionen $\mathcal{D}_P \supseteq \mathcal{D}_\Phi$, so dass

1. P orthogonal ist,
2. P keine Hintergrundfunktionen (neu) definiert ($\mathcal{D}_P \cap \mathcal{D}_B = \emptyset$), und
3. $P \cup B$ für alle $f(\boldsymbol{i}) \to o \in \Phi$, $f(\boldsymbol{i})$ zu o auswertet.

Als durchgehendes Beispiel nehmen wir die Funktion reverse zum Umkehren einer Liste. Wir repräsentieren Listen algebraisch mittels der Konstruktoren [] (leere Liste) und _:_

Listing 1: Spezifikation (vier Beispiele) für die reverse-Funktion

```
1 reverse ([], ys)                 → ys
2 reverse (x1 : [], ys)            → x1 : ys
3 reverse (x1 : x2 : [], ys)       → x2 : x1 : ys
4 reverse (x1 : x2 : x3 : [], ys)  → x3 : x2 : x1 : ys
```

Listing 2: Synthetisierte Implementation der reverse-Funktion

```
reverse ([], ys)      → ys
reverse (x : xs, ys)  → reverse (xs, x : ys)
```

(ein Element, linkes Argument, vorne an eine Liste, rechtes Argument, anfügen). Wir spezifizieren reverse mit einem zusätzlichen (Akkumulator-)Parameter, dessen Listenwert an die umgekehrte Liste angehängt wird.[2] Listing 1 zeigt vier Beispiele (wir spezifizieren keine Hintergrundfunktionen). IGOR2 synthetisiert die korrekte Lösung in Listing 2.

Die induktive Synthese eines Lösungs-KSs erfolgt per *uniformer Kostensuche* in einem Raum orthogonaler KSe, wobei $Var(r) \subseteq Var(l)$ für Regeln $l \rightarrow r$ (s. Abschnitt 2.1) *nicht* gelten muss. Wir nennen solche KSe und die entsprechenden Regeln und Körper *offen* und reguläre KSe (die die Eigenschaft erfüllen) *geschlossen*. Offene KSe haben keine eindeutigen Normalformen, repräsentieren also keine Funktionen. Ziel der Such-(Synthese-) operatoren ist es, geschlossene KSe zu erreichen. IGOR2s Syntheseoperatoren (Abschnitt 2.3) stellen sicher, dass alle konstruierten KS-Kandidaten die Bedingungen aus Def. 1 erfüllen. Jedes *geschlossene* KS ist daher eine Lösung. Die *Kosten* eines KSs sind definiert als die Anzahl disjunkter Muster in seinen Köpfen. "Günstigere" KSe berechnen die Beispiele also mit weniger Fallunterscheidungen und werden bevorzugt (*induktives Bias* [Mit80]). Das initiale Kandidaten-KS besteht aus einer Regel pro Zielfunktion.

Als initiale Hypothese einer Menge von Beispielregeln nimmt IGOR2 ihre *speziellste Generalisierung (least general generalization, LGG)* [Plo70]. D.h., falls alle Regeln dasselbe Symbol an einer bestimmten Position aufweisen, wird dieses Symbol behalten; falls die Symbole variieren, wird eine Variable eingefügt. *Dieselbe* Variable wird an zwei unterschiedlichen Positionen eingeführt, sofern die entsprechenden Subterme in den Beispielregeln an beiden Positionen dieselben sind. Der LGG für die Beispielregeln aus Listing 1 ist reverse(xs, ys) → zs. Die beiden Variablen xs, zs resultieren aus den unterschiedlichen Symbolen [] bzw. ys (1. Beispiel) und _:_ (Beisp. 2, 3, 4) an den entsprechenden Positionen. Diese initiale Regel ist *offen*, da die Variable zs nicht im Kopf erscheint.

[2] Man mag einwenden, dass wir mit dieser Art Spezifikation bereits eine bestimmte Lösung vorgeben. Das ist einerseits wahr und auch nicht unbedingt verkehrt: Grundsätzlich ist es sinnvoll, den Benutzer soviel Information wie möglich nicht nur über die Funktion im mathematischen Sinne, sondern auch über die Form der Implementation (z.B. ein bestimmtes Algorithmen-Muster) spezifizieren zu lassen, sofern er dies wünscht. Andererseits würde IGOR2 aber auch dann die reverse Funktion (anders) korrekt synthetisieren, wenn man den zweiten Parameter in der Spezifikation weg ließe. Wir haben die Spezifikation mit explizitem Akkumulator-Parameter gewählt, da dieses konkrete Problem interessant genug ist, um einige Möglichkeiten von IGOR2 darstellen zu können, gleichzeitig aber einfach genug, um die komplette Synthese durchzugehen.

2.3 Synthese-Operatoren

Wenn im Rahmen der Suche ein KS gewählt worden ist, wird eine der offenen Regeln r zur Verfeinerung ausgewählt. Eine Verfeinerung besteht aus einer Menge s von Nachfolgeregeln. IGOR2 wendet unabhängig voneinander drei Operatoren an, um Verfeinerungen zu berechnen: (i) Die Regel r wird in eine Menge von wenigstens zwei neuen Regeln mit nicht-unifizierenden Mustern, die spezieller als das von r sind, aufgesplittet; (ii) offene *Sub*terme im Körper von r werden als neue Unterprobleme betrachtet und entsprechend neue Hilfsfunktionen eingeführt; (iii) der Körper von r wird durch den (rekursiven) Aufruf einer definierten Funktion ersetzt. Nun angenommen, ein KS P und eine offene Regel $r \in P$ sind gewählt worden. Das Anwenden der Syntheseoperatoren resultiert in einer endlichen (möglicherweise leeren) Menge $\{s_1, \ldots, s_n\}$ von Verfeinerungen. Für jedes s_i $(1 \leq i \leq n)$ wird ein Nachfolge-Kandidaten-KS P_i generiert durch: $P_i = (P \setminus \{r\}) \cup s_i$.

Aufsplitten von Regeln durch Musterspezialisierung. Betrachten wir die Beispiele aus Listing 1 und die entsprechende initiale Regel reverse(xs, ys) $\rightarrow$ zs. Die Mustervariable xs resultiert aus den unterschiedlichen Konstruktoren [] (1. Beispiel) und _:_ (Beispiele 2, 3, 4) an der entsprechenden Position. Wir nennen eine solche Position, die eine *Variable* im Kopf einer initialen Regel und (unterschiedliche) *Konstruktoren* in den Beispieleingaben bezeichnet, *Pivotposition*. Der Splitting-Operator partitioniert die Beispiele nun entlang der unterschiedlichen Konstruktoren an der Pivotposition und erzeugt eine neue initiale Regel für jede Untermenge. In unserem Beispiel wird Beispiel 1 in eine, Beispiele 2, 3, 4 in eine zweite Untermenge sortiert, was zu folgenden neuen Regeln führt:

```
reverse ([], ys)     → ys
reverse (x : xs, ys) → z : zs
```

Da die neuen Regeln immer die unterschiedlichen Konstruktoren an der Pivotposition enthalten, ist sichergestellt, dass sie nicht unifizieren. Falls mehrere Pivotpositionen existieren, wird für jede eine Verfeinerung und ein entsprechendes neues KS generiert.

Behandlung von Unterproblemen. Die initiale Regel reverse(x : xs, ys) $\rightarrow$ z : zs für die Beispiele 2, 3, 4, die aus dem Aufsplitten der ursprünglichen initialen Regel resultierte, ist offen aufgrund der Variablen z, zs. Da diese beiden Variablen echte Subterme der rechten Seite sind, können sie als Unterprobleme betrachtet werden. Der entsprechende Operator ersetzt diese Variablen durch Aufrufe neuer (Hilfs-)Funktionen sub1, sub2,

```
reverse (x : xs, ys) → sub1 (x : xs, ys) : sub2 (x : xs, ys)
```

und leitet Spezifikationsregeln für diese ab, indem die entsprechenden Subterme der Beispielregeln der reverse Funktion genommen werden:[3]

[3] Diese (neuen) Beispiele für sub1, sub2 lassen erkennen, welche Funktionen diese berechnen würden, wenn sie fertig synthetisiert würden: sub1 soll das letzte Element der Eingabeliste liefern und sub2 die reverse Funktion angewendet auf die Eingabeliste verkürzt um das letzte Element. Tatsächlich würde reverse auf diese Weise synthetisiert, wenn ys als Akkumulator *nicht* spezifiziert wäre. Da wir diesen aber in den Beispielen spezifiziert haben, wird die kostengünstigere Variante mit Verwendung dieses Akkumulator-Parameters synthetisiert, wie wir im Folgenden sehen werden.

```
sub1 (x1 : [], ys)           → x1     sub2 (x1 : [], ys)           → ys
sub1 (x1 : x2 : [], ys)      → x2     sub2 (x1 : x2 : [], ys)      → x1 : ys
sub1 (x1 : x2 : x3 : [], ys) → x3     sub2 (x1 : x2 : x3 : [], ys) → x2 : x1 : ys
```

Der Syntheseschritt wird komplettiert durch das Berechnen initialer Regeln für diese neuen Funktionen, die dann zum Kandidaten-KS hinzugefügt werden:

```
reverse ([], ys)      → ys
reverse (x : xs, ys)  → sub1 (x : xs, ys) : sub2 (x : xs, ys)
sub1 (x : xs, ys)     → z
sub2 (x : xs, ys)     → zs
```

Konstruktion (rekursiver) Funktionsaufrufe. Zwei Varianten dieses Operators konstruieren verschieden mächtige Formen (rekursiver) Funktionsaufrufe. Wenn $f(\boldsymbol{p}) \to t$ eine offene Regel ist, generiert die erste Variante Verfeinerungen der Form $f(\boldsymbol{p}) \to f'(\boldsymbol{p'})$, die zweite Variante Verfeinerungen der Form $f(\boldsymbol{p}) \to f'(g_1(\boldsymbol{p}), \ldots, g_n(\boldsymbol{p}))$. Dabei ist f' eine definierte Ziel-, Hintergrund- oder in einem früheren Syntheseschritt eingeführte Hilfsfunktion (mglw. $f = f'$), $\boldsymbol{p'}$ eine Sequenz von Konstruktortermen und die g_i neue definierte Funktionen, die in weiteren Schritten induziert werden.

Betrachten wir nochmals die Beispiele für reverse (Listing 1) und die initiale Regel für Beispiele 2, 3, 4 aus dem Ergebnis des Splittings: reverse(x:xs, ys) → z:zs. Die erste Operatorvariante generiert daraus die Regel reverse(x:xs, ys) → reverse(p1, p2), wobei p1, p2 Konstruktorterme über den Variablen x, xs, ys sind. Damit dieser rekursive Aufruf die entsprechenden Beispiele 2, 3, 4 korrekt berechnet, muss offensichtlich als Vorbedingung gelten, dass die Beispiel*ausgaben* 2, 3, 4 von anderen (zu einfacheren Beispieleingaben) gehörenden Ausgaben subsumiert werden. Derartige Matches werden zunächst gefunden. Z.B. wird Ausgabe 2 (x1:ys) von Ausgabe 1 (ys) mit der Substitution [ys/x1:ys] subsumiert. Diese Substitution wird nun auf die entsprechende *Eingabe* 1 ([], ys) angewandt mit dem Ergebnis ([], x1:ys). Damit Beispiel 2 nun korrekt (durch einen rekursiven Aufruf von Beispiel 1) berechnet wird, müssen die zu findenden Argumente p1, p2 Eingabe 2 zur substituierten Eingabe 1 transformieren. Zwei passende Kandidaten werden gefunden: (p1, p2) = ([], x:ys) und (p1, p2) = (xs, x:ys). Der erste speziellere Kandidat wird aussortiert, weil er unpassend zur korrekten Berechnung der weiteren Beispiele ist. Wir erhalten daher als Nachfolge-KS die geschlossene und korrekte Lösung (Listing 2):

```
reverse ([], ys)      → ys
reverse (x : xs, ys)  → reverse (xs, x : ys)
```

Die zweite Operatorvariante wird nur angewendet, wenn die erste fehlschlägt und keine passenden Konstruktorterme als Argumente findet. Die Idee ist, dass in einem solchen Fall die Argumente mglw. durch rekursive (Hilfs-)Funktionen berechnet werden müssen, also irgendeine Form geschachtelter Funktionsaufrufe notwendig ist. In Abschnitt 3 werden wir Beispiele sehen, allerdings wird diese Variante aus Platzgründen hier nicht erklärt.

Listing 3: swap, induziert aus 6 Beispielen in 6.82 Sekunden

```
swap (x0 : x1 : xs, 0, 1)             →  x1 : x0 : xs
swap (x0 : x1 : x2 : xs, 0, n+2)      →  swap (x1 : swap (x0 : x2 : xs, 0, n+1), 0, 1)
swap (x0 : x1 : x2 : xs, n+1, m+2)    →  x0 : swap (x1 : x2 : xs, n, m+1)
```

Listing 4: weave, induziert aus 11 Beispielen in 33.35 Sekunden; man beachte die automatisch eingeführte rekursive Unterfunktion sub36, die das erste Element entfernt und die Listen rotiert

```
weave (nil)                          →  []
weave ((x:xs) :: xss)                →  x : weave (sub36((x:xs) :: xss))
sub36 ((x:[]) :: nil)                →  nil
sub36 ((x:[]) :: (y:ys) :: xss)      →  (y:ys) :: xss
sub36 ((x:y:xs) :: nil)              →  (y:xs) :: nil
sub36 ((x:y:xs) :: (y:ys) :: xss)    →  (y:ys) :: sub36 ((x:y:xs) :: xss)
```

2.4 Eigenschaften

Jedes KS, welches als Lösung gefunden wird, berechnet alle gegebenen Beispiele vollständig korrekt (was Termination für die gegebenen Beispieleingaben einschließt). Dies ist nicht trivial, da IGOR2 die Kandidaten nicht gegenüber den Beispielen testet (im Gegensatz zu E&T-Methoden). Außerdem hängen die Regeln eines KS, da sie sich gegenseitig (rekursiv) aufrufen, voneinander ab, werden aber *unabhängig* voneinander generiert. Der Beweis[4] erfolgt in zwei Schritten: Zunächst wird gezeigt, dass jeder einzelne Syntheseoperator nur Regeln konstruiert, die in einem extensionalen, von den anderen konstruierten Regeln unabhängigen Sinne, korrekt gegenüber den entsprechenden Beispielen sind. Zweitens wird gezeigt, wie daraus die Korrektheit des gesamten KSs folgt.

Der IGOR2-Problemraum, der durch ein initiales Kandidaten-KS und die beschriebenen Syntheseoperatoren definiert ist, ist zirkelfrei und endlich (falls die Tiefe von Funktionskompositionen beschränkt ist). Damit ist die Suche, die IGOR2 anwendet, terminierend. Weiterhin ist die Suche vollständig in dem Sinne, dass ein Lösungs-KS gefunden wird, sofern es eines gibt, das mit einer endlichen Folge von Anwendungen der Syntheseoperatoren erreichbar ist.

3 Experimente

Ein IGOR2-Prototyp ist in der interpretierten, termersetzungsbasierten Spezifikations- und Programmiersprache MAUDE [CDE+03] entwickelt und mehrere Experimente sind auf einem Intel Dual Core 2.33 GHz, 4GB RAM, Ubuntu-Rechner durchgeführt worden.

IGOR2 synthetisierte korrekte Implementierungen diverser Funktionen für natürliche Zah-

[4] Nachzulesen in [Kit10, Kapitel 4], wie auch die Beweise der weiteren Behauptungen.

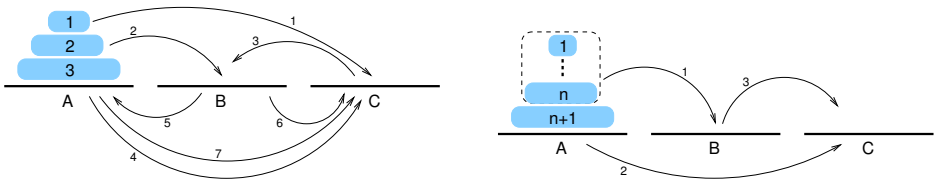

Abbildung 1: *Türme von Hanoi*: Links die Lösung für drei Scheiben (entspricht Listing 5), rechts die allgemeine rekursive Lösung (entspricht Listing 6, Pfeile 1 und 3 repräsentieren rekursive Aufrufe).

Listing 5: Beispiel für die *Türme von Hanoi* für drei Scheiben

```
Hanoi(3, a, b, c, s) → move(1, a, c, move(2, b, c, move(1, b, a,
                         move(3, a, c,
                           move(1, c, b, move(2, a, b, move(1, a, c, s)))))))
```

len, Listen, und Listen von Listen, darunter $=$ und $\leq$, Addition, Fakultät (gegeben Multiplikation) und die Ackermannfunktion (in 3 Sek. aus 17 Beispielen), reverse, take und drop (die ersten n Elemente einer Liste behalten bzw. entfernen), zip (zwei Listen zu einer Liste von Paaren zusammenfügen), swap (zwei Elemente einer Liste an spezifizierten Positionen tauschen, z.B. swap([a,b,c,d], 2, 4) → [a,d,c,b]), *Quicksort* (gegeben *Append* und *Partition*) und, für Listen von Listen, weave (in Rotation über alle gegebenen Listen ein Element nach dem anderen in eine einfache Liste schreiben). Außerdem konnte IGOR2 rekursive Strategien für Benchmark-Probleme im Problemlösen und Planen, wie z.B. für die *Türme von Hanoi*, *Rocket*, oder das Bauen von Türmen in der *Blocksworld*, lernen.

Die Funktion swap wurde z.B. mittels 6 Beispielen (keine Hintergrundfunktionen) spezifiziert, eingeschränkt auf Fälle, in denen die gegebenen Positionen in der Liste vorhanden und unterschiedlich waren und die erste Position kleiner als die zweite war. Listing 3 zeigt die induzierte korrekte Lösung. Die korrekte Lösung für weave ist in Listing 4 gezeigt. Für die *Türme von Hanoi* illustriert Abbildung 1 die Lösung für drei Scheiben sowie die allgemeine rekursive Lösung für beliebig viele Scheiben. Die rekursive Lösung (Listing 6) ist aus den Lösungen für ein bis drei Scheiben (für das 3-Scheiben-Beispiel siehe Listing 5) korrekt induziert worden.

IGOR2 wurde außerdem mit anderen IP-Systemen verglichen – u.a. mit seinem analytischen Vorgänger IGOR1 [KS06] und den funktionalen E&T-Systemen ADATE [Ols95] und MAGICHASKELLER [Kat07]. Tabelle 1 zeigt die Ergebnisse. Details und Ergebnisse für weitere Systeme (aus der induktiven logischen Programmierung) sind in [HKS09] beschrieben.

Listing 6: Rekursive Lösung für die *Türme von Hanoi*, induziert aus 3 Beispielen in 0.08 Sekunden

```
Hanoi (1, a, b, c, s)   → move (1, a, c, s)
Hanoi (n+1, a, b, c, s) → Hanoi (n, b, a, c, move (n+1, a, c, Hanoi (n, a, c, b, s)))
```

Tabelle 1: IGOR2 im Vergleich mit anderen funktionalen IP-Systemen

	ADATE	IGOR1	IGOR2	MAGICHASKELLER
Lasts	365.62	0.051	5.695	19.43
Last	1.0	0.005	0.007	0.01
Member	2.11	—	0.152	1.07
Odd/*Even*	—	—	0.019	—
ISort	83.41	—	0.105	0.01
ShiftR	20.14	0.041	0.127	157.32
Oddslist	466.86	$0.015^{\perp}$	$\odot$	$\times$

— out of scope $\times$ stack overflow $\odot$ time out (10 Minuten) $\perp$ falsches Ergebnis

4 Schluss

IP ist ein anspruchsvolles Forschungsgebiet mit sich abzeichnenden interessanten Anwendungen. IGOR2 ist ein IP-System, das wesentliche Restriktionen des analytischen Ansatzes aufhebt, dabei aber nicht auf ineffizientes Erzeugen-und-Testen zurückfällt. Experimente zeigen vielversprechende Resultate. Die gezeigten, von IGOR2 erfolgreich behandelten Probleme liegen außerhalb des Rahmens vorheriger analytischer Systeme und können von E&T-Systemen nicht mit der gleichen Effizienz synthetisiert werden. Bestimmte Programm-Formen können von IGOR2 jedoch nicht konstruiert werden. Z.B. kann Multiplikation zweier mittels 0 und *Succ* repräsentierter natürlicher Zahlen nur gefunden werden, wenn Addition bereits gegeben ist. Außerdem ist IGOR2 noch nicht effizient genug, wenn die E/A-Beispiele nicht genügend Struktur aufweisen.

Ein Nachteil analytischer Techniken inkl. IGOR2 ist, dass die E/A-Beispiele bis zu einer gewissen Komplexität vollständig sein müssen. Würde z.B. in Listing 1 eines der ersten drei Beispiele weggelassen, würde IGOR2 die reverse Funktion nicht mehr korrekt synthetisieren. E&T-Verfahren sind hier robuster. Ein Lösungsansatz wäre, $\exists$-quantifizierte Variablen in Beispielausgaben zu erlauben. Um jedoch generell robuster bzgl. fehlender und unpräziser Information zu werden, sollte ein probabilistischer Ansatz verfolgt werden.

Eine sinnvolle Erweiterung, die bereits begonnen wurde [HK10], ist das Einbeziehen von Funktionen höherer Ordnung wie z.B. map, filter, reduce. Solche Funktionen kapseln nützliche, terminierende Rekursionsmuster, so dass Funktionsdefinitionen kompakter werden und einfacher gefunden werden können. Außerdem ermöglicht dies eine bessere Analysierbarkeit der Klasse synthetisierbarer Funktionen.

Literatur

[BN99] Franz Baader und Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[CDE+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer und Carolyn Talcott. The Maude 2.0 System. In *Rewriting Techniques and Applications (RTA'03)*, Jgg. 2706 of *LNCS*, Seiten 76–87. Springer, 2003.

[Gul11] Sumit Gulwani. Automating String Processing in Spreadsheets using Input-Output Examples. In *38th SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 2011.

[HK10] Martin Hofmann und Emanuel Kitzelmann. I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP. In *ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation (PEPM 2010), Proceedings*, Seiten 93–100. ACM Press, 2010.

[HKS09] Martin Hofmann, Emanuel Kitzelmann und Ute Schmid. A Unifying Framework for Analysis and Evaluation of Inductive Programming Systems. In *Artificial General Intelligence (AGI'09)*, Seiten 55–60. Atlantis Press, 2009.

[Kat07] Susumu Katayama. Systematic Search for Lambda Expressions. In *6th Symposium on Trends in Functional Programming, selected Papers*, Seiten 111–126. Intellect, 2007.

[Kit10] Emanuel Kitzelmann. *A Combined Analytical and Search-Based Approach to the Inductive Synthesis of Functional Programs*. Dissertation, Fakultät Wirtschaftsinformatik und Angewandte Informatik, Otto-Friedrich Universität Bamberg, 2010.

[KS06] Emanuel Kitzelmann und Ute Schmid. Inductive Synthesis of Functional Programs: An Explanation Based Generalization Approach. *Journal of Machine Learning Research*, 7:429–454, 2006.

[Mit80] Tom M. Mitchell. The Need for Biases in Learning Generalizations. Bericht, Rutgers University, New Brunswick, NJ, 1980.

[NAI+03] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, und F.Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[NLK06] Negin Nejati, Pat Langley und Tolga Konik. Learning Hierarchical Task Networks by Observation. In *23rd International Conference on Machine Learning*, Seiten 665–672. ACM, 2006.

[Ols95] J. R. Olsson. Inductive Functional Programming using Incremental Program Transformation. *Artificial Intelligence*, 74(1):55–83, 1995.

[Plo70] Gordon D. Plotkin. A Note on Inductive Generalization. *Machine Intelligence*, 5:153–163, 1970.

[SK11] Ute Schmid und Emanuel Kitzelmann. Inductive Rule Learning on the Knowledge Level. *Cognitive Systems Research*, 12(3-4):237–248, 2011.

[Sum77] Phillip D. Summers. A Methodology for LISP Program Construction from Examples. *Journal of the ACM*, 24(1):161–175, 1977.

Dr. Emanuel Kitzelmann hat Informatik an der Universität Passau und der Technischen Universität Berlin studiert. Er war anschließend wissenschaftlicher Mitarbeiter bei Prof. Dr. Ute Schmid an der Universität Bamberg, wo er 2010 promovierte. Momentan ist er als Post-Doktorand am *International Computer Science Institute (ICSI)* in Berkeley, USA, tätig, gefördert vom Deutschen Akademischen Austauschdienst (DAAD). Dort forscht er hauptsächlich im Bereich des automatischen Lernens von hierarchischen Task-Netzwerken mittels induktiver Programmsynthese. Außerdem entwickelt er einen Algorithmus zur automatischen Synthese von XSLT-Stylesheets aus Beispiel-XML-Dokumenten.