GUPU - Eine Prolog-Lernumgebung
Deklaratives Programmieren mit Prolog und Constraints

Ulrich Neumerkel

Technische Universität Wien

http://www.complang.tuwien.ac.at/ulrich

- Programmierübungsumgebung GUPU
- Lesarten für logikorientiertes Programmieren
- Slicingtechniken

GUPU
Gesprächsunterstützende Programmierübungsumgebung

- Durchführung großer Programmier-Laborübung ($>$ 200 Teilnehmer) mit geringen Mitteln
- Ersatz für persönliche Betreuung: +schneller −anonymer
  neben Fragestunde und Tutorenbetreuung
- Ersatz für klassische Programmier-Prüfungen

  schriftlich: testen v.a. Konzentrationsfähigkeit, trickreiche Beispiele etc.

  mündlich: am Ende Vergeudung, Gesprächszeit in der Mitte nützlicher
- Vorverlegung der Beurteilung, „prüfungsimmanent“, laufend Adaptierung
- Vermeidung häufiger (betreuungsintensiver) Fehler
- Inkrementelle Entwicklung (seit 1992, $>$ 50 000 Zeilen Code, $>$ 50 000 Wörter Hilfetext)
- Integration bestehender Systeme (MIXtus, cTI)

GUPU
Gesprächsunterstützende Programmierübungsumgebung

Problembereiche:

- Geräteabhängigkeit 7h00-22h00, effektiv zentralisierter Ansatz
- Systemabhängigkeit — bleeding edge, Wartung für benutzte Programme
  SICStus Prolog, EMACS, Ghostscript, Unix, Linux-Kernel (mergemem)
- Akzeptanz bei großer Teilnehmerzahl
  (Anonymität, Gerüchte, „Totalüberwachung“)
  + Computer-, Usenet-, Chat-Kultur
  + Proaktives Ansprechen kritischer Bereiche (Zusammenarbeit etc.)
  + Vorbeurteilung unmittelbar, Gespräch in der Mitte, kritisierbar
  + auch klassische Betreuungselemente → Anonymität selbstgewählt
- Studenten gelangen schneller zu inhaltlichen Fehlern

Allgemeine Fehlerquellen

Größtes Problem: Inkonsistente Sicht v.a. durch überflüssige Zustände

Beispielangaben — Mißverständnisse

Programmtext — Syntax nicht robust

Fenster — überlappend
Editor
Datei — Namen, Kopien etc.
Ausführbares Programm
Testlauf — zu spät, falsch, im Nachhinein

Allgemeine Fehlerquellen & ihre Vermeidung in GUPU

Größtes Problem: Inkonsistente Sicht v.a. durch überflüssige Zustände

Lösung: Integrierte Programmierumgebung — Programmtext zentral

Beispielangaben — Mißverständnisse

Hinweise

Programmtext — Syntax nicht robust

Fenster — überlappend

Editor

Datei — Namen, Kopien etc.

Ausführbares Programm

Testlauf — zu spät, falsch, im Nachhinein

1, 2,

Allgemeine Fehlerquellen & ihre Vermeidung in GUPU

Größtes Problem: Inkonsistente Sicht v.a. durch überflüssige Zustände

Lösung: Integrierte Programmierumgebung — Programmtext zentral

Beispielangaben — Mißverständnisse

Hinweise, Vorbeurteilung, Fragen & Antworten im Text

Programmtext — Syntax nicht robust

Fenster — überlappend

Editor

Datei — Namen, Kopien etc.

Ausführbares Programm

Testlauf — zu spät, falsch, im Nachhinein

Allgemeine Fehlerquellen & ihre Vermeidung in GUPU

Größtes Problem: Inkonsistente Sicht v.a. durch überflüssige Zustände
Lösung: Integrierte Programmierumgebung — Programmtext zentral

Beispielangaben — Mißverständnisse
Hinweise, Vorbeurteilung, Fragen & Antworten im Text
Programmtext — Syntax nicht robust *Syntaxbaum als Erläuterung*

Fenster — überlappend
Editor
Datei — Namen, Kopien etc.
Ausführbares Programm
Testlauf — zu spät, falsch, im Nachhinein

Allgemeine Fehlerquellen & ihre Vermeidung in GUPU

Größtes Problem: Inkonsistente Sicht v.a. durch überflüssige Zustände

Lösung: Integrierte Programmierumgebung — Programmtext zentral

Beispielangaben — Mißverständnisse

Hinweise, Vorbeurteilung, Fragen & Antworten im Text

Programmtext — Syntax nicht robust *Syntaxbaum als Erläuterung*

Einschränkungen bezgl. Layout, Namen, problematische Konstrukte etc

Fenster — überlappend

Editor

Datei — Namen, Kopien etc.

Ausführbares Programm

Testlauf — zu spät, falsch, im Nachhinein

Allgemeine Fehlerquellen & ihre Vermeidung in GUPU

Größtes Problem: Inkonsistente Sicht v.a. durch überflüssige Zustände

Lösung: Integrierte Programmierumgebung — Programmtext zentral

Beispielangaben — Mißverständnisse

Hinweise, Vorbeurteilung, Fragen & Antworten im Text

Programmtext — Syntax nicht robust *Syntaxbaum als Erläuterung*

Einschränkungen bezgl. Layout, Namen, problematische Konstrukte etc

Fenster — ~~überlappend~~ *Fehlermeldungen im Text, meist zeilenbezogen*

~~Editor~~ *Inkonsistenz sichtbar und beschränkt*

~~Datei = Namen, Kopien etc.~~ *gesamter Text in einer Folge*

~~Ausführbares Programm~~ *implizites Übersetzen und Laden*

Testlauf — zu spät, falsch, im Nachhinein

1, 2, 3, 4, 5, 6,

Allgemeine Fehlerquellen & ihre Vermeidung in GUPU

Größtes Problem: Inkonsistente Sicht v.a. durch überflüssige Zustände
Lösung: Integrierte Programmierumgebung — Programmtext zentral

Beispielangaben — Mißverständnisse
Hinweise, Vorbeurteilung, Fragen & Antworten im Text
Programmtext — Syntax nicht robust *Syntaxbaum als Erläuterung*
Einschränkungen bezgl. Layout, Namen, problematische Konstrukte etc
Fenster — ~~überlappend~~ *Fehlermeldungen im Text, meist zeilenbezogen*
~~Editor~~ *Inkonsistenz sichtbar und beschränkt*
~~Datei = Namen, Kopien etc.~~ *gesamter Text in einer Folge*
~~Ausführbares Programm~~ *implizites Übersetzen und Laden*
Testlauf — zu spät, falsch, im Nachhinein *Tests, Viewer im Programmtext*

1, 2, 3, 4, 5, 6, 7,

Allgemeine Fehlerquellen & ihre Vermeidung in GUPU

Größtes Problem: Inkonsistente Sicht v.a. durch überflüssige Zustände
Lösung: Integrierte Programmierumgebung — Programmtext zentral

Beispielangaben — Mißverständnisse
Hinweise, Vorbeurteilung, Fragen & Antworten im Text
Programmtext — Syntax nicht robust *Syntaxbaum als Erläuterung*
Einschränkungen bezgl. Layout, Namen, problematische Konstrukte etc
Fenster — ~~überlappend~~ *Fehlermeldungen im Text, meist zeilenbezogen*
~~Editor~~ *Inkonsistenz sichtbar und beschränkt*
~~Datei = Namen, Kopien etc.~~ *gesamter Text in einer Folge*
~~Ausführbares Programm~~ *implizites Übersetzen und Laden*
Testlauf — zu spät, falsch, im Nachhinein *Tests, Viewer im Programmtext*
Testlauf nach jedem Sichern; Interaktion im Text
Programmfragmente als Erklärungen bei Fehlern

1, 2, 3, 4, 5, 6, 7, 8

GUPU — Inhalt

- pure, monotone Untermenge von Prolog + Constraints v.a. CLP(FD)
- wichtigstes Lehrziel: Verstehen von Relationen
- Lesarten Deutsch $\leftrightarrow$ Prolog
- spezifikationsnahes Programmieren
  1. Auffinden von Relationen
  2. Namensgebung
  3. Zusicherungen (Tests) vor Implementierung
  4. eigentliches Programmieren
- Lerntempo: Wochenrhythmus, flexible Nachtragsfristen
- Einzelbeurteilung, Abgabegespräch in der Mitte, Zusammenarbeit erwünscht
- Inhalt primär über Beispiele — insgesamt 8×10
- keine Projekte

Logikorientiertes Programmieren — Lesarten

- controlled deduction: ALGORITHM = LOGIC + CONTROL
- Traditionell: deklarative und prozedurale Lesart
  nur für sehr kleine Programme verwendbar
- Verfeinerung durch Transformationen: Selektive Lesarten

  Verallgemeinerung: Löschen von Zielen.

  vater(Vater) ←
  ***** ~~männlich(Vater)~~,
  kind_von(_Kind, Vater).

  Spezialisierung: Hinzufügen von Zielen (v.a. false/0).

  ~~verheiratet_mit(Gatte, Gattin) ⇐ **false**~~,
  ~~gatte_gattin(Gatte, Gattin)~~.
  verheiratet_mit(Gattin, Gatte) ←
  gatte_gattin(Gatte, Gattin).

Logikorientiertes Programmieren — Selektive Lesarten

+ ermöglichen informelles Lesen größerer Programme
+ Quelltexttreue, einfache Veranschaulichung (Durchstreichen, Zudecken)
+ kein neuer Formalismus wie Beweisbaum, Trace, Store
+ auch für (unvollständige) Constraints geeignet
+ hilfreich beim Schreiben neuer Programme
+ automatisierbar zur Diagnose aller üblichen Fehlerarten — transformationsbasierte Slicing-Techniken
  - fehlende Lösungen (insufficiency)
  - unerwartete Lösung (incorrectness)
  - Nichttermination

Lesarten — Slicing zur Fehlersuche

fehlende Lösung (insufficiency): maximale scheiternde Verallgemeinerun
erklärt *data inconsistency* und *modelling error*

unerwartete Lösung (incorrectness): maximale erfüllte Spezialisierung
(momentan nur mit false/0)

Nichttermination: maximale nichtterminierende Spezialisierung

Gemeinsame Eigenschaften:

+ Fehler in Fragment bedingt Fehler im ursprünglichen Programm
+ Fragment *muß* verändert werden
+ keine Benutzerinteraktion (daher keine Bedienungsfehler)
. meist mehrere Fragmente

Beispiel: Fehlende Lösung

← bruder_von(B, P). % Fehler

bruder_von(B, P) ←
    dif(B,P),
    männlich(B),
    kind_von(B,V),
    männlich(V),
    kind_von(P,V),
    kind_von(B,M),
    kind_von(P,M),
    weiblich(V).

männlich(franz_I).
männlich(joseph_II).
männlich(leopold_II).

weiblich(maria_theresia).

kind_von(joseph_II, maria_theresia).
kind_von(joseph_II, franz_I).
kind_von(leopold_II, maria_theresia).
kind_von(leopold_II, franz_I).

1,

Beispiel: Fehlende Lösung in maximaler Verallgemeinerung

← bruder_von(B, P). % Fehler

bruder_von(B, P) ←
* ~~dif(B,P),~~
* ~~männlich(B),~~
* ~~kind_von(B,V),~~
  männlich(V),
* ~~kind_von(P,V),~~
* ~~kind_von(B,M),~~
* ~~kind_von(P,M),~~
  weiblich(V).

männlich(franz_I).
männlich(joseph_II).
männlich(leopold_II).

weiblich(maria_theresia).

~~kind_von(joseph_II, maria_theresia).~~
~~kind_von(joseph_II, franz_I).~~
~~kind_von(leopold_II, maria_theresia).~~
~~kind_von(leopold_II, franz_I).~~

1, 2

Beispiel: Unerwartete Lösung

$\not\leftarrow$ kind_von(K, E), kind_von(E, K). % Fehler

kind_von(joseph_II, maria_theresia).
kind_von(joseph_II, franz_I).
kind_von(leopold_II, maria_theresia).
kind_von(marie_antoinette, maria_theresia).
kind_von(maria_theresia, marie_antoinette).
kind_von(leopold_II, franz_I).

1,

Beispiel: Unerwartete Lösung in maximaler Spezialisierung

$\not\leftarrow$ kind_von(K, E), kind_von(E, K). % Fehler

~~kind_von(joseph_II, maria_theresia).$\Leftarrow$ **false**.~~
~~kind_von(joseph_II, franz_I).$\Leftarrow$ **false**.~~
~~kind_von(leopold_II, maria_theresia).$\Leftarrow$ **false**.~~
kind_von(marie_antoinette, maria_theresia).
kind_von(maria_theresia, marie_antoinette).
~~kind_von(leopold_II, franz_I).$\Leftarrow$ **false**.~~

1, 2

Example: Nonterminating program

```
← ancestor_of(Anc, leopold_I).          % Does not terminate
child_of(karl_VI, leopold_I).
child_of(maria_theresia, karl_VI).
child_of(joseph_II, maria_theresia).
child_of(leopold_II, maria_theresia).
child_of(leopold_II, franz_I).
child_of(marie_antoinette, maria_theresia).
child_of(franz_I, leopold_II).

ancestor_of(Anc,Desc) ←
    child_of(Desc,Anc).
ancestor_of(Anc,Desc) ←
    child_of(Child, Anc),
    ancestor_of(Child, Desc).
```

1,

Example: Nonterminating program and minimal failure slice

← ancestor_of(Anc, leopold_I)., **false.** % Does not terminate

~~child_of(karl_VI, leopold_I).← **false.**~~

~~child_of(maria_theresia, karl_VI).← **false.**~~

~~child_of(joseph_II, maria_theresia).← **false.**~~

~~child_of(leopold_II, maria_theresia).← **false.**~~

child_of(leopold_II, franz_I).

~~child_of(marie_antoinette, maria_theresia).← **false.**~~

child_of(franz_I, leopold_II).

~~ancestor_of(Anc,Desc) ←**false,**~~

~~child_of(Desc,Anc).~~

ancestor_of(Anc,Desc) ←

child_of(Child, Anc),

ancestor_of(Child, Desc)., **false.**

1, 2.

Example: Nonterminating program

```
← ancestor_of(Anc, leopold_I).          % Does not terminate
child_of(karl_VI, leopold_I).
child_of(maria_theresia, karl_VI).
child_of(joseph_II, maria_theresia).
child_of(leopold_II, maria_theresia).
child_of(leopold_II, franz_I).
child_of(marie_antoinette, maria_theresia).
child_of(franz_I, leopold_II).

ancestor_of(Anc,Desc) ←
    child_of(Desc,Anc).
ancestor_of(Anc,Desc) ←
    ancestor_of(Child, Desc),
    child_of(Child, Anc).
```

1,

Example: Nonterminating program and minimal failure slice

← ancestor_of(Anc, leopold_I), **false.** % Does not terminate

~~child_of(karl_VI, leopold_I).← **false.**~~

~~child_of(maria_theresia, karl_VI).← **false.**~~

~~child_of(joseph_II, maria_theresia).← **false.**~~

~~child_of(leopold_II, maria_theresia).← **false.**~~

~~child_of(leopold_II, franz_I).← **false.**~~

~~child_of(marie_antoinette, maria_theresia).← **false.**~~

~~child_of(franz_I, leopold_II).← **false.**~~

~~ancestor_of(Anc,Desc) ← **false,**~~

~~child_of(Desc,Anc).~~

ancestor_of(Anc,Desc) ←

ancestor_of(Child, Desc), **false,**

~~child_of(Child, Anc).~~

1, 2.

```
←                    phrase(regexp(Expr), Xs0,Xs) terminates_if
    finiteground(Expr), boundlist(Xs0).
regexp([]) ⟶
    [].
regexp([E]) ⟶
    [E].
regexp({_Expr}) ⟶
    [].
regexp({Expr}) ⟶
    regexp(Expr),
    regexp({Expr}).
regexp(A*B) ⟶
    regexp(A),
    regexp(B).
```

1,

$\leftarrow$ phrase(regexp(Expr), Xs0,Xs) terminates_if
finiteground(Expr), boundlist(Xs0).

regexp([]) $\longrightarrow$
[].

~~regexp([E]) $\Longrightarrow$ **{false},**~~
~~[E].~~

~~regexp({_Expr}) $\Longrightarrow$ **{false},**~~
~~[].~~

regexp({Expr}) $\longrightarrow$
regexp(Expr),
regexp({Expr}), **{false}.**

~~regexp(A*B) $\Longrightarrow$ **{false},**~~
~~regexp(A),~~
~~regexp(B).~~

1, 2,

?: Min.f-slice explains with subst's missing termination proof

$\leftarrow$ **Expr = {[]},** phrase(regexp(Expr), Xs0,Xs) terminates_if
    ~~finiteground(Expr),~~ boundlist(Xs0).
regexp([]) $\longrightarrow$
    [].
~~regexp([E]) $\Longrightarrow$ {false},~~
    ~~[E].~~
~~regexp({_Expr}) $\Longrightarrow$ {false},~~
    ~~[].~~
regexp({Expr}) $\longrightarrow$ **{Expr = []}, % ... not yet implemented ...**
    regexp(Expr),
    regexp({Expr}), **{false}.**
~~regexp(A*B) $\Longrightarrow$ {false},~~
    ~~regexp(A),~~
    ~~regexp(B).~~

1, 2, 3.