Fakultät Informatik, Mathematik und Naturwissenschaften

Implementierung der Tiefensuche in KIV

eingereicht als

Prüfungsrelevante Studienleistung

im Studiengang Informatik (Master)

von

Ricardo Hofmann
Matthias Neubert
Stefan Veit

Leipzig, 24. Februar 2011

Betreuender Professor: Prof. Dr. Uwe Petermann

Inhaltsverzeichnis

1 Einleitung

2
Problembeschreibung

3
Spezifikation und Module

3.1 Spezifikation „digraph-dfs-proc“

Ausgehend von den Überlegungen in Kapitel 2 soll zunächst eine Tiefensuche als Prozedur umgesetzt werden. Dies bedeutet, dass die Rückgabe des Programmes ein Wahrheitswert ist. Enthält der Pfad G einen Weg zum Startknoten zur Zielknotenmenge, so wird TRUE zurück geliefert, anderfalls FALSE. Im folgenden Absatz sieht man die Signatur der Spezifikation:

digraph-dfs-proc =
enrich digraph-basic, union_bool_graph **with**
predicates deptSearchP : elem × list × digraph;

axioms

$\vdash z = [] \wedge a \in$_ v g $\vee$ a ' = [] $\rightarrow$ (deptSearchP(a, z, g) $\leftrightarrow$ false);

$\vdash pathp(x, g) \wedge x.first = a \wedge x.last = b \rightarrow (deptSearchP(a, b', g) \leftrightarrow true)$;

$\vdash a --> b \in$_ e g $\wedge$ b $\in$ z $\rightarrow$ (deptSearchP(a, z, g) $\leftrightarrow$ true);

$\vdash a --> b \in$_ e g $\wedge$ b -> c $\in$_ e g $\wedge$ c $\in$ z $\rightarrow$ (deptSearchP(a, z, g) $\leftrightarrow$ true);

end enrich

Die Spezifikation erweitert hierbei die vorhandenen Spezifikationen digraph-basic sowie union_bool_graph, welche Hilfssätze enthalten und im Kapitel 3.xx erläutert werden. Die in der Spezifikation definierte Prozedur deptSearchP bildet dabei drei Eingabeparameter elem × list × digraph auf einen Wahrheitswert ab. Dabei bezeichnet „elem“ den Startknoten, „list“ die Menge der Zielknoten und „digraph“ den Graphen. Des Weiteren werden in der Spezifikation verschiedene Axiome deklariert:

- Das erste Axiom beschreibt den Fall, dass unzulässige Parameter beim Prozeduraufruf übergeben werden. Dies ist der Fall, wenn keine Zielknoten angegeben werden oder a ein leerer Knoten ist. In diesem Fall existiert kein Pfad im Graph und somit soll die Prozedur FALSE zurückgeben.
- Das zweite Axiom nimmt Bezug auf die Signatur pathp, welche in der Graphenbibliothek implementiert wurde und überprüft, ob es einen Pfad x im Graphen g gibt. Ist dies der Fall und der Anfangsknoten des Pfades stimmt mit dem Startknoten a über ein und der letzte Knoten des Pfades ist ein Element der Zielknotenmenge, dann liefert die Prozedur deptSearchP TRUE zurück.
- Im dritten Axiom wird der Bezug zum Begriff der Kante im Graphen gebildet. Wenn es eine Kante von Knoten a nach Knoten b im Graphen g gibt und b in der Zielknotenmenge liegt, dann muss die Methode deptSearchP diesen Weg finden. Dieses Axiom ist ein Spezialfall des vierten Axioms.
- Im vierten Axiom wird der Begriff der Kante um die transitive Eigenschaft erweitert. Unsere Prozedur muss also auch dann einen Pfad finden bzw. TRUE zurückliefern, wenn es von Knoten a über einen anderen Knoten b einen Weg in die Zielknotenmenge gibt. Setzt man in diesem Fall b = c, hat man den Spezialfall, welcher in Axiom 3 beschrieben wurde.

3.2 Spezifikaton „Rekursiv“ - Axiome

3.3 Modul „dfs-it-1“

Das Modul „dfs-it-1“ implementiert die Spezifikation „digraph-dfs-proc“, also die Tiefensuche als Prozedur mit der Rückgabe eines Wahrheitswertes. Die genaue Implementierung ist unten aufgezeigt. Allgemein kann gesagt werden, dass zunächst die fehlerhaften Eingaben abgefangen werden, bevor der eigentliche Algorithmus zum Einsatz kommt. Dabei wird in einer Schleife so lange ein neuer Knoten expandiert, bis entweder ein Pfad gefunden wurde, oder alle erreichbaren Knoten abgearbeitet wurden. Für letzteren Fall wurde die Hilfsfunktion „getM“ einegführt, welche im Kapitel 3.xx noch genauer erläutert wird. In der while-Schleife wird wie erwähnt ein Knoten von der open-Liste untersucht und geschaut, ob von ihm erreichbare Knoten noch nicht auf der close-Liste sind. Dazu wird die Funktion „outsortSuccs“ verwendet,

welche aus der Graphenbibliothek stammt.
Unterhalb der Deklaration findet man noch einmal die Axiome. Hierbei wird aufgelistet, in wie weit die Axiome mit der Implementierung bewiesen werden konnten und wie die Abhängigkeiten untereinander sind.

decl

```
deptSearchP#(a, z, g; var bbool)
{
let open = [] + a, close = [], n0 = 0, a0 = a in
{
    bbool := false;
    n0 := getM(a, g) + 1;
    if ¬ (a ' = [] ∨ z = []) then
      if a ∈ z then bbool := true else
      {
        while n0 > 0 ∧ (bbool ↔ false) ∧ (open ≠ []) do
            {
              a0 := open.first;
              if a0 ∈ z then bbool := true else
              {
                n0 := n0 - 1;
                close := a0 ' + close;
                open := set2list(outsortSuccs(a0, list2set(close), g)) + open.rest
              }
            };
        if n0 = 0 then bbool := false
      }
}
}
```

Term-deptsearchpjsiz

⊢ deptSearchP#true

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : decl
- used by : Exp-02, Exp-03, Exp-04
- interactions : 47

- proof steps : 47

Exp-01

⊢ z = [] ∧ a ∈_v g ∨ a ' = [] → (deptSearchP#bbool ↔ false)

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : decl
- interactions : 0
- proof steps : 8

Exp-02

⊢ pathp(x, g) ∧ x.first = a ∧ x.last = b → (deptSearchP#bbool ↔ true)

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz
- interactions : 1
- proof steps : 2

Exp-03

⊢ a –> b ∈_e g ∧ b ∈ z → (deptSearchP#bbool ↔ true)

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz
- interactions : 3
- proof steps : 4

Exp-04

⊢ a –> b ∈_e g ∧ b –> c ∈_e g ∧ c ∈ z → (deptSearchP#bbool ↔ true)

- Type: a proof obligation
- must be proved
- a complete proof exists
- used lemmas : Term-deptsearchpjsiz
- interactions : 1
- proof steps : 2

Wie zu sehen ist, konnten alle Axiome bewiesen werden. Dabei wurde bei den Axiomen 2 bis 4 auf den Terminationsbeweis zurück gegriffen. In folgendem Abschnitt sind noch einmal die bei den Beweisen verwendeten Beweisschritte aufgeführt.

Statistic for each theorem:

	proof steps	interactions	automation	induction
Term-deptsearchpjsiz	47	47	0.0 %	nat
Exp-01	8	0	100.0 %	—
Exp-03	4	3	25.0 %	—
Exp-04	2	1	50.0 %	—
Exp-02	2	1	50.0 %	—

Heuristic applications:

heuristic	total applications	percentage
Interactive	52	82.5 %
symbolic execution	5	7.9 %
System	4	6.3 %
simplifier	1	1.5 %
weak unfold	1	1.5 %

Rule applications:

rule	total applications	percentage
assign right	13	20.6 %
simplifier	8	12.6 %
if right	6	9.5 %
normalize	4	6.3 %
insert lemma	4	6.3 %
skip right	4	6.3 %
assign left	3	4.7 %
while exit right	3	4.7 %
insert elim lemma	2	3.1 %
split left	2	3.1 %

while right	1	1.5 %
case distinction	1	1.5 %
cut formula	1	1.5 %
skip left	1	1.5 %
apply induction	1	1.5 %
if negative left	1	1.5 %
induction	1	1.5 %
call left	1	1.5 %
weakening	1	1.5 %
vardecls left	1	1.5 %
split right	1	1.5 %
call right	1	1.5 %
vardecls right	1	1.5 %
execute while	1	1.5 %

Lemma applications:

theorem	total applications	percentage
Term-deptsearchpjsiz	4	66.6 %
decl	2	33.3 %

3.4 Modul „dfs-rek-1“

3.5 Modul „dfs-rek-func“

3.6 „Hilfssätze“ - Spezifikation und Module

Bei der Bearbeitung des Projektes und der Beweisführung mussten verschiedene Ansätze aus der Graphenbibliothek zusammengeführt sowie Teilbeweise ausgelagert werden, da sie in mehreren Beweisen notwendig waren.
Zunächst werde die Spezifikation „union_bool_graph“ betrachtet, welche die notwendigen Pakete der Graphenbibliothek zusammenfasst. Folgenden die verwendete Spezifikation:

union_bool_graph = digraph-outsorted-neighbours + bool

Eine weitere Spezifikation ist „digraph-path“, in welcher die in Kapitel 3.1 erwähnte Signatur „getM“ definiert wird.

digraph-path =
enr¯ich union_bool_graph **with**
functions getM : elem × digraph → nat ; **axioms**

getM(a, @g) = 0;

getM(a, @g +v a) = 1;

¬ b ∈_ v g ∧ a ∈_ v g → getM(a, g +e a –> b) = getM(a, g) + 1;

end enrich

Die Methode „getM“ bekommt sowohl einen Knoten a sowie einen Graphen g übergeben und gibt eine Zahl m zurück. Diese bezeichnet hierbei die Anzahl der von dem Knoten a im Graphen g erreichbaren Knoten, wobei sowohl direkt erreichbare als auch über transitive Kanten erreichbare berücksichtigt werden. Auch hier werden wieder drei Axiome definiert:

- Axiom 1 gibt dann, dass in einem leeren Graphen kein Knoten von a erreicht werden kann.
- In Axiom 2 wird der leere Graph um einen Knoten a erweitert. Da Anzahl der von a erreichbaren Knoten ist in diesem Fall 1.
- Im dritten Axiom wird ein Knoten b, welche noch nicht im Graphen lag sowie eine Kante vom Knoten an zu Knoten b hinzugefügt. Die Anzahl der erreichbaren Knoten entspricht nun der Anzahl der erreichbaren Knoten ohne den Knoten b plus 1.

Die Spezifikation wird durch das Modul „dfs-temp“ implementiert. Folgend ist die Implementierung der Funktion „getM“ zu sehen.

```
getM#(a, g; var n)
{
let s20 = ∅, b99 = a in
{
   n := 0;
   s20 := g.vs;
   while s20 ≠ ∅ do
      {b99 := s20.max; s20 := s20.butmax; if path∃(a, b99, g) then n := n + 1}
}
}
```

Für einen gegebenen Knoten werden in einer Schleife alle anderen Knoten des Graphen betrachtet. Mit der Funktion „path∃“ aus der Graphenbibliothek wird dann überprüft, ob es einen Pfad zwischen den beiden Knoten gibt. Wenn ja, wird die Zählvariable erhöht.
Die Beweise der Axiome gelangen alle, wie sich in folgendem Abschnitt sehen lässt.

Term-getmjsiz: ⊢ getM# true

uses: (decl)
used by: (Exp-01 Exp-03)
Proof steps: 50, interactions: 50

Exp-01:

⊢

¬ b ∈_ v g ∧ a ∈_ v g → getM#getM#m = m0 + 1

uses: (Term-getmjsiz)
Proof steps: 2, interactions: 1

Exp-02: $\vdash$ getM# m = 0

uses: (decl)
Proof steps: 6, interactions: 6

Exp-03: $\vdash$ getM# m = 1

uses: (Term-getmjsiz)
Proof steps: 1, interactions: 1

Auch hier sei wieder auf die verwendeten Beweisschritte hingewiesen, welche in den folgenden Tabellen aufgeführt sind:

Statistic for each theorem:

	proof steps	interactions	automation	induction
Term-getmjsiz	50	50	0.0 %	nat
Exp-02	6	6	0.0 %	—
Exp-01	2	1	50.0 %	—
Exp-03	1	1	0.0 %	—

Heuristic applications:

heuristic	total applications	percentage
Interactive	57	98.2 %
System	1	1.7 %

Rule applications:

rule	total applications	percentage
insert rewrite lemma	14	24.1 %
cut formula	9	15.5 %
assign right	8	13.7 %
simplifier	6	10.3 %
weakening	3	5.1 %
call right	2	3.4 %
split right	2	3.4 %
vardecls right	2	3.4 %
case distinction	2	3.4 %
if right	1	1.7 %
while exit right	1	1.7 %
all right	1	1.7 %
normalize	1	1.7 %

invariant right	1	1.7 %
insert lemma	1	1.7 %
skip right	1	1.7 %
exists left	1	1.7 %
induction	1	1.7 %
all left	1	1.7 %

Lemma applications:

theorem	total applications	percentage
decl	2	66.6 %
Term-getmjsiz	1	33.3 %

3.7 Nicht beweisbare Module

4
Zusammenfassung

Unser Ziel im Rahmen des Projektes eine sinnvolle Verwendung bzw. der Graphenbibliothek zu erreichen, konnte unserer Meinung nach voll umgesetzt werden. Mit der Implementierung und dem Beweis der Tiefensuche ist es uns gelungen, einen der gängigen Suchalgorithmen für Graphen zu verifizieren. Dabei ist es uns gelungen, verschiedene Varianten des Suchverfahrens umzusetzen. Die iterative Variante baut dabei vor allem auf die Abarbeitung von Schleifen auf, die rekursive Variante benötigt die Beweise von verschachtelten Funktionsaufrufen. Mit diesen verschiedenen Varianten ist es uns gelungen, einen großen Teil der Möglichkeiten von KIV zu erforschen. Dabei sei aber auch angemerkt, dass KIV ein sehr komplexes und mächtiges Werkzeug ist. Über das bloße Grundverständnis hinaus muss man sich für jeden Anwendungsfall spezifische Lösungsszenarien aneignen und diese geschickt kombinieren. Dies gelang und leider auch nicht in allen Fällen, so dass zumindest ein Beweis nicht abgeschlossen werden konnte. Leider machte uns an einigen Stelle auch KIV selbst die Arbeit schwer, da es in einigen Bereichen nicht so funktioniert, wie man es sich intuitiv erhofft. Alles in allem war das Projekt aber sehr förderlich, um unser Verständnis für die Verifikation von Programmen zu erweitern.

Abbildungsverzeichnis

Literaturverzeichnis