Verified design by contract

case studies

Maassen, H.

Award date:
2008

Verified Design by Contract: Case Studies

Harald Maassen

11 augustus 2008

Inhoud

1 Introductie

1.1 Probleemstelling en Onderzoeksvragen

Een van de grote openstaande opgaven in de informatica is de zoektocht naar een verifying compiler: een compiler die beredeneert of een programma aan zijn specificatie voldoet. Al sinds o.a. Floyd [11] en Hoare [13] eind jaren 60 de fundering legden voor het formeel redeneren over programma's wordt er onderzoek gedaan naar het automatiseren van dit proces. Ondanks vooruitgang op dit gebied is het ideaal nog niet gehaald; tot noch toe zijn automatische provers vooral het domein van onderzoekers in plaats van gebruikers.

Hoare stelde echter onlangs dat de meeste factoren die vroeger de voortgang op programmaverificatie belemmerden tegenwoordig veel minder zwaar wegen [14]. De verifying compiler lijkt dichterbij dan ooit.

In dit verslag zullen een aantal huidige toolkits die gebruik maken van automatische verificatie worden vergeleken. Zulke toolkits bestaan, formeel gezien, uit drie onderdelen: een programmeertaal waarin een algoritme kan worden uitgedrukt, een specificatietaal waarin het beoogde doel van het algoritme wordt geformaliseerd (zie Design by Contract, 1.2.1) en een bewijzer die de twee aan elkaar koppelt door de verificatiecondities die bij de specificatie horen te genereren en die met de programmatekst te bewijzen).

We voeren deze vergelijking uit met behulp van een aantal case. We bestuderen bij de toolkits de manier waarop ze de drie onderdelen implementeren. We doen dit in de context van het doel van de afzonderlijke toolkits, de keuzes van de overige toolkits en de stappen die we nemen bij een formele, papieren afleiding.

Wat betreft de papieren afleidingen kijken we ook naar de structuur van het programma en de manier waarop correctheid daarmee kan worden verweven. We nemen hiervoor twee verschillende aanpakken, die beide een sterke aanhang in de literatuur genieten: Stepwise Refinement (zie 1.2.2) en Incremental Development (zie 1.2.3).

We stellen daarbij de volgende vragen:

1. Hoe verhouden de toolkits zich met het ideaal?

   Onze ideale verifying compiler is een toolkit die, zonder teveel inmenging en in een praktisch bruikbaar tijdsbestek, een volledig bewijs van een programma kan genereren, dan wel fouten in dat programma aanwijzen.

   De verifying compiler bestaat niet. Elke toolkit maakt concessies, gebaseerd op zijn eigen doelen. We bespreken welke keuzes er binnen de toolkits zijn gemaakt en de effecten hiervan. Als referentie voor het type bewijzen dat we snel en automatisch gegenereerd willen zien gebruiken we papieren afleidingen gebaseerd op de theorieën van o.a. Floyd, Hoare en Dijkstra [8][11][13].

2. (In hoeverre) zijn de toolkits praktisch bruikbaar?

   Geen van de toolkits neemt een al geschreven programma zonder annotatie en voert hierop een volledige analyse uit; ze hebben elk speciale invoer nodig. Ook kost het genereren en invullen van bewijsverplichtingen tijd dan wel moeite. We kijken hoe eenvoudig het is om onze papieren programmeerproblemen op te lossen met de toolkits. Van groot belang is hierbij de schaalbaarheid van de toolkits.

3. Welke features ontbreken bij de huidige tools om ze praktisch toepasbaar te maken?

   We kijken hier naar de grootste knelpunten die we tijdens de case studies tegenkomen. Daarbij is het interessant om te zien of zaken die in een toolset moeizaam gaan in de

anderen wellicht eenvoudiger zijn. Waar mogelijk wagen we ons zelfs aan enige suggesties tot verbetering.

De toolkits die we onderling vergelijken zijn:

- Eiffel, een programmeertaal met geïntegreerde design by contract constructies die via runtime asserties worden gecontroleerd.
- ESC, een specificatielaag op Java met aparte statische analysetool.
- Cocktail, een programmeertaal met interactieve stellingbewijzer gebaseerd op GCL
- Perfect Developer, een programmeertaal met design by contract en volledige statische analyse.

De vier gebruikte talen, evenals de modeltaal GCL, worden kort beschreven in paragraaf 1.3.

Om een idee te krijgen van de verschillende toolkits en hun ondersteuning voor (verified) Design by Contract, beginnen we met een paar korte case studies.

We bespreken voor elke toolkit twee algoritmes, namelijk de bounded linear search en de binary search. We beginnen daarbij met het formuleren van een model-afleiding in GCL, gebaseerd op die gegeven door Kaldewaij [16].

De bounded linear search is door zijn eenvoud een geschikt basisalgoritme dat toch de belangrijkste programmaconstructie (een loop) bevat. Ook de binary search bestaat in de basis uit een enkele lus, maar introduceert een non-triviale bewijsstap met betrekking tot gesorteerde arrays en gebruikt een ingewikkeldere invariant.

We hebben deze algoritmes in elk van de toolkits geïmplementeerd (details hiervan zijn te vinden in de appendix) en trekken daarover onze conclusies in paragraaf 2.3.

Daarna bekijken we de twee meest interessante toolkits gedetailleerder met behulp van een grotere case study: Dijkstra's kortste pad algoritme. Dit algoritme berekent voor een gewogen graaf de kortste afstand tussen een startvertex en alle overige vertices van die graaf.

Dit is een vrij pittig algoritme, mede doordat het zo compact is. Er gebeurt veel in een paar regels code.

Een interessant aspect ervan is dat er weinig formele afleidingen van dit algoritme bestaan. Bekende algoritmiekboeken [3] raken de bewijsverplichtingen slechts zijdelings en ook rigoureuzere werken (waaronder Dijkstra's eigen artikel [7][15]) en gaan niet volledig in op de onderliggende logica.

Er zijn andere projecten om Dijkstra's algoritme (automatisch) te bewijzen, onder andere met behulp van de B-methode of ACL2 [22]. Aan de grootte en complexiteit van deze werken is te zien dat een volledig bewijs voor het kortste pad algoritme geen triviale taak is.

De complexiteit van Dijkstra's algoritme staat ons toe de twee methodes (Stepwise Refinement en Incremental Development) te gebruiken bij het opstellen van een formele afleiding, en die te vergelijken met de werkwijze van de toolkits.

Ten slotte maken we de balans op, gebaseerd op de resultaten van het onderzoek. We vatten kort de conclusies van de eerste set case studies (besproken in 2.3) en de grote case study (3.5) samen, en trekken ten slotte enige overkoepelende conclusies.

1.2 Begrippen

1.2.1 Design by Contract

Floyd stelde dat we middels inductief redeneren eigenschappen van programma's van de vorm "als de initiële waardes van een programma voldoen aan relatie R_1, dan voldoen de waardes bij terminatie aan relatie R_2." kunnen bewijzen. Hoare [13] en Dijkstra [6] bouwden

hierop voort en introduceerden begrippen als Hoare tripels, weakest precondition en invarianten.

Meyer noemt deze relaties het contract van een programma [19]. In zijn programmeertaal Eiffel [20] (zie 1.3.2) kunnen deze contracten expliciet worden gespecificeerd en gecontroleerd.

Hoewel de term door Meyer is gepatenteerd, is deze inmiddels gemeengoed en wordt hij gebruikt bij alle toolkits die specificaties in de vorm van pre- en postcondities en invarianten gebruiken.

De in 1.3 besproken toolkits ondersteunen elk Design by Contract.

1.2.2 Stepwise Refinement

De ideale verifying compiler bewijst een programma in zijn geheel zonder voorkennis van de opbouw. Programma's worden echter in de nabije toekomst nog geschreven door mensen en hoewel een post-hoc analyse van een algoritme het vertrouwen erin kan versterken, stijgt de complexiteit hiervan exponentieel met de grootte. Ook biedt deze manier van werken geen soelaas tijdens het ontwerp van een algoritme, maar kan de programmeur pas na de laatste stap zien of zijn programma datgene doet wat hij voor ogen had.

Een oplossing voor dit probleem is Structured Programming [5]. Door een programma op te delen in logische, zelfstandige onderdelen (door aandacht te schenken aan de *structuur* van een programma) kan de complexiteit worden teruggedrongen. Van onderdelen kan worden geabstraheerd waardoor ze, met hun correctheidsargument, kunnen worden gebruikt in andere onderdelen. Volgens een inductief argument geldt: als alle delen correct zijn, is het geheel dat ook.

Structured Programming helpt de programmeur ook tijdens het ontwerp. De correctheid van elke toevoeging of verfijning kan in isolatie worden bekeken en steunt op die van het origineel.

Een vorm van Structured Programming is Stepwise Refinement [9][5]. Stepwise Refinement neemt de specificatie van het programma en gebruikt abstracte expressies om hieraan te voldoen. Een voorbeeld van een initiële versie van een Stepwise Refinement algoritme is:

```
begin "print first thousand prime numbers" end
```

In een aantal stappen zullen de "gaten" in het programma steeds meer concreet worden gemaakt totdat er een volledig programma overblijft. Zolang elke invulling voldoet aan zijn eigen specificatie, blijft het gehele programma correct:

```
begin
      var "table p";

      "fill table p with first thousand prime numbers";
      "print table p"
end
```

1.2.3 Incremental Development

Een tweede methode om programma's te verifiëren en te construeren is Incremental Development [23].

Deze methode begint met kleine, concrete programma's en breidt die stap voor stap uit met extra functionaliteit en specificatie. Ook hier geldt dat doordat programma's gebaseerd zijn op hun voorganger er een groot deel van het bewijs tussen iteraties kan worden meegenomen.

Het volgende programma berekent of de waarde *true* voorkomt in een boolean array:

```
const
      N: int;
      a: array of bool; { #a = N }
var
      r: bool;
      n: int;

n := 0;
r := false;

do n < N →
      r := r ∨ a[n];
      n := n + 1;
od
{ r ≡ (∃i:0 ≤ i < N: a[i]) }
```

We kunnen dit programma uitbreiden door middel van early termination waardoor *n* een witness wordt:

```
const
      N: int;
      a: array of bool; { #a = N }
var
      r: bool;
      n: int;

n := 0;
r := false;

do n < N ∧ ¬r →
      r := r ∨ a[n];
      n := n + 1;
od
{ r ≡ (∃i:0 ≤ i < N: a[i]) ∧ ( n ≥ N ∨ a[n] ) }
```

1.3 Toolkits

1.3.1 Guarded Command Language (GCL)

De Guarded Command Language is een theoretische programmeertaal, geïntroduceerd door Dijkstra [8], die dankzij een compacte operatieset en koppeling aan Hoare tripels [13] zeer geschikt is voor het analyseren en bewijzen van programma's.

De voorbeeldprogramma's uit de vorige paragrafen volgen de GCL syntax.

Bij de analyse van de verschillende case studies zullen we GCL als referentiepunt gebruiken. We beginnen telkens met een modelafleiding in GCL en vergelijken de uitwerkingen in de andere talen met deze afleiding.

In dit document wordt GCL hier en daar uitgebreid met datatypes als sets of functies en constructies als de foreach loop, waardoor we code als de volgende kunnen tegenkomen:

```
const a: array of bool;
foreach b ∈ a →
      r := r ∨ b
rof
```

1.3.2 Eiffel

Eiffel is een object georiënteerde programmeertaal, met (o.a.) geïntegreerde Design by Contract constructies. Eiffel is ontworpen door Meyer, als concrete implementatie van diens ideeën over object oriented programming [21]. Zo beschikt Eiffel over generic programming, (veilige) multiple inheritance, garbage collection en een focus op leesbare code, features waarvan we sommigen terugvinden in moderne talen als Java en C#.

Onze interesse gaat vooral uit naar Eiffel's ondersteuning van DBC en (automatische) verificatie.

Eiffel maakt gebruik van dynamische verificatie. De op runtime asserties gebaseerde contracten van een Eiffel programma worden tijdens de uitvoer van een programma gecontroleerd. Er volgt een foutmelding wanneer er een conditie of een invariant wordt geschonden. Dit in tegenstelling tot de annotatie van een GCL programma. Deze vormt een statisch bewijs: voor alle mogelijke invoerwaardes die aan de preconditie voldoen zal het programma na uitvoer de postconditie bewerkstelligen.

Hierdoor voldoet Eiffel per definitie niet aan het ideaalbeeld van de verifying compiler. Dit neemt niet weg dat de taal een interessant studiepunt is door zijn nauwe verwantschap met de concrete, veelgebruikte huidige talen als C++ en Java.

Als voorbeeld geven we hier het algoritme uit 1.2.3 weer als een Eiffel programma:

```
0. search(a: ARRAY [BOOLEAN]): BOOLEAN is
1.             local
2.                 r: BOOLEAN
3.                 x: INTEGER
4.         do
5.             r := FALSE
6.
7.             from
8.                 x := 0
9.             invariant
10.                P0a: 0 <= x
11.                P0b: x <= a.count
12.                -- P1: r = there exists an i in a[0..x-1] with
                              a[i]=true
13.            variant
14.                a.count - x
15.            until
16.                x = a.count
17.            loop
18.                r := r or a @ x
19.                x := x+1
20.            end
21.
22.            result := r
23.
24.          ensure
25.              -- Q: result = there exists an i in a with
                             a[i]=true
26.        end
```

1.3.3 ESC/Java

JML is een specificatielaag op Java die Design by Contract features toevoegt [2]. Zaken als invarianten en pre- en postcondities kunnen aan programma's worden toegevoegd in de vorm van speciaal geformatteerd commentaar, dat door verscheidene tools wordt herkend en gebruikt voor analyse.

JML is een open-source samenwerkingsverband tussen verschillende instanties en personen waaronder de universiteiten van Florida, Texas, Nijmegen en Dublin.

Een voorbeeld van een code-comment (bij een stack implementatie):

```
0.  /*@   public normal_behavior
1.    @     requires !theStack.isEmpty();
2.    @     assignable size, theStack;
3.    @     ensures theStack.equals(\old(theStack.trailer()));
4.    @ also
5.    @   public exceptional_behavior
6.    @     requires theStack.isEmpty();
7.    @     assignable \nothing;
8.    @     signals_only BoundedStackException;
9.    @*/
```

Een van de tools die JML-commentaar gebruiken is ESC. Een (extended) static checker die JML annotaties automatisch bewijst. ESC begon als onderzoeksproject van Compaq in 1997 [18] en is momenteel in handen van Kind Software, een onderdeel van de onderzoeksgroep van de universiteit van Dublin, in de vorm van ESC/Java2.

Belangrijk kenmerk van deze methode is dat hij niet streeft naar volledige verificatie. De originele gebruikershandleiding stelt dat ESC sound noch complete is [17]. Opvallend voorbeeld hiervan is dat loops in de originele ESC/Java niet volledig werden bewezen (later zijn hier wel mogelijkheden voor toegevoegd, maar die worden in dit verslag niet behandeld).

Door de open natuur van zowel JML als ESC verkeren de projecten continu in een staat van flux. Verschillende uitbreidingen en tools zijn beschikbaar. Wij bekijken de "standaard" ESC versie waarbij Simplify als stellingbewijzer wordt gebruikt.

1.3.4 Cocktail

De Cocktail tool, ontwikkeld aan de universiteit van Eindhoven door Franssen [12], is een interactieve stellingbewijzer in combinatie met een programma-editor, die helpt bij het afleiden van programma's volgens hun specificatie.

Cocktail bestaat uit een theoremprover waarin types, proposities en theorems kunnen worden opgesteld en een programma-editor waarin deze worden gebruikt in een concreet algoritme.

Het opstellen en bewijzen van een algoritme in Cocktail gebeurt via Stepwise Refinement: het toevoegen van statements leidt tot bewijsverplichtingen die via de interactieve stellingbewijzer worden gedicht.

Programma's worden in Cocktail opgesteld in GCL. Cocktail geeft zelf als deel van de annotatie van het programma aan wat de bewijsverplichtingen bij de gebruikte constructies zijn:

```
var r: bool
|[ var x: nat
      { True }
      …
      { P0(0) ∧ P1(0,false) }
      r := false;
      x := 0;
      while x < N do
            { P0(x) ∧ P1(x,r) ∧ (x < N=true) }
            …
            { P0(s(x)) ∧ P1(s(x),(r OR B(x))) }
            r := r OR b.x;
            x := s(x);
      od
      { P0(x) ∧ P1(x,r) ∧ (x<N=false) }
      …
      { Q(r) }
]|
```

De beletseltekens ("…") in het programma hierboven geven bewijsgaten aan die door de gebruiker gedicht dienen te worden.

Het bewijzen van deze gaten geschiedt met behulp van de interactieve stellingbewijzer. Hiermee kan een gebruiker tableau-bewijzen genereren zoals die voor de stelling:

```
ThPlus1: ∀y:nat.plus(0,y)=y
```

```
{ Apply with AxPlus1 }
plus(0,0)=0
{ Let: }
 y: nat
 { Assume: }
  plus(0,y)=y
  { Reflexivity }
  s(y)=s(y)
  { Leibniz }
  s(plus(0,y))=s(y)
  { Leibniz with AxPlus2 }
  plus(0,s(y))=s(y)
 { ⇒ intro }
 (plus(0,y)=y) ⇒ (plus(0,s(y))=s(y))
{ ∀-intro }
(∀y:nat.(plus(0,y)=y)⇒(plus(0,s(y))=s(y)))
{ Induction }
∀y:nat.plus(0,y)=y
```

1.3.5 Perfect Developer

Perfect Developer is een programmeertaal ontwikkeld door Escher Tech [1], die van de grond af aan opgebouwd is met het doel om 100% (of dicht bij de 100%) verifieerbare code te genereren.

Net als Eiffel gebruikt Perfect Developer hiervoor Design by Contract, waaraan de auteurs het woord verified prefixen om de kracht en integratie van de automatische stellingbewijzer te benadrukken [4].

Perfect Developer streeft ernaar contracten volledig te specificeren (in tegenstelling tot Eiffel waar contracten een ietwat beperkte expressiviteit kennen) en deze automatisch te verifiëren.

Daarnaast biedt PD de mogelijkheid om een abstract datamodel te definiëren met eigen contracten en de implementatie hieraan volgens representatie-invarianten te koppelen.

Stepwise Refinement is mogelijk via dit onderscheid tussen abstracte en concrete datamodellen en daarnaast kan in de code direct specificaties worden geplaatst die (nog) niet vergezeld gaan van een implementatie.

Door objectoriëntatie en modules kunnen bewijzen worden verdeeld in kleinere stukken, wat ook een op Incremental Development gebaseerde aanpak mogelijk maakt.

2 Kleine case studies

In de volgende paragrafen introduceren we de twee algoritmes en hun GCL afleiding. In paragraaf 2.3 beschrijven we onze bevindingen bij het implementeren ervan in de gekozen programmeertalen. Uitwerkingen in de afzonderlijke programmeertalen zijn opgenomen in de appendix.

2.1 Bounded Linear Search

Een loop in GCL heeft de volgende vorm:

do G → S **od**

Waarbij de bewijsregel geldt:

$\{ P \wedge G \}\ S\ \{ P \}$
$\{ P \wedge G \wedge VF = C \}\ S\ \{ VF < C \}$
$P \Rightarrow def.G$
$\underline{P \wedge G \Rightarrow VF \geq 0}$
$\{ P \}$ **do** $G \rightarrow S$ **od** $\{ P \wedge \neg G \}$

In de praktijk wordt voorafgaand aan de loop de invariant *P* geïnitialiseerd en dient de loop een gewenste postconditie *Q* te bewerkstelligen. Dit leidt tot de volgende standaardannotatie van een loop:

```
{ H }
I;
{ invariant P, Proof 0, bound VF }
do G → { P ∧ G } S { P, Proof 1 } od
{ Q, Proof 2, termination: Proof 3 }
```

Met als bewijsverplichtingen:

Proof 0) Initialisatie

$\{ H \}\ I\ \{ P \}$

Proof 1) Invariantie

$\{ P \wedge G \}\ S\ \{ P \}$

Proof 2) Finalisatie, bestaande uit:

Proof 2a) Finalisatie

$[\ (P \wedge \neg G) \Rightarrow Q\ ]$

Proof 2b) Welgedefinieerdheid

$[\ P \Rightarrow def.G\ ]$

Proof 3) Terminatie, bestaande uit

Proof 3a) Begrensdheid

$[\ (P \wedge G) \Rightarrow VF \geq 0\ ]$

Proof 3b) Voortgang

$\{ P \wedge G \wedge VF = C \}\ S\ \{ VF < C \}$

Vaak bestaan *S* en *I* uit eenvoudige toekenningen, waarvoor de bewijsverplichtingen zijn:

Proof) Toekenning

$\{ P \}\ v := w\ \{ Q \}$

$[P \Rightarrow Q(v := w)]$
$[P \Rightarrow def.w]$

Het bounded linear search algoritme is een concreet voorbeeld van een loop-gebaseerd algoritme; door zijn eenvoud zeer geschikt als eerste oefening.

Gegeven een boolean functie *B* en een bereik tussen *0* en *N*, bereken of *B* geldt voor een getal binnen het bereik.

```
|[    const N: int, { N ≥ 0 }
            B: int → bool; { ∀i: 0 ≤ i < N: def.B(i) }

      var x,r: int, bool;
      { true }
      x := 0,
      r := false;
      { P, bound VF }
      do x < N → { P ∧ x < N }
                 r := r ∨ b.x,
                 x := x + 1;
                 { P }
      od
      { Q }
]|
```

met:

$P \equiv P_0 \wedge P_1$
$P_0 \equiv 0 \leq x \leq N$
$P_1 \equiv r = (\exists i: 0 \leq i < x: b.i)$
$Q \equiv r = (\exists i: 0 \leq i < N: b.i)$

$VF = N-x$

Concreet zijn de bewijsverplichtingen voor de bounded linear search de volgende:

Proof 0) Initialisatie

$(True) \Rightarrow (0 \leq x \leq N)(x := 0, r := false)$

$(True) \Rightarrow (r = (\exists i: 0 \leq i < x: b.i))(x := 0, r := false)$

Proof 1) Invariantie

$0 \leq x \leq N \wedge r = (\exists i: 0 \leq i < x: b.i) \wedge x < N$
$\Rightarrow$
$(0 \leq x \leq N)(r := r \vee b.x, x := x+1)$

$0 \leq x \leq N \wedge r = (\exists i: 0 \leq i < x: b.i) \wedge x < N$
$\Rightarrow$
$(r = (\exists i: 0 \leq i < x: b.i))(r := r \vee b.x, x := x+1)$

Proof 2) Finalisatie

$0 \leq x \leq N \wedge r = (\exists i: 0 \leq i < x: b.i) \wedge \neg(x<N)$
$\Rightarrow$
$r = (\exists i: 0 \leq i < N: b.i)$

$0 \leq x \leq N \wedge r = (\exists i: 0 \leq i < x: b.i)$
$\Rightarrow$
$def.(x < N)$

Proof 3) Terminatie

$0 \leq x \leq N \wedge r = (\exists i: 0 \leq i < x: b.i) \wedge x < N$
$\Rightarrow$
$N-x \geq 0$

$0 \leq x \leq N \wedge r = (\exists i: 0 \leq i < x: b.i) \wedge x < N \wedge N-x=C$
$\Rightarrow$
$(N-x < C)(x := x+1, r := r \vee b.x)$

Proof) Toekenning

$[P \Rightarrow def.w]$

Het algoritme bevat meerdere toekenningen, maar de gedefinieerdheidseis is slechts op één plek toepasselijk (constanten zijn altijd gedefinieerd):

$0 \leq x \leq N \wedge r = (\exists i: 0 \leq i < x: b.i) \wedge x < N$
$\Rightarrow$
$def.(r \vee b.x)$

We zullen de hier gevonden condities als leidraad gebruiken bij onze vergelijking van de overige talen. Kernpunten hierbij zijn in hoeverre de condities zijn uit te drukken (of worden gevonden) door de behandelde talen en of ze (automatisch) kunnen worden bewezen. Voor een handmatige uitwerking verwijzen we de lezer naar [16].

2.2 Binary Search

We introduceren een primitieve versie van het binary search algoritme in GCL. Merk op dat we beginnen met een versie waarin gesorteerdheid van het array *f* geen enkele rol speelt. Deze eerste versie rekent dan ook niet uit of een element voorkomt in het array.

In een vervolgstap nemen we gesorteerdheid mee als preconditie en zullen we laten zien dat in het geval van een gesorteerd array dit algoritme daadwerkelijk membership berekent.

Binary Search, gedefinieerd als:

```
|[
      const
            N,A: int { N ≥ 1 };
            f: array[0..N] of int { f.0 ≤ A < f.N };
      var
            x,y: int;

      x,y := 0, N;

      do    x+1 ≠ y
            → |[ var h: int;
                  h := (x+y) div 2;

                  if f.h ≤ A → x := h
                  [] A < f.h → y := h
                  fi
               ]|
      od
      { 0 ≤ x < N ∧ f.x ≤ A < f.(x+1) }
]|
```

Met als annotatie:
Postconditie:

$Q \equiv Q_0 \wedge Q_1$
$Q_0 \equiv 0 \leq x < N$
$Q_1 \equiv f.x \leq A < f.(x+1)$

Merk op dat uit deze postconditie niet volgt of *A* deel uitmaakt van *f*.

Invariant:

$P \equiv P_0 \wedge P_1$
$P_0 \equiv 0 \leq x < y \leq N$
$P_1 \equiv f.x \leq A < f.y$

Variante functie:

$VF = y-x$

De bewijsverplichtingen bij dit programma zijn:

Proof 0) Initialisatie

$(N \geq 1 \wedge f.0 \leq A < f.N)$
$\Rightarrow$
$(0 \leq x < y \leq N)(x := 0, y := N)$

$(N \geq 1 \wedge f.0 \leq A < f.N)$
$\Rightarrow$
$(f.x \leq A < f.y)(x := 0, y := N)$

Proof 1) Invariantie

```
{ 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y }
|[    var h: int;
      h := (x+y) div 2;

      if f.h ≤ A → x := h
      [] A < f.h → y := h
      fi
]|
{ 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y }
```

We splitsen dit bewijs als volgt op, volgens de spelregels van de conditional:

```
{ 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y }
h := (x+y) div 2
{ 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y ∧ h = (x+y) div 2 }

( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y ∧ h = (x+y) div 2
∧ f.h ≤ A )
⇒
( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y )( x := h )

( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y ∧ h = (x+y) div 2
∧ A < f.h )
⇒
( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y )( y := h )
```

Proof 2) Finalisatie

```
( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ ¬(x+1 ≠ y) )
⇒
{ 0 ≤ x < N ∧ f.x ≤ A < f.(x+1) }
```

Proof 3) Terminatie

We splitsen:

```
( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y )
⇒
( y - x > 0 )

( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y  ∧ y-x = C ∧
h = x+y div 2 ∧ f.h ≤ A )
⇒
( y - x < C)( x := h )

( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y  ∧ y-x = C ∧
h = x+y div 2 ∧ A < f.h )
⇒
( y - x < C)( y := h )
```

Proof) Gedefinieerdheid

Als onderdeel van de if-constructie dient bewezen te worden dat de guards aan de welgedefinieerdheidseis voldoen:

```
( 0 ≤ x < y ≤ N ∧ f.x ≤ A < f.y ∧ x+1 ≠ y ∧ h = (x+y) div 2 )
⇒
(def.(f.h ≤ A) ∧ def.(A < f.h))
```

En dat ten minste één guard naar true evalueert:

$$(0 \le x < y \le N \wedge f.x \le A < f.y \wedge x+1 \neq y \wedge h = x+y \text{ div } 2)$$
$$\Rightarrow$$
$$(f.h \le A \vee a < f.h)$$

Ten slotte kunnen we, zoals eerder gezegd, met behulp van gesorteerdheid testen of *A* voorkomt in het array. Er geldt namelijk:

$$(0 \le x < N \wedge f.x \le A < f.(x+1) \wedge \text{"f is gesorteerd"})$$
$$\Rightarrow$$
$$(\exists i : 0 \le i < N : f.i = A) \equiv (f.x = A)$$

Het algoritme kan dus gebruikt worden (en wordt in de praktijk meestal gebruik) om lidmaatschap van een element in een gesorteerd array te bepalen. Er moet wel nog bewezen worden dat de zojuist aangenomen implicatie geldt.

In de praktijk volgt hierna een uitbreiding op het algoritme, waarin de eis $f.0 \le A < f.N$ wordt verwijderd. Aangezien het laatste element van het array nooit in de code wordt geïnspecteerd, kunnen we een "gedachte"-element achteraan het array toevoegen met als waarde ∞. We gaan hier in de voorbeelden echter niet op in.

Belangrijke verschillen met de bounded linear search is dat dit algoritme een branching structuur toevoegt aan de loop, waardoor het invariantiebewijs complexer is dan die van de bounded linear search. Daarnaast is de stap op het eind, die de postconditie van de loop omzet in een nieuwe postconditie een interessante, aangezien er naast puur logisch redeneren een aantal axioma's over getallen en gesorteerdheid nodig zijn om de implicatie rond te krijgen.

2.3 Discussie

In de appendices worden de uitwerkingen van beide algoritmes in de verschillende talen in detail uitgewerkt. We beperken ons tot de resultaten hiervan.

Op basis van de gemaakte stijloefeningen kunnen we enige conclusies trekken over de behandelde programmeermethodes. We kunnen de eigenschappen van de talen op een rijtje zetten, om zo de verschillen en overeenkomsten te benadrukken.

We geven aan de hand van onze ervaringen antwoord op de eerste onderzoeksvraag: hoe verhouden de toolkits zich met een ideale verifying compiler. We bekijken de kenmerken die een ideale verifying compiler bezit, en vergelijken in hoeverre deze beschikbaar zijn in de toolkits.

Object Oriëntatie

Het is belangrijk dat een programmeertaal in deze tijd voldoet aan een aantal high level eisen, voornamelijk object-georiënteerdheid en de bijbehorende aanwezigheid van een sterke basis aan types en operaties.

Zowel Eiffel, Java en Perfect Developer zijn hogere niveau programmeertalen. Cocktail programma's zijn, afgezien van het feit dat Cocktail niet naar machinecode compileert, puur procedureel.

Expressiviteit

Het criterium waarop de vier toolkits zijn gekozen (ondersteuning voor Design by Contract) wordt in principe door elke taal vervuld, maar de een doet dit wat rigoureuzer dan de ander.

In het bijzonder Eiffel heeft een beperkte expressiviteit voor het uitdrukken van contracten: Eiffel verifieert pre- en postcondities middels asserties, vergelijkbaar met de asserties die we vinden in non-DBC programmeertalen zoals C. De programmeur beschikt hierbij niet over logische constructies zoals quantoren, waardoor de uitdrukkingskracht beperkt is. De quantoren die we nodig hebben voor de binary en linear search hebben een beperkt domein,

waardoor ze wel in de taal zijn uit te drukken, maar dit gaat (door de dynamische verificatie) ten koste van efficiëntie.

ESC, Perfect Developer en Cocktail zijn gebaseerd op 1e orde logica met enige aanvullingen (zoals de inductiestelling van Cocktail en ingebouwde kennis van types als sets bij ESC en PD).

Statische verificatie

We willen ons ervan overtuigen dat onze code voldoet aan de contracten en wel door middel van bewijzen.

Eiffel bekijkt door dynamische verificatie slechts een beperkt deel van de toestandsruimte.

ESC's statische analyse, daarentegen, is sound noch complete. Dit is een fundamenteel probleem dat ontstaat uit de Java basis waarover geredeneerd moet worden: deze taal bevat constructies die redeneren zeer moeilijk maken [4]. Daarnaast bevat de uitgebreide bibliotheek aan Java types geen tegenhanger in de vorm van axioma's. Dit maakt bewijsstappen die niet zozeer op programmalogica steunen maar op specifieke eigenschappen van types moeilijk - iets wat ons ervan weerhield de laatste stap uit de binary search te bewijzen (zie appendix).

Cocktail bewijst programma's slechts partieel, de variante functie zowel als welgedefinieerdheid worden niet bewezen. De programmeur kan met wat kunstgrepen zichzelf dwingen om deze zaken te bewijzen (bijvoorbeeld door ze als invarianten of programmavariabelen mee te nemen), maar ze worden niet standaard door de tool meegenomen. Dit leidt ertoe dat een incorrect programma als correct kan worden gezien.

Automatische verificatie

Het liefst willen we bewijzen zoveel mogelijk door de computer laten oplossen.

Bij het interactieve Cocktail moeten we het leeuwendeel van het werk zelf doen; er bestaat wel een automatische stellingbewijzer maar deze is eerder een hulpmiddel om kleine delen van grote bewijzen te automatiseren dan dat deze het hele proces kan overnemen.

ESC en PD, daarentegen, zijn volledig geautomatiseerd tot het punt waar de bewijzer moeilijk te sturen is. Er bestaan hiervoor mogelijkheden, zoals het introduceren van axioma's of het specifiek laten uitrekenen van stellingen, maar directe toegang tot bewijzen is er niet.

Aan de andere kant willen we dat deze automatische verificatie snel gebeurt. De talen die minder proberen te bewijzen (ESC en Eiffel) nemen weinig tijd in beslag, terwijl Perfect Developer lang kan rekenen bij moeilijkere stappen (en zelfs enig debuggen nodig heeft).

Samenvattend

We zetten de genoemde eigenschappen op een rijtje:

	OO	Expressief	Geverifieerd	Automatisch
Eiffel	√	-[1]	-[2]	√
ESC/Java	√	√	-[3]	√
Cocktail	-	√	√	-[4]
PD	√	√	√	√

Afgaande op deze resultaten, lijkt Perfect Developer de meest interessante kandidaat voor verder onderzoek. De methode mag dan niet compleet zijn, maar claimt (en lijkt) wel sound te zijn. Als we naar de uitvoer van Perfect Developer kijken, dan heeft deze een tegenhanger voor elk van onze papieren bewijsverplichtingen (zie appendix) en we vragen ons af of dit ook voor grotere problemen het geval is.

Wat betreft de andere talen heeft Perfect Developer nog een ander interessant kenmerk: namelijk dat er nog weinig documentatie over te vinden is. De ESC/Java taal is een open project, over GCL zijn vele boeken te lezen [6][16] en ook Eiffel biedt door zijn beperkte verificatie weinig verrassingen.

Cocktail is om eenzelfde reden interessant. Het is een open (universitair) project, maar nog vrij jong en zonder uitgebreidere case studies.

In het vervolg van dit verslag zullen we ons daarom concentreren op Perfect Developer en Cocktail.

[1] Door het ontbreken van een aparte specificatietaal zijn veel contracten niet uit te drukken, zoals quantoren over een oneindig domein.

[2] Eiffel gebruikt dynamic verification waardoor moeilijk te bereiken toestanden mogelijk niet worden getest.

[3] ESC/Java komt dichtbij verified DBC, op twee zaken na. Ten eerste wordt een "lakse" methode gebruikt om loops te bewijzen: de loop wordt enkele malen uitgerold en invarianten en variante functie worden enkel voor dit vast aantal iteraties gecontroleerd. Ten tweede sluiten de contracten en de onderliggende Java code niet 100% op elkaar aan, daar ESC een toevoeging op Java is. Dit leidt ertoe dat ESC niet sound is (mbt tot het herkennen van foutieve programma's.

[4] Cocktail genereert automatisch de te bewijzen stellingen en helpt de gebruiker hierbij met de interactieve stellingbewijzer.

3 Dijkstra's Kortste Pad Algoritme

Alvorens het algoritme te bekijken, worden begrippen die in de loop van dit document zullen worden gebruikt kort toegelicht.

3.1 Basisbegrippen

Graaf

Een graaf $G = (V, E)$ wordt gedefinieerd als een verzameling vertices en edges, waarbij een edge een gesorteerd paar van vertices bevat.

We nemen aan dat tussen elk (gesorteerd) paar vertices ten hoogste één edge loopt (de graaf is simpel) en dat de graaf eindig is.

De set van vertices die via één edge bereikbaar zijn uit v noemen we de buren van v ofwel $v.adj$.

Pad

Op een graaf definiëren we een pad als een rij van vertices $p = \langle v_0, v_1, .., v_n \rangle$.

Wanneer een pad bestaat uit twee of meer vertices, dient er tussen elk opeenvolgend paar vertices in het pad een edge te lopen. Een pad kan ook bestaan uit één of nul vertices.

Niet-lege paden hebben een start en een eindvertex. De eigenschap dat een pad q tussen i en j loopt geven we aan als: $i \rightsquigarrow^{q} j$. Het kan zijn dat i gelijk is aan j.

Een edge tussen i en j kan als volgt worden weergegeven: $i \rightarrow j$.

Gewicht & Afstand

Iedere edge heeft een waarde, het gewicht van de edge. We gebruiken hiervoor de functie w. Deze functie geeft een gewicht terug bij invoer van een edge.

Het gewicht van een pad is de som van de gewichten van de edges die deel uitmaken van dat pad.

De afstand $\delta(i,j)$ tussen twee vertices i en j is het minimale gewicht van alle paden tussen i en j. Indien er geen pad bestaat is deze afstand dus ∞.

$$\delta(i,j) = \min p : i \rightsquigarrow^{p} j : w(p)$$

In het geval van een cyclus met een negatief gewicht kan de afstand de waarde $-\infty$ hebben. Dijkstra's algoritme legt een extra eis op aan de graaf om dit voorkomen:

Voor Dijkstra's algoritme geldt dat de graaf geen negatieve gewichten mag bevatten. Deze eigenschap is van groot belang voor de correctheid van het algoritme. Vaak kunnen grafen die hier niet aan voldoen worden afgebeeld op grafen die wel geschikt zijn voor Dijkstra's algoritme, bijvoorbeeld door cycli te verwijderen en alle edgegewichten met een constante waarde te verhogen.

Een kortste pad tussen twee vertices is een pad met een gewicht gelijk aan de afstand tussen die vertices.

3.2 Eigenschappen van paden

Hieronder volgen een aantal stellingen met betrekking tot kortste paden die regelmatig zullen terugkeren in de verschillende algoritmes en bewijzen.

Kortste Subpaden

Ieder subpad van een kortste pad is zelf een kortste pad.

> ***Stelling - Kortste Subpaden:***
>
> $a \rightsquigarrow i \rightsquigarrow^{p} j \rightsquigarrow b$ "is een kortste pad"
> $\Rightarrow$ p "is een kortste pad"

Een korte verdediging van deze stelling luidt als volgt:

Neem een kortste pad tussen twee vertices *a* en *b* en een willekeurig subpad *p* uit dit pad. De begin en eindvertices van *p* noemen we *i* en *j*:

$$a \rightsquigarrow i \rightsquigarrow^{p} j \rightsquigarrow b$$

Stel er zou een korter pad tussen i en j bestaan, genaamd *p'*. Dan bestaat er ook het pad:

$$a \rightsquigarrow i \rightsquigarrow^{p'} j \rightsquigarrow b$$

De lengte van dit pad is, doordat *p'* korter is dan *p*, korter dan de lengte van een kortste pad tussen *a* en *b*, wat in tegenspraak is met de originele aanname.

Paduitbreiding

Logischerwijs is de lengte van een kortste pad de som van de lengtes van de (niet-overlappende) kortste subpaden. Een speciaal geval is hierbij het extraheren van de laatste edge van een kortste pad:

> ***Stelling - Kortste Paduitbreiding:***
>
> $s \rightsquigarrow u \rightarrow v$ is een kortste pad
> $\Rightarrow$
> $\delta(s, v) = \delta(s, u) + w(u, v)$

Driehoeksongelijkheid

In het geval dat niet zeker is of de laatste edge deel uitmaakt van een kortste pad, is de afstand tot *u* plus het gewicht van de edge nooit minder dan de kortste afstand tot *v*:

> ***Stelling - Driehoeksongelijkheid:***
>
> $u \rightarrow v$
> $\Rightarrow$
> $\delta(s, v) \leq \delta(s, u) + w(u, v)$

Aangezien *w(u,v)* oneindig is wanneer er geen edge tussen *u* en *v* bestaat is de preconditie niet nodig:

> ***Stelling - Driehoeksongelijkheid:***
>
> $\delta(s, v) \leq \delta(s, u) + w(u, v)$

3.3 Stepwise Refinement in GCL

Ten eerste zullen we proberen Dijkstra's algoritme af te leiden middels Stepwise Refinement. We beginnen met een programma dat enkel een set van afstanden uitbreidt totdat deze alle vertices behelst en vullen stap voor stap de gaten in dit programma in totdat we een implementeerbare versie hebben.

3.3.1 Het Algoritme

Het uiteindelijke doel van het algoritme is het vullen van een array *d*, zodanig dat voor elke vertex de bijbehorende *d*-waarde gelijk is aan de kortste padsafstand naar die vertex vanuit een startvertex *s*.

```
R: { ∀i: i ∈ V: d(i) = δ(s,i) }
```

We gaan in een aantal stappen werken naar het volgende programma:

```
const
      V: set of vertex;
      E: set of edge;
      w: E → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }

var
      d: vertex → int;
      S: set of vertex;
      Q: set of vertex;

d(s) := 0;
Q := V\{s};
S := {s};
foreach v: v ∈ V\s.adj → d(v) := ∞ rof;
foreach v: v ∈ s.adj → d(v) := d(s)+w(s,v) rof;

do Q ≠ ∅ →
      var u: vertex;
      u := (some v: v ∈ Q: d(v) = (min d': d' ∈ Q: d'(v)));
      Q := Q \ {u};
      S := S ∪ {u};
      foreach v ∈ u.adj →
            d(v) := min(d(v), d(u) + w(u,v)
      rof
od;
{ ∀i: i ∈ V: d(i) = δ(s,i) }
```

Een korte uitleg van het programma in zijn geheel volgt hier en wordt verder uitgewerkt bij het stapsgewijs opbouwen in volgende paragrafen.

De gegevens van het algoritme bestaan uit de sets *V* en *E* die onze vertices en edges bevatten, vergezeld van een functie *w* die de gewichten van de edges retourneert. Onze laatste invoervariabele is de startvertex *s* van waaruit we de kortste afstanden gaan berekenen.

Over de invoer zijn een aantal zaken bekend ($H_{0..2}$), namelijk dat de edges lopen tussen vertices in *V*, dat alle edges een niet-negatief gewicht hebben en dat de startvertex s deel uitmaakt van *V*.

We introduceren drie variabelen:
De functie *d* (ook wel het array *d* genoemd) houdt voor vertices de kortst bekende afstand vanuit *s* bij, dit hoeft niet overeen te komen met de daadwerkelijke (kortste) afstand.

De set *S* bestaat uit die vertices waarvoor de afstand berekend is, terwijl voor de vertices in *Q* deze waarde nog niet bekend is.

We initialiseren deze variabelen als volgt: we voegen *s* toe aan de set bekende vertices *S* en zetten zijn afstand op *0*. De overige vertices vormen *Q*. Voor de buren van *s* schatten we de afstand *d* in op het gewicht van de verbindende edge en voor de resterende vertices initialiseren we *d* op oneindig. Merk op dat het type **int** ietwat losjes wordt gebruikt; er wordt aangenomen dat ∞ een geldige waarde is voor een variabele van dit type)

De hoofdlus van het programma bestaat uit de volgende stappen

- neem die vertex *h* uit *Q* met de laagste geschatte afstand. Deze afstand is gelijk aan de daadwerkelijke afstand tot *h*.
- voeg *h* toe aan *S* en verwijder hem uit *Q*.
- bereken voor alle buren van *h* de afstand vanuit *h* (gelijk aan de afstand tot *h* plus het gewicht van de verbindende edge). Indien deze nieuwe afstand kleiner is dan de huidige schatting, verlaag deze.

Wanneer de set van vertices waarvoor we de afstand nog niet kennen (*Q*) leeg is, weten we voor elke vertex de afstand en termineert ons programma.

We nemen dit algoritme als de laatste stap van onze afleiding. Dit wil niet zeggen dat deze versie "klaar" is.

Zo gebruiken we nog een aantal abstracte types zoals functies en sets, laten we in het midden hoe we de vertex *h* met minimale berekende afstand berekenen, houden we ons niet bezig met het opslaan en retourneren van de daadwerkelijke paden (alleen de afstanden) en missen we een laatste vereenvoudigingstap die onze initialisatie een stuk eleganter maakt (zie 3.3.6).

Toch zal blijken dat deze stap "dichtbij" genoeg is voor implementatie in Perfect Developer en tegelijkertijd al een stap te ver is in termen van bewijsbaarheid.

3.3.2 Programma opzet

Wanneer we alle gegevens en de gewenste postconditie op een rijtje zetten krijgen we de volgende programmastructuur:

```
const
      V: set of vertex;
      E: set of vertex × vertex;
      w: vertex × vertex → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }
var
      d: vertex → int;

"vul array d";
{ R }
```

De basisstrategie van dit algoritme is om de vertices in twee sets onder te verdelen: één voor alle vertices waarvan de afstand bekend is en één voor de onbekende vertices.

```
S: set of vertex;
Q: set of vertex;
```

Waarbij *S* de set is van bekende vertices en *Q* het restant van de graaf. Via een greedy algoritme wordt de set *S* vergroot totdat hij de gehele set *V* beslaat. De bijbehorende invarianten zijn:

```
P0: { S ⊆ V ∧ ∀v: v ∈ S: d(v) = δ(s,v) }
P1: { Q = V \ S }
```

Gegeven deze aanpak, volgt het volgende algoritme:

```
const
      V: set of vertex;
      E: set of vertex × vertex;
      w: vertex × vertex → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }
var
      d: vertex → int;
      S: set of vertex;
      Q: set of vertex;

S := ∅;
Q := V;
{ P0 ∧ P1 }

do Q ≠ ∅ → { Variante Functie: #Q }
      invariant
            P0: { S ⊆ V ∧ ∀v: v ∈ S: d(v) = δ(s,v) }
            P1: { Q = V \ S }

      var u: vertex;
      u := "een of andere vertex uit Q";
      Q := Q \ {u};
      S := S ∪ {u};
      d := ...;
      { P0 ∧ P1 }
od;
{ P ∧ (Q = ∅) ⇒ R }
```

3.3.3 De vertex u

We proberen of het mogelijk is om bij het kiezen en overhevelen van de vertex *u* er voor te zorgen dat de invarianten bewaard blijven. We hoeven dan bij het aanpassen van *d* alleen maar te kijken naar de waardes van vertices die niet deel uitmaken van de set *S*.

We kiezen "een of andere vertex *u*". Formeler gezien: het kiezen van een vertex uit *Q* doen we met de volgende actie:

```
u := (some v: v ∈ Q: PP(v))
```

Wat wil zeggen dat we een vertex *v* uit *Q* kiezen met een nader te bepalen eigenschap *PP(v)*. Zelfs als elke vertex in *Q* voldoet, geldt deze stellen. We zoeken dan een vertex met de eigenschap *true*. Al vlug zullen we echter een specifieke vertex nodig hebben.

De preconditie van deze keuze is dat er ten minste één zo'n vertex bestaat. De postconditie is dat de eigenschap geldt voor de gekozen vertex.

```
{ ∃v: v ∈ Q: PP(v) }
u := (some v: v ∈ Q: PP(v));
{ PP(u) }
```

Om ervoor te zorgen dat aan de preconditie van de some operatie wordt voldaan, voeren we een nieuwe invariant in:

P_2: $Q \neq \emptyset \Rightarrow (\exists v: v \in Q: PP(v))$

Deze invariant moet bewezen worden bij initialisatie en elke iteratie van de loop.

Het gehele programma kan als volgt worden geannoteerd:

```
const
      V: set of vertex;
      E: set of vertex × vertex;
      w: vertex × vertex → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }
var
      d: vertex → int;
      S: set of vertex;
      Q: set of vertex;

S := ∅;
Q := V;
"initialiseer P2";
{ P0 ∧ P1 ∧ P2 }

do Q ≠ ∅ → { Variante Functie: #Q }
      invariant
            P0: { S ⊆ V ∧ ∀v: v ∈ S: d(v) = δ(s,v) }
            P1: { Q = V \ S }
            P2: { Q ≠ ∅ ⇒ ( ∃v: v ∈ Q: PP(v) ) }

      var u: vertex;
      { ∃v: v ∈ Q: PP(v) }
      u := (some v: v ∈ Q: PP(v))
      { PP(u) }

      Q := Q \ {u};
      S := S ∪ {u};
      d := ...;
      { P0 ∧ P1 ∧ P2 }
od;
{ P ∧ (Q = ∅) ⇒ R }
```

3.3.4 Bepalen van PP

We zoeken een waarde voor *PP*, dus een wenselijke eigenschap voor de vertex *u*. Als we kijken naar invariant P_0, dan zien we:

$P_0(S := S \cup \{u\})$
$\equiv$ { def P_0 }
$(\forall v: v \in S: d(v) = \delta(s,v))(S := S \cup \{u\})$
$\equiv$ { substitutie }
$(\forall v: v \in S \cup \{u\}: d(v) = \delta(s,v))$
$\equiv$ { afsplitsing }
$(\forall v: v \in S: d(v) = \delta(s,v)) \wedge d(u) = \delta(s,u)$
$\Leftarrow$ { def P_0 }
$P_0 \wedge d(u) = \delta(s,u)$

We nemen voor *PP* daarom:

$PP(u): d(u) = \delta(s,u)$

En we zien dat de uitbreiding van *S* met een *u* waarvoor *PP* geldt de beide invarianten P_0 en P_1 in stand houdt.

Wat betreft de initialisatie weten we, vanwege H_1 en de axioma's over kortste paden:

$\delta(s,s) = 0$

Waardoor we P_2 kunnen initialiseren.

```
const
      V: set of vertex;
      E: set of vertex × vertex;
      w: vertex × vertex → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }
var
      d: vertex → int;
      S: set of vertex;
      Q: set of vertex;

S := ∅;
Q := V;
d(s) := 0;
{ P0 ∧ P1 ∧ P2 }

do Q ≠ ∅ → { Variante Functie: #Q }
      invariant
            P0: { S ⊆ V ∧ ∀v: v ∈ S: d(v) = δ(s,v) }
            P1: { Q = V \ S }
            P2: { Q ≠ ∅ ⇒ ( ∃v: v ∈ Q: d(v) = δ(s,v) ) }

      var u: vertex;
      u := (some v: v ∈ Q: d(v) = δ(s,v))
      { d(v) = δ(s,v) }

      Q := Q \ {u};
      S := S ∪ {u};
      { P0 ∧ P1 }
      d := ...;
      { P0 ∧ P1 ∧ P2 }
od;
{ P ∧ (Q = ∅) ⇒ R }
```

3.3.5 Aanpassen van d

Vervolgens dient het array *d* bijgewerkt te worden, zodanig dat invariant P_2 bewerkstelligd wordt en de invarianten P_0 en P_1 niet geschaad worden.

P_0 en P_1 hebben alleen betrekking op de *d*-waardes voor vertices in *S*. Door het array alleen te veranderen voor waardes buiten *S* blijven de invarianten in stand. Het is zaak om het array *d* zodanig aan te passen, dat er ten minste één "geschikte vertex" in het array zit. Dat wil zeggen:

$$\exists q:\ q \in Q:\ d(q) = \delta(s,q)$$

Ten eerste kijken we naar de definitie van de kortste padsafstand (we maken voor het gemak een gevalsonderscheid op bereikbaarheid):

$$\delta(i,j) = \left\{ \begin{array}{l} \min p : i \rightsquigarrow^{p} j : w(p) \\ \infty \end{array} \right\} \left\{ \begin{array}{l} als\ j\ bereikbaar\ is\ uit\ i \\ anders \end{array} \right\}$$

Het eenvoudigste geval is hier wanneer er een vertex onbereikbaar is. Door de *d*-waarde van zo'n vertex op oneindig te zetten wordt aan P_2 voldaan. Laten we deze onbereikbare vertices voor het moment even eenvoudigweg initialiseren op oneindig en ze verder negeren en ervan uitgaan dat de graaf volledig bereikbaar is.

We zoeken dan naar een *q*, bereikbaar uit *s*, waarvan de afstand te berekenen valt uit de bekende afstanden van de vertices in *S*.

We stellen dat zo'n gezochte *q* te vinden is in de buren van de set *S*:

$$S.adj = \{\ v \mid v \in Q \wedge \exists x:\ x \in S:\ x \rightarrow v\ \}$$

We weten dat *S.adj* niet leeg is, omdat *Q* dat niet is en de graaf bereikbaar is.

Neem een willekeurig kortste pad, met eindvertex *j* in *Q*. Dan:

$$s \rightsquigarrow p \rightarrow q \rightsquigarrow j\ \text{"is een kortste pad"}$$
$$\text{met:}$$
$$p \in S$$
$$q \in Q$$

Door de kortste subpaden eigenschap is het volgende pad ook een kortste pad:

$$s \rightsquigarrow p \rightarrow q\ \text{"is een kortste pad"}$$
$$\text{met:}$$
$$p \in S$$
$$q \in Q$$

De vertex *q* maakt logischerwijs deel uit van de set *S.adj*, waardoor dus geldt:

$$\exists p,q:\ p \in S \wedge q \in S.adj \wedge p \rightarrow q:\ s \rightsquigarrow p \rightarrow q\ \text{"is een kortste pad"}$$

We zoeken een manier de *d-waarde* van *q* te berekenen.

```
s ∈ S
∀i: i ∈ S: d(i) = δ(s,i)
Q ≠ ∅
P0..2
H0..2
```

```
{ zie boven }
∃p,q: p ∈ S ∧ q ∈ S.adj ∧ p → q: s ~> p → q "is een kortste pad"
```

```
neem p, q:
p ∈ S
q ∈ S.adj
p → q
s ~> p → q "is een kortste pad"
```

```
{ kortste pad uitbreiding ∧ P0 }
δ(s, q) = d(p) + w(p, q)
```

```
neem p': p' ∈ S ∧ p' → q
```

```
{ driehoeksongelijkheid ∧ P0 }
δ(s, q) ≤ d(p') + w(p', q)
```

```
{ noem Iq: { v | v ∈ S ∧ v → q } }
∀p': p' ∈ Iq: δ(s, q) ≤ d(p') + w(p', q)
{ p ∈ I }
∃p': p' ∈ Iq: δ(s, q) = d(p') + w(p', q)
{ definitie minimum }
δ(s, q) = min(p': p' ∈ Iq: d(p') + w(p', q))
```

```
∀p,q: p ∈ S ∧ q ∈ S.adj ∧ p → q ∧ s ~> p → q "is een kortste pad":
      δ(s, q) = min(p': p' ∈ Iq: d(p') + w(p', q))
```

```
stel:
∀q': q' ∈ S.adj:
     d(q') = min(p': p' ∈ S ∧ p' → q': d(p') + w(p', q'))
```

```
{ domeinbeperking }
δ(s, q) = d(q)
```

Het bewijst eindigt met de stelling dat

$$\delta(s, q) = d(q)$$

Waar we inderdaad naar op zoek waren. Om dit te laten gelden, moet de aanname in de laatste vlag waar zijn. We introduceren hiervoor een nieuwe invariant:

```
P3: ( ∀v: v ∈ S.adj:
        d(v) = (min x: x ∈ S ∧ x → v: (d(x) + w(x, v))
    )
```

Na iedere slag moet het array *d* worden aangepast om aan deze invariant te voldoen. De actie die hier van belang is, is de uitbreiding van *S*:

```
P₃(S := S ∪ {u})
≡ { definitie van P₃ }
( ∀v: v ∈ (S∪{u}).adj:
            d(v) = (min x: x ∈ S∪{u} ∧ x → v: (d(x) + w(x, v))
)
≡ { want: (S∪{u}).adj
           ≡ (S.adj\{u}) ∪ u.adj
           ≡ (S.adj\{u ∪ u.adj}) ∪ (u.adj\S) }
( ∀v: v ∈ (S.adj)\{u ∪ u.adj}:
            d(v) = (min x: x ∈ S∪{u} ∧ x → v: (d(x) + w(x, v))
)
∧
( ∀v: v ∈ (u.adj)\S:
            d(v) = (min x: x ∈ S∪{u} ∧ x → v: (d(x) + w(x, v))
)
⇐ { voor de eerste conjunct: (#x: x ∈ {u}: x → v) = 0 }
P₃ ∧
( ∀v: v ∈ (u.adj)\S:
            d(v) = (min x: x ∈ S∪{u} ∧ x → v: (d(x) + w(x, v))
)
≡ { gevalsonderscheid, afsplitsing en P₃ }
P₃ ∧
( ∀v: v ∈ (u.adj \ S):
      d(v) =
      (
            [] v ∈ S.adj → min(d(v), d(u) + w(u,v)
            [] v ∉ S.adj → d(u) + w(u,v)
      )
)
```

De nieuwe invariant P_3 is in feite een versterking van de oude P_2. P_2 is daarom overbodig en kan worden weggelaten.

Dit alles in beschouwing nemende wordt de code:

```
const
      V: set of vertex;
      E: set of edge;
      w: vertex × vertex → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }

var
      d: vertex → int;
      S: set of vertex;
      Q: set of vertex;

d(s) := 0;
S := ∅;
Q := V;
"voor alle onbereikbare vertices: d(v) := ∞";
{ P0..3 }

do Q ≠ ∅ →
      invariant
            P0: { S ⊆ V ∧ ∀v: v ∈ S: d(v) = δ(s,v) }
            P1: { Q = V \ S }
            P3: { ∀v: v ∈ S.adj:
                  d(v) = (min x: x ∈ S ∧ x → v: (d(x) +
w(x,v))
                }

      var u: vertex;
      u := (some v: v ∈ Q: d(v) = δ(s,v));
      Q := Q \ {u};
      S := S ∪ {u};
      foreach v ∈ (u.adj \ S) →
            if
            [] v ∈ S.adj → d(v) := min(d(v), d(u) + w(u,v)
            [] v ∉ S.adj → d(v) := d(u) + w(u,v)
            fi;
      rof
      { P0..3 }
od;
{ ∀i: i ∈ V: d(i) = δ(s,i) }
```

3.3.6 Initialisatieverfijning en Onbereikbare Vertices

In de vorige paragraaf hebben we onbereikbare vertices buiten beschouwing gelaten. We kunnen deze vertices correct initialiseren en tegelijk ons programma een stuk vereenvoudigen door de volgende invariant toe te voegen:

P_4: $\forall v: v \in Q \setminus S.adj: d(v) = \infty$

Merk op hoe alle onbereikbare vertices niet in *S.adj* kunnen zitten.

Helaas moet deze invariant nog ietwat aangepast worden voor de initialisatie van de loop. Eigenlijk gebruiken we de set "buren van *S*" als volgt:

$S=\emptyset \Rightarrow S.adj = \{s\}$

Hierdoor begint de loop met het kiezen van startvertex *s*. De invariant moet wel ietwat worden bijgeschaafd om in dit speciale geval te voorzien:

$$P_4\colon\ S=\varnothing \Rightarrow \forall v\colon\ v \in (Q \setminus \{s\})\colon\ d(v) = \infty$$
$$\neg(S=\varnothing) \Rightarrow \forall v\colon\ v \in (Q \setminus S.adj)\colon\ d(v) = \infty$$

Het invariantiebewijs van deze conditie maakt gebruik van de kennis dat *s* na één slag van de loop niet meer tot *Q* zal behoren:

$$\neg(S=\varnothing) \Rightarrow s \notin Q$$
$$\equiv$$
$$\neg(S=\varnothing) \Rightarrow (Q \setminus S.adj) = (Q \setminus S.adj \cup \{s\})$$

Voor het bewijs van deze sub-invariant is de kennis dat *s* de eerstgekozen vertex is vitaal, wat leidt tot alweer een invariant:

$$S=\varnothing \Rightarrow (\text{some } v\colon\ v \in Q\colon\ d(v) = \delta(s,v))) = s$$

Al met al is dit een weinig elegante oplossing. We zullen ervoor kiezen om in dit geval onze code ietwat aan te passen in het belang van het bewijs.

Wat we gaan doen is de eerste slag van de loop buiten de loop halen, waardoor het gevalsonderscheid niet meer nodig is. Dit verandert de initialisatiestatements van het programma enigszins en staat ons toe de originele handzame invariant P_4 te gebruiken:

$$P_4\colon\ \forall v\colon\ v \in (Q \setminus S.adj)\colon\ d(v) = \infty$$

Met deze nieuwe invariant kunnen we de keuze of vertex *v* al dan niet tot *S.adj* behoort weglaten, er geldt immers:

$$\min(\infty,\ d(u) + w(u,v))$$
$$=$$
$$d(u) + w(u,v)$$

Daarnaast is ook de controle of *v* tot *S* behoort niet meer noodzakelijk omdat geldt:

$$\min(d(v),\ d(u) + w(u,v)) \wedge v \in S$$
$$=\ \{\ d(v) = \delta(s,v)\ \}$$
$$d(v)$$

Wat vereist is in de code is een initialisatie van *d* op oneindig voor alle vertices behalve *s* en het apart uitvoeren van een eerste slag van de loop.

```
const
      V: set of vertex;
      E: set of edge;
      w: vertex × vertex → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }

var
      d: vertex → int;
      S: set of vertex;
      Q: set of vertex;

d(s) := 0;
Q := V\{s};
S := {s};
foreach v: v ∈ V\s.adj → d(v) := ∞ rof;
foreach v: v ∈ s.adj → d(v) := d(s)+w(s,v) rof;
{ P0..4 }

do Q ≠ ∅ →
      invariant
            P0: { S ⊆ V ∧ ∀v: v ∈ S: d(v) = δ(s,v) }
            P1: { Q = V \ S }
            P3: { ∀v: v ∈ S.adj:
                  d(v) = (min x: x ∈ S ∧ x → v: (d(x) +
w(x,v))
                }
            P4: { ∀v: v ∈ (Q \ S.adj): d(v) = ∞ }

      var u: vertex;
      u := (some v: v ∈ Q: d(v) = δ(s,v));
      Q := Q \ {u};
      S := S ∪ {u};
      foreach v ∈ u.adj →
            d(v) := min(d(v), d(u) + w(u,v)
      rof
      { P0..4 }
od;
{ ∀i: i ∈ V: d(i) = δ(s,i) }
```

3.3.7 Kiezen van u

Ten slotte moet er nog een vertex *u* uit de buren van de set *S* gevonden worden waarvoor de *d*-waarde gelijk is aan de kortste padlengte.

Zoals eerder beredeneerd bestaat deze vertex.

We nemen een vertex in *Q* met een *minimale d* waarde en beweren dat de *d* waarde van deze vertex gelijk is aan de kortste padlengte tot die vertex.

Stel dat de vertex *u* met minimale *d* waarde incorrect is, dus er bestaat een korter pad naar *u*. Dit kortere pad moet wel via een van de buren van de set *S* lopen, als volgt:

$$s \rightsquigarrow p \rightarrow q \rightsquigarrow u$$

met

```
s ~> p ∈ S
q ∈ Q
```

We weten voor alle mogelijke subpaden *s ~> p → q* de minimale padlengte, zoals opgeslagen in array *d*. Omdat echter *d(u) ≤ d(q)*, kan dit pad nooit korter zijn dan het pad dat direct naar *u* gaat.

Met deze laatste aanpassing is het programma compleet:

```
const
      V: set of vertex;
      E: set of edge;
      w: vertex × vertex → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }

var
      d: vertex → int;
      S: set of vertex;
      Q: set of vertex;

d(s) := 0;
Q := V\{s};
S := {s};
foreach v: v ∈ V\s.adj → d(v) := ∞ rof;
foreach v: v ∈ s.adj → d(v) := d(s)+w(s,v) rof;
{ P0..4 }

do Q ≠ ∅ →
      invariant
            P0: { S ⊆ V ∧ ∀v: v ∈ S: d(v) = δ(s,v) }
            P1: { Q = V \ S }
            P3: { ∀v: v ∈ S.adj:
                  d(v) = (min x: x ∈ S ∧ x → v: (d(x) +
w(x,v))
                }
            P4:{ ∀v: v ∈ (Q \ S.adj): d(v) = ∞ }

      var u: vertex;
      u := (some v: v ∈ Q: d(v) = (min d': d' ∈ Q: d'(v)));
      Q := Q \ {u};
      S := S ∪ {u};
      foreach v ∈ u.adj →
            d(v) := min(d(v), d(u) + w(u,v)
      rof
      { P0..4 }
od;
{ ∀i: i ∈ V: d(i) = δ(s,i) }
```

3.4 Incremental Development in GCL

In dit hoofdstuk leiden we, volgens een incrementele aanpak, een algoritme af voor het berekenen van het kortste pad af. We schrijven en bewijzen complete, functionele en implementeerbare programma's die op elkaar voortborduren en geleidelijk in complexiteit stijgen.

De algoritmes zijn compleet in die zin dat ze geen gaten bevatten zoals dat wel het geval was bij Stepwise Refinement. Ze zijn functioneel in dat elk algoritme een eigen probleem oplost, gerelateerd aan het kortste padprobleem. Ten slotte zijn deze algoritmes (vrijwel) rechtstreeks te vertalen in Perfect Developer code waardoor elke stap een werkend programma oplevert.

Elk algoritme is een uitbreiding van zijn voorganger, waardoor bewijzen grotendeels intact blijven en we ons alleen op de nieuwe constructies hoeven te richten.

De stappen zijn:

- Generiek kleuringsalgoritme
  waarin we de basis voor onze algoritmes opzetten, een algoritme dat de vertices van de graaf kleurt van wit naar grijs naar zwart.
- Bereikbaarheidsdeterminatie
  waarin we ons kleuringsalgoritme inzetten om bereikbaarheid te berekenen.
- Bereikbaarheid met witness
  waarin we naast het berekenen van bereikbaarheid ook een pad retourneren.
- Depth First Search
  waarin we bereikbaarheid niet meer willekeurig, maar volgens een depth first methode berekenen.
- Breadth First Search
  waarin we bereikbaarheid volgens een breadth first methode berekenen.
- Unit Distance
  waarin we de kortste afstand gemeten in aantal gepasseerde edges berekenen.
- Kortste Pad
  waarin we de kortste afstand gemeten in totaal van de gewichten van de gepasseerde edges berekenen.

Gegevens

De context van ons algoritme is een graaf, bestaande uit vertices en edges tussen die vertices, vergezeld van een startvertex, een doelvertex en een gewichtsfunctie. Bij elk volgend algoritme worden de onderstaande gegevens aangenomen:

```
type
      Edge: Vertex × Vertex;

const
      V: set of Vertex,
      E: set of Edge,
      weight: function of (Edge → nat),
      start: Vertex,
      target: Vertex;

known
      E ⊆ V×V,
      start ∈ V,
      target ∈ V;
```

3.4.1 Kleuringsalgoritme

Inleiding

We wijken af van het Incremental Development principe. We specificeren namelijk een programma dat volgens een greedy algoritme over de vertices itereert, maar welke criteria hierbij worden gebruikt en wat er precies met de behandelde vertices gebeurt, laten we open.

Door deze gaten in volgende iteraties te vullen en uit te breiden zullen we naar het uiteindelijke kortste pad algoritme toewerken.

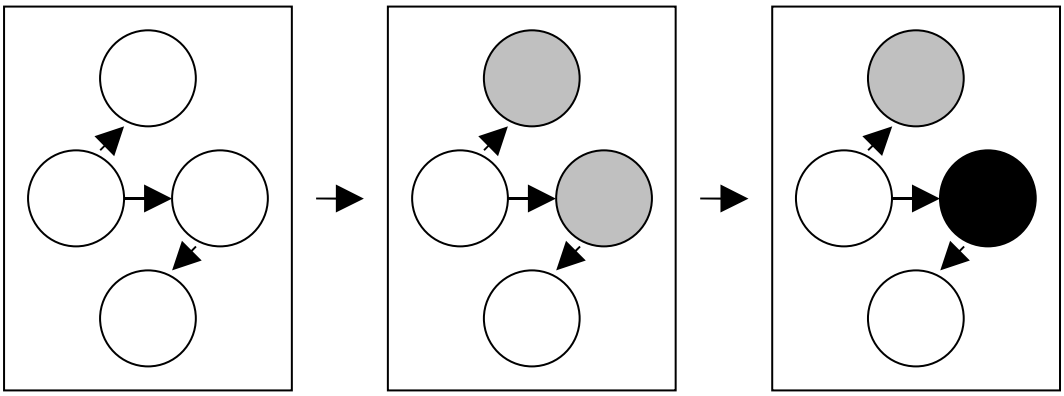
Deze stap is interessant om los uit te werken omdat we hierin het raamwerk voor het algoritme opstellen en zaken als voortgang en terminatie kunnen bewijzen.

We verdelen de vertices van de graaf onder in drie disjuncte sets: wit, zwart en grijs.

wit: Dit is de beginstatus van vertices. Witte vertices zijn "nog niet bekeken" door het algoritme. Er is niets over bekend.

zwart: We kleuren vertices zwart wanneer we ze "behandeld" hebben. In het kortste pad algoritme wordt een vertex zwart gekleurd wanneer de kortste afstand ervan berekend is. Iedere slag van het algoritme wordt exact één grijze vertex zwart gekleurd.

grijs: We kiezen de zwarte vertex steeds uit de grijze vertices. We gebruiken deze transitieset tussen wit en zwart voor vertices met bekende eigenschappen (zoals dat ze bereikbaar zijn vanuit de startvertex). Iedere iteratieslag kleuren we nul of meer witte vertices grijs.



Vertices worden achtereenvolgens wit, grijs en zwart gekleurd

Nieuwe Begrippen

De belangrijkste bewijsverplichtingen van dit algoritme zijn voortgang en terminatie.

Iedere slag wordt er precies één grijze vertex zwart gekleurd. Zwarte vertices veranderen nooit van kleur. Dit garandeert voortgang.

Daarnaast is het aantal zwarte vertices naar boven begrensd doordat de zwarte vertices een subset is van V en V eindig is. Dit garandeert terminatie.

Hetzelfde geldt voor het grijs kleuren van vertices: een vertex kan slechts één maal grijs worden gekleurd en het aantal vertices is eindig.

Formeel

We coderen de partitionering middels drie sets: B, G en W. Een vertex heeft exact één kleur, wat leidt tot de volgende invarianten:

$$P_0: B \cup G \cup W = V$$
$$P_1: B \cap G = \varnothing$$
$$P_2: B \cap W = \varnothing$$
$$P_3: G \cap W = \varnothing$$

Het bijbehorende programma ziet er als volgt uit:

```
var
      B, G, W: set of Vertex;

B := ∅;
G := {start};
W := V \ {start};

do G ≠ ∅ → { VF: #V - #B, inv: P0..3 }
      { ∃h: Vertex: h ∈ G }
      let h: Vertex satisfy h ∈ G;

      B, G := B ∪ {h}, G \ {h};

      do forall x ∈ V →
            if
                  x ∈ W → W,G := W \ {x}, G ∪ {x}
            []    x ∈ G → skip
            []    x ∈ B → skip
            fi
      od
od
```

Merk op dat we in deze eerste versie alle witte vertices meteen grijs kleuren, zodat we de keuze voor welke vertices grijs te kleuren dadelijk met een kleine modificatie kunnen implementeren.

Als variante functie van de hoofdlus gebruiken we:

```
#V - #B
```

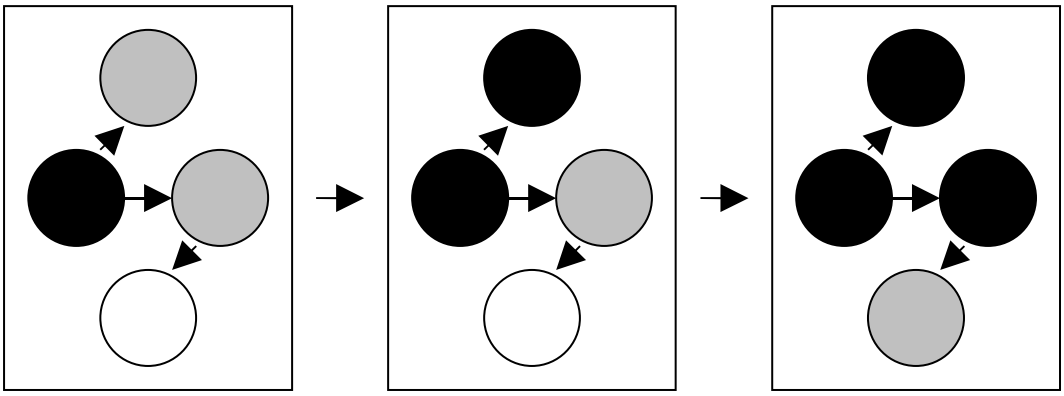
De *let* statement aan het begin van de lus roept een bewijsverplichting op: er moet ten minste één vertex zijn waarvoor de *satisfy* clausule geldt. Dat dit zo is volgt uit de guard van de lus.

3.4.2 Bereikbaarheid

Inleiding

We beperken het grijs kleuren van vertices. We ervoor om enkel opvolgers van zwarte vertices grijs te maken, zodat we de basis hebben van een bereikbaarheidsalgoritme.

We kleuren, wanneer we een grijze vertex zwart maken, alle directe opvolgers van die vertex grijs. Op deze manier wordt er een bereikbaarheidsboom opgebouwd. Wanneer er geen grijze vertices meer zijn, zijn alle bereikbare vertices vanuit start zwart gekleurd.



Opvolgers van zwarte vertices worden grijs gekleurd

Bereikbaarheid volgens kleinste dekpunt

We gebruiken een definitie van de bereikbaarheidsset zoals geïntroduceerd in [23], namelijk via een kleinste dekpunt.

We definiëren bereikbaarheid als volgt:

- *start* is bereikbaar
- alle opvolgers van bereikbare vertices zijn bereikbaar
- alle overige vertices zijn niet bereikbaar

Laten we de set van bereikbare vertices vanuit *start* aangeven met *R(start)*:

```
R(start) = "de bereikbare vertices uit start"
```

Dan, volgens onze definitie:

```
start ∈ R(start)
```

Alle opvolgers van bereikbare vertices, dus die vertices die via één enkele edge bereikbaar zijn vanuit een vertex in *B*, zijn bereikbaar:

```
start ∪ successors(start) ∈ R(start)

met (w ∈ successors(v) ≡ <v,w> ∈ E)
```

We passen de tweede regel nogmaals toe (en breiden de successorfunctie uit zodat deze ook hele sets als invoer kan nemen):

```
start ∪ successors(start) ∪ successors(successors(start))
∈ R(start)

met (w ∈ successors(V) ≡ ∃v: v ∈ V: <v,w> ∈ E)
```

Aangezien de successorfunctie de enige manier is om deze groeiende set van bekende bereikbare vertices uit te breiden en de set van bereikbare vertices eindig is, zal het herhaaldelijk toepassen van deze functie uiteindelijk "stabiliseren". Als we de groeiende set van *successors* van *start* als functie weergeven:

```
f(0) = start
f(n+1) = f(n) ∪ successors(f(n))
```

Dan zoeken we een punt waarop het toevoegen van *successors(f(n))* aan *f(n)* de set niet meer vergroot. Met andere woorden, we zoeken een dekpunt van *f*:

```
∃n:int:
      (f(n+1) = f(n) ∪ successors(f(n)) = f(n) = R(start))
```

Met wat rekenwerk:

```
f(n+1) = f(n) ∪ successors(f(n)) = f(n)
≡
successors(f(n)) ⊆ f(n)
```

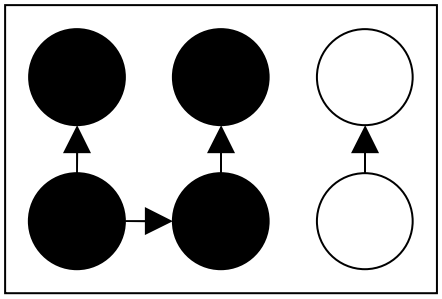
We zoeken naar de set *f(n)* waarvoor dit laatste geldt.

```
∃R: set: successors(R) ⊆ R ∧ f(n) = R
```

Helaas geldt niet voor elke mogelijke set *R* waarvoor geldt dat de *succesoren* een subset zijn van *R*. Neem bijvoorbeeld het volgende geval, waarbij *V* alle vertices in de graaf bevat:

```
successors(V) ⊆ V
```

Deze stelling is correct, maar *f(n)* is vaak niet gelijk aan *V*:



Er geldt

$$\neg\exists n: nat: f(n) = V$$

Onze functie *f* retourneert namelijk alleen subsets van de bereikbaarheidsset:

$$\forall n: nat: f(n) \subseteq R(start)$$

Door deze eis op te nemen komen we uit op de volgende stelling:

$$R \subseteq successors(R)$$
$$\wedge$$
$$R \subseteq R(start)$$
$$\Rightarrow$$
$$R = R(start)$$

Nieuwe Begrippen

Door het grijs kleuren van buren (directe opvolgers) van zwarte vertices, houden we een nieuwe invariant in stand met betrekking tot de grijze vertices: namelijk dat alle buren van zwarte vertices grijs (of zwart) zijn.

Met behulp van deze invariant kunnen we bewijzen dat ons algoritme voldoet aan de postconditie dat de set van zwarte vertices gelijk is aan de bereikbare vertices vanuit start. We definiëren de bereikbare vertices daarbij volgens de methode uit vorige paragraaf.

Onze postconditie is dat de set *B* gelijk is aan de bereikbare vertices vanuit *start* (namelijk *R(start)*):

$$R(start) = B$$

Volgens de vorige paragraaf volgt dit uit de volgende conjunctie:

$$B \subseteq R(start) \wedge successors(B) \subseteq B$$

We kunnen deze conjuncten afzonderlijk bewijzen aan de hand van ons programma.

Formeel

```
var
      out result: bool
      B, G, W: set of Vertex,


B := ∅;
G := {start};
W := V \ {start};

do G ≠ ∅ →
      let h: Vertex satisfy h ∈ G;

      B, G := B ∪ {h}, G \ {h};

      do forall x ∈ successors(h) →
            if
                  x ∈ W → W,G := W \ {x}, G ∪ {x}
            []    x ∈ G → skip
            []    x ∈ B → skip
            fi
      od
od

{ B = R(start) }
result := target ∈ B;
```

De nieuwe invariant definiëren we als volgt:

P_4: $successors(B) \setminus B \subseteq G$

Merk op dat de invariant P_4 spreekt over een subset, geen gelijkheid. Na de initialisatie bestaat *G* namelijk uit *start*, terwijl *B* leeg is en zou een gelijkheid incorrect zijn.

Zoals gezegd krijgt ons algoritme de postconditie:

Q_0: $B = R(start)$

Te bewijzen is dus:

$P_{0..4} \land G = \emptyset$
$\Rightarrow$
$B = R(start)$

En, zoals beredeneerd in 7.3.2:

$B \subseteq R(start) \land successors(B) \subseteq B$
$\Rightarrow$
$B = R(start)$

We dienen dus te bewijzen dat de invarianten en de negatie van de guard tezamen de twee conjuncten uit de bovenstaande stelling impliceren.

We schetsen kort het bewijs voor elk van de conjuncten:

- $B \subseteq R(start)$

  Bewijzen dat elke vertex in *B* bereikbaar is vanuit *start* kan worden gedaan door een witness-functie te introduceren. We zullen gedurende het opbouwen van *B* een functie *paths* bijhouden die paden van start naar vertices in *B* opslaat. We gaan hier in een volgende stap dieper op in.

- $successors(B) \subseteq B$

  We kunnen de nieuwe invariant P_4 en de negatie van de loop-guard gebruiken om dit te bewijzen:

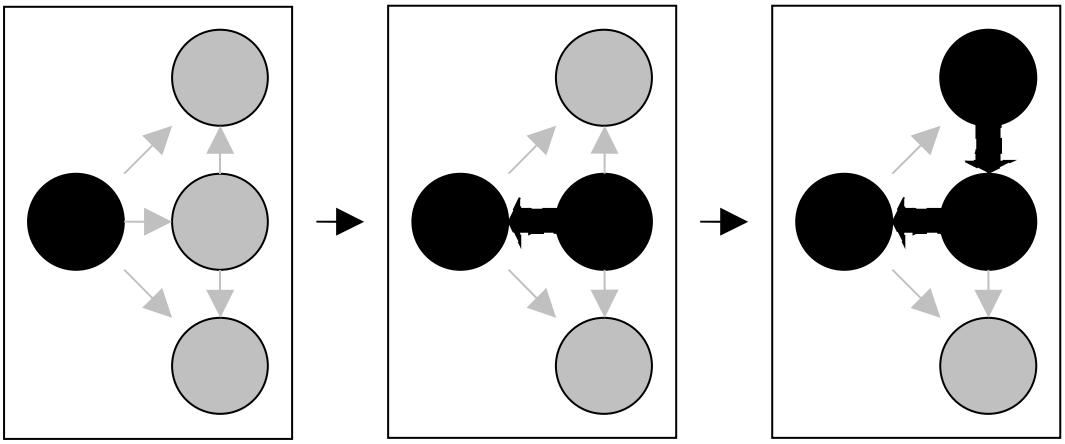
  $$successors(B) \setminus B \subseteq G \wedge G = \emptyset$$
  $$\Rightarrow$$
  $$successors(B) \subseteq B$$

3.4.3 Pad bijhouden

Inleiding

Zoals bij de vorige stap aangegeven bouwen we tijdens het uitbreiden van de set van zwarte vertices een boomstructuur op. We kunnen deze structuur expliciet maken door voor elke grijze vertex een verwijzing naar zijn voorganger op te slaan.

Het effect hiervan is dat we voor een bereikbare vertex het pad ernaartoe kunnen reconstrueren door de voorgangers te volgen tot aan de startvertex.



Tijdens het zwart kleuren van vertices wordt het pad opgeslagen

Nieuwe Begrippen

We introduceren een voorgangerfunctie *pred*, voor elke niet-witte vertex. Het is belangrijk dat deze functie well founded is, met als minimum element de startvertex. Als we de *pred* functie herhaaldelijk toepassen dienen we te "eindigen" bij de startvertex.

Formeel stellen we de volgende eis aan grijze en zwarte vertices:

$$\forall v: v \in G \cup B: \exists i: i \geq 0: pred^i(v) = start$$

Waarbij:

$$pred^0(v) = v$$
$$pred^{i+1}(v) = pred^i(pred(v))$$

Daarbij zijn predecessors verbonden via een edge:

$$\forall v: v \in (G \cup B) \setminus \{start\}: \langle pred(v), v \rangle \in E$$

Als extra postconditie geldt dat voor elke zwarte vertex een *pred*-lijst bestaat, wat betekent dat er voor elke zwarte vertex een pad bestaat.

Het reconstrueren van dit pad is eenvoudig (we gaan verder niet in op de details hiervan):

"het pad van start naar v"
=
<start, $pred^{i-1}$(v), $pred^{i-2}$(v), ... , pred(v), v>

Formeel

```
var
      B, G, W: set of Vertex,
      out pred: function of Vertex → Vertex;

B := ∅;
G := {start};
W := V \ {start};

do G ≠ ∅ →
      let h: Vertex satisfy h ∈ G;

      B, G := B ∪ {h}, G \ {h};

      do forall x ∈ successors(h) →
            if
                  x ∈ W → W,G := W \ {x}, G ∪ {x};
                           pred(x) := h
            []    x ∈ G → skip
            []    x ∈ B → skip
            fi
      od
od
```

We introduceren de nieuwe invariant:

P_5: $\forall v: v \in G \cup B: \exists i: i \geq 0: pred^i(v) = s$
P_6: $\forall v: v \in G \cup B: v = start \vee \langle pred(v), v \rangle \in E$

Voor uitbreiding van *G* met vertex *y*, waarvoor geldt: $\langle x,y \rangle \in E$ en $x \in B$ bewijzen we:

{ volgens P_4 }
$\exists i::pred^i(x) = start$
{ we kennen aan pred(y) x toe, dit mag want $\langle x,y \rangle \in E$ }
$pred(y) = x$
$\Rightarrow$
$pred^{i+1}(y) = pred^i(x) = start$

De invariant leidt tot de nieuwe postcondities:

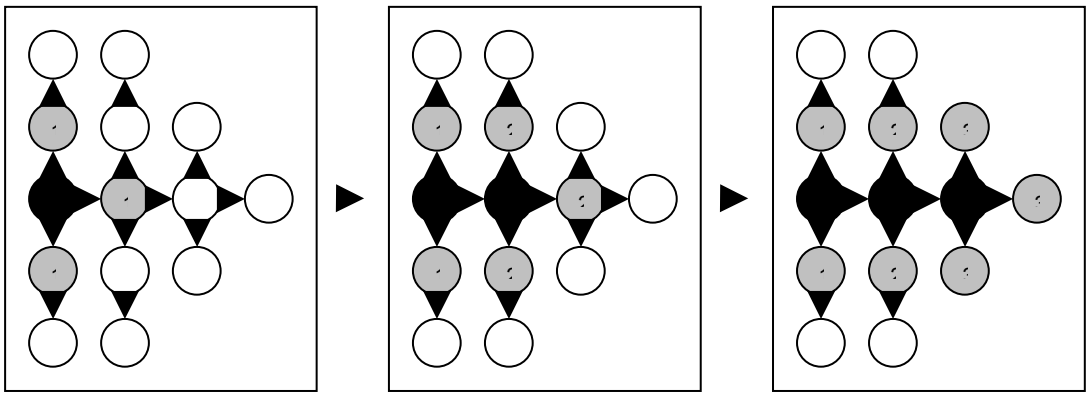
Q_1: $\forall v: v \in B: \exists i: i \geq 0: pred^i(v) = start$
Q_2: $\forall v: v \in B: v = start \vee \langle pred(v), v \rangle \in E$

Wat genoeg informatie is om een pad af te leiden van *start* naar *v*.

3.4.4 Depth First Search

Inleiding

Wanneer we de keuze van welke vertex zwart te kleuren beperken komen we in de buurt van de bekendere algoritmes [3]. Nemen we bijvoorbeeld telkens een van de "jongste" grijze vertices dan wordt ons algoritme een Depth First Search.



Depth First Search. De laatst toegevoegde vertices (= de jongste vertices) hebben de hoogste dieptewaarde.

Nieuwe Begrippen

We breiden telkens het langste bekende pad (in termen van aantal edges) uit. We hebben al een begrip voor de lengte van een pad:

P_5: $\forall v: v \in G \mathbin{++} B: \exists i: i \geq 0: pred^i(v) = s$

De *i* in deze stelling komt overeen met de lengte van het pad.

We maken deze variabele expliciet en noemen hem *depth*. Er geldt:

$\forall v: v \in B \mathbin{++} G: pred^{depth(v)}(v) = start$

Bij uitbreiding van *B* kunnen we die vertex uit *G* kiezen met de hoogste *depth* waarde.

Formeel

```
var
      B, G, W: set of Vertex,
      out pred: function of Vertex → Vertex,
      depth: function of Vertex → nat,

B := ∅;
G := {start};
W := V \ {start};
depth(start) := 0;

do G ≠ ∅ →
      let h: Vertex satisfy h ∈ G ∧ "depth(h) is maximaal";

      B, G := B ∪ {h}, G \ {h};

      do forall x ∈ successors(h) →
            if
                  x ∈ W → W,G := W \ {x}, G ∪ {x},
                           pred(x) := h,
                           depth(x) := depth(h) + 1
            []    x ∈ G → skip
            []    x ∈ B → skip
            fi
      od
od
```

We passen invariant P_5 aan om gebruik te maken van *depth*:

P_5: $\forall v: v \in G \mathbin{++} B: pred^{depth(v)}(v) = s$

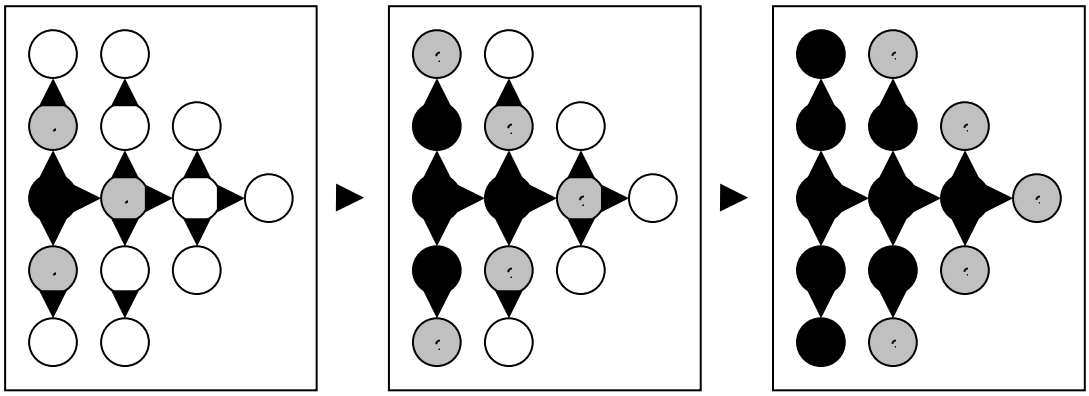
Voor uitbreiding van *G* geldt:

$\{ \langle h,x \rangle \in E \wedge h \in B \wedge x \in W \wedge pred^{depth(h)}(h) = start \}$
$G := G \cup \{x\},$
$pred(x) := h,$
$depth(x) := depth(h) + 1$
$\{ pred^{depth(x)}(x) = start \}$

3.4.5 Breadth First Search / Unit Distance

Inleiding

In plaats van de jongste grijze vertex kiezen we de oudste grijze vertex om zwart te kleuren.



Breadth First Search (tussen elk plaatje bevinden zich meerdere stappen van het algoritme). De dieptewaarde van de vertices komt overeen met de afstand.

Door vertices op deze manier zwart te kleuren ontstaat een breadth first search algoritme. Een voordeel van dit algoritme is dat de paden die op deze manier worden gevonden minimaal zijn in lengte (gemeten in aantal edges).

Nieuwe Begrippen

We kiezen ervoor telkens het kortste bekende pad uit te breiden. Doordat een uitbreiding van een pad de lengte altijd met exact één edge ophoogt is een uitbreiding van een kortste pad zelf ook een kortste pad. We zullen deze stelling in detail uitwerken in de volgende paragraaf.

De redenering in woorden is als volgt: stel dat de minimale *depth* waarde van de grijze vertices gelijk is aan *n*. Elk pad naar een grijze vertex via een andere grijze vertex zal dus altijd ten minste *n+1* edges bevatten en daardoor niet korter zijn.

Formeel

```
var
      B, G, W: set of Vertex,
      depth: function of Vertex → nat;
      pred: function of Vertex → Vertex
      out path: list of Vertex,
      out shortestdistance: int,

B := ∅;
G := {start};
W := V \ {start};
depth(start) := 0;

do G ≠ ∅ →
      let h: Vertex satisfy h ∈ G ∧ "depth(h) is minimaal";

      B, G := B ∪ {h}, G \ {h};

      do forall x ∈ successors(h) →
            if
                  x ∈ W → W,G := W \ {x}, G ∪ {x},
                           depth(x) := depth(h) + 1,
                           pred(x) := h;
            []    x ∈ G → skip
            []    x ∈ B → skip
            fi
      od
od

if
      target ∈ B → shortestdistance:= depth(target);
                   path :=
                   "het pad gegeven door pred^depth(target)(target)"
[]    target ∉ B → shortestdistance:= ∞;
                   path := <>
fi
```

We introduceren de specificatiefunctie *delta*, die de (kortste) afstand tot een vertex retourneert:

```
delta(a,b) = "de minimale afstand tussen a en b, gemeten in
              aantal edges"
```

Als postconditie stellen we dat de dieptewaarde van zwarte vertices gelijk is aan hun afstand.

Q_3: $\forall v: v \in B: depth(v) = delta(start,v)$

En we verkrijgen hieruit de nieuwe invariant P_7:

P_7: $\forall v: v \in B \cup G: depth(v) = delta(start,v)$

Hulp Invarianten

Voor het bewijs van deze invariant introduceren we een drietal hulpinvarianten. Deze invarianten gelden alleen voor deze incarnatie van ons algoritme: in de volgende stap incorporeren we de gewichten van edges, waardoor de invarianten effectief zinloos worden:

```
let d = min v: v ∈ G: depth(v)
PH0: ∀v: v ∈ B: delta(start,v) ≤ d
PH1: ∀v: v ∈ G: d ≤ delta(start,v) ≤ d + 1
PH2: ∀v: v ∈ W: d + 1 ≤ delta(start,v)
```

Ten eerste dienen we ons ervan te verzekeren dat de loop deze invarianten inderdaad in stand houdt. Als we kijken naar het zwart kleuren van een grijze vertex:

```
{ PH0..2 }
let h: Vertex satisfy h ∈ G ∧ "depth(h) is minimaal";
{ depth(h) = d }

{ depth(h)=d ∧ PH0 ∧ P6 ⇒ PH0(B := B ∪ {h}) }
B := B ∪ {h};
{ PH0..2 }

{ PH0 ∧ PH1 ⇒ PH0(G := G \ {h}) }
{ PH1 ∧ depth(h)=d ∧ P7 ⇒ PH1(G := G \ {h}) }
G := G \ {h};
{ PH0 ∧ PH1 }
```

Het bewijs van PH_2 voor het verwijderen van vertex *h* uit *G* is ietwat gecompliceerder.

Door het verwijderen van *h* uit *G* kan de waarde van *min v: v ∈ G: depth(v)* veranderen. Uit P_7 volgt:

```
(min v: v ∈ G: depth(v))(G := G \ {h})
=
min v: v ∈ G: depth(v)
óf
1 + (min v: v ∈ G: depth(v))
```

In het eerste geval blijft PH_2 triviaal gelden, aangezien de waarde van *d* niet veranderd . In het tweede geval (waarin de waarde van *d* dus wel veranderd is), geldt het volgende:

```
let d = min v: v ∈ G: depth(v)

∀v: v ∈ G: d = delta(start,v)
```

En uit PH_2:

```
∀v: v ∈ W: d ≤ delta(start,v)
```

Stel dat er een vertex *v* bestaat in *W* met *delta(start,v) = d*:
Dan bestaat er een kortste pad tussen *start* en *v* en heeft *v* in dat kortste pad een voorganger *x* met *delta(start,x)=d-1*.

Deze voorganger *x* kan volgens PH_2 niet wit zijn. Hij is dus grijs of zwart. Hij kan echter ook niet grijs zijn, omdat *∀v: v ∈ G: d = delta(start,v)*. De laatste optie is dat hij zwart is. Echter, volgens een eerder geïntroduceerde invariant P_4 zijn alle directe buren van zwarte vertices grijs - wat een tegenspraak is met het feit dat *v* wit is.

Hieruit leiden we af dat er geen vertex *v* in *W* bestaat met *delta(start,v) = d* waardoor:

```
∀v: v ∈ W: d+1 ≤ delta(start,v)
≡
PH2
```

Vervolgens wordt in de code een aantal witte buren van *h* grijs gekleurd. Deze actie schendt de hulpinvarianten niet, maar op dit moment dienen we onze invariant P_7 te bewijzen.

Bewijs van P7

Invariant P_7 luidt als volgt:

$$P_7: \forall v: v \in B \cup G: depth(v) = delta(start,v)$$

Het punt in de code waarop we moeten bewijzen dat de invariant geldig blijft is daar waar we de set *G* uitbreiden:

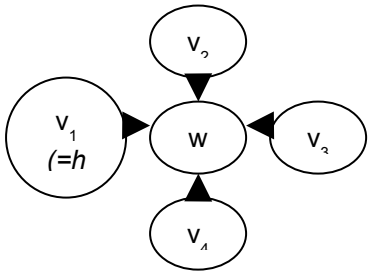
```
do forall x ∈ successors(h) →
      if
            x ∈ W → { P7 }
                    W,G := W \ {x}, G ∪ {x},
                    depth(x) := depth(h) + 1,
                    pred(x) := h;
                    { P7? }
      []    x ∈ G → skip
      []    x ∈ B → skip
      fi
od
```

In de code hierboven worden opvolgers van de vertex *h* (een grijze vertex met minimale *depth*) aan *G* toegevoegd en hun *depth* waarde wordt gelijkgesteld aan *depth(h) + 1*.

We bewijzen dat de *delta* van opvolgers van *h* gelijk is aan *depth(h) + 1*, waardoor P_7 in stand wordt gehouden.

We gebruiken een eigenschap van *delta*: namelijk dat de afstand naar een vertex (ongelijk aan *start*) gelijk is aan 1 plus het minimum van de afstanden naar de voorgangers.

We bekijken een opvolger $w \in W$ van *h*. De inkomende vertices van *w* noemen we v_1 t/m v_n:



merk op dat ($\exists i::v_i = h$). In dit voorbeeld i=1.

Er geldt:

$$delta(start,w) = min\ v: v \in v_{1..n}: 1 + delta(start,v)$$

Van een vertex *v* weten we dat deze zwart, grijs of wit is:

- $STEL: v \in B.$

  Deze situatie kan niet voorkomen. Wanneer een vertex aan *B* wordt toegevoegd worden alle opvolgers namelijk grijs gekleurd. En aangezien *w* wit is kan *v* dus niet zwart zijn.

- $STEL: v \in G$

  In dit geval is de afstand tot *v* bekend volgens P_7. Deze afstand is groter of gelijk aan de afstand tot *h*, aangezien *h* de minimale *depth* waarde heeft.

- $STEL: v \in W$

Als $v \in W$ dan is de afstand tot v, volgens PH_2, ten minste gelijk aan 1 + de afstand tot h, wat logischerwijs groter dan de afstand tot h is.

Hieruit volgt dat voor alle inkomende vertices $v_{1..n}$ geldt dat:

$$\forall v: v \in v_{1..n}: 1 + delta(start,v) \geq 1 + depth(h)$$

Waardoor:

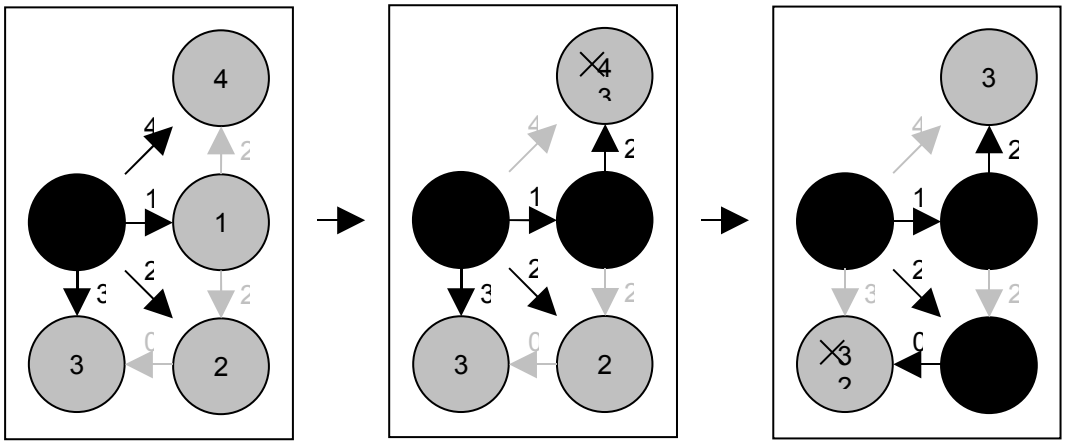
$$delta(start,w) = \min v: v \in v_{1..n}: 1 + delta(start,v) = 1 + depth(h)$$

3.4.6 Weighted Distance

Inleiding

De laatste stap transformeert ons programma naar zijn uiteindelijke vorm: die van Dijkstra's kortste pad algoritme.

We zullen in deze iteratie de gewichten van de edges meenemen in onze berekeningen. Tijdens het opbouwen van onze verbondenheidsboom houden we voor grijze vertices de kortst bekende afstand bij. Wanneer we een vertex toevoegen aan de set van zwarte vertices berekenen we voor elk van zijn buren de afstand via deze vertex. Als de nieuwe afstand korter is dan de al bekende, verlagen we kortst bekende afstand.



Dijkstra's algoritme. De edges die bijdragen aan de laagst bekende afstand zijn donker gekleurd. Merk op dat tweemaal de afstandswaarde van een vertex wordt verlaagd nadat een nieuw, korter, pad is gevonden.

Nieuwe Begrippen

We introduceren een array *distance*. Dit array heeft eenzelfde functie als het *depth* array in vorige iteraties, maar neemt daarbij de gewichten van edges mee. We hebben een nieuwe versie nodig van de volgende invariant:

$$P_7: \forall v: v \in B \cup G: depth(v) = delta(start,v)$$

Laten we de (kortste) afstand tussen twee vertices waarbij wél rekening wordt gehouden met de lengte van de verbindende edges *delta*$^+$ noemen. Voor de zwarte vertices geldt dat de afstand bekend is:

$$P_8: \forall v: v \in B: distance(v) = delta^+(start,v)$$

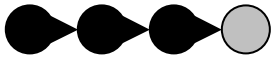
Voor de grijze vertices is de *distance* gelijk aan de kortst bekende afstand (het minimum van de afstanden van de zwarte voorgangers plus verbindende edges):

P_9: $\forall v: v \in G \setminus start:$
  $distance(v)$
  $=$
  $\min x: x \in predecessors(v) \cap B: delta^+(start,x) + weight(x,v)$

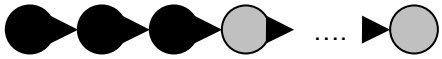
Het in stand houden van deze tweede nieuwe invariant (P_9) is eenvoudig: wanneer de set *B* uitgebreid wordt, kunnen we de *distance* waarde van alle grijze opvolgers berekenen en eventueel verlagen.

Het bewijs van P_8 steunt op het volgende punt: voor de grijze vertex *h*, met minimale *distance* waarde, geldt dat de *distance* gelijk is aan de *delta*$^+$.

Het bewijs hiervoor, in woorden, gaat als volgt: de *distance* van *h* is de kortst bekende afstand via een zwarte voorganger. Aangezien we de afstand tot deze zwarte vertices kennen (volgens P_8), mogen we *h* zwart kleuren mits er een kortste pad naar *h* is via een zwarte (directe) voorganger:



We bewijzen dat een kortste pad naar *h* met een grijze of witte voorganger nooit korter kan zijn dan een kortste pad met een zwarte voorganger. Zo'n pad heeft namelijk altijd de volgende vorm:



Om een kortere lengte op te leveren zal de afstand tot aan de eerste grijze vertex van dit pad ook kleiner moeten zijn, terwijl de door ons gekozen grijze vertex *h* een minimale *distance* waarde heeft.

Formeel

```
var
      B, G, W: set of Vertex,
      depth: function of Vertex → nat,
      distance: function of Vertex → nat
      pred: function of Vertex → Vertex,
      out path: list of Vertex,
      out shortestdistance: int;

B := ∅;
G := {start};
W := V \ {start};
depth(start) := 0;
distance(start) := 0;

do G ≠ ∅ →
      let h: Vertex satisfy h ∈ G ∧ "distance(h) is minimal";

      B, G := B ∪ {h}, G \ {h};

      do forall x ∈ successors(h) →

            var len: nat = distance(x) + weight(h,x);

            if
                  x ∈ W → W,G := W \ {x}, G ∪ {x},
                           distance(x) := len,
                           depth(x) = depth(h) + 1,
                           pred(x) := h;
            []    x ∈ G →
                  if
                        distance(x) > len → distance(x) := len;
                                            depth(x) := depth(h)+1
                                            pred(x) := h
                  []    distance(x) ≤ len → skip
                  fi
            []    x ∈ B → skip
            fi
      od
od

if
      target ∈ B → shortestdistance:= distance(target);
                   path :=
                   "het pad gegeven door pred^depth(target)(target)"
[]    target ∉ B → shortestdistance:= ∞;
                   path := <>
fi
```

De nieuwe invarianten:

P_8: *∀v: v ∈ B: distance(v) = delta⁺(start,v)*

P_9: *∀v: v ∈ G\start:*
distance(v)
=
min x: x ∈ predecessors(v) ∩ B: delta⁺(start,x) + weight(x,v)

Zoals eerder gezegd is P_9 eenvoudig te bewijzen en ligt de crux bij P_8. Deze invariant komt ter sprake bij het volgende deel van de code:

```
let h: Vertex satisfy h ∈ G ∧ "depth(h) is minimal";

{ P8 }
B := B ∪ {h};
{ P8 }
```

Te bewijzen:

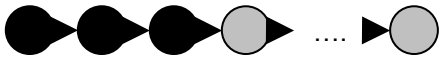
$$distance(h) = delta(start,h)$$

We bewijzen dit uit het ongerijmde. Stel de stelling is onwaar. Dan:

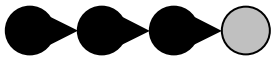
$$\exists p: p \in paths(start,h): length(p) > distance(h)$$

Waarbij we *paths* definiëren als de set van alle mogelijke paden tussen *start* en *h* en *length(p)* als de som van de gewichten van de edges in *p*.

Een pad vanuit *start* naar *h* zal altijd beginnen met één of meer zwarte vertices (in het geval *start* zwart is, voor het speciale geval waarin *G* bestaat uit enkel *start* en *B* leeg is geldt de invariant triviaal). Uit invariant P_4 volgt dat alle buren van *B* grijs zijn, dus zullen de zwarte vertices gevolgd worden door ten minste één grijze vertex:



Ook ons kleiner pad *p* heeft dus deze vorm. Als we het achterste deel van pad *p* weglaten, wordt de lengte ervan nog korter:



We noemen de laatste twee vertices van dit pad *a* en *b*. Er geldt:

$$delta^+(a) + weight(a,b) \leq length(p) \land$$
$$length(p) < distance(h) \land$$
$$\Rightarrow$$
$$delta^+(a) + weight(a,b) < distance(h)$$

Van *h* weten we, volgens P_9:

$$distance(h)$$
$$=$$
$$min\ x: x \in predecessors(h) \cap B: delta^+(start,x) + weight(x,h)$$

We weten ook dat *h* die vertex is waarvoor deze waarde minimaal is, dus:

$$h = min\ h': h' \in G: min\ x: x \in predecessors(h') \cap B: delta^+(start,x) + weight(x,h')$$

Gecombineerd:

$$delta^+(a) + weight(a,b)$$
$$<$$
$$min\ h': h' \in G: min\ x: x \in predecessors(h') \cap B: delta^+(start,x) + weight(x,h')$$

Aangezien $a \in predecessors(b)$ en $a \in B$ en $b \in G$:

$$\exists a,b: a \in predecessors(b) \land a \in B \land b \in G:$$
$$delta^+(a) + weight(a,b)$$
$$<$$
$$min\ a,b: a \in predecessors(b) \land a \in B \land b \in G:$$
$$delta^+(start,a) + weight(a,b)$$

Dit is een tegenspraak! Hierdoor geldt onze originele stelling:

$$distance(h) = delta^{+}(start,h)$$

Met deze invariant P_8 kunnen we rechtstreeks de uiteindelijke postconditie opstellen:

$$Q_4: \forall v: v \in B: distance(v) = delta^{+}(start,v)$$

En als we deze combineren met onze eerdere postconditie Q_0, die zegt dat de set B de bereikbare set van vertices vanuit *start* is:

$$Q_4 \wedge Q_0$$
$$\Rightarrow$$
$$\forall v: v \in V:$$
$$\quad \textbf{if } [v \in B] \rightarrow delta^{+}(start,v) = distance(v)$$
$$\quad\quad [v \notin B] \rightarrow delta^{+}(start,v) = \infty$$
$$\quad \textbf{fi}$$

3.5 Discussie

Zowel Cocktail als Perfect Developer blijken, toegepast op Dijkstra's kortste pad algoritme, bijzonder arbeidsintensief te zijn. Cocktail dwingt ons om elke stap met de interactieve prover te bewijzen, inclusief benodigde stellingen over types als arrays, sets, booleans, etc. Dit leidt tot veel en lange bewijzen die de kern van het algoritme vertroebelen.

Perfect Developer, aan de ander kant, tracht alles automatisch te bewijzen. Het slaagt hierin voortreffelijk voor de eenvoudigere bewijsverplichtingen, maar heeft beduidend meer moeite met de subtielere delen van het algoritme. Wanneer dit het geval is, is het aan de programmeur om door middel van trial and error te zoeken naar een alternatieve oplossing of extra stellingen die de prover helpen het bewijs rond te krijgen. Door een beperkte doorzichtigheid van de onderliggende prover leidt dit vaak tot giswerk.

We zijn erin geslaagd een bewijs voor het algoritme te verkrijgen met behulp van Cocktail, maar de correctheid van dit bewijs is, door enige kinderziektes van Cocktail, twijfelachtig (zie ook de conclusies in hoofdstuk 4 en de appendix). Onze pogingen om het algoritme te bewijzen met Perfect Developer hebben we bij zowel Stepwise Refinement als Incremental Development moeten staken (zie appendix).

Cocktail's werkwijze is gebaseerd of Stepwise Refinement. Incremental Development wordt niet expliciet ondersteund. Het volgen van de Stepwise Refinement stappen blijkt echter moeizaam. Dit is vooral een gevolg van de beknopte mogelijkheden van de editor. De interface is spaarzaam, het is bijvoorbeeld onmogelijk om bewijzen gedeeltelijk te wijzigen, subbewijzen uit te splitsen, bewijzen te versterken of verzwakken (wanneer dit logisch gezien sound is), of (in het geval van bewijzen die niet expliciet als theorema zijn vastgelegd maar als deel van het programma zelf zijn gemaakt) te bekijken. Dit leidt ertoe dat het volgen van de Stepwise Refinement stappen uit de GCL versie in veel gevallen tot herschrijven van al bewezen stellingen leidt. In de praktijk hebben we het kortste pad algoritme dan ook in één keer in zijn geheel bewezen en zijn de afleidingsstappen niet gevolgd.

Perfect Developer heeft dit probleem niet, aangezien de bewijzen in hun geheel gesloten zijn voor de gebruiker. In de code zelf is het eenvoudig om gaten open te laten (doordat specificaties met programmatekst kunnen worden gemengd), pre- en postcondities bij te werken, functies te splitsen, etc. Constructies voor object oriented programming zorgen daarnaast voor mogelijkheden tot Incremental Development. Een probleem bij deze keur aan mogelijkheden is echter dat ze geen tegenhanger in de prover hebben: het invullen van een specificatie houdt bewijzen voor omliggende code intact, maar de prover zal het gehele algoritme opnieuw proberen te bewijzen, waarbij het soms voor kan komen dat voorheen eenvoudig bewijsbare asserties door ongerelateerde wijzigingen meer tijd vereisen of zelfs onbewijsbaar worden.

Het opstellen en bewijzen van losse objecten, compleet met eigen modelspecificatie en verborgen implementatie, kan het schalingsprobleem enigszins oplossen. De integratie van

objecten verloopt echter niet altijd even vlekkeloos (zie de verhandeling over de natInf klasse in de appendix).

4 Conclusies

We vatten nu de tijdens het onderzoek gevonden antwoorden op onze initiële vragen samen, gebruikmakend van de resultaten besproken in 3.5 en 2.3.

4.1 Vergelijking met het ideaal (2.3)

Als we de behandelde programmeertalen bekijken kunnen we ze ruwweg in twee groepen verdelen: tools die proberen een algoritme in zijn geheel te verifiëren en tools die een "praktische" verificatie uitvoeren. De eerste groep, bestaande uit Cocktail en Perfect Developer, volgen het nauwst de papieren bewijzen. Al de modelbewijzen[1] worden door deze methodes opgesteld en volledig bewezen.

ESC en Eiffel, echter, helpen bij het zoeken naar fouten, maar geven geen garantie dat een programma geheel foutloos is: er wordt een onvolledige set aan bewijsverplichtingen gegenereerd.

Wat betreft de ondersteuning van Stepwise Refinement en Incremental Development hebben we dit onderscheid alleen bekeken bij Perfect Developer. Cocktail is geheel gebaseerd op Stepwise Refinement en voor de overige twee talen hebben we enkel eenvoudige éénstaps-algoritmes bekeken.

Perfect Developer ondersteunt beide methoden, maar met wat kanttekeningen. Wat betreft Stepwise Refinement kunnen er een model en concrete implementatie worden gedefinieerd, maar slechts via één stap. De taal staat daarnaast het gebruik van (ongeïmplementeerde) specificaties in de code toe, waardoor we de typische "gaten" kunnen noteren en compileren. De OO-functies staan weer een Incremental Development aanpak toe, hoewel deze minder geschikt is voor het uitbreiden van code die zich in één object, of in ons geval zelfs procedure, bevindt.

Een nadeel van Perfect Developer's ondersteuning voor Structured Programming is dat programma's niet incrementeel worden bewezen. Wanneer we een gat invullen in een correct programma worden oude bewijzen niet meegenomen tijdens het compileren. Wanneer als onderdeel van de nieuwe code veel nieuwe proposities en variabelen worden geïntroduceerd kunnen deze de oude bewijzen zelfs moeilijker maken, ook al hebben ze er geen invloed op.

4.2 Praktische bruikbaarheid (2.3 en 3.5)

Zowel PD als Cocktail vereisen een volledige specificatie van modules of methoden. Dit houdt in dat de programmeur een grote kennis van het algoritme en de bewijsstructuur dient te hebben. Ter voorbeeld: gesorteerdheid van een array is eenvoudig via een functie te testen en als postconditie te plaatsen in Eiffel, terwijl de soort definitie die gekozen wordt een grote invloed heeft op de afleiding in Cocktail en Perfect Developer (bepaalde formuleringen van de postconditie passen beter bij sommige algoritmes dan andere).

De strikte talen voegen een extra laag aan complexiteit aan het schrijven van een algoritme toe. Zelfs als de specificatie op een intuïtief niveau duidelijk is, is het noteren ervan op een manier waarmee een compiler kan rekenen vaak een kunst op zich. Esc en Eiffel vragen minder van de programmeur en zijn daardoor eenvoudiger in het gebruik.

Zowel Cocktail als Perfect Developer bevatten kinderziektes. Voor professioneel gebruik in een grootschalig project dienen zaken als de user interface, syntax, compiler en IDE te worden verbeterd. Vooral Cocktail lijdt hieronder, maar ook Perfect Developer's cryptische foutmeldingen en IDE doen onder voor ontwikkelomgevingen als Visual Studio of Eclipse. Daarnaast gebruikt Cocktail het gelimiteerde, procedurele GCL en bevat het geen compiler naar machinecode.

Ook lijden beide toolkits aan nadelen door hun gekozen manier van werken:

[1] Met als uitzondering dat Cocktail geen gedefinieerdheid of terminatie bewijst.

Het maken van een bewijs in Cocktail is minstens zo ingewikkeld als het (informeel) bewijzen van een programma op papier en zo'n informeel papieren bewijs wordt in de praktijk zelden voor een geheel programma opgesteld.

Het opstellen van bewijzen in Perfect Developer is in eerste instantie eenvoudiger. Totdat de bewijzer tegen problemen aanloopt. Soms ligt dit aan kleine zaken als het missen van axioma's, maar vaak zijn de stellingen eenvoudigweg te ingewikkeld voor (huidige) automatische bewijzers. Een groot probleem hierbij is dat het voor de gebruiker onmogelijk is om het onderscheid te maken tussen een stelling die niet op te lossen is vanwege de gebrekkige bewijzer, een stelling die niet op te lossen is vanwege foutieve code en een stelling die niet te bewijzen is vanwege foutieve annotatie.

Aan de andere kant hoeven we ons niet op 100% verificatie te richten - een programma in PD compileert ongeacht het aantal onbewijsbare stellingen. In die zin lijkt PD op een striktere versie van ESC. Specificatie is in PD echter niet optioneel zoals dat bij ESC wel is.

Eiffel, ten slotte heeft de minst betrouwbare output. Fout gedrag dat alleen bij randgevallen voorkomt zal alleen worden ontdekt als die randgevallen daadwerkelijk opduiken. Aan de andere kant is Eiffel de eenvoudigste taal om mee te werken: er hoeft geen aparte specificatietaal geleerd te worden omdat alle pre- en postasserties in termen van standaard programmafuncties worden gegeven. Dit houdt wel weer in dat sommige specificaties moeilijk (of niet) in te voeren zijn. Daarnaast kunnen runtime checks de efficiëntie in sommige gevallen zo zwaar benadelen dat ze praktisch onuitvoerbaar worden.

Deze vergelijking lijkt een schaal van betrouwbaarheid aan de ene kant en gebruiksgemak aan de andere kant voor te stellen. De betrouwbare talen zijn het meest geschikt voor korte kernberekeningen die een hoge mate van correctheid vereisen terwijl de talen aan het andere eind van het spectrum geschikter zijn voor minder kritieke code.

Onze ervaring met Cocktail en Perfect Developer is dat het gebruik van de methodes bij een middelgroot algoritme (Dijkstra's kortste pad) al overmatig complex wordt. Het is ons niet gelukt om het algoritme te bewijzen met Perfect Developer. Met Cocktail zijn we hier wel in geslaagd, met als kanttekening dat we, doordat we een bètaversie van het programma hebben gebruikt, tegen enkele problemen zijn opgelopen die ons niet 100% zeker maken van de correctheid van ons bewijs.

4.3 Mogelijke verbeteringen

Cocktail

We concentreren ons hier vooral op de twee methoden die we aan intensiever onderzoek hebben onderworpen middels Dijkstra's kortste pad algoritme.

Bij Cocktail valt ten eerste veel te verbeteren aan het gebruiksgemak. Er volgt hier een greep aan problemen en waar toepasselijk mogelijke oplossingen:

- Het is onmogelijk om een gedeeltelijk bewezen stelling op te slaan en hier later mee verder te werken.
- Stellingen die als deel van het programma zelf worden bewezen kunnen niet worden opgeslagen voor hergebruik, of zelfs herziening.
- Gegenereerde bewijzen zijn moeilijk na te lezen. Opties voor verbeteringen zijn de mogelijkheid om een bewijs (automatisch) op te schonen door elke stap die niet gebruikt wordt in een latere stap te verwijderen, het groeperen van stappen in een bewijs en die als groep te minimaliseren/uitvouwen, het markeren van een set "kernstappen" die de grote lijn van het bewijs uiteenzetten, het toevoegen van gebruikerscommentaar.
- Het is onmogelijk om axioma's te vervangen door (bewezen) stellingen. Hetzelfde geldt voor het vervangen van stellingen door dezelfde stelling met een nieuw (beter, eleganter) bewijs. Ook ín stellingen zelf zouden we onderdelen willen vervangen.

- Op eenzelfde wijze zou het gemakkelijk zijn om substellingen uit een grote stelling te halen en apart op te slaan.
- Bewijzen kunnen niet tijdens het bewijzen (of erna) worden versterkt of verzwakt. Een voorbeeld hiervan is een bewijs voor $A \Rightarrow B$, waar als neveneffect het bewijs voor $A \Rightarrow C$ wordt gegenereerd. De stelling $A \Rightarrow B \wedge C$ zou hier eenvoudig uit te extraheren kunnen zijn. Andersom kan een bewijs voor $A \Rightarrow C$ vastlopen door een te zwakke premisse en na een lang bewijs steken op $A \Rightarrow (B \Rightarrow C)$. De stelling $A \wedge B \Rightarrow C$ zou hieruit te extraheren kunnen zijn.
- Het groeperen van types en stellingen in herbruikbare modules zou het mogelijk maken om een sterke en uitbreidbare basis aan standaardstellingen en types te definiëren.

Bij Cocktail vormt de beperking tot eerste orde logica een probleem. We zouden hier graag wat meer toevoegingen op zien. Voorbeelden van zaken die moeizaam gaan zijn:

- Inductie. Er bestaat een inductieregel, maar deze is alleen gedefinieerd op natuurlijke getallen. Als we inductie op een set willen toepassen dan moeten we die eerst afbeelden op natuurlijke getallen.
- Speciale quantoren. Het definiëren van een minimumquantor is onmogelijk, waardoor we minimum als recursieve functie moeten definiëren, wat bewijzen nodeloos ingewikkeld maakt.
- Tactieken. We zijn gebonden aan de voorgedefinieerde tactieken voor het herschrijven van stellingen. Het zou handig zijn om zelf tactieken toe te voegen, bijvoorbeeld door een aantal van de standaard tactieken als macro of script te bundelen.

Oplossingen voor deze problemen zijn minder eenvoudig te vinden. Een wellicht niet al te elegante maar wel praktische suggestie zou zijn om speciale constructies in de toolkit aan te brengen voor elk van deze gewenste eigenschappen. We kunnen ons bijvoorbeeld voorstellen dat we via een menu optie een type kunnen afbeelden op de natuurlijke getallen en dat er dan via die afbeelding automatisch een inductietactiek voor dat type wordt gegenereerd.

Perfect Developer

Ook hier zouden we graag een sterk verbeterde gebruikersinterface zien. Zaken als autocomplete, uitgebreide compilermeldingen en waarschuwingen bij syntaxfouten, ingebouwde helpfunctie en automatische code formattering mogen in een huidige programmeertaal eigenlijk niet meer ontbreken.

Wat het formele gedeelte betreft is de ondoorzichtigheid een groot probleem. Het is moeilijk om de “juiste” implementatie van een algoritme te vinden. Wachttijden en onduidelijke compileerfouten maken variëren tussen oplossingen een kostbaar proces.

We zouden graag, ondanks de doelstelling van volledige automatische bewijzen, de bewijzer meer openstellen. Er bestaan manieren om de bewijzer te "sturen", maar deze zijn te beperkt. Features die we missen zijn:

- Het expliciet sturen van de bewijzer. Bijvoorbeeld door de stappen van een bewijs uit te schrijven en voor elke stap hints mee te geven van welke stellingen/axioma's gebruikt kunnen worden.
- Het bekijken en toevoegen van axioma's. Tijdens ons onderzoek bleek een axioma over sets te missen - we zouden deze graag handmatig toevoegen aan het gehele project (er bestaat een axioma functie maar deze is gebonden aan een klasse).
- De mogelijkheid om bewezen stellingen op te slaan (vergelijkbaar met Cocktail's context). Dit zorgt ervoor dat eenmaal bewezen stellingen niet telkens opnieuw bewezen hoeven te worden en, in combinatie met onze eerste suggestie, expliciet in vervolgstellingen kunnen worden gebruikt.

Vergelijking

Op een manier vormen Perfect Developer en Cocktail elkaars complement. Cocktail biedt door zijn interactieve stellingbewijzer de mogelijkheid om de meest ingewikkelde stellingen correct te bewijzen waarbij de editor (met enige verbeteringen op gebruiksgemak) de gebruiker in staat stelt om te experimenteren op zoek naar de oplossing en tegelijkertijd ervan verzekerd te zijn een correct bewijs te hebben wanneer het doel eenmaal bereikt is.

Perfect Developer biedt een arsenaal van types en stellingen en kan veel bewijzen snel en automatisch genereren – evenals een (in onze ervaring) complete set van bewijsverplichtingen bij een programma. Bij beide tools wensten we zo nu en dan dat we de mogelijkheid hadden om eventjes op de ander over te stappen. Wellicht zou een vereniging van de twee methodes, een compiler die een programma voor 90% bewijst en dan de gebruiker de mogelijkheid biedt om de moeilijke delen (met hulp) handmatig op te lossen, de sterke punten van de tools benadrukken en de zwakheden verhullen.

4.4 De verifying compiler

Is de verifying compiler een haalbaar project? We hebben een aantal toolkits bekeken die een subset van de bijbehorende problemen proberen op te lossen. Geen van de bestudeerde toolkits voldoet aan het ideaal en degene die hier het meest naar streeft (Perfect Developer) blijkt in de praktijk het moeilijkst te hanteren.

We zijn er nog niet. De automatische bewijzers zijn nog amper in staat om non-triviale bewijzen te leveren. Of automatische verificatie zulke vooruitgang gaat boeken dat dit geen probleem meer is zal de toekomst moeten uitwijzen. Haalbaarder lijkt op dit moment om een intuïtieve interface te bieden voor het handmatig bewijzen van de non-triviale onderdelen van een algoritme en de talrijke eenvoudigere bewijzen te automatiseren. Dat beide onderdelen hiervan afzonderlijk zijn te implementeren hebben we gezien, het combineren ervan zal ongetwijfeld een keur aan eigen problemen met zich meebrengen.

Daarnaast is een fundamenteel probleem, waar geen van de besproken toolkits een oplossing voor biedt, het feit dat een programma om bewezen te worden een specificatie nodig heeft. Vooral de striktere toolkits Cocktail en Perfect Developer vereisen een uitgebreide specificatie van pre- en postcondities, zowel als loopinvarianten en variante functies. Om deze op te stellen is over het algemeen een gedetailleerde kennis van het bewijs nodig; naïeve specificaties leiden tot uiterst gecompliceerde bewijzen. Een aspect waaraan in dit onderzoek (en de onderzochte toolkits) geen aandacht is geschonken is dan ook het automatisch extraheren van bruikbare specificaties uit programma's: een probleem dat opgelost dient te worden om het ideaal te bereiken.

Dat gezegd hebbende zijn de toolkits in staat tot indrukwekkende resultaten. Het aantal bewijsverplichtingen dat door Perfect Developer wél moeiteloos wordt geëxtraheerd en bewezen is groot. Het moet zeker mogelijk zijn om, in de trant van ESC, uitgebreide statische analyse toe te voegen aan hedendaagse compilers om ons voor veel voorkomende fouten te behoeden. Gespecialiseerde syntax, zoals we zien bij Eiffel en Perfect Developer, kan dit vereenvoudigen.

Referenties

[1] J. Carlton, D. Crocker, Perfect Developer: what it is and what it does, FACS Facts (newsletter of the BCS Formal Aspects of Computer Science special interest group), November 2004

[2] Y. Cheon, G.T. Leavens, Design by Contract with JML, http://www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf, 2006

[3] T.H. Cormen et. al., Introduction to Algorithms, second edition, MIT Press and McGraw-Hill, 2001

[4] D. Crocker, Safe Object-Oriented Software: the Verified Design-by-Contract paradigm, Proceedings of the Twelfth Safety-Critical Systems Symposium (ed. F.Redmill & T.Anderson), pp. 19-41, Springer-Verlag, London, 2004

[5] O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured Programming, Academic Press, London, 1972

[6] E.W. Dijkstra, A Discipline of Programming, Pentice Hall, Englewood Cliffs, N.J., 1976

[7] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik, No. 1, pp. 269–271, 1959

[8] E.W. Dijkstra, Guarded commands, non-determinacy and formal derivation of programs CACM, Vol. 18, No. 8, 1975, pp. 453–457

[9] E.W. Dijkstra, Program Development by Stepwise Refinement, CACM, Vol. 14, No. 4, 1971, pp. 221 - 227

[10] Escher Tech, List of verification condition types (PDF), http://www.eschertech.com/product_documentation/Verification%20Conditions%20Generated%20by%20Perfect%20Developer.pdf

[11] R.W. Floyd, Assigning Meaning to Programs, Proceedings of Symposium on Applied Mathematics, Vol. 19, J.T. Schwartz (Ed.), A.M.S., 1967, pp. 19-32

[12] M. Franssen, Cocktail: A Tool for Deriving Correct Programs, PhD. Thesis, Eindhoven University of Technology 2000

[13] C.A.R. Hoare, An axiomatic basis for computer programming, CACM, Vol. 12, No. 10, 1969, pp. 576-580.

[14] C.A.R. Hoare, The Verifying Compiler: A Grand Challenge for Computing Research, JACM, Vol. 50, No. 1, 2003, pp. 63-69

[15] A. Kaldewaij, Programmeren deel 3: Datastructuren en standaardalgoritmen, Bohn Stafleu van Loghum, 1993

[16] A. Kaldewaij, Programming: The Derivation of Algorithms, Prentice-Hall International Series In Computer Science, 1990

[17] K.R.M. Leino, G. Nelson, J.B. Saxe, ESC/Java User's Manual, SRC Technical Note 2000-002, October 12, 2000

[18] K.R.M. Leino et al. Extended static checking, http://research.compaq.com/SRC/esc/Esc.html, 2000.

[19] B. Meyer, Applying "Design by Contract", Computer (IEEE), Vol. 25, No. 10, 1992, pp. 40-51

[20] B. Meyer, Eiffel: The Language, Prentice Hall Object-Oriented Series, Paperback - 1991

[21] B. Meyer, Object-Oriented Software Construction, second edition, Prentice Hall, 1997

[22] J.S Moore & Q. Zhang, Theorem Proving in Higher Order Logics, Springer Berlin / Heidelberg, 2005

[23] J.C. Reynolds, The Craft of Programming, Prentice-Hall, 1981

Appendix

A Kleine Case Studies

A.1 Linear search

A.1.1 Eiffel

Het programma

```
0. linsearch(a: ARRAY [BOOLEAN]): BOOLEAN is
1.             local
2.                 r: BOOLEAN
3.                 x: INTEGER
4.          do
5.              r := FALSE
6.
7.              from
8.                  x := 0
9.              invariant
10.                 P0a: 0 <= x
11.                 P0b: x <= a.count
12.                 -- P1: r = there exists an i in a[0..x-1] with
                               a[i]=true
13.             variant
14.                 a.count - x
15.             until
16.                 x = a.count
17.             loop
18.                 r := r or a @ x
19.                 x := x+1
20.             end
21.
22.             result := r
23.
24.           ensure
25.               -- Q: result = there exists an i in a with
                             a[i]=true
26.         end
```

Merk op dat tekst voorafgegaan door "--" commentaar behelst en dat deze dus niet door Eiffel wordt gebruikt (regels 12 en 25).

Bewijsverplichtingen

De volgende bewijsverplichtingen worden door Eiffel (runtime) gecontroleerd:

Initialisatie, invariantie, finalisatie en terminatie (met als klein verschil dat in plaats van een constante *C* het getal *0* als ondergrens voor de VF wordt gebruikt). Dit zijn alle bewijsverplichtingen uit onze papieren versie.

Hier moeten wel wat kanttekeningen bij worden geplaatst:

Ten eerste werkt Eiffel met runtime asserties waardoor de condities niet volledig worden getest.

Ten tweede wordt gedefinieerdheid in die zin getest dat het programma aborteert wanneer een waarde niet is gedefinieerd, maar dit is nu juist het gedrag dat we willen voorkomen met dit bewijs.

Ten slotte is de assertietaal beperkt tot boolean expressies die berekenbaar zijn in de programmeertaal. Dit zorgt ervoor dat in dit geval de invariant P_1 en Q niet naar Eiffel termen kunnen worden vertaald, aangezien deze quantoren gebruiken.

Op dit laatste punt valt echter wel iets af te dingen, wat leidt tot de volgende wijziging.

Quantoren

Ook al kent Eiffel niet de benodigde quantoren, het is toch mogelijk om deze in beperkte mate zelf te definiëren. Het is namelijk mogelijk om functies te gebruiken in asserties, waardoor we de eenvoudige (eindige) quantoren van de linear search als functies kunnen implementeren.

We kunnen dit gebruik van functies in asserties ook voor andere doeleinden gebruiken. Stel dat we een gegeven implementatie van de (linear) search hebben, waarvan we mogen aannemen dat deze correct is. Dan kunnen we deze gebruiken als postconditie van onze eigen linear search, als volgt (let op regel 12 en 25):

```
1. linsearch(a: ARRAY [BOOLEAN]): BOOLEAN is
2.             local
3.                 r: BOOLEAN
4.                 x: INTEGER
5.         do
6.             r := FALSE
7.
8.             from
9.                 x := 0
10.            invariant
11.                P0a: 0 <= x
12.                P0b: x <= a.count
13.                P1: r = correct_search(a.subarray(0,x-1))
14.            variant
15.                a.count - x
16.            until
17.                x = a.count
18.            loop
19.                r := r or a @ x
20.                x := x+1
21.            end
22.
23.            result := r
24.
25.          ensure
26.              Q: result = correct_search(a)
27.        end
```

Uiteraard wordt het probleem enkel verplaatst naar de nieuwe functie "correct_search", maar het is bij ingewikkelde algoritmes vaak mogelijk om een eenvoudiger algoritme te gebruiken als controlefunctie. Zo kan een linear search gebruikt worden om de uitvoer van een binary search to controleren. Dit gaat wel ten koste van de efficiëntie van het algoritme, aangezien we nu zowel een binary search als meerdere linear searches uitvoeren.

In dit specifieke geval bestaat er een interne functie die een universele quantor implementeert over een (bounded) array. We kunnen deze gebruiken om onze existentiële quantor te implementeren.

We beginnen met een hulpfunctie om de universele quantor in een existentiële om te vormen:

```
isfalse(b: BOOLEAN): BOOLEAN is
            do
                  result := not b;
            end
```

En gebruiken deze als volgt:

```
0. linsearch(a: ARRAY [BOOLEAN]): BOOLEAN is
1.             local
2.                 r: BOOLEAN
3.                 x: INTEGER
4.          do
5.              r := FALSE
6.
7.              from
8.                  x := 0
9.              invariant
10.                 P0a: 0 <= x
11.                 P0b: x <= a.count
12.                 P1: r = not a.subarray(0,x-1).for_all
                            (agent isfalse)
13.             variant
14.                 a.count - x
15.             until
16.                 x = a.count
17.             loop
18.                 r := r or a @ x
19.                 x := x+1
20.             end
21.
22.             result := r
23.
24.           ensure
25.               Q: result = not a.for_all (agent isfalse)
26.         end
```

A.1.2 ESC/Java

Het programma

```
1.  public abstract class LinearSearch
2.  {
3.      //@ ensures 0 <= \result;
4.      public abstract /*@ pure */ int N();
5.
6.      //@ requires j >= 0;
7.      //@ requires j < N();
8.      public abstract /*@ pure */boolean B(int j);
9.
10.     //@ ensures \result == (\exists int i; 0 <= i && i < N(); B(i));
11.     public boolean linearSearch()
12.     {
13.         int x = 0;
14.         boolean r = false;
15.
16.         //@ maintaining 0 <= x;
17.         //@ maintaining x <= N();
18.         //@ maintaining r == (\exists int i; 0 <= i && i < x; B(i));
19.         //@ decreasing N() - x;
20.         while (x != N())
21.         {
22.             r = r || B(x);
23.             x = x + 1;
24.         }
25.         return r;
26.     }
27. }
```

De invarianten en annotatie van onze papieren versie zijn direct terug te vinden in het bovenstaande programma.

Verificatie

Dit programma genereert geen waarschuwingen bij verificatie. Dat dit niet meteen betekent dat het programma in orde is, is al eerder besproken. Toch geven we bij wijze van illustratie een voorbeeld van een foutief programma dat wordt goedgekeurd.

Zoals gezegd bewijst ESC loops niet volledig. Het gebruikt een systeem waarbij de loop enkele malen wordt ontvouwen en waarbij de invarianten alleen op het resultaat van deze ontvouwing worden gecheckt. We kunnen hier op de volgende manier "misbruik van maken":

```
1.  public abstract class LinearSearch
2.  {
3.      //@ ensures 0 <= \result;
4.      public abstract /*@ pure */ int N();
5.
6.      //@ requires j >= 0;
7.      //@ requires j < N();
8.      public abstract /*@ pure */boolean B(int j);
9.
10.     //@ ensures \result == (\exists int i; 0 <= i && i < N(); B(i));
11.     public boolean linearSearch()
12.     {
13.         int x = 0;
14.         boolean r = false;
15.
16.         //@ maintaining 0 <= x;
17.         //@ maintaining x <= N();
18.         //@ maintaining r == (\exists int i; 0 <= i && i < x; B(i));
19.         //@ decreasing N() - x;
20.         while (x != N())
21.         {
22.             r = r || B(x);
23.             x = x + 1;
24.
25.             if (x==5) { r = true; }
26.         }
27.         return r;
28.     }
29. }
```

Let op de toevoeging op regel 24. Aangezien ESC minder dan 5 uitvouwingen uitvoert zal de guard van de if-statement altijd op false evalueren. Hierdoor wordt ook voor dit programma geen waarschuwing gegeven. Dat het programma echter incorrect is, moge duidelijk zijn.

A.1.3 Cocktail

De gebruiker mag in Cocktail-bewijzen gebruik maken van axioma's, types, proposities, stellingen en tactieken.

Axioma's, types, proposities en stellingen worden in de theoremprover opgesteld (en in het geval van stellingen bewezen. Het bewijs hierboven, voor de stelling:

```
ThPlus1: ∀y:nat.plus(0,y)=y
```

Maakt gebruik van de axioma's:

```
AxPlus1: ∀x:nat.plus(x,0)=x
AxPlus2: ∀x,y:nat.plus(s(x),y)=s(plus(x,y))
```

Verder is *nat* een type. Dit type is standaard beschikbaar in Cocktail en representeert de natuurlijke getallen. Er zijn twee functies die een natuurlijk getal opleveren:

```
0: nat
s(n: nat): nat
```

Waarbij *s* de successorfunctie is. Elk natuurlijk getal is intern "de successorfunctie een aantal maal toegepast op *0*". Deze waarden worden wel weer herschreven naar de voor mensen leesbare vorm.

De gebruiker kan zelf nieuwe types definiëren. Bijvoorbeeld de booleans:

```
bool: *s
true(): bool
false(): bool

true ≠ false
∀b:bool.(b=true ∨ b=false)
```

Merk op dat deze booleans een door de gebruiker zelf gedefinieerd type zijn en geen relatie hebben tot interne booleans. De volgende stelling is bijvoorbeeld onzinnig:

```
ThBool: ∀a,b:bool.(a ⇒ b ≡ ¬b ∨ a)
```

De implicatie, negatie of disjunctie operator is niet gedefinieerd op het eigen gedefinieerde type bool!

We kunnen dit soort stellingen wel maken met behulp van proposities. Een propositie mapt een aantal invoerparameters op True of False (de interne booleans). We kunnen via proposities onze bool mappen op True of False:

```
isTrue(): *p

AxBool2: isTrue(true)
AxBool3: ¬isTrue(false)
```

Op *istrue(b)* zijn de operatoren wel standaard gedefinieerd en we kunnen onze stelling vertalen als:

```
ThBool: ∀a,b:bool.
     (isTrue(a) ⇒ isTrue(b) ≡ ¬isTrue(b) ∨ isTrue(a))
```

Tactieken zijn mogelijke stappen die in een vlaggenbewijs kunnen worden genomen, zoals inductie (bij natuurlijke getallen), ⇒-introductie of eliminatie, applicatie of herhaling van een axioma/stelling, etc.

Voor het bewijzen van het algoritme in Cocktail moet allereerst een inventarisatie worden gemaakt van de gebruikte types, datastructuren en operaties, omdat deze in Cocktail zullen moeten worden gedefinieerd. Daarna kan het programma worden ingevoerd en de benodigde stellingen worden bewezen.

Definities

booleans:

```
bool: *s
true(): bool
false(): bool

AxBool1: ∀b: bool.b = true ∨ b = false
AxBool2: true ≠ false

or(a,b:bool): bool

AxOr1: ∀b: bool.b OR true = true
AxOr2: ∀b: bool.b OR false = b
```

integers:

Het nat type is standaard beschikbaar in Cocktail. Aangezien de gebruikte integers in het algoritme nooit negatief zijn voldoet dit type en laten we een extra definitie voor negatieve waardes achterwege. De definitie voor + is ook niet noodzakelijk aangezien deze in het programma enkel wordt gebruikt om een integer met 1 op te hogen, wat hetzelfde betekent als de opvolger van die integer (de successorfunctie *s* in Cocktail).

```
smaller(a,b: nat): bool

AxSmaller1: ∀i:nat.0 < s(i) = true
AxSmaller2: ∀i:nat.i < 0 = false
AxSmaller2: ∀i,j:nat.s(i) < s(j) = i < j
```

constanten:

```
B(i:nat): bool
N: nat
```

B is een functie die voor het programma constant is. Merk op dat we *B* zonder bound definiëren. Gedefinieerdheid (range checking) komt slechts impliciet ter sprake wanneer de definitie van de functie gebruikt wordt in de bewijzen.

Het programma

```
var r: bool
|[ var x: nat
      { True }
      …
      { P0(0) ∧ P1(0,false) }
      r := false;
      x := 0;
      while x < N do
            { P0(x) ∧ P1(x,r) ∧ (x < N=true) }
            …
            { P0(s(x)) ∧ P1(s(x),(r OR B(x))) }
            r := r OR b.x;
            x := s(x);
      od
      { P0(x) ∧ P1(x,r) ∧ (x<N=false) }
      …
      { Q(r) }
]|
```

Waarbij P_0, P_1 en *Q* als volgt zijn gedefinieerd:

```
P0(x:nat): *p
DefP0: ∀x: nat.P0(x) = (x < s(N) = true)

P1(x:nat, r:bool): *p
DefP1: ∀x,r: nat,bool.
      P1(x,r) = ((r=true)=(∃i:nat.(i<x=true)∧(B(i)=true)))

Q(r:bool): *p
DefQ: ∀r: bool.
      Q(r) = (((r=true)=(∃i:nat.(i<N=true)∧(B(i)=true)))
```

Bewijsverplichtingen

De puntjes in het bovenstaande programma zijn de bewijsverplichtingen die met de interactieve prover bewezen dienen te worden.

De bewijzen die op deze manier gevonden worden zijn:

Proof 0) Initialisatie

```
{ True }
…
{ P0(0) ∧ P1(0,false) }
```

Proof 1) Invariantie

```
{ P0(x) ∧ P1(x,r) ∧ (x < N=true) }
…
{ P0(s(x)) ∧ P1(s(x),(r OR B(x))) }
```

Proof 2) Finalisatie, namelijk:

Proof 2a) Finalisatie

```
{ P0(x) ∧ P1(x,r) ∧ (x<N=false) }
…
{ Q(r) }
```

Proof) Toekenning

```
{ P } v := w { Q }

P ⇒ Q( v := w )
```

Dit bewijs is verweven in de andere bewijzen.

Bewijsverplichtingen die door Cocktail niet worden opgelegd zijn de overigen:

Proof 2) Finalisatie, namelijk:

Proof 2b) Welgedefinieerdheid

```
[ P ⇒ def.G ]
```

Proof 3) Terminatie, bestaande uit

Proof 3a) Begrensdheid

```
[ (P ∧ G) ⇒ VF ≥ 0 ]
```

Proof 3b) Voortgang

```
{ P ∧ G ∧ VF = C } S { VF < C }
```

Proof) Toekenning

```
{ P } v := w { Q }

[ P ⇒ def.w ]
```

Illustratie: invariantiebewijs

Alle bewijsgaten kunnen worden gedicht met behulp van de eerder omschreven axioma's. Ter illustratie van de vorm van een Cocktailbewijs wordt hier het invariantiebewijs beschreven (merk op dat hier en daar ter bevordering van de leesbaarheid de uitvoer lichtelijk is aangepast)

```
x: nat
```

r: bool

P0(x) ∧ P1(x,r) ∧ (x<N=true)

{ rewrite with DefP0 and DefP1 }

(x<s(N)=true) ∧ (r=true=(∃i:nat.(i<x=true) ∧ (B(i)=true)))

∧ x<N=true

{ ∧ - elim }

(x<s(N)=true)

{ ∧ - elim }

C) *(r=true=(∃i:nat.(i<x=true) ∧ (B(i)=true)))*

{ ∧ - elim }

x<N=true

{ leibniz with AxSmaller3 }

s(x)<s(N)=true

{ leibniz with DefP0 }

P) *P0(s(x))*

B) *(r OR B(x))=true*

{ instance of AxBool2 using B(x) }

G) *B(x)=true ∨ B(x)=false*

B(x)=true

∀i:nat.¬((i<s(x)=true) ∧ (B(i)=true))

{ instance using x }

A) *¬((x<s(x)=true) ∧ (B(x)=true))*

{ repeat ThSmaller1, using x }

x<s(x)=true

{ assumption }

B(x)=true

{ ∧ - intro }

x<s(x)=true ∧ B(x)=true

{ apply with A }

False

{ ⇒ intro }

∃i:nat.(i<s(x)=true) ∧ (B(i)=true)

{ ⇒ intro }

F) *B(x)=true ⇒ (∃i:nat.(i<s(x)=true) ∧ (B(i)=true))*

B(x)=false)

D) *∀i:nat.¬((i<s(x)=true) ∧ (B(i)=true))*

{ repeat B }

(r OR B(x))=true

{ rewrite }

(r OR false)=true

{ rewrite with AxOr2 }

r = true

{ repeat C }

(r=true=(∃i:nat.(i<x=true) ∧ (B(i)=true)))

{ apply }

∃i:nat.(i<x=true) ∧ (B(i)=true))

i: nat

i<x=true ∧ B(i)=true

{ instance of D using i }

¬((i<s(x)=true) ∧ (B(i)=true)))

```
{ ∀ intro }
∀x,r:nat,bool.P0(x) ∧ P1(x,r) ∧ (x<N=true)
⇒ P0(s(x)) ∧ P1(s(x),(r OR (B(x))))
```

Er wordt in dit bewijs gebruik gemaakt van hulpstellingen, die op hun beurt bewezen zijn met de axioma's.

```
ThSmaller1= ∀x:nat.x<s(x)=true
ThSmaller2= ∀i,x:nat.i<x=true ⇒ (i<s(x)=true)
ThSmaller3= ∀i,x:nat.(i<s(x)=true) ⇒ (i<x=true) ∨ (i=x)
ThOr1= ∀b:bool.true ∨ b = true
```

De inherente definitie van natuurlijke getallen is echter niet compleet en we moeten een axioma toevoegen om inductieve bewijzen rond te krijgen:

```
AxNat1: ∀i,j:nat.(s(i)=s(j) = (i=j))
```

A.1.4 Perfect Developer

We geven eerst een korte uitleg van de syntactische conventies van Perfect Developer.

Toekenningen:

PD kent geen rechtstreekse toekenningsoperator. In plaats daarvan bestaat de volgende constructie

```
change var
satisfy proposition
via code
```

In het geval van een toekenning wordt dit:

```
change var
satisfy var' = changedvar
```

Waarvoor een verkorte schrijfwijze de volgende is:

```
var! = changedvar
```

Quantoren:

Quantoren worden in PD als volgt genoteerd:

```
forall var :: domain :- proposition
forall var : type :- proposition
```

Operatoren:

Hier volgt een lijst van veelgebruikte operatoren en hun betekenis:

```
bo              ∨
ol
|
bo
ol
bo              ∧
ol
&
bo
ol
bo              ⇒
ol
==
>
bo
ol
bo              ≠
ol
~=
bo
ol
va              <
l
<
val
<v              val - 1
al
>v              val + 1
al
va              ordening,
l               bijv: 1 ~~ 2 = below@rank
~~
val
se              set ∪ set
t
++
set
se              set ∩ set
t
--
set
se              set ∩ set = ∅
t
##
set
se              set ⊆ set
t
<<
=
set
ty              type combinatie, een int||bool
pe              variabele kan zowel de waardes 3 als
||              true bevatten.
ty
pe
```

Een speciaal geval is de prime operator (`var'`). Primed variables worden onder andere gebruikt in loops en betekenen zoveel als de huidige waarde van de variabele op dat punt in e loop. Wanneer de variabele zonder prime verschijnt, wordt de initiële waarde, dus aan het begin van de loop, bedoelt. Een variante functie van een loop kan bijvoorbeeld zijn dat (een deel van) een variabele ongewijzigd blijft:

```
forall i :: x'..<N :- v[i] = v'[i].
```

Test Instellingen

Bij het verifiëren van de algoritmes in dit document is gebruik gemaakt van een 2 Ghz PC met 1 GB RAM geheugen.

Aan de checker kan, per bewijsconditie, een maximaal aantal secondes worden meegegeven voordat deze termineert. Er is afwisselend gebruik gemaakt van 120 seconden (kort) en 600 seconden (lang). Over het algemeen is er tijdens het opstellen van een algoritme gebruik gemaakt van een korte bewijstijd en is de lange bewijstijd gebruikt als laatste poging alvorens te concluderen dat een conditie niet binnen afzienbare tijd bewijsbaar is.

Het programma

We definiëren het algoritme wederom als deel van een wrapper klasse. Dit staat ons toe de constanten als leden van deze klasse te definiëren zonder ze expliciet te instantiëren.

```
1.  class linsearch ^=
2.  abstract
3.      var
4.          N: int;
5.
6.      invariant
7.          N >= 0;
8.
9.      function B(i:int): bool
10.     pre
11.         0 <= i < N
12.     ^=
13.         ?; // unspecified
14.
15.     function linsearch: bool
16.     ^=
17.         exists i::0..<N:-B(i)
18.     via
19.         var r: bool! = false;
20.
21.         loop
22.             var x:int! = 0;
23.         change
24.             r
25.         keep
26.             0 <= x' <= N,
27.             r' = (exists i::0..<x':-B(i))
28.         until
29.             x'>=N
30.         decrease
31.             N-x';
32.         //do
33.             r! = (r | B(x)),
34.             x! = x+1;
35.         end;
36.
37.         value(r);
38.     end;
```

Onze annotatie vinden we op de volgende plaatsen terug:

$P_0 \equiv 0 \leq x \leq N$
$P_1 \equiv r = (\exists i: 0 \leq i < x: b.i)$

De invarianten P_0 en P_1 staan onder de "keep" clausule op regel 25 en 26.

$Q \equiv r = (\exists i: 0 \leq i < N: b.i)$

De postconditie Q vinden we terug in de specificatie van de functie, op regel 16, samen met de returnwaarde r op regel 36.

$VF = N-x$

De variante functie heeft een eigen clausule decrease, op regel 30.

Merk verder op dat de loop herhaald wordt totdat de guard geldt in plaats van zolang de guard geldt. De originele guard $x<N$ wordt dus geïnverteerd tot $x \geq N$.

Ten slotte kunnen we in dit voorbeeld gedefinieerdheid voor functie B definiëren door de preconditie $0 \leq i < N$ op te nemen (regel 10). Merk op dat we afgezien van deze eis de functie niet specificeren.

Bewijsverplichtingen

Op de precieze wijze waarop de perfect developer bewijsverplichtingen tot stand komen zullen we niet ingaan; deze informatie wordt door Escher Tech niet vrijgegeven en het is niet onze taak om het programma te reverse engineeren. Er is wel een lijst waarin, in woorden, de verschillende door PD gegenereerde bewijsverplichtingen beschreven worden [10].

```
Proof of verification condition: Loop initialisation establishes
end condition or a valid variant
Proof of verification condition: Loop body establishes end
condition or preserves validity of variant
```

Deze condities vormen samen het bewijs voor begrensdheid van de variante functie. Er wordt een onderscheid gemaakt tussen de waarde van de variante functie na initialisatie en na een loop iteratie. Voor de waarde van C wordt 0 gekozen; het is aan de programmeur om zijn variante functie te transponeren indien noodzakelijk.

```
Proof of verification condition: Loop body establishes end
condition or decreases variant
```

Deze conditie komt overeen met bewijsverplichting 3b - afname van de variante functie.

```
Proof of verification condition: Loop initialisation establishes
loop invariant
Proof of verification condition: Loop initialisation establishes
loop invariant
```

Twee bewijzen voor initialisatie, voor P_0 en P_1.

```
Proof of verification condition: Loop body preserves loop
invariant
Proof of verification condition: Loop body preserves loop
invariant
```

Invariantiebewijzen van P_0 en P_1.

```
Proof of verification condition: Loop body only modifies objects
in 'change' list
```

Perfect Developer voegt een extra clausule aan loops toe waarin de variabelen die door de loop worden gewijzigd worden opgenomen. Dat dit ook het geval is, wordt als bewijsverplichting toegevoegd.

```
Proof of verification condition: Return value satisfies
specification
```

Komt overeen met bewijsverplichting 2a

```
Proof of verification condition: Precondition of 'B' satisfied
Proof of verification condition: Precondition of 'B' satisfied
Proof of verification condition: Precondition of 'B' satisfied
```

Ten slotte komen deze bewijzen overeen met de gedefinieerdheidseis, namelijk bij het aanroepen van de *B* functie (regels 16, 26, 32).

Illustratie: finalisatiebewijs

Om een idee te geven van de output van Perfect Developer wordt hier het bewijs van finalisatie verbatim herhaald:

```
Proof of verification condition: Return value satisfies specification
In the context of class: linsearch, declared at: linsearch.pd (7,1)
Condition generated at: linsearch.pd (43,9)
Condition defined at: linsearch.pd (23,9)
To prove: r27,9 = (exists i::0 .. <self.N • self.B(i))
Given: 0 ≤ self.N, r = false, (0 ≤ x') .& (x' ≤ self.N), r27,9 =
(exists i::0 .. <x' • self.B(i)), forall $x::$attributeNames(bool) •
different(r27,9.$x; r27,9) ==> r.$x=r27,9.$x, self.N ≤ x'
Proof:
[Take given term]
[4.0] (0 ≤ x') .& (x' ≤ self.N)
→ [simplify]
[4.9] (-1 < x') .& (-1 < (-x' + self.N))
[Take given term]
[6.0] self.N ≤ x'
→ [simplify]
[6.7] -1 < (-self.N + x')
→ [from term 4.9, -1 < (-self.N + x') is true if and only if 0 = (-x'
+ self.N)]
[6.8] 0 = (-x' + self.N)
[Take given term]
[5.0] r27,9 = (exists i::0 .. <x' • self.B(i))
→ [simplify]
[5.2] r27,9 = (exists i::(0 .. (-1 + x')).ran • self.B(i))
→ [from term 6.8, x' is equal to self.N]
[5.3] r27,9 = (exists i::(0 .. (-1 + self.N)).ran • self.B(i))
[Take goal term]
[1.0] r27,9 = (exists i::0 .. <self.N • self.B(i))
→ [from term 5.3, r27,9 is true if and only if exists i::(0 .. (-1 +
self.N)).ran • self.B(i)]
[1.1] (exists i::(0 .. (-1 + self.N)).ran • self.B(i)) = (exists i::0
.. <self.N • self.B(i))
→ [simplify]
[1.4] true
```

A.2 Binary search

A.2.1 Eiffel

Het Programma

```
0. binsearch(f: ARRAY [INTEGER]; A: INTEGER): INTEGER is
1.             -- returns an index for which f is smaller or equal
                  to A, and the next f is higher
2.             require
3.                   f.count-1 >= 1
4.                   f @ 0 <= A
5.                   A < f @ (f.count-1)
6.             local
7.                   x,y: INTEGER
8.                   h: INTEGER
9.             do
10.                  x := 0
11.
12.                  from
13.                        y := f.count-1
14.                  invariant
15.                        0 <= x
16.                        x < f.count-1
17.                        f @ x <= A
18.                        A < f @ y
19.                  variant
20.                        y-x
21.                  until
22.                        x+1 = y
23.                  loop
24.                        h := (x+y) // 2
25.
26.                        check
27.                              x < h
28.                              h < y
29.                        end
30.
31.                        if
32.                              f @ h <= A
33.                        then
34.                              x := h
35.                        else
36.                              y := h
37.                        end
38.                  end
39.
40.                  result := x
41.            ensure
42.                  0 <= result
43.                  result < f.count -1
44.                  f @ result <= A
45.                  A < f @ (result+1)
46.            end
```

Verificatiecondities

Hier geldt hetzelfde als bij de linear search. Wat betreft de extra condities gegenereerd door de conditional statement worden deze ontweken doordat er gebruik wordt gemaakt van een *else* clausule. Ook bij een *elseif* zou dit probleem er niet zijn aangezien de semantiek van de

if in Eiffel automatisch een *skip* invoert voor het geval geen van de voorgaande guards *true* zijn.

Gesorteerdheid

```
0. binsearch2(f: ARRAY[INTEGER]; A:INTEGER): BOOLEAN is
1.              -- check if A is in f[0..count-2]
2.              require
3.                    f.count-1 >=1
4.                    f @ 0 <= A
5.                    A < f @ (f.count-1)
6.                    issorted(f)
7.              do
8.                    result := (f @ binsearch(f,a) = a)
9.              ensure
10.                   result = not f.subarray(0,f.count-2).for_all
                              (agent isnotvalue(?, A))
11.             end
```

Ook hier gebruiken we een eigen gedefinieerde forall functie in de postconditie. De functie "isnotvalue" is een zelfde soort functie als de eerder gebruikte "isfalse". Het vraagteken dat als parameter aan deze functie mee wordt gegeven representeert de huidige waarde van de iterator.

Wat hiernaast nodig is, is een gesorteerdheidstest. Wederom zorgt het uitblijven van quantoren voor wat moeilijkheid. We moeten onze gesorteerdheidstest zelf schrijven:

```
0. issorted(f: ARRAY[INTEGER]): BOOLEAN is
1.              -- checks if f is sorted in nondecreasing order
2.              local
3.                    i: INTEGER
4.                    r: BOOLEAN
5.              do
6.                    if f.count <= 1 then
7.                          result := TRUE
8.                    else
9.                          from
10.                               i := 1
11.                               r := TRUE
12.                         invariant
13.                               1 <= i
14.                               i <= f.count
15.                               issorted(f.subarray (0,i-1))
16.                         variant
17.                               f.count - i
18.                         until
19.                               i = f.count
20.                         loop
21.                               r := r and f @ (i-1) <= f @ i
22.                               i := i + 1
23.                         end
24.
25.                         result := r;
26.                   end
27.             ensure
28.                   -- f is sorted
29.             end
```

Merk op dat we gesorteerdheid definiëren als zijnde die waarde die uit dit algoritme volgt. De link met de conceptuele betekenis is voor een deel verloren gegaan. In die zin is de invariant op regel 15 komisch.

A.2.2 ESC/Java

Het Programma

```
1.  public abstract class BinarySearch
2.  {
3.     /** The function that describes what is being sought. */
4.     //@ requires j >= 0;
5.     public abstract /*@ pure @*/ int f(int j);
6.
7.     /** The last integer in the search space, this describes the
8.      * domain of f, which goes from 0 to the result.
9.      */
10.    //@ ensures 0 <= \result;
11.    public abstract /*@ pure @*/ int N();
12.
13.    //@ ensures f(0) <= A() && A() < f(N());
14.    public abstract /*@ pure @*/ int A();
15.
16.    //@ ensures 0 <= \result && \result < N();
17.    //@ ensures f(\result) <= A() && A() < f(\result+1);
18.    public int binarySearch()
19.    {
20.            int x,y;
21.
22.            x = 0;
23.            y = N();
24.
25.            //@ maintaining 0 <= x;
26.            //@ maintaining x < y;
27.            //@ maintaining y <= N();
28.            //@ maintaining f(x) <= A();
29.            //@ maintaining A() < f(y);
30.            //@ decreasing y-x;
31.            while (!(x+1  == y))
32.            {
33.                    int h;
34.
35.                    h = (x+y) / 2;
36.
37.                    if (f(h) <= A())
38.                    {
39.                            x = h;
40.                    }
41.                    else
42.                    {
43.                            y = h;
44.                    }
45.    }
46.    return x;
47. }
```

Bewijsverplichtingen

ESC geeft geen waarschuwingen bij dit algoritme.

Opvallend is dat ook zonder toevoegen van de loop-invarianten en variante functie er geen waarschuwingen worden gegeven, terwijl deze weldegelijk nodig zijn voor het bewijs van de potconditie.

Gesorteerdheid

We proberen ook nog de laatste stap van het algoritme te implementeren, namelijk het introduceren van gesorteerdheid en de daaruit volgende eigenschap dat de binary search gebruikt kan worden om het bestaan van elementen in een array te bepalen:

```
1.  //@ ensures \result == (\exists int i; 0 <= i && i < N(); f(i) == A());
2.  //@ requires (\forall int i; 0 <= i && i < N(); (\forall int j; 0 <= j &&
    j < N(); (i <= j ==> f(i) <= f(j))));
3.  public boolean binarySearch2()
4.  {
5.    int x = binarySearch();
6.    //@ assert 0 <= x;
7.    //@ assert x < N();
8.    //@ assert f(x) <= A();
9.    //@ assert A() < f(x+1);
10.
11.   //@ assert f(x) <= A();
12.   //@ assert f(x) == A() || f(x) < A();
13.
14.   //@ assert f(x) != A() ==> f(x) < A();
15.   //@ assert f(x) < A() ==> (\forall int i; 0 <= i && i < x; f(i) < A());
16.   //@ assert (\forall int i; 0 <= i && i < x; f(i) < A())
          ==> (\forall int i; 0 <= i && i < x; f(i) != A());
17.   //@ assert f(x) != A() ==> (\forall int i; 0 <= i && i < x;
          f(i) != A());
18.
19.   //@ assert A() < f(x+1);
20.   //@ assert (\forall int i; x < i && i <= N(); A() < f(i));
21.   //@ assert (\forall int i; x < i && i <= N(); f(i) != A());
22.
23.   /*@ assert (
24.     @            (\forall int i; 0 <= i && i < x; f(i) != A())
25.     @            &&
26.     @            (\forall int i; x < i && i <= N(); f(i) != A())
27.     @            &&
28.     @            f(x) != A()
29.     @        ) ==> (\forall int i; 0 <= i && i <= N(); f(i) != A());
30.     @*/
31.
32.   //@ assert (f(x) != A()) ==> (!(\exists int i; 0 <= i && i < N();
    f(i)==A()));
33.
34.   //@ assert (f(x) == A()) ==> ((\exists int i; 0 <= i && i < N();
    f(i)==A()));
35.
36.   return f(x) == A();
```

Zoals bij de linear search het geval was, hebben we de stappen van het bewijs uitgeschreven met behulp van asserties. ESC heeft moeite met een aantal van deze stappen:

```
BinarySearch: binarySearch2() ...
------------------------------------------------------------------
C:\ESCJava-2\myprojects\BinarySearch.java:67: Warning: Possible
assertion failure (Assert)
//@ assert f(x) < A() ==> (\forall int i; 0 <= i && i  ...
------------------------------------------------------------------

C:\ESCJava-2\myprojects\BinarySearch.java:69: Warning: Possible
assertion failure (Assert)
//@ assert f(x) != A() ==> (\forall int i; 0 <= i && i ...
------------------------------------------------------------------

C:\ESCJava-2\myprojects\BinarySearch.java:72: Warning: Possible
assertion failure (Assert)
//@ assert (\forall int i; x < i && i <= N(); A() < f( ...
------------------------------------------------------------------

C:\ESCJava-2\myprojects\BinarySearch.java:73: Warning: Possible
assertion failure (Assert)
//@ assert (\forall int i; x < i && i <= N(); f(i) !=  ...
------------------------------------------------------------------

C:\ESCJava-2\myprojects\BinarySearch.java:84: Warning: Possible
assertion failure (Assert)
```

```
//@ assert (f(x) != A()) ==> (!(\exists int i; 0 <= i  ...
------------------------------------------------------------

C:\ESCJava-2\myprojects\BinarySearch.java:89: Warning:
Postcondition possibly not established (Post)

Associated declaration is:

//@ ensures \result == (\exists int i; 0 <= i && i < N();
f(i) ...
```

(merk op dat de gegeven uitvoer slechts ter illustratie dient, de gegeven regelnummers komen niet overeen met die uit de code listing erboven)

Wanneer we de vergelijking trekken met de onbewijsbare postconditie van de linear search, lijkt het erop dat deze stappen te ingewikkeld zijn voor de ESC prover.

We laten dit voorbeeld verder voor wat het is. Het is mogelijk om een andere bewijzer aan ESC toe te wijzen, waardoor we in theorie de benodigde stellingen kunnen invoeren of interactief bewijzen, maar dit valt buiten het bereik van dit document en past niet in ons doel van het vinden van een systeem voor automatische statische verificatie van programma's.

We komen eenzelfde soort problemen tegen bij Perfect Developer in het volgende paragraaf.

A.2.3 Cocktail

Definities

We maken dit keer een onderscheid tussen de booleans als programma-datatype en hun verband met de interne Cocktail Booleans. Dit staat ons toe om vertalingen te maken die het rekenen vereenvoudigen.

Zo kunnen we op deze wijze de Boolean relatie '<' definiëren als propositie.

Booleans:

```
bool: *s
true(): bool
false(): bool

AxBool1: ∀b: bool.b=true ∨ b=false

istrue(): *p

AxBool2: istrue(true)
AxBool3: ¬istrue(false)
```

Integers:

Ook hier gebruiken we naturals in plaats van integers.

```
AxNat1: ∀i,j:nat.(s(i)=s(j) = (i=j))
```

Let op het verschil tussen de propositie en de in guards bruikbare booleanse waarde. We beginnen met de proposities:

```
smaller(a,b: nat): *p
```

```
AxSmaller1: ∀i:nat.0 < s(i)
AxSmaller2: ∀i:nat.¬(i < 0)
AxSmaller3: ∀i,j:nat.s(i) < s(j) ≡ i < j

leq(a,b:nat): *p

AxLeq: ∀a,b:nat.a ≤ b ≡ ( a < b ∨ a=b )
```

En we maken speciale functies voor de gebruikte guards:

```
smallerGuard(a,b:nat): bool

AxSmallerGuard: ∀a,b:nat.
      istrue(smallerGuard(a,b)) ≡ (a < b)

leqGuard(a,b:nat): bool

AxLeqGuard: ∀a,b:nat.
      istrue(leqGuard(a,b)) ≡ ( a ≤ b )

notequalGuard(a,b:nat): bool

AxNotEqualGuard: ∀a,b:nat.istrue(notequalGuard(a,b)) ≡ (a ≠ b)
```

Voor de duidelijkheid wordt voor deze functies de volledige naam gebruikt en niet de gebruikelijke infix notatie. In het programma zelf zal dit wel worden gedaan - het is aan de lezer om uit de context op te maken of het om de propositie of de functie gaat.

Midpoint:

Voor het zoeken van het middelpunt tussen x en y (h := x+y div 2) zijn er twee mogelijkheden. We kunnen de operaties in kwestie, namelijk div en plus, definiëren in Cocktail en de stelling bewijzen dat x+y div 2 een punt tussen x en y is. De tweede mogelijkheid is om de precieze betekenis van x+y div 2 buiten beschouwing te laten en de operatie alleen op deze eigenschap te definiëren.

Het voordeel van de eerste aanpak is dat deze cruciale eigenschap van h wordt bewezen. Het nadeel is dat dit een fors bewijs is dat mathematisch eenvoudig is in te zien.

We kiezen voor de tweede aanpak en komen later op dit bewijs terug.

```
midpoint(a,b:nat):nat

AxMidpoint: ∀a,b:nat.a < midpoint(a,b) < b
```

Hier valt overigens Cocktail's gebrek aan voortgangsbewijs op: als we *a* meenemen als mogelijke waarde voor het midpoint kunnen we geen voortgang garanderen, maar alle door Cocktail opgelegde bewijsverplichtingen desalniettemin invullen.

Arrays:

We definiëren arrays dit keer wat implicieter dan bij de bounded linear search. Aangezien het gebruikte array een constante is, hoeft voor arrays alleen een lees-operatie te worden gedefinieerd.

```
array: *s
arrayGet(a:array,i:nat): nat
```

Merk op dat we de bounds van het array, i.e.: gedefinieerdheid, niet meenemen in de definitie.

Constanten, gegevens:

```
f: array
N: nat
A: nat

AxConstf1: f(0) ≤ A
AxConstf2: A < f(N)

AxConstN: 1 ≤ N
```

Het programma

```
var x: nat
|[ var y: nat
  { True }
  …
  { P0(0,N) ∧ P1(0,N) }
  x := 0
  y := N;
  while s(x)≠y do
  |[
    { P0(x,y) ∧ P1(x,y) ∧
     ( (s(x)≠y)=true) }
    …
    { P0(x,y)) ∧ P1(x,y) ∧ midpoint(x,y)=midpoint(x,y) }
    h := midpoint(x,y)
    if f(h) ≤ A then
      { P0(x,y) ∧ P1(x,y) ∧ (h=midpoint(x,y)) ∧
        ((f(h) ≤ A)=true) }
      …
      { P0(h,y) ∧ P1(h,y) }
      x := h
    else
      { P0(x,y) ∧ P1(x,y) ∧ (h=midpoint(x,y)) ∧
        ((f(h) ≤ A)=false) }
      …
      { P0(x,h) ∧ P1(x,h) }
      y := h
    fi
  ]|
  od;
  { P0(x,y) ∧ P1(x,y) ∧ ((s(x) ≠ y)=false) }
  …
  { Q0(x) ∧ Q1(x) }
]|
```

Waarbij P_0, P_1 en Q_0, Q_1 als volgt zijn gedefinieerd:

```
P0(x,y:nat): *p
DefP0: ∀x,y: nat.P0(x,y) ≡
       0 ≤ x ∧ x < y ∧ y ≤ N

P1(x,y:nat): *p
DefP1: ∀x,y: nat.P1(x,y) ≡
       f(x) ≤ A ∧ A < f(y)
```

```
Q0(x:nat): *p
DefQ0: ∀x: nat.
      Q0(x) ≡ x < N

Q1(x:nat): *p
DefQ1: ∀x: nat.
      Q1(x) ≡ f(x) ≤ A ∧ A < f(s(x))
```

Bewijsverplichtingen

Wederom bewijst Cocktail initialisatie, invariantie en finalisatie met uitzondering van welgedefinieerdheid. Door de wijze waarop de if-statement in Cocktail gedefinieerd is, is de eis dat één van de guards true is altijd waar. Er wordt in de programmavertaling impliciet meegenomen dat $\neg(f(h) \leq A) \equiv (A < f(h))$.

Ook voor de if geldt dat welgedefinieerdheid niet wordt gecontroleerd.

Uitwerking Midpoint

Het bewijs voor de stelling dat x+y div 2 een correcte manier is om het middelpunt tussen x en y te vinden wordt in [16] als volgt gegeven:

```
x < (x+y) div 2 < y
≡ { calculus }
x + 1 ≤ (x+y) div 2 ≤ y - 1
⇐ { div 2 is ascending }
2x + 2 ≤ x+y ≤ 2y - 2
≡ { calculus }
x + 2 ≤ y
≡ { calculus }
x < y ∧ x+1 ≠ y
⇐ { definition of P0 }
P0 ∧ x+1 ≠ y
```

Een Cocktail variant van dit bewijs bestaat uit 77 stappen en gebruikt de volgende definities en hulpstellingen:

```
div(a,b:nat):nat
times(a,b:nat):nat
plus(a,b:nat):nat
less(a,b:nat):*p
leq(a,b:nat):*p
between(a,b,c:nat):*p

AxDiv: ∀a,b,c:nat.0<b ⇒
      a div b = c
      ≡
      ∃r:nat.(a=(b*c)+r) ∧ (0 ≤ r < b)
```

```
AxPlus1: ∀n:nat.n+0=n
AxPlus2: ∀a,b:nat.a+b=b+a
AxBetween1: ∀a,b,c:nat.a < b < c ≡ 2*a < 2*b < 2*c
AxBetween2: ∀a,b,c:nat.a+1 < b+1 < c+1 ⇒ a < b < c
AxBetween3: ∀n:nat.0<n<2 ≡ n=1
AxTwice: ∀n:nat.2*n=n+n
AxLess1: ∀a,b:nat.a<b ≡ ∀c:nat.a+c < b+c
AxLess2: ∀a,b:nat.a<b ⇒ ∀c:nat.a < b+c
AxLess3: 0 < 2
AxLessLeq1: ∀a,b:nat.a < b ≡ a+1 ≤ b
```

De hulpstellingen, aangenomen als axioma's, dienen zelf ook bewezen te worden in Cocktail. Dit kan door middel van constructieve definities en inductieve bewijzen, waarvan de details niet zijn opgenomen.

Gesorteerdheid

Afgezien van een kleine verandering in de programmatekst is de kern van deze uitbreiding de volgende stelling:

```
∀x:nat.Q0(x) ∧ Q1(x) ∧ sorted(f)
        ⇒
        f(x)=A ≡ ∃i:nat.f(i)=A
```

Waarbij gesorteerdheid als volgt kan worden gedefinieerd:

```
∀f:array.sorted(f) ≡
        (∀i,j:nat.i≤j ≡ f(i)≤f(j))
```

Dit bewijs kan met behulp van de interactieve bewijzer worden opgesteld (ongeveer 150 stappen) en gebruikt hulpstellingen zoals:

```
a<b ⇒ a≤b
a≤b ∨ b<a
¬(a=b ∧ a<b)
¬(a<b ∧ b≤a)
a<b ⇒ (b=s(a) ∨ s(a)<b)
```

A.2.4 Perfect Developer

Het Programma

```
1. class binsearch ^=
2. abstract
3.     var
4.         N: nat,
5.         A: nat,
6.         f: seq of int;
7.
8.     invariant
9.         N > 0,
10.        #f = >N,
11.        f[0] <= A,
12.        A < f[12];
13.
14.    function binsearch: int
15.    satisfy
16.        0 <= result < N,
17.        f[result] <= A < f[>result]
18.    via
19.        var x: int! = 0;
20.
21.        loop
22.            var y: int! = N;
23.        change
24.            x
25.        keep
26.            x' < y' <= N,
27.            f[x'] <= A < f[y']
28.        until
29.            x'+1=y'
30.        decrease
31.            y'-x';
32.        //do
33.            var h: int;
34.
35.            h != ((x+y)/2);
36.
37.            assert x < h < y;
38.
39.            if
40.                [f[h] <= A]: x != h;
41.                [A  < f[h]]: y != h;
42.            fi;
43.
44.        end;
45.
46.        value(x);
47.    end;
48.end;
```

Bewijsverplichtingen

Zoals bij het eerdere voorbeeld worden invariantie, finalisatie en terminatie bewezen.

Nieuwe, door PD gegenereerde, bewijsverplichtingen komen overeen met de extra verplichtingen voor de if-constructie. Namelijk het abortiebewijs dat ten minste één van de guards true is en extra gedefinieerdheidseisen.

Opvallend is de hoeveelheid aan gedefinieerdheidseisen: waar deze op papier meestal als triviaal worden weggelaten, zorgt de volhardendheid van PD ervoor dat ze allen boven water komen. Het programma bevat 8 array indexeringen, 4 casts en een div operatie.

Gesorteerdheid

Gesorteerdheid is een voorgedefinieerde propositie van lijsten in Perfect Developer. We voegen als invariant toe dat de lijst *f* is gesorteerd en definiëren voor de extra stap een tweede functie:

```
1. function binsearch2: bool
2. ^=
3.     (exists i::0..<N:-(f[i]=A))
4. via
5.     var x: int! = binsearch;
6.
7.     var r: bool! = (f[x]=A);
8.
9.     value(r);
10.end;
```

Helaas is deze stap een brug te ver voor de PD prover. De uitvoer van het programma vertelt ons het volgende:

```
Failed to prove verification condition: Return value satisfies
specification
In the context of class: binsearch, declared at: binsearch2.pd
(7,1)
Condition generated at: binsearch2.pd (104,9)
Condition defined at: binsearch2.pd (75,9)
To prove: r = (exists i::0 .. <self.N • self.f[i as nat] = self.A)
Reason: Exhausted rules
Could not prove any of:
self.f[#self.f - 1] ≤ self.A
~(self.f[$c_i] = self.A)
self.f[1 + x] ≤ (1 + self.f[x])
self.f[#self.f - 1] ≤ (1 + self.f[x])
self.f[1 + x] ≤ self.A
In the case that:
0 ≤ $c_i
(1 + $c_i) < #self.f
```

Eerste Poging: asserties

Als eerste poging spellen we het bewijs uit via een aantal asserties, vergelijkbaar met de asserties die we in ons JML programma toevoegden:

```
1.  assert
2.      f[x] <= A,
3.      (f[x] <= A) ==> (f[x] = A | f[x] < A),
4.      (f[x] ~= A) ==> (f[x] < A),
5.  *   (f[x] < A) ==> (forall i::0..<x:-f[i] < A),
6.      (forall i::0..<x:-f[i] < A) ==>
            (forall i::0..<x:-f[i] ~= A),
7.  *   (f[x] ~= A) ==> (forall i::0..<x:-f[i] ~= A);
8.
9.  assert
10.     A < f[>x],
11. *   (A < f[>x]) ==> (forall i::>x..N:- A < f[i]),
12.     (forall i::>x..N:- A < f[i]) ==>
            (forall i::>x..N:- f[i] ~= A),
13. *   (forall i::>x..N:- f[i] ~= A);
14.
15. assert
16.     (f[x] ~= A) ==> ((forall i::0..<x:-f[i] ~= A)
            & (forall i::>x..N:- f[i] ~= A) & (f[x] ~= A)),
17. *   ((forall i::0..<x:-f[i] ~= A) &
            (forall i::>x..N:- f[i] ~= A) &
            (f[x] ~= A)) ==> (forall i::0..N:-f[i]~=A),
18.     (forall i::0..N:-f[i]~=A) ==>
            ((exists i::0..<N:-(f[i]=A))=false),
19. *   (f[x] ~= A) ==> ((exists i::0..<N:-(f[i]=A))=false);
20.
21. assert
22.     (f[x] = A) ==> ((exists i::0..<N:-(f[i]=A))=true);
```

Een aantal hiervan zijn niet bewijsbaar door de prover, deze zijn aangegeven met een *.

Opvallend is dat de assertie op regel 4 niet te bewijzen is, aangezien deze direct volgt uit gesorteerdheid. We testen een verwante eigenschap:

```
property sortedB(a,b:nat,g:seq of int)
pre
    a < #g,
    b < #g,
    a <= b,
    g.isndec
assert
    g[a] <= g[b];
```

Ook deze toch vrij fundamentele eigenschap blijkt onbewijsbaar.

Tweede poging: eigen gesorteerdheid

De gesorteerdheidseis (*g.isndec*) lijkt niet geschikt te zijn voor dit bewijs. Een kijkje in de specificatie leert ons dat gesorteerdheid als volgt is gedefinieerd:

```
isndec ^=
    #self <= 1
    |
    (forall i::1..<#self :-
        (self[i] ~~ self[<i]) ~=  below@rank);
```

De ~~ operator is een ordeningsoperator en kan de waarden below, same of above hebben. In GCL termen:

$$
\begin{aligned}
&f.isndec \\
&\equiv \\
&\#f \leq 1 \\
&\vee \\
&\forall i: 0 < i < \#f: \neg(f[i] < f[i-1])
\end{aligned}
$$

Deze definitie is niet handig voor het binary search probleem, aangezien we stellingen willen bewijzen over hele subreeksen van de functie. We proberen een eigen gesorteerdheidsfunctie te introduceren:

```
function issorted(f:seq of int): bool
^=
    forall a::0..<#f:-forall b::0..<#f :-
        (a <= b ==> f[a] <= f[b]);
```

Dit is een letterlijke kopie van de property waar de prover eerder problemen mee had. We vervangen de gesorteerdheidseis in ons programma door de onze en kijken nogmaals naar de asserties:

```
1. assert
2.     f[x] <= A,
3.     (f[x] <= A) ==> (f[x] = A | f[x] < A),
4.     (f[x] ~= A) ==> (f[x] < A),
5. *   (f[x] < A) ==> (forall i::0..<x:-f[i] < A),
6.     (forall i::0..<x:-f[i] < A) ==>
           (forall i::0..<x:-f[i] ~= A),
7. *   (f[x] ~= A) ==> (forall i::0..<x:-f[i] ~= A);
8.
9. assert
10.    A < f[>x],
11.*   (A < f[>x]) ==> (forall i::>x..N:- A < f[i]),
12.    (forall i::>x..N:- A < f[i]) ==>
           (forall i::>x..N:- f[i] ~= A),
13.*   (forall i::>x..N:- f[i] ~= A);
14.
15.assert
16.    (f[x] ~= A) ==> ((forall i::0..<x:-f[i] ~= A)
           & (forall i::>x..N:- f[i] ~= A) & (f[x] ~= A)),
17.**  (
         (forall i::0..<x:-f[i] ~= A)
         &
         (forall i::>x..N:- f[i] ~= A)
         &
         (f[x] ~= A)
       )
       ==> (forall i::0..N:-f[i]~=A),
18.    (forall i::0..N:-f[i]~=A) ==>
           ((exists i::0..<N:-(f[i]=A))=false),
19.**  (f[x] ~= A) ==> ((exists i::0..<N:-(f[i]=A))=false);
20.
21.assert
22.    (f[x] = A) ==> ((exists i::0..<N:-(f[i]=A))=true);
```

Er zijn nog steeds asserties die problemen geven (gemarkeerd met een tweede *).

Als we kijken naar regel 16, dan staat hier eigenlijk een vrij eenvoudige stelling:

$$
\begin{array}{l}
\quad \forall i{:}0 \le i < x{:}\ f[i] \neq A \\
\quad \wedge \\
\quad f[x] \neq A \\
\quad \wedge \\
\quad \forall i{:}\ x+1 \le i \le N{:}\ f[i] \neq A \\
\Rightarrow \\
\quad \forall i{:}\ 0 \le i \le N{:}\ f[i] \neq A
\end{array}
$$

De volgende problematische bewijsregel (18) voegt 15, 16 en 17 samen. De bewijsregels zijn als volgt opgebouwd:

$$
\begin{array}{l}
15)\ A \Rightarrow B \\
16)\ B \Rightarrow C \\
17)\ C \Rightarrow D \\
18)\ A \Rightarrow D
\end{array}
$$

Een divide en conquer manier van het ontdekken van het exacte punt waar de prover faalt, is soms mogelijk, maar biedt vaak ook enkel frustratie. Niet alleen is dit een (onhandige) manier van direct bewijzen, wat in strijd is met het principe achter PD, maar het is ook foutgevoelig doordat een onbewezen assertie dan wel aan de prover kan liggen dan wel aan human error.

We hebben echter nog een laatste optie: het fors opschroeven van de aan de prover toegestane tijd. De twee niet oplosbare bewijzen hierboven falen met een time-out (in tegenstelling tot een exhaustion of rules) en wellicht dat het slechts een kwestie van tijd is.

Derde poging: extra tijd

We zetten de toegestane tijd op 10 minuten per bewijsverplichting en laden ons programma nogmaals door de bewijzer.

Na een kleine 10 minuten slaagt Perfect Developer erin om onze binary search als correct te bewijzen. Het langste bewijs duurde 260 seconden.

Het is mogelijk om via eenzelfde soort trial and error methode het aantal hulpasserties te minimaliseren en op eenzelfde manier kan wellicht een link tussen onze gesorteerdheidseis en de standaard isndec eis worden gelegd. We gaan hier verder niet op in. Merk hierbij op dat iedere aanpassing mogelijk 520 seconden nodig heeft ter controle, omdat dit de tijd is die de twee problematische asserties nodig hebben.

Alternatieve oplossing:

Er wordt ook een oplossing voor de binary search gegeven als voorbeeld bij Perfect Developer. We gebruiken dit als een alternatieve werkwijze:

```
1. final class Table of X
2. require X has total operator ~~(arg) end
3. ^=
4. abstract
5.   var members: seq of X;
6.
7.   // Invariant states that the members are in a non-
        decreasing order
8.   invariant members.isndec;
9.
10.interface
11.  // Find the index of the first element that ranks with or
     below the parameter
12.  function search(x: X): nat
13.    satisfy result <= #members,
14.      forall z::0..<result :- x > members[z],
15.      forall z::result..<#members :- ~(x > members[z])
16.    via
17.      // Implement with a binary search
18.      var i: nat != 0;
19.      loop
20.        var k: int != #members;
21.        change i
22.        keep
23.          0 <= i',
24.          i' <= k',
25.          k' <= #members,
26.          forall z::0..<i' :- x > members[z],
27.          forall z::k'..<#members :- ~(x > members[z])
28.        until i' = k'
29.        decrease k' - i';
30.        // Start of loop body...
31.        // Set up the partition value
32.        let p ^= (i + k)/2;
33.        assert i <= p < k;
34.        if
35.         [x > members[p]]:
36.        i! = >p;
           // If x ranks above members[p] then move i to >p
37.         []:
38.          k! = p;
             // If x ranks with or below members[p] then move
                k to p
39.        fi
40.      end;
41.      value i;
42.    end;
43.
44.  // Count operator
45.  operator #: nat
46.    ^= #members;
47.
48.  // Index operator to access the members of the list
49.  operator [](index: nat): X
50.    pre index < #self
51.    ^= members[index];
52.
53.  // Build from a set
54.  build{mem: set of X}
55.    post members! = mem.permndec;
56.end;
```

Deze versie van het algoritme kan in 33 seconden worden bewezen.

Wat we zien is dat deze versie iets lakser is in de postconditie. De postconditie bestaat in feite uit de twee asserties die moeilijk te bewijzen waren in onze eerste poging en die wel bewijsbaar werden toen we een eigen versie van gesorteerdheid gebruikte. De laatste stap, namelijk het afleiden van membership met behulp van de binary search, wordt in deze versie niet gezet.

Daarnaast wordt gesorteerdheid opgenomen in de invariant, via regels 25 en 26 en de klasse invariant op regel 7. Het opnemen van de invarianten in regels 25 en 26 helpt blijkbaar bij ons probleem dat ze niet rechtstreeks konden worden afgeleid uit (de interne definitie van) gesorteerdheid.

Uit nieuwsgierigheid voegen we de laatste stap zelf toe:

De postconditie van de binary search volgens PD is:

```
forall z::0..<result :- x > members[z],
forall z::result..<#members :- ~(x > members[z])
```

In GCL termen:

$$\forall i: 0 \leq i < r: f(i) < x$$
$$\forall i: r \leq i < N: x \leq f(i)$$

De eis voor membership blijft dus:

$$x = f(r)$$

En aangezien de functie *search* een getal kleiner of gelijk aan de lengte van het array kan retourneren doen we een kleine gevalsonderscheidsstap:

```
1. function search2(x:X):bool
2. ^=
3.     (exists i::0..<#members:-(members[i]~~x=same@rank))
4. via
5.     var i: int! = search(x);
6.     var r: bool;
7.
8.     if
9.     [i = #members]: r! = false;
10.    [i < #members]: r! = (members[i]~~x = same@rank);
11.    fi;
12.
13.    value(r);
14.end;
```

Ook in deze versie leidt deze stap tot problemen (ook met 10 minuten bewijstijd).

B Dijkstra's Kortste Pad

B.1 Stepwise Refinement (Cocktail)

Notatie:

In deze paragraaf wordt gebruik gemaakt van Cocktail stellingen, maar ook van stellingen van een hogere orde, die niet direct in Cocktail kunnen worden ingevoerd:

```
istrue(true) = True
istrue(true) = ¬False
istrue(false) = False
```

```
∀p:*p.(p ⇒ False) ≡ ¬p
```

Stellingen zoals deze laatste (buiten de omranding) zijn van een hoger niveau dan voor de gebruiker toegankelijk is. Ze kunnen worden aangeroepen in de interactieve stellingbewijzer maar er bestaat geen mogelijkheid om ze toe te voegen of te veranderen. In het vervolg zal vaker van dit soort "pseudo-cocktail" gebruik gemaakt worden. Ze zijn herkenbaar aan het feit dat daadwerkelijke cocktail definities omrand zijn. Ook worden er soms onomrande stellingen gebruikt als "opmars" naar een daadwerkelijke cocktail stelling.

Ter bevordering van de leesbaarheid is in dit document ietwat losser omgesprongen met naamgevingen dan in Cocktail het geval is. Sommige functienamen worden 'overloaded' gebruikt voor verschillende invoerwaardes en er wordt gebruik gemaakt van wiskundige notaties zoals "∈" die in Cocktail niet beschikbaar zijn. Ook worden hier en daar de typeringen van argumenten weggelaten wanneer de functie een speciale notatie heeft.

B.1.1 Basis

Ten eerste zullen we een basis definiëren van bruikbare types. Het algoritme maakt gebruik van natuurlijke getallen (met oneindig), booleans, sets, arrays, e.d. en deze zullen moeten worden gespecificeerd in Cocktail.

Booleans

```
bool: *s
true(): bool
false(): bool

true ≠ false
∀b:bool.(b=true ∨ b=false)
```

Operaties op booleans

```
not(b:bool): bool
∀b:bool.not(b) ≠ b
```

Naturals

Naturals zijn een standaard beschikbaar type:

```
nat: *s
0: nat
s(n:nat): nat
```

Dit type verschilt van door de gebruiker definieerbare types op een belangrijk punt: er is een speciale inductie tactiek beschikbaar om stellingen over naturals te bewijzen:

```
∀p:*p(n:nat).
        (p(0) ∧ (∀n:nat.p(n) ⇒ p(s(n))))
        ⇒
        ∀n:nat.p(n)
```

Verder voegen we nog zelf twee stellingen over natuurlijke getallen toe die noodzakelijk zijn voor een aantal inductieve bewijzen:

```
∀n:nat.s(n) ≠ 0
∀a,b:nat.s(a)=s(b) ≡ a=b
```

Extended Naturals (nat ∪ {∞})

Ons programma vereist dat de natuurlijke getallen worden uitgebreid met de waarde oneindig.

```
nat⁺: *s
∞(): nat⁺
natvalue(n:nat): nat⁺

∀n:nat⁺.(n=∞) ∨ (∃i:nat.n=natvalue(i))
∀i:nat.natvalue(i) ≠ ∞
```

Operaties op nat⁺

```
n1 < n2: *p
n1 ≤ n2: *p
n1 + n2: nat⁺
min(n1,n2): nat⁺
```

De operaties en proposities hierboven kunnen, net als types, constructief worden gedefinieerd. Voor de *plus* geldt bijvoorbeeld:

```
∀n:nat.0+n=n
∀n,m:nat.s(m)+n=s(m+n)
```

Bruikbare stellingen kunnen nu met inductie worden bewezen, zoals:

```
∀m,n:nat.m+n = n+m
```

Er is echter voor gekozen om deze constructieve definities niet te gebruiken en in plaats hiervan een aantal stellingen over de functies direct (als axioma) aan te nemen. Voor de bewijzen van het algoritme zijn namelijk slechts een paar, eenvoudig te controleren, stellingen noodzakelijk:

```
∀n:nat⁺.min(∞,n)=n
∀a,b:nat⁺.(min(a,b)=a)≡(a ≤ b)
∀a,b:nat⁺.(min(a,b)=min(b,a))
∀n:nat⁺.n + ∞ = ∞
∀n:nat⁺.∞ + n = ∞
∀n:nat⁺.n ≤ n
∀n:nat⁺.n ≤ ∞
∀n:nat⁺.¬(∞ < n)
∀a,b:nat⁺.a ≤ b ∨ b ≤ a
∀a,b:nat⁺.(a≠∞) ∧ b≠∞) ⇒ a+b < ∞
∀a,b:nat⁺.¬(a < b ∧ b ≤ a)
∀a,b,c:nat⁺.(a+b < c) ⇒ (a < c)
∀a,b,c:nat⁺.(a ≤ b ∧ b ≤ c) ⇒ (a ≤ c)
∀a,b,c:nat⁺.(a ≤ b ∧ b < c) ⇒ (a < c)
∀a,b,c,d,e:nat⁺.(a ≤ b+c ∧ d ≤ a+e) ⇒ (d ≤ b+c+e)
```

Sets

```
set: *s
empty(): set
```

Operaties/Stellingen

Definiërende eigenschap van een set is de membership functie:

```
n ∈ z: *p
∀n:vertex.n ∉ empty
```

Alle operaties en eigenschappen van sets kunnen worden gedefinieerd naar hun effect op de membership. Een definitie van de onderliggende structuur van de set is niet nodig.

```
∀z1,z2: set.(z1 = z2) ≡ (∀i: vertex.(i ∈ z1)≡(i ∈ z2))
```

Een aantal operaties/proposities op sets is:

```
z1 ⊆ z2: *p
z1 ∪ z2: set
z1\z2: set
z++{n}: set
z--{n}: set

∀z1,z2: set.(z1 ⊆ z2) ≡ (∀i: vertex.(i ∈ z1 ⇒ i ∈ z2))
∀z1,z2: set.∀n:vertex.(n ∈ z1 ∪ z2) ≡ (n ∈ z1 ∨ n ∈ z2)
∀z1,z2: set.∀n:vertex.(n ∈ z1\z2) ≡ (n ∈ z1 ∧ ¬n ∈ z2)
∀n,n2:vertex.∀z:set.(n ∈ z++{n2}) ≡ (n ∈ z ∨ (n=n2))
∀n,n2:vertex.∀z:set.(n ∈ z--{n2}) ≡ (n ∈ z ∧ ¬(n=n2))
```

In ons programma hebben we ten slotte nog een functie nodig om een element uit een set op te halen en een booleanse functie die retourneert of de set leeg is (voor de guard van de hoofdloop):

```
any(z:set): vertex
∀z:set.z≠empty ⇒ any(z) ∈ z
```

Deze *any* functie wordt later nog aangepast om een element met een specifieke eigenschap uit de set te retourneren.

```
isempty(z:set): bool
∀z:set.isempty(z)=true ≡ z=empty
```

Arrays

Arrays worden gedefinieerd via hun getter en setter functies, als volgt:

```
array:*s
get(a:array,index:vertex): nat⁺
      (notatie: a(i))
set(a:array,index:vertex,value:nat⁺): array
      (notatie: a[i:v])
```

De relatie tussen getter en setter is als volgt (voor $i \neq j$):

```
a[i:v](i) = v
a[i:v](j) = a(j)
```

In Cocktail:

```
∀d,d2:array.∀i:vertex.∀v:nat⁺.(d2=d[i:v]) ≡
      ((d2(i)=v) ∧ (∀x:vertex.(x≠i) ⇒ (d2(i)=d(i))))
```

Ook hier geldt dat een specifiekere constructieve definitie mogelijk zou zijn, maar overbodig is.

B.1.2 Grafen

Aangezien ons algoritme zich bezig houdt met grafen is het noodzakelijk een aantal types en begrippen de specificeren.

Aan de ene kant hebben we concrete, in programmatekst bruikbare, types nodig zoals vertexes, edges en de gewichtsfunctie. Aan de andere kant zullen voor de bewijzen begrippen als het kortste pad en bereikbaarheid moeten worden gedefinieerd.

Vertexes, edges, weight

Een graaf *(V,E)* bestaat uit vertexes en edges:

```
vertex:*s
an_edge:*s
```

Waarbij een edge twee vertexes omhelst:

```
edge(v1,v2:vertex): an_edge
```

Tevens is er een set voor edges. Voor details zie het *set* type hierboven. Er worden in het programma geen transformaties op de *edge_set E* uitgevoerd, waardoor slechts een beperkte subset van stellingen/operaties herhaald hoeft te worden.

```
edge_set: *s
```

Er bestaat een gewichtsfunctie *w*, die voor elke edge (of vertexpaar) een natuurlijk getal dan wel oneindig oplevert (de edge_set *E* wordt later behandeld):

```
w(v1,v2:vertex): nat⁺
∀v1,v2:vertex.edge(v1,v2) ∈ E ⇒ w(v1,v2) ≠ ∞
```

In tegenstelling tot andere types zullen we voor paden wel een volledige constructieve definitie geven, aangezien we een fiks aantal stellingen over (kortste) paden zullen bewijzen:

Paden

```
path:*s
∅: path
el(v:vertex,p:path): path

∀p:path.p=∅ ∨ ∃v:vertex,p2:path.p = el(v,p2)
∀v:vertex,p:path.∅ ≠ el(v,p)

extendpathwithvertex(p:path,v:vertex):path
      (notatie p→v)
∀v:vertex.∅→v = el(v,∅)
∀v1,v2:vertex,p:path.el(v1,p)→v2 = el(v1,p→v2)

length(p:path):nat+
length(∅)=0
∀v:vertex.length(el(v,∅))=0
∀v1,v2:vertex,p:path.
      length(el(v1,el(v2,p))=length(el(v2,p))+w(v1,v2)
```

We definiëren ook een propositie voor het begrip *u* ~>^{p} *v*, wat wil zeggen "*p* is een pad van *u* naar *v*". Merk op dat een pad niet per se uit aangrenzende vertexes hoeft te bestaan.

```
isconnectedbypath(u,v:vertex,p:path):*p
      (notatie: u ~>^p v)

first(p:path):vertex
last(p:path):vertex

∀v:vertex,p:path.first(el(v,p))=v
∀v:vertex.first(∅) ≠ v

∀v1,v2:vertex,p:path.last(el(v1,el(v2,p))=last(el(v2,p))
∀v:vertex.last(el(v,∅))=v
∀v:vertex.last(∅) ≠ v

∀v1,v2:vertex.¬(v1 ~>^∅ v2)
∀v1,v2,v:vertex,p:path.v1 ~>^el(v,p) v2 ≡
      first(el(v,p))=v1 ∧ last(el(v,p))=v2
```

Verder definiëren we voor het opdelen van paden:

```
take(p:path,n:nat): path
drop(p:path,n:nat): path
in_path(v:vertex,p:path): *p
      (notatie v in p)

∀p:path.take(p,0) = ∅
∀n:nat.take(∅,n) = ∅
∀v:vertex,p:path,n:nat.take(el(v,p),s(n)) = el(v,take(p,n))

∀p:path.drop(p,0) = p
∀n:nat.drop(∅,n) = ∅
∀v:vertex,p:path,n:nat.drop(el(v,p),s(n)) = drop(p,n)

∀v:vertex. ¬(v in ∅)
∀v1,v2:vertex,p:path.v1 in el(v2,p) ≡ (v1 = v2 ∨ v1 ∈ p)
```

B.1.3 Begrippen

Bereikbaarheid

Bereikbaarheid kan recursief worden gedefinieerd:

```
isreachable(v1,v2:vertex):*p
∀v:vertex.isreachable(v,v)
∀v1,v2:vertex.v1 ≠ v2 ⇒
      isreachable(v1,v2)
      ≡
      edge(v1,v2) ∈ E
      ∨
      ∃x:vertex.edge(v1,x) ∈ E ∧ isreachable(x,v2)
```

Afstand

De postconditie van het algoritme is dat de verschillende waardes van het array *d* overeenkomen met de kortste afstand tussen vertexes. Om te weten of ons programma hieraan voldoet zullen we dit begrip "kortste afstand" moeten specificeren:

$$\delta(i,j) = \begin{Bmatrix} \min p : i \sim\!> {}^{p}\ j : w(p) \\ \infty \end{Bmatrix} \begin{Bmatrix} als\ j\ bereikbaar\ is\ uit\ i \\ anders \end{Bmatrix}$$

```
delta(v1,v2:vertex):nat+
```

We gebruiken deze definitie niet rechtstreeks, maar in plaats ervan gebruiken we de volgende recursieve eigenschap van korte paden. We bewijzen deze niet in Cocktail.

```
∀v:vertex.delta(v,v)=0

∀v1,v2:vertex.v1≠v2 ⇒
      delta(v1,v2)
      =
      min i: i ∈ predecessors(v2): delta(v1,i) + w(i,v2)
```

Deze minimumquantor bestaat niet rechtstreeks in Cocktail en we definiëren hem volgens de volgende regel. Het minimum over een domein *D* volgens een functie *f* noemen we *fmini(D)*:

```
fmini(D) ≡ min i:i ∈ D: f(i)
```

Dan geldt voor *fmini(D)*:

```
fmini(∅)=∞
∀i:i ∈ D.fmini(D) = min(f(i),fmini(D--{i}))
```

Als we deze regel loslaten op de definitie van *delta* en substitueren als volgt:

```
delta(v1,v2) = fmini(predecessors(v2))
f(i) = delta(v1,i)+w(i,v2)

fmini(∅) = ∞
∀i: i ∈ D.fmini(D) = min(f(i),fmini(D--{i}))
```

Merk op dat de functie *f* de parameters *v1* en *v2* gebruikt en we moeten deze dus doorgeven:

```
f(v1,v2,i) = delta(v1,i)+w(i,v2)
fmini(v1,v2,∅) = ∞
∀i: i ∈ D.fmini(v1,v2,D) = min(f(v1,v2,i),fmini(v1,v2,D--{i}))
```

De uiteindelijke cocktailversie ziet er als volgt uit:

```
predecessors(v:vertex): set
∀v1,v2:vertex.edge(v1,v2) ∈ E ≡ v1 ∈ predecessors(v2)

delta(v,v)=0
∀v1,v2:vertex.v1 ≠ v2 ⇒
     delta(v1,v2)=fmini(v1,v2,predecessors(v2))

fmini(v1,v2:vertex, D:set):nat+

∀v1,v2:vertex.fmini(v1,v2,∅)=∞
∀v1,v2,i:vertex.∀D:set.i ∈ D ⇒
     fmini(v1,v2,D) =
     min(delta(v1,i)+w(i,v2), fmini(v1,v2,S--{i}))
```

Deze uiteindelijke vorm is bruikbaar voor het bewijzen van stellingen met behulp van inductie naar de lengte van de set *S*.

Kortste Paden

Een kortste pad is een pad met als lengte de kortste afstand tussen begin en eindvertex:

```
isshortestpath(p:path):*p
isshortestpath(∅)
∀v1,v2:vertex,p:path.v1 ~>p v2 ⇒
     isshortestpath(p) ≡ length(p)=delta(v1,v2)
```

Uit de (originele) definitie van delta volgen nog twee belangrijke eigenschappen, die we als axioma opnemen:

```
∀v1,v2:vertex.∃p:path.v1 ~>p v2 ∧ isshortestpath(p)
∀v1,v2:vertex,p:path.v1 ~>p v2 ⇒ delta(v1,v2) ≤
pathlength(p)
```

B.1.4 Programma Gerelateerde Begrippen

Ten slotte nog enige specifieke gegevens voor ons programma. Zo zijn er de sets *V* en *E*, een aantal gegevens over de graaf ($H_{0..2}$) en dienen onze invarianten te worden gespecificeerd:

Gegevens:

Om onnodig herhalen in de proposities te voorkomen worden de gegevens van het Dijkstra algoritme rechtstreeks gedefinieerd (het begrip gewicht is al eerder genoemd als onderdeel van padlengtes).

```
V: set
E: edge_set
w(v1,v2:vertex): nat+
start: vertex
```

We weten:

H_0: { $E \subseteq V \times V$ }
H_1: { $\forall e: e \in E: w(e) \geq 0$ }
H_2: { $s \in V$ }

In cocktail (H_1 geldt automatisch door de typerestrictie op *w*)

```
start ∈ V
∀v1,v2:vertex.(edge(v1,v2) ∈ E) ⇒ (v1 ∈ V ∧ v2 ∈ V)
```

Buren

Het begrip buren van (een vertex of een set van vertices) wordt gebruikt in invariant P_3, maar is door zijn veelvuldig gebruik in bewijzen een eigen paragraaf waard:

Voor de buren van een enkele vertex:

```
neighbours(v:vertex): set
∀v1,v2:vertex.(v2 ∈ neighbours(v1)) ≡ v1≠v2 ∧ edge(v1,v2) ∈
E
```

En voor de buren van een set van vertexes:

```
set_neighbours(S:set): set
∀v2:vertex.∀S:set.(v2 ∈ set_neighbours(S))
      ≡
      (∪v: v ∈ S: neighbours(v))\S
```

Deze verenigingsquantor is niet aanwezig in Cocktail. We vertalen hem de volgende propositie:

```
∀v2:vertex.∀S:set.(v2 ∈ set_neighbours(S))
≡
(v2 ∉ S ∧ (∃v1:vertex.v1 ∈ S ∧ edge(v1,v2) ∈ E))
```

Hulpfuncties

Sommige delen van het programma zijn niet verder geïmplementeerd en bewezen. Het gaat voornamelijk om het vinden van een minimaal element uit een array (implementeerbaar middels een priority queue) en het verlagen van de geschatte afstand voor alle buren van een vertex (implementeerbaar met een eenvoudige iteratie).

Voor deze functies wordt de specificatie direct gebruikt. Voor *getminimalelem* is dat als volgt:

```
getminimalelem(Q:set,d:array): vertex
∀Q:set.∀d:array.(Q≠empty) ⇒
     getminimalelem(Q,d) ∈ Q
     ∧ ∀i:vertex.(getminimalelem(Q,d) ≤ d(i))
```

Voor het verlagen van de tot nu toe berekende afstand tot buren geldt:

```
lowerneighbours(u:vertex,d:array) =
|[
      foreach v ∈ u.adj →
            d(v) := min(d(v), d(u) + w(u,v)
      rof
]|
```

Wat overeenkomt met de volgende specificatie:

```
lowerneighbours(u:vertex,d:array): array

∀d,d2:array.∀u:vertex.(d2=lowerneighbours(u,d)
     ≡ (( ∀i:vertex.i ∈ neighbours(u) ⇒
          (d2(i) = min(d(i), d(u)+w(u,i))))
       ∧ (∀i:vertex.i ∉ neighbours(u) ⇒
          (d2(i) = d(i)))
)
```

B.1.5 Invarianten

De invarianten zullen moeten worden gedefinieerd in termen van de eerder gegeven begrippen en in een vorm die het mogelijk maakt om hun invariantie later te bewijzen:

P_0

P_0: { $S \subseteq V \wedge \forall v: v \in S: d(v) = \delta(s,v)$ }

```
P0a(S:set): *p
∀S:set.P0a(S) ≡ S ⊆ V

P0b(S:set,d:array): *p
∀S:set.∀d:array.P0b(S,d)
     ≡ (∀v:vertex.v ∈ S ⇒ (d(v) = delta(start,v)))
```

P_1

P_1: { $Q = V \setminus S$ }

```
P1(S,Q:set): *p
∀S,Q:set.P1(S,Q) ≡ Q=V\S
```

Merk op dat er geen P_2 bestaat, aangezien deze tijdens de originele papieren afleiding overbodig werd gemaakt door P_3.

P_3

P_3: { $\forall v: v \in S.adj:$
$d(v) = (\min x: x \in S \wedge x \rightarrow v: (d(x) + w(x,v))$
}

P_3 bevat wederom een minimumquantor die moet worden omgeschreven. We gebruiken dezelfde methode als we bij de definitie van delta gebruikten:

```
min i:i ∈ D: f(i)
≡
fmini(D)

fmini(∅)=∞
∀i:element.∀D:domain.i ∈ D ⇒ fmini(D) = min(f(i),fmini(D--
{i}))
```

Wanneer we een minimumquantor volgens deze regel definiëren, kunnen we de volgende stelling afleiden:

```
m = fmini(D)
≡
(∃i: i ∈ D: m = f(i)) ∧
(∀i: i ∈ D: m ≤ f(i))
```

Deze tweede schrijfwijze is handzamer omdat het de noodzaak tot inductie op de setgrootte bij veel bewijzen onnodig maakt.

P_3 wordt:

```
P3(S:set,d:array): *p
minimaldistance(S:set, v:vertex, d:array): nat+
∀S:set.∀d:array.P3(S,d)
     ≡ (∀v:vertex.v ∈ set_neighbours(S) ⇒
        (d(v) = minimaldistance(S,v,d)))

∀m:nat+.∀S:set.∀v:vertex.∀d:array.m=minimaldistance(S,v,d)
     ≡ (
        (∃x:vertex. x ∈ S ∧ edge(x,v) ∈ E ∧
        (m=d(x)+w(x,v)))
       ∧
        (∀x:vertex. (x ∈ S ∧ edge(x,v) ∈ E) ⇒
        (m ≤ d(x)+w(x,v)))
        )
```

P_4

P_4:{ ∀v: v ∈ (Q \ S.adj): d(v) = ∞ }

```
P4(S,Q:set,d:array): *p
∀S,Q:set.∀d:array.P4(S,Q,d)
     ≡ (∀v:vertex.v ∈ Q\(set_neighbours(S)) ⇒ (d(v)= ∞))
```

P_5

En ten slotte dient zich een extra invariant aan, in de vorm van P_5:

```
P5(S): *p
∀S:set.P5(S) ≡ start ∈ S
```

Deze invariant is noodzakelijk omdat de begintoestand van een loop niet als gegeven wordt meegenomen bij de bewijzen van invariantie. In de papieren versie gebruiken we deze kennis rechtstreeks, terwijl hij hier expliciet dient te worden meegegeven.

B.1.6 Het Programma

```
var d: array
|[ var S: set
  |[ var Q: set
    { True }
    …
    { Invariant(empty++{start}, V--{start}, init_d(d)) }
    Q := V -- {start};
    S := empty ++ {start};
    d := init_d(d);

    while not(isemptyb(Q)) do
      |[ var u: nat;
        { Invariant(S,Q,d) ∧ (not(isemptyB(Q))=true) }
        …
        { Invariant(S++{ getminimalelem(Q,d)},
                    Q--{ getminimalelem(Q,d)},
                    lowerneighbours(d, getminimalelem(Q,d)) }
        u := getminimalelem(Q,d)
        Q := Q--{u};
        S := S++{u};
        d := lowerneighbours(d,u);
      ]|
    od;
    { Invariant(S,Q,d) ∧ (not(isemptyb(Q))=false) }
    …
    { Post(d) }
  ]|
]|
```

De functie *init_d* die gebruikt wordt bij de initialisatie heeft als specificatie dat het array *d* wordt geinitialiseerd zoals we dat eerder hebben besproken: de *d* van *start* is nul, voor alle buren is de *d* gelijk aan de lengte van de verbindende edge en voor de overige vertices is *d* oneindig. Net als bij de *lowerneighbours* functie geldt dat de uitwerking van dit stuk programma als triviaal mag worden beschouwd.

De propositie *Invariant(S,Q,d)* die in de annotatie wordt gebruikt is een verkorte schrijfwijze voor de conjunctie van de invarianten $P_{0..5}$.

B.1.7 Afgeleide stellingen

Het is zaak om de bewijsverplichtingen die uit dit programma voortkomen rond te krijgen. De eerder genoemde definities en stellingen zijn voldoende om dit te bereiken, maar er zijn veel stappen nodig om tot dit resultaat te komen.

Intialisatie en finalisatie volgen uit de context, het meeste bewijswerk zit in de invariantiebewijzen. We kunnen elke afzonderlijke invariant bewijzen als volgt:

$$\forall S,Q{:}set.\forall d{:}array.Invariant(S,Q,d) \wedge (not(isemptyb(Q))=true) \Rightarrow$$

$$P_n(S{+}{+}\{getminimalelem(Q,d)\},\ Q{-}{-}\{getminimalelem(Q,d)\},\ lowerneighbours(d,getminimalelem(Q,d)))$$

De resulterende annotatie bestaat uit vele honderden regels aan bewijs. We stippen hieronder kort een paar substellingen aan die we gebruiken als onderdeel van de bewijzen:

Kortste subpaden:

```
∀n:nat,p:path.isshortestpath(p) ⇒ isshortestpath(drop(p,n))
∀n:nat,p:path.isshortestpath(p) ⇒ isshortestpath(take(p,n))
```

Driehoeksongelijkheid:

```
∀v1,v2:vertex.delta(start,v2) ≤ delta(start,v1) + w(v1,v2)
```

Pad-uitbreiding:

```
∀v1,v2,v3:vertex,p:path.
     v1 ~>^p v2 ∧ isshortestpath(p) ∧ isshortestpath(p→v3)
     ⇒
     delta(v1,v3) = delta(v1,v2)+w(v2,v3)
```

Relatie tussen delta en bereikbaarheid:

```
∀v1,v2:vertex.¬isreachable(v1,v2) ≡ (delta(v1,v2) = ∞)
```

Bestaan van een korter pad:

```
∀p:path,v1,v2:vertex.
     v1 ~>^p v2 ∧ v1 ≠ v2 ∧ pathlength(p) ≠ delta(v1,v2)
     ⇒
     ∃x:vertex.delta(v1,x)+w(x,v2) < pathlength(p)
```

Onbereikbaarheid:

```
∀st,v1,v2:vertex.¬isreachable(st,v2) ∧ edge(v1,v2) ∈ E ⇒
     ¬isreachable(st,v1)
```

Eigenschappen van subpaden

```
∀start,u,v:vertex.∀p:path.
     start ~>^p u ∧ isshortestpath(p) ∧   v in p
     ⇒ delta(start,v) ≤ delta(start,u)

∀start,u,v:vertex.∀p:path.
     start ~>^p u ∧ isshortestpath(p) ∧   v in p
     ⇒ isreachable(start,u)
```

Grenzen oversteken:

Een uiterst belangrijke stelling is de volgende, die gebruikt wordt bij de volgende redenering: als er een pad begint bij een vertex in de set *S* en eindigt bij een vertex die niet tot *S* behoort, dan bestaat er als deel van het pad een edge tussen *S* en "niet *S*":

```
∀S:set.∀u:vertex.∀p:path.
      start ~>° u ∧ isshortestpath(p) ∧
      start ∈ S ∧ u ∉ S ∧ isreachable(start,u)
      ⇒ ∃a,b:vertex.(
      a ∈ S ∧ b ∉ S ∧
      a in p ∧ b in p ∧
      edge(a,b) ∈ E ∧
      (delta(start,b)=delta(start,a)+w(a,b)))
```

B.2 Stepwise Refinement (Perfect Developer)

We doen een poging om de stappen uit hoofdstuk 4 te volgen in Perfect Developer.

Perfect Developer biedt ondersteuning voor deze methode op de volgende manieren:

Ten eerste kunnen we een verfijningstap zetten tussen de interface van een algoritme en de implementatie ervan.

Ten tweede biedt PD op het implementatieniveau de *change satisfy* constructie die ons instaat stelt om een deel code slechts te specificeren zonder het te implementeren en de *?* operator stelt ons instaat om (delen van) specificaties ongedefinieerd te laten.

Al vroeg zal echter blijken dat we op teveel problemen stuiten om deze tak voort te zetten, de complexiteit van het probleem speelt ons (en Perfect Developer) parten.

B.2.1 Implementatie in Perfect Developer

Dijkstra's algoritme maakt gebruik van een aantal termen zoals grafen met edges en vertices, (kortste) paden en oneindigheid. Zelfs als we kijken naar onze allereerste versie van het algoritme worden deze termen gebruikt:

```
const
      V: set of vertex;
      E: set of vertex × vertex;
      w: vertex × vertex → int;
      s: vertex;
known
      H0: { E ⊆ V × V }
      H1: { ∀e: e ∈ E: w(e) ≥ 0 }
      H2: { s ∈ V }
var
      d: vertex → int;

"vul array d";
{ ∀i: i ∈ V: d(i) = δ(s,i) }
```

Merk het gebruik van delta op in de postconditie.

We dienen dus al in een vroeg stadium een definitie te vormen voor al deze begrippen, terwijl er echter een grote keuzeruimte is aan mogelijke oplossingen en we niet weten welke definities het meest geschikt zullen zijn (denk ook aan onze herformulering van het begrip gesorteerdheid die bij de binary search tot een oplossing leidde).

We geven een paar voorbeelden van te maken keuzes en gaan dan specifiek in op één hiervan.

Paden als klasse

Het begrip pad leent zich goed voor een omschrijving als een klasse. Een probleem hierbij, echter, is dat paden intrinsiek verbonden zijn aan grafen. In onze representatie is een graaf een generieke klasse waarbij het type van de vertices een parameter is.

Een pad klasse hoort hetzelfde type vertices te bevatten als de graaf waar hij bij hoort. Evenzo dienen de vertices in een pad ook voor te komen in de bijbehorende graaf. Deze eisen creëren een eigen set met invarianten voor de nieuwe klasse die het bewijzen potentieel kunnen bemoeilijken.

Een alternatief hiervoor is om de pad-klasse los te laten en de verschillende gewenste eigenschappen te vangen in predicaten middels functies van de graaf.

```
1.  class Path of X
2.  require X has operator =(arg) end
3.  ^=
4.  abstract
5.      var
6.          G: Graph of X,
7.          V: seq of X;
8.
9.      invariant
10.         forall v::V :- v in G.V;
11.
12. interface
13.     ghost operator =(arg);
14.
15.     build{g: Graph of X, v: seq of X}
16.     pre
17.         forall i::v :- i in g.V
18.     post
19.         G != g,
20.         V != v;
21. end;
```

Een voordeel van deze methode is dat we een aantal operaties in deze klasse kunnen vangen zoals de conversie naar een sequence van edges, het berekenen van de lengte en cycli.

```
1.      function getEdges: seq of pair of (X,X)
2.      ^=
3.          for i::0..(#V-2) yield pair of (X,X) {V[i], V[>i]};
4.
5.      function getLength: natInf
6.      ^=
7.          + over (for i::getEdges yield G.getWeight(i));
8.
9.      function hasCycles: bool
10.     ^=
11.         forall v::V :- v#V = 1;
```

Wat echter wel het geval is, is dat de graaf klasse methoden biedt om de sets *V* en *E* uit te breiden. Wanneer de graaf *G* wordt veranderd, kan ons pad ongeldig worden. We kunnen ook deze relatie weglaten en functies als getLength als onderdeel van de graph klasse definiëren:

```
final class Path of X
require X has operator =(arg) end
^=
abstract
    var
        V: seq of X;

interface
    ghost operator =(arg);

    function V;

    build{v: seq of X}
    post
        V != v;

    function getEdges: seq of pair of (X,X)
    ^=
        for i::0..(#V-2) yield pair of (X,X) {V[i], V[>i]};

    function hasCycles: bool
    ^=
        forall v::V :- v#V = 1;
end;
```

en in Graph:

```
    function pathWeight(p: Path of X): natInf
    pre
        forall v::p.V :- v in V
    ^=
        + over (for i::p.getEdges yield getWeight(i));
```

Paden als predikaat

Een alternatief is om paden te zien als een eigenschap van lijsten. Een lijst van vertices is een pad wanneer alle vertices verbonden zijn door edges. Deze functie kan als onderdeel van de graaf klasse worden gedefinieerd:

```
function isConnected(a,b: X): bool
^=
    exists i::E :- i.x = a & i.y = b;

property(h,v: X)
pre
    pair of (X,X) {h,v} in E
assert
    isConnected(h,v);

function isPath(p: seq of X): bool
^=
    (#p = 1) |
    (forall i::1..<#p :- isConnected(p[i-1],p[i]));
```

Gewicht & Afstand

Hierboven is al ingegaan op het gewicht van een pad. Erin wordt een speciale waarde *natInf* gebruikt in plaats van *nat*. De uitleg hiervoor is als volgt:

Een problematisch verschil tussen de papieren definitie en die uit PD is dat we op papier het type van de natuurlijke getallen hebben uitgebreid met de waarde ∞, zonder dit expliciet te definiëren. We zullen dit in PD wel moeten doen.

Aangenomen dat we een klasse *natInf* hebben, bestaande uit de natuurlijke getallen met oneindig, is het gewicht van een edge als volgt te implementeren:

```
1. function getWeight(p: pair of (X,X)): natInf
2. pre
3.     p.x in V,
4.     p.y in V
5. ^=
6.     (
7.         [p in E]: natInf{W[p] as nat||Infinite},
8.         []: natInf{inf@Infinite as nat||Infinite}
9.     );
```

Kortste Padsafstand

De kortste pad afstand *δ(i,j)* tussen twee vertices *i* en *j* is het minimale gewicht van alle paden tussen *i* en *j*. Indien er geen pad bestaat is deze afstand ∞.

$$\delta(i,j) = \begin{cases} \min p : i \sim\!> {}^{p}\ j : w(p) \\ \infty \end{cases} \begin{cases} als\ j\ bereikbaar\ is\ uit\ i \\ anders \end{cases}$$

De begrippen kortste pad en kortste padsafstand zijn sleutelbegrippen voor het algoritme. Ze worden gebruikt in invarianten en in de postconditie.

In conventionele programmeertalen hoeft dit begrip niet gedefinieerd te worden, aangezien hij in de daadwerkelijke code niet voorkomt. Perfect Developer, echter, combineert klassieke code met vastgelegde asserties, invarianten en pre- en postcondities en heeft dus wel een definitie van δ nodig.

Een aantal mogelijke tactieken om dit te doen:

Rechtstreekse Definitie

Met het gebruik van quantoren kan de definitie van een kortste padslengte rechtstreeks worden ingevoerd. Het resultaat is afhankelijk van de gekozen definitie van paden, maar ziet er ongeveer als volgt uit:

```
1.  function shortestPathLength(start, target: X): natInf
2.  pre
3.      start in V,
4.      target in V
5.  ^=
6.      (
7.          [start=target]: 0,
8.          [~isReachable(start,target)]:
                natInf{inf@Infinite as nat||Infinite},
9.          []: ( let paths ^=
                those p:Path of X :-
                      p.V.head = start & p.V.last = target;
10.
11.             any q:: paths :- forall qq:: paths :-
                      pathWeight(q) <= pathWeight(qq)
12.           )
13.     );
```

Deze constructie maakt gebruik van zogenaamde keuze expressies *those* en *any*. De *those* operator zoekt een subset over een oneindig aantal instanties van de klasse Path of X, waardoor deze functie onuitvoerbaar wordt.

Een remedie zou zijn om middels een functie alle mogelijke paden van de graaf uit te rekenen. Deze set is eindig wanneer de graaf non-cyclisch is en kan dus gebruikt worden in deze definitie. Het moge echter duidelijk zijn dat dit praktisch niet uitvoerbaar is en we zullen deze optie niet verder uitwerken.

Recursieve Definitie

Een wat elegantere manier om de kortste padsafstand te definiëren is volgens de volgende recursieve eigenschap:

$$\delta(a,b) = 0 \text{ als } a = b$$
$$= \min i: i \in b.pred: \delta(a,i) + w(i,b)$$

Deze eigenschap geldt alleen wanneer *b* bereikbaar is vanuit *a*. Dit zorgt ervoor dat de definitie niet helemaal perfect is:

```
function isReachable(s, t: X): bool
//  nondeterministisch
pre
    s in V,
    t in V
^=
    ( [exists i::E :- i.x = s & i.y = t]: true,
      []: exists i::getNeighbours(s) :- isReachable(i,t)
    );

axiom (a,b: X)
pre
    a in V,
    b in V,
    isReachable(a,b)
assert
    #getPredecessors(b) > 0;

function shortestPathLength2(s, l: X): nat
pre
    s in V,
    l in V,
    isReachable(s,l)
^=
    (
        (
            for i:: getPredecessors(l) yield
              W[pair of (X,X) {i,l}] +
              shortestPathLength(s,i)
        )
    ).min;
```

Merk op dat beide functies *isReachable* en *shortestPathLength* non-deterministisch zijn. Wanneer de graaf geen cycli bevat (of in het geval van de kortste padlengte geen cyclische kortste paden) zijn ze dit niet, maar het encoderen van deze eigenschap zorgt weer voor eigen moeilijkheden.

Een definitie naar het gedrag

Een derde mogelijkheid is om het kortste pad niet te rechtstreeks te definiëren, maar dit te doen via zijn eigenschappen - stellingen zoals de driehoeksongelijkheid worden als axioma's opgenomen voor een eigenschap "kortste padlengte" die elke vertex heeft maar die nergens concreet wordt gedefinieerd.

```
ghost function Distance(a,b:X): natInf
^= ?;

axiom driehoek(a,b,c:X) ^=
pre
    a,b,c in V
^=
    Distance(a,b) <= Distance(a,c) + W(c,b);
```

Naar gelang we ze nodig hebben kunnen we nieuwe axioma's toevoegen aan deze lijst.

'Oneindig' wegwerken in de code

Zoals gezegd maken de definities vaak gebruik van oneindig als waarde voor natuurlijke getallen, waardoor we deze extra waarde in Perfect Developer zullen moeten implementeren.

Hiernaast wordt oneindig niet alleen in het specificatiedeel gebruikt, maar ook in de concrete code. We zoeken dus een uitvoerbare manier om met oneindig als waarde om te gaan.

Een aantal mogelijkheden:

We kunnen een aantal extra stappen toevoegen aan onze papieren uitwerkingen om de waarde oneindig uit de code te vervangen door operaties op een lager niveau. Dat dit mogelijk is blijkt uit het feit dat conventionele talen die ook geen beschikking hebben over oneindig als waarde toch werkende implementaties van de algoritmes kennen.

In PD geeft dit ons wat meer vrijheid bij de definitie van oneindig, aangezien we geen uitvoerbare code hoeven te produceren. We dienen wel nog steeds een definitie op te geven waar de verifier mee kan rekenen.

'Oneindig' wegwerken in specificatie

We kunnen ook onze specificatie zo aanpassen dat de waarde nergens meer voorkomt, zowel in de implementatie als de specificatie. Dit zorgt wel voor een potentieel stukken complexere uitwerking van de algoritmes, met fors gebruik van gevalsonderscheid.

'Oneindig' implementeren in Perfect Developer

Perfect Developer biedt de mogelijkheid om een eigen type te creëren, met standaardoperaties en rekenregels. We proberen om een type te *natInf* produceren dat natuurlijke getallen zowel als de waarde oneindig accepteert.

De specificatie van *natInf* is als volgt:

```
∀i,j: i,j ∈ nat:

i + j = i + j
∞ + i = ∞
∞ + ∞ = ∞
∞ – i = ∞
```

Het resultaat van ∞ - ∞ is ongedefinieerd, waardoor de - operatie de preconditie krijgt dat het tweede lid niet oneindig is.

B.2.2 Voorbeeld: Nat+

Ter illustratie bespreken we deze laatste mogelijkheid in detail.

We introduceren een klasse voor het bijhouden van natuurlijke getallen en oneindig, als volgt:

```
1.  axiom Tmp1  // fixes omission in proof rules
2.      assert forall i: natInf :-
            forall j: natInf :- (i=j | i<j) ==> i <= j;
3.
4.  class Infinite ^= enum inf end;
5.
6.  final class natInf ^=
7.  abstract
8.      var
9.          val: nat||Infinite
10. interface
11.     function getValue: nat||Infinite ^= val;
12.
13.     function isInfinite: bool
14.     ^=
15.         val within Infinite;
16.
17.     function natVal: nat
18.     pre
19.         val within nat
20.     ^=
21.         val is nat;
22.
23.     build{n: nat||Infinite}
24.     post
25.         val != n;
26.
27.     total operator ~~(other)
28.     ^=
29.         (
30.             [isInfinite & other.isInfinite]: same@rank,
31.             [isInfinite & ~other.isInfinite]: above@rank,
32.             [~isInfinite & other.isInfinite]: below@rank,
33.             [~isInfinite & ~other.isInfinite]:
                    (natVal) ~~ (other.natVal)
34.         );
35.
36.
37.     operator -(other: natInf): natInf
38.     pre
39.         ~other.isInfinite,
40.         (~isInfinite & ~other.isInfinite) ==>
    (other.natVal) <= (natVal)
41.     ^=
42.         (
43.             [isInfinite]:
                    natInf{inf@Infinite as nat||Infinite},
44.             []: natInf{((natVal) - (other.natVal))}
45.         );
46.
47.     operator +(other: natInf): natInf
48.         associative commutative identity natInf{0}
49.     ^=
50.         (
51.             [(getValue within Infinite) |
                 (other.getValue within Infinite)]:
                     natInf{inf@Infinite as nat||Infinite},
```

```
52.             [(getValue within nat) &
                 (other.getValue within nat)]:
                     natInf{((natVal) + (other.natVal))}
53.         );
```

Deze klasse implementeert ons nieuwe type. Een aantal eigenschappen zoals totaalheid van de vergelijkingsoperator (~~) en associativiteit, commutativiteit en identity 0 van de + operator, zijn in de definitie hierboven opgegeven en worden bewezen door de verifier.

We specificeren ook een aantal kenmerken van ons nieuwe type en laten deze doorrekenen door Perfect Developer Het axioma tmp1 helpt bij het bewijs van de eerste eigenschap.

```
1.      property (a: natInf)
2.      assert
3.          a <= natInf{inf@Infinite as nat||Infinite}
4.      proof
5.      assert
6.          (a.isInfinite ==>
                a = natInf{inf@Infinite as nat||Infinite}),
7.          (~a.isInfinite ==>
                a < natInf{inf@Infinite as nat||Infinite})
8.      end;
9.
10.     property (a,b: natInf)
11.     assert
12.         a + b = b + a;
13.
14.     property (a,b: natInf)
15.     pre
16.         a.isInfinite,
17.         b.isInfinite
18.     assert
19.         a = b;
20.
21.     property (a,b: natInf)
22.     pre
23.         a.isInfinite
24.     assert
25.         a + b = natInf{inf@Infinite as nat||Infinite};
26.
27.     property(a,b: natInf)
28.     pre
29.         ~a.isInfinite,
30.         ~b.isInfinite
31.     assert
32.         ((a + b).natVal) = (a.natVal) + (b.natVal);
33.
34.     property(a: natInf)
35.     pre
36.         ~a.isInfinite
37.     assert
38.         natInf{inf@Infinite} - a =
                natInf{inf@Infinite as nat||Infinite};
39. end;
```

Opmerkingen

Het genereren van de klasse hierboven is geen eenvoudige taak. De tegengekomen problemen in het kort zijn:

- We hebben een axioma nodig voor een missende regel:

  $a \leq b \Rightarrow a < b \vee a = b$

Het achterhalen dat deze regel niet voorradig is vereist het stapsgewijs debuggen zoals we dat zagen bij de binary search.

- Associativiteit van de '+' operator werkt alleen dankzij de exacte definitie van de constructor.

De constructor neemt een parameter van het type nat||Infinite. Oorspronkelijk gebruikten we constructoroverloading en genereerden we dit gecombineerde type zelf:

```
build{v: nat}
post
    val != (v as nat||Infinite);

build{i: Infinite}
post
    val != (i as nat||Infinite);
```

- De klasse dient "final" te zijn. Wanneer inheritance mogelijk is krijgt de prover de bewijzen niet rond.

Testcase

Het is belangrijk ons ervan te overtuigen dat de prover met ons nieuwe type om kan gaan. Als dit zo is kunnen we het gebruiken in onze algoritmes. Om die reden volgt hier een korte case study.

We definiëren het minimum van een reeks *R* als:

$$infmin(r) = \infty \quad als \ r = \langle\rangle$$
$$= min\ i: i \in r: i$$

Voor het berekenen van dit minimum gebruiken we het volgende eenvoudige algoritme:

```
const
      R: list of nat;
var
      S: list of nat;
      Q: list of nat;
      i: natInf;

i := ∞;
S := A;
Q := <>;

do Q ≠ A → { Variante Functie: #S }
      invariant
            P0: { S ⊆ A ∧ Q ⊆ A }
            P1: { Q + S = A }
            P2: { i = infmin(Q) }

      var u: nat;
      u := S.head;

      Q := Q + u,
      S := S.tail,
      i := min(i, natInf(u))
od;
{ P2 ∧ (Q = A) ⇒ i = infmin(A) }
```

In Perfect Developer ziet dit er als volgt uit:

```
1.  function findmin(a: seq of natInf): natInf
2.  ^=
3.      (
4.          [#a=0]: natInf{inf@Infinite as nat||Infinite},
5.          [#a>0]: a.min
6.      )
7.  via
8.      var i: natInf;
9.      var
10.         s,q: seq of natInf;
11.
12.     i != natInf{inf@Infinite as nat||Infinite};
13.     s != a, q != seq of natInf {};
14.
15.     loop
16.     change
17.         s,q,i
18.     keep
19.         q' <<= a,
20.         s' <<= a,
21.         q'++s' = a,
22.         i' = findmin(q')
23.     until
24.         #q'=#a
25.     decrease
26.         #s';
27.
28.         var j: natInf;
29.         j != s.head;
30.
31.         s != s.tail,
32.         q != q.append(j),
33.         i != min(i,j);
34.     end;
35.
36.     value(i);
37. end;
```

De prover heeft in deze versie echter wat problemen met het bewijzen van invariantie. Met name de belangrijkste invariant *i = findmin(q)* blijkt moeilijk te zijn.
Na deze kwestie voor te leggen aan de PD developers, werd de volgende verbetering voorgedragen, die echter nog steeds de hoofdinvariant onbewijsbaar laat:

```
1.  function findmin2(a: seq of natInf): natInf
2.   ^=
3.      (
4.          [a.empty]: natInf{inf@Infinite as nat||Infinite},
5.          []: a.min
6.      )
7.   via
8.      let
9.          infinite ^= natInf{inf@Infinite as nat||Infinite};
10.
11.     var i: natInf != infinite;
12.
13.     loop
14.         var index: (nat in 0..#a)! = 0;
15.     change
16.         i
17.     keep
18.         i' = findmin2(a.take(index'))
19.     until
20.         index'=#a
21.     decrease
22.         #a - index';
23.
24.         let i_old ^= i;
25.         let index_old ^= index;
26.
27.         assert            // onbewijsbaar
28.             min(i, a[index]) = a.take(index+1).min
29.
30.         i! = min(i, a[index]),
31.         index!+ 1;
32.
33.         assert
34.             i = a.take(index).min
35.         proof
36.             assert
37.                 min(i_old, a[index_old]) =
                        a.take(index_old+1).min,
38.                 i = min(i_old, a[index_old]),
39.                 index = index_old + 1
40.         end;
41.     end;
42.
43.     value(i);
44. end;
```

De dikgedrukte asserties zijn hier later bijgevoegd en het lukt PD om de invariantie te bewijzen met deze asserties. Het probleem verschuift echter en de eerste assertie blijkt onbewijsbaar voor de prover.

Op dit punt besluiten we deze tactiek na veel proberen naast ons neer te leggen.

B.3 Incremental Development (Perfect Developer)

B.3.1 Basis

We beginnen met een basisklasse die de gegevens van de graaf bijhoudt. We gebruiken een generieke parameter *X* voor het type van de vertex, zodat onze graaf klasse op vergelijkbare wijze gebruikt kan worden als lists of sets. Aangezien gewichten pas in de laatste stap worden gebruikt, maken we nog geen keuze voor de representatie hiervan.

```
1.  final class Graph of X
2.  require X has operator =(arg) end
3.  ^=
4.  abstract
5.      var
6.          V: set of X,
7.          E: set of pair of (X,X);
8.
9.      invariant
10.         forall i::E :- i.x in V & i.y in V;
11.
12. interface
13.     ghost operator =(arg);
14.
15.     function V;
16.     function E;
```

B.3.2 Kleuringsalgoritme

Vertaling

```
var
      B, G, W: set of Vertex;

B := ∅;
G := {start};
W := V \ {start};

do G ≠ ∅ → { VF: #V - #B, inv: P0..3 }
      { ∃h: Vertex: h ∈ G }
      let h: Vertex satisfy h ∈ G;

      B, G := B ∪ {h}, G \ {h};

      do forall x ∈ V →
            if
                  x ∈ W → W,G := W \ {x}, G ∪ {x}
            []    x ∈ G → skip
            []    x ∈ B → skip
            fi
      od
od
```

Outer Loop

We beginnen met een implementatie van de outer loop:

```
1.  function colorize(start:X): bool
2.  pre
3.      start in V
4.  satisfy
5.      result = true
6.  via
7.      var B, G, W: set of X;
8.
9.      B! = set of X {};
10.     G! = set of X {start};
11.     W! = V.remove(start);
12.
13.     loop
14.     change
15.         B,G
16.     keep
17.         B' ++ G' ++ W = V,
18.         B' ## G',
19.         B' ## W,
20.         W ## G'
21.     until
22.         G'.empty
23.     decrease
24.         #V - #B';
25.     //do
26.         var h: X! = any i::G:-true;
27.
28.         G! = G.remove(h),
29.         B! = B.append(h);
30.
31.         // forall-loop here
32.     end;
33.
34.     value(true)
35. end;
```

Inner Loop

We voegen nu de inner loop toe. In onze papieren versie gebruiken we een forall constructie. In Perfect Developer zal deze echter moeten worden uitgeschreven als een loop, voorzien van annotatie:

```
do forall x ∈ V →
      if
            x ∈ W → W,G := W \ {x}, G ∪ {x}
      []    x ∈ G → skip
      []    x ∈ B → skip
      fi
od
```

We noemen de set elementen waarover de forall loop itereert "tmp":

```
var tmp: set of Vertex;
tmp := V;

do tmp ≠ ∅ →
      let x: Vertex satisfy h ∈ tmp;
      tmp := tmp \ {x};

      if
            x ∈ W → W,G := W \ {x}, G ∪ {x}
      []    x ∈ G → skip
      []    x ∈ B → skip
      fi
od
```

Deze loop heeft als variante functie

#tmp

Begrensdheid van de variante functie bewijzen we met de invariant:

tmp ⊆ V

Vertaald wordt dit:

```
1.  var tmp: set of X! = those x::V :- x in V;
2.
3.  loop
4.  change
5.      tmp, G, W
6.  keep
7.      tmp' <<= tmp,
8.      B ++ G' ++ W' = V,
9.      B ## G',
10.     B ## W',
11.     W' ## G',
12. until
13.     tmp'.empty
14. decrease
15.     #tmp';
16. //do
17.     var x: X;
18.
19.     change x satisfy x' in tmp;
20.
21.     tmp! = tmp.remove(x);
22.
23.     if
24.         [x in W]: W! = W.remove(x);
25.                   G! = G.append(x);
26.         []
27.     fi;
28. end;
```

Merk op hoe we in de inner loop de invarianten van de outer loop herhalen.

Bij het toevoegen van deze inner loop aan de code van de outer loop dienen we de changelist van deze outer loop aan te passen (regel 14, 8.2.1.1). De set W wordt nu namelijk ook gewijzigd door de loop.

Het gehele programma ziet er als volgt uit:

```
function colorize(start:X): bool
pre
    start in V
satisfy
    result = true
via
    var B, G, W: set of X;

    B! = set of X {};
    G! = set of X {start};
    W! = V.remove(start);

    loop
    change
        B,G,W
    keep
        B' ++ G' ++ W' = V,
        B' ## G',
        B' ## W',
        W' ## G'
    until
        G'.empty
    decrease
        #V - #B';
    //do
        var h: X! = any i::G:-true;

        G! = G.remove(h),
        B! = B.append(h);

        var tmp: set of X! = those x::V :- x in V;

        loop
        change
            tmp, G, W
        keep
            tmp' <<= tmp,
            B ++ G' ++ W' = V,
            B ## G',
            B ## W',
            W' ## G',
        until
            tmp'.empty
        decrease
            #tmp';
        //do
            var x: X;

            change x satisfy x' in tmp;

            tmp! = tmp.remove(x);

            if
                [x in W]: W! = W.remove(x);
                          G! = G.append(x);
                []
```

```
57.             fi;
58.         end;
59.     end;
60.
61.     value(true)
62. end;
```

Verificatie

Perfect Developer ondervindt geen problemen bij het bewijzen van de invarianten en variante functie van de outer loop. Althans, totdat we de inner loop aan het programma toevoegen.

Hoewel de inner loop zelf door PD bewezen wordt, zorgt de toevoeging ervan dat er problemen ontstaan bij de begrensdheidseis van de outer loop:

> Failed to prove verification condition: Loop body establishes end condition or preserves validity of variant
> To prove: 0 ≤ (#**self**.V - (#$B_{loopend}$ **as int**))
> Reason: Exceeded time limit
> Could not prove:
> ~(0 = (#$G_{loopstart\_79,9}$ + #$W_{loopstart\_79,9}$))

Een mogelijkheid van verwarring is het feit dat we, aangezien de inner loop de set W verandert, deze set ook aan de changelist van de outer loop hebben moeten toevoegen.

Toch is de regel waar PD moeite mee heeft: namelijk dat #G + #W ≠ 0, niet moeilijk te bewijzen. Uit de guard van de loop volgt namelijk al dat #G > 1. We voegen een property toe om dit bewijs op weg te helpen:

```
1. property sum(a,b:int) // needed to prove (2)
2. pre
3.     a >= 0,
4.     b > 0
5. assert
6.     a+b > 0;
```

De property heeft inderdaad het gewenste resultaat - de variant van de outer loop kan zonder problemen worden bewezen.

Er treedt echter een bevreemdende bijwerking op: de variante functie van de inner loop zorgt plots voor problemen:

> Failed to prove verification condition: Loop initialisation establishes end condition or a valid variant
> To prove: 0 ≤ #tmp
> Reason: Exhausted rules
> Could not prove any of:
> 0 ≤ (1 + #$B_{loopstart\_79,9}$ + #$W_{loopstart\_79,9}$)
> 0 ≤ (#$B_{loopstart\_79,9}$ + #$G_{loopstart\_79,9}$ + #$W_{loopstart\_79,9}$)

Zou het zo zijn dat er geen regel is die zegt dat sets per definitie geen negatief aantal leden kunnen hebben We voegen een property toe:

```
1. property setsize(a:set of X)
2. assert
3.     #a >= 0;
```

Deze property genereert geen waarschuwingen. Hij helpt echter ook niet met de foutmelding. We voegen een assertie toe aan het begin van de loop:

```
1. var tmp: set of X! = those x::V :- x in V;
2.
3. assert #tmp >= 0;
4.
5. loop
6. // etc.
```

We zouden in dit geval de gewenste stelling via een omweg kunnen bewijzen. We weten namelijk dat de set van grijze vertices niet leeg is en dat deze een subset van V is. Daaruit volgt weer dat ook tmp (tmp bevat initieel alle vertices uit V) dit ook niet is.

In plaats hiervan zullen we van de gewraakte stelling simpelweg een aanname maken. We doen dit door hem te bewijzen vanuit het ongerijmde, als volgt:

```
1. assert #tmp >= 0
2. proof
3.     assert false
4. end;
```

Perfect Developer zal een error genereren in het bewijs: namelijk dat de stelling "false" niet bewezen kan worden. Daarna zal, met "false" als aanname, wel een bewijs worden gegenereerd voor de bijbehorende assertie #tmp ≥ 0.

Er bestaat in Perfect Developer geen directere methode om in de code aannames te plaatsen. Er bestaat wel een axiom keyword waarmee we stellingen, los van het algoritme, kunnen aannemen. We zagen echter al dat de stelling zelf in property setsize zonder problemen wordt bewezen, dus dat hier niet het probleem ligt.

Wat we missen is een manier om de bewijzer duidelijker aan te geven welke stellingen/eigenschappen van belang zijn bij een bewijs. Een constructie als de volgende is helaas onmogelijk in Perfect Developer:

```
1. assert #tmp >= 0
2. proof
3.     use property setsize(tmp)
4. end;
```

Met de laatste toevoeging worden er verder geen waarschuwingen gegenereerd.

Specificeren van Inner Loop

Een aanpassing die we kunnen maken is het vervangen van de inner loop door een specificatie met behulp van een change satisfy clausule.

Het voordeel van deze manier van werken is dat we bewijsverplichtingen duidelijk splitsen - door expliciet te vertellen wat onze inner loop dient te bewerkstelligen hoeft Perfect Developer deze informatie niet zelf af te leiden. Daarnaast is het ook voor onszelf makkelijker om te onderzoeken waar bewijzen eventueel wringen - is onze specificatie van de inner loop onvoldoende om de outer loop te bewijzen, of is onze code van de inner loop onvoldoende om zijn specificatie te bewijzen?

In deze eerste versie van het algoritme doet de inner loop niet meer dan het grijs kleuren van alle witte vertices. We vervangen hem dus met:

```
1. change W,G satisfy
2.     B ++ G' ++ W' = V,
3.     B ## G',
4.     B ## W',
5.     W' ## G',
6.     G' = G ++ W,
7.     W'.empty;
```

De laatste twee regels zijn voor correctheid van het algoritme overbodig. De inner loop mag de sets G en W willekeurig veranderen zonder dat de variante functie van de outer loop geschaad wordt. Voor de volledigheid voegen we ze toch toe.

Perfect Developer vult alle bewijsverplichtingen probleemloos in met deze specificatie.

Een volgende stap is het koppelen van onze implementatie aan deze specificatie, middels een "via" clausule.

Onze nieuwe, versterkte, postconditie kan echter niet zonder meer bewezen worden. We zullen extra invarianten moeten toevoegen om de extra postcondities te bewijzen:

```
1. W' <<= tmp',
2. G' ++ W' = G ++ W
```

De gehele inner loop wordt:

```
1.  change
2.      W, G
3.  satisfy
4.      B ++ G' ++ W' = V,
5.      B ## G',
6.      B ## W',
7.      W' ## G',
8.      G' = G ++ W,
9.      W'.empty
10. via
11.     var tmp: set of X! = those x::V :- x in V;
12.
13.     assert
14.         #tmp >= 0
15.     proof
16.         assert false
17.     end;
18.
19.     loop
20.     change
21.         tmp, G, W
22.     keep
23.         tmp' <<= tmp,
24.         B ++ G' ++ W' = V,
25.         B ## G',
26.         B ## W',
27.         W' ## G',
28.         W' <<= tmp',
29.         G' ++ W' = G ++ W
30.     until
31.         tmp'.empty
32.     decrease
33.         #tmp';
34.     //do
35.         var x: X;
36.
37.         change x satisfy x' in tmp;
38.
39.         tmp! = tmp.remove(x);
40.
41.         if
42.             [x in W]: W! = W.remove(x);
43.                       G! = G.append(x);
44.             []
45.         fi;
46.     end;
47. end;
```

Deze versie met expliciete specificatie van de inner loop wordt door Perfect Developer bewezen zonder onverwachte problemen (de assertie op regel 12 blijft noodzakelijk).

Een voordeel van deze werkwijze is dat we de hele inner loop kunnen verplaatsen naar een afzonderlijke functie die de specificatie van de change satisfy clausule deelt.

Partitionering middels kleuringsfunctie

We hebben ervoor gekozen om de partitionering te implementeren met drie sets en een aantal invarianten.

Een alternatief hiervoor is om een functie te gebruiken waarmee we elke vertex afbeelden op een kleur. We definiëren een enumeratietype Color en een mapping color:

Set Representatie

In onze papieren versie gebruiken we een set-representatie voor de kleuring. We volgen dezelfde methode in Perfect Developer (zie 1.3.2.2 voor een alternatieve aanpak).

Outer Loop

Verificatie

Perfect Developer probeert terminatie van deze loop te bewijzen, maar heeft problemen met de begrensdheidseis.

> Failed to prove verification condition: Loop body establishes end condition or preserves validity of variant
> To prove: 0 ≤ (**#self**.V - (#$B_{loopend}$ **as int**))
> Reason: Exhausted rules
> Could not prove:
> 0 < **#self**.V

Het lukt blijkbaar niet om te bewijzen dat het aantal elementen in V groter is dan 0. Dat dit wel zo is, volgt uit de preconditie dat start ∈ V.

We proberen dit middels een property te bewijzen, maar ook dit lukt niet. Pas als we de property in een axioma veranderen kan ons programma correct worden bewezen:

```
axiom member(s: set of X, m: X)
  pre
    m in s
  assert
    #s > 0;
```

Dit lijkt een "gat" te zijn in de bewijsregels voor sets. Later zullen andere set-eigenschappen die ogenschijnlijk eenvoudig zijn voor problemen leiden.

Inner Loop

```
1.  opaque function colorize(start:X): bool
2.  pre
3.      start in V
4.  satisfy
5.      result = true
6.  via
7.      var B, G, W: set of X;
8.
9.      B! = set of X {};
10.     G! = set of X {start};
11.     W! = V.remove(start);
12.
13.     assert // (1) Needed to prove bound of variant
14.         0 < #V;
15.
16.     loop
17.     change
18.         B,G,W
19.     keep
20.         B' ++ G' ++ W' = V,
21.         B' <<= V,
22.         G' <<= V,
23.         W' <<= V,
24.         B' ## G',
25.         B' ## W',
26.         W' ## G'
27.     until
28.         G'.empty
29.     decrease
30.         #V - #B';
31.     //do
32.         var h: X! = any i::G:-true;
33.
34.         G! = G.remove(h),
35.         B! = B.append(h);
36.
37.         var tmp: set of X! = those x::V :- x in V;
38.
39.         loop
40.         change
41.             tmp, G, W
42.         keep
43.             tmp' <<= tmp,
44.             B ++ G' ++ W' = V,
45.             //B <<= V,
46.             G' <<= V,
47.             W' <<= V,
48.             B ## G',
49.             B ## W',
50.             W' ## G'
51.         until
52.             tmp'.empty
53.         decrease
54.             #tmp';
55.         //do
```

```
56.             var x: X;
57.
58.             change x satisfy x' in tmp;
59.
60.             tmp! = tmp.remove(x);
61.
62.             if
63.                 [x in W]: W! = W.remove(x);
64.                           G! = G.append(x);
65.                 []
66.             fi;
67.         end;
68.     end;
69.
70.     value(true)
71. end;
```

De inner loop begint op regel 38. Merk op dat we de invarianten van de outer loop herhalen in de inner loop.

Verificatie

Ook hier ondervindt PD een probleem, ditmaal een timeout.

> Failed to prove verification condition: Loop body establishes end condition or preserves validity of variant
> To prove: 0 ≤ (**#self**.V - (#$B_{loopend}$ **as int**))
> Reason: Exceeded time limit
> Could not prove:
> ~(0 = (#$G_{loopstart\_56,9}$ + #$W_{loopstart\_56,9}$))

Het lijkt erop dat onze inner loop het bewijs voor de variante functie van de outer loop heeft aangetast. Als we kijken naar de hint zien we dat de bewijzer moeite heeft met het feit dat #G + #W ongelijk aan 0 is. Dit dit waar is, is in te zien door te stellen dat de set G nooit slinkt en geïnitialiseerd wordt op {start}.

Wederom proberen we eerst om het probleem te isoleren middels asserties (we voegen deze toe na regel 30):

```
assert
  #G > 0,
  #W >= 0,
  #G + #W > 0;
```

De derde assertie geeft (merkwaardigerwijs) problemen. Om de prover op weg te helpen voegen we een property toe:

```
property sum(a,b:int)
pre
    a >= 0,
    b > 0
assert
    a+b > 0;
```

Met deze extra property slaagt de prover erin om ons algoritme correct te bewijzen.

Functie Representatie

Als alternatief op de set-representatie gebruiken we een enumeratietype voor de drie gebruikte kleuren en beelden we onze vertices af op waardes van dit type. Op deze wijze

worden een aantal invarianten die we bij onze set-representatie expliciet moesten maken onnodig. Zo kan met deze representatie een vertex nooit meer dan één kleur hebben.

```
opaque function colorize(start:X): bool
pre
    start in V
satisfy
    result = true
via
    var C: map of (X -> Color);

    C! = map of (X -> Color)
        {for x::V.remove(start) yield
        (pair of (X,Color) {x,white@Color})};
    C! = C.append(start -> gray@Color);

    assert C.dom = V;

    loop
        change
            C
        keep
            C'.dom = V
        until
            #(those x::V:-C'[x] = gray@Color) = 0
        decrease
            #V - #(those x::V:-C'[x] = black@Color);
        //do

            var h: X! = any i::V:- C[i] = gray@Color;

            C[h]! = black@Color;

            var tmp: set of X! = those x::V :- x in V;

            loop
            change
                tmp, C
            keep
                tmp' <<= tmp,
                C'.dom = V
            until
                tmp'.empty
            decrease
                #tmp';
            //do
                var x: X! = any v::tmp :- true;

                tmp! = tmp.remove(x);

                if
                    [C[x] = white@Color]:
                       C[x]! = gray@Color;
                    []
                fi;
            end;
        end;


    value(true)
```

```
55. end;
```

PD ondervindt een timeout bij beide onderdelen van het terminatiebewijs.

Het lijkt erop dat PD de regels die we door onze kleuringsmapping impliciet hebben gemaakt niet kent. Om dit te testen geven we onze eerdere set-invarianten op in termen van de mapping:

```
1.  property mappings(C: map of (X -> Color))
2.  assert
3.      #(those x::C.dom:- C[x] = gray@Color)
4.      + #(those x::C.dom:- C[x] = black@Color)
5.      + #(those x::C.dom:- C[x] = white@Color)
6.      = #C.dom,
7.
8.      (those x::C.dom:- C[x] = gray@Color)
           ## (those x::C.dom:- C[x] = white@Color),
9.      (those x::C.dom:- C[x] = gray@Color)
           ## (those x::C.dom:- C[x] = black@Color),
10.     (those x::C.dom:- C[x] = white@Color)
           ## (those x::C.dom:- C[x] = black@Color);
11.
12. property mappings2(C: map of (X -> Color))
13. pre
14.     #(those x::C.dom :- C[x] = gray@Color) ~= 0
15. assert
16.     #(those x::C.dom :- C[x] = gray@Color) > 0,
17.     #(those x::C.dom :- C[x] = black@Color) < #C.dom;
```

De asserties op regels 2 (t/m 5), 15 en 16 zijn onbewijsbaar, wat ons vermoeden staaft.

We kunnen deze eigenschappen als axioma's opnemen, maar ook dan blijven de terminatiebewijzen onopgelost. Dit alles leidt ons ertoe deze tactiek te verwerpen. Opvallend is overigens hoe het toevoegen van asserties de bewijstijd voor eerder bewezen condities verhoogt, wat deze werkwijze zelfs wanneer hij succesvol blijkt relatief onwenselijk maakt.

Kleuringsklasse

Een aanvulling op ons algoritme is om een aparte klasse te schrijven voor het kleuren. Op deze wijze kunnen de bewijsverplichtingen van de kleurklasse in die klasse worden opgenomen, tezamen met mogelijke axioma's of properties, zonder dat deze in het hoofdprogramma tot een overvloed aan asserties, met negatieve gevolgen voor de bewijsduur, leiden.

We kunnen zo de gewenste kenmerken van onze subprogramma's via postcondities aangeven in plaats van de prover al het werk te laten doen.

```
1.  final class Coloring of X
2.  require X has operator =(arg) end
3.  ^=
4.  abstract
5.      var
6.          dom: set of X,
7.          B,W,G: set of X;
8.
9.      invariant
10.         B ++ G ++ W = dom,
11.         B <<= dom,
12.         G <<= dom,
13.         W <<= dom,
14.         B ## G,
15.         B ## W,
16.         W ## G
```

De interface van de kleuringsklasse geeft de sets B, W en G weer, die een kleuring voorstellen van de elementen van dom. Hiernaast biedt de klasse methodes aan voor het veranderen van kleuren, zoals:

```
1.  schema !makeBlack(x:X)
2.  pre
3.      x in G
4.  post
5.      change G,B satisfy
6.          B' = B.append(x),
7.          G' = G.remove(x)
8.  assert
9.      #G' <= #G,
10.     #B' >= #B;
11.
12. schema !makeGray(x:X)
13. pre
14.     x in W
15. post
16.     change W,G satisfy
17.         W' = W.remove(x),
18.         G' = G.append(x)
19. assert
20.     #W' <= #W,
21.     #G' >= #G;
```

Met gebruik van deze klasse wordt onze loop:

```
1.  opaque function colorize(start:X): bool
2.  pre
3.      start in V
4.  satisfy
5.      result = true
6.  via
7.      var C: Coloring of X! = Coloring of X {V};
8.
9.      C!makeGray(start);
10.
11.     assert
12.         0 <= #C.G <= #V
13.         proof
14.             assert
15.                 0 <= #C.G <= #C.dom,
16.                 #C.dom = #V
17.             end;
18.
19.     loop
20.         change
21.             C
22.         keep
23.             C'.dom = V
24.         until
25.             #C'.G = 0
26.         decrease
27.             #V - #C'.B;
28.         //do
29.
30.             var h: X! = any i::C.G:-true;
31.
32.             C!makeBlack(h);
33.
34.             // inner loop
35.
36.         end;
37.
38.     value(true)
39. end;
```

Opvallend detail is de assertie op regel 10. Deze is niet noodzakelijk voor het bewijs van de loop, maar we voegen hem toe als extra controle. De prover faalt er echter in om deze assertie te bewijzen wanneer we de proof clausule op regels 12 - 16 niet opnemen.

We breiden de code uit met de inner loop:

```
opaque function colorize(start:X): bool
pre
    start in V
satisfy
    result = true
via
    var C: Coloring of X! = Coloring of X {V};

    C!makeGray(start);

    //assert
    //    0 <= #C.G <= #V
    //    proof
    //        assert
    //            0 <= #C.G <= #C.dom,
    //            #C.dom = #V
    //        end;

    loop
        change
            C
        keep
            C'.dom = V
        until
            #C'.G = 0
        decrease
            #V - #C'.B;
        //do
            var h: X! = any i::C.G:-true;

            C!makeBlack(h);

            var tmp: set of X! = V;

            loop
            change
                tmp, C
            keep
                tmp' <<= tmp,
                C'.dom = V
            until
                tmp'.empty
            decrease
                #tmp';
            //do
                var x: X! = any v::tmp :- true;
                tmp! = tmp.remove(x);

                if
                    [x in C.W]: C!makeGray(x);
                    []
                fi;
            end;
        end;

    value(true)
```

```
57. end;
```

Het toevoegen van deze inner loop zorgt ervoor dat het bewijs van de variante functie van de outer loop faalt.

> Failed to prove verification condition: Loop body establishes end condition or decreases variant
> To prove: (**#self**.V - (#$C_{loopend}$.B **as int**)) < (**#self**.V - (#$C_{loopstart\_72,9}$.B **as int**))
> Reason: Exceeded time limit
> Could not prove:
> #$C_{loopstart\_72,9}$.B < #$C_{loopend}$.B
>
> Failed to prove verification condition: Loop body establishes end condition or preserves validity of variant
> To prove: 0 ≤ (**#self**.V - (#$C_{loopend}$.B **as int**))
> Reason: Exceeded time limit
> Could not prove any of:
> #$C_{loopend}$.B ≤ (1 + #$C_{loopstart\_72,9}$.W)
> #$C_{loopend}$.B ≤ (#$C_{loopstart\_72,9}$.W + #$C_{loopstart\_72,9}$.G)
> #$C_{loopend}$.B ≤ (1 + #$C_{loopstart\_72,9}$.B + #$C_{loopstart\_72,9}$.W)
> #$C_{loopend}$.B ≤ (#$C_{loopstart\_72,9}$.B + #$C_{loopstart\_72,9}$.W + #$C_{loopstart\_72,9}$.G)

We kunnen met behulp van extra asserties en invarianten proberen om dit algoritme bewezen te krijgen, maar de verwachting is dat zelfs wanneer we hierin slagen we een zodanig ingewikkeld algoritme krijgen dat het maken van uitbreidingen erop onmogelijk is.

In plaats hiervan zetten we ons initiële plan van divide and conquer verder door en vervangen we de inner loop door een methode. Door de inner loop als een enkele functie te beschouwen wordt het bewijs van onze functie eenvoudiger.

Als we kijken naar de inner loop dan is de annotatie ervan vrij eenvoudig. Alle witte vertices worden grijs gekleurd:

$$\{\ \}\ \textit{inner loop}\ \{\ G = G' \cup W \wedge W = \varnothing\ \}$$

In Perfect Developer termen:

```
opaque schema !doInner(h:X)
pre
    h in B
post
    change G,W satisfy
        G' = G ++ W,
        W'.empty;
```

De originele functie, gebruikmakend van deze doInner functie wordt:

```
1.  function colorize(start:X): bool
2.  pre
3.      start in V
4.  satisfy
5.      result = true
6.  via
7.      var C: Coloring of X! = Coloring of X {V};
8.
9.      C!makeGray(start);
10.
11.     loop
12.         change
13.             C
14.         keep
15.             C'.dom = V,
16.             #C'.G = 0 ==> #C'.W = 0
17.         until
18.             #C'.G = 0
19.         decrease
20.             #V - #C'.B;
21.         //do
22.             var h: X! = any i::C.G:-true;
23.
24.             C!makeBlack(h);
25.             C!doInner(h);
26.         end;
27.
28.     assert
29.         forall v::V :- v in C.B;
30.
31.     value(true)
32. end;
```

Wanneer we de functie laten bewijzen genereert PD de volgende melding:

> Failed to prove verification condition: Loop body preserves loop invariant
> To prove: $C_{loopend}$.dom ≈ **self**.V
> Reason: Exhausted rules
> Could not prove any of:
> $b_x **in** $C_{loopstart\_72,9}$.W
> $b_x **in** $C_{loopstart\_72,9}$.G
> $b_x **in** $C_{loopend}$.B
> $b_x **in** $C_{loopend}$.W
> $b_x **in** $C_{loopend}$.G
> In the case that:
> $b_x **in** $C_{loopstart\_72,9}$.B

Opvallend is dat we niet te maken hebben met een time out maar met een uitputting van alle regels. Merk op dat zowel V als C.dom nergens in de loop worden gewijzigd.

We voegen asserties toe aan de doInner procedure om deze eigenschappen expliciet te maken, zoals de volgende eigenschap:

```
property domUnchanged(h:X)
pre
    h in B
assert
    (self after it!doInner(h)).dom = dom;
```

Deze eigenschap zelf kan niet bewezen worden. Dit is verontrustend aangezien de doInner methode in zijn postconditie duidelijk stelt dat de variabele dom niet wordt veranderd (alle veranderde variabelen dienen in de postconditie te worden genoemd). Hiernaast lukt het ook met deze eigenschap niet om de variante functie van de outer loop correct te bewijzen. Het is zelfs zo dat wanneer een assertie die deze eigenschap letterlijk spiegelt in de outer loop wordt opgenomen deze onbewijsbaar blijft:

(De assertie op regel 25,26 is een letterlijke kopie van de property domUnchanged en zou dus via deze property bewezen moeten worden)

```
1.  opaque function colorize(start:X): bool
2.  pre
3.      start in V
4.  satisfy
5.      result = true
6.  via
7.      var C: Coloring of X! = Coloring of X {V};
8.
9.      C!makeGray(start);
10.
11.     loop
12.         change
13.             C
14.         keep
15.             C'.dom = V,
16.             #C'.G = 0 ==> #C'.W = 0
17.         until
18.             #C'.G = 0
19.         decrease
20.             #V - #C'.B;
21.         //do
22.             var h: X! = any i::C.G:-true;
23.
24.             C!makeBlack(h);
25.
26.             assert
27.                 (C after it!doInner(h)).dom = C.dom;
28.
29.             C!doInner(h);
30.         end;
31.
32.     assert
33.         forall v::V :- v in C.B;
34.
35.     value(true)
36. end;
```

Opacity

Dat Perfect Developer faalt op asserties terwijl ze eerder via een property zijn vastgesteld is zorgwekkend. We leggen dit probleem voor aan Escher Tech en krijgen de volgende reactie:

"…declaring the schema opaque prevents PD from expanding it, and it is then failing to extract some of the state information - in particular, the information that variable 'dom' has not changed…"

Perfect Developer is een work in progress en deze functionaliteit zal in de toekomst worden toegevoegd. Op het moment speelt het opaque keyword ons echter parten.

Opacity is de eigenschap van een algoritme dat deze non-deterministisch is. In ons algoritme bevindt zich de "any" operator, die op non-deterministische wijze een element uit de set G kiest. Om deze operator te kunnen gebruiken dient het gehele algoritme als non-deterministisch te worden gemarkeerd met het keyword opaque.

Gelukkig kunnen we de opaque modifier weglaten zonder een foutmelding. Perfect Developer genereert enkel een waarschuwing.

Gewapend met deze kennis verwijderen we de opaque modifier bij doInner en inderdaad worden al onze properties, zowel als de bewijsverplichtingen van de outer loop, zonder problemen bewezen*[1].

De volledige code:

[1] Op enkele set properties na, die we als axioma's toevoegen aan colorize.

```
final class Coloring of X
require X has operator =(arg) end
^=
abstract
    var
        dom: set of X,
        B,W,G: set of X;

    invariant
        B ++ G ++ W = dom,
        B <<= dom,
        G <<= dom,
        W <<= dom,
        B ## G,
        B ## W,
        W ## G

interface
    function dom;

    function B,G,W;

    schema !makeBlack(x:X)
    pre
        x in G
    post
        change G,B satisfy
            B' = B.append(x),
            G' = G.remove(x)
    assert
        #G' <= #G,
        #B' >= #B;

    schema !makeGray(x:X)
    pre
        x in W
    post
        change W,G satisfy
            W' = W.remove(x),
            G' = G.append(x)
    assert
        #W' <= #W,
        #G' >= #G;

    property (x:X)
    pre
        x in G
    assert
        (self after it!makeBlack(x)).dom = self.dom;

    property setsizes
    assert
        #dom >= 0,
        #B >= 0,
        #W >= 0,
        #G >= 0;
```

```
57.
58.     axiom setsizes2
59.     assert
60.         0 <= #G <= #dom,
61.         0 <= #W <= #dom,
62.         0 <= #B <= #dom;
63.
64.     property setsizes3
65.     pre
66.         #G ~= 0
67.     assert
68.         0 < #dom - #B;
69.
70.     axiom nonnegativesetsizes
71.     assert
72.         #B + #W + #G >= 0;
73.
74.     property (x:X)
75.     pre
76.         x in G
77.     assert
78.         #((self after it!makeBlack(x)).B) = >#B,
79.         #((self after it!makeBlack(x)).G) = <#G;
80.
81.     schema !doInner(h:X)
82.     pre
83.         h in B
84.     post
85.         G! = G ++ W,
86.         W! = set of X {}
87.     assert
88.         dom' = dom;
89.
90.     build{d: set of X}
91.         post
92.             dom! = d,
93.             W! = d,
94.             G! = set of X {},
95.             B! = set of X {};
96. end;
```

```
1.  function colorize(start:X): bool
2.  pre
3.      start in V
4.  satisfy
5.      result = true
6.  via
7.      var C: Coloring of X! = Coloring of X {V};
8.
9.      C!makeGray(start);
10.
11.     loop
12.         change
13.             C
14.         keep
15.             C'.dom = V
16.         until
17.             #C'.G = 0
18.         decrease
19.             #V - #C'.B;
20.         //do
21.             var h: X! = any i::C.G:-true;
22.
23.             C!makeBlack(h);
24.             C!doInner(h);
25.         end;
26.
27.     value(true)
28. end;
```

Re: vorige oplossingen

Interessant is om te kijken of het verwijderen van opacity nog effect heeft op onze eerdere oplossingen. Dit is echter niet het geval.

Bij de eerste oplossing was de functie niet opaque en gebruikte we een change satisfy clausule om onze iterator te kiezen ipv de any operator.

De tweede oplossing faalde op een aantal kerneigenschappen van enumeratietypes, die ook met opacity moeite blijven geven.

De derde oplossing, vóór de keuze werd gemaakt om het kleuren een eigen klasse te geven, faalde juist doordat we de eigenschappen van de inner loop niet expliciet maakten. Het verwijderen van de opaque modifier verandert hier niets aan.

Implementatie van Inner Loop

Nu we de outer loop bewezen hebben met behulp van een externe interne loop, resteert nog het bewijzen van deze interne loop zelf. We voegen code toe aan onze doInner methode als volgt:

```
1.  schema !doInner(h:X)
2.  pre
3.      h in B
4.  post
5.      G! = G ++ W,
6.      W! = set of X {}
7.  via
8.      var tmp: set of X! = dom;
9.
10.     assert #tmp >= 0
11.     proof assert false end;
12.
13.     loop
14.     change
15.         tmp, G, W
16.     keep
17.         tmp' <<= tmp,
18.         W' <<= tmp',
19.         W' <<= W,
20.         G' = G ++ (W -- W')
21.     until
22.         tmp'.empty
23.     decrease
24.         #tmp';
25.     //do
26.         var x: X;
27.
28.         change x satisfy x' in tmp;
29.
30.         tmp! = tmp.remove(x);
31.
32.         if
33.             [x in W]:
34.                 W! = W.remove(x),
35.                 G! = G.append(x);
36.
37.             []
38.         fi;
39.     end;
40. end
41. assert
42.     dom' = dom;
```

Perfect developer genereert de bekende loop verificatiecondities en bewijst deze probleemloos.

Opmerking

Initieel miste de invariant op regel 18 (W' <<= W). Dit leidde tot onbewijsbaarheid van de daaropvolgende invariant, zonder enig aanknopingspunt. Een eerste reactie was om de invariant inductief te bewijzen: na elke loop iteratie geldt G_loopend = G_loopstart ++ (W_loopstart -- W_loopend). Dit is echter niet genoeg om de invariant G_loopend = G_initial + + (W_initial -- W_loopend) te bewijzen.

Voor een implementatie in PD blijft kennis van het bewijs noodzakelijk, omdat het missen van invarianten zoals deze grote gevolgen kan hebben.

Post Asserties

Zoals gezegd in de introductie van deze eerste stap, zorgt dit algoritme ervoor dat alle vertices zwart gekleurd zijn. De perfect developer versie geeft nog geen uitvoer, maar we kunnen deze eigenschap die op het eind van het algoritme geldt testen middels een assertie (hieronder op regel 14).

```
1.  function colorize(start:X): bool
2.  pre
3.      start in V
4.  satisfy
5.      result = true
6.  via
7.      var C: Coloring of X! = Coloring of X {V};
8.
9.      C!makeGray(start);
10.
11.     loop
12.         …
13.         end;
14.
15.     assert
16.         forall v::V :- v in C.B;
17.
18.     value(true)
19. end;
```

PD heeft echter moeite met deze assertie:

Failed to prove verification condition: Assertion valid
To prove: **forall** v::**self**.V • v **in** $C_{57,9}$.B
Reason: Exhausted rules
Could not prove:
$a_v **in** $C_{57,9}$.B
In the case that:
$a_v **in** $C_{57,9}$.W

We voegen de volgende hulp assertie toe:

```
assert
    #C.W = 0;
```

Ook deze kan niet bewezen worden. Op het einde van een loop iteratie (dus na uitvoer van doInner) geldt de assertie ook. Wanneer we hem daar toevoegen wordt hij zonder problemen bewezen:

```
1.  var C: Coloring of X! = Coloring of X {V};
2.
3.  C!makeGray(start);
4.
5.  assert
6.      #C.G ~= 0;
7.
8.  loop
9.      change
10.         C
11.     keep
12.         C'.dom = V,
13.     until
14.         #C'.G = 0
15.     decrease
16.         #V - #C'.B;
17.     //do
18.         var h: X! = any i::C.G:-true;
19.
20.         C!makeBlack(h);
21.
22.         C!doInner(h);
23.
24.         assert
25.           #C.W = 0;
26.     end;
27.
28. assert
29.     #C.W = 0;
30.
31. assert
32.     forall v::V :- v in C.B;
```

De link tussen de waarheid van de assertie na een loop iteratie en de waarheid na de gehele loop lijkt niet gelegd te worden. Ook de assertie #C.G ~= 0 aan het begin van de loop (die impliceert dat de loop ten minste één keer wordt uitgevoerd) helpt niet.

We bekijken de papieren bewijsregels. Voor finalisatie geldt dat op het einde van een loop geldt dat de invarianten gelden samen met de negatie van de guard. Met deze regel in ons achterhoofd voegen we de volgende loop-invariant toe:

```
#C'.G = 0 ==> #C'.W = 0
```

Deze strategie werpt vruchten af en onze assertie wordt goedgekeurd.

B.3.3 Verbondenheid

Zonder Kleuringsklasse

In ons eerste PD programma hoeven we enkel de initialisatie van de inner loop te veranderen. We gebruikten een set tmp om over te itereren die we op V initialiseerden. Ditmaal initialiseren we tmp op de successors van h:

```
var tmp: set of X! = those x::V :- pair of (X,X)
{h,x} in E;
```

Hoewel er verder niets verandert aan de belangrijke kenmerken van de loop (er wordt iedere slag één vertex zwart gemaakt en het feit dat de inner loop al dan niet enkele witte vertexes grijs kleurt heeft op de voortgang geen invloed) ontstaan er toch enige verificatieproblemen:

> Failed to prove verification condition: Loop initialisation establishes end condition or a valid variant
> To prove: 0 ≤ #tmp
> Reason: Exhausted rules
> Could not prove:
> (#((**those** y::$G_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, y} **in self**.E) ** (**those** y::$B_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, y} **in self**.E)) + #((**those** z::$W_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, z} **in self**.E) ** (**those** y::$B_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, y} **in self**.E)) + #((**those** y::$G_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, y} **in self**.E) ** (**those** z::$W_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, z} **in self**.E)))
> ≤ (#((**those** z::$W_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, z} **in self**.E) ** (**those** y::$G_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, y} **in self**.E) ** (**those** y::$B_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, y} **in self**.E)) + #(**those** y::$G_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, y} **in self**.E) + #(**those** z::$W_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, z} **in self**.E) + #(**those** y::$B_{loopstart\_59,9}$ • **pair of** (X, X){$h_{82,13}$, y} **in self**.E))

> Failed to prove verification condition: Loop body preserves loop invariant
> To prove: (($B_{loopend}$ ++ $G_{loopend}$) ++ $W_{loopend}$) ≈ **self**.V
> Reason: Exceeded time limit
> Could not prove any of:
> ~($x_{108,17}$ **in** $B_{loopstart\_59,9}$)
> $x_{108,17}$ **in** $G_{loopstart\_89,13}$
> $x_{108,17}$ **in** $G_{loopstart\_59,9}$
> Could not prove:
> $f_x **in** $G_{loopstart\_89,13}$
> In the case that:
> $f_x **in** $W_{loopstart\_59,9}$
> Could not prove:
> $f_x **in** $W_{loopstart\_89,13}$
> In the case that:
> $f_x **in** $W_{loopstart\_59,9}$

> Failed to prove verification condition: Loop body preserves loop invariant
> To prove: $B_{loopend}$ ## $G_{loopend}$
> Reason: Exceeded time limit
> Could not prove any of:
> ~($x_{108,17}$ **in** $B_{loopstart\_59,9}$)
> $x_{108,17}$ **in** $G_{loopstart\_59,9}$
> $x_{108,17}$ **in** $W_{loopstart\_59,9}$
> $x_{108,17}$ **in** $G_{loopstart\_89,13}$

De eerste van deze drie lijkt een missend theorema op sets te zijn, wat we bij onze colorizer al tegenkwamen. Het toevoegen van een globaal axioma helpt hier echter niet - het wordt niet geïnstantieerd met de tmp variabele:

```
axiom (s: set of X)
assert
    #s >= 0;
```

We voegen daarom een lokaal axioma toe. Er bestaat in PD geen directe manier om iets aan te nemen, maar we kunnen iets dergelijks doen met de volgende constructie:

```
assert #tmp >= 0
    proof assert false end;
```

De prover zal de waarschuwing geven dat "false" niet te bewijzen is, maar de assertie erboven wel met deze "kennis" bewijzen.

De twee latere asserties maken we weer expliciet in de relevante statements en dit helpt de prover genoeg op weg om ze correct te bewijzen:

```
if
    [x in W]:

        W! = W.remove(x),
        G! = G.append(x);
        assert
            B ## G,
            B ++ G ++ W = V;
    []
fi;
```

Ter volledigheid proberen we de bekende tactiek van het opschroeven van de bewijstijd, zonder de hulpasserties (het bovenstaande gebruikt overigens minder dan 30 seconden voor het langste bewijs).

Ook zonder hulpasserties komt de prover uiteindelijk tot de conclusie dat aan de verificatiecondities wordt voldaan. De twee moeilijke condities kosten dan wel zo'n 100 seconden.

Met Kleuringsklasse

Specificatie

We voegen een parameter toe aan onze doInner procedure en veranderen de specificatie als volgt:

```
1.  schema !doInner2(s:set of X)
2.  pre
3.      s <<= dom
4.  post
5.      G! = G ++ (those x::s :- x in W),
6.      W! = W -- (those x::s :- x in W)
7.  assert
8.      dom' = dom,
9.      #B' = #B,
10.     #G' >= #G,
11.     #W' <= #W;
```

Vervolgens laten we onze loop body deze doInner aanroepen met de opvolgers van h:

```
1.  function colorize(start:X): bool
2.  pre
3.      start in V
4.  satisfy
5.      result = true
6.  via
7.      var C: Coloring of X! = Coloring of X {V};
8.
9.      C!makeGray(start);
10.
11.     loop
12.         change
13.             C
14.         keep
15.             C'.dom = V
16.         until
17.             #C'.G = 0
18.         decrease
19.             #V - #C'.B;
20.         //do
21.             var h: X! = any i::C.G:-true;
22.
23.             C!makeBlack(h);
24.
25.             C!doInner2(successors(h));
26.
27.             assert
28.                 #V - #C.B >= 0;
29.         end;
30.
31.     value(true)
32. end;
```

Initieel heeft de prover wat problemen met het rond krijgen van het bewijs voor begrensdheid, maar met de toegevoegde asserties aan doInner (regel 6-10 in de eerste listing) slaagt de prover hier wel in.

Implementatie

Wederom hebben we onze inner loop alleen gedefinieerd op zijn postconditie, maar geen implementatie gegeven. We doen dit nu.

```
1.  schema !doInner2(s:set of X)
2.  pre
3.      s <<= dom
4.  post
5.      G! = G ++ (those x::s :- x in W),
6.      W! = W -- (those x::s :- x in W)
7.  via
8.      var tmp: set of X! = s;
9.
10.     assert #tmp >= 0
11.     proof assert false end;
12.
13.     let W_initial ^= W;
14.     let G_initial ^= G;
15.
16.     loop
17.     change
18.         tmp, G, W
19.     keep
20.         tmp' <<= tmp,
21.         (those x::s :- x in W') <<= tmp',
22.         (those x::s :- x in W') <<= (those x::s :- x in W),
23.         W' = W -- (
               (those x::s :- x in W) -- (those x::s :- x in
    W')
            ),
24.         G' = G ++ (
               (those x::s :- x in W) -- (those x::s :- x in
    W')
            ),
25.         G'## W'
26.     until
27.         tmp'.empty
28.     decrease
29.         #tmp';
30.     //do
31.
32.         var x: X;
33.
34.         change x satisfy x' in tmp;
35.
36.         tmp! = tmp.remove(x);
37.
38.         if
39.             [x in W]:
40.                 assert
41.                     G = G_initial ++
                        (
                          (those i::s :- i in W_initial)
                          --
                          (those i::s :- i in W)
                        ),
42.                     x ~in G,
43.                     x ~in G_initial,
44.                     x in (those i::s :- i in W_initial),
45.                     x in (those i::s :- i in W),
```

```
46.                         x ~in
                            (
                              (those i::s :- i in W_initial)
                              --
                              (those i::s :- i in W)
                            );
47.
48.                     W! = W.remove(x),
49.                     G! = G.append(x);
50.
51.                     assert
52.                         G = G_initial ++
                            (
                               (those i::s :- i in W_initial)
                               --
                               (those i::s :- i in W)
                            );
53.                 []
54.             fi;
55.         end;
56. end
57. assert
58.         dom' = dom,
59.         #B' = #B,
60.         #G' >= #G,
61.         #W' <= #W;
```

Helaas slaagt de prover er niet in om dit algoritme te bewijzen. Invariantie van de laatste invariant vormt een probleem.

Opvallend is dat in de if clausule de eerste set asserties allemaal door de prover bewezen worden (39-45), maar de assertie op het eind niet. Deze is echter wel handmatig af te leiden uit de statements en de erboven expliciet gemaakte aannames.

We laten deze tak voor wat het is en proberen een alternatieve aanpak van de inner loop en zijn invarianten.

Implementatie 2

In plaats van direct in termen van de set constructor "those" te werken, kunnen we de nieuwe invarianten zo kiezen dat ze overeenkomen met de oude:

```
1.  schema !doInner3(s:set of X)
2.  pre
3.      s <<= dom
4.  post
5.      G! = G ++ (those x::s :- x in W),
6.      W! = W -- (those x::s :- x in W)
7.  via
8.      var tmp: set of X! = s;
9.
10.     assert #tmp >= 0
11.     proof assert false end;
12.
13.     var Ws: set of X! = (those x::s :- x in W);
14.
15.     loop
16.     change
17.         tmp, G, W, Ws
18.     keep
19.         tmp' <<= tmp,
20.         Ws' <<= tmp',
21.         Ws' <<= Ws,
22.         G' = G ++ (Ws -- Ws'),
23.         W' = W -- (Ws -- Ws')
24.     until
25.         tmp'.empty
26.     decrease
27.         #tmp';
28.     //do
29.
30.         var x: X;
31.
32.         change x satisfy x' in tmp;
33.
34.         tmp! = tmp.remove(x);
35.
36.         if
37.             [x in W]:
38.                 assert
39.                     x in Ws;
40.
41.                 W! = W.remove(x),
42.                 G! = G.append(x),
43.                 Ws! = Ws.remove(x);
44.             []
45.         fi;
46.     end;
47. end
48. assert
49.     dom' = dom,
50.     #B' = #B,
51.     #G' >= #G,
52.     #W' <= #W;
```

We zien dat door introductie van de set Ws (de witte elementen uit s), we de invarianten grotendeels ongewijzigd kunnen laten. Deze versie van de inner loop kan dan ook zonder problemen worden bewezen.

Wel zouden we de variabele Ws liever als pure bewijsvariabele willen zien, maar doordat deze moet worden gewijzigd in de loop is dit niet mogelijk. Er bestaat geen functie in PD voor het declareren van lokale ghost variabelen.

B.3.4 Pad bijhouden

Definities

Ook in Perfect Developer moeten we het begrip pad specificeren:
We proberen de volgende definities:

```
function isConnected(a,b: X): bool
^=
    exists i::E :- i.x = a & i.y = b;

function isPath(p: seq of X): bool
^=
    forall i::1..<#p :- isConnected(p[i-1],p[i]);
```

En we laten de prover afsplitsing bewijzen:

```
property subPath(p:seq of X, v: X)
    pre
        #p > 1,
        isPath(p),
        isConnected(p.last,v)
    assert
        isPath(p.append(v));
```

Gehele Paden opslaan

Voordat we ons wagen aan de predecessorlijsten en het bijbehorende algoritme om paden op te bouwen, gebruiken we een eenvoudigere manier: we houden voor iedere vertex het complete pad dat tot die vertex leidt bij.

```
function colorize(start,target:X): seq of X
pre
    start in V,
    target in V
satisfy
    #result = 0
    |
    (
      #result > 0 &
      result.head = start &
      result.last = target &
      isPath(result)
    )
via
    var B, G, W: set of X;
    var path: map of (X -> seq of X);

    B! = set of X {};
    G! = set of X {start};
    W! = V.remove(start);

    path! = map of (X -> seq of X) {};
    path! = path.append(start -> seq of X{start});

    assert // (1) Needed to prove bound of variant
        0 < #V;

    loop
    change
        B,G,W, path
    keep
        start ~in W',
        B' ++ G' ++ W' = V,
        B' <<= V,
        G' <<= V,
        W' <<= V,
        B' ## G',
        B' ## W',
        W' ## G',
        path'.dom = (B'++G'),
        forall p::path'.ran :- #p > 0,
        forall p::path'.ran :- isPath(p),
        forall p::path'.ran :- p.head = start,
        forall v::path'.dom :- path'[v].last = v
    until
        G'.empty
    decrease
        #V - #B';
    //do
        assert
            #G > 0,
            #W >= 0,
            #G + #W > 0;
              // (2) Needed to prove bound of variant

        var h: X;

```

```
51.         change h satisfy h' in G;
52.
53.         G! = G.remove(h),
54.         B! = B.append(h);
55.
56.         var tmp: set of X! = those x::V :-
                                    pair of (X,X) {h,x} in E;
57.
58.         assert #tmp >= 0
59.         proof assert false end;
60.
61.         loop
62.         change
63.             tmp, G, W, path
64.         keep
65.             start ~in W',
66.             forall v::tmp' :- isConnected(h,v),
                  // 3: invariant holds initially
67.             tmp' <<= tmp,
68.             B ++ G' ++ W' = V,
69.             //B <<= V,
70.             G' <<= V,
71.             W' <<= V,
72.             B ## G',
73.             B ## W',
74.             W' ## G',
75.             path'.dom = (B++G'),
76.             forall p::path'.ran :- #p >0,
77.             forall p::path'.ran :- isPath(p),
78.             forall p::path'.ran :- p.head = start,
79.             forall v::path'.dom :- path'[v].last = v
80.         until
81.             tmp'.empty
82.         decrease
83.             #tmp';
84.         //do
85.             var x: X;
86.
87.             change x satisfy x' in tmp;
88.
89.             tmp! = tmp.remove(x);
90.
91.             if
92.                 [x in W]:
93.                     assert
94.                         ~(x in path.dom);
95.
96.                     assert
97.                         ~(x = start)
98.                         proof
99.                             assert
100.                                start ~in W
101.                        end;
102.
103.                    assert
104.                        forall p::path.ran :- isPath(p),
```

```
105.                        forall p::path.ran :-
                                   p.head = start;

106.                    assert
107.                        path[h].last = h,
108.                        isConnected(h,x),
109.                        isConnected(path[h].last,x),
110.                        isPath(path[h].append(x));
111.
112.                    W! = W.remove(x),
113.                    G! = G.append(x),
114.                    path! = path.append
                                (x -> path[h].append(x));
115.
116.                    assert
117.                        B ## G;
118.
119.                    assert
120.                        forall p::path.ran :- isPath(p),
121.                        forall p::path.ran :-
                                   p.head = start;
122.
123.                    assert
124.                        B ++ G ++ W = V;
125.
126.                    assert
127.                        forall p::path.ran :- #p >0;
128.
129.                    assert
130.                        forall p::path.ran :- isPath(p);
131.                    assert
132.                        forall p::path.ran :-
                                   p.head = start;
133.                    assert
134.                        forall v::path.dom :-
                                   path[v].last = v;
135.
136.                []
137.            fi;
138.        end;
139.    end;
140.
141.    if
142.        [target in B]: value(path[target]);
143.        [target ~in B]: value(seq of X{});
144.    fi
145.end;
```

Bovenstaande code bestaat voor een groot deel uit asserties die ons ervan verzekeren dat de invarianten goed gekozen zijn en kunnen worden bewezen. Nieuw is dat de functie een postconditie heeft gekregen (regel 4). Deze moet kunnen worden bewezen uit de postconditie van de loop.

Nieuwe invarianten die met de padadministratie te maken hebben zijn:

```
path'.dom = (B++G'),
forall p::path'.ran :- #p >0,
forall p::path'.ran :- isPath(p),
forall p::path'.ran :- p.head = start,
forall v::path'.dom :- path'[v].last = v
```

Wanneer, op het eind van de loop, de set G leeg is, geldt dus (onder aanname dat target $\in$ B) dat path[target] het pad van start naar target bevat.

Deze invarianten zorgen ervoor dat voor alle vertices in B een pad bekend is - de "eenvoudige" helft van de bi-implicatie. De andere richting krijgen we niet in PD bewezen.

Zoals eerder opgemerkt stijgt de complexiteit van de bewijzen naarmate de annotatie toeneemt. Dit algoritme wordt correct bewezen na 20 minuten rekentijd. Dit bemoeilijkt het zoeken naar knelpunten in dien mate, dat we ons in het vervolg alleen nog maar zullen concentreren op de versie die gebruik maakt van een externe klasse voor de inner loop.

Predecessors Opslaan

We proberen hetzelfde te bereiken door een predecessormapping te gebruiken. Ten eerste voegen we de mapping toe aan ons originele algoritme:

```
function colorize(start:X): seq of X
pre
    start in V
satisfy
    #result = 0
via
    var B, G, W: set of X;
    var pred: map of (X -> X);

    B! = set of X {};
    G! = set of X {start};
    W! = V.remove(start);

    pred! = map of (X -> X) {};

    assert // (1) Needed to prove bound of variant
        0 < #V;

    loop
    change
        B,G,W, pred
    keep
        start ~in W',
        B' ++ G' ++ W' = V,
        B' <<= V,
        G' <<= V,
        W' <<= V,
        B' ## G',
        B' ## W',
        W' ## G',
        pred'.dom = (B'++G').remove(start),
        pred'.dom <<= V,
        pred'.ran <<= V,
        forall v::pred'.dom :- isConnected(pred'[v],v)
    until
        G'.empty
    decrease
        #V - #B';
    //do
        assert
            #G > 0,
            #W >= 0,
            #G + #W > 0;
              // (2) Needed to prove bound of variant

        var h: X;

        change h satisfy h' in G;

        G! = G.remove(h),
        B! = B.append(h);

        var tmp: set of X! = those x::V :-
                                pair of (X,X) {h,x} in E;

        assert #tmp >= 0
        proof assert false end;
```

```
56.
57.         loop
58.         change
59.             tmp, G, W, pred
60.         keep
61.             start ~in W',
62.             tmp' <<= tmp,
63.             B ++ G' ++ W' = V,
64.             //B <<= V,
65.             G' <<= V,
66.             W' <<= V,
67.             B ## G',
68.             B ## W',
69.             W' ## G',
70.             forall v::tmp' :- isConnected(h,v),
71.             pred'.dom = (B++G').remove(start),
72.             pred'.dom <<= V,
73.             pred'.ran <<= V,
74.             forall v::pred'.dom :- isConnected(pred'[v],v)
75.         until
76.             tmp'.empty
77.         decrease
78.             #tmp';
79.         //do
80.             var x: X;
81.
82.             change x satisfy x' in tmp;
83.
84.             tmp! = tmp.remove(x);
85.
86.             if
87.                 [x in W]:
88.                     assert
89.                         ~(x in pred.dom);
90.
91.                     assert
92.                         ~(x = start)
93.                         proof
94.                             assert
95.                                 start ~in W
96.                         end;
97.
98.                     W! = W.remove(x),
99.                     G! = G.append(x),
100.                    pred! = pred.append(x -> h);
101.
102.                assert
103.                    B ## G;
104.
105.                assert
106.                    B ++ G ++ W = V;
107.            []
108.        fi;
109.    end;
110. end;
111.
```

```
112.     value(seq of X {})
113.end;
```

Zoals te zien in regels 4 en 111 worden er nog geen uitspraken over de postconditie gedaan.

We stellen enkele invarianten op aan de hand van de eisen van de predecessorfunctie, te zien in regels 30-33.

Inmiddels begint de complexiteit van deze alles-in-één versie zich te wreken: zelfs vrij eenvoudige bewijsverplichtingen vergen veel rekentijd, in het bijzonder de assertie B ## G, die direct volgt uit de invarianten. De reden hiervoor is de uitbreiding van de "kennisset" met de vele invarianten en asserties die elk op hun eigen moment noodzakelijk zijn maar tegelijkertijd de bewijzen van onafhankelijke condities bemoeilijken.

In deze versie wordt nog geen daadwerkelijk pad uitgerekend en geretourneerd. We doen dit nu.

Pad Uitrekenen

Zoals gezegd hoeft voor het vinden van een pad vanaf de startvertex naar de doelvertex enkel de lijst van predecessors beginnend bij de eindvertex te worden berekend.

Deze eigenschap volgt echter niet uit de huidige invarianten. Met name de volgende kenmerken dienen expliciet te worden gemaakt:

- Een pad gevonden op deze wijze is niet oneindig (er zijn geen loops in de predecessors).
- Het pad begint bij de startvertex.

Deze onderdelen vinden we ook terug in de uitvoer van de PD prover. We voeren een loop in die het pad opstelt middels de predecessors:

```
1.  function CalculatePath(B: set of X, target: X,
                pred: map of (X -> X), start: X): seq of X
2.  pre
3.      target in B,
4.      start in B,
5.      pred.dom = B.remove(start),
6.      pred.ran = B,
7.      forall v::pred.dom :- isConnected(pred[v],v)
8.  satisfy
9.      #result > 0,
10.     result.head = start
11. via
12.     var r: seq of X;
13.     var current: X;
14.
15.     r! = seq of X {target};
16.     current! = target;
17.
18.     assert
19.         isPath(r);
20.
21.     loop
22.     change
23.         current, r
24.     keep
25.         #r' >0,
26.         r'.head = current',
27.         isPath(r'),
28.         current' in pred.ran
29.     until
30.         current'=start
31.     decrease
32.         #B - #r';
33.     //do
34.         assert #B - #r > 0
35.
36.         assert
37.             current in pred.dom,
38.             isConnected(pred[current],current),
39.             r.head = current;
40.
41.         current! = pred[current];
42.
43.         let v ^= current;
44.
45.         assert
46.             isPath(r.prepend(v))
47.             proof
48.                 assert
49.                     #r>0,
50.                     isPath(r),
51.                     isConnected(v,r.head);
52.             end;
53.
54.         r! = r.prepend(current);
55.
56.         assert
```

```
57.             isPath(r);
58.     end;
59.
60.     value(r);
61. end;
```

Begrensdheid van de variante functie #B - #r kan niet worden bewezen, aangezien er loops in de predecessorlijst kunnen voorkomen. Voortgang kan wel worden bewezen, aangezien elk element uit B een predecessor heeft en die predecessor zelf in B zit en hieruit volgt impliciet dat de lijst eindigt (of begint, afhankelijk van uw perspectief) bij de startvertex.

Naast dit verwachte probleem struikelt de prover ook op iets anders. De assertie "isPath(r.prepend(v))", op regel 45, kan niet worden bewezen. De drie asserties uit de proof clausule zijn wel bewijsbaar (48,49,50), maar helpen niet de conclusie te bereiken dat de hoofdassertie geldt. Nader onderzoek wijst uit dat een letterlijke vertaling van dit bewijs als property wel bewezen wordt:

```
property legalprepend(v:X, r:seq of X)
pre
    #r>0,
    isPath(r),
    isConnected(v,r.head)
assert
    isPath(r.prepend(v));
```

Wellicht dat de prover uiteindelijk onze assertie ook weet op te lossen, maar dit lukt niet binnen de maximumtijd (van 10 minuten). We nemen hem daarom op als lokaal axioma (door als bewijs de stelling False op te geven).

B.3.5 Depth bijhouden

Predecessors Ordenen

Terug naar ons terminatiebewijs. We willen bewijzen dat de predecessorlijsten eindig zijn en wel met de startvertex achteraan/vooraan.

Een manier om dit te doen is het zoeken naar extra kennis die ons helpt de extra loop (het opstellen van een pad middels de predecessorlijst) te bewijzen. We kunnen de vertices ordenen tijdens het kleuren van de graaf en eisen dat een predecessor van een vertex een lagere ordening heeft dan die vertex. Dit waarborgt voortgang van het algoritme.

We stellen een ordening op door de vertices te nummeren wanneer we deze aan B toevoegen. Door als extra eis op te nemen dat de vertices in een pad oplopend genummerd zijn verzekeren we ons ervan dat paden eindig zijn.

Een bijkomend voordeel van deze aanpak is dat het ordenen van vertices later ook gebruikt kan worden voor depth-first en breadth-first search en afgeleide algoritmes.

We dienen wel aannemelijk te maken dat een pad begint bij de startvertex. Dit doen we door de invariant toe te voegen dat elke vertex met een waarde groter dan 0 een predecessor heeft.

Het, inmiddels vrij forse, programma ziet er als volgt uit:

```
1.  ghost function colorize(start,target:X): seq of X
2.  pre
3.      start in V,
4.      target in V
5.  satisfy
6.      #result = 0
7.  via
8.      var B, G, W: set of X;
9.      var pred: map of (X -> X);
10.     var depth: map of (X -> nat);
11.
12.     B! = set of X {};
13.     G! = set of X {start};
14.     W! = V.remove(start);
15.
16.     pred! = map of (X -> X) {};
17.     depth! = map of (X -> nat) {};
18.
19.     depth! = depth.append(start -> 0);
20.
21.     assert // (1) Needed to prove bound of variant
22.         0 < #V;
23.
24.     loop
25.     change
26.         B,G,W, pred, depth
27.     keep
28.         start in G'++B',
29.         pred'.dom = (B'++G').remove(start),
30.         B' ++ G' ++ W' = V,
31.         B' <<= V,
32.         G' <<= V,
33.         W' <<= V,
34.         B' ## G',
35.         B' ## W',
36.         W' ## G',
37.         pred'.ran <<= B',
38.         forall v::pred'.dom :- isConnected(pred'[v],v),
39.         depth'.dom = (B'++G'),
40.         pred'.dom.append(start) = depth'.dom,
                  // volgt eigenlijk uit andere invs
41.         pred'.ran <<= depth'.dom, // volgt ook uit invs
42.         forall v::pred'.dom :- depth'[v] >
    depth'[pred'[v]],
43.         depth'[start] = 0
44.     until
45.         G'.empty
46.     decrease
47.         #V - #B';
48.     //do
49.         assert
50.             #G > 0,
51.             #W >= 0,
52.             #G + #W > 0;
                  // (2) Needed to prove bound of variant
53.
```

```
54.         var h: X;
55.
56.         change h satisfy h' in G;
57.
58.         G! = G.remove(h),
59.         B! = B.append(h);
60.
61.         var tmp: set of X! = those x::V :- pair of (X,X)
    {h,x} in E;
62.
63.         assert #tmp >= 0
64.         proof assert false end;
65.
66.         loop
67.         change
68.             tmp, G, W, pred, depth
69.         keep
70.             start in B ++ G',
71.             forall v::tmp' :- isConnected(h,v),
72.             pred'.dom = (B++G').remove(start),
73.             tmp' <<= tmp,
74.             B ++ G' ++ W' = V,
75.             //B <<= V,
76.             G' <<= V,
77.             W' <<= V, // exceeded time: loop body
                              preserves invariant
78.             B ## G',
79.             B ## W',
80.             W' ## G',
81.             pred'.ran <<= B,
82.             forall v::pred'.dom :- isConnected(pred'[v],v),
83.             depth'.dom = (B++G'),
84.             pred'.dom.append(start) = depth'.dom,
                  // volgt eigenlijk uit andere invs
85.             pred'.ran <<= depth'.dom, // volgt ook uit invs
86.             forall v::pred'.dom :-
                  depth'[v] > depth'[pred'[v]],
                    // exceeded time: pre of [] satisfied
                       (pred[v] in depth.dom),
                       exceeded time: loop preserves invariant
87.             depth'[start] = 0
88.         until
89.             tmp'.empty
90.         decrease
91.             #tmp';
92.         //do
93.             var x: X;
94.
95.             change x satisfy x' in tmp;
96.
97.             tmp! = tmp.remove(x);
98.
99.             if
100.                [x in W]:
101.                    assert
102.                        ~(x in pred.dom);
103.
```

```
104.                    assert
105.                        ~(x = start)
106.                        proof
107.                            assert
108.                                start ~in W
109.                        end;
110.
111.                    W! = W.remove(x),
112.                    G! = G.append(x),
113.                    pred! = pred.append(x -> h),
114.                    depth! = depth.append(x -> depth[h] +
    1);
115.
116.                    assert
117.                        forall v::pred.dom :-
                                isConnected(pred[v],v),
118.                        depth.dom = (B++G),
119.                        pred.dom <<= depth.dom,
                               // volgt eigenlijk uit
                                  andere invs
120.                        pred.ran <<= depth.dom,
                               // volgt ook uit invs
121.                        forall v::pred.dom :-
                                 depth[v] > depth[pred[v]],
                             // exceeded time: pre of []
                                satisfied (pred[v] in
                                depth.dom),
                                exceeded time: assertion holds
122.
123.                        depth[start] = 0;
124.
125.                    assert
126.                        B ## G;
127.
128.                    assert
129.                        B ++ G ++ W = V;
                             // exceeded time: assertion holds
130.                []
131.            fi;
132.        end;
133.    end;
134.
135.    value(seq of X {})
136.end;
```

Fin

Het programma hierboven genereert vele fouten in de bewijzer en is door de vele asserties en afwijkingen van ons originele plan moeilijk te lezen en bevat waarschijnlijk zelfs fouten.

Op dit punt staken we dan ook onze pogingen. Een bewijsronde duurt langer dan 60 minuten, wat het uitproberen van verschillende oplossingen en formuleringen praktisch gezien onmogelijk maakt.

Het grote probleem is dat we, om de prover te helpen met zijn bewijs, moeten afwijken van ons originele plan. Kleine verschillen stapelen zich op en maken het programma steeds complexer. Dit proces is zeer foutgevoelig, wat het werk voor de prover extra moeilijk maakt, maar het is voor ons onmogelijk om het extra werk te herkennen als een poging om foutieve stellingen te bewijzen versus missende rekenkracht om correcte stellingen te bewijzen.

We baseren hierop onze conclusie in hoofdstuk 4.