TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Kadi Sammul

Võrrandi lahendamine kombineeritud laiendamise ja kitsendamise abil Coqis

Magistritöö (30 EAP)

Juhendaja: Kalmer Apinis, PhD

Tartu 2024

Võrrandi lahendamine kombineeritud laiendamise ja kitsendamise abil Coqis

Lühikokkuvõte:

Programmianalüüsi abil väärtuste hulkadega võrrandisüsteemi lahendamine on aeglane ja mälumahukas protsess, seetõttu on kasulikum kasutada laiendamist. Laiendamisfunktsioon teeb aga analüüsi ebatäpseks, täpsuse taastamiseks on võimalik rakendada kitsendamist. Kombineeritult laiendamise ja kitsendamise kasutamiseks loome Coqis algoritmi, kus lisaks lahenduse leidmisele garanteerime ka analüüsi termineeruvuse.

Võtmesõnad: Coq, programmianalüüs, kombineeritud laiendamine ja kitsendamine

CERCS: P175 Informaatika, süsteemiteooria

Constraint Solving via Combined Widening and Narrowing in Coq

Abstract:

In program analysis constraint solving via sets of values is time-consuming and needs a lot of memory. Therefore, it is wiser to use widening. Widening makes analysis inaccurate - to restore accuracy narrowing can be used. Using combined widening and narrowing we create an algorithm in Coq, where in addition to finding a solution we guarantee the termination of the analysis.

Keywords: Coq, program analysis, combined widening and narrowing

CERCS: P175 Informatics, systems theory

Sisukord

1. Sissejuhatus

Staatilist programmianalüüsi kasutatakse programmide käitumise kindlaks määramiseks programmi käivitamata. Selle jaoks kasutatakse võrrandisüsteeme koos nende lahendajatega. Võtame näiteks lihtsa koodijupi, kus suurendame *i* väärtust nullist sajani.

```
i = 0;
while (i <= 99) {
     i++;
}
```

Tsükli sees enne *i* suurendamist on $i \in \{0, 1, 2, \dots, 100\}$, peale suurendamist $i \in \{100\}$. Programmianalüsaator suudab sellise koodijupi jaoks genereerida võrrandisüsteemi, kus intervallidega tähistame hulki võimalike väärtustega:

$$\mathrm{i}_1 \supseteq \{0\}$$
$$\mathrm{i}_2 \supseteq i_1 \cup i_4$$
$$\mathrm{i}_3 \supseteq i_2 \cap [-\infty, 99]$$
$$\mathrm{i}_4 \supseteq i_3 + 1$$
$$\mathrm{i}_5 \supseteq i_2 \cap [100, \infty].$$

Antud süsteemi lahenduseks saame võimalikud väärtused, kus kõik võrrandisüsteemi tingimused on täidetud. Näiteks:

$$\mathrm{i}_1 \coloneqq \{0\}$$
$$\mathrm{i}_2 \coloneqq \{0, 1, 2, \dots, 100\}$$
$$\mathrm{i}_3 \coloneqq \{0, 1, 2, \dots, 99\}$$
$$\mathrm{i}_4 \coloneqq \{1, 2, 3, \dots, 100\}$$
$$\mathrm{i}_5 \coloneqq \{100\}.$$

See on ka vähim lahendus, mis antud süsteemi lahendab. Kuigi niisuguse analüüsi käigus saab leida kogu vajaliku info, on tegu väga aeglase protsessiga – väärtuste hulgad on väga suured ning võtavad seetõttu ka palju mälu ja nagu programmi täites, on vaja teha ka analüüsi käigus 100 iteratsiooni. Ehk i_2 alustab tühja hulgana, siis sisaldab ühte elementi, seejärel kahte jne.

Seetõttu on kasulikum arvutada intervallidega ja kasutada laiendamist. Selle jaoks saab programmianalüsaator genereerida võrrandisüsteemi, kus intervallid ongi intervallid:

$$\mathrm{i}_1 \sqsupseteq [0, 0]$$
$$\mathrm{i}_2 \sqsupseteq i_1 \sqcup i_4$$

$$\mathrm{i}_3 \sqsupseteq i_2 \sqcap [-\infty, 99]$$
$$\mathrm{i}_4 \sqsupseteq i_3 + 1$$
$$\mathrm{i}_5 \sqsupseteq i_2 \sqcap [100, \infty].$$

Laiendamisfunktsioon (ingl *widening*) teeb analüüsi ebatäpseks - väärtuste alumise või ülemise tõkke muutumisel visatakse vastav tõke ära. Ehk kui i_3 muutub iteratsiooni käigus [0, 0] intervallist [0, 1] intervalliks, siis ülempiir loetakse iteratsiooni käigus kasvavaks, seega loetakse tegelikuks intervalliks [0, ∞]. Selle käigus aga kaotame täpsuses ja saame lahenduse, mis küll täidab kõiki võrrandi tingimusi, aga ei ole parim lahendus:

$$\mathrm{i}_1 \coloneqq [0, 0]$$
$$\mathrm{i}_2 \coloneqq [0, \infty]$$
$$\mathrm{i}_3 \coloneqq [0, \infty]$$
$$\mathrm{i}_4 \coloneqq [1, \infty]$$
$$\mathrm{i}_5 \coloneqq [100, \infty].$$

Täpsuse taastamiseks on võimalik võrrandeid uuesti rakendada.

$$\mathrm{i}_3 \coloneqq [0, \infty] \sqcap [-\infty, 99] = [0, 99]$$
$$\mathrm{i}_4 \coloneqq [0, 99] + 1 = [1, 100]$$
$$\mathrm{i}_5 \coloneqq [0, 100] \sqcap [100, \infty] = [100, 100]$$
$$\mathrm{i}_2 \coloneqq [0, 0] \sqcup [1, 100] = [0, 100]$$

Võrrandite uuesti rakendamiseks võib teoorias vaja minna lõpmatu arv samme, seega on kirjanduses pakutud välja rakendada kitsendamisfunktsiooni (ingl *narrowing*), mis taastab intervallide puuduvad ülemised või alumised tõkked. Laiendamist ja kitsendamist kasutades saame i_2 saja sammu asemel leida kolme sammuga – [0, 0], [0, ∞], [0, 100].

Kitsendamist saab probleemideta rakendada kui analüüsi sammud on monotoonsed. Kui laiendamisega lahendus on leitud, siis iga parema poole uuesti arvutamine tagastab sama või väiksema tulemuse. Seega saab kitsendamist rakendada ja ka selle tulemus on sama või väiksem. Sarnaselt, iga järgnev parema poole arvutamine tagastab sama või väiksema tulemuse. Sellisel juhul kitsendamine tagab termineeruvuse ja tulemus on võrrandisüsteemi lahendus.

Aga näiteks, kontekstitundlikus analüüsis monotoonsus ei kehti. Sellisel juhul võib väärtuste täpsustamise käigus jooksev lahendus jõuda vähimast lahendustest ajutiselt väiksemate väärtusteni – olukord, milleks klassikaline kitsendamisfunktsioon valmis ei ole. Selliste juhtude jaoks on välja pakutud lahendus – kombineeritud laiendamine ja kitsendamine (ingl *intertwined*

widening and narrowing), mis võimaldab väärtuste suurenedes lahendajal uuesti laiendamisele üle minna [1]. Kombineeritud laiendamise ja kitsendamise käigus toimuvaid laiendamise ja kitsendamise faaside vaheldumisi ei ole üldjuhul võimalik hinnata – seega võib protsess jääda faaside vahel vahelduma ning termineerumine pole garanteeritud. Seega Amato jt [1] lahendus ei anna garantiisid mittemonotoonsete funktsioonide korral.

Analüüsi termineerumise garanteerimiseks pakume välja operaatori, mis teeb monotoonsetel juhtudel vaheldumisi laiendamist ja kitsendamist, mittemonotoonsuse korral vaid laiendamist. Tähistame laiendamist sümboliga ∇ ja kitsendamist sümboliga Δ.

$$x \square_s y = \begin{cases} (y,\ 1) & \textit{kui } s = 0 \\ (x \Delta y,\ 2) & \textit{kui } (s = 1 \vee s = 2) \wedge y \leq x \\ (x \nabla y,\ 1) & \textit{kui } s = 1 \\ (x \nabla y,\ 1) & \textit{kui } s = 2 \wedge \neg (y \leq x) \\ (x \nabla y,\ 3) & \textit{muudel juhtudel} \end{cases}$$

Otsime *f' : D*N → D*N* püsipunkti, kus *D* on täielik võre ja *f' (d, s) := d* $\square_s$ *f(d)* s.t. tehniliselt kasutatakse seisundite hulka *N = {0,1,2,3}*. Püsipunkti leidmiseks arvutatakse jada alustades mingist algseisundist $y_0 = (\sqcup\emptyset, 0)$.

Käesoleva töö eesmärk on näidata, et antud operaatori abil on võimalik püsipunkti arvutamise läbi leida võrrandisüsteemile lahendusi. Töö käigus näidatakse, et selline lähenemine annab korrektse tulemuse ja lahenduse leidmine on termineeruv. Töö autori panus ongi selleks antud algoritmi Coqis tõestamine.

Peatükist 2 leiab lühiülevaate vajalikest Coqis kasutatavatest vahenditest, peatükist 3 algoritmi kirjeldamiseks vajalikud mõisted ning peatükist 4 ülevaate algoritmi olulisematest osadest Coqis. Coqis teostatud algoritmi tõestustega leiab lisadest (*I Tõestus Coqis*).

2. Coq raamistik

Coq on formaalsete tõestuste haldussüsteem, mis võimaldab kirjutada matemaatilisi definitsioone, algoritme ja teoreeme ning neid tõestada [2].

2.1 Kasutatud võtmesõnad

Coqis olevad andmestruktuuride deklaratsioonid, funktsioonid, tõestatud väited ja muu kood paikneb moodulites. Moodulitel võib olla ka tüüp ehk signatuur.

```
1  Module Type POSet.
2  Parameter D: Type.
3  Parameter leq : D -> D -> bool.
4
5  Axiom leq_refl: forall {x},
6  leq x x = true.
7  Axiom leq_trans: forall {x y z},
8  leq x y = true -> leq y z = true -> leq x z = true.
9  Axiom leq_antisym: forall {x y},
10 leq x y = true -> leq y x = true -> x=y.
11 End POSet.
```

Joonis 1. Moodul osalise järjestusega.

Defineerime näitena mooduli osalise järjestuse jaoks (*Joonis 1*), selleks anname moodulile nime (rida 1). Coqis funktsiooni tüübi defineerimiseks kasutatakse võtmesõna *Parameter* [3]. Selle abil täpsustatakse elemendi tüüpi või määratakse funktsioonile ette antavate parameetrite ja funktsiooni tagastusväärtuse tüübid. Määramegi oma näites, et tähega D viitame mingile väärtuse tüübile (rida 2), ning loome funktsiooni (rida 3), kus kontrollime kahe tüübiga D elemendi puhul, kas esimene element on teisest väiksem või sellega võrdne.

Aksioomide jaoks kasutatakse võtmesõna *Axiom* (*Joonis 1*), matemaatilistele aksioomidele kohaselt neid ei tõestata vaid kasutatakse alusväidetena, mis loetakse kehtivateks [3]. Seega saab nende abil kirja panna funktsioonide omadusi, näiteks refleksiivsus, transitiivsus, jne. Lisamegi oma näites olevale funktsioonile juurde omadused, mis defineerivad osalise järjestuse.

Refleksiivsuse defineerimiseks anname omadusele nime (rida 5) ning täpsustame, et see kehtib iga mingi elemendi x korral. Seejärel saame öelda, et $x \leq x$ on alati tõene (rida 6). Transitiivsus kehtib iga mingi kolme elemendi x, y ja z korral (rida 7). Kui kehtivad väited $x \leq$

y ja $y \leq z$, siis teame, et kehtib ka $x \leq z$ (rida 8). Antisümmeetrilisus kehtib iga mingi kahe elemendi *x* ja *y* korral (rida 9). Kui kehtivad väited $x \leq y$ ja $y \leq x$, siis teame, et $x = y$ (rida 10).

```
 1 Module NatPOset: POSet.
 2 Inductive Nat := O : Nat (* O on null *)
 3 | S : Nat -> Nat. (* ühe võrra suurendamine *)
 4
 5 Definition D := Nat.
 6
 7 Fixpoint leq (x y: Nat) :=
 8     match x, y with   O, O   => true
 9                     | S x, S y => leq x y
10                     | _  , _   => false
11     end.
12
13 Lemma leq_refl: forall {x}, leq x x = true.
14 Proof.
15     intros. induction x.
16     + simpl. reflexivity.
17     + simpl. apply IHx.
18 Qed.
19
20 Lemma leq_trans: forall {x y z},
21   leq x y = true -> leq y z = true -> leq x z = true.
22 Admitted.
23
24 Lemma leq_antisym: forall {x y},
25   leq x y = true -> leq y x = true -> x=y.
26 Admitted.
27
28 End NatPOset.
29
30 Print Assumptions NatPOset.leq_refl.
31   (* Closed under the global context *)
32 Print Assumptions NatPOset.leq_trans.
33   (* Axioms: NatPOset.leq_trans *)
34
35
36 Module Algo (Import S:POSet).
37 Definition thrice (x:D) (f: D -> D): D := f (f (f x)).
38 End Algo.
```

Joonis 2. Moodul funktsioonidega.

Loome uue näitena uue mooduli, mis kasutab eelmise näite osalise järjestuse moodulit (rida 1). Tüüpide defineerimiseks kasutatakse võtmesõna *Inductive* (*Joonis 2*), mis võimaldab kasutada ka rekursiivseid definitsioone [4]. Mitterekursiivsete tüüpide defineerimiseks saaks kasutada ka *Varianti*, aga selles töös oli rekursiooni võimaldamine oluline. Defineerime rekursiivselt

naturaalarvud, kus eksisteerib väärtus 0 (rida 2) ning funktsioon arvu ühe võrra suurendamiseks (rida 3).

Võtmesõna *Definition* (*Joonis 2*) võimaldab defineerida erinevaid funktsioone [5]. *Definition* ei luba rekursiivseid funktsioone. Anname esimeses näites kasutatud tüübi D väärtuseks defineeritud naturaalarvud (rida 5).

Rekursiivsete tüüpidega funktsioonide defineerimiseks kasutatakse võtmesõna *Fixpoint* (*Joonis 2*), mis võimaldab kujuvõrdluse (ingl *pattern matching*) abil induktiivseid objekte käsitleda [4]. Lõpmatute rekursiivsete funktsioonide defineerimine pole lubatud. Määrame ka eelmises näites kasutatud funktsiooni *leq* töö naturaalarvude korral (rida 7). Selleks kontrollime parameetrite *x* ja *y* võimalikke väärtusi (rida 8). Rekursiooni baasiks on olukord, kus mõlemad väärtused jõuavad nulli (rida 8). Kui mõlemad on suuremad kui null, siis vahendame mõlemat väärtust ühe võrra ja liigume rekursiivselt edasi (rida 9). Muudel juhtudel saame samuti kohe väärtuse tagastada (rida 10).

Tõestamist vajavate tingimuste kirja panemiseks kasutatakse võtmesõnu *Lemma* (*Joonis 2*) ja *Theorem* [5]. Coqi mõistes ei ole nendel erinevusi, matemaatiliselt kasutatakse lemmasid väiksemate abistavate sammude tõestamiseks erinevates teoreemides. Tõestuse alustamiseks on võtmesõna *Proof*, lõpetamiseks *Qed* [6].).

Esimeses moodulis defineerisime refleksiivsuse, transitiivsuse ja antisümmeetrilisuse aksioomidena, nüüd käsitleme neid aga naturaalarvude kontekstis (read 13, 20 ja 24), seega peame näitama, et need jätkuvalt kehtivad. Tõestame esimeses näites kasutatud refleksiivsuse (rida 13). Selleks toome sisse leiduva muutuja ning teeme üle selle induktsiooni (rida 15). Esimeses harus tõestame induktsiooni baasi (rida 16) ning teises sammu (rida 17). Tõestamata jätmiseks kasutame *Admitted* (read 22 ja 26), aga ei saa varjata, et osa tõestamata jätsime. Kontrollina saame moodulis olevad funktsioonid lasta Coqil üle kontrollida – refleksiivsus loetakse täielikult tõestatuks (rida 30) aga transitiivsuse korral tulevad tõestamata osad välja (rida 32).

2.2 Kasutatud taktikad

Coqis on tõestamise jaoks vaja erinevate taktikate kasutamise käigus näidata, et tõestus on täielik [7]. Selle jaoks saab tõestuse seisundi jagada erinevateks tõestamist vajavateks eesmärkideks, kus igal eesmärgil on eeldused ja järeldus. Eeldustega *x: A* ja *y: B* ja järeldusega

C eesmärgi saab kirja panna kujul *x: A, y:B* ⊢ *C*. *A*, *B* ja *C* on väited, *x* ja *y* viited vastavatele eeldustele.

Taktikaid kasutades saab muuta tõestuste seisundeid. Enamasti töötavad taktikad esimest tõestamist vajava eesmärgi peal, osade taktikate puhul on võimalik täpsustada, millisel eesmärgil taktikat rakendada soovitakse. Taktikad võivad ka ebaõnnestuda. Väide on tõestatud, kui tõestamist vajavaid eesmärke rohkem ei ole. Seejärel saab kirjutada *Qed*, mis viitab, et väide on täielikult tõestatud.

Siit leiab mõned levinumad taktikad, mille abil on võimalik suur osa tõestustest ära teha, ülejäänutega tutvumiseks tuleks vaadata dokumentatsiooni [8].

Taktika *intro* on kasutatav, kui järeldus on implikatsioon või algab üldsuse kvantoriga [7]. Selle käigus viiakse järelduses olev eeldus terve eesmärgi eelduseks. Seisundist ⊢ *A* →*B* kasutades *intro H* jõutakse seisundisse *H: A* ⊢ *B*. Üldsuse kvantori puhul on võimalik vastupidise protsessi jaoks kasutada *generalize dependent* [7].

Taktika *exists* on mingi leiduva elemendi väärtuse täpsustamiseks [9]. Täpsustatud väärtus omastatakse vastavale elemendile. Seisundist ⊢ *exists n, n > 4* kasutades *exists 5* jõutakse seisundisse ⊢ *5 > 4*.

Samaväärsuste asendamiseks on *rewrite* [10]. Asendamise suunda on võimalik täpsustada. Seisundist *H: x = y* ⊢ *x + 3 > y* kasutades *rewrite H* jõutakse seisundisse *H: x = y* ⊢ *y + 3 > y*. Sarnane on ka *subst*, mis teeb ära kõik eeldustest saadavad asendused [10]. Mingi konkreetse eelduse rakendamiseks eesmärgil on *apply* [7].

Konjunktsiooni lahutamiseks kaheks väiteks on *split* [9]. Seisundist ⊢ *A* ∧ *B* kasutades *split* saadakse seisundid ⊢ *A* ja ⊢ *B*. Disjunktsiooni haru valimiseks kasutatakse *left* ja *right* [9]. Seisundist ⊢ *A* ∨ *B* kasutades *left* saadakse seisund ⊢ *A*. Lause või väärtuse kõikvõimalikeks juhtumiteks tükeldamiseks kasutatakse käsku *destruct* [9].

Printsiibi *ex falso quodlibet* - vääratest eeldustest on võimalik kõike järeldada - rakendamiseks on *exfalso* [7]. Seisundist *H: x > 5, V: x < 3* ⊢ *x = 8* kasutades *exfalso* jõutakse seisundisse *H: x > 5, V: x < 3* ⊢ *False*. Kui eelduste hulgas on vastuolu, siis vastava tõestamise teeb ära *discriminate* [9]. Näiteks seisundist *2 = 3* ⊢ *False* kasutades *discriminate* jõutakse seisundisse, kus midagi rohkem vaja tõestada ei ole.

Tõestuse käigus jõutud ühesuguste võrdsuste poolteni tõestab lõpuni *reflexivity* [11]. Seisundist $\vdash A = A$ kasutades *reflexivity* jõutakse seisundisse, kus rohkem midagi tõestada ei ole. Võrdsuste poolte vahetamiseks on *symmetry* [11]. Seisundist $\vdash A = B$ kasutades *symmetry* jõutakse seisundisse $\vdash B = A$.

Uue eelduse lisamiseks kasutatakse käsku *pose proof,* eelduse eemaldamiseks käsku *clear* [7]. Kui on seisund $\vdash A$ ja lemma *L: B,* siis kasutades *pose proof L* jõutakse seisundisse $H: B \vdash A$, kus tehes *clear H* on võimalik taastada algne seisund $\vdash A$.

Olemasoleva eesmärgi lihtsamateks triviaalsemateks alameesmärkideks lahutamiseks ja nende automaatseks tõestamiseks on *intuition* [10]. Automaatse tõestamise jaoks on käsk *auto* [12], mis üritab olemasolevate eelduste abil eesmärki tõestada. Näiteks suudab taktika *intuition* seisundist $H: A \leftrightarrow B, V: A \vdash B$ jõuda seisundisse, kus tõestatavaid eesmärke rohkem ei ole, *auto* antud olukorras seda ei suuda. Lineaarseid matemaatilisi eesmärke suudab automaatselt tõestada *lia* [13]. Seisundist $H: x > 5, V: x < 3 \vdash False$ saab *lia* käsu abil samuti seisundi, kus rohkem tõestatavaid eesmärke ei ole.

Taktikad *eauto* [12], *edestruct* [9], *erewrite* [11] ja *eexists* [9] kasutavad metamuutujaid. Need on head juhtudel, kus täpne väärtus pole veel ilmne ja on kasulik väärtuste sobitamist edasi lükata. Seisundist $\vdash exists\ n,\ n > 4$ rakendades taktikat *eexists* saab seisundi $\vdash ?n > 4$. Metamuutujate lõplikud väärtused peavad selginema hiljemalt tõestuse lõpuks.

Sama taktika korduvaks kasutamiseks kasutatakse *repeat* [14], mis töötab kuni ebaõnnestub, ja *do* [14], mille puhul tuleb täpsustada soovitud rakendamiste arv. Taktika rakendamiseks nii, et ebaõnnestumisel viga ei visata, kasutatakse *try* [14].

Matemaatilise induktsiooni rakendamiseks on *induction* [9]. Seisundist $x \vdash x \geq 0$ rakendades taktikat *induction x* saab seisundid $\vdash 0 \geq 0$ ja $IHx: x \geq 0 \vdash S\ x \geq 0$. Induktiivse hüpoteesi lahutamine igaks võimalikuks juhuks toimub käsuga *inversion* [9].

Lokaalsete definitsioonide ja konstantide lihtsustamiseks kasutatakse käsku *unfold* [11]. Funktsioonide ja teiste normaliseerimise jaoks on ka *simpl* [11]. Väikeste variatsioonidega teeb sama ka *cbn* [11].

3. Algoritm

Võrrandisüsteemi lahendamiseks kasutame intervalle koos laiendamisega (ingl *widening*). Analüüsi käigus tekkiva ebatäpsuse eemaldamiseks rakendame võrrandeid uuesti kitsendamise (ingl *narrowing*) käigus, mis taastab intervallide puuduvad ülemised või alumised tõkked. Kitsendades on oht, et töö käigus jooksev lahendus jõuab vähimast lahendusest nii väiksemate kui suuremate väärtusteni. Selle tõttu võtame kasutusele kombineeritud laiendamise ja kitsendamise (ingl *intertwined widening and narrowing*) [1].

Analüüsi termineerumise garanteerimiseks kasutatakse töö juhendaja välja pakutud operaatorit, mis teeb monotoonsetel juhtudel vaheldumisi laiendamist ja kitsendamist, mittemonotoonsuse korral vaid laiendamist.

3.1 Mõisted

Binaarne seis R on **osaline järjestus**, kui ta on refleksiivne $\forall x: xRx$, transitiivne $\forall x, y, z: xRy \wedge yRz \Rightarrow xRz$ ja antisümmeetriline $\forall x, y: xRy \wedge yRx \Rightarrow x = y$. Hulga $X \subseteq D$ **ülemine tõke** $y \in D$ on element, mille puhul kehtib $\forall x \in X, x \leq y$. **Ülemine raja** on hulga *X* vähim ülemine tõke. *D* on **täielik võre**, kui $(D, \leq)$ on osaline järjestus, kus iga *D* alamhulgal on ülemine raja ehk $\forall X \subseteq D, \sqcup X \in D$. Funktsiooni $f: D \to D$ **püsipunktiks** nimetatakse punkti *x: D*, mille puhul $f(x) = x$. Funktsioon *f* on **monotoonne**, kui iga *x* ja *y* puhul kehtib $x \leq y \Rightarrow f(x) \leq f(y)$.

Püsipunkti leidmise kiirendamine ehk **laiendamine** (ingl *widening*) toimub operaatoriga **∇**, mille puhul kehtivad tingimused:

1. $\sqcup \{x, y\} \leq x \nabla y$
2. iga lõpmatu kasvava *D* elementide jada $x_0 < x_1 < x_2 < \cdots < x_n < \cdots$ puhul järgmise jada $y_0 = x_0, y_{i+1} = y_i \nabla x_{i+1}$ korral y_i stabiliseerub $(\exists j, y_j = y_{j+1})$.

Monotoonsete funktsioonide puhul saab kiirendusega püsipunkti arvutamise tulemust parandada, näiteks rakendades sellele veel mõned korrad vastavat funktsiooni. Nimetame selle **püsipunkti täpsustuseks**. **Püsipunkti täpsustuse leidmise kiirendamine** ehk **kitsendamine** (ingl *narrowing*) toimub operaatoriga Δ, mille puhul kehtivad tingimused:

1. $y \leq x \Rightarrow y \leq (x \Delta y) \leq x$
2. iga lõpmatu kahaneva *D* elementide jada $x_0 > x_1 > x_2 > \cdots > x_n > \cdots$ puhul järgmise jada $y_0 = x_0, y_{i+1} = y_i \Delta x_{i+1}$ korral y_i stabiliseerub $(\exists j, y_j = y_{j+1})$.

Kiirendusega püsipunkti arvutamist ja kiirendusega täpsustamist nimetatakse ka **kahe faasiga püsipunkti arvutamiseks**.

4. Algoritm Coqis

Defineerime Coqis täieliku võre kitsendamise ja laiendamisega, ehitame sellele toetuvad kiirendamise protsessid ja neid hõlmavad seisundid ning näitame, et kiirendamise protsess stabiliseerub. Siis on võimalik defineerida püsipunkti leidmine kiirendamiste kaudu nii, et püsipunktini jõudmine on garanteeritud.

4.1 Täielik võre

Otsime püsipunkti täieliku võre jaoks, seega vajame täielikku võre, samuti püsipunkti leidmiseks kasutatavaid operatsioone.

```
Parameter D: Type.
Parameter leq : D -> D -> bool.
```

Joonis 3. Täielik võre, väiksem või võrdne.

```
Axiom leq_refl: forall {x}, leq x x = true.
Axiom leq_trans: forall {x y z},
    leq x y = true -> leq y z = true -> leq x z = true.
Axiom leq_antisym: forall {x y},
    leq x y = true -> leq y x = true -> x=y.
```

Joonis 4. Refleksiivsus, transitiivsus, antisümmeetrilisus.

```
Parameter join: D -> D -> D.
Axiom join_bigger: forall {x y},
    leq x (join x y) = true /\ leq y (join x y) = true.
Axiom join_least: forall {x y d},
    leq x d = true /\ leq y d = true -> leq (join x y) d = true.
```

Joonis 5. Ülemine tõke, ülemine raja.

Defineerime täieliku võre, kus D on elementide tüüp, ning nende elementide jaoks funktsiooni *leq* (*less or equal)*, mis tähistab operatsiooni ≤ (väiksem või võrdne), et kontrollida, kas esimene argument on võres väiksem kui teine (*Joonis 3*). Lisame antud funktsioonile osalist järjestust defineerivad omadused – refleksiivsus, transitiivsus ja antisümmeetrilisus (*Joonis 4*). Täieliku võre mõiste on defineeritud ülemise raja kaudu, seega lisame ka ülemise raja (*join_least*) ja selle definitsioonis kasutatud ülemise tõkke (*join_bigger*) (*Joonis 5*).

```
Parameter widen: D -> D -> D.
Axiom widen_bigger: forall {x y},
  leq (join x y) (widen x y) = true.
Axiom widen_idempotent: forall {x}, widen x x = x.

Parameter narrow: D -> D -> D.
Axiom narrow_middle: forall {x y},
    leq x y = true -> leq x (narrow y x) = true
                   /\ leq (narrow y x) y = true.
Axiom narrow_old: forall {x y},
    leq x y = false -> narrow y x = y.
```

Joonis 6. Laiendamine ja kitsendamine.

Soovime rakendada kitsendamist ja laiendamist, seega lisame definitsioonid püsipunkti leidmise kiirendamise ehk laiendamise ja püsipunkti täpsustuse leidmise kiirendamise ehk kitsendamise jaoks (*Joonis 6*). Selleks defineerime mõlemal juhul funktsiooni parameetrite ja tagastusväärtuse tüübid ja lisame aksioomid. Laiendamise tulemus on alati vähemalt sama suur kui laiendamisel kasutatud elementide ülemine tõke. Ühesuguste elementide laiendamine väärtust ei muuda, saame sama elemendi tagasi. Iga kahe elemendi kitsendamisel on kitsendamise tulemus oma väärtuselt kitsendamiseks kasutatud elementide väärtuste vahel või nendega võrdne. Kui üritame kitsendada mingit väärtust temast suurema väärtusega, siis saame tulemuseks väiksema väärtuse.

```
Fixpoint accelerate (op:D->D->D) (xs:list D): list D :=
    match xs with
    | nil    => nil
    | x::nil => x::nil
    | x::xs' =>
        match accelerate op xs' with
        | nil   => x::nil
        | y::ys => op y x :: y :: ys
        end
    end.
```

Joonis 7. Kiirendamine rekursiivselt.

```
Fixpoint accelerateS {WS} (op:D->D->WS->D*WS)
          (xs:list D) (w:WS): (list D)*WS :=
    match xs with
    | nil    => (nil, w)
    | x::nil => (x::nil, w)
    | x::xs' =>
        match accelerateS op xs' w with
        | (nil, w')   => (x::nil, w')
        | (y::ys, w') =>
            let '(x', w'') := op y x w' in
            (x' :: y :: ys, w'')
        end
    end.
```

Joonis 8. Kiirendamine seisundiga rekursiivselt.

Mõlemat pidi kiirendamist saab teha ühesuguste funktsioonidega sõltuvalt etteantud parameetritest, seega defineerime meetodid kiirendamise jaoks, mis etteantud operaatori ja elementide jada korral rakendab antud operaatorit jada elementidel ja loob saadud tulemustest uue jada (*Joonis 7*). Lisame kiirendamise jaoks ka versiooni, kus kanname kaasas seisundit (*Joonis 8*). Rakendades rangelt kasvaval jadal $x_0 < x_1 < x_2$ antud funktsiooniga laiendamist, saame *accelerate widen* $[x_2, x_1, x_0]$ = [*widen* (*widen* (x_0 x_1) x_2), *widen* (x_0 x_1), x_0].

```
Fixpoint unstable (xs:list D): Prop :=
    match xs with
    | x::xs' =>
        match xs' with
        | nil => True
        | y::ys => x <> y /\ unstable xs'
        end
    | _ => True
    end.
```

Joonis 9. Ebastabiilsus.

```
Parameter widen_bound: nat.
Axiom widen_term: forall xs,
    unstable (accelerate widen xs) -> length xs <= widen_bound.

Parameter narrow_bound: nat.
Axiom narrow_term: forall xs,
    unstable (accelerate narrow xs) -> length xs <= narrow_bound.
```

Joonis 10. Kiirendamine stabiliseerub.

Nimetame võre elementide jada stabiilseks kui jadas leiduvad järjestikused võrdsed elemendid, vastasel juhul ebastabiilseks (*Joonis 9*). Laiendamise puhul leidub pikkus *widen_bound*, millest pikema rangelt kasvava listi *xs* puhul antud listi laiendamine ehk *accelerate widen xs* stabiliseerub. Samamoodi leidub kitsendamise puhul pikkus *narrow_bound*, millest pikema rangelt kahaneva listi *xs* puhul antud listi kitsendamine ehk *accelerate narrow xs* stabiliseerub. Seega saab vastavatel jada vastavat pidi kiirendamist teostada mingi konkreetse arvu sammude jagu, mille järel on garanteeritud stabiliseerumine (*Joonis 10*).

4.2 Seisundi uuendamine

Soovime rakendada kitsendamist ja laiendamist kombineeritult, seega vajame etappide meeles pidamiseks seisundeid. Samuti garanteerime kiirendamise protsessi(de) stabiliseerumise.

```
Inductive WS :=
    | Copy
    | BoxW
    | BoxN
    | Widen.
```

Joonis 11. Seisundid.

Määrame erinevad seisundid, milles saame olla kiirendamise protsesside käigus (*Joonis 11*). Alustame seisundist *Copy*. Sealt siirdume seisundisse *BoxW*, kus tegeleme laiendamisega ehk püsipunkti leidmise kiirendamisega või liigume seisundisse *BoxN*, kus tegeleme kitsendamisega ehk püsipunkti täpsustuse leidmise kiirendamisega, või loobume sellest ja liigume seisundisse *Widen*. Seisundis *Widen* tehakse vaid laiendamist, kitsendamisse sealt ei liiguta.

```
Definition update (o d: D) (ws:WS): D*WS :=
    match ws with
    | Copy  => (d, BoxW)
    | Widen => (widen o d, Widen)
    | BoxW  =>
        if leq d o then
            (narrow o d, BoxN)
        else
            (widen o d, BoxW)
    | BoxN  =>
        if leq d o then
            if leq o d then
                (o, BoxW)
            else
                (narrow o d, BoxN)
        else
            (widen o d, Widen)
    end.
```

Joonis 12. Seisundi uuendamine.

Defineerime seisundite vahel liikumised vastavalt käsitlusel olevatele väärtustele (*Joonis 12*). Igast seisundist on võimalik liikuda vastavalt eelnimetatud loogikale uude seisundisse. Alustame liikumisega laiendamisse, edasi on võimalik laiendamise ja kitsendamise vahel varieerida kui väärtused jõuavad haripunktini. Lõpuks vaid laiendame.

```
Definition update_bound := 2+2*widen_bound+narrow_bound.
```

Joonis 13. Uuendamise ülempiir.

Seisundite vahel liikudes saab stabiliseerumisteni laiendamist teha kaks korda, kitsendamist ühe korra. Seega defineerime kiirendamise protsesside käigus tehtavate sammude jaoks maksimaalse väärtuse, mis on kahekordse maksimaalse võimaliku laiendamise käigus tehtavate sammude arvu ja maksimaalse võimaliku kitsendamise käigus tehtavate sammude arvu summa (*Joonis 13*). Seisundite vahel liikumise käigus lisandub veel kaks sammu.

```
Theorem update_term: forall xs ws,
    unstable (fst (accelerateS update xs ws)) ->
    length xs <= update_bound.
```

Joonis 14. Seisundi kiirendamine stabiliseerub mingi konkreetse arvu sammude järel.

Defineerimise eelnevalt ülempiiri, mis koosneb kiirendamise käigus tehtavatest laiendamiste ja kitsendamise stabiliseerumiseks vajalikest maksimaalsetest sammudest. Tehes läbi need kolm

protsessi, mis kõik stabiliseeruvad, peab ka kogu protsess stabiliseeruma. Kui seda ei juhtu, tehti järelikult vähem samme kui vaja. Seega saades mingi jada kiirendamise tulemusena ebastabiilse jada, oleme kiirendamiseks kasutatud jada abil teinud vähem samme kui stabiliseerumiseks vajalik (*Joonis 14*).

Näidates Coqis tõestamise kaudu, et *update_term* mingi lõpliku arvu sammude järel kindlasti stabiliseerub, oleme saavutanud töö peamise eesmärgi - saame garanteerida, et antud algoritmiga lahenduse leidmine on termineeruv protsess.

4.3 Seisundi kiirendamine stabiliseerub

Nüüd vaatleme mõnda abilemmat, mida oli vaja seisundi kiirendamise stabiliseerumise (*Joonis 14*) tõestamiseks. Esiteks kiirendamisfunktsiooni rakendamise tulemusena ebastabiilse jada saamine tähendab, et pole veel läbi tehtud kõiki laiendamiste ja kitsendamise etappi.

```
Lemma accS_to_acc_app: forall xs w l w',
    accelerateS update xs w = (l,w') ->
    unstable l ->
    (exists c w1 n1 w2,
        accelerate widen  w2 ++
        accelerate narrow n1 ++
        accelerate widen  w1 ++ c = l
            /\ length c <= 1 /\
                (w'=Copy    -> w2=nil /\ n1=nil /\ w1=nil) /\
                (is_BoxW w' -> w2=nil /\ n1=nil) /\
                (is_BoxN w' -> w2=nil)
    ).
```

Joonis 15. Kiirendamise tulemuseks ebastabiilne jada.

Rakendame kiirendamist mingil elementide jadal ning saame tulemusena ebastabiilse jada. Saadud jada on võimalik jagada kaheks laiendamisega ja üheks kitsendamisega saadud jadaks. Eelnevalt näitasime, et nii laiendamine kui kitsendamine on stabiliseeruvad protsessid, seega on kolme stabiliseeruva protsessi järjest tegemine samuti stabiliseeruv. Mis tähendab, et saades tulemuseks ebastabiilse jada, ei saa me veel olla teinud läbi kõiki kolme protsessi (*Joonis 15*).

```
Lemma accelerate_len: forall xs' f xs,
    accelerate f xs' = xs ->
    length xs' = length xs .
```

Joonis 16. Jada kiirendamise pikkus.

```
Lemma accelerateS_len: forall xs' ws' xs ws,
    accelerateS update xs' ws' = (xs, ws) ->
    length xs' = length xs .
```

Joonis 17. Jada kiirendamise pikkus seisundiga.

Teiseks vajame abilemmat, mis määrab kiirendamise tulemusel saadud jada pikkuse. Kiirendamise käigus rakendatakse igale etteantud jada elemendile vastavat operatsiooni ja saadud väärtused pannakse tulemusjadaks kokku. See tähendab, et iga originaaljada elemendi kohta on tulemusjadas täpselt üks element, ehk mingi elementide jada kiirendamise järel saadav tulemus on sama pikk kui jada, mille abil kiirendamine toimus. Defineerime selle nii seisundi täpsustusega (*Joonis 17*) kui ilma (*Joonis 16*).

Esimese lemma abil saame kiirendamisel saadud ebastabiilse tulemusjada jagada neljaks osaks, millest kaks on saadud laiendamiste ja üks kitsendamise tulemusel. Teise lemma abil teame, et antud ebastabiilne jada on sama pikk kui tõestamist vajavas eesmärgis olev jada. Ehk siis antud neljaosalise jada pikkus ei tohi olla suurem kui *update_bound* ehk *2 + 2 * widen_bound + narrow_bound*. Kasutades teadmist, et kiirendamise etapid stabiliseeruvad, saame edasi näidata, et igas osas kasutatud jada on piiratud vastavalt laiendamise puhul väärtusega *widen_bound* ja kitsendamise korral väärtusega *narrow_bound*. Neljanda osa pikkus jääb alati maksimaalselt üheks. Seega osasid liites ei saa pikkus olla suurem kui *update_bound*.

4.4 Püsipunkti leidmine

Lõpliku arvu tundmatutega võrrandisüsteemi saab lihtsustada kujule, kus meil on ainult üks funktsioon üle ennikute võre. Näiteks sissejuhatuses toodud võrrandisüsteem saab kujutada järgnevalt:

$$\mathrm{F}(\mathrm{i}_1, \mathrm{i}_2, \mathrm{i}_3, \mathrm{i}_4, \mathrm{i}_5) \coloneqq ([0,0], i_1 \sqcup i_4, i_2 \sqcap [-\infty, 99], \mathrm{i}_3 + 1, i_2 \sqcap [100, \infty])$$

Algse võrrandisüsteemi lahendamiseks piisab leida selle funktsiooni püsipunkt. Funktsiooni *F* püsipunktiks on väärtus *x*, mille korral *F x = x*.

```
Fixpoint lfp_from (gas:nat) (f:D->D) (w:WS) (x:D) :=
    match gas with
    | 0    => None
    | S g' =>
        let '(y, w') := next f x w in
        if leq x y && leq y x then
            Some x
        else
            lfp_from g' f w' y
    end.
```

Joonis 18. Püsipunkti arvutamine.

```
Definition next (f:D->D) (d:D) (w:WS) : D*WS :=
    update d (f d) w.
```

Joonis 19. Kiirendamise samm.

Defineerime funktsiooni püsipunkti otsimiseks (*Joonis 18*) kasutades funktsiooni seisundi uuendamiseks (*Joonis 19*), et püsipunkti kiiremini leida. Kiirendamise käigus andsime ette kõik jada elemendid, millel soovitud funktsiooni rakendasime. Nüüd soovime kõiki väärtusi jadana kohe ette mitte anda. Selleks leiame iga järgmise elemendi eelmise leitud väärtuse põhjal, ehk uuendame seisundit, rakendades iga kord nõutud operatsiooni viimasel leitud väärtusel. Stabiliseerumiseni jõudes saame leitud väärtuse tagastada ehk tagastame püsipunkti *Some*. Kui etteantud sammude abil stabiliseerumiseni ei jõua, siis protsess püsipunktini ei jõudnud ja tagastame *None*.

```
Lemma lfp_from_term: forall f, lfp_from update_bound f Copy bot <> None.
```

Joonis 20. Püsipunkti leidmine garanteeritud.

Eelnevalt näitasime, et jada kiirendamise protsessidel on olemas maksimaalne konkreetne arv samme *update_bound*, mille järel on garanteeritud stabiliseerumine. Püsipunkti otsides kasutame sama seisundi uuendamise funktsiooni nagu kiirendades ning töötame sarnase printsiibiga. Seega saame näidata, et ka püsipunkti otsimine stabiliseerub *update_bound* sammuga, ehk võttes püsipunkti leidmiseks tehtavate sammude arvuks samuti *update_bound*, garanteerime püsipunkti otsides reaalse tulemuseni jõudmise (*Joonis 20*).

Püsipunkti otsimine on termineeruv protsess, seega saame luua Coqis püsipunkti arvutamise funktsiooni tüübiga $(D \to D) \to D$, mis arvutab püsipunti ehk *lfp_term F = d* parajasti siis kui *fp_from update_bound f bot = Some d*.

Vaatame sissejuhatuses kasutatud näite implementatsiooni Coqis (*II Võre näide Coqis*). Intervallidega arvutamiseks ja erinevate võrede ühendamiseks on vastavalt võred *IntervalLat* ja *PairLat*. Loome iga võrrandisüsteemi muutuja jaoks võre ning ühendame need. Meeldetuletuseks käsitletav võrrandisüsteem intervallidega:

$$\mathrm{i}_1 \supseteq [0,0]$$
$$\mathrm{i}_2 \supseteq i_1 \sqcup i_4$$
$$\mathrm{i}_3 \supseteq i_2 \sqcap [-\infty, 99]$$
$$\mathrm{i}_4 \supseteq i_3 + 1$$
$$\mathrm{i}_5 \supseteq i_2 \sqcap [100, \infty].$$

```
Definition F (d: D): D :=
    let '(i1, i2, i3, i4, i5) := d in
    ([0,0]
    ,L1.join i1 i4
    ,meet i2 [-∞,99]
    ,inc i3
    ,meet i2 [100,∞]
    ).
```

Joonis 21. Võrrandisüsteem Coqis.

Väärtustame võrrandisüsteemi muutujad (*Joonis 21*) ning rakendame nendel loodud püsipunkti leidmise algoritmi *lfp_term*. Saame tulemuseks ([0, 0], [0, 100], [0, 99], [1, 100], [100, 100]) ehk samad tulemused nagu sissejuhatuses pärast täpsuse taastamist lootsime saada:

$$\mathrm{i}_1 \coloneqq [0,0]$$
$$\mathrm{i}_2 \coloneqq [0,100]$$
$$\mathrm{i}_3 \coloneqq [0,99]$$
$$\mathrm{i}_4 \coloneqq [1,100]$$
$$\mathrm{i}_5 \coloneqq [100,100].$$

4.5 Püsipunkti leidmine garanteeritud

Tõestasime Coqis, et *lfp_from* leiab võrrandisüsteemile lahenduse ja termineerub. Vaatleme detailsemalt, kuidas selleni jõudsime. Esmalt defineerime püsipunkti otsimise funktsiooni kujul, kus kogume protsessi käigus kõik leitud vaheväärtused mingisse jadasse.

```
Fixpoint lfp_from_trace (gas:nat) (f:D->D) (w:WS) (x:D) (acc:list D) :=
    match gas with
    | 0    => (x::acc, w)
    | S g' =>
        let '(y, w') := next f x w in
        if leq x y && leq y x then
            (y :: x :: acc, w')
        else
            lfp_from_trace g' f w' y (x :: acc)
    end.
```

Joonis 22. Jada pikendamine.

Leiame etteantud sammude võrra elemente, küsides viimase elemendi põhjal iga kord uue elemendi ja lisades saadud tulemuse elementide jadasse. Kuna puudub garantii, et jadas juba elemendid eksisteerivad, siis protsessi alustades küsime eraldi elemendi, millest alustada ja lisame selle ise pikendamise käigus jadasse. Jada pikendame kuni nõutud sammude arv saab otsa või jada stabiliseerub (*Joonis 22*) .

```
Lemma lfp_from_trace_is_accelerateS_update: forall g xs' ws' x acc ws f,
    accelerateS update xs' ws'= (x::acc, ws) ->
    exists xs,
        lfp_from_trace g f ws x acc = accelerateS update (xs++xs') ws'.
```

Joonis 23. Jada pikendamine on kiirendamine.

Nüüd saame näidata, et jada pikendamine ja kiirendamise protsessid on samasuguse tulemusega. Võtame aluseks mingi jada. Tehes sellel jadal kiirendamist, saame tulemuseks mingi uue jada, kus on originaaljada elementidel kõigil rakendatud etteantud funktsiooni. Selle uue jada peal on võimalik teha pikendamist, võttes esimeseks väärtuseks saadud jada esimese elemendi ja juba olemasolevaks jadaks ülejäänud tulemuse. Sellise jada pikendamise käigus saame sama jada nagu siis, kui kõige esimese kiirendamise asemel oleks teinud kiirendamist mingi kindlasti leiduva pikema jada peal. Seega saame väita, et jada pikendamine ja jada kiirendamine on põhimõtteliselt sama protsess *(Joonis 23*).

```
Lemma lfp_from_trace_unstable_bound:
    forall g xs' ws' f ws x acc xs ws'',
    accelerateS update xs' ws' = (x::acc, ws) ->
    unstable (x::acc) ->
    lfp_from_trace g f ws x acc = (xs, ws'') ->
    unstable xs -> length xs <= update_bound.
```

Joonis 24. Jada pikendamine stabiliseerub mingi konkreetse arvu sammude järel.

Näitasime, et jada pikendamine on sama, mis natuke teistsuguse jada kiirendamine, ning iga jada pikendamise jaoks sobiv kiirendatav jada leidub. Seega eksisteerib iga jada pikendamise korral vastav kiirendamiseks mõeldud jada, mille otsimine stabiliseerub. Kuna aga pikendamine jõuab sellises olukorras alati sama tulemuseni, siis peab ka pikendamine alati stabiliseeruma. Kuna oleme näidanud, et jada kiirendamine stabiliseerub mingi konkreetse arvu sammude järel, siis saame seda väita ka jada pikendamise kohta (*Joonis 24*).

```
Lemma lfp_from_trace_from_lfp_from: forall g f x ws r acc tl,
    unstable (x::acc) ->
    fst(lfp_from_trace g f ws x acc) = r::tl ->
    not (unstable (r::tl)) ->
    lfp_from g f ws x = Some r.
```

Joonis 25. Püsipunkti leidmine jada pikendamisega.

Vaadates püsipunkti leidmise ja jada pikendamise algoritme, tehakse mõlemas ühesuguseid samme, erinevus on tagastusväärtuses. Jada pikendamise algoritmis kogutakse kõik vahepealsed leitud väärtused kokku jadasse, püsipunkti algoritm leiab vaid lõppväärtuse, milleks on konkreetne väärtus olukorras, kus jada pikendamisel jõutakse stabiilse jadani. Seega kui on võimalik mingi arvu sammude abil jõuda jada pikendades stabiilse jadani, siis tehakse püsipunkti algoritmis läbi täpselt samad sammud ja jõutakse mingi konkreetse väärtuseni. Mis tähendab, et sama arvu sammudega on garanteeritud püsipunktini jõudmine. Ehk jõudes mingi konkreetse arvu sammudega ebastabiilset jada pikendades stabiilse jadani, on sama arvu sammudega originaaljada esimesest elemendist otsimist alustades garanteeritud püsipunktini jõudmine (*Joonis 25*).

5. Kokkuvõte

Pakkusime välja kitsendamise ja laiendamise kombineeritult rakendamiseks operaatori, mis teeb monotoonsetel juhtudel vaheldumisi laiendamist ja kitsendamist, mittemonotoonsuse korral vaid laiendamist, kus otsitakse $f' : D^*N \to D^*N$ püsipunkti, kus $f'(d, s) := d \,\square_s\, f(d)$ ning püsipunkti leidmiseks arvutatakse jada alustades mingist algseisundist $y_0 = (\sqcup\emptyset, 0)$.

$$x \,\square_s\, y = \begin{cases} (y, 1) & \text{kui } s = 0 \\ (x \Delta y, 2) & \text{kui } (s = 1 \vee s = 2) \wedge y \leq x \\ (x \nabla y, 1) & \text{kui } s = 1 \\ (x \nabla y, 1) & \text{kui } s = 2 \wedge \neg(y \leq x) \\ (x \nabla y, 3) & \text{muudel juhtudel} \end{cases}$$

Operaatori rakendamiseks defineerisime Coqis algoritmi, mille käigus rakendatakse kombineeritult laiendamist ja kitsendamist, mis kõik on termineeruvad protsessid. Seetõttu on ka kogu protsess termineeruv. Tänu sellele saame garanteerida tulemuseni jõudmise.

Töö eesmärgiks oligi näidata, et antud operaatori abil on võimalik püsipunkti arvutamise läbi leida võrrandisüsteemile lahendusi. Selle saavutamiseks tõestasime edukalt Coqis, et implementatsioon antud algoritmist on termineeruv.

Viidatud kirjandus

[1] Amato G., Scozzari F., Seidl H., Apinis K., Vojdani V. Efficiently intertwining widening and narrowing, *Science of Computer Programming*, 2016. https://www.sci.unich.it/-~amato/papers/scp16.pdf (02.05.2024).

[2] The Coq Proof Assistant. https://coq.inria.fr/ (11.04.2024).

[3] Reference manual of Coq: Functions and assumptions¶ https://coq.inria.fr/doc/V8.19.0/-refman/language/core/assumptions.html (11.04.2024).

[4] Reference manual of Coq: Inductive types and recursive functions https://coq.inria.fr/doc/-V8.19.0/refman/language/core/inductive.html (11.04.2024).

[5] Reference manual of Coq: Definitions https://coq.inria.fr/doc/V8.19.0/refman/language/-core/definitions.html (11.04.2024).

[6] Reference manual of Coq: Proof mode https://coq.inria.fr/doc/V8.19.0/refman/proofs/-writing-proofs/proof-mode.html (11.04.2024).

[7] Reference manual of Coq: Tactics https://coq.inria.fr/doc/V8.19.0/refman/proof-engine/-tactics.html (11.04.2024).

[8] Reference manual of Coq https://coq.inria.fr/doc/V8.19.0/refman/index.html (02.05.2024).

[9] Reference manual of Coq: Reasoning with inductive types https://coq.inria.fr/doc/-V8.19.0/refman/proofs/writing-proofs/reasoning-inductives.html (16.04.2024).

[10] Reference manual of Coq: Solvers for logic and equality https://coq.inria.fr/doc/-V8.19.0/refman/proofs/automatic-tactics/logic.html (16.04.2024).

[11] Reference manual of Coq: Reasoning with equalities https://coq.inria.fr/doc/V8.19.0/-refman/proofs/writing-proofs/equality.html (11.04.2024).

[12] Reference manual of Coq: Programmable proof search https://coq.inria.fr/doc/V8.19.0/-refman/proofs/automatic-tactics/auto.html (16.04.2024).

[13] Reference manual of Coq: Micromega: solvers for arithmetic goals over ordered rings https://coq.inria.fr/doc/V8.19.0/refman/addendum/micromega.html (16.04.2024).

[14] Reference manual of Coq: Ltac https://coq.inria.fr/doc/V8.19.0/refman/proof-engine/-ltac.html (16.04.2024).

Lisad

I Tõestus Coqis

Tööga on kaasas fail „Update.v“, kust leiab algoritmi implementatsiooni ja tõestuse Coqis.

II Võre näide Coqis

Tööga on kaasas fail „Example.v“, kust leiab võre implementatsiooni ja püsipunkti arvutamise näite antud võrel ning sissejuhatuses kasutatud näite implementatsiooni Coqis. Faili Coqis jooksutamiseks on vaja (samas kaustas) ka põhifaili „Update.v“, millel tuleb teha „*Compile buffer*“, seejärel saab näitefaili jooksutada.

III Litsents

Mina, Kadi Sammul

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose **Võrrandi lahendamine kombineeritud laiendamise ja kitsendamise abil Coqis**, mille juhendaja on Kalmer Apinis, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Kadi Sammul

13.05.2024