CHALMERS |  GÖTEBORGS UNIVERSITET

Formalisering av algoritmer och matematiska bevis

En formalisering av Toom-Cook algoritmen i Coq med SSReflect

Kandidatarbete inom Datateknik och Matematik

JESPER ANDERSSON, ÅSA LIDESTRÖM, DANIEL OOM, ANDERS SJÖBERG, NICLAS STÅHL

Chalmers tekniska högskola
Göteborgs universitet
Institutionen för Data- och Informationsteknik

Göteborg, Sverige, Juni 2013

Formalisering av Algoritmer och Matematiska Bevis
En formalisering av Toom-Cook algoritmen i Coq med SSReflect

JESPER ANDERSSON,
ÅSA LIDESTRÖM,
DANIEL OOM,
ANDERS SJÖBERG,
NICLAS STÅHL

Examiner: Anders Mörtberg

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2013

Abstract

Computer-aided formalization of mathematics has progressed in the last decade with the formalization of very large and complex proofs such as the proof of the Four color theorem and the Feit-Thompson theorem. In this report we present a formal proof of the Toom-Cook algorithm using the Coq proof assistant together with the SSReflect extension. The Toom-Cook algorithm is used to multiply polynomials and can also be used for integer multiplication.

Sammanfattning

Utvecklingen av datorstödd formalisering av matematik har gått framåt det senaste decenniet med formaliseringen av mycket långa och komplicerade bevis såsom beviset för Fyrfärgssatsen och Feit-Thompsons sats. I den här rapporten presenterar vi ett formellt bevis för algoritmen Toom-Cook med hjälp av bevisassistenten Coq tillsammans med tillägget SSReflect. Toom-Cook är en algoritm för att multiplicera polynom och kan även användas för att multiplicera heltal.

Tacksägelser

Ett stort tack till vår handledare Anders Mörtberg som har ställt upp för oss och kommit med idéer och råd under arbetets gång. Vi vill även tacka Ana Bove och Guilhem Moulin som hjälpte oss att komma igång med Coq och ställde upp då Anders inte hade möjlighet till detta. Till sist vill vi tacka alla dem som hjälpt till genom att läsa, granska och korrigera rapporten under dess framväxande.

Innehåll

Kapitel 1

Inledning

Under senare år har några mycket långa och komplexa matematiska bevis presenterats. Några av dem har också byggt på en mycket omfattande analys av tusentals fall utförda av datorprogram. Det är mycket tidsödande och svårt, om ens praktiskt möjligt att för hand kontrollera varje steg i ett sådant bevis. Få matematiker har tid, lust och den speciella kompetensen inom just det specifika matematiska området för att kunna eller vilja ägna år åt att kontrollera korrektheten hos ett sådant bevis. Dessutom finns risken att ett fel i ett bevis ändå inte upptäcks vid kontroll om beviset är hundratals- eller tusentals sidor långt[1].

Bevisassistenter är datorsystem för att formalisera och verifiera bevis. Genom att använda bevisassistenter kan risken för att bevisen innehåller fel minimeras och öka tilltron till att de är korrekta.

Fyrfärgssatsen[2] och Feit-Thompsons sats[3] är två exempel på satser vars bevis har formaliserats och kontrollerats i bevisassistenter. Fyrfärgssatsen säger att varje karta kan färgläggas med fyra färger så att inga regioner med gemensam gräns får samma färg. Det ursprungliga beviset var över hundra sidor långt och byggde också på datorprogrammerad analys av miljontals fall. Detta gjorde beviset svårt att kontrollera och kontroversiellt. Beviset formaliserades och verifierades i bevisassistenten Coq 1995.

De verktyg som används vid formalisering och datorverifiering kan även användas för att verifiera programkod. Formella metoder är således intressant för både programmerare och matematiker. Dagens stora och komplexa programvaror skulle kunna utnyttja formella metoder. Det kan vara användbart i kritiska system, till exempel medicinsk utrustning, där det inte får bli fel eller i system som inte kan uppdateras i efterhand som hårdvarunära mjukvara på ROM-minnen.

De mest använda metoderna idag för att kontrollera kod bygger på att testa om koden ger korrekt resultat för olika indata. På detta sätt har man en chans att upptäcka om koden innehåller fel, men det visar inte att koden saknar fel eftersom det i många fall är omöjligt att testa alla kombinationer av indata. Formella metoder skulle i dessa fall kunna användas till att garantera korrekthet hos koden.

Ett exempel på projekt för att verifiera kod är CompCert. Det utforskar möjligheten att utveckla formellt bevisade kompilatorer. Anledningen att man vill ha en formellt bevisad kompilator är att vid vissa optimeringar kan kompilatorn

skapa buggar och beräkningsfel. Projektets viktigaste resultat hittills är en fungerande C-kompilator som är bevisad i Coq och som stödjer hela ANSI C (som är den första standardiserade versionen av C) med få undantag[4].

Matematiska programvaror som MATLAB spelar en stor roll för beräkningar inom forskning och industri och det finns därmed ett stort intresse av att de är pålitliga. De är dock inte buggfria. Ett sätt att göra dem mer pålitliga skulle vara att formalisera de ingående algoritmerna och visa att de är korrekt implementerade[5].

Coq är en av flera avancerade bevisassistenter som det bedrivs aktiv forskning kring. Några andra är Agda[6] som utvecklats på Chalmers. Z3[7] som har utvecklats av Microsoft och kan användas tillsammans med flera stora ickefunktionella språk som Python, C och .NET. HOL-light som används av Intel för att bevisa att vissa hårdvarukomponenter fungerar korrekt och i det pågående projektet att formalisera Keplers förmodan[8].

Toom-Cook är en algoritm för att multiplicera polynom men kan även användas för att multiplicera heltal. Den är intressant eftersom den har en bättre asymptotisk tidskomplexitet än polynommultiplikation utförd direkt enligt definitionen [1].

1.1 Syfte

Syftet med det här projektet är att formalisera och bevisa korrektheten hos en generell version av Toom-Cook-algoritmen för multiplikation av polynom. Detta skall göras i Coq som är en interaktiv bevisassistent.

1.2 Rapportens upplägg

I kapitel 2 beskrivs projektets arbetsmetod. I kapitel 3 introduceras formalisering av matematik och bevisassistenter. Kapitel 4 beskriver bevisassistenten Coq som har använts i projektet. Kapitel 5 definierar algoritmen Toom-Cook och ger ett informellt bevis för att den är korrekt. I kapitel 6 presenteras formaliseringen och beviset av Toom-Cook i Coq och i kapitel 7 diskuteras Coq, formalisering och algoritmen.

[1] För definition av polynommultiplikation, se appendix A.1.

Kapitel 2

Metod och genomförande

I det här kapitlet beskrivs tillvägagångssättet för formaliseringen av algoritmen.

2.1 Litteraturstudie och färdighetsträning

Som första steg i arbetet genomfördes en studie med målet att projektdeltagarna skulle få tillräckligt med kunskaper för att kunna programmera och skapa bevis med hjälp av Coq och SSReflect. Som material användes bland annat *Software Foundations* av Benjamin C. Pierce som är kursmaterial till en grundkurs i Coq[9], *Coq in a Hurry* av Yves Bertot som är en kort introduktionsartikel till Coq[10] och *SSReflect tutorial* av Georges Gonthier som är en introduktion till SSReflect [11]. Som en övning i att hantera polynom i SSReflect gjordes även ett bevis av Karatsuba-algoritmen som redan är formaliserad i Coq. I kapitel 4 beskrivs Coq.

2.2 Definition och bevis för hand

Först gjordes en informell men detaljerad definition av en generell version av Toom-Cook. Ett detaljerat bevis gjordes också på papper för senare användning i implementationen. Resultatet av detta steg i implementationen finns i kapitel 5.

2.3 Implementation i Haskell

I samband med framtagningen av den informella definitionen och beviset gjordes en implementation av Toom-Cook för heltal i Haskell. Den gjordes först för Toom-Cook-3 men generaliserades sedan till Toom-Cook-m. Implementationen testades med testfallsgeneratorn QuickCheck genom att jämföra implementationen av Toom-Cook med den heltalsmultiplikation som finns definierad i Haskell. Resultatet av denna fas i projektet finns i appendix B.1.1.

Anledningen till att en första implementation gjordes i Haskell var att gruppen inte hade någon erfarenhet av Coq innan projektet och man genom att först implementera en version av algoritmen i Haskell som kunde testas kunde få en bättre förståelse för hur Toom-Cook fungerar.

2.4 Implementering och bevis i Coq

När informella definitionen och beviset och den praktiska implementation var färdiga, utfördes implementeringen av Toom-Cook och dess bevis i Coq med hjälp av SSReflect. Först skapades en definition av algoritmen i Coq med hjälp av definitionen av algoritmen i Haskell och den informella definitionen och som låg nära dessa definitioner. Denna definition omarbetades senare under utarbetandet av beviset för att underlätta vissa bevissteg. Det formella beviset delades liksom det informella upp i ett huvudbevis och mindre lemman, bland annat för att underlätta arbetsfördelningen inom gruppen. Resultatet av detta beskrivs i avsnitt 6.1 och avsnitt 6.2.

2.5 Avgränsningar

Algoritmen i Coq har implementerats med de låsta typerna för bland annat polynom och matriser i SSReflect som beskrivs i avsnitt 4.4. Den kan alltså inte användas för att beräkna produkten mellan konkreta polynom. Om tid hade funnits hade en eller flera exekverbara versioner av Toom-Cook implementerats, som sedan skulle visats korrekt genom att relatera dem till implementationen med de låsta typerna (se avsnitt 4.4 för mer information låsning). Detta tillvägagångssätt för implementation av matematiska algoritmer diskuteras i avsnitt 7.4.

Kapitel 3

Bevisassistenter och formalisering av matematik

Coq är en *bevisassistent* som används för att formalisera och bevisa matematik och kod. I det här avsnittet ges en introduktion till vad ett matematiskt bevis är och vad som menas med formalisering av matematik. Sedan beskrivs vad en bevisassistent är och kan göra.

3.1 Matematiska bevis och datorverifiering

Vanlig matematisk text är en blandning mellan formler och naturligt språk. Ett matematiskt bevis i en kursbok eller vetenskaplig artikel kan sägas vara en skiss av ett fullständigt bevis, ett argument, som ska övertyga läsaren om att satsen är korrekt och beviset är giltigt. Varje litet logiskt bevissteg behöver inte tas med, särskilt inte om den avsedda läsaren är van vid typen av bevis som det handlar om. Dessa steg kan läsaren själv fylla i.

Det är också vanligt att många saker får framgå av sammanhanget. Till exempel kan symbolen 1 stå för olika saker, bland annat det naturliga talet 1 som är efterföljare till 0, det rationella talet $\frac{1}{1}$ eller det multiplikativa enhetselementet i en ring[1]. Oftast kan en mänsklig läsare ur sammanhanget förstå vilken betydelse av 1 som avses i ett visst fall.

För att ett datorsystem ska kunna kontrollera bevis krävs att det finns en exakt definition av vad det innebär att ett bevis är giltigt. En definition är att

> "... the correctness of a mathematical text is verified by comparing it, more or less explicitly, with the rules of a formalized language" – [12].

Givet den definitionen måste ett bevis *formaliseras* för att ett datorsystem mekaniskt ska kunna kontrollera det. Det innebär att sådant som en mänsklig läsare förstår ur sammanhanget måste göras explicit och bevissteg som är så små att de ses som självklara för en människa måste också skrivas ut.

Datorsystemet kan sedan kontrollera om varje steg i beviset följer från föregående steg eller från axiom genom en bestämd mängd fastslagna *härledningsregler*.

[1] För en definition av ringar, se appendix A.1

Dessa är enkla logiska regler om hur man får gå från givna premisser till slutsatser. Till exempel säger härledningsregeln *modus ponens* att man ur förutsättningarna A och $A \to B$ får dra slutsatsen att B gäller. Systemet kontrollerar också om de uttryck man skriver in är tillåtna, det vill säga om man använt rätt syntax. $\forall 3 \overset{\pm}{\in} \leftrightarrow =$ är till exempel inget välformat uttryck. Det har ingen matematisk betydelse även om de ingående symbolerna är matematiska och logiska symboler. Ett annat exempel på felaktig syntax eller felformade uttryck är om man försöker definiera en funktion

```
exempel_funktion (n : nat) : bool := n + 2.
```

eller med matematisk notation

$$\begin{aligned} exempel\_funktion : \mathbb{N} &\to \{sant, falskt\} \\ n &\mapsto n + 2 \end{aligned}$$

som enligt specifikationen ska gå från naturliga tal (`nat`) till boolska sanningsvärden (`bool`) men som samtidigt sägs ska anta värdet $n + 2$ för varje naturligt tal n.

För att formalisera matematik i en bevisassistent måste man alltså bestämma de exakta reglerna som bevisassistenten ska följa:

- vilka härledningsregler som ska vara tillåtna,
- vilka axiom som skall användas,
- vilka symboler det är tillåtet att forma uttryck med,
- och vilka uttryck av dessa symboler som är godkända.

Eftersom det finns olika möjliga val för alla dessa punkter finns det olika *formella språk* eller *logiska system*. Ett val av en uppsättning härledningsregler, axiom och regler för vilka symboler och uttryck som är tillåtna ger oss *ett* möjligt logiskt system.

De flesta logiska system som används för att formalisera matematik kan dock uttrycka och härleda ungefär samma matematik och logik, möjligen på något olika sätt, eftersom skaparna har varit intresserade av att fånga och beskriva naturliga logiska resonemang.

En viktig skillnad är dock den mellan intuitionistisk och klassisk logik. En utvidgning av *intuitionistisk typteori*[13] är det logiska system som finns i grunden till Coq[14]. Det går att se den som en bevisbarhetslogik. Skillnaden mellan den och klassisk logik kan illustreras genom följande exempel: I klassisk logik är det en logisk sanning att $A \vee \neg A$ gäller för alla satser A [2]. Så om A inte är sann måste $\neg A$ vara sann[15].

I intuitionistisk logik däremot låter vi "A är giltig" betyda "det finns ett bevis för A". Om A inte är giltig finns det alltså inget bevis för A. Men bara för att det inte finns något bevis för A betyder det inte att det istället nödvändigtvis finns ett bevis för $\neg A$. Detta betyder att vissa motsägelsebevis som är giltiga i klassisk logik inte kommer vara giltiga i intuitionistisk logik[16].

Om vi i klassisk logik har antagit att $\neg A$ gäller och visat att detta leder till en motsägelse så kan vi, eftersom då $A \vee \neg A$ är en logisk sanning måste minst en av A och $\neg A$ måste gälla, därmed dra slutsatsen att A gäller.

[2] Detta brukar kallas lagen om det uteslutna tredje.

3.2 Bevisassistenter

Ovan diskuteras vad som krävs för att ett datorsystem ska kunna kontrollera giltigheten i matematiska bevis. De tidigaste bevissystemen för datorer kunde endast göra detta så användaren var själv tvungen att skriva in varje enskilt logiskt steg i det formella beviset, medan datorsystemet bara passivt kontrollerade att dessa var giltiga.

Senare system ger användaren möjligheten att istället skriva ett *bevisskript*, som är en rad instruktioner till systemet som talar om hur det ska bygga upp det formella beviset, utan att användaren själv behöver mata in varje enskilt logiskt steg. Instruktionerna i bevisskriptet kan ha formen av *taktiker*, som säger till bevisassistenten vilken form av bevisstrategi som ska användas under olika steg i beviset. Till exempel kan användaren instruera bevisassistenten om att beviset ska göras med taktiken induktion över någon parameter i satsen.

Användaren kan skriva bevisskriptet interaktivt. Då ger användaren taktikinstruktionerna till systemet en i taget. Efter varje instruktion kontrollerar systemet om taktiken gick att genomföra, och efter varje genomförd taktik uppdaterar systemet informationen som visas för användaren om var i beviset man nu befinner sig och vad som återstår att bevisa. Bevisassistenten bygger på detta sätt steg för steg upp det formella beviset efter instruktionerna från användaren och kontrollerar efter hand att de ger upphov till ett korrekt bevis. Bevisassistenter kan utformas så att användaren kan ange att vissa steg i beviset ska lösas automatiskt eftersom de är så enkla att systemet själv kan hitta de härledningsregler och axiom som krävs för att visa dem.

Helt automatiserade bevismaskiner finns också. Då behöver användaren bara formulera ett antagande, och systemet söker sedan själv efter ett bevis för detta. De klarar dock ofta inte av att hitta komplicerade matematiska bevis inom skiljda matematiska områden[17][11].

Kapitel 4

Programmeringsspråket och bevisassistenten Coq

I det här avsnittet ges en introduktion till Coq. Coq är en interaktiv bevisassistent som innehåller ett beroendetypat funktionellt programmeringsspråk och kan användas till att utveckla formella bevis. Dessa två delar bygger på ett *grundspråk*, det vill säga det funktionella språket, som inte är helt olikt Haskell, samt ett *taktikspråk* som används för att skriva bevis. Program och bevis som är skrivna i Coq går att exportera till programspråken Haskell, OCaml och Scheme vilket gör att man kan skriva och bevisa de mest kritiska delarna i ett program i Coq och sedan exportera dem och köra dem tillsammans med icke bevisad kod.

4.1 Beroendetypning

För att förklara beroendetypning använder vi begreppen *termer* och *typer*. Förenklat sett kan man säga att termer är värden eller funktioner och typer är samlingar (matematiska mängder) av värden eller termer. Notera att varje term har en typ. Vi går nu gå igenom hur ett funktionellt programmeringsspråk med beroendetypning kan vara uppbyggt.

4.1.1 Termer som beror på termer

Den grundläggande idén med funktionell programmering är att *termer kan bero på termer*. I ett teoretiskt funktionellt språk skulle det kunna innebära att givet någon typ, låt oss använda de naturliga talen $\mathbb{N}$ i det här exemplet, skulle man kunna skriva

$$\begin{aligned} &\mathit{compose} : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \\ &\mathit{compose}\ f\ g\ x = f\ (g\ x) \end{aligned}$$

där termerna f, g och *compose* är funktioner och termen x är ett värde. Här säger man att termen *compose* beror på termerna f, g och x. Funktionen *compose* definierar funktionskomposition, oftast noterad med en cirkel som $(f \circ g)\ (x)$.

4.1.2 Termer som beror på typer

Utöver termer som beror på termer kan man även lägga till *termer som beror på typer*, vilket brukar kallas polymorfism. För att bygga vidare på vårt exemplet ovan skulle det kunna se ut så här:

$$compose' : (b \to c) \to (a \to b) \to a \to c$$
$$compose'\ f\ g\ x = f\ (g\ x)$$

Här har vi istället för att använda typen $\mathbb{N}$ använt de godtyckliga typerna a, b och c vilket innebär att termen $compose'$ beror på typerna a, b och c. Det låter oss med en enda definition av $compose'$ anropa funktionen med termer av olika typer, istället för att definiera *compose* för varje kombination av typer.

Ytterligare ett exempel på polymorfism är *polymorfiska typer*. Till exempel kan vi skriva

$$\textbf{\textit{data}}\ List\ a\ \textbf{\textit{where}}$$
$$Nil : List\ a$$
$$Cons : a \to List\ a \to List\ a$$

där $List\ a$ är en polymorfisk datatyp för listor. Vi skulle sedan kunna ge typen för en funktion som ger tillbaka det första elementet ur en lista:

$$head : List\ a \to a$$

4.1.3 Typer som beror på typer

Nästa steg i utvecklingen mot beroendetypning är att låta *typer bero på typer*. Med denna funktionalitet kan man definiera så kallade typoperatorer. Typoperatorer är funktioner som tar in typer som parametrar och ger ut typer som resultat, till skillnad från vanliga funktioner som tar in termer som parametrar och ger ut termer som resultat.

Ett exempel på en typoperator är den kartesiska produkten ($\times$). Typoperatorn $\times$ tar in två typer och ger ut en ny typ ofta kallad en produkttyp. Eftersom typer är mängder blir kartesiska produkten av två typer alla möjliga kombinationer av typernas element. Vi skulle även kunna använda oss av en typoperator som representerar union av disjunkta mängder ($+$). Unionen av två typer skulle ge oss en ny typ, ofta kallad en summatyp, som innehåller alla element från båda typerna.

4.1.4 Typer som beror på termer

Det sista steget i byggandet är att lägga till *typer som beror på termer*, även känt som beroendetypning. Beroendetypning innebär väldigt förenklat att man på typnivå kan använda samma funktioner och värden som man kan använda på termnivå. Till exempel låter detta oss definiera vektorer, det vill säga listor av en bestämd längd. Detta gör vi genom att lägga till en parameter som representerar vektorns längd i typen och öka denna med ett när vi lägger till ett element.

$$\textbf{\textit{data}}\ Vector\ a\ (n \in \mathbb{N})\ \textbf{\textit{where}}$$
$$Empty : Vector\ a\ 0$$
$$Element : a \to (m \in \mathbb{N}) \to Vector\ a\ m \to Vector\ a\ (m+1)$$

Sedan kan vi ge typen till funktioner som till exempel att slå ihop två vektorer:

$$concatenate : n \in \mathbb{N} \to m \in \mathbb{N} \to Vector\ a\ n \to Vector\ a\ m \to Vector\ a\ (n + m)$$

eller en funktion som tar ut första elementet på samma sätt som *head* för listor:

$$head' : (n \in \mathbb{N}) \to Vector\ a\ (n + 1) \to Vector\ a\ n$$

Notera att denna funktionen inte tillåter tomma vektorer som argument eftersom man inte kan ta bort ett element som inte finns, kompilatorn kontrollerar detta genom att matcha typen på den inskickade vektorn med värdet $n+1$. Om vektorn har en typ där värdet $n = 0$ matchar inte typen eftersom det inte finns $m \in \mathbb{N}$ sådant att $m + 1 = 0$. Detta till skillnad från definitionen av *head* för listor som helt enkelt inte kan definieras för tomma listor och i Haskell kraschar program som använder *head* på tomma listor.

Värt att notera är att inte bara funktionella språk har beroendetypade funktioner. Ett exempel på beroendetypning som de flesta inom datavetenskap är bekanta med är `printf` i C. Här bestäms antalet parametrar i funktionen av antalet % -tecken i den första strängen och vilken typ det ska vara på dessa parametrar bestäms av vilken bokstav som står efter respektive %-tecken.

```
printf("%s is %d years old and %f.1cm long", name, age, length)
```

4.2 Grundspråk

Grundspråket i Coq kallas Gallina och är som vi redan nämnt funktionellt och beroendetypat. Liksom i andra programmeringsspråk definierar man funktioner och deras typ. Det är dock inte tillåtet i Coq att definiera funktioner som inte terminerar. Det vill säga, i de flesta språk kan man skriva

```
loop :: Integer -> Integer
loop n = loop (n + 1)
```

och kompilatorn kommer att kompilera och låta dig köra funktionen. Men `loop` kommer aldrig ge något resultat och kommer snurra i all evighet. Kompilatorn till Coq tillåter inte denna typen av definitioner utan kommer att ge kompileringsfel om den stöter på liknande funktioner. För att Coq ska acceptera en rekursiv funktion krävs det att någon av parametrarna i det rekursiva anropet närmar sig basfallet eller så måste användaren ange ett bevis för att funktionen är garanterad att avsluta. Det går till exempel inte att implementera Collatz funktion som beskrivs i ekvation 4.1 även om den terminerar för alla hittills kända värden.

$$T(n) = \begin{cases} n/2, & \text{om } n \equiv 0 \pmod 2 \\ 3n + 1, & \text{om } n \equiv 1 \pmod 2 \end{cases} \tag{4.1}$$

För att göra en definition i Coq använder man nyckelordet `Definition`. En definition av funktionskomposition kan se ut så här:

```
Definition compose
  (a b c: Type)
  (f: b -> c)
  (g: a -> b)
  (x: a) : c := f (g x).
```

I definitionen av `compose` måste vi först ge typerna `a`, `b` och `c` explicit, till skillnad från i HASKELL där de är definierade implicit. Parametrarna `f`, `g` och `x` följer sedan direkt. I COQ används punkt (`.`) för att avsluta en definition likt hur man avslutar ett påstående (*engelska: statement*) i programmeringsspråket C med semikolon (;). För att anropa `compose` måste man ge typer, funktioner och värdet. Till exempel:

```
Definition add1 (x: nat) := x + 1.
Definition mul2 (x: nat) := x * 2.

Definition example : nat := compose nat nat nat add1 mul2 5.
```

Notera att programmerare är lata och vill skriva så lite som möjligt. I COQ kan man därför se till att typ parametrarna `a`, `b` och `c` automatiskt sätts in på rätt ställe i funktionsanropen. För att göra detta behöver man dock ändra på definitionen av `compose` och sätta klammerparenteser runt parametrarna istället:

```
Definition compose'
  {a b c: Type}
  (f: b -> c)
  (g: a -> b)
  (x: a) : c := f (g x).
```

Vi kan även använda så kallade lambdauttryck, vilket betyder funktioner definierade utan namn som används direkt där man skriver dem. Syntaxen för lambdauttryck är `fun parameters => application`. Med dessa förändringar kan vi skriva

```
Definition example' : nat :=
  compose' (fun x => x + 1) (fun x => x * 2) 5.
```

COQ har även datatyper och de definieras med hjälp av `Inductive`. En definition av naturliga tal kan se ut så här:

```
Inductive nat : Type :=
  | O: nat
  | S: nat -> nat.
```

Typen för varje konstruktor måste ges explicit. Vi kan även definiera vektorer, för att blanda in lite beroendetypning:

```
Inductive vec (X: Type) : nat -> Type :=
  | empty: vec X O
  | element: forall n, X -> vec X n -> vec X (S n).
```

Sedan kan vi definiera en typsäker version av `tail` med hjälp av mönstermatchning:

```
Definition tail (X: Type) (n: nat) (v: vec X (S n)) : vec X n :=
  match v with
  | element X _ xs => xs
  end.
```

Mönstermatchning innebär att man med hjälp av något värde tillhörande någon typ bestämmer vad som ska göras för varje konstruktor till typen. Notera i det här fallet att funktionen bara är definierad för vektorer `vec X n` där `n > 0`. Vi

behöver således inte hantera tomma vektorer i mönstermatchningen. Värt att notera är likheten mellan mönstermatchningen i Coq och `case` satser i Haskell.

För att definiera en rekursiv funktion använder vi `Fixpoint` istället för `Definition`:

```
Fixpoint concatenate
  (X: Type)
  {n m: nat}
  (a: vec X n)
  (b: vec X m) : vec X (n+m) :=
    match a with
    | empty => b
    | element _ x xs => element x (concatenate X xs b)
    end.
```

Detta var en mycket kort introduktion till Coq som funktionellt programmeringsspråk och har därför utelämnat bland annat modulhantering, typklasser, kanoniska strukturer, poster, partiell applicering och notationer. Dessa ämnen är också viktiga för att förstå Coq men är bortom målet med denna rapport.

4.3 Taktikspråk

Taktikspråk i Coq heter Ltac och är vad som används för att bygga de formella bevisen. Kort sagt skrivs bevis genom att sätta upp satsen som ska bevisas som uttryck. Sedan används så kallade taktiker för att modifiera uttrycket tills satsen är bevisad. Arbetsgången liknar på så sätt bevis med papper och penna, dock mycket mer detaljerat.

För att påbörja ett bevis skrivs först `Lemma` eller `Theorem` följt av namnet på beviset. Sedan skrivs kolon (:) följt av satsens kvantifikationer, det vill säga de objekt uttrycket beror på. Efter det följer ett kommatecken (,) och satsens påstående följt av punkt (.). Sedan är det tillåtet att skriva nyckelordet `Proof`, men det är bara syntaktiskt socker och betyder inget för kompilatorn. Till sist kommer beviset självt uppbyggt med hjälp av *taktiker* följt av `Qed.` för att avsluta beviset. För att bevisa till exempel en tautologi:

```
Theorem tautology : forall (A B C: Prop),
  (A -> B) -> (B -> C) -> A -> C.
Proof.
  intros A B C Hab Hbc Ha.
  apply Hbc.
  apply Hab.
  apply Ha.
Qed.
```

När satsen är definierad går systemet in i bevisläget. Då visar systemet ett så kallat kontext och en lista med mål som behöver bevisas. Med kontext menas de hypoteser som användaren kan använda sig av i beviset. Genom att använda sig av taktiker modifieras målen och kontexten. Efter `Proof.` i `tautology` ser kontexten och målet ut på följande sätt:

```
1 subgoals
______________________________________(1/1)
```

```
forall A B C : Prop, (A -> B) -> (B -> C) -> A -> C
```

Taktiken `intros` flyttar antaganden från målet till kontexten så att de kan användas i andra taktiker. I `tautology` flyttar vi upp alla implikationerna och ger dem namn, det leder till följande kontext och mål:

```
1 subgoals
A : Prop
B : Prop
C : Prop
Hab : A -> B
Hbc : B -> C
Ha : A
______________________________________(1/1)
C
```

Nu är målet att bevisa `C`, det kan vi göra genom att applicera hypotesen `Hbc` med `apply` taktiken. Det resulterar i att vi behöver bevisa `B` vilket vi kan göra med hypotesen `Hab`. Till sist får vi målet `A` vilket kan bevisas med hypotesen `Ha`. Kontexten visar då texten `No more subgoals` och beviset kan avslutas med `Qed`.

4.4 SSReflect

SSReflect är ett tillägg till Coq som utvecklats av bland annat Georges Gonthier. Namnet kommer från small-scale reflection (småskalig reflektion) vilket är en typ av bevismetodologi. Denna typ av bevismetodologi lämpar sig särskilt väl för att formalisera långa matematiska bevis. En central del i small-scale reflection är att arbeta med olika men ekvivalenta representationer, bland annat kan till exempel boolesk logik användas för att resonera kring och "räkna" med motsvarande propositioner, så kallad "boolean reflection".

SSReflect har ett stort bibliotek med matematiska satser som är formellt bevisade, bland annat finns stora delar av linjär algebra och viktiga resultat inom ändlig gruppteori formaliserade. Namnsystemet för satser är även utformat på ett systematiskt sätt så att det går smidigt att söka i biblioteket, med viss erfarenhet går det att gissa sig till namnet på en viss sats.

Vad gäller själva taktikspråket introducerar SSReflect bara tre nya taktiker men utökar funktionaliteten i ett antal redan existerande taktiker. Till exempel har omskrivningstaktiken `rewrite` utökats så att den erbjuder samma funktionalitet som en mängd separata taktiker i Coq. Detta överlapp mellan taktiker gör att flera gamla taktiker i Coq blir överflödiga. I stället har vi färre taktiker att hålla reda på i SSReflect. Utöver rent funktionella förändringar tillhandahåller SSReflect även verktyg för en bättre layout och struktur av bevisen.

En del definitioner är låsta i SSReflect, det gäller till exempel definitionerna av matriser och polynom. Låset på definitionerna är till för att undvika att expandera definitionerna när vi försöker förenkla uttryck, då det kan leda till tunga typchecknings-beräkningar som tar mycket lång tid. En konsekvens av att definitionerna är låsta är att vi inte kan utföra beräkningar med dem.

4.5 Introduktion till utvecklingsmiljön CoqIde

För närvarande finns det två olika utvecklingsmiljöer för Coq. Det är CoqIde som är en fristående utvecklingsmiljö och Proof General som är ett emacsläge. Vi väljer att endast presentera CoqIde då vårt projekt endast använder oss av denna utvecklingsmiljö.

4.5.1 Översikt av CoqIde

CoqIde är en interaktiv utvecklingsmiljö där användaren kan formulera bevis och skriva program samt att interaktivt bevisa dessa. I följande avsnitt förklaras de viktigaste delarna med hjälp av skärmdumpar med exempelkod.

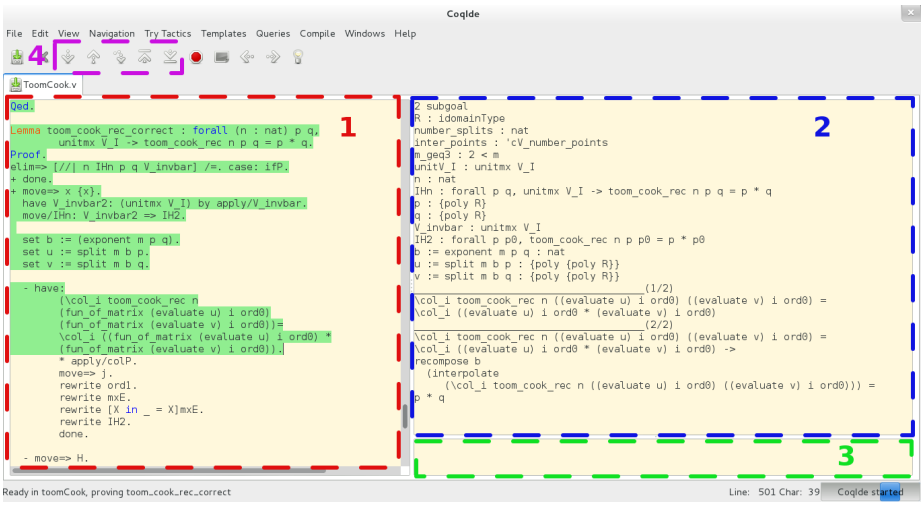
Figur 4.1 är en översikt av hela utvecklingsmiljön och dess olika delar

1. Textredigerare. I den här rutan skriver användaren sina program och bevis. Den grönmarkerade (skuggade om du inte har färgskrivare) koden har tolkats av systemet.

2. Textfönster för mål och kontext. I bevisläge visas här vilka mål som ska uppnås och vilka hypoteser som för närvarande finns i kontexten. Dessa uppdateras efter varje utförd taktik.

3. Textfönster för meddelanden. Här dyker felmeddelanden, svar på gjorda sökningar och övrig information upp.

4. Knappar för att stega framåt eller bakåt i koden. När vi stegar framåt tolkas koden som stegas förbi av systemet.

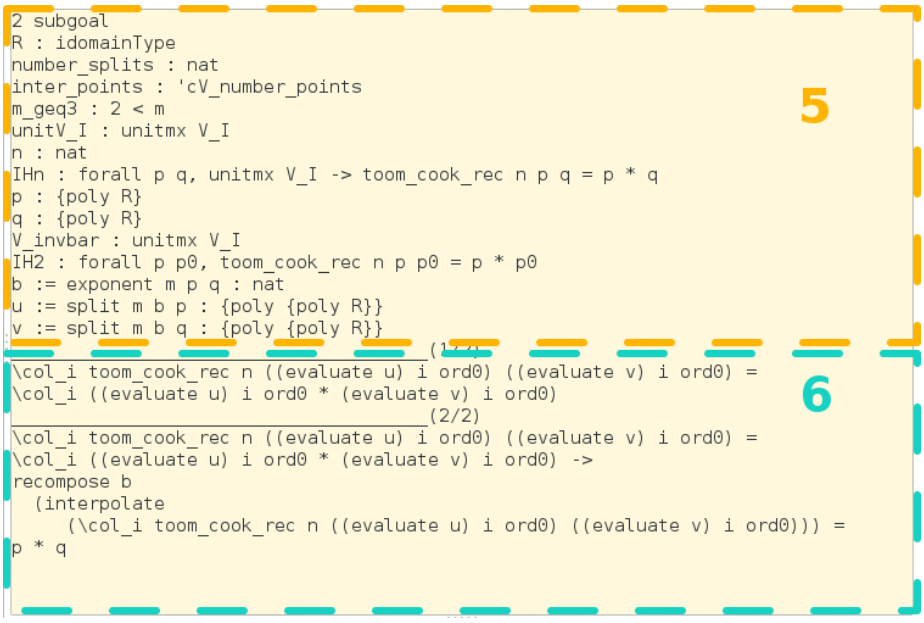


Figur 4.1: Översikt av de olika delarna i CoqIde

Figur 4.2 är en förstoring av ruta 2 i Figur 4.1

5. Kontext, här visas vilka hypoteser och variabler som vi för tillfället har i beviset.

6. Mål, här visas vilka mål som ska uppnås. Det översta målet är det som användaren arbetar med för tillfället och det är det målet som kommer att påverkas av nästkommande taktik.

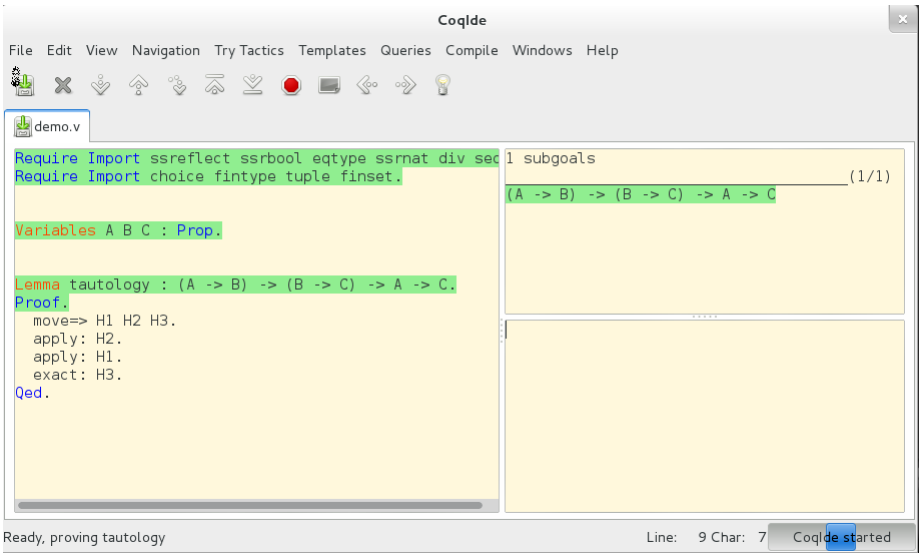


Figur 4.2: Textfönster för kontext och mål. Figuren visar en förstoring av ruta 2 i figur 4.1

4.5.2 Tolkning av kod

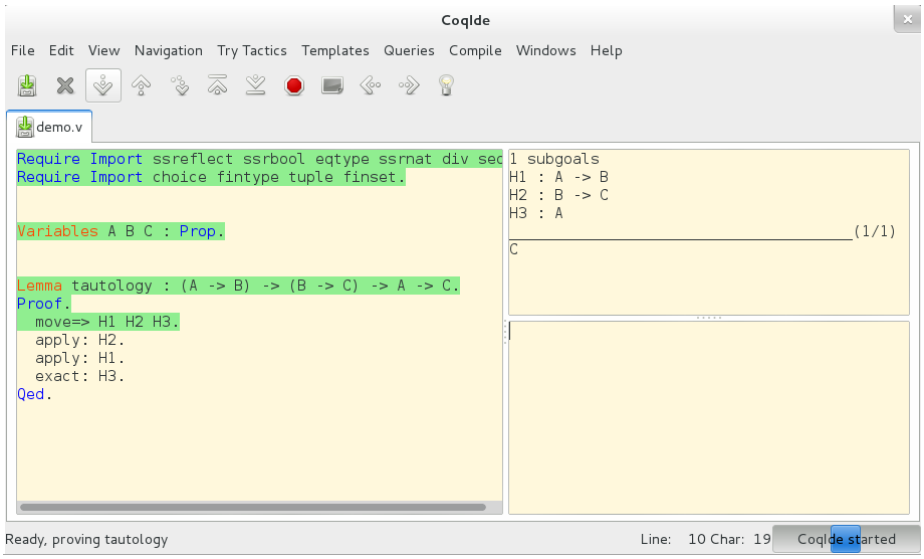
Koden tolkas en sats i taget och de delar av koden som har blivit tolkade markeras med en grön färg och det går inte längre att göra några ändringar i dessa. Om man skulle vilja göra en ändring får man stega tillbaka i programmet och göra ändringen. I följande sekvens av skärmdumpar visas några steg i ett bevis.

Figur 4.3 visar ett exempel på ett bevis för att följande påstående är en tautologi. En tautologi är en logisk sats som alltid är sann oberoende av sanningsvärdena hos delsatserna.

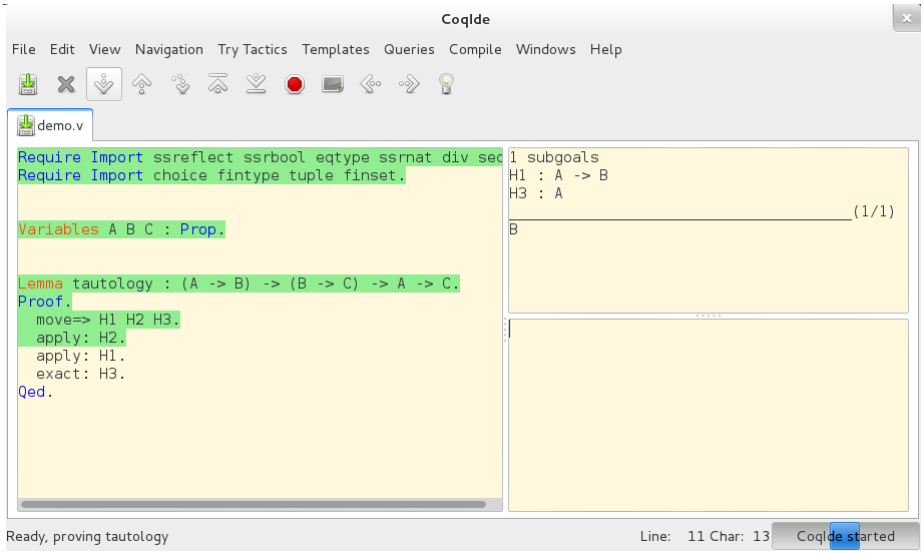


Figur 4.3: Exempel på ett bevis för en tautologi i Coq

Figurerna 4.4 och 4.5 visar den interaktiva biten mellan CoqIde och användaren. Notera hur mål och kontext uppdateras efter att en taktik används och att den taktik som har tolkats blir grönmarkerad.



Figur 4.4: Vi har nu flyttat hypoteserna $(A \to B)$, $(B \to C)$ och A från målet till kontexten med taktiken `move`



Figur 4.5: Vi har nu använt oss av hypotesen $(B \to C)$ och vi kan se att målet nu har ändrat sig från C till B

Kapitel 5

Toom-Cook-algoritmen

Toom-Cook är en algoritm för att multiplicera två polynom och är namngiven efter Andrei Toom och Stephen Cook.

Algoritmen är intressant eftersom den har en bättre asymptotisk tidskomplexitet än naiv[1] polynommultiplikation där man multiplicerar varje term i det ena polynomet med varje term i det andra polynomet vilket har tidskomplexiteten[2] $\mathcal{O}\left(n^2\right)$, där n är graden på det största av de två polynomen som skall multipliceras. Eftersom problemet att multiplicera två heltal kan reduceras till att multiplicera två polynom kan algoritmen även användas för heltalsmultiplikation. Denna reduktion kan göras utan att den asymptotiska tidskomplexiteten försämras. Detta innebär att man kan uppnå en bättre asymptotisk tidskomplexitet än den för naiv heltalsmultiplikation som lärs ut i grundskolan och har tidskomplexiteten $\mathcal{O}\left(n^2\right)$, där n är antalet siffror i det största talet.

Det finns flera varianter av Toom-Cook. Toom-Cook-m är en enskild instans av Toom-Cook som delar polynomen som skall multipliceras i m delar. Vanligtvis när man talar om Toom-Cook syftar man på Toom-Cook-3. Ett intressant specialfall av Toom-Cook är Toom-Cook-2 som under vissa förutsättningar svarar mot Karatsuba-algoritmen vilken beskrivs i kapitel 5.1.

Toom-Cook-m har tidskomplexiteten $\mathcal{O}\left(n^{\log_2(2m-1)/\log_2 m}\right)$ [18], där m är graden på det största polynomet. Konstanten som döljs av $\mathcal{O}(\cdot)$-notationen växer med m och har en betydande praktisk inverkan. För heltalsmultiplikation finns algoritmer som bygger på diskret Fouriertransform och har en ännu bättre asymptotisk tidskomplexitet, till exempel Schönhage-Strassen-algoritmen.

Både Toom-Cook och algoritmer som bygger på diskret Fouriertransform används i praktiken. I till exempel GMP, The GNU Multiple Precision Arithmetic Library, används Schönhage-Strassen-algoritmen samt olika instanser av Toom-Cook-algoritmen för multiplikation av heltal [19].

[1] Enligt definitionen i appendix A.1

[2] Funktionen $T(n)$ är $\mathcal{O}(f(n))$ om det existerar konstanter $c > 0$ och $n_0 \geq 0$ så att $T(n) \leq c \cdot f(n)$ för alla $n \geq n_0$. $\mathcal{O}(\cdot)$-notationen beskriver funktionens asymptotiska beteende, det vill säga hur funktionen växer med ökande storlek på argumentet.

5.1 Karatsuba

Karatsuba-algoritmen bygger på ett enkelt knep och visar tydligt varför vi kan förbättra den asymptotiska tidskomplexiteten jämfört med naiv polynommultiplikation. Säg att vi vill multiplicera polynomen p och q. Vi kan då dela upp polynomen p och q i två delar så att:

$$p(x) = p_0 + p_1 x^{\frac{n}{2}}$$
$$q(x) = q_0 + q_1 x^{\frac{n}{2}}$$

Där n är graden på det största polynomet. Multiplikationen kan då skrivas som:

$$\begin{aligned} p(x)q(x) &= (p_0 + p_1 x^{\frac{n}{2}})(q_0 + q_1 x^{\frac{n}{2}}) \\ &= p_0 q_0 + (q_0 p_1 + p_0 q_1) x^{\frac{n}{2}} + p_1 q_1 x^n \end{aligned}$$

Tricket i Karatsuba är följande enkla omskrivning som gör att vi blir av med en multiplikation så att vi istället för fyra multiplikationer endast behöver tre distinkta multiplikationer:

$$p_0 q_0 + ((p_1 + p_0)(q_1 + q_0) - p_1 q_1 - p_0 q_0) x^{\frac{n}{2}} + p_1 q_1 x^n$$

Tidskomplexiteten för Karatsuba-algoritmen är då:

$$T(n) = 3T(\lceil n/2 \rceil) + cn + d$$

där n är graden på det största polynomet och c och d är konstanter.

Genom att skriva ut rekursionsekvationen får vi att Karatsuba-algoritmen har den asymptotiska tidskomplexiteten $\mathcal{O}(n^{\log_2 3})$.

5.2 Definition av Toom-Cook

Det första steget i formaliseringen av Toom-Cook-algoritmen är en informell men detaljerad definition och bevis av algoritmen och att den är korrekt. Detta avsnitt är ägnat åt den. Versionen av algoritmen som presenteras bygger på den i [20] och presentationen av den följer källans. För en introduktion till integritetsområden och polynom, se appendix A.1.

Låt R vara ett integritetsområde och låt $p, q \in R[x]$, där

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n,$$
$$q(x) = b_0 + b_1 x + \cdots + b_s x^s$$

med $0 \leq s \leq n$. I detta avsnitt definierar vi $Toom-Cook-m\ (p, q)$, för naturliga tal $m \geq 2$, som resultatet av algoritmen nedan.

5.2.1 Gradkontroll

Om $\operatorname{grad} p = n \leq 2$, låt $Toom - Cook - m\ (p, q) = p \cdot q$, där $p \cdot q$ beräknas med naiv polynommultiplikation, annars gå vidare till steg 5.2.2.

5.2.2 Uppdelning

Låt

$$b = \left\lfloor \frac{1 + \operatorname{grad} p}{m} \right\rfloor + 1 = \left\lfloor \frac{1+n}{m} \right\rfloor + 1. \tag{5.1}$$

För $f \in R[x]$ och

$$f = qx^k + r \tag{5.2}$$

med $q, r \in R[x]$ och $r = 0$ eller $\operatorname{grad} r \leq \operatorname{grad} x^k$, låt

$$f/x^k = q \tag{5.3}$$

$$f \bmod x^k = r. \tag{5.4}$$

Nu definierar vi $u, v \in (R[x])\,[y]$. Låt

$$u(y) = u_0 + u_1 y + \cdots + u_{m-1} y^{m-1} \tag{5.5}$$

där

$$u_k = \left(p/x^{bk}\right) \bmod x^b \tag{5.6}$$

och

$$v(y) = v_0 + v_1 y + \cdots + v_{m-1} y^{m-1} \tag{5.7}$$

där

$$v_k = \left(q/x^{bk}\right) \bmod x^b \tag{5.8}$$

Vi vill alltså att $u(x^b) = p(x)$ och $v(x^b) = q(x)$.

5.2.3 Evaluering

Nu ska vi beräkna $w = u \cdot v$. Vi kommer göra detta genom interpolation i steg 5.2.5. Eftersom

$$\operatorname{grad} w = \operatorname{grad} u + \operatorname{grad} v \leq m - 1 + m - 1 = 2m - 2 = d$$

kan koefficienterna i w bestämmas om vi vet värdet av w i $d+1$ skiljda punkter. Vi evaluerar w i punkterna $\alpha_0, \ldots, \alpha_d$, där $\alpha_i \in R$. För att göra det beräknar vi i detta steg först $u(\alpha_i)$ och $v(\alpha_i)$ genom matrismultiplikation. I nästa steg beräknas sedan $w(\alpha_i) = u(\alpha_i) \cdot v(\alpha_i)$. Låt

$$V_e = \begin{pmatrix} \alpha_0^0 & \alpha_0^1 & \cdots & \alpha_0^{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_d^0 & \alpha_d^1 & \cdots & \alpha_d^{m-1} \end{pmatrix}. \tag{5.9}$$

Då får vi

$$V_e \cdot \begin{pmatrix} u_0 \\ \vdots \\ u_{m-1} \end{pmatrix} = \begin{pmatrix} u(\alpha_0) \\ \vdots \\ u(\alpha_d) \end{pmatrix} \tag{5.10}$$

och motsvarande för v.

5.2.4 Rekursiv multiplikation

Vi beräknar $w(\alpha_i) = u(\alpha_i) \cdot v(\alpha_i)$ för i $= 0, \ldots, d$ rekursivt genom att anropa Toom-Cook med $u(\alpha_i)$ och $v(\alpha_i)$ som argument. Notera att $u(\alpha_i)$, $v(\alpha_i) \in R[x]$.

5.2.5 Interpolation

Vi bestämmer koefficienterna i $w(y) = w_0 + w_1 y + \ldots + w_d y^d$ genom interpolation. Vi vet värdet av $w(y) = u(y) \cdot v(y)$ i $d+1$ punkter. Om

$$V_I = \begin{pmatrix} \alpha_0^0 & \alpha_0^1 & \cdots & \alpha_0^d \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_d^0 & \alpha_d^1 & \cdots & \alpha_d^d \end{pmatrix} \tag{5.11}$$

så är

$$V_I \cdot \begin{pmatrix} w_0 \\ \vdots \\ w_d \end{pmatrix} = \begin{pmatrix} w(\alpha_0) \\ \vdots \\ w(\alpha_d) \end{pmatrix} \tag{5.12}$$

och därmed är

$$\begin{pmatrix} w_0 \\ \vdots \\ w_d \end{pmatrix} = V_I^{-1} \cdot \begin{pmatrix} w(\alpha_0) \\ \vdots \\ w(\alpha_d) \end{pmatrix} \tag{5.13}$$

förutsatt att V_I är inverterbar. Detta är den om determinanten till V_I är ett inverterbart element i R[21]. Eftersom V_I är en Vandermondematris så är

$$\det V_I = \prod_{0 \le i < j \le d} (\alpha_i - \alpha_j) \tag{5.14}$$

5.2.6 Sammansättning

Vi får slutligen $p(x) \cdot q(x)$ genom att evaluera w i x^b eftersom $w(x^b) = u(x^b) \cdot v(x^b)$.

5.3 Exempel på Toom-3

Antag att vi vill multiplicera två polynom $p, q \in \mathbb{Z}_5[x]$[3]. Vi låter 0, 1, 2, 3 och 4 beteckna elementen i $\mathbb{Z}_5[x]$ och vi antar

$$p(x) = 4 + x + 3x^2 + 3x^3 + 4x^4$$
$$q(x) = 1 + 2x + 2x^2 + x^3 + 3x^4$$

[3] $\mathbb{Z}_5$ betecknar heltalen modulo 5.

Uppdelning Vårt b i uppdelningssteget, som säger i hur stora delar vi ska dela upp polynomen, ges av

$$b = \left\lfloor \frac{1 + \text{grad p}}{\text{Toom-Cook-instans}} \right\rfloor + 1 = \left\lfloor \frac{1+4}{3} \right\rfloor + 1 = 2$$

Alltså delar vi polynomen vid potenser av x som är multipel av 2 och låter delarna vara koefficienter i polynomen u och v:

$$u(y) = (4 + x) + (3 + 3x)y + 4y^2$$
$$v(y) = (1 + 2x) + (2 + x)y + 3y^2$$

Evaluering Om vi väljer evalueringspunkterna

$$0, 1, 2, 3, 4$$

så blir evalueringsmatrisen

$$V_e = \begin{pmatrix} 0^0 & 0^1 & 0^2 \\ 1^0 & 1^1 & 1^2 \\ 2^0 & 2^1 & 2^2 \\ 3^0 & 3^1 & 3^2 \\ 4^0 & 4^1 & 4^2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 4 \\ 1 & 4 & 1 \end{pmatrix}$$

Genom att multiplicera evalueringsmatrisen med vektorerna av koefficienter för våra polynom u och v får vi vektorer med polynomens värden i evalueringspunkterna.

$$\begin{pmatrix} u(0) \\ u(1) \\ u(2) \\ u(3) \\ u(4) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 4 \\ 1 & 4 & 1 \end{pmatrix} \begin{pmatrix} 4 + x \\ 3 + 3x \\ 4 \end{pmatrix} = \begin{pmatrix} 4 + x \\ 1 + 4x \\ 1 + 2x \\ 4 \\ 3x \end{pmatrix}$$

$$\begin{pmatrix} v(0) \\ v(1) \\ v(2) \\ v(3) \\ v(4) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 4 \\ 1 & 4 & 1 \end{pmatrix} \begin{pmatrix} 1 + 2x \\ 2 + x \\ 3 \end{pmatrix} = \begin{pmatrix} 1 + 2x \\ 1 + 3x \\ 2 + 4x \\ 4 \\ 2 + x \end{pmatrix}$$

Rekursiv multiplikation Genom att multiplicera u och v i de evaluerade punkterna rekursivt får vi värdet i evalueringspunkterna av $u \cdot v = w$, där w är det polynom som skall interpoleras fram

$$\begin{pmatrix} w(0) \\ w(1) \\ w(2) \\ w(3) \\ w(4) \end{pmatrix} = \begin{pmatrix} u(0) \cdot v(0) \\ u(1) \cdot v(1) \\ u(2) \cdot v(2) \\ u(3) \cdot v(3) \\ u(4) \cdot v(4) \end{pmatrix} = \begin{pmatrix} 4 + 4x + 2x^2 \\ 1 + 2x + 2x^2 \\ 2 + 3x + 3x^2 \\ 1 \\ x + 3x^2 \end{pmatrix}$$

Interpolation Vi vet att

$$V_I = \begin{pmatrix} 0^0 & 0^1 & 0^2 & 0^3 & 0^4 \\ 1^0 & 1^1 & 1^2 & 1^3 & 1^4 \\ 2^0 & 2^1 & 2^2 & 2^3 & 2^4 \\ 3^0 & 3^1 & 3^2 & 3^3 & 3^4 \\ 4^0 & 4^1 & 4^2 & 4^3 & 4^4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 3 & 1 \\ 1 & 3 & 4 & 2 & 1 \\ 1 & 4 & 1 & 4 & 1 \end{pmatrix}$$

och

$$V_I \cdot \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix} = \begin{pmatrix} w(0) \\ w(1) \\ w(2) \\ w(3) \\ w(4) \end{pmatrix}$$

där $w_0, \ldots, w_4$ är koefficienterna i polynomet w. Determinanten till en Vandermondematris kan räknas ut på följande sätt

$$\begin{aligned} \det V_I &= (0-1)(0-2)(0-3)(0-4)(1-2)(1-3)(1-4)(2-3)(2-4)(3-4) \\ &= 3 \\ &\neq 0, \end{aligned}$$

så V_I är inverterbar eftersom varje nollskiljt element i $\mathbb{Z}_5[x]$ har en multiplikativ invers. Vi får då koefficienterna till w genom att

$$\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix} = V_I^{-1} \cdot \begin{pmatrix} w(0) \\ w(1) \\ w(2) \\ w(3) \\ w(4) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 4 & 2 & 3 & 1 \\ 0 & 4 & 1 & 1 & 4 \\ 0 & 4 & 3 & 2 & 1 \\ 4 & 4 & 4 & 4 & 4 \end{pmatrix} \begin{pmatrix} w(0) \\ w(1) \\ w(2) \\ w(3) \\ w(4) \end{pmatrix} =$$

$$= \begin{pmatrix} 4 + 4x + 2x^2 \\ 1 + 2x^2 \\ 2 + 3x^2 \\ 2 + 3x \\ 2 \end{pmatrix}$$

Då har vi

$$w(y) = (4 + 4x + 2x^2) + (1 + 2x^2)y + (2 + 3x^2)y^2 + (2 + 3x)y^3 + 2y^4$$

Sammansättning Produkten $p(x) \cdot q(x)$ får vi slutligen genom att evaluera w i x^b, alltså i x^2.

$$\begin{aligned} w(x^2) &= (4 + 4x + 2x^2) + (1 + 2x^2)x^2 + (2 + 3x^2)(x^2)^2 + (2 + 3x)(x^2)^3 + 2(x^2)^4 \\ &= 4 + 4x + 2x^2 + x^2 + 2x^4 + 2x^4 + 3x^6 + 2x^6 + 3x^7 + 2x^8 \\ &= 4 + 4x + 3x^2 + 4x^4 + 3x^7 + 2x^8 \end{aligned}$$

5.4 Informellt bevis av Toom-Cook

I detta stycke visar vi att för $p, q \in R[x]$ så är $p \cdot q = Toom - Cook - m\ (p, q)$. Vi gör detta med hjälp av två lemman, vars bevis kommer efter beviset av 5.4.1.

Proposition 5.4.1. *Antag att R är ett integritetsområde och att $p, q \in R[x]$. Om det finns $\alpha_0, \ldots, \alpha_{2m-2} \in R$ så att $\prod_{0 \leq i < j \leq d}(\alpha_i - \alpha_j)$ är inverterbar i R, så är med dessa punkter som interpolationspunkter*

$$Toom - Cook - m\ (p, q) = p \cdot q. \tag{5.15}$$

Bevis. Vi visar propositionen med induktion över $\operatorname{grad} p$.

Basfall. När $n = \operatorname{grad} p \leq 2$ så gäller (5.15) enligt steg 5.2.1 i algoritmen eftersom vi räknar ut produkten direkt.

Induktionssteg. Antag att $p, q \in R[x]$, där

$$\begin{aligned} p(x) &= a_0 + a_1 x + \cdots + a_n x^n, \\ q(x) &= b_0 + b_1 x + \cdots + b_s x^s \end{aligned}$$

med $0 \leq s \leq n$.

Antag också att $n > 2$ och att $Toom - Cook - m\ (f, g) = f \cdot g$ gäller för polynom av grad mindre än n. Eftersom $n > 2$ så går vi vidare till steg 5.2.1 i algoritmen och skapar u och v från p och q. När detta är gjort evaluerar vi i steg 3 u och v i punkterna $\alpha_0, \ldots, \alpha_{2m-2}$ genom att multiplicera vektorn av deras koefficienter med evalueringsmatrisen. I steg 5.2.4 anropar vi Toom-Cook igen, nu med argumenten $u(\alpha_i) \cdot v(\alpha_i)$ för $i = 0, \ldots, 2m-2$. Dessa är polynom i $R[x]$. Eftersom $\operatorname{grad} u(\alpha_i), \operatorname{grad} v(\alpha_i) < n$ enligt lemma 5.4.2 så är $Toom - Cook - m\ (u(\alpha_i), v(\alpha_i)) = u(\alpha_i) \cdot v(\alpha_i)$ enligt induktionsantagandet. I steg 5.2.5 skall vi bestämma koefficienterna i $w(y) = u(y) \cdot v(y)$. Detta gör vi genom att lösa matrisekvationen (5.12). Eftersom interpolationsmatrisen V_I enligt antagande är inverterbar så ges koefficienterna entydigt av (5.13). I steg 5.2.6 evaluerar vi $w(y) = u(y) \cdot v(y)$ i x^b. Lemma 5.4.3 ger att $u(x^b) = p(x)$ och att $v(x^b) = q(x)$. Då $w(y) = u(y) \cdot v(y)$ så är $w(x^b) = u(x^b) \cdot v(x^b) = p(x) \cdot q(x)$. □

Lemma 5.4.2. *Antag att $p(x) \in R[x]$ och att $\operatorname{grad} p \geq 2$. Antag också att $u(y)$ är definierad enligt steg 5.2.2, uppdelning, i algoritmen och att $\alpha \in R$. Då är $\operatorname{grad} u(\alpha) < \operatorname{grad} p$.*

Bevis. Eftersom $\alpha \in R$ så är

$$\operatorname{grad} u(\alpha) = \operatorname{grad}(u_0 + u_1\alpha + \cdots + u_{m-1}\alpha^{m-1}) \leq \max \operatorname{grad} u_k,$$

där $k = 0, 1, \ldots, m-1$.

$u_k = p(x)/x^{kb} \bmod x^b$, så enligt definition av mod så är $\operatorname{grad} u_k < b$. Nu återstår bara att visa att $b \leq \operatorname{grad} p = n$. Vi har definierat $b = \left\lfloor \frac{1+n}{m} \right\rfloor + 1$. Om $n \geq 3$ och $m \geq 2$ så är

$$n - b = n - \left(\left\lfloor \frac{1+n}{m} \right\rfloor + 1 \right) \geq \frac{nm - n - m - 1}{m} \geq 0,$$

så $\operatorname{grad} u(\alpha) < b \leq \operatorname{grad} p(x) = n$. □

Lemma 5.4.3. *Antag att $p(x) \in R[x]$ och att b och $u(y)$ är definierade som i steg 5.2.2 i algoritmen. Då är $u(x^b) = p(x)$.*

Bevis. Vi har att

$$\begin{aligned} u(x^b) &= \sum_{i=0}^{m-1} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} \\ &= \sum_{i=0}^{m-2} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + p(x)/x^{b(m-1)} (\mathrm{mod}\, x^b) x^{b(m-1)}. \end{aligned}$$

Påstående 1. Den sista termen i $u(x^b)$ kan skrivas om till $p(x)/x^{b(m-1)} x^{b(m-1)}$ eftersom $\mathrm{grad}\, p - b(m-1) < b$.

Med $\mathrm{grad}\, p = n$ och $b = \left\lfloor \frac{1+n}{m} \right\rfloor + 1$ har vi att

$$\begin{aligned} b - (n - b(m-1)) &= mb - n \\ &= m\left(\left\lfloor \frac{1+n}{m} \right\rfloor + 1 \right) - n \\ &= m\left(\frac{1+n}{m} - \left\{ \frac{1+n}{m} \right\} \right) + m - n \\ &= 1 + n - m\left\{ \frac{1+n}{m} \right\} + m - n \\ &= 1 + m\left(1 - \left\{ \frac{1+n}{m} \right\} \right) > 0 \end{aligned}$$

eftersom $0 < 1 - \{x\} \leq 1$ för alla reella tal [4] x. Graden av $p(x)/x^{b(m-1)}$ är $\max\{n - b(m-1), 0\}$ som är mindre än b, och därmed är $p(x)/x^{b(m-1)} \bmod x^b = p(x)/x^{b(m-1)}$, vilket visar Påstående 1. Så

$$u(x^b) = \sum_{i=0}^{m-2} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + p(x)/x^{b(m-1)} x^{b(m-1)}.$$

För alla polynom $a(x)$ gäller att

$$a(x) = a(x) \bmod x^b + \left(a(x)/x^b \right) x^b \tag{5.16}$$

då $a(x) \bmod x^b$ och $p(x)/x^b$ är resten respektive kvoten vid division med x^b. Så om vi kan visa att $u(x^b) = p(x) \bmod x^b + (p(x)/x^b) x^b$ så är $u(x^b) = p(x)$. Och om sedan

$$\sum_{i=0}^{k} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + (p(x)/x^{b(k+1)}) x^{b(k+1)} = \tag{5.17}$$

$$\sum_{i=0}^{k-1} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + (p(x)/x^{bk}) x^{bk} \tag{5.18}$$

för $k \geq 1$ så ger induktion över k att $u(x^b) = p(x)$. Nu återstår alltså bara att visa att (5.17) = (5.18).

[4] $\{x\}$ betecknar $x - \lfloor x \rfloor$. Se [22].

$$\sum_{i=0}^{k} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + p(x)/x^{b(k+1)} x^{b(k+1)} =$$

$$\sum_{i=0}^{k-1} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + \left(p(x)/x^{bk} \bmod x^b\right) x^{bk} + \left(p(x)/x^{b(k+1)}\right) x^{b(k+1)} =$$

$$\sum_{i=0}^{k-1} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + \left(p(x)/x^{bk} \bmod x^b + \left(p(x)/x^{b(k+1)}\right) x^b\right) x^{bk} =$$

$$\sum_{i=0}^{k-1} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + (p(x)/x^{bk} \bmod x^b + ((p(x)/x^{bk})/x^b) x^b) x^{bk} =$$

(på grund av (5.16))

$$\sum_{i=0}^{k-1} p(x)/x^{bi} (\mathrm{mod}\, x^b) x^{bi} + (p(x)/x^{bk}) x^{bk}$$

□

Kapitel 6

Formell implementation av Toom-Cook

Här beskrivs det huvudsakliga resultatet i projektet: den formella implementationen och beviset av Toom-Cook i Coq med SSReflect. Den fungerar också som en illustration på hur definitioner och bevis ser ut och fungerar i Coq med SSReflect. De viktigaste funktionerna och lemmana till algoritmen presenteras. Den fullständiga källkoden finns i appendix B.2.

6.1 Implementation av Toom-Cook i Coq med SSReflect

I det här avsnittet förklaras hur Toom-Cook-algoritmen är implementerad i SSReflect och hur den överensstämmer med den informella definitionen av algoritmen.

6.1.1 Inledande definitioner och funktioner

Algoritmen är implementerad med en rad små funktioner, en rekursiv del och sedan en huvudfunktion. Till en början kommer alltså många variabler, definitioner och antaganden radas upp, som vi har försökt att ge så självförklarande namn som möjligt.

```
Section toomCook.
Variable R : idomainType.
Implicit Types p q : {poly R}.
```

Först öppnas en ny sektion, inom vilken vi kan låta `R` beteckna ett godtyckligt men fixt integritetsområde och att variabelnamnen `p` och `q` alltid kommer beteckna polynom i detta område, så vi inte behöver ange typen på `p` och `q` när vi använder de namnen.

```
Variable number_splits : nat.
Definition m : nat := number_splits.
Definition number_points := (2 * m) .-1.
```

Sedan parametriseras funktionen på `m` genom att ange den som ett godtyckligt men fixt naturligt tal. `m` är den parameter som precis som i den informella algoritmen anger vilken Toom-Cook-instans vi har. Det gör att vi inom sektionen inte explicit behöver ge `m` som argument till varje funktion vi definierar, men att vi ändå kan använda det i funktionerna. Vi definierar också `number_points` som det antal interpolationspunkter vi behöver för Toom-Cook-m .

```
Variable inter_points : 'cV[{poly R}]_(number_points).
```

Vi låter `inter_points` beteckna en kolonnvektor vars element tillhör $R[x]$. Detta ska vara en vektor som innehåller interpolationspunkterna. I den informella algoritmen låter vi interpolationspunkterna tillhöra R. Det gjorde det lättare att visa att graden på polynomen skulle minska vid rekursiva anrop av Toom-Cook, vilket krävdes för induktionsantagandet som var över graden på polynomen, se avsnitt 5.4. Det kommer dock visa sig längre ner att det i formella fallet är lämpligare att definiera interpolationspunkterna som element i $R[x]$. För att algoritmen ska vara tillräckligt snabb ska de dock vara konstanta polynom (alltså egentligen element i R) eller av låg grad.

```
Hypothesis m_neq_0 : 0 < m.
```

I den informella algoritmen har vi som antagande att `m` är större än 2. Den rekursiva delen av formella algoritmen kräver bara att `m` är större än 0. Algoritmen kommer dock inte fungera som Toom-Cook med `m` mindre än 2.

```
Definition V_e : 'M[{poly R}]_(number_points, m) :=
  \matrix_(i < number_points, j < m) (inter_points i 0)^+j.
```

Här definieras evalueringsmatrisen `V_e` som en `number_points` $\times$ `m`-matris med element som tillhör $R[x]$. `\matrix_(i < number_points, j < m)(inter_points i 0)^+j` säger att matrisen på plats `(i,j)`, med `(i < number_points)` och `(j < m)` ska ha `i`:te elementet ur kolonnvektorn `inter_points` upphöjt till `j`.

```
Definition V_I : 'M[{poly R}]_(number_points) :=
  \matrix_(i < number_points, j < number_points)
    (inter_points i 0)^+j.
```

Här definieras interpolationsmatrisen på motsvarade sätt som evalueringsmatrisen, skillnaden är att det blir en matris av dimension `number_points` $\times$ `number_points` istället.

```
Definition exponent (m: nat) p q : nat :=
  (maxn (divn (size p) m) (divn (size q) m)).+1.
```

Här definieras funktionen `exponent`. Den motsvarar det b som definieras i ekvation (5.1) i steg 5.2.2 om den anropas med m:et i Toom-Cook-m. `size p` tar graden av polynomet `p` + 1, `divn` är exakt division för naturliga tal och `maxn` ger det största av två naturliga tal. Funktionerna `size`, `divn` och `maxn` är implementerade i SSReflect-biblioteket. Beteckningen `k.+1` står för efterföljaren till `k`.

```
Definition split (n b: nat) p : {poly {poly R}} :=
  \poly_(i < n) rmodp (rdivp p 'X^(i * b)) 'X^b.
```

`split` tar två naturliga tal och ett polynom som inparametrar och returnerar ett polynom i $(R[x])[y]$. `rmodp` och `rdivp` ger oss kvoten och resten vid polynomdivision se ekvationerna (5.4) och (5.3). Funktionerna `\poly_(i<n)`, `rmodp`

och `rdivp` bildar tillsammans motsvarande polynom $u(y) = (p/x^0 \ mod \ x^b)y^0 + \cdots + (p/x^{(n-1)b} \ mod \ x^b)y^{n-1}$ där `n`, `b` och `p` är inparametrarna.

```
Definition evaluate (u: {poly {poly R}}) :
  'cV[{poly R}]_(number_points) := V_e *m (poly_rV u)^T.

Definition interpolate (u: 'cV[{poly R}]_(number_points)) :
  {poly {poly R}} := rVpoly (invmx V_I *m u)^T.
```

Här definieras själva matrismultiplikationerna, se ekvation (5.10) och (5.13). `evaluate` ger tillbaka en vektor med ett polynoms värden i interpolationspunkterna. Funktionen `poly_rV` ger tillbaka en radvektor med ett polynoms koefficienter (motsvarande funktion finns inte implementerad i SSReflect för kolonnvektorer, och för att kunna använda resultat från biblioteket i bevisen använder vi den istället för att definiera en egen kolonnversion). `^T` transponerar radvektorn för att kunna multiplicera den med evalueringsmatrisen.

Matrismultiplikation är bara definierat i SSReflect mellan matriser och vektorer av samma typ. Eftersom polynomet `u` som ska evalueras har koefficienter i $R[x]$, så att vektorn av dess koefficienter ligger i $R[x]$, måste matrisens element också ligga i $R[x]$. Det är därför som vi låter interpolationspunkterna ligga i $R[x]$ här istället för i R. Det hade gått att lösa det på andra sätt, men det hade gjort bevisen för `evaluate` och särskilt `interpolate` mycket mer komplicerade.

```
Definition recompose (b: nat) (w: {poly {poly R}})
  : {poly R} := w.['X^b].
```

Här sker evalueringen av `w` i x^b där `b` är en inparameter.

6.1.2 Den rekursiva funktionen och huvudfunktionen

Nu har hjälpfunktionerna till algoritmen definierats. Nu definieras själva algoritmen. Det görs med hjälp av två funktioner. En rekursiv del (som i Coq definieras med kommandot `Fixpoint`) som egentligen gör allt jobbet, och en yttre del som bara anropar den rekursiva delen med ett visst argument. Anledningen till detta är att Coq bara tillåter rekursiva funktioner som säkert terminerar. Antingen behöver Coq lätt förstå att något av argumenten i funktionen minskar mot ett basfall med varje rekursivt anrop, eller så får användaren själv bevisa att det hon eller han vill ska vara rekursionsparametern, som i det här fallet är graden på polynomen, minskar.

Ett sätt att komma runt detta är att definiera funktionen med ett extra argument, i det här fallet `n`, ett naturligt tal vars enda uppgift är att minskas med 1 för varje rekursivt anrop av funktionen. Dessutom definierar man två basfall för rekursionen. Ett för när det extra argumentet `n` når 0, och ett för den verkliga rekursionsparametern. Då godkänner Coq funktionen, och användaren slipper göra ett krångligt bevis för att rekursionsparametern avtar.

Sedan definierar man en yttre funktion, som anropar den rekursiva funktionen och ger det extra argumentet värdet av den riktiga rekursionsparametern.

I `toom_cook_rec` är basfallen `n = 0` och graden av något av polynomen mindre än 2. Då utförs naiv polynommultiplikation. Falluppdelningen mellan basfall och rekursionsfall för extraargumentet `n` görs med en `match`-sats. Sedan görs falluppdelning mellan basfall och rekursionsfall för den riktiga rekursionsparame-

tern med en `if ... then ... else`-sats. Nedan beskrivs stegen i funktionen ett och ett.

```
Fixpoint toom_cook_rec (n: nat) p q : {poly R} :=
  match n with
  | 0%N => p * q
  | n'.+1 => if (size p <= 2) || (size q <= 2) then p * q else
        let b := exponent m p q in
        let u := split m b p in
        let v := split m b q in
        let u_a := evaluate u in
        let v_a := evaluate v in
        let w_a := \col_i toom_cook_rec n' (u_a i 0) (v_a i 0) in
        let w := interpolate w_a
         in recompose b w
  end.
```

```
| 0%N => p * q
```

Om `n` är lika med 0 så sker direkt multiplikation mellan polynomen.

```
| n'.+1 => if (size p <= 2) || (size q <= 2) then p * q else
```

Om `n` är `n'.+1`, det vill säga en efterföljare till något naturligt tal `n'`, med andra ord större än 0, så kontrolleras graden på polynomen. Det motsvarar steg 5.2.1 i den informella algoritmen. Om graden på något av polynomen är mindre än 2 multipliceras de direkt. Annars går vi vidare i algoritmen.

```
let b := exponent m p q in
```

Här låter vi `b` få värdet från funktionen `exponent` vilket som nämnts ovan är samma värde som variabeln b får i den informella algoritmen i ekvation (5.1), eftersom `m` är m:et i Toom-Cook-m och `p` och `q` är polynomen som ska multipliceras.

```
let u := split m b p in
let v := split m b q in
```

Härnäst definierar vi som i den informella algoritmen `u` och `v`, som fås från `p` och `q` genom uppdelningen i ekvationerna (5.5) och (5.7) i steg 5.2.2.

```
let u_a := evaluate u in
let v_a := evaluate v in
```

Här sker motsvarigheten till steg 5.2.3, evalueringen av `u` och `v` i interpolationspunkterna. `u_a` och `v_a` kommer vara kolonnvektorer.

```
let w_a := \col_i toom_cook_rec n' (u_a i 0) (v_a i 0) in
```

Det här motsvarar steg 5.2.4 i den informella algoritmen, den rekursiva multiplikationen. `w_a` är en vektor som på plats `i` har resultatet av att anropa `toom_cook_rec` igen, nu med `n'` som inparameter, vilken har värdet n-1, och `i`:te elementet i `u_a` och `v_a`. (`(u_a i 0)` och `(v_a i 0)` är `u` och `v` evaluerade i den `i`:te interpolationspunkten.)

```
let w := interpolate w_a
in recompose b w
```

När rekursionssteget är klart så sker interpolationssteget som motsvarar steg 5.2.5, där inversen av interpolationsmatrisen multipliceras med kolonnvektorn `w_a` och returnerar polynomet `w` som är produkten av polynomen u och v. Till sist sker sammansättningen som motsvarar steg 5.2.6, där polynomet `w` evalueras i x^b, och därmed har vi fått produkten av polynomen `p` och `q`.

Slutligen definieras den yttre funktionen `toom_cook`, där `toom_cook_rec` anropas med maximum av graden av polynomen plus 1. När `toom_cook_rec` anropas med detta tal eller något högre tal kommer den fungera som Toom-Cook gör i den informella definitionen i avsnitt 5.2.

```
Definition toom_cook p q : {poly R} :=
  toom_cook_rec (maxn (size p) (size q)) p q.
```

6.2 Formellt bevis av Toom-Cook

Det här avsnittet presenterar det formella beviset för att den implementerade algoritmen är korrekt. Många av detaljerna i beviset är rent tekniska och förklaras inte närmare. Jämförelser görs med det informella beviset i avsnitt 5.4. De viktigaste taktikerna som används förklaras under bevisets gång. Variablerna `p` och `q` betecknar genomgående polynom över ett integritetsområde.

I beviset används, när så är möjligt, redan tidigare bevisade resultat från SSReflects bibliotek. En lång rad lemman om bland annat grundläggande egenskaper hos polynom, naturliga tal, matriser och summor finns tillgängliga.

Det formella bevisets struktur är modellerad efter det informella bevisets i avsnitt 5.4, med ett huvudbevis som bygger på ett flertal lemman. De formella lemmana är dock många fler än de för det informella beviset, 15 jämfört med 2. Det beror delvis på att påståenden delats upp i flera delpåståenden för att underlätta arbetsfördelningen inom gruppen, men i första hand på att resonemangssteg som i det informella beviset ses som så triviala att de inte ens nämns också måste visas när beviset formaliseras. En ytterligare anledning är att det är lättare att göra omskrivningar och slutledningssteg på mindre delpåståenden separat. Ett 50-tal mer generella lemman ur SSReflect-biblioteket och ett par lemman som byggde på grundläggande egenskaper hos polynomdivision som gick utanför projektets avgränsningar och som tillhandahållits av projektets handledare, Anders Mörtberg, har använts i huvudbeviset och i beviset av lemmana.

Nedan beskrivs huvudbeviset rad för rad, och därefter beskrivs några av de mer intressanta lemmana utan bevis.

6.2.1 Huvudbeviset

Detta är den formella motsvarigheten till prop 5.4.1 för den implementerade algoritmen. Eftersom `toom_cook` bara utvärderar `toom_cook_rec` med ett visst argument så beror dess korrekthet helt på följande sats, som säger att `toom_cook_rec` är korrekt:

```
Lemma toom_cook_rec_correct : forall (n : nat) p q,
  unitmx V_I -> toom_cook_rec n p q = p * q.
```

När detta lemma är bevisat följer `toom_cook_correct` direkt:

```
Lemma toom_cook_correct : forall p q,
  toom_cook p q = p * q.
Proof. move=> p q. by apply: toom_cook_rec_correct. Qed.
```

Det formella beviset för att den rekursiva delen av algoritmen är korrekt har en struktur som liknar beviset av prop 5.4.1. Den största skillnaden är att induktionen i det formella fallet görs över antalet rekursiva anrop av algoritmen istället för att som i beviset av proposition 5.4.1 göra induktionen över graden av de polynom som multipliceras. Det beror på att den rekursiva delen av den implementerade algoritmen `toom_cook_rec` är definierad med en oberoende variabel n, så som beskrivits i 6.1.2. I stället för att induktionen som i det informella beviset bygger på att graden av de polynom som Toom-Cook anropas med minskar vid rekursiva anrop av algoritmen bygger den på att variabeln n minskas vid rekursiva anrop. I båda fallen innebär dock induktionsantagandet att vi antar att algoritmen fungerar korrekt för nästa lägre rekursiva anrop. Den andra stora skillnaden mellan det informella och det formella beviset är att många tekniska detaljer, som kan tas för givna i ett informellt bevis måste visas explicit. Dessutom måste hänsyn tas till hur matriser och polynom är representerade i SSReflect.

Här nedan ges den fullständiga programkoden för att beviset av att den rekursiva delen av algoritmen är korrekt, givet vissa lemman:

```
1  Lemma toom_cook_rec_correct : forall (n : nat) p q,
2   unitmx V_I -> toom_cook_rec n p q = p * q.
3  Proof.
4   elim=> [ // | n IHn p q V_inver] /=.
5   set b := exponent m p q.
6   set u := split m b p.
7   set v := split m b q.
8   case: ifP => [ // | _ ].
9   have ->:
10  \col_i toom_cook_rec n ((evaluate u) i 0) ((evaluate v) i 0) = \
       col_i ((evaluate u) i 0 * (evaluate v) i 0).
11  apply/colP => j.
12  by rewrite mxE [X in _ = X]mxE (IH2 _ _ V_inver).
13  rewrite 2!matrix_evaluation.
14  have ->:
15  \col_i (((\col_j u.[inter_points j 0]) i 0 * (\col_j v.[
       inter_points j 0]) i 0) = \col_i (u * v).[(inter_points i 0)].

16  apply/colP => k.
17  by rewrite 4!mxE -hornerM.
18  rewrite toom_cook_interpol //; last by apply: size_split_mul.
19  rewrite /recompose hornerM.
20  rewrite 2?recompose_split //.
21  rewrite /b exponentC.
22  by apply: exp_m_degree.
23  by apply: exp_m_degree.
24 Qed.
```

På rad 1 och 2 och formuleras lemmat och på rad 3 anges att beviset startar genom `Proof`. Rad 4 anger med `elim=>` att beviset kommer göras med induktion

över den variabel som står först i lemmat, `n`. Basfallet är trivialt precis som i proposition 5.4.1 eftersom `toom_cook_rec` är vanlig multiplikation när `n` = 0 och delmålet det utgör avslutas direkt med hjälp av `//`. Till höger om `|` ges instruktioner till induktionssteget. Variabeln `n'` där `n' + 1` = `n`, för det `n` som induktionen görs över instansieras. Induktionsantagandet `IHn`, polynomen `p` och `q` och antagandet `V_inver` om att interpolationsmatrisen är inverterbar instansieras också. Till sist skriver `/=` om målet genom att utveckla definitionen av `toom_cook_rec`.

Raderna 5 till 7 sätter kortare beteckningar på några av uttrycken i målet för att öka bevisets läslighet. Efter det har målet och kontexten blivit:

```
n' : nat
IHn : forall p q, unitmx V_I -> toom_cook_rec n' p q = p * q
p : {poly R}
q : {poly R}
V_inver : unitmx V_I
b := exponent m p q : nat
u := split m b p : {poly {poly R}}
v := split m b q : {poly {poly R}}
______________________________________(1/1)
(if (size p <= 2) || (size q <= 2)
  then p * q
  else recompose b
    (interpolate
      (\col_i toom_cook_rec n'
        ((evaluate u) i 0)
        ((evaluate v) i 0)))) = p * q
```

Nästa steg är sedan att skriva om målet så att vi kan använda induktionsantagandet `IHn`.

Först tar vi hand om `if ... then ... else` - uttrycket i vänsterledet. Det säger att vänsterledet är lika med `p * q` om graden av `p` eller `q` är mindre än 2 (eftersom multiplikation i algoritmen då utförs direkt enligt definitionen) och annars lika med det mer komplicerade uttrycket som står efter `else`. Dessa två uttryck skulle motsvarat basfallet och induktionssteget om induktionen hade gjorts över graden av p och q, som i proposition 5.4.1.

Nu görs istället på rad 8 en falluppdelning över `if ... then ... else`-uttrycket, med `ifP`, som inte ger oss något nytt induktionsantagande. I det första fallet när `(size p <= 2)|| (size q <= 2)` är höger- och vänsterled i målet trivialt lika och löses liksom det triviala målet på rad 4 direkt med `//`. När det är gjort är målet:

```
recompose b
  (interpolate
    (\col_i
      toom_cook_rec n'
        ((evaluate u) i 0)
        ((evaluate v) i 0))) = p * q
```

och vi vill använda induktionsantagandet

```
IHn : forall p q, unitmx V_I -> toom_cook_rec n' p q = p * q
```

tillsammans med antagandet `V_inver` om att `V_I` är inverterbar för att skriva om

```
toom_cook_rec n' ((evaluate u) i 0) ((evaluate v) i 0)
```

till `((evaluate u)i 0)* ((evaluate v)i 0)` inne i kolonnvektorn

```
\col_i toom_cook_rec n' ((evaluate u) i 0)((evaluate v) i 0)
```

Eftersom definitionen `\col_i` är låst för beräkning och uttrycket beror av indexet `i` i kolonnvektorn kan vi inte direkt skriva om uttrycket, se avsnitt 4.4. Så för att kunna göra det öppnar vi ett nytt delmål med taktiken `have` på rad 9 där vi bara arbetar med denna del av vänsterledet. Där använder vi på rad 11 lemmat `colP` som låter oss visa att två vektorer är lika genom att visa att elementen på motsvarande platser är lika i båda vektorerna. (Då vektorer är implementerade i SSReflect som funktioner från mängden av index i vektorn $\{0, \ldots, k\}$ till mängden av element i vektorn säger lemmat mer specifikt att två vektorer är lika om de extensionellt sett är samma funktion, det vill säga om de antar samma värden för samma argument). På rad 12 kan vi sedan slutligen visa att uttrycket i vänsterledet i `have`-satsen är lika med uttrycket i högerledet i `have`-satsen med hjälp av induktionshypotesen. Då har målet blivit:

```
recompose b
  (interpolate
    (\col_i
      ((evaluate u) i 0 * (evaluate v) i 0))) = p * q
```

Det återstår nu att visa tre saker: För det första att evalueringsfunktionen `evaluate` faktiskt ger tillbaka en vektor med ett polynom evaluerat i interpolationspunkterna, för det andra att interpolationsfunktionen `interpolate` givet dessa vektorer ger oss koefficienterna i produktpolynomet `u * v` och för det tredje att `recompose`-funktionen under vissa förutsättningar är vänsterinvers till `split`-funktionen (eftersom `u` och `v` ju fås från `p` och `q` genom `split`).

Det första och det andra som ska visas är enkla följder av definitionerna av matrismultiplikation och -invers och visas inte explicit i det informella beviset. I det formella beviset visas detta i två lemman, `matrix_evaluation` och `toom_cook_interpol`, som används på rad 13 respektive rad 18 att skriva om målet till

```
recompose b
  (interpolate (\col_i (u * v).[inter_points i 0])) = p * q
```

och sedan till

```
recompose b (u * v) = p * q
```

Lemmat `toom_cook_interpol` har som antagande att graden på det polynom vars koefficienter ska hittas genom interpolation, $u \cdot v$, är mindre än antalet interpolationspunkter, så när det åberopas uppkommer ytterligare ett delmål att visa:

```
______________________________________(2/3)
size (u * v) <= number_points
```

Det görs på rad 18 genom lemmat `size_split_mul`. Detta är ytterligare något som är självklart i det informella beviset, eftersom vi har definierat u och v

så att det är uppfyllt. Vi behöver också visa att interpolationsmatrisen `V_I` är inverterbar, men eftersom det är ett av antagandena i kontexten är det trivialt och visas på rad 18 med `//`.

Sedan utvecklar vi på rad 19 definitionen av `recompose` och använder lemmat `hornerM` som säger att $(p \cdot q)(x) = p(x) \cdot q(x)$, för att skriva om målet till:

```
u.['X^b] * v.['X^b] = p * q
```

Då kan vi på rad 20 använda lemmat `recompose_split`, som ungefär motsvarar lemma 5.4.3 i det informella beviset, för att visa att "sammansättningen" av de uppdelade polynomen genom att evaluera i x^b är korrekt och därmed skriva om `u.['X^b]` till `p` och `v.['X^b]` till `q`. Då har vi visat att vänsterledet i det ursprungliga målet är lika med högerledet `p * q`.

Det som återstår är sedan att bevisa motsvarigheten till Påstående 1 i lemma 5.4.3, som gör att villkoren för att `recompose_split` ska kunna användas på `p` och `q` är uppfyllda.

```
______________________________________(1/2)
size q <= m * b
______________________________________(2/2)
size p <= m * b
```

Det görs på rad 22 till 23 med lemmat `exp_m_degree` som säger i stort sett detta, efter att först på rad 21 ha skrivit om det ena av målet med ett lemma för att funktionen `exponent` är kommutativ, vilket avslutar det formella beviset för att Toom-Cook ger ett korrekt resultat.

Lemman till huvudbeviset

Förutom de 50-tal lemman ur SSReflect-biblioteket som har används i huvudbeviset har ett 15-tal specifika lemman till huvudsatsen formulerats och bevisats. Här presenteras några av dem. Det viktigaste lemmat är

```
Lemma recompose_split : forall (f: {poly R}) (b: nat),
  size f <= m * b ->
  (split m b f).['X^b] = f.
```

som säger att `recompose` (som evaluerar ett polynom i x^b) är vänsterinvers till `split` för polynom som har tillräckligt låg grad jämfört med argumenten `m` och `b`. Detta motsvarar ungefär lemma 5.4.3 i det informella beviset om man hade haft Påstående 1 som ett antagande istället för att visa det.

Beviset av `recompose_split` bygger på fem mindre lemman som motsvarar delpåståenden i lemma 5.4.3 och dessutom några lemman som visats av projektets handledare Anders Mörtberg eftersom de bygger på grundläggande egenskaper hos SSReflect-funktionen `rdivp` för polynomdivision som gick utanför projektets ramar. `recompose_split_lemma1` används för att visa `recompose_split_lemma2`, som sedan används för att visa `recompose_split_lemma3`, som används för att visa huvudlemmat tillsammans med motsvarigheten till Påstående 1, som visas separat i

```
Lemma exp_m_degree_lemma : forall p,
  m > 0 ->
  size p <= m * succn (size p %/ m).
```

Det första lemmat `recompose_split_lemma1` är en specik instans av (5.16), dvs av att

$$a(x) = a(x) \bmod x^b + \left(a(x)/x^b\right) x^b$$

.

```
Lemma recompose_split_lemma1 : forall (f: {poly R}) (k b: nat),
  (rmodp (rdivp f 'X^(k*b)) 'X^(b)) * 'X^(k*b) +
  (rdivp f 'X^(k.+1*b)) * 'X^(k.+1*b) =
  (rdivp f 'X^(k*b)) * 'X^(k*b).
```

Det andra lemmat motsvarar att (5.17) = (5.18), det vill säga induktionssteget i lemma 5.4.3.

```
Lemma recompose_split_lemma2 : forall (f: {poly R}) (k b: nat),
  \sum_(i < k.+1)
    (rmodp (rdivp f 'X^(i*b)) 'X^b) * 'X^(i*b) +
    (rdivp f 'X^(k.+1*b)) * 'X^(k.+1*b) =
  \sum_(i < k)
    (rmodp (rdivp f 'X^(i*b)) 'X^b) * 'X^(i*b) +
    (rdivp f 'X^(k*b)) * 'X^(k*b).
```

Det tredje lemmat svarar mot att (5.17) = $p(x)$.Givet induktionssteget som görs i `recompose_split_lemma2` och basfallet som ges av `recompose_split_lemma1` så fås

```
Lemma recompose_split_lemma3 : forall (f : {poly R}) (k b : nat),
  \sum_(i < k.+1)
    (rmodp (rdivp f 'X^(i*b)) 'X^b)*'X^(i*b) +
    (rdivp f 'X^(k.+1*b))*'X^(k.+1*b)
  = f.
```

Detta används slutligen för att visa `recompose_split`.

Kapitel 7

Diskussion

I det här kapitlet diskuteras våra tankar och intryck av bevisassistenten Coq och formalisering i allmänhet. Även de val och begränsningar som har gjorts under implementationen av Toom-Cook samt val av interpolationspunkter kommer att diskuteras.

7.1 Är det realistiskt att formalisera bevis och kod?

Implementationen av algoritmen och beviset i Coq var mycket tidskrävande, även för en relativt liten algoritm med ett kortare matematiskt bevis. Flera veckors arbete med flera personer behövdes för att färdigställa beviset. Jämförelsevis tog implementationen i Haskell samt testning med QuickCheck mindre än en dag. Frågan är om detta är en rättvis jämförelse eftersom vi hade mycket mer erfarenhet av Haskell än Coq. Även om mer erfarenhet skulle ha kortat ner utvecklingstiden markant tar formella bevis mycket längre tid att utveckla än att implementera algoritmen i till exempel Haskell.

Erfarenhet är inte den enda lösningen, Coq är ett forskningsprojekt och lider av diverse problem. Till exempel interaktionsproblem med programvaran, de grafiska gränssnitten som finns har en del buggar och saknar många funktioner som finns i andra moderna utvecklingsmiljöer för andra språk. Vidare är dokumentationen för standardbiblioteket dålig eller obefintlig. När man sitter och bevisar går väldigt mycket tid åt till att enbart sitta och söka i källkoden för att hitta rätt lemma. Om man kan lösa dessa problemen skulle tiden som behövs kortas ner avsevärt.

Att använda Coq för att bevisa kod är kanske i dagsläget bara relevant om det är väldigt viktigt att koden är korrekt. För matematiker kan det vara ett mer relevant verktyg eftersom Coq stoppar direkt när man gjort fel istället för att små enkla fel smyger sig igenom hela beviset.

7.2 Är beroendetypning användbart?

Beroendetypning är bra, men innan man kan motivera beroendetypning behöver man veta vad som är bra med ett starkt typsystem. Typer specificerar restriktio-

ner för vad kod kan göra, kompilatorn kan sedan se till att dessa restriktioner följs. Det kan alltså ses som ett verktyg som hjälper till att få koden korrekt. Eftersom typer är en slags dokumentation tvingas också programmeraren att tänka på *vad* det är koden ska göra istället för *hur* detta ska göras. Det ses som positivt då många buggar uppkommer då kod skrivs utan att ha koll på vad den ska göra. Denna motivering gäller generellt för typsystem och kan syfta på till exempel Haskell. Men det gäller även för beroendetypade system då beroendetypning är ett generellare system.

Beroendetypning gör att det ofta framgår exakt vad en funktion gör bara genom att läsa typdeklarationen. Detta är en stor fördel när man söker efter en specifik funktion som man inte vet namnet på men man vet vilka typer den ska ha. Att typerna ofta förklarar vad funktionerna gör innebär att programmerare inte behöver sätta sig in i avancerad kod i funktionen utan det räcker med att läsa vilka typer funktionen har. Ett exempel på detta är bevis i Coq där typerna specificerar vad som ska bevisas medan funktionskropparna ofta är väldigt svårförstådda och inte säger så mycket om vad som bevisas.

Ett exempel är matriser med dimensionerna i typen som gör att man inte kan tappa bort en dimension eller av misstag vända på dem. För matrismultiplikation vet vi till exempel att om man multiplicerar en $m \times n$-matris och en $n \times o$-matris får man en $m \times o$-matris. Med denna kunskap kan man definiera multiplikationsrutinen i stil med:

```
matMul :: Matrix m n -> Matrix n o -> Matrix m o
matMul = ...
```

Den här definitionen ger två fördelar:

- Det går inte att skicka in fel sorts matriser. Den andra matrisen måste ha lika många rader som den första matrisen har kolumner, annars kompilerar koden inte.
- När man implementerar själva multiplikationen är man tvungen att få rätt dimensioner på resultatmatrisen.

Om man skulle definiera en version av `matMul` där matrisernas storlek inte ingår i typen skulle man kunna anropa funktionen med matriser med felaktiga dimensioner. Gör man det kan programmet krascha vid körning med något mindre hjälpsamt felmeddelande i stil med "index out of bounds". Eller så går det ännu värre och funktionen kastar bort hälften av det tänkta resultatet. Denna typ av problem kan man slippa när man använder beroendetypning.

7.3 Implementeringen av Toom-Cook

Nedan diskuteras några val för definitionen och implementationen av algoritmen i avsnitt 5.2 och avsnitt 6. Några möjliga effektiviseringar vid implementation av exekverbara algoritmer tas också upp.

Den version av algoritmen som implementerats i det här projektet använder sig av matrismultiplikation och -invertering i evaluerings- och interpolationsstegen. Det motiverades delvis av praktiska skäl: det skulle gått utanför projektets ramar att i detalj undersöka alternativa metoder för dessa steg. Dessutom finns det

många algoritmer för att optimera matrisoperationer som skulle kunna användas för att effektivisera en exekverbar Toom-Cook-algoritm baserad på den i det här projektet. Det finns dock inte andra möjliga sätt att implementera interpolations- och evalueringsstegen på.

Evaluering I avsnitt 5.2.3 i algoritmen multipliceras evalueringsmatrisen V_e med vektorn av koefficienter till polynomet som ska evalueras. Detta är inte det enda möjliga sättet att evaluera polynomen på. För att optimera en beräkningsbar version av algoritmen kan det finnas möjlighet att välja ordning på additionerna och multiplikationerna i evalueringen så att det krävs färre operationer än vad som skulle ske med matrismultiplikation enligt definitionen[18].

Interpolation I avsnitt 5.2.5 antas det att interpolationsmatrisen V_I är inverterbar. Det gör att koefficienterna i polynomet kan fås genom att multiplicera inversen av V_I med vektorn av polynomets värden i interpolationspunkterna.

I en kropp (där alla element utom 0 har en multiplikativ invers) kommer V_I vara inverterbar om och endast om ekvationssystemet som ges av ekvation(5.12) är entydigt lösbart, vilket det kommer vara om interpolationspunkterna $\alpha_0, \ldots, \alpha_d$ är $d+1$ skilda punkter.

Detta gäller dock inte generellt i ett integritetsområde. Då kan ekvationssystemet 5.12 vara entydigt lösbart utan att V_I är inverterbar, eftersom en matris över ett integritetsområde är inverterbar om och endast om dess determinant är ett inverterbart element[21]. (V_I kommer dock vara inverterbar i bråkkroppen över integritetsområdet.)

Därför kan det i en del integritetsområden, där det finns för få inverterbara element eller där de element som ändå ger upphov till en inverterbar matris inte är lämpade för en effektiv matrisinvertering, vara lämpligt eller nödvändigt att använda andra ekvationslösningsmetoder än matrisinvertering.

I de integritetsområden där matrisinverteringen fungerar finns också möjligheter att effektivisera ekvationslösningen genom att använda till exempel LU-faktorisering. Bodrato[18][23] beskriver en algoritm för att givet interpolationspunkterna söka efter optimala följder av evalueringss- och interpolationsoperationer.

Generell eller specifik implementation En generell men exekverbar version av Toom-Cook skulle kanske kunna implementeras. Den skulle dock kräva en av två saker. Antingen att de önskade interpolationspunkterna gavs som argument vid varje applikation av algoritmen eller att interpolationspunkter för varje möjligt integritetsområde sparades eller genererades med hjälp av någon algoritm när Toom-Cook anropades. Eftersom Toom-Cook-m går att definiera för godtyckligt stora heltal m skulle en helt generell implementation kräva att det gick att generera godtyckligt många lämpliga interpolationspunkter eftersom antalet interpolationspunkter är $2m-1$. Det går naturligtvis inte i ett ändligt integritetsområde.

Vid praktiska tillämpningar av Toom-Cook är det dock mer rimligt att implementera en eller några instanser av den, till exempel Toom-Cook-3 och Toom-Cook-5, för någon eller några typer av integritetsområden. Bodrato[20][18][23] beskriver optimerade implementationer för karaktäristik 2, 3, 5 och 0 av bland annat Toom-Cook-3. GNU Multiple Precision Arithmetic Library använder

Toom-Cook-3, -4, -6.5 och -8.5 (där -6.5 och -8.5 är algoritmer speciellt anpassade för att multiplicera operander med stor storleksskillnad) för att multiplicera heltal inom olika storleksgränser[19].

När en sådan implementering är gjord kan algoritmen appliceras utan att man behöver ge interpolationspunkter som argument och man behöver inte räkna ut evaluerings- och interpolationsmatriserna vid varje applikation.

Implementationens påverkan på beviset Hur ett steg i algoritmen implementeras har en stor betydelse för hur komplicerat det sedan är att visa algoritmens korrekthet. Till exempel *recompose_split* är ett lemma i vårt formella bevis som säger att vår uppdelningsfunktion `split` är högerinvers till sammansättningsfunktionen `recompose` under vissa förutsättningar. Trots att det är ett ganska enkelt lemma så är det formella beviset relativt långt och komplicerat. Detta beror till stor del på att vi har implementerat uppdelningen, `split`, med polynomdivision och modulo för polynom vilket i sin tur ledde till svårigheter då vi behövde arbeta med de listor som utgör den underliggande strukturen för polynomtypen. Det bör finnas en mer naturlig definition av uppdelningsfunktionen som gör att beviset för *recompose_split* går att skriva på ett fåtal rader.

Vi diskuterade en alternativ implementation av `split` definierad utan polynomdivision och modulo för polynom som såg ut på följande vis:

```
Definition split (n b: nat) p : {poly {poly R}} :=
  \poly_(i < n) \poly_(j < b) p`_(i*b+j).
```

Vilket betyder $\sum_{i=0}^{n-1} \sum_{j=0}^{b-1} (p_{ib+j} x^j) y^i$ där inparametrarna är samma som i den ursprungliga `split`-definitionen. Vi hann dock inte testa om detta var en effektivare definition.

7.4 Formalisering och matematiska algoritmer

Mörtberg, Dénès och Siles presenterar i [5] en metodologi för att implementera effektiva matematiska algoritmer och visa deras korrekthet. De delar in processen i tre steg: I det första definieras algoritmen med hjälp av de låsta men uttrycksfulla datatyperna i SSReflect:s bibliotek och bevisas. I det andra steget översätts denna algoritm till en algoritm definierad med enklare typer i SSReflect som det går att utföra beräkningar med, och denna visar man ger samma resultat som algoritmen i det första steget för motsvarande argument. I det tredje steget kan algoritmen från steg 2 översättas till något annat språk, till exempel Haskell.

Implementationen av Toom-Cook i det här projektet kan sägas svara mot det första steget i den här processen. Om tid hade funnits i projektet skulle nästa steg varit att implementera en exekverbar algoritm som skulle motsvara steg 2 i processen ovan.

I SSReflect representeras ett polynom som ett par av en lista och ett bevis för att det sista elementet i listan inte är 0 och matriser som ändliga funktioner, se avsnitt 6.2. I den exekverbara versionen av Toom-Cook-algoritmen skulle polynom representeras med vanliga listor och matriser med listor av listor. Polynom- och matrisoperationer för dessa representationer finns implementerade i biblioteket CoqEAL[24]. Där finns också funktioner som översätter mellan olika representationer av polynom och matriser.

Genom att sedan visa att samma resultat fås om man först applicerar vår redan implementerade Toom-Cook-algoritm och sedan översätter resultatet till polynomlistor som om man först översätter polynomen till listor och sedan applicerar listversionen av Toom-Cook-algoritmen på dessa skulle man då kunna säkerställa att listversionen ger korrekta resultat för de argument som är intressanta, vilka är listor där det sista elementet är nollskiljt, som därmed är listversioner av polynom.

7.5 Val av interpolationspunkter

Interpolationspunkterna är en intressant del i algoritmen. Beroende på val av m i Toom-Cook-m som väljs så avgörs också hur många punkter som behövs, nämligen $2m-1$, för att entydigt bestämma koefficienterna i polynomet. Det förutsätter att integritetsområdet är tillräckligt stort, eftersom punkterna måste vara olika, och att punkterna också väljs så att interpolationsmatrisen blir inverterbar. Om integritetsområdet innehåller få element så är en lösning för att få tillgång till fler punkter att vi även tar punkter ur polynomringen. Punkterna bör dock vara av så låg grad som möjligt eftersom att polynomen u och v från algoritmen, se ekvationerna (5.5) och (5.7), helst ska avta i grad snabbt för att få så få rekursiva anrop som möjligt, se avsnitt 5.2.4, för att algoritmen ska vara effektiv. När polynomen u och v evalueras i interpolationspunkterna så tillhör de evaluerade polynomen $R[x]$, alltså $u(\alpha_0), \ldots, u(\alpha_{2m-2}), v(\alpha_0), \ldots, v(\alpha_{2m-2}) \in R[x]$. Graden av dessa polynom avgörs av graden av koefficienterna till u respektive v, alltså $u_0, \ldots, u_{m-1}, v_0, \ldots, v_{m-1} \in R[x]$, tillsammans med graden som interpolationspunkterna bidrar med när polynomen evalueras. Denna grad bör inte vara större än graden av ursprungspolynomen, alltså de polynom som skulle multipliceras från början, för det tar bort lite av poängen med algoritmen då idén med algoritmen är att dela upp polynomen så graden avtar.

En annan lösning är att definiera polynomen u och v så att de beror på två variabler istället för en. Denna definition är tagen ur [20] och lyder

$$u(y,z) = \sum_{i=0}^{m-1} u_i z^{m-1-i} y^i$$

där $u(x^b, 1) = u$, analogt för v. Detta leder till att man evaluerar i RxR eller i $R[x]$x$R[x]$ så att interpolationspunkterna är par av element i R eller i $R[x]$. Till exempel evaluera i punkten $(0,1)$ i $u(y,z)$ ger u_0 vilket motsvarar den konstanta termen i $u(y)$ och evaluera $(1,0)$ i $u(y,z)$ ger u_{n-1} som motsvarar den ledande koefficienten i u(y). Om en punkt (α, β) väljs så får inte en multipel till denna punkten väljas, alltså $(\lambda\alpha, \lambda\beta)$ där λ är ett element i ringen eller polynomringen, eftersom det ger upphov till en linjärt beroende interpolationsmatris och då kan den inte inverteras.

Litteratur

[1] J. Harrison, "Formal proof—theory and practice", *Notices of the AMS*, vol. 55, nr 11, s. 1395–1406, 2008.

[2] G. Gonthier, "Formal proof–the four-color theorem", *Notices of the AMS*, vol. 55, nr 11, s. 1382–1393, 2008.

[3] M. Aschbacher, "The status of the classification of the finite simple groups", *Notices of the AMS*, vol. 51, nr 7, s. 736–740, 2004.

[4] INRIA. (). CompCert webpage, url: `http://compcert.inria.fr/` (hämtad 2013-04-23).

[5] M. Dénès, A. Mörtberg och V. Siles, "A refinement-based approach to computational algebra in coq", i *Interactive Theorem Proving*, Springer, 2012, s. 83–98.

[6] U. Norell. (). Agda homepage, url: `http://wiki.portal.chalmers.se/agda/pmwiki.php` (hämtad 2013-05-20).

[7] Microsoft. (). Z3 homepage, url: `http://z3.codeplex.com/` (hämtad 2013-05-20).

[8] T. C. Hales, "Formal proof", *Notices of the AMS*, vol. 55, nr 11, s. 1370–1380, 2008.

[9] B. C. Pierce, C. Casinghino, M. Greenberg, C. Hriţcu och B. Y. Vilhelm Sjöberg, "Software foundations", *Course notes, online at http://www.cis.upenn.edu/~bcpierce/sf*, 2012.

[10] Y. Bertot, "Coq in a Hurry", *arXiv preprint cs/0603118*, 2006.

[11] G. Gonthier, R. S. Le m. fl., "An SSReflect tutorial", 2009.

[12] N. Bourbaki, *Elements of Mathematics : Theory of Sets*. Addison-Wesley, 1968.

[13] P. Martin-Lof och G. Sambin, *Intuitionistic type theory*. Bibliopolis Naples,, Italy, 1984, vol. 17.

[14] Y. Bertot och P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Incorporated, 2004.

[15] C. Bennet, *Första ordningens logik*. Studentlitteratur, 2004.

[16] H. Barendregt och H. Geuvers, "Proof-assistants using dependent type systems", *Handbook of automated reasoning*, vol. 2, s. 1149–1238, 2001.

[17] H. Geuvers, "Proof assistants: history, ideas and future", *Sadhana*, vol. 34, nr 1, s. 3–25, 2009.

[18] M. Bodrato, "Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0", i *Arithmetic of Finite Fields*, Springer, 2007, s. 116–133.

[19] T. Björnlund. (). The gnu multiple precision arithmetic library documentation, url: `http://gmplib.org/manual/` (hämtad 2013-05-18).

[20] M. Bodrato, "Notes on low degree toom-cook multiplication with small characteristic", *Preprint, Centro" V. Volterra", Universita di Roma" Tor Vergata*, 2007.

[21] S. Sombatboriboon, W. Mora och Y. Kemprasit, "Some results concerning invertible matrices over semirings", *Sci. Asia*, vol. 37, s. 130–135, 2011.

[22] R. L. Graham, D. E. Knuth och O. Patashnik, *Concrete mathematics: a foundation for computer science*. Addison-Wesley Reading, 1989, vol. 2.

[23] M. Bodrato och A. Zanoni, "Integer and polynomial multiplication: towards optimal toom-cook matrices", i *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, ACM, 2007, s. 17–24.

[24] M. Dénès. (2013-05-20). Documentation for coqeal, url: `http://www.maximedenes.fr/coqeal/`.

[25] J. R. Durbin, *Modern algebra: An introduction*. John Wiley & Sons, 2008.

Bilaga A

Matematikteori

A.1 Algebra

Definitioner och satser är tagna från[25].

Definition A.1.1. *En ring är en mängd R tillsammans med två operationer på R, nämligen addition $+$ och multiplikation $\cdot$ s.a.*

- $a + (b + c) = (a + b) + c$ *för alla* $a, b, c \in R$.
- *Det finns ett neutralt element* $0 \in R$ *s.a.* $a + 0 = 0 + a = a$ *för varje* $a \in R$.
- *För varje* $a \in R$ *finns ett element* $-a \in R$ *s.a.* $a + (-a) = (-a) + a = 0$.
- $a + b = b + a$ *för alla* $a, b \in R$.
- $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ *för alla* $a, b, c \in R$.
- $a \cdot (b + c) = a \cdot b + a \cdot c$ *för alla* $a, b, c \in R$.

Det neutrala elementet 0 *kallas för nolla.*

Exempel. Betrakta mängden av alla 2×2-matriser vars element är reella tal. Mängden tillsammans med operationerna matrisaddition och matrismultiplikation bildar en ring, där det neutrala elementet är $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Definition A.1.2. *En ring R sägs vara kommutativ om $a \cdot b = b \cdot a$ för alla $a, b \in R$.*

Definition A.1.3. *Ett element e i R sägs vara ett multiplikativ enhetselement till en ring om $e \cdot a = a \cdot e = a$ för varje $a \in R$.*

Definition A.1.4. *Ett element $a \neq 0$ i en kommutativ ring R sägs vara en nolldelare i R om det finns ett elementet $b \neq 0$ s.a. $a \cdot b = 0$.*

Definition A.1.5. *En kommutativ ring med multiplikativ enhetselement $e \neq 0$ och inga nolldelare sägs vara ett integritetsområde.*

Definition A.1.6. *Låt R vara en kommutativ ring. Ett polynom i variabeln x över R är ett uttryck på formen*

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

där koefficienterna $a_0, a_1, \ldots, a_n$ är element i R. Om $a_n \neq 0$, då är heltalet n graden av polynomet, och a_n är dess ledande koefficient. Två polynom i x är

lika om och endast om koefficienterna av samma grad är lika. Mängden av alla polynom i x över R betecknas $R[x]$.

Definition A.1.7. *Låt*

$$p(x) = a_0 + a_1x + \cdots + a_mx^m$$

och

$$q(x) = b_0 + b_1x + \cdots + b_nx^n$$

vara polynom över en kommutativ ring R. Då

$$p(x)q(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + \cdots + a_mb_nx^{x+n}$$

koefficienten av x^k är

$$a_0b_k + a_1b_{k-1} + a_2b_{k-2} + \cdots + a_kb_0$$

Sats A.1.8. *Om R är ett integritetsområde, då är $R[x]$ ett integritetsområde.*

Som följd av den här satsen är även $(R[x])[y]$ ett integritetsområde.

Bilaga B

Kod

B.1 Haskell-kod

B.1.1 ToomCook.hs

```
{-# LANGUAGE LambdaCase, BangPatterns,
             MagicHash, UnboxedTuples #-}
module ToomCook where

import Data.Ratio
import GHC.Base
import GHC.Integer

matVecMul :: Num a => [[a]] -> [a] -> [a]
matVecMul mat vec = map (sum . zipWith (*) vec) mat

log10 :: Integer -> Integer
log10 n = assert (n >= 0) $ go 0 n
  where
    go !acc  0  = acc
    go !acc !n' = go (acc + 1) (n' `quotInteger` 10)

data ToomCook = ToomCook
  { toomK :: Int
  , toomMat :: [[Integer]]
  , toomInvMat :: [[Rational]]
  }

{-# INLINE baseExponent #-}
baseExponent :: Int -> Integer -> Integer -> Integer
baseExponent k n m = assert (k > 0) $
  1 + max
    (log10 n `quotInteger` fromIntegral k)
    (log10 m `quotInteger` fromIntegral k)

{-# INLINE split #-}
```

```
-- |Split an integer into a big-endian list of integers using a given base
split :: Int -> Integer -> Integer -> [Integer]
split k b = assert (b > 0) $ go k []
  where
    go  0  acc _ = acc
    go !k' acc n = case n `quotRemInteger` b of
      (# n', x' #) -> go (k'-1) (x':acc) n'

-- |Merge a splitted integer using some base
-- This is the inverse of split.
merge :: Integer -> [Integer] -> Integer
merge b = recompose b . reverse

{-# INLINE evaluate #-}
-- |Evaluate the splits using the evaluation matrix
-- Note that since the splitlist is big-endian, either we have
-- to reverse each row of the evaluation matrix, or reverse the
-- splitlist.
evaluate :: [[Integer]] -> [Integer] -> [Integer]
evaluate = matVecMul

{-# INLINE interpolate #-}
interpolate :: [[Rational]] -> [Integer] -> [Integer]
interpolate mat vec = map numerator $ matVecMul mat $ map fromIntegral vec

{-# INLINE recompose #-}
recompose :: Integer -> [Integer] -> Integer
recompose b = go 1 0
  where
    go  _  !acc []     = acc
    go !b' !acc (x:xs) = go (b * b') (acc + b' * x) xs

{-# INLINE toomCook #-}
toomCook :: ToomCook -> Integer -> Integer -> Integer
toomCook !t !n !m | n < 0 && m < 0 = toomCook t (abs n) (abs m)
                  | n < 0          = negate $ toomCook t (abs n) m
                  | m < 0          = negate $ toomCook t n (abs m)

                  | n <= 100 || m <= 100 = n * m

                  | otherwise =
  let b   = 10^(baseExponent (toomK t) n m)
      n'  = split (toomK t) b n
      m'  = split (toomK t) b m
      n'' = evaluate (toomMat t) n'
      m'' = evaluate (toomMat t) m'
      r   = zipWith (toomCook t) n'' m''
      r'  = interpolate (toomInvMat t) r
   in recompose b r'
```

B.1.2 Settings.hs

```
module Settings where

import Data.Ratio
import ToomCook

-- NOTE: The rows in the evaluation matrices are inverted

toom1 :: ToomCook
toom1 = ToomCook 1 [[1]] [[1]]

karatsuba :: ToomCook
karatsuba = ToomCook 2
  [ reverse [ 1 , 0 ]
  , reverse [ 1 , 1 ]
  , reverse [ 0 , 1 ]
  ]
  [ [1  , 0 , 0  ]
  , [-1 , 1 , -1 ]
  , [0  , 0 , 1  ]
  ]

toom3 :: ToomCook
toom3 = ToomCook 3
  [ [ 0 , 0  , 1 ]
  , [ 1 , 1  , 1 ]
  , [ 1 , -1 , 1 ]
  , [ 4 , -2 , 1 ]
  , [ 1 , 0  , 0 ]
  ]
  [ [ 1    , 0   , 0   , 0    , 0  ]
  , [ 1%2  , 1%3 , -1  , 1%6  , -2 ]
  , [ -1   , 1%2 , 1%2 , 0    , -1 ]
  , [ -1%2 , 1%6 , 1%2 , -1%6 , 2  ]
  , [ 0    , 0   , 0   , 0    , 1  ]
  ]

toom4 :: ToomCook
toom4 = ToomCook 4
  [ reverse [ 1 , 0  , 0 , 0  ]
  , reverse [ 1 , 1  , 1 , 1  ]
  , reverse [ 1 , -1 , 1 , -1 ]
  , reverse [ 1 , -2 , 4 , -8 ]
  , reverse [ 1 , 2  , 4 , 8  ]
  , reverse [ 1 , 3  , 9 , 27 ]
  , reverse [ 0 , 0  , 0 , 1  ]
  ]
  [ [ 1     , 0     , 0     , 0      , 0     , 0     , 0   ]
  , [ -1%3  , 1     , -1%2  , 1%20   , -1%4  , 1%30  , -12 ]
```

```
  , [ -5%4  , 2%3   , 2%3   , -1%24  , -1%24 , 0     , 4   ]
  , [ 5%12  , -7%12 , -1%24 , -1%24  , 7%24  , -1%24 , 15  ]
  , [ 1%4   , -1%6  , -1%6  , 1%24   , 1%24  , 0     , -5  ]
  , [ -1%12 , 1%12  , 1%24  , -1%120 , -1%24 , 1%120 , -3  ]
  , [ 0     , 0     , 0     , 0      , 0     , 0     , 1   ]
  ]
```

B.1.3 Properties.hs

```
module Main where

import Settings
import Test.QuickCheck
import ToomCook

genK :: Gen Int
genK = choose (2, 10)

genNum :: Gen Integer
genNum = choose (100000000, 999999999)

propToomCookCorrect :: ToomCook -> Property
propToomCookCorrect t = forAll genNum $ \n -> forAll genNum $ \m ->
  toomCook t n m == n * m

propSplitCorrect :: Property
propSplitCorrect = forAll genK $ \k -> forAll genNum $ \n ->
  let b = 10^baseExponent k n n in n == merge b (split k b n)

main :: IO ()
main = do
  quickCheck (propToomCookCorrect karatsuba)
  quickCheck (propToomCookCorrect toom3)
  quickCheck (propToomCookCorrect toom4)
  quickCheck propSplitCorrect
```

B.2 Coq-kod

```
Require Import ssreflect ssrbool eqtype ssrnat ssrfun.
Require Import seq tuple choice.
Require Import finalg finfun fingroup finset fintype.
Require Import bigop matrix ssralg.
Require Import mxpoly poly polydiv.
Require Import div zmodp.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensives.
```

```
Import GRing.Theory Pdiv.Ring Pdiv.CommonRing Pdiv.RingMonic.
Open Scope ring_scope.

Section missingLemmas.
Lemma leq_pred_pred : forall (m n: nat), m <= n -> m.-1 <= n.-1.
Proof.
  move=> m n leqH. rewrite -2!subn1. by apply/(leq_sub2r 1): leqH.

Qed.
End missingLemmas.

Section toomCook.

Variable R : idomainType.
Implicit Types p q : {poly R}.

Variable number_splits : nat.
Definition m : nat := number_splits.
Definition number_points := (2 * m) .-1.
Variable inter_points : 'cV[{poly R}]_(number_points).

Hypothesis m_neq_0 : 0 < m.

Definition V_e : 'M[{poly R}]_(number_points, m) :=
  \matrix_(i < number_points, j < m) ((inter_points i 0))^+j.

Definition V_I : 'M[{poly R}]_(number_points) :=
 \matrix_(i < number_points, j < number_points) ((inter_points i
    0))^+j.

Hypothesis unitV_I : unitmx V_I.

Definition exponent (m: nat) p q : nat :=
  (maxn (divn (size p) m) (divn (size q) m)).+1.

Definition split (n b: nat) p : {poly {poly R}} :=
  \poly_(i < n) rmodp (rdivp p 'X^(i * b)) 'X^b.

Definition evaluate (u: {poly {poly R}}) : 'cV[{poly R}]_(
    number_points) :=
  V_e *m (poly_rV u)^T.

Definition interpolate (u: 'cV[{poly R}]_(number_points)) : {poly
     {poly R}} :=
  rVpoly (invmx V_I *m u)^T.

Definition recompose (b: nat) (w: {poly {poly R}}) : {poly R} :=
  w.['X^b].

Fixpoint toom_cook_rec (n: nat) p q : {poly R} :=
```

```
  match n with
  | 0%N => p * q
  | n'.+1 => if (size p <= 2) || (size q <= 2) then p * q else
       let b := exponent m p q in
       let u := split m b p in
       let v := split m b q in
       let u_a := evaluate u in
       let v_a:= evaluate v in
       let w_a := \col_i toom_cook_rec n' (u_a i 0) (v_a i 0) in
       let w := interpolate w_a
        in recompose b w
  end.

Lemma split_size_leq_m: forall (p: {poly R}) (b: nat),
  size (split m b p) <= m.
Proof.
  by move=> p b; rewrite size_poly.
Qed.

Lemma matrix_evaluation : forall p (b: nat),
  evaluate (split m b p) = \col_j (split m b p).[(inter_points j
      0)].
Proof.
  move=> p b.
  apply/matrixP => i j.
  rewrite !mxE /=.
  rewrite (@horner_coef_wide _ m).
  apply: eq_bigr => k _.
  by rewrite !mxE mulrC.
  by apply: split_size_leq_m.
Qed.

Lemma toom_cook_interpol_lemma0 : forall (f: {poly {poly R}}),
  size f <= number_points ->
  invmx V_I *m \col_i f.[inter_points i 0] = (poly_rV f)^T.
Proof.
  move=> f fsizeH.
  rewrite -[X in _ = X](mulKmx unitV_I).

  have->: \col_i f.[inter_points i 0] = V_I *m (poly_rV f)^T.
    apply/matrixP => i j.
    rewrite !mxE (@horner_coef_wide _ number_points).
    apply: eq_bigr => k _.
    by rewrite !mxE mulrC.
    by apply: fsizeH.
  done.
Qed.

Lemma toom_cook_interpol : forall (f: {poly {poly R}}),
  size f <= number_points ->
```

```
  (interpolate (\col_i (f.[(inter_points i 0)]))) = f.
Proof.
  move=> f leq.
  by rewrite -{2}(poly_rV_K leq) /interpolate
    (toom_cook_interpol_lemma0 leq) trmxK.
Qed.

Lemma rdivpXn_drop : forall p n, rdivp p 'X^n = Poly (drop n p).
Proof.
elim/poly_ind=> [n|p c ih [|n]]; first by rewrite rdiv0p polyseq0.

  rewrite expr0 rdivp1 drop0; apply/poly_inj.
  by rewrite (@PolyK _ 1 (p * 'X + c%:P)) //; case: (p * 'X + c%:
      P).
rewrite {1}[p](rdivp_eq (monicXn _ n)) mulrDl -mulrA -exprSr -
    addrA.
rewrite rdivp_addl_mul_small ?rmodp_addl_mul_small ?monicXn //.
  rewrite -cons_poly_def ih polyseq_cons.
  by have [-> /=| //] := nilP; rewrite polyseqC; case: (c == 0).
rewrite size_polyXn size_MXaddC ltnS; case: ifP => // _.
by rewrite (leq_trans (ltn_rmodpN0 _ _)) ?monic_neq0 ?monicXn ?
    size_polyXn.
Qed.

Lemma drop_addn : forall n m (s : seq R) , drop (m + n) s = drop
    m (drop n s).
Proof.
by elim=> [m s|m ih n [] //= a l]; rewrite ?addn0 ?drop0 // addnS
     ih.
Qed.

Lemma last_drop c n (s : seq R) : n < size s -> last c (drop n s)
     = last c s.
Proof.
elim/last_ind: s => //= s a _ hs.
by rewrite drop_rcons ?last_rcons; rewrite size_rcons ltnS in hs.
Qed.

Lemma recompose_split_lemma0 p m n :
  rdivp p ('X^m * 'X^n) = rdivp (rdivp p 'X^m) 'X^n.
Proof.
rewrite -exprD !rdivpXn_drop drop_addn.
by apply/polyP=> i; rewrite ?(coef_Poly,nth_drop) addnCA.
Qed.

Lemma rdivXn_size p n : size (rdivp p 'X^n) = (size p - n)%N.
Proof.
rewrite rdivpXn_drop -size_drop (@PolyK _ 1 (drop n p)) //.
have [hsp|hsp] := ltnP n (size p); last by rewrite drop_oversize
    // oner_neq0.
```

```
by rewrite last_drop // {hsp}; case: p.
Qed.

Lemma recompose_split_lemma1 : forall (f: {poly R}) (k b: nat),
  (rmodp (rdivp f 'X^(k*b)) 'X^(b)) * 'X^(k*b) +
  (rdivp f 'X^(k.+1*b)) * 'X^(k.+1*b) =
  (rdivp f 'X^(k*b)) * 'X^(k*b).
Proof.
  move => f k b.
  by rewrite {1}mulSnr mulSn 2!exprD mulrA -mulrDl addrC
    recompose_split_lemma0 -(rdivp_eq (monicXn _ _) _).
Qed.

Lemma recompose_split_lemma2 : forall (f: {poly R}) (k b: nat),
  \sum_(i < k.+1)
    (rmodp (rdivp f 'X^(i*b)) 'X^b) * 'X^(i*b) + (rdivp f 'X^(k
        .+1*b)) * 'X^(k.+1*b)
  =
  \sum_(i < k)
    (rmodp (rdivp f 'X^(i*b)) 'X^b) * 'X^(i*b) + (rdivp f 'X^(k*b)
        ) * 'X^(k*b).
Proof.
  move=> f k b.
  symmetry.
  by rewrite big_ord_recr //= -recompose_split_lemma1 addrA.
Qed.

Lemma recompose_split_lemma3 : forall (f : {poly R}) (k b : nat),
  \sum_(i < k.+1)
    (rmodp (rdivp f 'X^(i*b)) 'X^b)*'X^(i*b) +
    (rdivp f 'X^(k.+1*b))*'X^(k.+1*b)
  = f.
Proof.
  move=> f k b.
  elim: k => [ | n IH ].
    rewrite big_ord_recr /=.
    rewrite big_ord0 add0r mul0n rdivp1 mulr1 mul1n addrC.
    rewrite -(rdivp_eq (monicXn _ _) _) //.
    by rewrite recompose_split_lemma2 IH.
Qed.

Lemma recompose_split : forall (f: {poly R}) (b: nat),
  size f <= m * b ->
  (split m b f).['X^b] = f.
Proof.
  move=> f b.
  case: m => [ | [ H | m' H ] ].
    + rewrite mul0n size_poly_leq0.
      move/eqP ->.
      by rewrite horner_poly big_ord0.
```

```
    + rewrite mul1n in H.
      rewrite horner_poly big_ord_recr /=.
      rewrite big_ord0 mul0n !expr0 mulr1 rdivp1 add0r.
      rewrite rmodp_small //.
      by rewrite size_polyXn.

    + rewrite horner_poly big_ord_recr /=.
      rewrite rmodp_small.
      rewrite -exprM.
      rewrite mulnC.
      rewrite mulnC.

      have ->:
          \sum_(i < m'.+1) (rmodp (rdivp f 'X^(i * b)) 'X^b * 'X^b
               ^+ i) =
          \sum_(i < m'.+1) (rmodp (rdivp f 'X^(i * b)) 'X^b * 'X^(
              i*b)).
        apply: eq_bigr => j t.
        by rewrite -exprM mulnC.

    by rewrite recompose_split_lemma3.

    by rewrite size_polyXn ltnS rdivXn_size leq_subLR addnC -mulSn
        .
Qed.

Lemma exp_m_degree_lemma : forall p,
  m > 0 -> size p <= m * succn (size p %/ m).
Proof.
  by move=> p H; rewrite mulnC; apply: (ltnW (ltn_ceil (size _) _)
      ).
Qed.

Lemma exp_m_degree : forall p q,
  size p <= m * exponent m p q.
Proof.
  move=> p q.
  rewrite /exponent.
  suff: size p <= m * (size p %/ m).+1.
  move=> sp.

  have: succn (size p %/ m) <= succn (maxn (size p %/ m) (size q
      %/ m)).
    by apply/leq_maxl.
  move=> H.

  have: m * succn (size p %/ m) <= m * succn (maxn (size p %/ m)
      (size q %/ m)).
    by apply/leq_mul.
```

```
  move=> G.
  by apply: (leq_trans sp G).
  by apply: exp_m_degree_lemma.
Qed.

Lemma exponentC : forall p q, exponent m p q = exponent m q p.
Proof.
  by move=> p q; rewrite /exponent maxnC.
Qed.

Lemma size_split_mul : forall p q,
  size (split m (exponent m p q) p * split m (exponent m p q) q)
      <= number_points.
Proof.
  move=> p q.
  set b := (exponent m p q).
  set u := split m b p.
  set v := split m b q.
  move: (split_size_leq_m p b) (split_size_leq_m q b).
  move: (size_mul_leq u v) => sizeH sizeu sizev.
  rewrite /number_points mul2n -addnn.
  rewrite (leq_trans sizeH) //.
  by apply: leq_pred_pred (leq_add sizeu sizev).
Qed.

Lemma toom_cook_rec_correct : forall (n : nat) p q,
  toom_cook_rec n p q = p * q.
Proof.
  elim=> [ // | n' IH p q ] /=.
    set b := exponent m p q.
    set u := split m b p.
    set v := split m b q.
    + case: ifP => [ // | _ ].
      * have ->:
            \col_i toom_cook_rec n' ((evaluate u) i 0) ((evaluate
                v) i 0) =
            \col_i ((evaluate u) i 0 * (evaluate v) i 0).
          apply/colP => j.
          by rewrite mxE [X in _ = X]mxE IH.

      rewrite /recompose.
      rewrite !matrix_evaluation.

      * have ->:
            \col_i ((\col_j u.[inter_points j 0]) i 0 *
              (\col_j v.[inter_points j 0]) i 0) =
            \col_i (u * v).[(inter_points i 0)].
          apply/colP => k.
          by rewrite 4!mxE -hornerM.
```

```
      rewrite toom_cook_interpol ?hornerM ?recompose_split //.

    rewrite /b exponentC.
    by apply/exp_m_degree.
    by apply/exp_m_degree.
    by apply: size_split_mul.
Qed.

Definition toom_cook p q : {poly R} :=
  toom_cook_rec (maxn (size p) (size q)) p q.

Lemma toom_cook_correct : forall p q,
  toom_cook p q = p * q.
Proof.
  move=> p q. by apply: toom_cook_rec_correct.
Qed.

End toomCook.
```