Jere Pehkonen

Ariteetti- ja tietotyyppigeneerinen ohjelmointi Coq-todistusassistentilla

Tietotekniikan Pro gradu -tutkielma

14. maaliskuuta 2021

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Jere Pehkonen

Yhteystiedot: jeanpehk@gmail.com

Ohjaajat: Sampsa Kiiskinen ja Tuomo Rossi

Työn nimi: Ariteetti- ja tietotyyppigeneerinen ohjelmointi Coq-todistusassistentilla

Title in English: Arity-Generic Datatype-Generic Programming in the Coq Proof Assistant

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 68+0

Tiivistelmä: Tutkielman tavoitteena on luoda toimiva kirjasto ariteetti- ja tietotyyppigeneeristä ohjelmointia varten. Ariteetti- ja tietotyyppigeneerinen ohjelmointi edesauttaa toisteisen lähdekoodin määrän vähentämistä sekä määritelmien uudelleenkäyttöä, mikä helpottaa lähdekoodin ylläpitoa ja vähentää yksinkertaisten virheiden määrää. Kirjasto muodostetaan Coq-todistusassistentilla, hyödyntäen sen ominaisuuksia funktionaalisena ja riippuvasti tyyppitettynä ohjelmointikielenä. Lisäksi kirjaston toteutuksessa pyritään käyttämään universumipolymorfismia toisteisten määritelmien välttämiseksi. Coqille ei ole saatavilla ariteetti- ja tietotyyppigeneeristä kirjastoa, joten kirjaston luonnin seurauksena saadaan ariteetti- ja tietotyyppigeneerisyyden tuomat edut Coqille.

Avainsanat: Coq, geneerinen ohjelmointi, polymorfismi, riippuvat tyypit

Abstract: The goal of this thesis is to create a functional library for arity-generic datatype-generic programming. Arity-generic datatype-generic programming reduces the amount of code needed and helps reuse definitions, which leads to easier maintenance and decreases the amount of simple errors. The library is done using the Coq proof assistant by using its features as a functional and dependently-typed programming language. The library will also use universe polymorphism to avoid the need for duplicated definitions in the library. There is no available arity-generic datatype-generic library for Coq so the library will bring the benefits of arity-generic datatype-generic programming to Coq.

Keywords: Coq, generic programming, polymorphism, dependent types

Termiluettelo

Agda	on riippuvasti tyypitetty funktionaalinen ohjelmointikieli, jonka alkuperäinen kehittäjä on Ulf Norell.
Ariteetti	tarkoittaa funktion parametrien määrää.
CoC	eli *Calculus of Constructions* on tyyppiteoria, joka toimi Coqin alkuperäisenä perustana ja jonka kehittivät Thierry Coquand ja Gerard Huet.
CIC	eli *Calculus of Inductive Constructions* on Coqin perustana oleva tyyppiteoria, jonka alkuperäiset kehittäjät ovat Thierry Coquand ja Christine Paulin-Mohring.
Coq	on interaktiivinen todistusassistentti, jota voidaan käyttää riippuvasti tyypitettynä funktionaalisena ohjelmointikielenä. Coq on nimetty sen pääasiallisen kehittäjän Thierry Coquandin mukaan.
CoqIDE	on graafinen työkalu, jota voidaan käyttää kehitysympäristönä Coqille.
Muodollinen todistus	on todistus, joka johdetaan ennaltamääritettyjen sääntöjen pohjalta ja joka on mekaanisesti varmistettavissa.
Hahmonsovitus	(engl. *pattern matching*) on ohjelmoinnin muoto, jossa tehdään valintoja käsiteltävän datan rakenteen perusteella.
Idris	on riippuvasti tyypitetty funktionaalinen ohjelmointikieli.
Kielilaajennos	on laajennos, joka lisää ohjelmointikielen tarjoamia ominaisuuksia.
Kind	on tyypin tyyppi.
Luonnollinen päättely	(engl. *natural deduction*) on menetelmä, jossa loogista päättelyä suoritetaan syntaktisten päättelysääntöjen mukaan.
Johdonmukaisuus	(engl. *consistency*) on formaalin järjestelmän ominaisuus. Johdonmukaisesta järjestelmästä ei voi johtaa ristiriitaa järjestelmän sääntöjä noudattamalla.
Lambdakalkyyli	on formaali järjestelmä ja universaali laskennan malli.

PowerPC Assembly	on symbolinen konekieli.
Proof General	on Emacs-pohjainen käyttöliittymä Coqille sekä muutamalle muulle todistusassistentille.
Todistusassistentti	on interaktiivinen ympäristö, jossa voi rakentaa muodollisia todistuksia.
Tyyppiteoria	tutkii tyyppijärjestelmiä. Jotkin tyyppiteoriat voivat toimia myös matematiikan perustana.
Riippuva tyypitys	(engl. *dependent typing*) on tyypityksen muoto, jossa tyypit voivat riippua arvoista.

Kuviot

Taulukot

Sisältö

1 Johdanto

Tässä tutkielmassa tarkastellaan ariteetti- ja tietotyyppigeneeristä ohjelmointia Coqin avulla. Coq on todistusassistentti, jota voidaan käyttää funktionaalisena ja riippuvasti tyypitettynä ohjelmointikielenä. Tarkastelu toteutetaan tekemällä mallitoteutus ariteetti- ja tietotyyppigeneerisestä kirjastosta sekä toteuttamalla malliesimerkit tietotyyppigeneerisen sekä ariteetti- ja tietotyyppigeneerisen funktion luonnista kirjaston avulla.

Geneerinen ohjelmointi yleistää operaatioden toimintaa. Lisäksi geneerisyyden muotoja on useita ja niihin voidaan päästä monin eri keinoin. Yleensä geneerisyys kuitenkin ilmenee operaatioiden parametrisaation kautta, esimerkiksi parametrisen polymorfismin muodossa. Ariteettigeneerisyydessä parametrisaatio tehdään parametrien määrän suhteen, tietotyyppigeneerisyydessä taas parametrien rakenteen suhteen.

Ariteetti- ja tietotyyppigeneerinen ohjelmointi edesauttaa ohjelmointikoodin uudelleenkäyttöä ja vähentää tarvittavaa koodimäärää. Toteutetun kirjaston avulla määriteltyjä geneerisiä funktioita voidaan käyttää argumenteilla, jotka eroavat toisistaan sekä niiden tietotyypin että ariteetin, eli funktion parametrien määrän, suhteen. Geneerisyyden hyötynä on, että näille funktioille ei tarvitse määritellä erillisiä versioita tapauksille, joissa niiden saamat argumentit ovat erilaiset.

Kirjasto on tutkielman lähdekatsauksen perusteella kolmas ariteetti- ja tietotyyppigeneerinen kirjasto, toinen riippuvasti tyypitetyllä kielellä toteutettu ja ainoa Coqilla toteutettu. Kirjaston toteutuksen lisäksi tutkielman uutena panoksena on universumipolymorfismin käyttö kirjaston määritelmien yksinkertaistamiseksi ja samalla kielen johdonmukaisuuden säilyttämiseksi.

Weirich ja Casinghino (2010) ovat toteuttaneet ariteetti- ja tietotyyppigeneerisen kirjaston Agda-ohjelmointikielellä. Tutkielman kirjaston toteutus pohjautuu rakenteeltaan tähän kirjastoon. Weirich ja Casinghino nostivat toteutuksestaan esille myös muutamia puutteita, joihin tutkielmassa erityisesti vastataan: jotta kirjaston määritelmät voitiin pitää yksinkertaisina, käytettiin kirjastossa kielilaajennosta `type-in-type`, joka rikkoi kielen johdonmukaisuuden. Weirich ja Casinghino toteuttivat kirjastosta myös johdonmukaisen version, jossa

kuitenkin koodimäärä oli selkeästi suurempi. Tutkielman toteutuksessa yhdistetään määritelmien yksinkertaisuus ja kielen johdonmukaisuus käyttämällä Coqin tarjoamaa universumipolymorfismia. Universumipolymorfismin ansiosta tyyppimääritelmiä ei tarvitse toistaa eri tyyppitasoilla, mikä vähentää toisteisen koodin määrää. Samalla kieli pysyy johdonmukaisena, minkä laajennoksen `type-in-type` käyttö rikkoo. Kielen johdonmukaisuuden säilyttämisen ansiosta voitaisiin kirjaston ominaisuuksia todistaa käyttämällä Coqin todistustilaa. Tutkielman puitteissa ei kuitenkaan suoriteta muodollisia todistuksia kirjaston toimivuudesta, vaan keskitytään Coqin ominaisuuksiin funktionaalisena ohjelmointikielenä.

Kirjaston toteutuksen lisäksi tutkielmassa arvioidaan kirjastoa. Arvioinnin päätavoitteena on varmistaa, että kirjastolle asetetut tavoitteet täyttyvät, mikä tarkoittaa toimivan ariteetti- ja tietotyyppigeneerisen kirjaston luontia universumipolymorfismia hyödyntäen. Lisäksi kirjastoa verrataan joidenkin ominaisuuksien osalta aiemmin toteutettuihin geneerisiin kirjastoihin sekä tarkastellaan universumipolymorfismin tuomia hyötyjä. Arvioinnissa myös esitellään toteutuksessa tiedossa olevat puutteet sekä pohditaan toteutuksessa esiintyneitä vaikeuksia ja ongelmakohtia.

Seuraavassa luvussa tarkastellaan tarkemmin Coqia ohjelmointikielenä. Kolmannessa luvussa käsitellään geneeristä ohjelmointia. Neljännessä luvussa käydään läpi tutkielman tutkimusmenetelmää. Viidennessä luvussa esitellään kirjaston toteutus ja kuudennessa luvussa arvioidaan sen onnistumista. Lisäksi viimeisessä luvussa luodaan tiivistetty yhteenveto tutkielman tuloksista.

2 Coq

Coqiin viitataan yleisesti termeillä todistusassistentti (engl. *proof assistant*), interaktiivinen lausetodistaja (engl. *interactive theorem prover*) ja riippuvasti tyypitetty funktionaalinen ohjelmointikieli (engl. *dependently-typed functional programming language*). Tämän tutkielman yhteydessä Coqia katsotaan pääasiallisesti riippuvasti tyypitetyn funktionaalisen ohjelmointikielen näkökulmasta. Lisäksi tarkastellaan hieman Coqin ominaisuuksia todistusassistenttina, koska osa kirjaston määritelmistä vaatii todistustilan käyttöä.

2.1 Tausta

Coqin ensimmäinen toteutus aloitettiin vuonna 1984 Gerald Huetin ja Thierry Coquandin toimesta ja julkaistiin vuonna 1988. Ensimmäinen toteutus ei kuitenkaan vielä kattanut induktiivisia tyyppejä, jotka lisäsi Christine Paulin-Mohring ja joka julkaistiin vuonna 1990. Suurin osa Coqin toteutuksesta on tapahtunut ranskalaisessa tutkimuslaboratorio INRIAssa. (Coq Development Team 2020)

Coq on työkalu, joka soveltuu erityisen hyvin absoluuttista luottamusta vaativiin tehtäviin. Sen avulla voidaan esittää ohjelmien vaatimukset ja kehittää ohjelmia, jotka toteuttavat nämä vaatimukset (Bertot ja Casteran 2004). Coqilla toteutetut ohjelmat ovat siten, itse Coqin toteutukseen luottaen, todistetusti vaatimusmäärittelyn mukaan toimivia eikä niitä rikkovien ohjelmien luominen kielen avulla ole mahdollista. Ohjelmien muodollinen todistus eroaa selkeästi esimerkiksi yksikkötestauksesta, jossa vaatimusten mukaista toimintaa ei voida luvata. Yksikkötestaus pystyy ainoastaan lisäämään luottamusta ohjelman toimintaan lisäämällä testien määrää. Toisaalta ohjelmien muodollinen todistus on vaativampaa, joten sen hyödyt saadaan parhaiten esille ympäristöissä, joissa ohjelmien oikeellisuus on erityisen tärkeää (Bertot ja Casteran 2004).

Vaatimusmäärittelyt täyttävien ohjelmien lisäksi voidaan Coqilla kehittää matemaattisia todistuksia. Matemaattisia todistuksia rakennetaan interaktiivisesti Coqin todistustilan avulla, joka hyödyntää todistuksien varmentamisessa Coqin tyyppijärjestelmää. Coqia voidaankin sen todistusominaisuuksien vuoksi kutsua todistusassistentiksi tai interaktiiviseksi lauseto-

distajaksi. (Bertot ja Casteran 2004)

Coqin avulla on jo toteutettu erilaisia laajoja formalisointeja. Matematiikassa esimerkkinä formalisoinnista on muun muassa neliväriteoreeman todistus, jossa matemaattiset konseptit käännettiin Coqin avulla todistettaviksi tietorakenteiksi ja ohjelmiksi (Gonthier 2005). Pelkästään yhden teoreemaan todistamisen lisäksi on olemassa myös laajempia matemaattisia formalisointeja, kuten `UniMath`-kirjasto, joka pyrkii formalisoimaan mahdollisimman suuren osan matematiikasta Coqin avulla (Voevodsky, Ahrens, Grayson ym.). Varsinaisessa ohjelmistokehityksessä taas on Coqilla formalisoitu muun muassa optimoiva C-kääntäjä, joka pystyy kääntämään suuren osan C-kielestä PowerPC assembly-kieleksi (Leroy 2009).

2.2 Syntaksi ja perusteet

Tässä luvussa käydään tiivistetysti läpi Coqin syntaksin perusteet sekä yleisimmät Coqissa esille tulevat konseptit. Esittely tehdään, jotta tulevat koodiesimerkit sekä kirjaston lähdekoodi olisivat helpommin lähestyttäviä. Ajoittain esimerkeissä käytetään tukena Haskellia, koska oletuksena on, että se on suurimmalle osalle lukijoista tutuin syntaksi funktionaalisille ohjelmointikielille. Coqia käyttävissä koodiesimerkeissä seurataan Coqin standardikirjaston (Coq Development Team 2021) esimerkkejä.

Yksi Coqin olennaisimmista toiminnoista ovat induktiiviset tyypit, joiden avulla Coqissa luodaan uusia tyyppejä. Induktiivisia tyyppejä käytetään tutkielman kirjastossa yleisimpänä uusien tyyppien muodostustapana. Intuitiivisesti ajateltuna induktiiviset tyypit ovat tyyppejä, jotka voidaan muodostaa äärellisellä määrällä niiden konstruktoreja (Harper 2016). Coqissa induktiiviset tyypit laajentavat esimerkiksi Haskellin tarjoamia algebrallisia tietotyyppejä, sekä kielilaajennoksena saatavia yleistettyjä algebrallisia tietotyyppejä (engl. *generalized algebraic data types* eli *GADTs*), mahdollistamalla määritelmien tyyppiriippuvuuden (Chlipala 2013).

Induktiivisia tyyppejä muodostetaan Coqissa avainsanan `Inductive` avulla. Esimerkiksi luonnolliset luvut voidaan Coqissa määritellä seuraavasti:

```
Inductive nat : Set :=
```

```
| O : nat
| S : nat -> nat.
```

Induktiivisia tyyppejä määriteltäessa tulee käyttäjän ilmaista määritelmän tyyppi, joka on tässä tapauksessa `Set`. Samoin tyypin konstruktoreille, jotka erotellaan pystyviivalla, tulee määritellä tyypit. Vastaava määritelmä voitaisiin luoda esimerkiksi Haskellilla seuraavasti:

```
data Nat
  = O | S Nat
```

Tyyppien määrittelyn lisäksi on ohjelmointikielessä olennaista pystyä määrittelemään termejä. Termejä voidaan Coqissa määritellä esimerkiksi avainsanalla `Definition`, mikä onkin tutkielman kirjaston toteutuksessa yleisin tapa määritellä ei-rekursiiviset funktiot. Esimerkiksi funktio, joka antaa luonnollisen luvun edeltäjän voidaan määritellä seuraavasti:

```
Definition pred n :=
  match n with
    | O => n
    | S u => u
  end.
```

Määritelmässä avainsanaa `Definition` seuraava `pred` on määritelmän nimi. Sitä seuraavat kirjainyhdistelmät ovat sen parametrit, joita on tässä tapauksessa vain yksi eli `n`. Määritelmässä käytetään Coqin tyyppijärjestelmää hyödyksi siten, että määritelmän ja sen parametrin tyyppejä ei nimetä eksplisiittisesti. Sama määritelmä voidaan kuitenkin esittää kielen syntaksiin tutustumattomalle kenties selkeämmin näyttämällä tyypitykset eksplisiittisesti:

```
Definition pred (n : nat) : nat :=
  match n with
    | O => n
    | S u => u
  end.
```

Hyötynä eksplisiittisyydessä on se, että funktion parametrin sekä paluuarvon tyypitys voidaan selkeästi huomata suoraan määritelmästä. Lisäksi Coqin tyyppijärjestelmän tyypinpäät-

tely ei ole päätösongelma.

Määritelmässä `pred` on olennaista avainsanan `match` käyttö, joka on vain syntaksia induktioperiaatteelle. Sitä käytetään hahmonsovitukseen (engl. *pattern matching*), mikä tarkoittaa induktiivisen tyypin rakenteellista analyysia sekä toiminnan valitsemista tähän rakenteeseen perustuen (Coq Development Team 2020).

Avainsanan `Definition` käyttö on kuitenkin vain yksi tapa määritellä funktioita, sillä se voidaan tehdä myös avainsanalla `Fixpoint`. Sitä käytetään, kun tarvitaan rekursiivisia funktiomääritelmiä (Coq Development Team 2020). Esimerkiksi yhteenlasku luonnollisten lukujen avulla voidaan määritellä seuraavasti:

```
Fixpoint add n m :=
  match n with
    | O => m
    | S p => S (add p m)
  end.
```

Rekursiivisten funktioiden täytyy Coqissa noudattaa syntaktisia rajoituksia, jotka pitävät huolta siitä, että funktion määritelmä ei johda ikuiseen silmukkaan (Coq Development Team 2020). Coqin dokumentaation esimerkkiä seuraamalla, voidaan edellinen yhteenlaskua toimittava funktio määritellä siten, että Coqin tyyppitarkastaja ei määritelmää hyväksy:

```
Fixpoint add n m :=
  match n with
    | O => m
    | p => S (add p m)
  end.
```

Ongelmana määritelmässä on arvot, joilla `add`-funktiota rekursiivisesti kutsutaan. Ongelma johtuu siitä, että Coqin tulee pystyä määritelmästä päättelemään, että argumentti, jonka suhteen rekursiota suoritetaan, on vähenevä (Coq Development Team 2020). Määritelmän tapauksessa taas näin ei ole, sillä rekursiivinen kutsu sisältää saman alkuperäisen arvon `p`.

Laskentaa Coqissa pystyy suorittamaan esimerkiksi komennon `Compute` avulla, joka suo-

rittaa sille annetun lausekkeen sekä tulostaa käyttäjälle sen antaman tuloksen. Komento `Compute` käyttää laskennan suorittamiseen arvolla kutsumista (engl. *call-by-value*), eikä se ole ainoa tapa suorittaa laskentaa (Coq Development Team 2020). Tässä tutkielmassa komentoa `Compute` koodiesimerkeissä käytettäessä kuvataan sen tuloksena antama arvo seuraavalla rivillä kommentointuna. Esimerkiksi funktion `add` toimintaa pystytään tutkielman periaatteiden mukaisesti esittämään seuraavasti:

```
Compute add 1 2.
  (* = 3 : nat *)
```

Coq on vanha kieli ja siten siihen on kertynyt useita tapoja esittää sama asia. Näin ollen on hyvä huomioida, että tässä luvussa esitetyt syntaktiset esimerkit ovat vain yksi tapa määritellä asioita Coqissa ja että syntaksin ja kielen syvempi ymmärrys vaatii tarkempaa analyysiä. Syntaksin ja konseptien tarkat määritelmät ovat saatavilla Coqin dokumentaatiosta.

2.3 Riippuva tyypitys

Coqin pohjalla oleva tyyppiteoria Calculus of Inductive Constructions on riippuvasti tyypitetty, mikä tarkoittaa sitä, että siinä määritelty tyyppi voi riippua myös arvoista (Verbruggen, Vries ja Hughes 2008). Riippuva tyypitys eroaa yleisimmistä ohjelmointikielten tyypityksistä siinä, että yleensä tyypit voivat riippua vain muista tyypeistä. Esimerkiksi Haskellissa muista tyypeistä riippuva algebrallinen tietotyyppi `Either` määritellään seuraavasti (Marlow 2010):

```
data Either a b
  = Left a | Right b
```

Määritelmästä voidaan huomata, että tyyppi `Either` riippuu tyyppimuuttujista `a` ja `b`. Haskellin syntaksin mukaan näiden molempien on kuitenkin oltava tyyppejä (eli niiden tyypin on oltava Haskellin määritelmien mukaisesti `kind`) eikä arvoja, mikä tekisi Haskellista riippuvasti tyypitetyn.

Riippuva tyypitys ei ole kuitenkaan Coqille uniikkia, sillä muita riippuvasti tyypitettyjä kieliä ovat esimerkiksi Agda ja Idris (Norell 2008; Brady 2013). Myös Haskellissa riippuvan

tyypityksen osittainen mallintaminen on mahdollista `singleton`-kirjaston avulla (Eisenberg ja Weirich 2012). Lisäksi Eisenberg (2017) esittää väitöskirjassaan kielilaajennoksen Haskelliin, jonka avulla riippuva tyypitys voidaan kieleen lisätä.

Tyypillinen esimerkki riippuvasta tyypityksestä on vektori, eli lista, joka on indeksoitu pituutensa perusteella:

```
Inductive vec (A : Type) : nat -> Type :=
| vnil : vec A O
| vcons : forall n, A -> vec A n -> vec A (S n).
```

Määritelmässä luodaan induktiivinen tietotyyppi, joka indeksoidaan tyypin `nat` tyyppimuuttujalla. Tyyppi `nat` vastaa Coqissa luonnollista lukua. Määritelmällä pituuden nolla vektori voidaan luoda vain konstruktorilla `vnil`. Lisäksi sitä suurempien listojen muodostus `vcons`-konstruktorilla vaatii jo olemassa olevan vektorin, jolloin listan oikea pituus välttämättä seuraa tyypin mukana. Esimerkiksi tyypin `vec bool 1` määritelmä voidaan Coqissa luoda edellä mainitulla vektorin määritelmällä seuraavasti:

```
vcons bool O true (vnil bool)
```

Näin riippuvan tyypityksen avulla voidaan luoda tyyppimääritelmiä, jotka pitävät huolta tiettyjen invarianttien pitävyydestä. Voisimme esimerkiksi seuraavasti huolehtia siitä, että funktion saamat argumentit ja paluutyyppi ovat aina samanpituisia:

```
Definition first (n : nat) (a : vec bool n) (b : vec bool n)
: vec bool n :=
  a.
```

Funktiossa vain yksinkertaisesti valitaan kahdesta annetusta argumentista ensimmäinen. Määritelmän tyypitys kuitenkin pitää huolta siitä, että sen argumentit sekä paluutyyppi ovat aina samanpituisia, koska ne tyyppimääritelmässä indeksoidaan samalla luonnollisella luvulla `n`. Vastaavasti voisimme muokata määritelmää siten, että paluutyypin on oltava aina annettua argumenttia yhden suurempi:

```
Definition larger (n : nat) (a : vec bool n)
: vec bool (S n) :=
```

```
    vcons true a.
```

Muokkaus onnistuu käyttämällä vektorin määritelmää ja lisäämällä argumenttina annettuun vektoriin arvo `true`. Määritelmässä tyyppitarkastaja pitää huolta siitä, että tyyppimääritelmän invariantit pitävät. Jos olisimme esimerkiksi määritelleet funktion sisällön edellisen määritelmän tapaan vain palauttamalla parametrin `a`, ei tyyppitarkastaja olisi hyväksynyt määritelmää, koska palautetun vektorin pituus ei olisi ollut annetun argumentin vektorin pituutta pidempi. Seurauksena virheelle oltaisiin saatu seuraava Coqin tyyppitarkastimen virheilmoitus:

```
In environment
n : nat
a : vec bool n
The term "a" has type "vec bool n"
while it is expected to have type "vec bool (S n)"
```

Esitettyä periaatetta noudattamalla voidaan Coqissa lisätä määritelmiin invariantteja, joiden halutaan pitävän. Tyyppitarkastin pitää huolta monimutkaistenkin invarianttien noudattamisesta riippuvan tyypityksen avulla.

2.4 Tyyppiuniversumit

Universumit ovat tyyppejä, joita voidaan muodostaa vain tyyppejä luovilla operaatioilla (Harper ja Pollack 1991). Toisin sanoen, ne ovat “tyyppien tyyppejä”. Näitä universumeja kutsutaan Coqissa *Sorteiksi* (Coq Development Team 2020).

Coqissa myös kaikilla tyypeillä on oma tyyppinsä (Coq Development Team 2020). Mutta jos kerran kaikilla tyypeillä on tyyppinsä, niin mikä sitten on tyyppien tyyppi? Ongelmaa voidaan lähestyä määrittelemällä ensin tyyppi `Type`, joka kattaa kaikki tyyppien tyypit. Määrittelyn etuna on se, että koska kaikkien tyyppien tyyppien tulee siihen kuulua, saadaan selkeä rajaus niiden koolle. Valitettavasti myös haittapuoli on selkeä: näin määritelty tyyppiuniversumi sisältää myös itsensä, mikä johtaa Girardin paradoksiin ja rikkoo kielen johdonmukaisuuden (Coquand 1986). Coqissa itsensä sisältävää tyyppiuniversumia `Type` voidaan

käyttää komentoriviargumentin `-type-in-type` avulla.

Jotta kieli voidaan pitää johdonmukaisena, ratkaistaan tyyppiuniversumien muodostus Coqissa äärettömän ja hierarkkisen tyyppiuniversumien joukon avulla. Coqin tyyppijärjestelmä rakentuu perustasorttien (engl. *base sorts*) sekä tyyppiuniversumien `Type` pohjalle. Coqin dokumentaatio (Coq Development Team 2020) esittelee, että perustasortit muodostuvat loogisten propositioiden tyypistä `Prop`, tiukkojen propositioiden (engl. *strict propositions*) tyypistä `SProp` sekä pienien joukkojen, joka sisältää esimerkiksi yleisesti ohjelmoinnissa käytetyt tyypit `bool` ja `nat`, tyypistä `Set`. Perustasorttien lisäksi on Coqin tyyppijärjestelmässä ääretön määrä hierarkkisia tyyppiuniversumeja `Type(i)`, jossa $i \geq 1$. Sorttien joukko voidaan määritellä yhtälön 2.1 mukaisesti (Coq Development Team 2020).

$$S \equiv \{SProp,\ Prop,\ Set,\ Type(i) \mid i \in N\} \tag{2.1}$$

Tärkeää tyyppiuniversumien muodostuksessa on myös universumien keskinäinen rakenne. Coqin dokumentaatio (Coq Development Team 2020) määrittelee kaikkien perustasorttien tyypiksi tyypin `Type(1)`. Lisäksi, jotta itseviittaus voidaan välttää, määritellään tyypin `Type(i)` tyypiksi `Type(i+1)`. Kokonaisuutena tyyppiuniversumeiksi muodostuu yhtälön 2.2 esittämä rakenne.

$$SProp : Type.1,\ Prop : Type.1,\ Set : Type.1,\ Type.i : Type.i + 1,\ i \geq 1 \tag{2.2}$$

Rakentamalla tyyppitasot hierakkisesti ja äärettömästi, mikään tyyppi ei enää ole itse itsensä tyyppi ja Girardin paradoksi vältetään. Lisäksi Coq pystyy hoitamaan tyyppitasojen määrittelyt itse, eikä käyttäjän yleensä tarvitse niitä manuaalisesti asettaa. Näin ollen voidaan, ainakin käytännön ohjelmoinnin näkökulmasta katsoen, kielen käyttäjänä usein ajatella, että tyypin `Type` tyyppi on `Type`. (Coq Development Team 2020)

2.5 Universumipolymorfismi

Universumipolymorfismi on kielilaajennos Coqille, joka tekee mahdolliseksi geneeristen määritelmien kirjoittamisen ja uudelleenkäytön eri tasoisilla universumeilla (Coq Development Team 2020). Coqin virallisen dokumentaation versio 8.12 huomauttaa myös, että universumipolymorfismin tila on Coqissa vielä kokeellinen. Vaikka universumipolymorfismi

ei sinänsä ole olennainen osa Coqia kielenä, vaan vain kielilaajennos, käsitellään sitä tässä teoriaosuudessa, koska se on tärkeä osa tutkielmassa toteutettavaa kirjastoa. Esitellyissä koodiesimerkeissä käytetään pohjana Coqin dokumentaation esimerkkejä.

Yksinkertainen esimerkki universumipolymorfismista voidaan esittää esimerkiksi identiteettifunktion avulla:

```
Definition identity (A : Type) (a : A) :=
  a.
```

Koska määritelmät ovat Coqissa oletuksena universumimonomorfisia eivätkä universumipolymorfisia, asetetaan määritelmälle tyyppi aliluvun 2.4 esimerkkien mukaisesti. Näin ollen saadaan Coqin toimesta uutena vaatimuksena se, että määritelmän muodostamien tyyppien tulee olla pienempiä kuin asetettu tyyppi (Sozeau ja Tabareau 2014). Toisin sanoen, jos määritelmän `identity` tyyppi olisi esimerkiksi `Type(3)`, tulee sen parametrin `A` tyypin olla pienempi kuin `Type(3)`. Tällöin sopiva tyypinpäättelyn toteamus voisi olla esimerkiksi `nat : Set`, `Set : Type(1)` tai `Type(1) : Type(2)`. Esimerkkinä päätelmistä voidaan muodostaa seuraavat määritelmät, jotka Coqin tyyppitarkastin hyväksyy:

```
Definition identityNat :=
  identity nat 1.
Definition identitySet :=
  identity Set bool.
Definition identityType :=
  identity Type Set.
```

Määritelmä muuttuu kuitenkin ongelmalliseksi, jos sille yritetään antaa argumenttina itsensä:

```
Fail Definition selfid :=
  identity _ identity.
```

Huomioitavaa määritelmässä on alaviivan käyttö, jolla annetaan Coqin tyyppitarkastajan päätellä funktion kutsumiseen tarvittava tyyppi implisiittisesti ilman, että käyttäjän sitä tarvitsee funktiolle tarjota. Lisäksi komennolla `Fail` suoritetaan normaalitilanteessa virheellinen komento onnistuneesti ja annetaan tuloksena teksti, joka varmistaa komennon epäonnis-

tumisen sekä antaa saadun virheilmoituksen. (Coq Development Team 2020)

Määritelmä epäonnistuukin seuraavalla tyyppitarkastimen virheilmoituksella:

```
The command has indeed failed with the message:

The term "identity" has type
"forall A : Type@{identity.u0}, A -> A" while it is
expected to have type "?A" (unable to find a well-typed
instantiation for "?A": cannot ensure that
"Type@{identity.u0+1}" is a subtype of
"Type@{identity.u0}").
```

Ongelmana on, että määritelmää `identity` rakentaessaan luo Coq universumin, jota pienempi sen argumenttien tulee olla. Tässä tapauksessa toinen parametri `a` on itse funktio `identity`, joten merkitsemättä jätetyn ensimmäisen parametrin `A` tulee olla, Coqin tyyppitasojen sääntöjen mukaisesti, sitä suurempi. Näin ollen tyyppitarkastimen virheen mukaisesti ei voida hyväksyä sitä, että parametrin `A` tyypin indeksi on `identity.u0+1`, kun itse määritelmälle se on sitä pienempi `identity.u0`.

Tyyppitasojen vaikeuksia pystytään helpottamaan käyttämällä universumipolymorfismia. Universumipolymorfismia voidaan käyttää joko asettamalla se käyttöön kielilaajennoksena komennolla `Set Universe Polymorphism` tai aloittamalla haluttu määritelmä avainsanalla `Polymorphic`. (Coq Development Team 2020)

Esitelty identiteettifunktio voidaan määritellä universumipolymorfiseksi esimerkiksi seuraavasti:

```
Polymorphic Definition pid (A : Type) (a : A) :=
  a.
```

Määritelmän avulla voidaan antaa funktiolle `pid` toisena argumenttina se itsensä ja aiemmasta eroten määritelmä hyväksytään tyyppitarkastimen toimesta:

```
Definition selfpid (A : Type) (a : A) :=
```

```
pid _ pid.
```

Coqin dokumentaation mukaan määritelmä onnistuu, koska universumipolymorfismin ansiosta määritelmän tyyppitasoa ei sidota globaalisti, kuten normaalisti tehdään, vaan ainoastaan määritelmän tasolla. Näin ollen määritelmää voidaan uudelleenkäyttää eri tyyppitasoilla (Coq Development Team 2020). Tyyppitasojen hyödyntäminen voidaan huomata myös tarkastelemalla luotua `selfpid`-määritelmää komennon `Print` avulla, joka antaa seuraavat tyypitystiedot:

```
selfpid =
pid@{selfpid.u0}
  (forall A : Type@{selfpid.u1}, A -> A)
  pid@{selfpid.u1}
    : forall A : Type@{selfpid.u1}, A -> A
(* {selfpid.u1 selfpid.u0} |= selfpid.u1 < selfpid.u0 *)
```

Tyypityksistä voidaan huomata, että määritelmää `pid` käytetään nyt kahdella eri tyyppitasolla: `selfpid.u0` ja `selfpid.u1`. Tämä eroaa aiemmasta virheellisestä määritelmästä, jossa määritelmää käytetään vain tyyppitasolla `identity.u0`. Lisäksi tyyppitasoille asetetaan yhtälön 2.3 mukainen rajoite, joka pitää huolen siitä, että argumenttina annettu `selfpid` on tyyppitasoltaan itse määritelmän tyyppitasoa pienempi.

$$selfpid_{u1}\ selfpid_{u0} \models selfpid_{u1} < selfpid_{u0} \tag{2.3}$$

Luvun esityksen mukaisesti voidaan universumipolymorfismin avulla tyyppitasoja ja rajoitteita käyttämällä luoda määritelmiä, joita Coqin tyyyppitarkastin ei normaalitilanteessa hyväksyisi. Tässä tutkielmassa universumipolymorfismin hyötynä on erityisesti se, että kirjastossa tarvittavia määritelmiä ei tarvitse toistaa eri tyyppitasoilla, vaan universumipolymorfismia voidaan käyttää hyödyksi yhden, tyyppitasojen suhteen geneerisen, määritelmän luomiseen.

2.6 Todistustila

Kuten tutkielmassa on jo aiemmin mainittu, voidaan Coqia käyttää myös todistusassistenttina. Todistusassistenttina Coqin toimintaa voidaan kuvata interaktiivisena toimintana käyttäjän ja assistentin välillä, jossa käytetään hyödyksi todistustaktiikoita, joiden avulla todistuksessa voidaan siirtyä eri vaiheisiin (Coq Development Team 2020).

Todistusassistenttina Coqia käytetään hyödyntämällä Coqin todistustilaa. Todistustilaan voidaan siirtyä esimerkiksi käyttämällä avainsanaa `Theorem` tai päättämällä tavallinen funktiomääritelmä pisteeseen. Lisäksi, jos todistus on onnistunut, voidaan todistustilasta poistua esimerkiksi komennon `Qed` tai `Defined` avulla. Coqin dokumentaatio havainnollistaa komentojen eroa esittämällä, että komennolla `Qed` varmennettavat todistukset eivät ole käytettävissä varsinaisessa laskennassa, vaan ne ainoastaan tallennetaan läpinäkymättöminä vakioina. Läpinäkymättömyys mahdollistaa rinnakkaisuuden ja todistusten laiskan tarkistamisen (Coq Development Team 2020). Läpinäkymättöminä vakioina tallentaminen ei ole haitallista esimerkiksi matemaattisia todistuksia tehdessä, mutta koska tutkielman kirjaston toteutuksessa todistustilaa käytetään hyödyksi nimenomaan laskentaa suorittavien määritelmien luomiseen, käytetään niissä kaikissa avainsanaa `Defined`.

Todistustilan käyttökelpoiseen hyödyntämiseen tarvitaan myös jonkinlaista interaktiivista tilaa, missä todistusta on helpompi seurata. Coqissa interaktiivisia todistustiloja ovat esimerkiksi CoqIDE, Proof General sekä Coqin mukana tuleva Coqtop (Coq Development Team 2020). Coqtopin yksinkertaisuuden vuoksi on sen avulla selkeää käydä läpi helppo esimerkki, mikä esitetään seuraavaksi. Esimerkki esittelee todistustilan olennaisimmat ominaisuudet tiivistetysti.

Coqtop käynnistetään komentoriviltä komennolla ‘coqtop‘. Käynnistämisen jälkeen voidaan määritellä todistettavaksi asetettava teoreema. Esimerkissä todistettavana on yksinkertaisesti, että $2+2=4$. Esimerkeissä harmaalla taustalla olevat kuvaukset vastaavat interaktiivisen tilan antamia syötteitä ja muut käyttäjän komentoja. Todistus aloitetaan määrittelemällä todistettava teoreema:

```
Coq < Theorem two_plus_two_eq_4 : 2 + 2 = 4.
```

Määritelmän esittelyn jälkeen siirrytään todistustilaan ja todistuksen tila tulee käyttäjälle näkyviin. Todistuksen tila kertoo käyttäjälle todistuksessa jäljellä olevat vaiheet sekä niille asetetut tavoitteet:

```
1 subgoal
  ============================
  2 + 2 = 4
```

Tila kertoo käyttäjälle, että koko lauseen todistusta varten on jäljellä yksi todistettava lause, jonka tavoite on $2+2=4$. Koska esimerkkitodistus on yksinkertainen, on tässä vaiheessa riittävää vain todistaa se taktiikalla `reflexivity`:

```
two_plus_two_eq_4 < reflexivity.
```

Koska annettu taktiikka on riittävä ainoan tavoitteena olevan lauseen todistamiseen, ilmoittaa interaktiivinen tila, että jäljellä ei ole todistettavia tavoitteita:

```
No more subgoals.
```

Nyt todistus voidaan päättää avainsanalla `Qed`, jolloin Coq tallentaa todistuksen:

```
two_plus_two_eq_4 < Qed.
```

Lisäksi, koska todistus on viimeistelty komennolla `Qed`, voidaan sitä myös tarkastella komennon `Print` avulla:

```
Coq < Print two_plus_two_eq_4.
```

Tuloksena komento esittää annetun termin määritelmän:

```
two_plus_two_eq_4 = eq_refl
  : 2 + 2 = 4
```

Esitetyillä komennoilla voidaan suorittaa yksinkertaisen lauseen todistus. Monimutkaisempien lauseiden todistus on usein huomattavasti työläämpää, mutta esitetty esimerkki kattaa perusperiaatteet, joita todistuksien muodostaminen todistustilan avulla vaatii. Esimerkistä

pystyy huomaamaan myös todistustilan vuorovaikutteisen luonteen: käyttäjä esittää todistustaktiikoita, joiden perusteella todistuksen tavoitteet ja tila muuttuvat. Toimintaperiaatteeltaan todistustila muistuttaa siten esimerkiksi logiikassa käytettyä luonnollista päättelyä.

3 Geneerinen ohjelmointi

Tässä luvussa esitellään ensin geneeristä ohjelmointia yleisesti, minkä jälkeen tarkastellaan erityisesti parametrista polymorfismia, tietotyyppigeneerisyyyttä, ariteettigeneerisyyttä sekä ariteetti- ja tietotyyppigeneerisyyttä. Näitä geneerisyyden muotoja esitellään, koska niillä on olennainen rooli tutkielman kirjaston toiminnassa. Ariteetti- ja tietotyyppigeneerisyyden kohdalla näin on, koska tutkielmassa rakennetaan ariteetti- ja tietotyyppigeneerinen kirjasto. Parametrista polymorfismia taas käsitellään, koska useissa kirjaston määritelmissä käytetään sitä hyödyksi ja koska siihen liittyvät konseptit ovat helposti rinnastettavissa muihin geneerisyyden muotoihin.

3.1 Geneerisyys yleisesti

Geneerisellä ohjelmoinnilla pyritään parantamaan ohjelmointikielen joustavuutta turvallisesti. Kielen turvalliseen joustavuuteen voidaan kuitenkin päästä useilla eri keinoilla, minkä vuoksi geneerinen ohjelmointi voidaan myös ymmärtää monin eri tavoin. Lisäksi erilaisia geneerisen ohjelmoinnin muotoja on olemassa useita. Yleensä geneerisyys kuitenkin ilmenee operaatioiden parametrisaation kautta, yleistämällä operaation toiminnan yhdellä määritelmällä usealle eri ilmentymälle. (Gibbons 2007)

Useista ohjelmoinnin muodoista huolimatta, on geneerisessä ohjelmoinnissa yhdistävinä pidettyjä tavoitteita. Lämmel ja Jones (2003) määrittelevät yhdeksi yhteiseksi tavoitteeksi toisteisen ja yksinkertaisen lähdekoodin määrän vähentämisen. Lähdekoodin määrän vähentämisen myötä geneerisestä ohjelmoinnista on ohjelmoijan kannalta hyötyä puuduttavan työmäärän vähenemisen, koodin uudelleenjärjestelyn helpottumisen sekä yksinkertaisten virheiden todennäköisen määrän vähenemisen kautta (Lämmel ja Jones 2003). Lisäksi Gibbons (2007) määrittelee geneerisen ohjelmoinnin tavoitteeksi ohjelmointikielen joustettavuuden lisäämisen. Hänen mukaansa joustettavuuden lisäämisen tulisi myös tapahtua vaarantamatta kielen turvallisuutta.

Geneerisyydestä kannattaa huomioida myös se, että eri geneerisyyden muodot ja käytetyt termit ajoittain vaihtelevat, mikä voi aiheuttaa sekaannusta. Esimerkiksi Gibbons (2007) mää-

rittelee geneerisen ohjelmoinnin parametrisaationa tietorakenteen rakenteen eikä sen sisällön suhteen. Toisaalta muun muassa C# yleensä käsittää geneerisen ohjelmoinnin tarkoittavan erityisesti yhtä geneerisyyden muotoa: parametrista polymorfismia (Kennedy ja Syme 2000). Tässä tutkielmassa geneerisyydellä tarkoitetaan nimenomaan Gibbonsin laajempaa määritelmää. Lisäksi eri geneerisyyden muodot erotellaan tutkielmassa toisistaan erillisillä termeillä, jotka ovat: parametrinen polymorfismi, universumipolymorfismi, ariteettigeneerisyys, tietotyyppigeneerisyys sekä kaksi viimeistä geneerisyyden muotoa yhdistävä ariteetti- ja tietotyyppigeneerisyys. Muut geneerisyyden muodot jätetään tutkielman ulkopuolelle. Lisäksi universumipolymorfismia tarkastellaan erillisesti aliluvussa 2.5, koska tutkielman toteutuksen kannalta olennaisinta on erityisesti sen käyttö Coqissa kielilaajennoksena.

3.2 Parametrinen polymorfismi

Yksi konkreettinen ja yleinen geneerisen ohjelmoinnin muoto on parametrinen polymorfismi, jossa funktion toiminta ei voi olla riippuvainen tyyppiargumentin sisällöstä. Funktio ei siten voi tarkastella polymorfisten argumenttiensa sisältöä vaan se pystyy ainoastaan uudelleenjärjestelemään niitä (Gibbons 2007). Esimerkiksi universumipolymorfismia käsitelleessä aliluvussa 2.5 esitelty identtiteettifunktio on parametrisesti polymorfinen, koska siinä samaa funktiota voidaan kutsua useilla eri tyypin argumenteilla. Samoin esimerkiksi listoja yhdistävä funktio `append` voidaan muodostaa parametrista polymorfismia hyödyntäen Coqissa seuraavasti:

```
Fixpoint append (A : Type) (a : list A) (b : list A)
: list A :=
  match a with
  | nil => b
  | cons x xs => x (append _ xs b)
  end.
```

Olennaista parametrisessa polymorfismissa on funktion parametrisoiminen tyyppimuuttujalla. Esitetyssä malliesimerkissä parametrisointi tapahtuu parametrin `A` avulla, minkä seurauksena funktiota voidaan käyttää usean eri tyypin listoilla. Määritelmää `append` voidaan käyttää esimerkiksi seuraavasti sekä luonnollisilla luvuilla että totuusarvoilla:

```
Definition lnat1 := cons 1 (cons 2 nil).
Definition lnat2 := cons 3 nil.

Definition lbool1 := cons true nil.
Definition lbool2 := cons false nil.
```

Esimerkeillä voidaan suorittaa laskentaa komennon `Compute` avulla, jolloin saadaan tuloksena seuraavat kommentoidut arvot:

```
Compute app _ lnat1 lnat2.
  (* = (1 :: 2 :: 3 :: nil) : list nat *)

Compute app _ lbool1 lbool2.
  (* = (true :: false :: nil) : list bool *)
```

Esitetyistä funktioista sekä niiden käytöstä voidaan selkeästi huomata se, miten parametrisen polymorfismin käyttö edesauttaa toisteisen koodin vähentämistä: ei ole tarvetta muodostaa erillistä funktiota jokaiselle tyypille, vaan kaikille voidaan käyttää samaa määritelmää.

Parametrisen polymorfismin tarjoama yleistettävyys toiminnan suhteen voi johtaa myös laajempiin seurauksiin, mistä mielenkiintoisena esimerkkinä on Wadlerin yleistämä huomio siitä, että voimme jokaiselle samaa tyyppiä olevalle polymorfiselle funktiolle johtaa teoreeman, jonka ne kaikki toteuttavat. Englanniksi huomiota kutsutaan termillä *parametricity theorem* ja popularisoituna sanonnalla *theorems for free*. (Wadler 1989)

3.3 Tietotyyppigeneerisyys

Tietotyyppigeneerisyyden käsitteen esitteli Gibbons (2007), tarkoituksenaan erottaa se selkeästi muista geneerisyyden muodoista. Tietotyyppigeneerisyys erotetaan esimerkiksi parametrisesta polymorfismista siinä, että tietotyyppigeneerisyydessä parametrisaatiota ei tehdä tietorakenteen sisällön suhteen, kuten parametrisessa polymorfismissa, vaan sen oman rakenteen suhteen. Tietotyyppigeneerisyys voidaan tiivistetysti määritellä datan rakenteen hyödyntämisenä. (Gibbons 2007)

Eri lähtökohtia tietotyyppigeneerisyyden toteutukseen on useita. Magalhães ja Löh (2012) tarjoavat esimerkkinä Haskell-ohjelmointikielen, jossa on laaja valikoima kirjastoja tietotyyppigeneeristä ohjelmointia varten. Runsauden taustalla on selkeästi parhaan vaihtoehdon puute sekä eri kirjastojen vaihtelevat vahvuudet ja heikkoudet. Heidän mukaansa runsaus onkin haitta tietotyyppigeneeriseen ohjelmointiin tutustuvan käyttäjän kannalta, koska jo omiin tarpeisiin sopivan kirjaston valinta aiheuttaa selkeää lisätyötä. (Magalhães ja Löh 2012)

Tietotyyppigeneerisyys lisää ohjelmointikielten joustavuutta, mikä helpottaa lähdekoodin uudelleenjärjestelyä sekä laajempaa uudelleenkäyttöä (Gibbons 2007; Vries ja Löh 2014). Vries ja Löh (2014) havainnollistavat tietotyyppigeneerisyyden lisäämää joustavuutta toteamalla, että tietotyyppigeneerisessä ohjelmoinnissa määritelmät mukautuvat muuttuviin tietotyyppeihin tarpeen mukaan, mikä saavutetaan käyttämällä hyödyksi tietotyyppien rakennetta operaation määrittelyssä.

Vaikka tietotyyppigeneerisyyttä voidaan hyödyntää esimerkiksi Haskellissa kielilaajennosten kautta, onnistuu tietotyyppigeneerisyyden esittäminen riippuvalla tyypityksellä selkeästi vain kielen perusominaisuuksia käyttäen. Tietotyyppigeneerisyyden esittäminen tarvitsee syntaktisen esitysmuodon geneerisille tyypeille ja funktiot, jotka muuttavat ilmentymiä näistä esitysmuodoista lähdekielen tyyppeihin. Syntaktiset esitysmuodot voidaan muodostaa esimerkiksi induktiivisilla tyypeillä. Esitysmuotoja kutsutaan joskus *universumityypeiksi*, mutta ne eivät tarkoita samaa kuin aliluvun 2.4 Coqin tyyppijärjestelmään kuuluvat sisäänrakennetut tyyppiuniversumit. (Chlipala 2013)

Esitetään seuraavaksi yksinkertainen konkreettinen esimerkki tietotyyppigeneerisyydestä käyttämällä Coqia. Aloitetaan muodostamalla syntaktinen esitysmuoto tietotyyppigeneerisyyttä varten, mikä voidaan tehdä luomalla uusi induktiivinen tyyppi seuraavasti:

```
Inductive Ty : Type :=
  | TUnit : Ty
  | TNat : Ty
  | TProd : Ty -> Ty -> Ty
  | TSum : Ty -> Ty -> Ty.
```

Muodostettujen tyyppikonstruktorien avulla pystytään esittämään Coqin tyyppejä. Esitystä

varten tarvitaan kuitenkin myös funktio, joka kääntää syntaktisen esitysmuodon konstruktoreja Coqin tyypeiksi. Käännösfunktio voidaan määritellä esimerkiksi seuraavasti:

```
Fixpoint decode (t : Ty) : Type :=
  match t with
  | TUnit => unit
  | TNat => nat
  | TProd a b => prod (decode a) (decode b)
  | TSum a b => sum (decode a) (decode b)
  end.
```

Muodostetuilla määritelmillä saadaan koottua kaikki tarvittavat rakennuspalat yksinkertaisen geneerisen määritelmän rakentamiseksi. Esimerkiksi identiteettifunktio voidaan niiden avulla esittää seuraavasti:

```
Fixpoint gid (t : ty) (x : decode t) : decode t :=
  x.
```

Funktiota on mahdollista sen tyyppimääritelmän nojalla kutsua millä tahansa arvolla, jonka tyyppi on `decode t`. Esimerkiksi tyypin `nat` syntaktista esitysmuotoa `TNat` ja tyypin `unit * nat` esitysmuotoa `TProd TUnit TNat` voidaan funktion käytössä hyödyntää seuraavasti:

```
Compute gid TNat 5.
  (* = 5 : decode TNat *)
Compute gid (TProd TUnit TNat) (tt, 3).
  (* = (tt, 3) : decode (TProd TUnit TNat) *)
```

Muodostetun syntaktisen esitysmuodon rajoitukset on kuitenkin selkeät ja huomattavat: tyyppejä ei voida esittää rekursiivisesti, joten esitysmuoto rajoittuu yksinkertaiseen esitysmuodon rakennusosien toistoon. Lisäksi esitysmuodosta puuttuu muuttujat. Kattavan ariteetti- ja tietotyyppigeneerisyyden ilmaisua varten tarvitaan itse tutkielman kirjaston toteutuksessa huomattavasti ilmaisuvoimaisempaa esitysmuotoa: kindeilla indeksoitua universumia.

3.4 Ariteettigeneerisyys

Ariteettigeneerisyys eroaa edellä esitellyistä parametrisesta polymorfismista sekä tietotyyppigeneerisyydestä siinä, että ariteettigeneerisyydessä operaatiot yleistetään niiden parametrien määrän suhteen. Ariteettigeneerisyydestä löytyy lukuisia esimerkkejä muun muassa dynaamisesti tyypitetystä ja funktionaalisesta Scheme-ohjelmointikielestä. Ariteettigeneerisyys on kuitenkin huomattavasti harvinaisempaa staattisesti tyypitetyissä kielissä niiden tyyppijärjestelmien vaatimusten vuoksi. Riippuvasti tyypitetyissä kielissä on kuitenkin käyttäjän mahdollista ohjelmoida tyyppitasolla, mikä helpottaa ariteettigeneerisyyden esittämistä. (Weirich ja Casinghino 2010)

Ariteettigeneerisyyttä ei ole akateemisissä lähteissä käsitelty laajasti. Weirich ja Casinghino (2010) käsittelevät artikkelissaan ariteettigeneerisyyttä Agdassa osana ariteetti- ja tietotyyppigeneerisen kirjastonsa luontia, Allais (2019) puolestaan keskittyy artikkelissaan nimenomaan ariteettigeneerisyyteen Agdassa. Lisäksi Fridlender ja Indrika (2000) käyvät läpi ariteettigeneerisyyttä Haskellissa muutaman esimerkkifunktion kautta ja Serrano ja Miraldo (2018) laajemmin Haskellin kielilaajennosten avulla.

Seuraavaksi havainnollistetaan ariteettigeneerisyyttä staattisesti tyypitetyissä funktionaalisissa ohjelmointikielissä esimerkkien kautta. Fridlender ja Indrika (2000) käsittelevät artikkelissaan ariteettigeneerisyyttä seuraavien Haskellin oletuskirjastosta (Marlow 2010) löytyvien funktioiden avulla:

```
repeat   :: a -> [a]
repeat x = xs where xs = x:xs

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

zipWith          :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith z (a:as) (b:bs)
                 = z a b : zipWith z as bs
zipWith _ _ _    = []
```

```
zipWith3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
zipWith3 z (a:as) (b:bs) (c:cs)
                  = z a b c : zipWith3 z as bs cs
zipWith3 _ _ _ _ = []
```

Funktioista `repeat` toistaa annettua argumenttia ikuisesti, `map` taas suorittaa argumenttina annetun funktion kaikille toisena argumenttina annetuille lista-alkioille, muodostaen niiden tuloksesta uuden listan. Jäljelle jäävät `zipWith` ja `zipWith3` toimivat vastaavasti, mutta eri määrällä argumentteja. Fridlender ja Indrika (2000) suosivatkin nimiä `zipWith0` ja `zipWith1` Haskellin standardikirjaston `repeat` ja `map` sijaan.

Määritelmiin pohjautuen Fridlender ja Indrika (2000) huomauttavat, että esitetyt funktiot ovat kaikki ilmentymiä samasta operaatiosta, niillä on vain eri *ariteetti* eli niiden parametrien määrä vaihtelee. Tämän huomion he yleistävät yhdeksi ariteettigeneeriseksi määritelmäksi `zipWith`, jota he esittävät seuraavasti:

```
zipWith :: (a1 -> ... -> an -> b) ->
                    [a1] -> .. -> [an] -> [b]
zipWith f (a1:as1) ... (an:asn)
                  = f a1 ... an : zipWith f as1 ... asn
zipWith _ _ ... _ = []
```

Haskellia käyttäessään he kuitenkin joutuvat turvautumaan erityisesti funktioita varten määriteltyjen numeraalien käyttöön (Fridlender ja Indrika 2000). Weirich ja Casinghino (2010) puolestaan esittävät kuinka määritelmä voidaan luoda suoraviivaisemmin riippuvalla tyypityksellä ja hyödyntämällä luonnollisia lukuja tyyppimääritelmien indeksoinnissa.

Vaikka ariteettigeneerisyys ei funktioissa ole yhtä yleistä kuin tietotyyppigeneerisyys, on silti useampia funktioita, jotka ovat ariteettigeneerisiä. Esimerkiksi Fridlender ja Indrika (2000) määrittelevät `map` esimerkkinsä lisäksi ariteettigeneerisen `taut` funktion, joka on tosi kaikilla syötteillä. Lisäksi Weirich ja Casinghino (2010) esittelevät ariteetti- ja tietotyyppigeneeriset, eli siten myös ariteettigeneeriset, `eq` ja `unzip` funktiot. Funktioista `eq` vertaa syötteiden yhtäsuuruutta, `unzip` taas kääntää annetun rakenteen ja tulotyypin sisällä olevat arvot

vastakkaisesti rakenteen sisään ja yhdistää ne tulotyypiksi.

3.5 Ariteetti- ja tietotyyppigeneerisyys

Ariteetti- ja tietotyyppigeneerisessä, tai tuplasti geneerisessä (engl. *doubly-generic*), ohjelmoinnissa yhdistetään ariteettigeneerisyys ja tietotyyppigeneerisyys. Yhdistämisen seurauksena saadaan geneerisiä määritelmiä, joita voidaan kutsua sekä eri tyypeillä että eri määrällä argumentteja. (Weirich ja Casinghino 2010)

Koska jo pelkästään ariteettigeneerisyyttä käsitellään kirjallisuudessa vain vähän, on luonnollisesti ariteetti- ja tietotyyppigeneerisyydestä niukasti lähdemateriaalia. Ainoat tutkielman lähdekatsauksessa löydetyt ariteetti- ja tietotyyppigeneerisyyyttä käsittelevät lähteet olivat *Arity-Generic Datatype-Generic Programming* (Weirich ja Casinghino 2010) sekä *Generic Programming of All Kinds* (Serrano ja Miraldo 2018). Lisäksi Serrano ja Miraldo mainitsivat artikkelissaan, että heidän toteutukseensa ariteettigeneerisyyden lisäävä osuus oli hyvin samankaltainen Weirichin ja Casinghinon toteutuksen kanssa.

4 Tutkimus

Tutkielman osana toteutetaan pienimuotoinen ariteetti- ja tietotyyppigeneerinen kirjasto Coq-todistusassistentilla. Toteutus tehdään käyttämällä tutkielmaan sovellettua suunnittelutieteen mallia, jonka tuloksena saadaan uusi ohjelmistoartefakti. Toteutuksen lisäksi artefaktia arvioidaan suunnittelutieteen malleja apuna käyttäen. Tässä luvussa esitellään ensin suunnittelutiedettä yleisemmin, minkä jälkeen käydään tarkemmin läpi sen sovellusta tutkimuksessa.

4.1 Suunnittelutiede

Hevner ym. (2004) jakavat suurimman osan tietojärjestelmien tutkimuksesta käyttäytymis- ja suunnittelutieteisiin. Suunnittelutiede pyrkii heidän mukaansa kehittämään ihmisten ja organisaatioiden kykyjä erityisesti uusien artefaktien luomisen kautta, toisin kuin käyttäytymistiede, joka on kiinnostunut ihmisiin ja organisaatioihin liittyvien mallien muodostuksesta ja todentamisesta (Hevner ym. 2004). March ja Smith (1995) puolestaan jakavat tietojärjestelmien tutkimuksen luonnontutkimukseen ja suunnittelutieteeseen. Heidän määrittelynsä mukaan suunnittelutieteessä pyritään luomaan uusia asioita, jotka ovat ihmisille hyödyksi ja luonnontieteessä, joka kattaa myös käyttäytymistieteet, taas pyritään ymmärtämään todellisuutta (March ja Smith 1995).

Käyttäytymis- ja suunnittelutieteelliset näkökannat eivät ole toisiaan poissulkevia, vaan molemmat tukevat toisiaan onnistuneen kokonaisuuden luomisessa. Hevner ym. (2004) havainnollistavat tätä esittämällä, että suunnittelutieteen näkökannan painotuksen haittana on teknologian ylipainotus, jolloin sen teoriapohja jää liian vähälle huomiolle. Toisaalta käyttäytymistieteen painotus puolestaan heikentää kykyä arvioida teknologisten vaatimusten ja ominaisuuksien tilaa, mikä voi näyttäytyä vanhojen ja puutteellisten teknologioiden arviointina sekä niiden käyttönä teorioiden pohjana. Näin ollen molempia näkökulmia tarvitaan onnistuneiden kokonaisuuksien muodostamiseen. (Hevner ym. 2004)

Suunnittelutieteellinen tutkimus rakentuu kahden vaiheen varaan: rakentamiseen ja arviointiin. Rakentamisvaiheen avulla voidaan osoittaa, että artefaktin muodostus on ylipäänsä annetuissa rajoitteissa mahdollista. Arviointi taas mahdollistaa arviointikriteerien muodosta-

misen ja käyttämisen toteutetun artefaktin arvioinnissa. Vaiheita voidaan verrata myös luonnontieteisiin, joissa vastaavasti tutkimus perustuu löytöihin tai hypoteeseihin, joita pyritään oikeuttamaan. (March ja Smith 1995)

Hevner ym. (2004) esittävät seitsemän ohjenuoraa suunnittelutieteellisen tutkimuksen tekemistä varten. Ohjenuorat on esitetty kokonaisuudessaan taulukossa 1. Ohjenuoriin liittyen he kuitenkin huomauttavat myös, että eivät suosittele ohjeiden orjallista noudattamista, vaan suosivat tutkimuksen tekijöiden käyttävän omia taitojaan sekä luovuuttaan päätelläkseen, mitä näistä ohjenuorista tutkimukseen olisi soveliasta hyödyntää. Ohjenuorien esittelyn taustalla on edesauttaa suunnittelutieteen vaatimusten ymmärtämistä ja siten he suosivat ohjenuorien huomioimista suunnittelutieteellistä tutkimusta tehdessä, jotta tutkimus olisi onnistunut. (Hevner ym. 2004)

Suunnittelutieteellisen tutkimuksen tuloksena saatu artefakti ei ole rajoittunut vain ohjelmistoihin. March ja Smith (1995) jakavat artefaktit karkeasti neljään eri luokkaan: konstrukteihin, malleihin, metodeihin ja ilmentymiin. Konstruktit viittaavat kielellisiin tulkintoihin ja symboleihin, mallit esimerkiksi abstrakteihin esitysmuotoihin, metodit algoritmeihin ja työtapoihin ja ilmentymät toteutuksiin ja prototyyppisysteemeihin. (March ja Smith 1995)

Hevner ym. (2004) esittävät arviointivaiheen tärkeänä osana tutkimusprosessia, jossa varmistetaan, että artefakti täyttää sille asetetut vaatimukset. Heidän mallinsa, joka näkyy kokonaisuudessaan taulukossa 2, mukaan arviointia voidaan suorittaa usean eri ominaisuuden suhteen. Ominaisuuksia voivat olla esimerkiksi suorituskyky, käytettävyys, toiminnallisuus ja artefaktin sopivuus suhteutettuna organisaation toimintaan. He esittävät myös, että arviointia voidaan suorittaa usealla eri tavalla, tärkeää on kuitenkin muistaa huomioida arvioidun artefaktin ominaisuudet ja käyttötarkoitukset, jotta arviointi on kohdennettu oikeita ominaisuuksia varten. (Hevner ym. 2004)

4.2 Suunnittelutiede tässä tutkielmassa

Tutkielmassa toteutetaan artefaktina prototyyppimäinen ariteetti- ja tietotyyppigeneerinen kirjasto käyttäen ohjelmointikielenä funktionaalista ja riippuvasti tyypitettyä Coqia. Artefaktin toteutuksen tarkoituksena on tuottaa ariteetti- ja tietotyyppigeneerinen kirjasto Coqil-

Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Taulukko 1. Suunnittelutieteen ohjenuorat (Hevner ym. 2004)

le, koska sellaista ei ole saatavilla. Täten tutkimuksen avulla pystytään tuottamaan uutta tietoa. Lisäksi erona muihin ariteetti- ja tietotyyppigeneerisiin kirjastoihin, käytetään tutkielman toteutuksessa hyödyksi universumipolymorfismia.

Tutkielman toteutus perustuu taulukon 1 suunnittelutieteen ohjenuoriin (Hevner ym. 2004). Artefaktin toteuttaminen tehdään muodostamalla konkreettinen ohjelmistokirjasto. Ongel-

1. Observational	Case Study: Study artifact in depth in business environment.
	Field Study: Monitor use of artifact in multiple projects.
2. Analytical	Static Analysis: Examine structure of artifact for static qualities (e.g., complexity).
	Architecture Analysis: Study fit of artifact into technical IS architecture.
	Optimization: Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behavior.
	Dynamic Analysis: Study artifact in use for dynamic qualities (e.g., performance).
3. Experimental	Controlled Experiment: Study artifact in controlled environment for qualities (e.g., usability).
	Simulation - Execute artifact with artificial data.
4. Testing	Functional (Black Box) Testing: Execute artifact interfaces to discover failures and identify defects.
	Structural (White Box) Testing: Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation.
5. Descriptive	Informed Argument: Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact's utility.
	Scenarios: Construct detailed scenarios around the artifact to demonstrate its utility.

Taulukko 2. Suunnittelutieteen arviointimallit (Hevner ym. 2004)

man relevanssiin liittyen tutkielmassa toteutetaan kirjasto ohjelmointikielelle, jolle vastaavaa ei vielä ole saatavilla ja tästä näkökulmasta toteutus on myös relevantti. Tutkielmassa

ei kuitenkaan huomioida suositusta, että ongelman tulisi olla myös tärkeä ja relevantti liiketoiminnan kannalta. Tähän liittyen tutkielmassa näkökulmana on ettei maisterintutkielman aiheen tulisi olla määriteltävissä ainakaan ensisijaisesti liiketoiminnan ehtojen nojalla. Toteutetulle artefaktille suoritetaan myös arviointia suunnittelutieteiden periaatteiden mukaisesti ja tutkimuksen täsmällisyydestä huolehditaan noudattamalla Jyväskylän yliopiston tarjoamia ohjesääntöjä sekä seuraamalla esiteltyjä suunnittelutieteen periaatteita toteutukseen sovellettuna. Tutkielmassa pyritään huomioimaan tutkielman kommunikaatiosta siten, että se olisi ymmärrettävissä myös aiheeseen tutustumattomille. Vaikeutena kommunikaatiossa on tutkielman aiheen luonne, jonka ymmärrys vaatii kohtuullisen paljon teknistä tutustumista ja joka on myös suhteellisen vähän käsitelty. Ymmärrystä on pyritty edesauttamaan muodostamalla esimerkit konkreettisesti kaavioiden ja koodiesimerkkien kautta sekä pitämällä esimerkit, silloin kun mahdollista, tarkoituksellisen yksinkertaisina.

Ohjenuorista, jotka Hevner ym. (2004) esittelevät, ei ainoana tutkimuksessa huomioida suunnittelun roolia etsintäprosessina. Ohjenuorassa suunnittelutiedettä kuvataan iteratiivisena prosessina, jossa toiston ja kokeilujen kautta pyritään löytämään tehokas ratkaisu ongelmaan. Tutkielmassa suunnittelun roolia etsintäprosessina ei käytetä periaatteena, koska toteutuksen luonne on prototyyppimäinen ja prioriteettina on saada aikaan toimiva versio. Erillisten versioiden muodostus ja vertailu ei ole tutkielman puitteissa mahdollista aikarajoitteiden vuoksi.

Toteutetun artefaktin arvioinnin pohjana käytetään taulukossa 2 esiteltyjä arvioinnin metodeja. Arvioinnin periaatteina toimii artefaktin analyyttinen arviointi sekä staattisesti että dynaamisesti. Arvioinnin perustana on artefaktille asetetut tavoitteet. Tavoitteiden mukaan artefaktin tulee olla toimiva, universumipolymorfismia hyödyntävä ja Coqilla toteutettu ariteetti- ja tietotyyppigeneeristä ohjelmointia tukeva kirjasto. Tavoitteiden täyttymistä arvioidaan dynaamisesti toteuttamalla esimerkit tietotyyppigeneerisestä sekä ariteetti- ja tietotyyppigeneerisestä funktiosta kirjaston avulla. Lisäksi esimerkkejä suoritetaan niiden toiminnan testaamista varten. Arviointia toteutetaan myös staattisesti arvioimalla lähdekoodia sen koon ja rakenteen suhteen. Lähdekoodin arvioinnissa apuna on Coqin tyyppijärjestelmän ilmaisukyky, jonka seurauksena määritelmien tyyppitiedot sisältävät paljon luotettavaa informaatiota määritelmien toiminnasta. Lisäksi toteutettua lähdekoodia verrataan toteutuksen pohjana

oleviin kirjastoihin erityisesti määritelmien koon ja monimutkaisuuden suhteen.

5 Kirjaston toteutus

Kirjaston toteutuksen tuloksena saatiin toimiva ariteetti- ja tietotyyppigeneerinen kirjasto. Kirjasto toteutettiin Coqin versiolla 8.11.0 ja se on julkisesti saatavilla (Pehkonen 2021). Tässä luvussa käydään ensin lyhyesti läpi kirjaston toteutuksen lähtökohtia, minkä jälkeen tarkastellaan tarkemmin kirjaston rakennetta ja toteutusta muun muassa koodiesimerkkien ja kuvioiden avulla. Lisäksi esitellään kirjaston määritelmien avulla luodut malliesimerkit tietotyyppigeneerisestä sekä ariteetti- ja tietotyyppigeneerisestä funktiosta. Lopuksi käydään vielä lyhyesti läpi muita samankaltaisia toteutuksia.

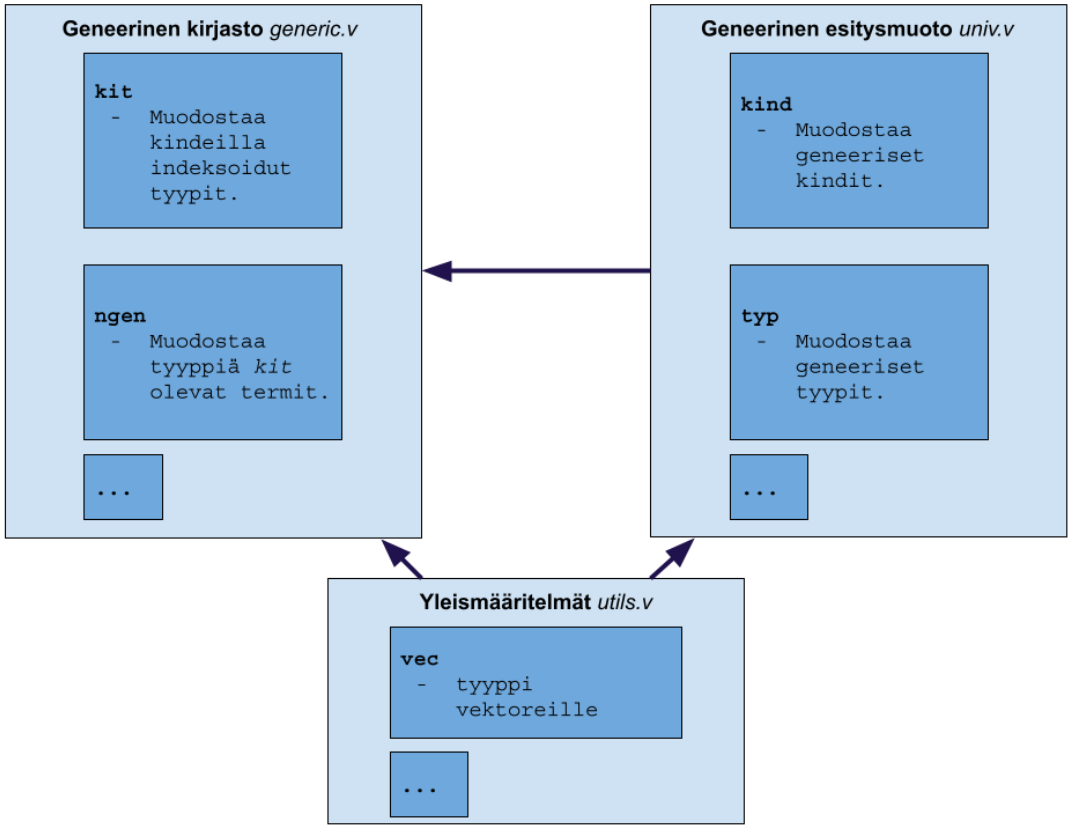
5.1 Kirjaston lähtökohdat

Weirich ja Casinghino (2010) toteuttivat ariteetti- ja tietotyyppigeneerisen kirjaston käyttämällä Agdaa, joka on myös riippuvasti tyypitetty ohjelmointikieli. Lisäksi Verbruggen, Vries ja Hughes (2008) toteuttivat tietotyyppigeneerisen kirjaston Coqilla perustuen toteutukseen, jonka alkuperäisesti esitteli Hinze (2002) käsitellessään tietotyyppigeneeristä ohjelmointia. Tutkielman kirjasto pohjautui rakenteeltaan näihin toteutuksiin.

Kirjastossa pyrittiin vastaamaan erityisesti yhteen puutteeseen, jonka Weirich ja Casinghino (2010) nostivat toteutuksestaan esille: lähdekoodin määritelmien selkeyttämiseksi jouduttiin siinä käyttämään `type-in-type`-kielilaajennosta, mikä rikkoo tyyppijärjestelmän johdonmukaisuuden eikä kirjastossa siten pystytä luotettavasti hyödyntämään todistusmenetelmiä. Weirich ja Casinghino toteuttivat kirjastosta myös johdonmukaisen version, mutta siinä määritelmiä jouduttiin uudelleenmäärittelemään eri tyyppitasoilla, mikä lisäsi toisteisen lähdekoodin määrää. Johdonmukaisuuden säilyttämiseksi ja turhan toiston välttämiseksi käytettiin tutkielman toteutuksessa hyödyksi Coqin kielilaajennoksena tarjoamaa universumipolymorfismia, mikä mahdollisti toisteisten määritelmien välttämisen pitäen kuitenkin kirjaston määritelmät luotettavina todistusmenetelmiä varten. Universumipolymorfismin käyttöä ehdottivat artikkelissaan myös Weirich ja Casinghino (2010) mahdollisena ratkaisuna määritelmien muodostukselle.

5.2 Kirjaston rakenne

Kirjaston lähdekoodi on jaoteltu kolmeen erilliseen tiedostoon: `univ.v`, `generic.v` sekä `utils.v`. Kirjaston geneerisen esitysmuodon sisältää `univ.v`. Geneerisen esitysmuodon avulla muodostetaan ne kirjaston funktiot ja tyyppitason määritelmät, joilla varsinaisia geneerisiä funktioita voidaan muodostaa. Geneeristen funktioiden muodostukseen tarvittavat määritelmät puolestaan sijatsevat tiedostossa `generic.v`. Lisäksi tiedostossa `utils.v` on yleisiä käytettyjä tietorakenteita ja funktioita, joita Coqin standardikirjasto ei joko tarjoa ollenkaan tai joille toteutuksessa nähtiin olevan selkeämpää tarjota itse määritellyt vaihtoehdot. Kirjaston rakennetta ja olennaisimpia määritelmiä esitellään kuviossa 1.



Kuvio 1. Kirjaston yleisrakenne. Nuolet osoittavat tiedostojen välisiä riippuvuuksia.

Toteutetun kirjaston lisäksi tutkielman lähdekoodi sisältää esimerkit tietotyyppigeneerisen sekä ariteetti- ja tietotyyppigeneerisen funktion määrittelystä kirjaston avulla. Esimerkit löytyvät tiedostosta `examples.v`. Tiedoston sisällä on esimerkit jaoteltu Coqin `Section`-komennon avulla erillisiin nimettyihin jaksoihin `dtgen` ja `aritydtgen`. Tiedosto sisältää

myös esimerkkejä funktioiden käytöstä sekä niiden antamista tuloksista. Lisäksi tiedostossa `proofs.v` on muutama yksinkertainen todistus tyypeistä, joita geneerisen esitysmuodon avulla voidaan muodostaa.

5.3 Geneerinen esitysmuoto

Tietotyyppigeneerisen kirjaston luominen vaatii geneerisen esitysmuodon tyypeille. Kuten luvussa 3.3 jo mainittiin, kutsutaan geneeristä esitysmuotoa usein *tyyppiuniversumiksi*, mutta tässä tutkielmassa sitä kutsutaan joko syntaktiseksi tai geneeriseksi esitysmuodoksi, jotta selkeästi erotettaisiin se luvun 2.4 tyyppiuniversumeista. Lisäämällä käännösfunktiot geneerisille esitysmuodoille voidaan luoda funktioita, jotka operoivat niiden rakenteen mukaan. Geneeriset esitysmuodot ja niiden käännösfunktiot mahdollistavat useilla eri tyypeillä toimivien geneeristen funktioiden määrittelyn. Tutkielmassa toteutetun geneerisen esitysmuodon lähdekoodi löytyy kirjaston tiedostosta `univ.v`.

Kirjaston pohjana on kindeilla indeksoitu geneerinen esitysmuoto. Geneeristä esitysmuotoa varten tarvitsee Coqin avulla muodostaa kolme siihen sisältyvää rakennetta: kindit, tyyppivakiot ja lambdakalkyyli. Lambdakalkyyli on formalismi, jonka avulla voidaan esittää anonyymeja funktiota ja joka on useiden funktionaalisten ohjelmointikielten perustana. Yhdistettynä näiden rakenteiden esityskyky on riittävä ariteetti- ja tietotyyppigeneerisyyden esittämiseen.

Kindit, eli tyyppien tyypit, esitetään Coqin induktiivisten tyyppien avulla. Tutkielman kirjastossa ne määritellään seuraavasti:

```
Inductive kind : Type :=
  | Ty : kind
  | F : kind -> kind -> kind.
```

Kindeja voidaan muodostaa kahdella konstruktorilla: joko itsenäisen konstruktorin `Ty` avulla tai käyttämällä konstruktoria `F`, joka tarvitsee argumentteina lisäksi kaksi muuta kindia.

Koska määriteltyjen kindien avulla pystytään esittämään tyyppien tyypit, voidaan niiden avulla muodostaa tietorakenteet myös geneerisen esitysmuodon varsinaisille tyypeille. Yksinkertaisimpana vaiheena tässä on tyyppivakioiden esittäminen. Tyyppivakiot ovat atomisia

tyyppejä, jotka voidaan muodostaa suoraan joko itsensä tai muiden määriteltyjen tyyppivakioiden avulla. Tyyppivakiot määritellään kirjastossa seuraavasti:

```
Inductive const : kind -> Type :=
  | Nat : const Ty
  | Unit : const Ty
  | Sum : const (F Ty (F Ty Ty))
  | Prod : const (F Ty (F Ty Ty)).
```

Tyyppivakioista `Nat` ja `Unit` ovat yksinkertaisimmat, sillä niiden tyyppi on `const Ty`. Toisaalta `Sum` ja `Prod` tarvitsevat molemmat kaksi tyyppiargumenttia tyyppinsä muodostamiseen, joten niiden tyyppi on hieman monimutkaisempi `const (F Ty (F Ty Ty))`.

Kindien ja tyyppivakioiden jälkeen tarvitaan geneerisen esitysmuodon toteutusta varten vielä tietorakenteet lambdakalkyylille sekä muuttujille. Tyyppimuuttujia varten tulee lisäksi ensin määritellä konteksti, josta kindit voidaan hakea. Konteksti määritellään tutkielman kirjastossa listana kindeja:

```
Definition ctx : Type :=
  list kind.
```

Kirjastossa tyyppimuuttujat muodostetaan käyttämällä kontekstia ja kindia indekseinä. Lisäksi tyyppimuuttujien konstruktoreissa `Vz` ja `Vs` käytetään hyödyksi De Bruijn -indeksointia, minkä avulla muuttujat voidaan yhdistää kindeihin ilman niiden nimeämistä. Kokonaisuudessaan tyyppimuuttujat määritellään tutkielman kirjastossa seuraavasti:

```
Inductive tyvar : ctx -> kind -> Type :=
| Vz : forall G k, tyvar (k :: G) k
| Vs : forall G k k', tyvar G k -> tyvar (k' :: G) k.
```

Kindien, tyyppivakioiden ja tyyppimuuttujien määrittelyn jälkeen tarvitsee enää yhdistää kokonaisuus geneeriseksi esitysmuodoksi lambdakalkyylin avulla. Geneerinen esitysmuoto `typ` määritellään kirjastossa seuraavasti:

```
Inductive typ : ctx -> kind -> Type :=
| Var : forall G k,
```

```
        tyvar G k -> typ G k
| Lam : forall G k1 k2,
        typ (k1 :: G) k2 -> typ G (F k1 k2)
| App : forall G k1 k2,
        typ G (F k1 k2) -> typ G k1 -> typ G k2
| Con : forall G k,
        const k -> typ G k.
```

Määritelmässä `Var` tarkoittaa muuttujaa, `Lam` lambda abstaktiota, `App` lambda applikaatiota ja `Con` tyyppivakiota. Geneerisen esitysmuodon `typ` avulla voidaan esittää kielen vakiona tarjottavia tyyppejä sekä muita monimutkaisempia tyyppejä. Esimerkiksi Coqin `sum` ja `unit` tyypit geneerisen esitysmuodon avulla muodostetaan seuraavasti:

```
Definition tsumc :=
  fun ctx => Con ctx Sum.
Definition tunitc :=
  fun ctx => Con ctx Unit.
```

Hieman monimutkaisempaa on määritellä tyyppejä, jotka eivät ole suoraan esitettävissä tyyppivakioiden avulla. Vaikeus johtuu muun muassa siitä, että geneerisen esitysmuodon avulla tyyppien määrittely vaatii ymmärrystä myös lambdakalkyylista, sillä määritelmät esitetään sen kielen mukaisesti. Kuitenkin esimerkiksi Coqin `option`-tyyppi voidaan geneerisen esitysmuodon avulla esittää seuraavasti:

```
Definition tmaybe : ty (F Ty Ty) :=
  Lam
    (App
      (App tsumc tunitc)
        (Var (Vz _ _))).
```

Olennaista määritelmän toiminnassa on muuttujien käyttö, koska tyyppi ei ole suoraan tyyppivakioiden avulla esitettävissä. Lisäksi määritelmässä käytetään `Sum`-tyyppiä hyödyksi, jotta pystytään erottelemaan ne tapaukset, joissa tyyppi sisältää tyyppiargumenttia vastaavan arvon niistä, joissa näin ei ole. Esimerkiksi Coqissa vastaava tehdään `option`-tyypin kohdalla

konstruktoreilla `None` ja `Some`.

Samaan tyyliin voidaan myös kaksi tyyppimuuttujaa ottava `sum` määritellä geneerisen esitysmuodon avulla käyttämättä suoraan tyyppivakiota `Sum`:

```
Definition teither : ty (F Ty (F Ty Ty)) :=
   Lam (Lam
     (App (App tsumc
       (Var (Vs _ (Vz _ _)))) (Var (Vz _ _)))).
```

Käyttämällä Coqin todistusominaisuuksia voidaan myös todistaa, että tyyppi `sum` todella vastaa tyyppiä `teither`:

```
Theorem sum_eq_teither (A B : Type)
  : sum A B = decodeClosed teither A B.
Proof.
  unfold decodeClosed; unfold decodeType; simpl.
  reflexivity.
Qed.
```

Viimeisenä vaiheena geneerisen esitysmuodon osalta tarvitaan funktiot, jotka kääntävät geneeriset tyypit Coqin omiin tyyppeihin. Muutoksia varten muodostettiin kirjastossa käännösfunktiot kindeille ja tyypeille seuraavasti:

```
Fixpoint decodeKind (k : kind) : Type :=
  match k with
  | Ty => Type
  | F k1 k2 => decodeKind k1 -> decodeKind k2
  end.

Fixpoint decodeType (k : kind) (G : ctx) (t : typ G k)
: env G -> decodeKind k :=
  match t in typ G k return env G -> decodeKind k with
  | Var _ _ x => fun e =>
                   slookup x e
```

```
| Lam _ _ _ t1 => fun e =>
                    fun y => decodeType t1 (econs _ y e)
| App _ _ _ t1 t2 => fun e =>
                       (decodeType t1 e)
                       (decodeType t2 e)
| Con _ _ c => fun e =>
                 decodeConst c
end.
```

Määritelmässä näyttämättä jätetty `slookup` etsii muuttujan ympäristöstä ja `decodeConst` kääntää geneerisen muodon tyyppivakion Coq tyypiksi. Määritelmien avulla onnistuu kirjastossa tyyppien muunnos geneerisen esitysmuodon ja tavallisten Coqin tyyppien välillä.

Kokonaisuutena luvussa käsiteltyjen esitysmuotojen avulla pystytään rakentamaan tarvittavat määritelmät ariteetti- ja tietotyyppigeneeristen funktioiden muodostusta varten. Olennaista geneerisessä esitysmuodossa on itse rakennetun esitysmuodon hyödyntäminen kielen tavallisten tyyppien esittämisessä sekä muunnosfunktioiden käyttö eri esitysmuotojen välillä liikkumisessa.

5.4 Ariteetti- ja tietotyyppigeneerinen kirjasto

Geneerisen esitysmuodon määrittelyjen jälkeen voidaan muodostaa varsinainen ariteetti- ja tietotyyppigeneerinen kirjasto. Kirjastosta esitellään olennaisimmat geneerisyyteen vaadittavat tietorakenteet ja funktiot. Toiminnaltaan määritelmistä tärkeimmät ovat tyyppien erikoistamisen toteuttava `kit` sekä termien erikoistamisen määrittelevä `ngen`. Määritelmien suuren lukumäärän vuoksi jäävät monet esittelemättä, mutta määritelmät löytyvät kokonaisuudessaan tutkielman toteutuksen lähdekoodin tiedostosta `generic.v`.

Keskeistä ariteetti- ja tietotyyppigeneerisen kirjaston luomisessa on käyttää hyödyksi määriteltyjä geneerisiä esitysmuotoja sekä niiden käännösfunktioita, jotta voidaan muodostaa kaikille geneeristä esitysmuotoa oleville argumenteille toimivia geneerisiä funktioita. Geneerisiä funktioita varten muodostettiin tutkielman kirjastossa tyyppitason funktio `kit`, eli *kind-indexed type*, joka määrittelee geneerisyyteen tarvittavan tyypin annettujen argument-

tien perusteella. Konkreettisesti se toiminnallaan muuntaa annetun tyyppejä sisältävän vektorin uudeksi tyyppimääritelmäksi. Tutkielman kirjastossa `kit` on määritelty seuraavasti:

```
Fixpoint kit (n : nat) (k : kind)
  (b : vec Type (S n) -> Type) :
  vec (decodeKind k) (S n) -> Type :=
  match k return vec (decodeKind k) (S n) -> Type with
  | Ty => fun vs => b vs
  | F k1 k2 => fun vs => quantify
    (fun As => kit k1 b As -> kit k2 b (zap vs As))
  end.
```

Olennaista määritelmän toiminnassa on parametrin `b` käyttö, joka määrittelee tyypin muodon perustapauksille. Sitä hyödyntäen `kit` muodostaa tarvittavan tyypin tyyppejä sisältävän vektorin avulla, kääntäen ne uudeksi funktiotyypiksi.

Määritelmää voidaan selkeyttää esimerkin kautta, käyttämällä argumenttina funktiota `Map`, joka määritellään seuraavasti:

```
Definition Map : vec Type 2 -> Type :=
  fun vs => vhd vs -> vhd (vtl vs).
```

Esitetyssä määritelmässä `Map` yksinkertaisesti muuntaa kaksi vektorissa olevaa tyyppiä funktiotyypiksi. Esimerkiksi tyypit `nat` ja `unit` sisältävä vektori muuttuisi funktiotyypiksi `nat -> unit`. Funktioita `Map` ja `kit` hyödyntämällä voidaan määritellä tietotyyppigeneeristen funktioiden paluutyypit. Esimerkiksi ylläoleva esimerkki voidaan laskea Coqissa suoraan myös funktion `kit` avulla:

```
Compute kit Ty Map (vcons nat (vcons unit (vnil _))).
  (* = nat -> unit
    : Type *)
```

Seuraava tärkeä vaihe ariteetti- ja tietotyyppigeneerisyyden rakentamisessa on käyttää `kit`-määritelmää paluutyyppinä uudessa funktiossa, joka sisältää geneerisyyden kannalta olennaisen toiminnallisuuden. Funktio ottaa argumentteina geneeristä esitysmuotoa olevan tyy-

pin sekä vaaditun toiminnan kaikille geneerisen esitysmuodon vakioille. Niiden avulla se voi muodostaa funktion, joka erikoistaa geneeriseen määritelmään tarvittavan termin. Weirich ja Casinghino (2010) kutsuvat termin erikoistavaa funktiota nimellä `ngen` ja Verbruggen, Vries ja Hughes (2008) nimellä `specTerm`. Tutkielman kirjastossa funktio on `ngen` ja se määritellään seuraavasti:

```
Definition ngen (n : nat) (b : vec Type (S n))
(k : kind) (t : ty k) (ce : tyConstEnv b)
: kit k b (repeat (decodeClosed t)) :=
  eqkit _ _ (ngen' _ nnil ce).
```

Keskeistä määritelmässä `ngen` on parametri `t`, joka esittää geneerisen esitysmuodon tyyppistä termiä tyhjässä kontekstissa sekä parametri `ce`, joka määrittelee miten funktio toimii geneerisen esitysmuodon vakioille. Lisäksi `ce` tarvitsee oman tyyppitason määritelmänsä `tyConstEnv`. Suurin osa määritelmän varsinaisesta työstä tapahtuu funktiossa `ngen'`, joka on jätetty tutkielman kirjallisesta osuudesta pois sen laajuuden sekä monimutkaisuuden vuoksi. Määritelmä löytyy kuitenkin tarvittaessa tutkielman kirjaston lähdekoodista. Sisällöltään se määrittelee miten tarvittava termi rakennetaan parametrin `t` konstruktoreille. Lisäksi siinä tulee huomioida mahdollisten tyyppimuuttujien vuoksi konteksti, jota ei määritelmässä `ngen` tarvitse tehdä, koska sitä kutsuttaessa on konteksti vielä suljettu. Kontekstin puuttuminen näkyy myös tyyppimääritelmän indeksoinnissa termillä `decodeClosed t`.

Määritelmät `kit` ja `ngen` ovat riittäviä ariteetti- ja tietotyyppigeneeristä kirjastoa varten. Niiden muodostus vaatii useiden muiden määritelmien käyttöä, mutta kokonaisuutena ne sisältävät kaiken kirjaston kannalta olennaisen toiminnallisuuden. Määritelmien avulla geneerisiä funktioita muodostaessa tulee kirjaston käyttäjän rakentaa funktion toiminta kaikille geneerisen esitysmuodon vakioille sekä esittää tarvittava tyyppi perustapauksille. Kirjaston käytöstä esitetään seuraavaksi esimerkki laatimalla tietotyyppigeneerinen funktio kirjaston avulla.

5.5 Tietotyyppigeneerinen `gmap`

Koska kirjasto on ariteetti- ja tietotyyppigeneerinen, voidaan sen avulla muodostaa myös pelkästään tietotyyppigeneerisiä tai pelkästään ariteettigeneerisiä funktioita. Esimerkkinä käydään läpi tietotyyppigeneerisen funktion muodostus kirjaston määritelmien avulla.

Esimerkkifunktiona toimii tietotyyppigeneerinen funktio `gmap`, joka löytyy kokonaisuudessaan tutkielman lähdekoodin tiedostosta `examples.v`. Olennaista funktion toiminnassa on, että sille argumenttina annettu funktio suoritetaan kaikille toisen argumentin sisältämille arvoille. Esimerkiksi, jos argumenttina olisi identiteettifunktio ja pari, suoritettaisiin identiteettifunktio molemmille parin arvoille ja funktion palautusarvo olisi sama kuin alkuperäinen pari.

Tietotyyppigeneerisen funktion `gmap` muodostukseen tarvitaan tyyppitason määritelmänä aiemmin luvussa 5.4 esiteltyä määritelmää `Map`. Määritelmän lisäksi tulee kirjaston käyttäjän tarjota funktion toiminta kaikille geneerisen esitysmuodon vakioille, mikä voidaan tehdä esimerkiksi seuraavasti:

```
Definition doSum (A1 B1 : Type)
  : (A1 -> B1) -> forall (A2 B2 : Type),
    (A2 -> B2) -> (A1 + A2) -> (B1 + B2) :=
    fun f =>
      fun _ _ g sa =>
        match sa with
        | inl a => inl (f a)
        | inr b => inr (g b)
        end.

Definition doProd (A1 B1 : Type)
  : (A1 -> B1) -> forall (A2 B2 : Type),
    (A2 -> B2) -> (A1 * A2) -> (B1 * B2) :=
    fun f _ _ g sa =>
      (f (fst sa), g (snd sa)).
```

```
Definition mapConst : tyConstEnv Map :=
  fun k c =>
    match c in const k
    return kit k Map (repeat _ (decodeClosed (Con _ c)))
    with
    | Nat => fun n => n
    | Unit => fun _ => tt
    | Prod => doProd
    | Sum => doSum
    end.
```

Määritelmissä `doProd` ja `doSum` muodostavat geneeristen vakioiden toimnnan tulo- ja summatyypeille ja ne on määritelty erillisissä funktioissa selkeyden vuoksi. Lisäksi määritelmien muodostus erillisesti helpottaa tyyppimääritelmien selvittämistä ja siten tyyppitarkastajan läpäisemistä. Määrittelyjen avulla voi käyttäjä muodostaa tietotyyppigeneerisen funktion `gmap` antamalla määritellyn funktion `mapConst` argumenttina termin erikoistavalle funktiolle `ngen`:

```
Definition gmap (k : kind) (t : ty k)
: kit k Map _ :=
  ngen _ t (@mapConst).
```

Muodostettu määritelmä on tietotyyppigeneerinen ja sitä varten tarvitsee käyttäjän vain muodostaa tyyppitason määritelmä sekä vakioiden toiminta. Geneeristä määritelmää voisi käyttää esimerkiksi parin tai aiemmin luvussa 5.3 itse muodostetun tyypin `tmaybe` avulla esimerkiksi seuraavasti:

```
Compute tprod _ _ (fun a => a + 1)
              _ _ (fun b => negb b)
                  (1, true)
  (* = (2, false) *)
Compute gmap tmaybe _ _ (fun _ => false)
                        (inl tt).
  (* = inl () *)
```

```
Compute gmap tmaybe _ _ (fun n => n + 1)
                        (inr 3).
  (* = inr 4 *)
```

Käsitellyssä luvussa on esitelty miten kirjaston avulla voidaan luoda tietotyyppigeneerinen funktio `gmap`. Seuraavaksi esitetään hieman monimutkaisempi ariteetti- ja tietotyyppigeneerinen funktio `ngmap`, joka toimii mallina ariteetti- ja tietotyyppigeneerisen funktion määrittelystä kirjaston avulla.

5.6 Ariteetti- ja tietotyyppigeneerinen `ngmap`

Ariteetti- ja tietotyyppigeneerisen määritelmän luonti kirjaston avulla on esiteltyä tietotyyppigeneeristä funktiota `gmap` monimutkaisempaa. Suurin osa monimutkaisuudesta johtuu vaikeuksista geneerisen esitysmuodon vakioiden toiminnan määrittelyssä, mikä puolestaan seuraa Coqin tyyppitarkastimen asettamista vaatimuksista.

Jotta tietotyyppigeneerisestä funktiosta `gmap` saataisiin muodostettua ariteetti- ja tietotyyppigeneerinen versio, tulee `kit` indeksoida eri tyyppitason määritelmällä. Aiemmasta, luvussa 5.4 esitetystä, määritelmästä `Map` voidaan huomata, että sen tyyppi on `vec Type 2 -> Type`. Näin ollen siihen on kiinnitettynä parametrien määrä eikä `gmap` siten voi olla geneerinen niiden suhteen. Parametrien määrän geneerisyyden puute korjataan määrittelemällä uusi tyyppitason funktio `nMap`, jolla `kit` indeksoidaan. Tutkielman lähdekoodin esimerkissä `nMap` määritellään seuraavasti:

```
Fixpoint nMap {n : nat} (v : vec Type (S n)) : Type :=
  match v with
  | @vcons _ O x xs => x
  | @vcons _ (S n') x xs => x -> nMap xs
  end.
```

Määritelmästä voidaan huomata, että nyt vektorina saatavat tyypit indeksoidaan parametrin `n` avulla eikä niiden määrää siten ole sidottu muuten, kuin että vektorin `v` tulee sisältää vähintään yksi tyyppi. Määritelmä vastaa kuitenkin toimintansa suhteen määritelmää `Map` ja erikoistapauksessa $n = 2$ antavat ne samoilla argumenteilla kutsuttaessa myös samat tyypit

tuloksena. Tyyppien vastaavuus voidaan myös todistaa käyttämällä Coqin todistustilaa:

```
Theorem map_eq_nmap2 (v : vec Type 2)
: Map v = nMap v.
Proof.
  unfold Map.
  dependent destruction v;
  dependent destruction v.
  reflexivity.
Qed.
```

Todistuksessa on huomionarvoista taktiikan `dependent destruction` hyödyntäminen, joka olettaa heterogeenisen yhtäsuuruuden aksiooman `JMeq_eq` muodossa.

Tyyppitason määritelmän muokkauksen lisäksi tulee määritellä uudelleen myös funktion toiminta kaikille geneerisen esitysmuodon vakioille, mikä aiheutti tutkielman toteutuksessa selkeitä ongelmia. Vaikeutena vakioiden toiminnan määrittelyssä oli ennen kaikkea Coqin tyyppitarkastimen hyväksynnän saanti. Vakioiden toimintojen määrittelyssä jouduttiin usein turvautumaan myös Coqin todistustilan käyttöön. Todistustilan käytön etuna oli Coqin todistustaktiikoiden hyödyntäminen, mikä helpotti tyypintarkastukseen liittyviä ongelmia, mutta samalla ajoittain heikensi määritelmien luettavuutta ja selkeyttä. Vakioiden määrittelyssä käytettiin usein myös vastaavaa rakennetta kuin esimerkeissä, jotka Weirich ja Casinghino (2010) esittelevät omassa toteutuksessaan. Silti huomattavana erona näihin määritelmiin oli se, että heidän määritelmänsä olivat yleensä yhden tai kahden rivin pituisia, kun taas tutkielman esimerkkien määritelmät venyivät usein selkeästi pidemmiksi. Määritelmien koon eroavaisuuksissa syynä oli käytettyjen kielien erot ja varsinkin tyyppitarkastimien vaatimukset riippuvasti tyypitettyjen määritelmien tarkastuksessa.

Toiminnan määrittely `Unit`-vakiolle oli yksinkertaista. Tutkielman esimerkissä toiminta määriteltiin seuraavasti:

```
Fixpoint cUnit (n : nat)
  : kit Ty nMap (repeat (n:=(S n))
    (decodeClosed (Con [] Unit))) :=
```

```
  match n with
  | O => tt
  | S n' => fun x => cUnit n'
  end.
```

Sen sijaan `Nat`-vakion määrittely aiheutti enemmän vaikeuksia ja sen tuloksena olikin selkeästi vastaavaa tietotyyppigeneeristä määritelmää monimutkaisempi versio. Vakio on tutkielman esimerkissä määritelty seuraavasti:

```
Fixpoint cNat (n : nat)
: kit Ty nMap (repeat (n:=S n)
(decodeClosed (Con [] Nat))) :=
  let f := (fix cNat' (n' : nat)
    : kit Ty nMap (repeat (n:=(S n'))
      (decodeClosed (Con [] Nat))) :=
      match n'
      return kit Ty nMap (repeat (n:=S n')
      (decodeClosed (Con [] Nat))) with
      | O => O
      | S O => fun x => x
      | S (S m) => fun x y => cNat' m
  end) in
  match n
  return kit Ty nMap (repeat (n:=S n)
  (decodeClosed (Con [] Nat))) with
  | O => O
  | S O => fun x => x
  | S (S m) =>
    if Nat.odd m
    then fun x y => cNat m
    else fun x => f (S m)
  end.
```

Ongelmana määritelmässä on Coqin vaatimus siitä, että funktio ei koskaan mene ikuiseen

silmukkaan. Vaatimuksesta johtuen toteutuksessa päädyttiin turvautumaan erilliseen `let`-lauseeseen määritelmän sisällä rekursion päättymisen osoittamiseksi. Toiminnaltaan funktio palauttaa aina viimeiseksi annetun argumentin.

Vakion `Prod` toiminnan kohdalla turvauduttiin Coqin todistustilan käyttöön. Toiminta määriteltiin apufunktiolla `cProd`, joka määrittelee vakion toiminnan muodossa, jossa arvot otetaan vastaan yksi kerrallaan funktiona eikä `prod`-tyyppisenä. Apufunktio määriteltiin tutkielman esimerkissä seuraavasti:

```
Fixpoint cProd (n : nat)
: forall (va : vec Type (S n)), nMap va
-> forall (vb : vec Type (S n)), nMap vb
-> nMap (zap (zap (repeat
        (decodeClosed (Con [] Prod))) va) vb).
  Proof.
    destruct n;
    intros VA a VB b;
    pose proof veq_hdtl (v:=VA) as pfa;
    pose proof veq_hdtl (v:=VB) as pfb.
    - apply pair.
      + rewrite pfa in a; apply a.
      + rewrite pfb in b; apply b.
    - simpl; intros pr.
      destruct pr as [pa pb].
      apply cProd.
      + rewrite pfa in a. apply a. apply pa.
      + rewrite pfb in b. apply b. apply pb.
  Defined.
```

Myös vakion `Sum` toiminnan määrittelyssä turvauduttiin Coqin todistustilaan. Määrittelyssä tarvitut funktiot eivät vaativuudeltaan eronneet vastaavista vakion `Prod` määritelmistä, mutta vakion rakenteen vuoksi sen määrittelystä tuli kokonaisuutena selkeästi laajempi. Lisäksi tulee huomata, että toiminnaltaan ariteetti- ja tietotyyppigeneerinen `ngmap` on osittainen. Funktion suoritus voi johtaa virheeseen, jos sitä kutsutaan vakiossa `Sum` eri konstruktoreil-

la. Virhetilanne käsiteltiin määritelmässä käyttämällä ylimääräistä aksioomaa `error`, jotta määritelmät voitiin pitää selkeämpinä. Aksioomaa käyttävän version lisäksi toteutettiin funktiosta versio `optNgmap`, joka käyttää aksiooman sijaan tyyppiä `option` mahdollisen virheen käsittelyyn.

Määritelmissä vakion `Sum` toiminta eroteltiin ensin muodostamalla apufunktio `cSumLeft`, joka käsittelee vasemmanpuoleisella konstruktorilla `inl` muodostetut tapaukset:

```
Fixpoint cSumLeft (n : nat)
: forall (va : vec Type (S n)),
  nMap va -> forall (vb : vec Type (S n)),
  nMap (zap (zap (repeat sum) va) vb).
Proof.
  intros VA a VB. simpl.
  pose proof veq_hdtl (v:=VA) as pfa;
  pose proof veq_hdtl (v:=VB) as pfb.
  destruct n as [| n'].
  - rewrite pfa in a. simpl in a. apply (inl a).
  - intros x. destruct x as [l | r].
    + rewrite pfa in a. simpl in a.
      apply cSumLeft.
      pose proof a l as pa; apply pa.
    (* error case *)
    + exfalso; apply err.
Defined.
```

Periaatteena funktion toiminnassa on erotella tapaukset argumentin konstruktorin perusteella, mikä tapahtuu komennon `destruct x` avulla. Komennon seurauksena saadaan kaksi tapausta, joista jälkimmäinen on virheellinen, koska se on muodostettu konstruktorilla `inr` ja joka käsitellään aksioomalla `error`.

Lisäksi vakiota `Sum` varten muodostettiin apufunktio `cSumRight`, joka käsittelee oikeanpuoleisella konstruktorilla `inr` muodostetut tapaukset:

```
Fixpoint cSumRight (n : nat)
```

```
: forall (va : vec Type (S n)) (vb : vec Type (S n)),
  nMap vb -> nMap (zap (zap (repeat sum) va) vb).
Proof.
  intros VA VB b. pose proof veq_hdtl (v:=VB) as pfb.
  destruct n as [| n'].
  - rewrite pfb in b. apply (inr b).
  - intros x; destruct x as [left | right].
    (* error case *)
    + exfalso; apply error.
    + rewrite pfb in b. simpl in b.
      pose proof b right as pb.
      pose proof cSumRight n' (vtl VA) (vtl VB) pb as pf.
      apply pf.
Defined.
```

Toiminnaltaan `cSumRight` on hyvin samankaltainen kuin `cSumLeft`. Olennaisena erona on, että nyt vasemmanpuoleisen konstruktorin `inl` tapaus käsitellään virheellisenä.

Muodostettuja erillisiä apufunktioita hyödyntämällä pystyttiin määrittelemään vakioiden toiminta koko `Sum`-vakiolle. Myös koko vakion toimintaan käytettiin Coqin todistustilaa. Vakio määriteltiin tutkielman esimerkissä seuraavasti:

```
Fixpoint cSum (n : nat)
: forall (va : vec Type (S n)), nMap va
-> forall (vb : vec Type (S n)), nMap vb
-> nMap (zap (zap (repeat sum) va) vb).
Proof.
  intros VA a VB b.
  destruct n as [| n'].
  (* choose arbitrary case inr *)
  - simpl. rewrite veq_hdtl in b.
    simpl in b. apply (inr b).
  - intros x; destruct x as [lr | rt].
    + rewrite veq_hdtl in a. simpl in a.
```

```
      pose proof a lr as pfa.
      apply cSumLeft; apply pfa.
    + rewrite veq_hdtl in b. simpl in b.
      pose proof b rt as pfb.
      apply cSumRight; apply pfb.
Defined.
```

Määritelmässä tapauksessa, jossa `n` on 0, valitaan satunnaisesti tapaus `inr`. Muuten määritelmä käyttää hyödyksi apufunktioita `cSumLeft` ja `cSumRight` tarvittavan toiminnan muodostukseen.

Koska `Sum` oli viimeinen määritelmää vailla oleva vakio, on enää jäljellä ainoastaan vakioiden yhdistäminen yhdeksi kaikkien vakioiden toiminnan kattavaksi funktioksi. Kaikkien vakioiden toiminta määriteltiin seuraavasti:

```
Definition nmapConst {n : nat} : tyConstEnv (@nMap n).
  fun k c =>
    match c with
    | Nat => cNat _
    | Unit => cUnit _
    | Prod => ltac:(curryk (cProd (n:=n)))
    | Sum => ltac:(curryk (cProd (n:=n)))
    end.
```

Kaikkien vakioiden toiminnassa ei tarvinnut suoraan käyttää Coqin todistustilaa, koska määritelmä pystyttiin muodostamaan käyttämällä hyödyksi itse rakennettua todistustaktiikkaa `curryk`, joka rakentaa tarvittavan todistuksen vakioille `Prod` ja `Sum` niiden apufunktioiden avulla. Käytetty todistustaktiikka määriteltiin seuraavasti:

```
Ltac curryk const :=
  apply (curryKind (F Ty (F Ty Ty)));
  simpl;
  apply const.
```

Vakioiden määrittelyn jälkeen on selkeästi vaikein osa ohi. Uusien määritelmien avulla voi-

daan ariteetti- ja tietotyyppigeneerinen `ngmap` määritellä yksinkertaisesti seuraavasti:

```
Definition ngmap (n : nat) (k : kind) (t : ty k)
: kit k nMap (repeat (decodeClosed t)) :=
  ngen _ t (@nmapConst n).
```

Ainoa selkeä ero tietotyyppigeneeriseen versioon lopullisessa määritelmässä on se, että nyt tyyppitarkastin ei pysty automaattisesti päättelemään funktiolle `nMap` annetun vektorin kokoa, vaan se tulee esittää eksplisiittisesti paluutyypin tyyppimäärittelmässä. Vektorin koon esitys toteutetaan funktiolla `repeat`, joka toistaa lausekkeen `decodeClosed t` sisältöä parametriin `n` perustuen ja antaa saadun tuloksen vektorissa.

Toteutetuista määritelmistä voidaan huomata, että kokonaisuutena ariteetti- ja tietotyyppigeneerisen funktion luominen on, ainakin funktion `map` tapauksessa, huomattavasti haastavampaa kuin pelkästään tietotyyppigeneerisen funktion määrittely. Suurin haaste ariteetti- ja tietotyyppigeneerisen funktion määrittelyssä vaikuttaisi olevan Coqin käytössä, sillä vastaavien määritelmien muodostus esimerkiksi Agdalla tulisi olla yksinkertaisempaa. Agdaa käyttävien määritelmien suhteellinen yksinkertaisuus voidaan todeta määritelmistä, jotka Weirich ja Casinghino (2010) toteuttivat omassa ariteetti- ja tietotyyppigeneerisessä esimerkissään.

Coqin todistustilan käyttö auttoi selkeästi määritelmien tarkastuksessa, mutta sillä oli myös haittapuolensa, sillä määritelmistä tuli todistustilan käytön seurauksena vaikeammin luettavia. Todistustilaa käyttävien määritelmien läpikäynti onkin järkevintä Coqin todistustilan avulla, suorittamalla todistusprosessi läpi askel kerrallaan.

5.7 Kirjaston käytön rajapinta

Tutkielman kirjaston toteutuksessa muodostettiin myös yksinkertainen rajapinta ariteetti- ja tietotyyppigeneeristen funktioiden määrittelyä varten. Rajapinnan etuna käyttäjälle on selkeät vaatimukset siitä, mitä geneeristen määritelmien luontiin vaaditaan.

Rajapinta toteutettiin käyttämällä hyödyksi Coqin avainsanaa `Record`, jonka avulla voidaan muodostaa listaus vaadittujen määritelmien tyypityksistä. Käyttäjältä vaaditaan funktio, joka muodostaa tyypityksen geneerisen funktion perustapauksille sekä määritelmät, jotka

rakentavat funktion toiminnan geneerisen esitysmuodon tyyppivakioille. Lisäksi tarjottujen määritelmien tulee noudattaa kirjaston asettamia tyyppisääntöjä. Tutkielman toteutuksessa rajapinta määriteltiin seuraavasti:

```
Record NGen (n : nat) : Type := nGen
{
  transform : vec Type (S n) -> Type;
  cunit : kit Ty transform
      (repeat (decodeClosed (Con [] Unit)));
  cnat : kit Ty transform
      (repeat (decodeClosed (Con [] Nat)));
  csum : kit (F Ty (F Ty Ty)) transform
      (repeat (decodeClosed (Con [] Sum)));
  cprod : kit (F Ty (F Ty Ty)) transform
      (repeat (decodeClosed (Con [] Prod)));
}.
```

Rajapinnassa `transform` on määritelmä, jota tarvitaan geneerisen esitysmudon tyyppivakioiden muodostukseen, loput ovat määritelmiä itse tyyppivakioiden toiminnalle. Kokonaisuutena määritelmien muodostus rajapinnan avulla ei juurikaan eroa niiden määrittelystä ilman rajapintaa. Sen päätarkoituksena onkin vain tarjota selkeät vaatimukset, jotka kirjaston käyttäjän tulee toteuttaa ja siten sen käyttö varsinaisissa geneerisissä funktioissa on vapaaehtoista. Kirjastosta toteutettiin myös versio, joka käyttää rajapinnan määärittelyjä tyyppitason funktion `tyConstEnv` sijaan määritelmässä `ngen`. Se on saatavilla lähdekoodin versionhallinnan haarasta `record`.

5.8 Muut toteutukset

Ariteetti- ja tietotyyppigeneerisestä ohjelmoinnista ei juurikaan ole saatavilla toteutettuja kirjastoja. Kirjasto, jonka Weirich ja Casinghino (2010) toteuttivat on ainoa tutkielman lähdekatsauksen nojalla löydetty riippuvasti tyyypitetyllä ohjelmointikielellä tehty toteutus. Heidän toteutuksensa lisäksi on kuitenkin muun muassa Verbruggen, Vries ja Hughes (2008) sekä Altenkirch, McBride ja Morris (2006) esitelleet tietotyyppigeneeriset kirjastot, joiden

geneerinen esitysmuoto on kindeilla indeksoitu. Toteutuksiin liittyen Weirich ja Casinghino (2010) esittävätkin, että niiden ilmaisukyky olisi riittävä myös ariteettigeneerisyyden esittämiseen. Rakenteeltaan tutkielmassa toteutettu kirjasto vastaa hyvin pitkälti edellämainittuja kirjastoja eikä sen osalta tarjoa uusia tuloksia. Tutkielman kirjaston uusina tuloksina onkin ariteetti- ja tietotyyppigeneerisyyden lisääminen Coqille sekä universumipolymorfismin käyttö kirjaston määritelmien luonnissa.

6 Arviointi

Tässä luvussa käydään läpi tutkielmassa toteutetun ariteetti- ja tietotyyppigeneerisen kirjaston arviointia. Arviointia suoritettiin staattisesti lähdekoodia läpikäymällä sekä dynaamisesti esimerkkejä muodostamalla ja suorittamalla. Arvioinnissa onnistumista tarkasteltiin erityisesti asetettujen tavoitteiden näkökulmasta. Tutkielmassa asetettuina tavoitteina olivat toimivan ariteetti- ja tietotyyppigeneerisen kirjaston toteuttaminen sekä universumipolymorfismin käyttö määritelmissä esiintyvän toiston välttämiseksi ja kielen johdonmukaisuuden säilyttämiseksi. Johdonmukaisuuden säilyttäminen mahdollistaisi myös kirjaston ominaisuuksien formaalin todistamisen riippuvan tyypityksen avulla. Lisäksi toteutusta verrattiin ajoittain ariteetti- ja tietotyyppigeneeriseen Agda-kirjastoon, jonka Weirich ja Casinghino (2010) toteuttivat, sekä tietotyyppigeneeriseen Coq-kirjastoon, jonka Verbruggen, Vries ja Hughes (2008) toteuttivat. Vertailua suoritettiin, koska kirjastoja käytettiin toteutuksen pohjana. Lisäksi ajoittain oli hyödyllistä tarkastella Agdan käytöstä seuraavia eroja Coqiin verrattuna. Luvussa käydään läpi myös tutkielman toteutuksessa tiedossa olevia puutteita sekä niiden syitä ja mahdollisia ratkaisuja. Lopuksi esitellään myös kirjaston toteutuksessa ilmenneitä vaikeuksia.

6.1 Tavoitteiden täyttyminen

Asetettujen tavoitteiden osalta kirjasto täyttää vaatimukset, sillä tutkielman tuloksena saatiin toimiva ja Coqilla ohjelmoitu ariteetti- ja tietotyyppigeneerinen kirjasto. Lisäksi toteutuksessa käytettiin onnistuneesti hyödyksi universumipolymorfismia.

Toimivuuden osoittamiseksi muodostettiin tutkielmassa varsinaisen kirjaston toteutuksen lisäksi malliesimerkit geneeristen funktioiden määrittelystä sen avulla. Muodostetuista esimerkeistä `gmap` on tietotyyppigeneerinen ja `ngmap` ariteetti- ja tietotyyppigeneerinen. Lisäksi, funktion `ngmap` osittaisuudesta johtuen, muodostettiin tyyppiä `option` hyödyntävä versio `optNgmap`, jossa ei tarvinnut käyttää ylimääräistä aksioomaa. Esimerkit ovat kokonaisuudessaan saatavilla tutkielman toteutuksen lähdekoodin tiedostosta `examples.v`.

Tutkielman kirjaston määritelmiä ei tarvinnut toistaa eri tyyppitasoilla universumipolymor-

fismin käytön ansiosta. Lisäksi kieli pystyttiin säilyttämään johdonmukaisena, mikä mahdollistaisi kirjaston ominaisuuksien todistamisen esimerkiksi jatkokehityksessä. Määritelmien toiston puutteen voi varmistaa kirjaston julkaistua lähdekoodia läpikäymällä. Samoin julkaistusta lähdekoodista voi huomata, että kirjastoon ei tarvinnut käyttää kielen johdonmukaisuuden rikkovaa Coqin `-type-in-type` komentoriviargumenttia.

6.2 Tiedossa olevat puutteet

Kirjaston toteutuksessa ei huomioitu tietotyyppien isomorfismeja, joita Weirich ja Casinghino (2010) hyödynsivät omassa toteutuksessaan ja jota varten he lisäsivät geneeriseen esitysmuotoonsa uuden konstruktorin, joka vähensi toisteisen koodin määrää kirjaston käyttäjän näkökulmasta. Tietotyyppien isomorfismien puuttumisen vuoksi tutkielman kirjastossa geneeristen tyyppien määrittely on käyttäjän kannalta vaikeampaa ja virhealttiimpaa. Tyyppien määrittelyn vaikeus esiintyy sellaisten tyyppien kohdalla, jotka eivät suoraan vertaannu geneerisen esitysmuodon vakiotyyppeihin. Malliesimerkkinä monimutkaisempien geneeristen tyyppien luonnista tutkielman lähdekoodissa esiteltiin kuitenkin `tmaybe` sekä `teither`. Määritelmät löytyvät kokonaisuudessaan tutkielman lähdekoodin tiedostosta `examples.v`.

Kuten luvun 5.6 määritelmistä pystyy huomaamaan, voi ariteetti- ja tietotyyppigeneeristen funktioiden määrittely kirjaston avulla olla varsin työlästä. Ero on selkeä myös verrattuna esimerkiksi luvun 5.5 tietotyyppigeneerisen funktion määritelmiin. Näin ollen käyttäjän kannalta voikin ajoittain olla järkevämpää käyttää vain tietotyyppigeneerisyyttä, jos ariteettigeneerisyydelle ei selkeää tarvetta ole, jotta määritelmät voidaan pitää selkeämpinä ja helpommin rakennettavina. Suurimpana vaikeutena ariteetti- ja tietotyyppigeneeristen funktioiden muodostuksessa oli geneeriselle esitysmuodolle määriteltyjen tyyppivakioiden toiminnan määrittely. Tyyppivakioiden määrittelyssä suurin este oli Coqin käyttämän tyyppiteorian vaatimukset ja siten tyyppitarkastimen läpäiseminen. Määritelmien muodostuksessa jouduttiinkin laajalti turvautumaan Coqin todistustilaan, minkä seurauksena määritelmistä muodostui kohtuullisen pitkiä ja vaikealukuisia. Lisäksi, jos määritelmiä verrataan vastaaviin, jotka Weirich ja Casinghino esittelevät omassa toteutuksessaan, on niiden ero todella selkeä. Vaikka tyyppimääritelmät funktioille ovat usein samanlaiset, pystyvät he kirjastossaan määrittelemään vastaavat vakioiden toiminnat yleensä vain rivin tai kahden mittaisilla määritel-

millä. Määrittelyjen lyhyydessä on taustalla Agdan tyyppitarkastimen kevyemmät vaatimukset ja vahvempi päättely, varsinkin riippuvasti tyypitetyssä hahmonsovituksessa (Weirich ja Casinghino 2010). Ratkaisuna tähän olisi voinut olla käyttää riippuvasti tyypitetyissä tyyppimääritelmissä enemmän apuna avainsanoja `Fixpoint` ja `Definition`, jotta tyypit olisi voitu luoda funktioiden kautta. Funktioiden käyttö tyyppimääritelmissä olisi eronnut yleisesti kirjastossa käytetystä tyylistä, jossa tyypit luodaan induktiivisina tyyppeinä. Funktioiden vahvempi käyttö olisi voinut ajoittain helpottaa tyyppitarkastimen läpäisemistä ja esimerkiksi Verbruggen, Vries ja Hughes (2008) käyttivät omassa Coqilla toteutetussa tietotyyppigeneerisessä kirjastossaan pääasiallisesti funktioilla määriteltyjä tyyppejä. Toinen keino määritelmien muodostuksen helpottamiseen Coqissa olisi voinut olla `Equations`-lisäosa, joka tehostaa riippuvan hahmonsovituksen käyttöä (Sozeau ja Mangin 2019).

Kirjaston toteutuksessa jouduttiin käyttämään Coqin `-impredicative-set` komentoriviargumenttia, joka muokkaa Coqin tyyppiteoriaa tekemällä sortista `Set` impredikatiivisen (Coq Development Team 2020). Coqin vanhemmissa versioissa `Set` oli oletuksena impredikatiivinen, mutta tämä muutettiin, koska se johti johdonmukaisuuden rikkoutumiseen esimerkiksi valinta-aksioomaa ja kolmannen poissuljetun lakia käytettäessä (Chlipala 2013). Koska tutkielmassa Coqia käytetään nimenomaan ohjelmointikielen näkökulmasta, ei komentoriviargumentin käytön tulisi kuitenkaan johtaa kielen johdonmukaisuuden menettämiseen. Impredikatiivisuutta tarvittiin kirjaston toteutuksessa geneerisen esitysmuodon muodostuksessa. Sama puute oli myös tietotyyppigeneerisessä kirjastossa, jonka Verbruggen, Vries ja Hughes (2008) toteuttivat, eikä tämän tutkielman toteutuksessa ongelmaan onnistuttu löytämään parempaa ratkaisua.

Tutkielman kirjastossa ei pystytty päättelemään ariteetti- ja tietotyyppigeneeristen määritelmien saamien argumenttien määrää implisiittisesti, koska Coqin tyyppijärjestelmä ei kanna implisiittisyyttä tyyppien osana. Kyvyttömyys päätellä argumentti implisiittisesti vähentää geneeristen määritelmien käyttömukavuutta, mutta määritelmiä voidaan kuitenkin onnistuneesti käyttää esittämällä argumentti eksplisiittisesti.

6.3 Toteutuksessa esiintyneet vaikeudet

Vaikeuksia toteuksessa aiheutti joidenkin funktioiden määrittely todistustilan avulla. Ongelmana oli erityisesti Coqin todistustaktiikan `destruct` käyttö riippuvasti tyypitetyille määritelmille, missä osasyynä oli jo aiemmin mainittu Coqin riippuvasti tyypitetyn hahmonsovituksen käytön vaikeus esimerkiksi Agdaan verrattuna. Riippuvasti tyypitettyyn hahmonsovitukseen liittyviä ongelmia voi Coqissa paikoitellen korjata käyttämällä todistustaktiikkaa `dependent destruction`, mutta puutteena siinä on ylimääräisen aksiooman käyttö, minkä vuoksi sitä ei voi käyttää hyödyksi laskennan suorittamisessa. Näin ollen se olisi voinut olla hyödyllinen esimerkiksi matemaattisia todistuksia tehtäessä, mutta laskennan puuttumisen vuoksi sitä ei voinut hyödyntää tutkielman kirjaston toteutuksessa. Riippuvan tyypityksen hahmonsovitukseen liittyvät ongelmat korjattiin lopulta toteuttamalla erilliset taktiikkaa `destruct` muistuttavat rakenteellista analyysiä suorittavat apumääritelmät. Toisena mahdollisuutena olisi voinut olla Coqin komennon `Derive Dependent Inversion` käyttö. Apumääritelmistä ensimmäinen oli `rectS`, jota tarvittiin vektorityyppien todistuksia varten ja joka saatiin Coqin standardikirjastosta. Vaikeampi tapaus oli rakenteellisen analyysin määrittely geneerisen esitysmuodon tyypille `tyvar`, jolle valmiita määritelmiä ei ollut saatavilla. Siihen liittyvää apumääritelmää tarvittiin tyyppimuuttujia ympäristöstä etsivän funktion `nlookup` muodostamiseen. Lisäksi ongelmia aiheutti se, ettei vastaavanlaisten riippuvasti tyypitettyjen määritelmien todistukselle juurikaan ollut tarjolla lähdemateriaalia, joka olisi ongelmaa ja sen ratkaisutapoja kuvannut. Parhaan selityksen ongelmaan tarjosi blogikirjotuksessaan Wilcox (2014), jonka esittämiä periaatteita noudattaen määritelmä saatiin muodostettua nimellä `tvcase`. Määritelmä löytyy tutkielman lähdekoodin tiedostosta `univ.v`.

Ariteetti- ja tietotyyppigeneeristen kirjastojen vähäinen määrä aiheutti myös ongelmia. Toteutuksen teossa oli tarjolla vähän lähdemateriaalia, jonka pohjalta kirjastoa rakentaa, minkä seurauksena jouduttiin turvautumaan vahvasti Agda-kirjastoon, jonka Weirich ja Casinghino (2010) toteuttivat.

Yleisemmällä tasolla vaikeuksia aiheutti tutkielman kirjoittajan vähäinen kokemus Coqin käytöstä ohjelmointikielenä, mitä lisäsi kielen ongelmakohtien vähäinen dokumentaatio sekä kielen kohtuullisen korkea oppimiskäyrä. Vaikeuksien seurauksena osa kirjaston määri-

telmistä voi olla laadukkaammin toteutettavissa kokeneemman Coq-ohjelmoijan toimesta.

7 Yhteenveto

Tutkielman tavoitteena oli toimivan ariteetti- ja tietotyyppigeneerisen kirjaston luominen sekä universumipolymorfismin hyödyntäminen kirjaston määritelmien muodostuksessa. Tavoitteet saavutettiin onnistuneesti luomalla prototyyppimäinen kirjasto.

Kirjaston muodostuksessa käytettiin pohjana Agdalla toteutettua ariteetti- ja tietotyyppigeneeristä kirjastoa (Weirich ja Casinghino 2010), minkä seurauksena tutkielmassa ilmeni selkeitä eroja Coqin ja Agdan välillä. Coqin tyyppitarkastimen tarkemmat vaatimukset aiheuttivat ajoittain vaikeuksia määritelmien viimeistelyssä ja tekivät valmiista määritelmistä usein pitkiä ja monimutkaisia. Määritelmien muodostuksen vaikeudet itse kirjaston luonnissa eivät olisi erityisen haitallisia, mutta vaikeudet esiintyvät myös geneeristen funktioiden tyyppivakioiden muodostuksessa, mikä vaikeuttaa kirjaston käyttöä. Vaikeudet kuitenkin esiintyvät lähinnä ariteetti- ja tietotyyppigeneerisissä määritelmissä, joten tietotyyppigeneeristen määritelmien muodostus kirjaston avulla ei tulisi olla yhtä monimutkaista.

Vaikeuksia tutkielman toteutuksessa aiheutti ariteettigeneerisyyttä käsittelevien lähteiden vähäinen määrä. Lisäksi tutkielman kirjoittajan vähäinen kokemus Coqista ohjelmointikielenä sekä joidenkin esiintyneiden ongelmien harvinaisuus aiheutti ajoittain hankaluuksia. Osa kirjaston määritelmistä voikin olla turhan monimutkaisia ja tiivistettävissä kokeneemman Coq-ohjelmoijan toimesta.

Kirjastolle jäi myös jatkokehitettäviä ominaisuuksia, joita ei tutkielman puitteissa toteutettu. Tietotyyppien isomorfismit edesauttaisivat kirjaston käyttäjän kannalta geneerisen esitysmuodon tyyppien muodostusta, mikä helpottaisi kirjaston käyttöä varsinkin monimutkaisten tyyppien kohdalla. Kirjaston käyttöä vaikeuttaa myös implisiittisten argumenttien päättelyn puutteet. Lisäksi kirjaston ominaisuuksien muodollinen todistus lisäisi luottamusta kirjaston toimintaan.

Lähteet

Allais, Guillaume. 2019. “Generic Level Polymorphic N-Ary Functions”. Teoksessa *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development,* 14–26. https://doi.org/10.1145/3331554.3342604.

Altenkirch, Thorsten, Conor McBride ja Peter Morris. 2006. “Generic Programming with Dependent Types”. Teoksessa *Proceedings of the 2006 International Conference on Datatype-Generic Programming,* 209–257.

Bertot, Yves, ja Pierre Casteran. 2004. *Interactive Theorem Proving and Program Development.* SpringerVerlag.

Brady, Edwin. 2013. “Idris, a general-purpose dependently typed programming language: Design and implementation”. *Journal of Functional Programming* 23 (05): 552–593. https://journals.cambridge.org/article_S095679681300018X.

Chlipala, Adam. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

Coq Development Team, The. 2020. *The Coq Reference Manual.* Zenodo. https://doi.org/10.5281/zenodo.4021912.

———. 2021. *The Coq Standard Library.* http://coq.inria.fr/library/.

Coquand, Thierry. 1986. “An Analysis of Girard’s Paradox”. Teoksessa *1st Symposium on Logic in Computer Science.*

Eisenberg, Richard A. 2017. *Dependent Types in Haskell: Theory and Practice.* arXiv: 1610.07978 [cs.PL].

Eisenberg, Richard A., ja Stephanie Weirich. 2012. “Dependently Typed Programming with Singletons”. *SIGPLAN Not.* (New York, NY, USA) 47 (12): 117–130. https://doi.org/10.1145/2430532.2364522.

Fridlender, Daniel, ja Mia Indrika. 2000. “Do we need dependent types?” *Journal of Functional Programming* 10 (4): 409–415.

Gibbons, Jeremy. 2007. “Datatype-Generic Programming”. Springer Berlin Heidelberg.

Gonthier, Georges. 2005. “A computer-checked proof of the Four Colour Theorem”.

Harper, Robert. 2016. *Practical Foundations for Programming Languages.* 2. painos. Cambridge University Press.

Harper, Robert, ja Robert Pollack. 1991. “Type checking with universes”. *Theoretical Computer Science* 89 (1): 107–136. http://www.sciencedirect.com/science/article/pii/030439759090108T.

Hevner, Alan R., Salvatore T. March, Jinsoo Park ja Sudha Ram. 2004. “Design Science in Information Systems Research”. *MIS Quarterly* 28 (1): 75–105. http://www.jstor.org/stable/25148625.

Hinze, Ralf. 2002. “Polytypic values possess polykinded types”. *Science of Computer Programming* 43 (2): 129–159. http://www.sciencedirect.com/science/article/pii/S0167642302000254.

Kennedy, Andrew, ja Don Syme. 2000. “Design and Implementation of Generics for the .NET Common Language Runtime”. *SIGPLAN Not.* (New York, NY, USA) 35 (4): 0—12. https://doi.org/10.1145/381694.378797.

Leroy, Xavier. 2009. “Formal verification of a realistic compiler”. *Communications of the ACM* 52 (7): 107–115. https://hal.inria.fr/inria-00415861.

Lämmel, Ralf, ja Simon Peyton Jones. 2003. “Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming”. *SIGPLAN Not.* (New York, NY, USA) 38 (3): 26–37. https://doi.org/10.1145/640136.604179.

Magalhães, José Pedro, ja Andres Löh. 2012. “A Formal Comparison of Approaches to Datatype-Generic Programming”. Teoksessa *Proceedings Fourth Workshop on Mathematically Structured Functional Programming,* 76:50–67. Tallinn, Estonia.

March, Salvatore T., ja Gerald F. Smith. 1995. “Design and natural science research on information technology”. *Decision Support Systems* 15 (4): 251–266. http://www.sciencedirect.com/science/article/pii/0167923694000412.

Marlow, S. 2010. “Haskell 2010 Language Report - Standard Prelude”. https://www.haskell.org/definition/haskell2010.pdf.

Norell, Ulf. 2008. “Dependently Typed Programming in Agda”. Teoksessa *Proceedings of the 6th International Conference on Advanced Functional Programming,* 230–266. Berlin, Heidelberg: Springer-Verlag.

Pehkonen, Jere. 2021. *Arity-generic datatype-generic programming in Coq.* http://www.github.com/jeanpehk/doublygen.

Serrano, Alejandro, ja Victor Cacciari Miraldo. 2018. “Generic Programming of All Kinds”. *SIGPLAN Not.* (New York, NY, USA) 53 (7): 41–54.

Sozeau, Matthieu, ja Cyprien Mangin. 2019. “Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq”. *Proc. ACM Program. Lang.* (New York, NY, USA) 3. https://doi.org/10.1145/3341690.

Sozeau, Matthieu, ja Nicolas Tabareau. 2014. “Universe Polymorphism in Coq”. Teoksessa *Interactive Theorem Proving.* Vienna, Austria. https://hal.inria.fr/hal-00974721.

Wadler, Philip. 1989. “Theorems for Free!” Teoksessa *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture,* 347–359. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/99370.99404.

Weirich, Stephanie, ja Chris Casinghino. 2010. “Arity-Generic Datatype-Generic Programming”. Teoksessa *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification,* 15–26. New York, NY, USA: Association for Computing Machinery.

Verbruggen, Wendy, Edsko de Vries ja Arthur Hughes. 2008. “Polytypic programming in COQ”. Teoksessa *Proceedings of the ACM SIGPLAN Workshop on Generic Programming,* 49–60. New York, NY, USA: Association for Computing Machinery.

Wilcox, James. 2014. *Dependent Case Analysis in Coq without Axioms.* https://jamesrwilcox.com/dep-destruct.html.

Voevodsky, Vladimir, Benedikt Ahrens, Daniel Grayson ym. *UniMath — a computer-checked library of univalent mathematics.* https://github.com/UniMath/UniMath.

Vries, Edsko de, ja Andres Löh. 2014. “True Sums of Products”. Teoksessa *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming,* 83–94. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2633628.2633634.