VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS

Programavimo kalbos su priklausomais tipais transliavimas į Go programavimo kalbą

Dependently typed programming language translation to Go programming language

Bakalauro darbas

Atliko: 4 kurso studentas
Justas Tvarijonas (parašas)

Darbo vadovas: Partn. Doc. Viačeslav Pozdniakov (parašas)

Recenzentas: Dr. Karolis Petrauskas (parašas)

Vilnius – 2021

Santrauka

Darbe analizuojamas bei įgyvendinamas funkcinės programavimo kalbos su priklausomais tipais transliavimas į Go programavimo kalbą. Teorinėje darbo dalyje aprašomos Agda bei Go programavimo kalbos, egzistuojantys transliatoriai ir jų panašumai.

Praktinėje dalyje apibrėžiama Go funkcijų vidinė sintaksė skirta generuoti Go kodą bei transliavimo taisyklės, kurios apibrėžia transliavimą iš Agda funkcijų bemedės duomenų tipą į šiame darbe apibrėžtą Go vidinės sintaksės duomenų tipą. Pagal šias aprašytas taisykles buvo įgyvendintas transliatorius, kuris parodo šių taisyklių veiksmingumą. Įgyvendinus transliatorių darbe aprašomi našumo testai, kurie lygina sugeneruoto Go kodo našumą su standartinio transliatoriaus sugeneruotu Haskell kodu. Buvo pastebėta, kad dabartinė implementacija našumu stipriai nusileidžia standartinio transliatoriaus sugeneruotam Haskell kodui. Verifikavimui atlikti transliavimo kodas iš Haskell buvo perkeltas į Agda bei su konkrečiomis funkcijomis buvo parodyta, kad yra įmanoma Agda bemedės sintaksės kodą paversti į Go vidinę sintaksę neprarandant informacijos.

Raktiniai žodžiai: funkcinė programavimo kalba su priklausomais tipais, Agda, Go, kodo transliavimas

Summary

This paper includes analysis and implementation of translator from dependently typed programming language to Go programming language. The theoretical part of this paper describes Agda and Go programming languages, existing translators and their similarities.

In the practical part we define internal syntax of Go functions and translation rules which define transliations from Agda functions treelesss syntax to our defined Go internal syntax data type. A translator has been implemented according to these described rules, which demonstrates the effectivness of these rules. After the implementation of the translator, paper describes benchmark tests that compare generated Go code with the Haskell code generated by the standart translator. It was identified that current translator implementation is fairly slower that the Haskell code generated by the standart Haskell translator. In order to verify the translator, transformation code was transferred from Haskell to Agda, and with specific functions, it was shown that it is possible to convert Agda's treeless syntax code to Go's internal syntax without loosing information.

Keywords: a dependently typed programming language, Agda, Go, code translation

TURINYS

Įvadas

Programavimas yra instrukcijų davimas kompiuteriui, kuris jas vykdo nepaisant to ar jos yra prasmingos ar ne. Kadangi žmonija tampa vis labiau priklausoma nuo kompiuterių, kurie atsiranda kiekviename mūsų gyvenimo aspekte, tampa vis svarbiau kuo labiau sumažinti klaidų kiekį programiniame kode. Tai bandoma įgyvendinti įvairiais metodais, tokiais, kaip detalus projektavimas ar testų rašymas. Pastarasis yra vienas dažniausiai naudojamų metodų, tačiau testavimas parodo ne klaidų nebuvimą, o tik tai, kad jos yra [**UHC**].

Tipai programavimo kalbose leidžia programuotojui nurodyti numatytą programos elgseną tipų pavidalu. Tipai ne tik padeda programuotojui rašyti teisingą programą, bet taip pat, kompiuteris gali patikrinti ar sukurta programa veikia taip, kaip buvo užrašyta. Paprasčiausias pavyzdys būtų programa, kuri gavusi sąrašą skaičių, prie kiekvieno elemento prideda vienetą, šios programos tikslas yra priimti sąrašą bei sąrašą grąžinti.Tai labai abstraktus apibrėžimas, ką ši programa atlieka, tačiau tai suteikia tam tikrą informaciją apie šios programos veikimą.

Priklausomų tipų sistemos [**schematicApproach**] leidžia tipams būti priklausomais nuo konkrečių reikšmių. Programavimo kalbos [**agda_book**; **idris**], kuriose yra naudojami priklausomi tipai leidžia sukurti tikslesnius tipus, kurie padeda daugiau klaidų aptikti kompiliavimo metu, vietoje to, kad tos pačios klaidos išliktų nepastebėtos iki programos veikimo pradžios. Su tipais, kurie turi daugiau informacijos mes daugiau žinome apie galimus programos parametrus bei rezultatus. Taip pat, priklausomi tipai programavimo kalbose suteikia galimybę rašyti įrodymus.

Viena iš programavimo kalbų su priklausomais tipais, kuri yra ir šio darbo objektas, yra Agda. Tai funkcinė programavimo kalba, kurios sintaksė yra panaši į plačiau žinomos funkcinės programavimo kalbos Haskell [**haskell**] sintaksę. Dabartinis Agda įgyvendinimas tipų patikrinimais siekia užtikrinti, kad programos bei įrodymai būtų teisingi. Tuo tarpu Go programavimo kalba [**Go**] yra greitai kompiliuojama, bet ne taip griežtai tipizuota, taigi atsiranda poreikis tam tikras vietas suprogramuoti su Agda programavimo kalba, verifikuoti jas Agda viduje, bei tada sugeneruoti Go kodo bibliotekas, kurias jau galėtų panaudoti egzistuojantis Go kodas.

Šio darbo tikslas - suprojektuoti Agda kalbos transliavimo į Go programavimo kalbą taisykles bei realizuoti transliatorių parodantį šių taisyklių veikimą.

Uždaviniai

1. Sudaryti Agda funkcijų transliavimo taisykles
2. Pagal aprašytas taisykles įgyvendinti Agda kalbos transliatorių
3. Verifikuoti įgyvendintą Agda kalbos transliatorių
4. Identifikuoti įgyvendinto transliatoriaus trūkumus

1. Priklausomi tipai

Priklausomi tipai yra tokie tipai, kurie priklauso nuo tipų reikšmių [**depentantTypesAtWork**]. Pavyzdžiui galime apibrėžti tipą A_n - sveikasis skaičius, mažesnis už n. Sakome, kad A_n priklauso nuo skaičiaus n arba A_n yra tipų, kurie yra indeksuoti pagal skaičių n, šeima. Parametrizuoti tipai, tokie kaip sąrašas susidedantis iš elementų A paprastai nėra vadinami priklausomais tipais, kadangi šios tipų šeimos yra indeksuojamos kitų tipų, o ne tipų reikšmių, kaip yra su priklausomais tipais [**depentantTypesAtWork**].

1.1. Programavimo kalbos su priklausomais tipais

Šiuo metu yra sukurta nemažai programavimų kalbų, kurios palaiko priklausomus tipus, dalis jų, pavyzdžiui Coq [**coq**], buvo sukurtos kaip vadinamasis įrodymų asistentas, kurių pagalba yra rašomi formalūs įrodymai. Kai kurios kalbos, kurios taip pat yra įvardijamos, kaip įrodymų asistentai, pavyzdžiui Agda [**agda_book**], gali būti naudojamos ne tik kaip įrodymų asistentai, bet taip pat ir programinės įrangos kūrimui. Egzistuoja ir atvirkščias variantas, kur kalbą yra įvardijama kaip bendrinės paskirties programavimo kalba, tačiau taip pat gali būti naudojama kaip įrodymų asistentas, pavyzdžui Idris [**idris**]. Šiame poskyryje bus trumpai apžvelgtos kelios iš aukščiau paminėtų programavimo kalbų.

1.1.1. Idris

Idris yra grynai funkcinė bendrosios paskirties programavimo kalba su priklausomais tipais, kuri taip pat gali būtų naudojama kaip įrodymų asistentas. Idris yra pakankamai stipriai įtakotas Haskell [**haskell**], taigi turi nemažai panašumų su Haskell, ypač sintaksėje bei tipuose, nors Idris turi labiau pažengusią tipų sistema [**idris**]. Idris gali būtų kompiliuojama į C arba JavaScript kodą.

1.1.2. Agda

Agda, yra grynai funkcinė kalba, kuri yra įrodymų asistentas, tačiau gali būti naudojama ir programinės įrangos kūrimui. Pirmoji šios kalbos versija buvo išleista 1999 metais. Dabartinė versija (Agda 2) buvo suprojektuota ir įgyvendinta Gothenburgo universiteto tyrimų inžinieriaus Ulf Norell, tai yra visiškas originalios Agda sistemos pertvarkymas. Kaip ir jos pirmtakai, Agda palaiko indukcinius duomenų tipus, šablonų derinimą (*angl.* pattern matching) bei nutraukimo tikrinimą (*angl.* termination checking) [**agda_overview**]. Agda nėra universali skaičiavimo požiūriu (ne „Turing complete“). Šiuo metu oficialiai palaikomos 2 kompiliatorių posistemės: MAlonzo skirta transliavimui į Haskell, bei kita posistemė skirta transliavimui į JavaScript

Kadangi šiame darbe aprašomas Agda kalbos transliavimas, apie šią kalbą bus labiau išsiplėsta kitame skyriuje.

2. Agda programavimo kalba

Kaip ir minėta buvusiame skyriuje, Agda yra grynai funkcinė programavimo kalba, kurios pagrindinė paskirtis yra įrodymų asistentas. Agda turi įprastus programavimo konstruktus kaip duomenų tipai, šablonų derinimas, „let“ išraiška, moduliai bei įrašai. Ši programavimo kalba pasižymi sintakse panašia į Haskell programavimo kalbą.

2.1. Loginė sistema

Agda pagrindui buvo pasirinkta Zhaohui Luo vieninga priklausomų tipų teorija (*angl.* unified theory of dependent types, UTT) [**agdaInitial**], kuri apjungia Martin-Löf loginę sistemą su tipų visatomis bei Coquand-Huet konstrukcijų skaičiavimą [**schematicApproach**]. Galime paminėti keletą sintaksės elementų egzistuojančių šioje loginėje sistemoje:

- Priklausomas funkcijos tipas

  ```
  (x : A) -> B
  ```

  Kur funkcija priima argumentą x, kurio tipas yra A, bei grąžina rezultatą, kurio tipas yra B, B tipas gali priklausyti nuo argumento x.

- Visatos - tai tokie tipai (žymimi „Set“), kurių elementai yra kiti tipai.

Agda loginė sistema yra intuicionistinė, arba dar kitaip vadinama konstruktyvioji. Ši logika remiasi konstruktyviuoju įrodymu, joje teisingi yra tik tie dalykai, kuriuos įmanoma sukonstruoti. Dėl šios priežasties šioje logikoje nėra klasikinėje logikoje naudojamo neįtrauktų vidurių dėsnio (*angl.* law of excluded middle) bei dvigubo neiginio teoremos [**intuitionistic**].

2.2. Sintaksė

Galime išskirti šiuos Agda programavimo kalbos sintaksinius bruožus:

- Agda programavimo kalboje, priešingai negu didelėje dalyje kitų kalbų (kaip Haskell ar Go), konstruktoriai bei identifikatoriai neturi praktiškai jokių leksinių apribojimų, ko pasekoje, identifikatorius galima pavadinti bet kokiais norimais simboliais, pavyzdžiui, toks duomenų tipas yra visiškai legalus:

  ```
  data _? (α : Set) : Set where
    # : α ?
    _C_ : α -> α ? -> α ?
  ```

- Tarpai turi didelę įtaką programos kodui, jie naudojami kaip atskiriamieji simboliai
- Yra galimybė apibrėžti operatorių pirmumą, kokia tvarka jie turi būti vykdomi

2.3. Numanomi argumentai

Numanomų argumentų mechanizmas leidžia išmesti tas programos dalis, kurias gali išvesti tipo tikrintuvas. Tarkime turime dvi funkcijas:

```
f : (x : A) -> B x
g : {x : A} -> B x
```

Skirtumas tarp šių dviejų funkcijų yra skliaustų tipas, g funkcija (su riestiniais skliaustais) priima numanomas argumentus, t.y. pati Agda bandys išsiaiškinti, kokia reikšmė turi būti toje vietoje.

2.4. Duomenų tipai

Pagrindinis būdas Agda programavimo kalboje apibrėžti duomenų tipus yra per indukcinius bei indukcinius-rekursinius duomenų tipus, kurie yra panašūs į algebrinius duomenų tipus kitose kalbose (pavyzdžiui Haskell). Natūraliuosius skaičius būtų galima apibrėžti taip:

```
data Nat : Set where
  zero : Nat
  suc : Nat -> Nat
```

Iš esmės šis apibrėžimas reiškia, kad turime duomenų tipą Nat, kuris turi du konstruktorius: zero ir suc, zero atspindi natūralųjį skaičių 0, o suc n yra prie n pridėtas 1. Pagal indukcijos principą, jeigu n yra natūralusis skaičius, tai ir n + 1 bus natūralusis skaičius. Tarkime natūralųjį skaičių 3 galėtume atvaizduoti tokia eilute: suc (suc (suc zero)).

Agda palaiko indukcines šeimas (pagal kurias yra sukurti apibendrinti algebriniai duomenų tipai Haskell programavimo kalboje), šiose šeimose kiekvienas duomenų tipo konstruktorius gali grąžinti skirtingą tipą. Kadangi Agda yra su priklausomais tipais, galima abiprėžti vektoriaus tipą, kuris priklauso nuo skaičiaus n:

```
data Vec (A : Set) : Nat -> Set where
  [] : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

Šiame apibrėžime „(A : Set)“ yra Vec duomenų tipo elemento parametras. Naujai apibrėžto Vec A tipas yra:

```
Nat -> Set
```

Tai reiškia, kad Vec A yra Set šeima, kuri yra indeksuojama pagal natūralųjį skaičių Nat.

Pasinaudoję jau aprašytais duomenų tipais, galime sukurti funkciją, kuri iš Vec grąžina pirmą elementą:

```
head : {A : Set}{n : Nat} -> Vec A (suc n) -> A
head (x :: xs) = x
```

Šį funkcija priima Vec, kuris yra ne mažesnio ilgio nei 1 (funkcijos aprašyme „suc n“ užtikrina,

kad vektoriaus ilgis yra bent 1), bei grąžina paskutinį jo elementą. Kadangi vektorius tikrai yra ne tuščias, nėra privaloma tikrinti to varianto, kai jis neturi elementų. Ši funkcija priima sąrašą, kurį sudaro bet kokio duomenų tipo elementai, atitinkamai galime susikurti įrašo tipą Zmogus, kuris laiko tam tikrą informaciją apie žmogų, bei perrašyti šią funkciją taip, kad ji veiktų tik su zmogaus tipo sąrašu:

```
record Zmogus : Set where
field
   vardas : String
   amzius : Nat

pirmasZmogus : {n : Nat} -> Vec Zmogus (suc n) -> Zmogus
pirmasZmogus (x :: xs) = x
```

Ši funkcija veiks taip pat, kaip ir aukščiau aprašyta funkcija, tačiau bandydami perduoti sąrašą su kitokio tipo elementais, gausime klaidą.

2.5. Verifikavimas

Agda kalboje (kaip ir daugelyje kitų programavimo kalbų su priklausomais tipais) yra galimybė indukcijos bei šablonų derinimo pagalba rašyti įrodymus, tipų pavidalu galima užrašyti skirtingas teoremas. Pavyzdžiui, galime susikurti duomenų tipą, kuris nurodo lygybę, bei funkciją cong, kuri parodo, kad jeigu a ir b yra lygūs, tai juos perdavus į funkciją f, rezultatas taip pat bus lygus:

```
data _==_ {A : Set}(x : A) : A -> Set where
  refl : x == x
cong : { A B : Set} -> {a b : A} -> (f : A -> B) -> a == b -> f a == f b
cong f refl = refl
```

Šioje funkcijoje agda tikrina a ir b kintamuosius, tačiau kadangi jie yra pateikti, kaip numanomi argumentai, jų tikrinime nematome.

Pasinaudodami šiuo duomenų tipu bei funkcija, galime įrodyti natūraliųjų skaičių asociatyvumą:

```
associativity : (m n p : Nat) -> ((m + n) + p) == (m + (n + p))
associativity zero n p = refl
associativity (suc m) n p = cong suc (associativity m n p)
```

Tai vienas iš būdų, kaip Agda kalboje galima įrodyti natūraliųjų skaičių asociatyvumą, tačiau jų egzistuoja ir daugiau.

Šiuose poskyriuose paminėti pagrindiniai, tačiau ne visi Agda programavimo kalbos konstruktai, likę konstruktai turėtų būtų įtraukti į ateityje atliekamus darbus.

3. Go programavimo kalba

Go programavimo kalba yra greitai kompiliuojama statiškai tipizuota programavimo kalba, kurios sintaksė yra panaši į supaprastintą C kalbos sintaksę [**Go**]. Vertas paminėta punktas, kad Go programavimo kalba šiuo metu nepalaiko bendrinių (*angl.* generic) tipų, tačiau jau yra patvirtintas pasiūlymas, kuriuo siūlomas bendrinių tipų įgyvendinimas[1]. Planuojama šiuos pakeitimus išleisti su 1.18 versija 2022 metų pradžioje.

3.1. Sintaksė

Galime išskirti šiuos Go programavimo kalbos sintaksinius bruožus:

- Operacijas galima atskirti naujos eilutės simboliu arba kabliataškiu.
- Identifikatorių vardai turi prasidėti raide arba apatiniu brūkšniu (_), po kurių gali sekti skaičiai, „unicode“ raidės arba apatiniai brūkšniai.
- Tarpai neturi įtakos programos veikimui, todėl gražus lygiavimas nėra privalomas, kad programa kompiliuotųsi.

3.2. Valdymo srauto konstruktai

Galime išskirti šiuos valdymo srauto konstruktus Go programavimo kalboje:

- „if-else“ - Šis konstruktas tikrina sąlyginį teiginį, jeigu jo reikšmė yra tiesa, tada vykdomas kodas, kuris yra riestinių skliaustų viduje, jeigu ne, tada vykdomas kodas esantis už jų. Verta paminėti, kad Go kalboje nėra populiaraus šio konstrukto sutrumpinimo su simboliais „?“ ir „:“.
- „switch-case“ - Šis konstruktas yra pakankamai panašus į prieš tai minėtą, tačiau suteikia galimybę patogiau patikrinti daugiau variantų, įprastas „switch-case“ atrodo taip:

```
switch reiksme {
   case reiksme1:
   ...
   case reiksme2:
   ...
   default:
   ...
   }
```

Su šiuo konstruktu taip pat galima lyginti ne tik paprastas reikšmes bet ir tipus. Šie du konstruktai atitinka Agda kalbos šablonų tikrinimą.

[1] https://go.googlesource.com/proposal/+/refs/heads/master/design/43651-type-parameters.md

- „for“ ir „while“ konstruktai - Go kalboje yra tokie konstruktai kaip „for“ ar „while“, kurie leidžia norimą veiksmą atlikti numatytą kiekį kartų.

3.3. Struktūros

Go kalboje struktūra (*angl.* struct) yra kintamųjų kolekcija, kuri paprastai apibrėžiama kaip naujas tipas. Jos pagalba galima sugrupuoti duomenis į vieną įrašą, pavyzdžiui galime apsirašyti struktūra, kuri savyje turi žmogaus duomenis:

```
type zmogus struct {
   vardas string
   amzius int
}
```

Norėdami sukurti naują sukurtos struktūros tipo kintamąjį tai atliekame užrašydami struktūros vardą ir riestiniuose skliaustuose nurodydami kintamųjų reikšmes, pavyzdžiui:

```
var zmogus1 zmogus = zmogus{"Justas", 23}
```

Taip pat yra galimybė sukurti tuščią struktūrą, tokios struktūros tipo kintamojo sukūrimas atrodytų taip:

```
tuscia{}
```

3.4. Sąsajos

Sąsajos (*angl.* interfaces) yra metodų aprašymų rinkinys, jis apibrėžia, kokius veiksmus gali atlikti tipas:

```
type zmogausPaieska interface {
   paskutinisZmogus(zmones []zmogus) zmogus
}
```

Kurdami naują metodą galime nurodyti apsirašytą sąsaja kaip to metodo priimamo argumento tipą, tuo atveju metodo argumentas privalo įgyvendinti tuos metodus kurie yra aprašyti mūsų sukurtoje sąsajoje. Panašiai kaip ir su struktūromis, galima sukurti sąsaja, kuri neturi jokio metodo, tokią sąsają įgyvendina visi tipai, Išnaudodami tokią sąsają galima gauti panašų veikimą į bendrinius sudėtinius tipus, bet apie tie bus išsiplėta vėlesniame skyriuje.

3.5. Funkcijos

Kurdami Go funkcijas privalome nurodyti kokio tipo argumentus ši funkcija priima, bei kokio tipo reikšmę grąžina (jeigu grąžina). Pasinaudodami aukščiau aprašyta struktūra bei Go masyvo tipu galime aprašyti alternatyvias funkcijas aprašytas Agda skyriuje:

```
func head(elementai interface{}) interface{} {
```

```
    switch nt := elementai.(type) {
        case []zmogus:
            return nt[0]
        default:
            panic("Nezinomas tipas")
    }
}
func pirmasZmogus(zmones []zmogus) zmogus {
    return zmones[0]
}
```

Funkcija, kuri priima ir grąžina zmogus tipo struktūrą yra paprasta, tačiau kadangi Go kalboje nėra bendrinių tipų, negalime aprašyti šios funkcijos nenurodydami konkretaus masyvo elementų tipo, nebent jos viduje išrašysime visus tipus, kuriuos ji priima. Taip pat, kadangi funkcija head grąžina abstraktų interface tipo rezultatą, jį gavus ir norint iš jo gauti vardą, reiktų tą rezultatą dar paversti atgal į tipą zmogus.

Galime palyginti šias Agda ir Go funkcijas, kurios atlieka tą patį uždavinį:

- Agda kalboje parašytos funkcijos užtikrina, kad funkcija nepriims tokios sąrašo, kurio apdoroti negalės, tuo tarpu Go kalboje aprašytai funkcijai galime perduoti tuščią sąrašą, ko pasekoje gautume vykdymo laiko (*angl.* runtime) klaidą.
- Agda kalboje naudodami bendrinį tipą priimančia funkciją, jos rezultatas iš karto gaunamas su tuo tipu, kuris buvo paduotas, tuo tarpu Go kalboje jis yra sąsajos tipo, iš kurio jį reikia paversti atgal į norimą tipą.

3.6. Bendrinių tipų pasiūlymas

Kaip jau buvo minėta šio skyriaus pradžioje, yra jau patvirtintas bendrinių tipų pasiūlymas, kurį planuojama išleisti 2022 metų pradžioje, tačiau šiuo metu egzistuoja Go įrankis, kuriuo pagalba galima transliuoti Go kodą su bendriniais tipais į šiuo metu palaikomą Go kodą.

Pagal šiame pasiūlyme aprašytas taisykles, bendriniai tipai bus pritaikomi metodams, struktūroms bei sąsajoms. Tarkime galima būtų sukurti metodą, kuris priima sąrašą bei grąžina jo pirmą elementą:

```
func head[T any](sarasas []T) T {
    return sarasas[0]
}
```

Šiame funkcijos aprašyme „[T any]“ aprašo bendrinį tipą, kuris neturi jokių apribojimų. Tarkime, norint priimti sąrašą tik iš sveikųjų skaičių, galima būtų apsirašyti tipų sąrašą, bet jį nurodyti kaip apribojimą bendriniui tipui:

```
type sveikasisSkaicius interface {
    type int, int8, int16, int32, int64
```

```
}
func head[T sveikasisSkaicius](sarasas []T) T {
  return sarasas[len(sarasas)-1]
}
```

Šiuo atveju metodas „head“ priima sąrašus, kurių elementai yra tik vieno iš aukščiau nurodytų tipų. Atitinkamai galima apibrėžti ir struktūras bei sąsajas, tačiau svarbu užtikrinti, kad kompiliatorius būtų pajėgus surasti tipus, kuriuos reikia įterpti vietoje bendrinio tipo, pavyzdžiui, tokia funkcija nėra teisinga, kadangi kompiliatorius neturi galimybės išsiaiškinti grąžinimo tipo:

```
func id[T any](elementas interface{}) T {
  return elementas.(T)
}
```

3.7. Paketų importavimas

Leidžiant Go programinį kodą yra keletą galimybių kaip pasinaudoti kitais paketais. Pirmasis būdas yra juos padedant į Go bazinį aplanką, kuriame yra kitos pagrindinės bibliotekos. Rašydami Go kodą ir norėdami pasinaudoti konkrečia biblioteka turime nurodyti kelią iki tos bibliotekos paketo nuo bazinio Go aplanko. Antrasis būdas, norint importuoti lokalius paketus reikia susikurti modulio failą, tada leisdami programą iš to paties aplanko, kuriame yra aprašytas modulis, galime importuoti kitus paketus nurodydami modulio vardą, po jo nurodant tikslų kelią iki importuojamo paketo.

4. Susiję darbai

Šiuo metu Agda repozitorijoje egzistuoja 2 transliatoriai:

- MAlonzo, kuris Agda programinį kodą paverčia į Haskell, iš kurio jis yra sukompiliuojamas naudojant Glasgow Haskell kompiliatorių (*angl.* Glasgow Haskell Compiler, GHC)
- JavaScript, kuris Agda programinį kodą paverčia į JavaScript kodą, šio transliatoriaus pagrindinė paskirtis yra internetinių programų kūrimas [**html_agda**]

Taip pat egzistuoja ir daugiau transliatorių, kurie nėra pridėti prie oficialios Agda repozitorijos, tai tokie transliatoriai, kaip Utrecht bei Epic [**Utrecht**; **Epic**]. Šie ir aukščiau paminėti transliatoriai bus giliau apžvelgti šiame skyriuje.

4.1. Haskell programavimo kalbos transliatoriai

4.1.1. MAlonzo transliatorius

Numatytasis ir vienas iš 2 agda transliatorių. Kaip jau ir minėta, jis nukreiptas į Haskell programavimo kalbą, iš kurios, naudojant GHC, gaunami vykdomieji failai (*angl.* executable files). Tai labiausiai subrendęs transliatorius, jo sugeneruotos programos veikia greičiau, negu vėliau sukurtų transliatorių [**Utrecht**; **Epic**]. Vienas minusas, kurį galima išskirti apie šį transliatorių yra tai, kad norėdama, kad programa praeitų GHC kompiliatorių, transformuodama kodą MAlonzo posistemė turi panaudoti nesaugius priverstinius tipų vertimus (*angl.* unsafeCoersion, tai metodas Haskell kalboje) prieš kiekvieną funkcijos rezultatą bei argumentą, o tai išplečia programos kodą. Taip pat, šie tipų vertimai gali neigiamai įtakoti GHC atliekant į tipus orientuotą optimizaciją.

4.1.2. Utrecht transliatorius

Skirtingai negu prieš tai minėtas transliatorius, jis konvertuoja į Utrecht Haskell kompiliatoriaus (*angl.* Utrecht Haskell Compiler, UHC) kodą. UHC Core, skirtingai negu GHC Core yra negriežtas, ko pasekoje šiame kompiliatoriuje nėra būtinybės naudoti nesaugius prievartinius tipų vertimus [**Utrecht**].

Kaip eksperimentinis transliatorius buvo pridėtas prie Agda repozitorijos, tačiau su vėlesnėmis versijomis pašalintas dėl labai didelių priklausomybių nuo kitų bibliotekų.

4.2. Javascript programavimo kalbos transliatorius

Vienas iš 2 agda transliatorių esančių oficialioje repozitorijoje. Agda kodą transliuoja į netipizuotą lambda skaičiavimą (*angl.* untyped lambda calculus) ir tada transformuoja į ECMASript (dar žinomą kaip JavaScript) [**html_agda**]. Šis transliatorius buvo sukurtas su intencija, kad būtų galima rašyti internetines aplikacijas su Agda programavimo kalba.

4.3. Epic programavimo kalbos transliatorius

Epigram kompiliatorius (*angl.* Epigram Compiler, EPIC) - griežta funkcinė kalba, kuri yra sukurta, tam, kad būtų naudojama kaip posistemė funkcinėms kalboms su priklausomais tipais, norint gauti paleidžiamąjį failą, EPIC yra paverčiama į C programavimo kalbos kodą. Šis transliatorius stipriai įtakojo Utrecht transliatoriaus vystymą [**Utrecht**].

Pagal šiame darbe atliktus metrikų patikrinimus šis transliatorius atsilieka programų veikimo greičiu nuo MAlonzo, tačiau jo sugeneruotas kodas yra mažesnių apimčių.

5. Agda transliavimas į Go

Šiame skyriuje aprašomi egzistuojančių transliatorių bendri bruožai, Agda programavimo kalbos tarpinė reprezentacija bei galimas Agda programavimo kalbos konstruktų transliavimas į Go programavimo kalbos konstruktus.

5.1. Egzituojančių transliatorių panašumai

Apžvelgus šiuo metu esančius Agda kalbos transliatorius galima įžvelgti keletą bendrų bruožų:

- Visi iš šių transliatorių naudoja išorinę funkcijos sąsają (*angl.* foreign function interface), kurios pagalba galima susieti Agda funkcijas ir duomenų tipus su tos kalbos tipais į kurią yra konvertuojamas Agda kodas.
- Visi iš šių transliatorių atlieka daugkartinius žingsnius kodą verčiant į kitos kalbos kodą, paprastai Agda kodas pirmiausia yra paverčiamas į vidinę sintaksę iš kurios jis verčiamas į tarpine reprezentacija, kuri yra skirtinga kiekvienam transliatoriui ir tik tada iš tos tarpinės reprezentacijos kodas yra verčiamas arba į taikomos kalbos kodą arba į taikomos kalbos tarpinę reprezentaciją.
- Visi iš šių transliatorių taikosi į funkcinę arba lambda skaičiavimo tipo kalbą (pasak A. Jeffrey ECMAScript tarpinė reprezentacija yra netipizuotas lambda skaičiavimas su įrašais [**html_agda**])

5.2. Agda programavimo kalbos tarpinė reprezentacija

Kompiliuojant agda failus jie praeina pro kelis žingsnius, kol pasiekia reprezentaciją, kuri yra naudojama transformavimui į kitas programavimo kalbas:

```
*.agda file
   ==[ parser (Lexer.x + Parser.y) ]==>
Concrete syntax
   ==[ nicifier (Syntax.Concrete.Definitions) ]==>
'Nice' concrete syntax
   ==[ scope checking (Syntax.Translation.ConcreteToAbstract) ]==>
Abstract syntax
   ==[ type checking (TypeChecking.Rules.*) ]==>
Internal syntax
   ==[ Agda.Compiler.ToTreeless ]==>
Treeless syntax
   ==[ different backends (Compiler.MAlonzo.*, Compiler.JS.*, ...) ]==>
Source code
   ==[ different compilers (GHC compiler, ...) ]==>
Executable
```

1 pav. Žingsniai kol Agda kodas yra paverčiamas kitos kalbos vykdomuoju failu [**AgdaWeb**]

Pagal šį paveikslėlį matome, kad kompiliuojant Agda failą jis praeina pro tokias stadijas:

- Konkreti sintaksė - tai auškto lygio „cukruota“ (*angl.* sugared) sintaksė, ji visiškai atitinka programuotojo įvestą kodą
- Abstrakti sintaksė - tai kodo versija prieš tipų patikrinimus, į ją įeina apimties analizė, operatorių pirmenybės išsiaiškinimas. Šioje reprezentacijoje yra atliekami tipų patikrinimai
- Vidinė sintaksė - tai paskutinis sintaksės transformavimo etapas prieš atiduodant ją posistemėms. Šiame etape termai yra tinkamai apibrėžti ir tinkamai tipizuoti. Kuriant vidinę sintaksę yra atliekami termų saugumo patikrinimai:
  - Funkcijų baigtumo - patikrinama ar rekursiškai aprašytos funkcijos yra baigtinės.
  - Duomenų tipų pozityvumo - užtikrinama, kad duomenų tipo indukcinio konstruktoriaus parametro argumentai nėra to paties tipo, kaip aprašomas duomenų tipas.
  - Funkcijų apimties - patikrinama ar šablonų patikrinimas apima visus galimus variantus.

  Vidinės sintaksės atvaizdavimui yra naudojamas Haskell duomenų tipas Term.

```
data Term = Var {-# UNPACK #-} !Int Elims
          | Lam ArgInfo (Abs Term)
          | Lit Literal
          | Def QName Elims
          | Con ConHead ConInfo Elimsrs.
          | Pi (Dom Type) (Abs Type)
          | Sort Sort
          | Level Level
          | MetaV {-# UNPACK #-} !MetaId Elims
```

2 pav. Haskell duomenų tipas, kuris yra naudojamas vidinės sintaksės atvaizdavimui

- Bemedė sintaksė (*angl.* Treeless syntax) - tai sintaksė, kuri skirta kompiliatoriaus posistemei (*angl.* compiler backend) naudoti. Tai yra šiek tiek žemesnio lygio stuktūra negu vidinė sintaksė ir joje nėra atliekami tipų patikrinimai. Šioje sintakėse naudojama atvėjo išraiškos (*angl.* case expressions) vietoje atvėjo medžių (*angl.* case trees), kurie naudojami vidinėje sitaksėje. Šios išraiškos yra Agda funkcijos reprezentacija vidinėje bei bemedėje sintaksėje, iš esmės jos yra panašios, pagrindinis skirtumas yra tas, kad atvėjo išraiškos yra A-normalios formos (*angl.* A-normal-form), tai reiškia, kad kiekvienam patikrinimui yra naudojamas tik vienas kintamasis, kai tuo tarpu atvėjo medžiuose gali būti naudojama ne vienas kintamasis. Žemiau galime matyti bemedę sintaksę išreikštą Haskell duomenų tipu „TTerm“:

```
data TTerm = TVar Int
           | TPrim TPrim
           | TDef QName
           | TApp TTerm Args
           | TLam TTerm
           | TLit Literal
           | TCon QName
           | TLet TTerm TTerm
           | TCase Int CaseInfo TTerm [TAlt]
           | TUnit
           | TSort
           | TErased
           | TCoerce TTerm
           | TError TError
```

3 pav. Haskell duomenų tipas kuriuo išreikšta bemedė sintaksė

Tarkime turime paprastą sudėties funkcija Agda kalboje:

```
_+_ : Nat -> Nat -> Nat
zero + m = m
suc n + m = suc (n + m)
```

Pavertus į bemedę struktūrą ta pati funkcija atrodo taip:

```
λ a b ->
  case a of
    demo.Nat.zero -> b
    demo.Nat.suc c -> demo.Nat.suc (demo._+_ c b)
```

Čia matome, kad šioje stuktūroje turime atvėjo išraiškas po vieną kintamąjį, šablonų derinimas vyksta nurodant pilną konstruktoriaus vardą (šiuo atveju „demo“ yra modulio pavadinimas)

5.3. Agda Duomenų tipų konvertavimas į Go

Go programavimo kalba neturi nei indukcinių nei bendrinių algebrinių tipų, todėl reikia surasti reprezentaciją Go kalboje, kuri yra panašiausia į Agda indukcinius tipus.

Vienas iš būdų, kaip galima aprašyti Bendrinius algebrinius tipus (arba indukcinius tipus) yra sąsajos (ang. interface) ir struktūros (ang. struct) pagalba. Tarkime turime sąrašo duomenų tipą Agda kalboje:

```
data Vec (A : Set) : Nat -> Set where
  empty : Vec A zero
  append : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

Iš esmės panašią struktūrą Go kalboje galima sukurti duomenų tipą pavertus į sąsają, o konstruktorius į struktūras:

```
type Vec interface{}
type empty struct{}
type append struct {
   value interface{}
   tail Vec
}
```

Jeigu norėtume sukurti sąrašą iš 2 elementų Go kalboje, galėtume parašyti tokią eilutę:

```
append{1, append{4, empty{}}}
```

Kadangi šioje struktūroje elementus saugome kaip „interface“ tipo elementus, gavę rezultatą bei norėdami naudoti funkcijas dirbančias su konkrečiu tipu, juos reikia paversti atgal į tipą, kuris buvo įdėtas į duomenų struktūrą.

5.4. Agda funkcijų konvertavimas į Go

5.4.1. Vidinė Go sintaksės reprezentacija

Norint transformuoti Agda kodą į kitos kalbos kodą paprastai naudojama tam tikra vidinė kodo reprezentaciją į kurią pirmiausia yra paverčiamas Agda kodas. Galime apribrėžti Haskell duomenų tipą, kuris reprezentuos Go kalbos poaibį, į kurį transformuojame Agda funkcijas:

```
data GoTerm = Self
            | Local LocalId
            | Global GlobalId
            | GoVar Nat
            | GoSwitch GoTerm [GoCase]
            | GoMethodCall MemberId [GoMethodCallParam]
            | GoCreateStruct MemberId [GoTerm]
            | GoIf GoTerm GoTerm GoTerm
            | GoLet String GoTerm GoTerm
            | PrimOp GoTerm GoTerm GoTerm
            | ReturnExpression GoTerm TypeId
            | Integer Integer
            | Const String
            | UndefinedTerm
            | GoErased
            | Null
```

4 pav. Haskell duomenų tipas kuriuo išreikšta Go kalbos poaibio vidinė sintaksė

Šis duomenų tipas veikia kaip tarpinė reprezentacija transformuojant Agda kodą į Go. Transformuojant Agda funkcijų kodą, jis pirmiausia yra paverčiamas į šią tarpinę reprezentaciją, iš kurios vėliau yra generuojamas Go kodas.

Anksčiau pademonstruotas Haskell duomenų tipas apibrėžiantis bemedę sintaksę (3 pav.) bei šis duomenų tipas (4 pav.) apibrėžia galimus funkcijos kūno elementus (be funkcijos aprašymo), taigi galime susieti šiuos du duomenų tipus nurodydami šiame darbe palaikomas transformavimo taisykles:

1 lentelė. Agda transformavimus vaizduojanti lentelė

Bemedės struktūros Term reikšmė	**Go Term reikšmė**
TVar Int	GoVar Nat
TDef QName	GoMethodCall MemberId []
TCase Int CaseInfo TTerm [TAlt]	GoSwitch GoVar [GoCase]
TApp TTerm(TDef) [TTerm]	GoMethodCall MemberId [GoMethodCallParam]
TApp TTerm(TCon) [TTerm]	GoCreateStruct MemberId [GoTerm]
TApp TTerm(TVar) [TTerm]	GoMethodCall GoVar [GoMethodCallParam]
TApp TTerm(TPrif PIf) [TTerm]	GoIf GoTerm GoTerm GoTerm
TApp TTerm(TPrim) [TTerm]	PrimOp GoTerm GoTerm GoTerm
TLet TTerm TTerm	GoLet GoTerm GoTerm
TLit Literal	GoLit GoLiteral
TCon QName	GoCreateStruct MemberId []
TPrim TPrim	PrimOp GoTerm GoTerm GoTerm

Šioje lentelėje pažymėti atliekami transformavimai. Jeigu reikšmė įrašyta skliausteliuose, reiškia, kad tikrinama tipo konkreti reikšmė esanti skliausteliuose. Atlikus transformavimą iš bemedės struktūros į mūsų aprašytą Go termą vėliau yra generuojamas Go kodas

- GoVar a - kintamasis, kurio reikšmė yra ’a’ simbolis, kurio reikšmė yra padidinta parametro „a“ reikšmės dydžiu. Pavyzdžiui jeigu a reikšmė yra 2, tada kintamojo vardas bus ’c’
- GoMethodCall merberId goTerms - Go metodo kvietimas, kur memberId yra metodo vardas, o goTerms yra šio metodo argumentai, kurie savyje laiko GoTerm bei, jeigu reikia, argumento tipą.
- GoSwtich goVar cases - tai yra Go „switch-case“ konstruktas, kur goVar nurodo kintamąjį, kurio reikšmė yra tikrinama, o cases apibrėžia galimus variantus
- GoCreateStruct memberId goTerms - Go struktūros kūrimas, kur memberId yra structūros vardas, o goTerms yra šios structūros elementų reikšmės
- GoIf term1 term2 term3 - Go sąlyginis sakinys, kur term1 yra primityvi if sakinio operacija, kuri gražina loginę reikšmę, pagal kurią yra vykdoma arba term2, arba term3 operacijos.
- PrimOp goTerm1 goTerm2 goTerm3 - Go primityvi operacija, kur goTerm1 apibrėžia primityvios operacijos tipą, o goTerm2 ir goTerm3 yra šios operacijos argumentai
- GoLet goTerm1 goTerm2 - tai naujo kintamojo sukūrimas, goTerm1 apibrėžia to kintamojo reikšmę, o goTerm2 yra tolesnę funkcijos operacija
- GoLit GoLiteral - tai primityvi reikšmė (šiuo metu palaikoma tik sveikojo skaičiaus reikšmė)

5.4.2. Paprastos funkcijos konvertavimas

Tarkime naudojame tą patį sąrašą į kurį jau įdėti keli natūralieji skaičiai, norėdami sudėti tuos natūraliuosius skaičius, Agda kalboje tokios funkcijos atrodytų taip:

```
add : Nat -> Nat -> Nat
add zero m = m
add (suc n) m = suc (n + m)

sum : List Nat -> Nat
sum empty = zero
sum (append value tail) = add value (sum tail)
```

Go programavimo kalboje galėtume sukurti tokius metodus:

```
func add(a interface{}) func(interface{}) Nat {
   return func(b interface{}) Nat {
      switch type_a := a.(type) {
         case Nat_zero:
            return b
         case Nat_suc:
            c := type_a._1
            _ = c
            return Nat_suc{add(c)(b)}
         default:
            _ = type_a
            panic("Unreachable")
      }
   }
}
func sum(a List) Nat {
  switch type_a := a.(type) {
  case List_empty:
    return Nat_zero{}
   case List_append :
      b := type_a.a;
       _ = b;
      c := type_a.b;
       _ = c;
    return add(sum(c))(b)
  default:
    panic("error")
  }
}
```

Kur natūralusis skaičius apibrėžtas taip:

```
type Nat interface{}
type Nat_zero struct{}
type Nat_suc struct {
   value Nat
}
```

Pirmojoje Go funkcijoje matome, kad funkcijos yra suskaidytos į mažesnes funkcijas po vieną argumentą, kad atitiktų funkcinio programavimo principus ir esant poreikiui į kitą metodą būtų galima paduoti funkcija add(5), kuri prie duoto natūraliojo skaičiaus prideda 5. Tačiau toks užrašymas prideda sudėtingumo funkcijoms, kadangi Go kalboje privaloma tiksliai nurodyti funkcijos grąžinimo tipą, taigi kuo daugiau parametrų priima funkcija, tuo labiau išsiplečia jos grąžinamas tipas.

5.4.3. Standartiniai tipai

Agda kalboje pragmų pagalba yra įmanoma nurodyti tam tikrų tipų atitikmenis Haskell kalboje, tuo galima pasinaudoti ir konvertuodami kodą į Go. Prie natūraliojo skaičiaus duomenų tipo pridėjus eilutę

```
{-# BUILTIN NATURAL Nat #-}
```

Transliavimo metu galime nurodyti į kokį konkrečiai tipą reikia konvertuoti labiausiai naudojamus tipus, tokius kaip sveikieji skaičiai, sąrašai ar loginis tipas. Šios pragmos yra skirtos transliavimui į Haskell kalbą, tačiau dalį jų galime panaudoti ir transliavime į Go.

Konvertuodami natūraliuosius skaičius į Go programavimo kalbą turime kelis variantus:

- Primityvus tipas „int“
- Didelių skaičių biblioteka „math“, kuri turi „bigInt“ tipą
- C kalbos bibliotekos „gmp“ sveikojo skaičiaus todo kuris yra sukurtas kaip „bigInt“ bibliotekos alternatyva.

Primityvus tipas yra greitesnis už šias bibliotekas, tačiau yra stipriai ribojamas maksimalaus skaičiaus dydžio, todėl norint kad duomenų tipas atitiktų Agda duomenų tipą, šiame darbe renkamasi tarp math ir gmp bibliotekų. Galime palyginti šių bibliotekų našumą skaičiuodami faktorialo funkciją dideliems skaičiams:

2 lentelė. „math“ ir „gmp“ bibliotekų faktorialo skaičiavimo efektyvumo palyginimas

	math	**gmp**
n = 100	0.02ms	0.31ms
n = 1000	0.39ms	3.41ms
n = 10000	38.85ms	54.57ms
n = 100000	18.58s	10.86s

Kaip matome šioje lentelėje, „math“ biblioteka veikia efektyviau su mažesniais skaičiais ir tik su pačiais didžiausiais skaičiais veikia lėčiau negu „gmp“ biblioteka. Tačiau santykinis skirtumas yra didesnis skaičiuojant mažesnes reikšmes, todėl šiame darbe buvo pasirinkta naudoti „math“ bibliotekos „bigInt“ tipą Agda natūraliesiems skaičiams atvaizduoti.

Panaudojus pragmą, su kuria nurodome į kokį tipą transliuojame natūralųjį skaičių, taip pat turime nurodyti metodų atitikmenis, kurie yra skirti dirbti su šiuo primityviu tipu, tai tokios operacijos kaip sudėtis, atimtis, daugyba, dalyba bei lygybė. Šiam tikslui pasiekti buvo sukurtas pagalbinis paketas, kurio metodai yra kviečiami norint atlikti aukščiau paminėtas operacijas. Pavyzdžiui, prie tai aprašytame „Nat“ duomenų tipo aprašyme nurodę, kad tai yra standartinis natūralusis skaičius, gauname tokį prieš tai aprašytą sudėties metodą:

```
func add(a interface{}) func(interface{}) *big.Int {
   return func(b interface{}) *big.Int {
      if ( helper.Equals((big.NewInt(0)),(a)) ) {
         return b.(*big.Int)
      } else {
         c := helper.Minus((a),(big.NewInt(1)))
         return helper.Add((big.NewInt(1)),(add(c) (b))).(*big.Int)
      }
   }
}
```

Šiame metode naudojamas mūsų nurodytas Go kalboje egzituojantis tipas, kuris yra tinkamas saugoti natūraliesiems skaičiams.

5.4.4. Bendrinių tipų konvertavimas

Bendriniai tipai šioje realizacijoje būtini transliuojant funkcijas, kurios vienas iš parametrų yra funkcijos tipo. Pavyzdžiui, jeigu turime funkciją „Map“, kuri pritaiko tam tikrą funkciją kiekvienam sąrašo elementui, vienas būdas ją transformuoti į Go būtų paverčiant visų argumentų tipus į „interface“ tipą:

```
func Map(a func(interface{}) interface{}) func(List) List {
```

```
    return func(b List) List {
        . . . .
    }
}
```

Tačiau šiuo atveju „Map“ funkciją galėtume kviesti perduodami tik tuos metodus, kurie priima ir grąžina „interface“ tipo elementus. Todėl, jeigu turėtume sąrašą sveikųjų skaičių ir norime prie jų pridėti 5, negalime pasinaudoti jau aprašyta funkcija „add“, kadangi, jeigu į „Map“ perduotume pirmą parametrą „add(NewInt(5))“, gautume kompiliavimo klaidą, nes nesutaptų priimamos ir perduodamos funkcijų aprašai. Šiai problemai išspręsti galime pasinaudoti bendriniais tipais iš prieš tai minėto pasiūlymo. Tuo atveju „Map“ funkciją galima būtų aprašyti taip:

```
func Map[T1, T0 any](a func(T1) T0) func(List) List {
    return func(b List) List {
        . . . .
    }
}
```

Taip aprašius funkciją, kompiliuodami Go kodą, sugeneruosime skirtingas „Map“ variacijas kiekvienam kvietimui kuris perduoda funkciją su skirtingu aprašu.

5.4.5. Raktinio žodžio „return“ pridėjimas

Transliavimo metu yra sudėtinga nustatyti, kurioje vietoje naudoti „return“ raktinį žodį, kadangi tuo metu dar nėra aišku, kokia tiksliai operacija bus naudojama ir ar tai yra paskutinė operacija šakoje. Šiai situacijai išspręsti, po metodo transliavimo yra kviečiama funkcija, kuri pereina per visą medį, tikrindama ar operacija toje šakoje yra paskutinė. Šiam tikslui pasiekti buvo apsibrėžtas sąrašas, kuris nurodo operacijas, kurios grąžina konkrečią reikšmę, tai tokios operacijos kaip metodo kvietimas, struktūros sukūrimas ar tiesiog kintamojo grąžinimas. Jeigu operacija yra paskutinė, tada tos operacijos duomenų tipas yra apvelkamas nauju tipu „ReturnExpression“, kuris kodo generatoriui nurodo, kad ši operacija pradžioje turi turėti „return“ raktinį žodį.

Taip pat, kaip matoma Go tarpinės reprezentacijos Haskell duomenų tipe (4 pav.), „ReturnExpression“ tipas taip pat saugo grąžinamo tipo informaciją. Tai yra reikalinga norint užtikrinti, kad grąžinamas toks tipas, kurį nurodo metodo aprašymas.

5.5. Įrodymai

Transliuoti įrodymų iš Agda programavimo kalbos į Go nėra reikalinga bei prasminga, tai nėra daroma nei MAlonzo, nei JavaScript transliatoriuose. Verčiant šių funkcijų kodą iš Agda abstrakčios sintaksės į bemedę yra gaunamas rezultatas „TErased“, kuris nusako, kad tokia funkcija gali būti pašalinta.

5.6. Visų kintamųjų bei importuotų bibliotekų panaudojimas

Kadangi Go neleidžia kompiliuoti kodo, kuriame yra nenaudojamų kintamųjų ar importuotų bibliotekų, transliuodami Agda kodą privalome užtikrinti, kad tokių situacijų neatsitiktų. Kaip jau buvo galima matyti Go kodo pavyzdžiuose, transliuodami Agda koda yra parašome priskyrimą, pavyzdžiui, „_ = c“. Ši eilutė reiškia, kad kintamasis „c“ yra priskiriamas tuščiam identifikatoriui.

Transliuodami Agda kodą taip pat galime susidurti su situacija, kai nurodomos bibliotekos, kurios tame pakete nėra naudojamos. Norint išvengti tokių klaidų, transliavimo metu kiekviename pakete pridedame papildomą tipo kintamąjį „type GoImportable int“. Šis kintamasis yra panaudojamas kiekviename pakete, kuris importuoja kitą paketą. Tarkime, turime „package a“ ir „package b“, kur paketas a importuoja paketą „b“, tada paketo „a“ kode pridėsime eilutę „type _ a.GoImportable“. Ši eilutė užtikrina, kad Go kompiliatorius neišmes klaidos dėl nenaudojamos importuotos bibliotekos.

5.7. Simbolių užkodavimas verčiant į Go

Kaip jau buvo minėta ankstesniame skyriuje, Agda leidžia praktiškai bet kokius vardus funkcijoms, bei duomenų tipams, kai tuo tarpu Go kalboje jie yra žymiai labiau ribojami. MAlonzo kompiliatoriuje į Haskell tai apeinama duomenų struktūroms, konstruktoriams, bei funkcijoms įvardinti naudojant naujai sukurtus vardus, pavyzdžiui:

```
name2 = "demo.Nat"
d2 = ()
data T2 = C4 | C6 T2
```

kur d2 yra funkcijos vardas, T2 yra duomenų tipo vardas, o name2 yra eilutės tipo kintamasis, kuris saugo funkcijos vardą Agda kode. Toks metodas yra tinkamas, kai yra generuojamas vykdomasis failas, bet norint sugeneruoti kodą, kuris yra kviečiamas iš Go kodo toks sprendimas yra netinkamas.

Paprasčiausias sprendimas šiam klausimui būtų apriboti simbolius, kuriuos galima naudoti rašant Agda kodą. Tačiau tuo atveju norėdami transliuoti Agda standartinės bibliotekos kodą susiduriame su problemomis, kadangi nėra galimybių perrašyti visą standartinės bibliotekos kodą pakeičiant visus Go kalboje nepriimamus simbolius. Šiai problemai apeiti galime sukurti užkodavimą, kuris transliavimo metu patikrina ar kiekvienas transliuojamas simbolis yra apatinis brūkšnys arba „unicode“ raidės simbolis. Jeigu ši sąlygą yra nepatenkinama, tada simbolį verčiame į jo dešimtainio skaičiaus reprezentaciją, prie jos pridedami raidę „u“, pavyzdžiui, versdami dolerio ženklą „$“, gauname kodą „u36“.

6. Našumo testai

Šiame skyriuje aprašomi 3 skirtingų funkcijų veikimo greičio bei atminties naudojimo palyginimai. Lyginimas atliekamas tarp Agda kodo sugeneruoto į Go bei kodo sugeneruoto į Haskell naudojantis Malonnzo kompiliatoriumi. Visi palyginimai buvo leidžiami kompiuteryje su „AMD Ryzen 7 3750H“ procesoriumi, bei 16GB atminties. Go kalboje matavimai atlikti naudojantis baziniu „testing“ paketu, o Haskell kalboje „criterion“ bei „weigth“ bibliotekomis.

Pirmasis testas atliktas su faktorialo funkcija (žr. 7 priedas):

3 lentelė. Go ir Haskell faktorialo skaičiavimo efektyvumo palyginimas

		Go	**Haskell**
n = 100	laikas	0.02ms	0.0044ms
	atmintis	11.6KB	9.25KB
n = 1000	laikas	0.39ms	0.12ms
	atmintis	0.59MB	0.63MB
n = 10000	laikas	38.85ms	11.78ms
	atmintis	74.4MB	84.11MB
n = 100000	laikas	18.58s	1.35s
	atmintis	9.38GB	11.28GB

Kaip matoma šioje lentelėje, Haskell veikia greičiau su visais n, tačiau Go kodas naudoja mažiau atminties pasiekus n = 100000.

Antrasis testas skaičiuoja pirminių skaičių kiekį iki duoto natūralausiaus skaičiaus n (žr. 7 priedas). Kaip matoma lentelėje žemiau, Ši funkcija paversta į Go, palyginus su Haskell, veikia labai lėtai, su n reikšme 1000, nėra gaunamas rezultatas kadangi pritrūkstama atminties.

4 lentelė. Go ir Haskell pirminių skaičių kieko radimo funckija

		Go	**Haskell**
n = 100	laikas	27.07ms	0.15ms
	atmintis	8.75MB	0.17MB
n = 500	laikas	44.7s	3.24ms
	atmintis	13.7GB	4MB
n = 1000	laikas	-	13.07ms
	atmintis	-	16.1MB

Atlikus analizę buvo pastebėta, kad didžioji dalis laiko yra sunaudojama rekursiniuose kvietimuose todėl, kad „let“ išraiškoje yra priskiriamas metodo iškvietimas. Tai nesukelia problemų Haskell kalboje, kadangi yra taikomas tingus inicializavimus, kurio dėka reikšmė nėra skaičiuojama, kol ji nėra naudojama. Šiaip problemai išspręsti siūlomas pakeitimas, nepriskirti medoto kvietimo kintamąjam, tačiau jį pridėti toje vietoje, kur yra naudojamas tas kintamasis. Atlikus šį pakeitimą pirminių skaičiaus kieko skaičiavimo funkcijoje gaunami žymiai geresni rezultatai:

5 lentelė. Go ir Haskell pirminių skaičių kieko radimo funckija

		Go	**Haskell**
n = 100	laikas	1.86ms	0.15ms
	atmintis	0.65MB	0.17MB
n = 500	laikas	44.24ms	3.24ms
	atmintis	15.33MB	4MB
n = 1000	laikas	178ms	13.07ms
	atmintis	60.7MB	16.1MB
n = 10000	laikas	19.9s	1.27s
	atmintis	6GB	1.6GB

Trečiasis testas atliktas leidžiant funkciją, kuri burburo rikiavimo metodu surikiuoja duotą sąrašą (žr. 7 priedas). Visų šio sąrašo elementų dydis yra iki 10.

6 lentelė. Go ir Haskell bubble sort funkcija

		Go	**Haskell**
n = 100	laikas	19.24ms	0.81ms
	atmintis	57.78MB	pending
n = 1000	laikas	1.93s	0.09s
	atmintis	6.16GB	pending

Galime matyti, kad rezultatai santykinai yra prastesni negu prieš tai leistų funkcijų. Buvo pastebėta, kad didelė dalis laiko, iki 30%, yra išnaudojama atliekant šiukšlių surinkimą (*angl.* garbage collection). Taip pat, veikimo greitis numatomai turėtų žymiai pagreitėti implementavus išorinę funkcijos sąsaja bei šį vektoriaus tipą pavertus į Go sąrašo tipą.

7. Verifikavimas

Ateities darbai

1. Išorinė funkcijos sąsaja. Šiuo metu nėra galimybės tiesiogiai susieti Agda funkcijų bei duomenų tipų su konkrečiais Go kalbos elementais. Tam tikslui pasiekti yra reikalinga išorinė funkcijos sąsaja, su kuria pragmu pagalba būtų galima nurodyti kokia Go funkcija ar struktūra atitinka pasirinktas Agda funkcijas ar duomenų tipus.
2. Bendriniai duomenų tipai. Šiuo metu transformuojami Agda duomenų tipai yra paverčiami į tokius duomenų tipas, kurių laukai yra tuščios sąsajos tipo, tai reiškia, kad metodai dirbantys su jais, savo aprašyme negali naudoti bendrinių tipų. Tai iš dalies galėtų būti sprendžiama išorinės funkcijos sąsajos pagalba tam tikrus tipus verčiant į jau egzistuojančias Go struktūras, tačiau norint naudotis visais Agda privalumais yra reikalingi bendriniai duomenų tipai aprašyti Go kalboje.
3. Įgyvendinti įrašo tipų transliavimą.
4. Įgyvendinti automatinį Go modulio sukūrimą ar paketų perkėlimą į Go bibliotekų aplanką. Importuojant skirtingus Go paketus, jie yra imami arba iš go bibliotekų aplanko, arba, jeigu yra nurodytas modulis, iš modulio aplanko. Dėl šios priežasties reikalingas funkcionalumas, kuris leidžia arba perkelti sukompiliuotas klases į Go bibliotekų aplanką, jeigu jos yra bendros visiems projektams, arba sukurti Go modulį, kad to nereikėtų kiekvieną kartą daryti rankinių būdu.

Rezultatai

1. Aprašytos Agda funkcijų transformavimo taisyklės
2. Įgyvendintos aprašytos taisyklės
3. Pateikti našumo palyginimai tarp Go ir Haskell sugeneruotų programų
4. Nurodyti esamo transliatoriaus trūkumai
5. Pateikti siūlomi ateities darbai

Išvados

1. Agda kodą galima transliuoti į Go kalbos kodą
2. Su šiuo transliatoriaus įgyvendinimu sugeneruotas kodas veikia 15-20 kartų lečiau, negu kodas sugeneruotas į Haskell

1 priedas. Agda kodas paverstas į kitas kalbas

Agda sąrašo tipas ir funkcijos, kurios sudeda sąrašo elementus ir grąžina natūralųjį skaičių paverstos į Haskell ir Javascript kalbas.
Haskell:

```
name46 = "demo.List"
d46 a0 = ()
data T46 = C50 | C52 AgdaAny T46
name36 = "demo.add"
d36 :: Integer -> Integer -> Integer
d36 v0 v1
  = case coe v0 of
      0 -> coe v1
      _ -> let v2 = subInt (coe v0) (coe (1 :: Integer)) in
             coe addInt (coe (1 :: Integer)) (coe d24 (coe v2) (coe v1))
name54 = "demo.sum"
d54 :: T46 -> Integer
d54 v0
  = case coe v0 of
      C50 -> coe (0 :: Integer)
      C52 v1 v2 -> coe d36 (coe v1) (coe d54 (coe v2))
      _ -> MAlonzo.RTE.mazUnreachableError
```

JavaScript:

```
exports["List"] = {};
exports["List"]["empty"] = a => a["empty"]();
exports["List"]["append"] = a => b => c => c["append"](a,b);
exports["add"] = a => b => (
  agdaRTS.uprimIntegerEqual(agdaRTS.primIntegerFromString("0"),a)? b:
    agdaRTS.uprimIntegerPlus(
    agdaRTS.primIntegerFromString("1"),
    exports["_+_"](
      agdaRTS.uprimIntegerMinus(a,agdaRTS.primIntegerFromString("1"))
    )(b)
  )
  );
exports["sum"] = a => a({
    "append": (b,c) => exports["add"](b)(exports["sum"](c)),
    "empty": () => agdaRTS.primIntegerFromString("0")
  });
```

2 priedas. Agda faktorialo funkcija paversta į kitas kalbas

Agda faktorialo funkcija paverstą į Haskell ir Go.

Agda:

```
_! : Nat -> Nat
0 ! = 1
(suc n) ! = (suc n) * (n !)
```

Haskell:

```
name32 = "demo._!"
d32 :: Integer -> Integer
d32 v0
  = case coe v0 of
      0 -> coe (1 :: Integer)
      _ -> let v1 = subInt (coe v0) (coe (1 :: Integer)) in
           coe mulInt (coe v0) (coe d32 (coe v1))
```

Go:

```
func F_u33(A interface{}) *big.Int {
  if helper.Equals((big.NewInt(0)), (A)) {
    return helper.Id(big.NewInt(1)).(*big.Int)
  } else {
    B := helper.Minus((A), (big.NewInt(1)))
    return helper.Id(helper.Multiply((A), (F_u33(B)))).(*big.Int)
  }
```

3 priedas. Agda pirminių skaičiaus kieko radimo funkcija paversta į kitas kalbas

Agda pirminių skaičiaus kiekio radimo funkcija paverstą į Haskell ir Go.

Agda:

```
isPrime : Nat -> Bool
isPrime 0 = false
isPrime 1 = false
isPrime n with dividersCount n n
  where
  dividersCount : Nat -> Nat -> Nat
  dividersCount 0 _ = 0
  dividersCount _ 0 = 0
  dividersCount (suc n) m with mod-helper 0 n m n
  dividersCount (suc n) m | 0 = suc (dividersCount n m)
  dividersCount (suc n) m | _ = dividersCount n m
isPrime n | 2 = true
isPrime n | _ = false

countPrimes : Nat -> Nat
countPrimes 0 = 0
countPrimes (suc n) with isPrime (suc n)
countPrimes (suc n) | false = countPrimes n
countPrimes (suc n) | true = suc (countPrimes n)
```

Haskell:

```
name290 = "demo.isPrime"
d290 :: Integer -> Bool
d290 v0
   = let v1
           = let v1 = coe du298 (coe v0) (coe v0) in
             case coe v1 of
               2 -> coe MAlonzo.Code.Agda.Builtin.Bool.C10
               _ | coe geqInt (coe v1) (coe (2 :: Integer)) ->
                     coe MAlonzo.Code.Agda.Builtin.Bool.C8
               1 -> coe MAlonzo.Code.Agda.Builtin.Bool.C8
               _ -> coe MAlonzo.Code.Agda.Builtin.Bool.C8 in
     case coe v0 of
       0 -> coe MAlonzo.Code.Agda.Builtin.Bool.C8
       1 -> coe MAlonzo.Code.Agda.Builtin.Bool.C8
       _ -> coe v1
name298 = "demo._.dividersCount"
```

```
d298 :: Integer -> Integer -> Integer -> Integer
d298 v0 v1 v2 = du298 v1 v2
du298 :: Integer -> Integer -> Integer
du298 v0 v1
  = case coe v0 of
      0 -> coe (0 :: Integer)
      _ -> let v2 = subInt (coe v0) (coe (1 :: Integer)) in
           let v3
                 = let v3 = remInt (coe v1) (coe v0) in
                   let v4 = coe du298 (coe v2) (coe v1) in
                   case coe v3 of
                     0 -> coe addInt (coe (1 :: Integer)) (coe du298 (coe
                          v2) (coe v1))
                     _ -> coe v4 in
           case coe v1 of
             0 -> coe (0 :: Integer)
             _ -> coe v3
name324 = "demo.countPrimes"
d324 :: Integer -> Integer
d324 v0
  = case coe v0 of
      0 -> coe (0 :: Integer)
      _ -> let v1 = subInt (coe v0) (coe (1 :: Integer)) in
           let v2 = d290 (coe v0) in
           if coe v2
             then coe addInt (coe (1 :: Integer)) (coe d324 (coe v1))
             else coe d324 (coe v1)
```

Go:

```
func FisPrime(A interface{}) Agda_Builtin_Bool.Bool {
  B := F_NameIdu32296u321274468892439276514_dividersCount(A)(A)
  if helper.Equals((big.NewInt(2)), (B)) {
    return helper.Id(Agda_Builtin_Bool.Bool_true{}).(Agda_Builtin_Bool.Bool)
  } else {
    if helper.Equals((big.NewInt(1)), (B)) {
      return helper.Id(Agda_Builtin_Bool.Bool_false{}).(Agda_Builtin_Bool.Bool)
    } else {
      if helper.MoreOrEquals((B), (big.NewInt(2))) {
        return
            helper.Id(Agda_Builtin_Bool.Bool_false{}).(Agda_Builtin_Bool.Bool)
      } else {
        return
            helper.Id(Agda_Builtin_Bool.Bool_false{}).(Agda_Builtin_Bool.Bool)
```

```
        }
      }
    }

    if helper.Equals((big.NewInt(0)), (A)) {
      return helper.Id(Agda_Builtin_Bool.Bool_false{}).(Agda_Builtin_Bool.Bool)
    } else {
      if helper.Equals((big.NewInt(1)), (A)) {
        return helper.Id(Agda_Builtin_Bool.Bool_false{}).(Agda_Builtin_Bool.Bool)
      } else {
        return helper.Id(B).(Agda_Builtin_Bool.Bool)
      }

    }
}

func F_NameIdu32296u321274468892439276514_dividersCount(A interface{})
    func(interface{}) *big.Int {
    return func(B interface{}) *big.Int {
      if helper.Equals((big.NewInt(0)), (A)) {
        return helper.Id(big.NewInt(0)).(*big.Int)
      } else {
        C := helper.Minus((A), (big.NewInt(1)))
        D := helper.Mod((B), (A))
        E := F_NameIdu32296u321274468892439276514_dividersCount(C)(B)
        if helper.Equals((big.NewInt(0)), (D)) {
          return helper.Id(helper.Add((big.NewInt(1)),
              (F_NameIdu32296u321274468892439276514_dividersCount(C)(B)))).(*big.Int)
        } else {
          return helper.Id(E).(*big.Int)
        }
        if helper.Equals((big.NewInt(0)), (B)) {
          return helper.Id(big.NewInt(0)).(*big.Int)
        } else {
          return helper.Id(D).(*big.Int)
        }

      }
    }
}

func FcountPrimes(A interface{}) *big.Int {
    if helper.Equals((big.NewInt(0)), (A)) {
```

```
      return helper.Id(big.NewInt(0)).(*big.Int)
   } else {
      B := helper.Minus((A), (big.NewInt(1)))
      C := FisPrime(A)
      switch type_C := C.(type) {
      case Agda_Builtin_Bool.Bool_false:
      return helper.Id(FcountPrimes(B)).(*big.Int)
      case Agda_Builtin_Bool.Bool_true:
      return helper.Id(helper.Add((big.NewInt(1)), (FcountPrimes(B)))).(*big.Int)
      default:
         _ = type_C
         panic("Unreachable")
      }
   }
}
```

4 priedas. Agda rikiavimo funkcija paversta į kitas kalbas

Agda sąrašo rikiavimo funkcija paverstą į Haskell ir Go.

Agda:

```
{-# TERMINATING #-}
bubblesortiter :  {n} -> Vec Nat n -> Vec Nat n
bubblesortiter (appendV x (appendV y xs)) with x > y
bubblesortiter (appendV x (appendV y xs)) | true = appendV y (bubblesortiter
    (appendV x xs))
bubblesortiter (appendV x (appendV y xs)) | false = appendV x (bubblesortiter
    (appendV y xs))
bubblesortiter x = x


{-# TERMINATING #-}
bubblesort' :  {n} -> Vec Nat n -> Nat -> Vec Nat n
bubblesort' {n} xs i with n eq i
bubblesort' {n} xs i | true = xs
bubblesort' {n} xs i | false = bubblesort' (bubblesortiter xs) (suc i)

bubblesort :  {n} -> Vec Nat n -> Vec Nat n
bubblesort xs = bubblesort' xs 0
```

Haskell:

```
name338 = "demo.bubblesortiter"
d338 :: Integer -> T152 -> T152
d338 v0 v1
  = case coe v1 of
      C160 v3 v4
        -> case coe v0 of
             _ | coe geqInt (coe v0) (coe (1 :: Integer)) ->
                   case coe v4 of
                     C160 v6 v7
                       -> case coe v0 of
                            _ | coe geqInt (coe v0) (coe (2 :: Integer)) ->
                                  let v8 = d12 (coe v3) (coe v6) in
                                  if coe v8
                                    then coe
                                           C160 v6
                                           (d338
                                              (coe subInt (coe v0) (coe
                                                 (1 :: Integer)))
                                              (coe C160 v3 v7))
```

```
                                        else coe
                                                  C160 v3
                                                  (d338
                                                     (coe subInt (coe v0) (coe
                                                        (1 :: Integer)))
                                                     (coe C160 v6 v7))
                                  _ -> coe v1
                            _ -> coe v1
                      _ -> coe v1
          _ -> coe v1
name366 = "demo.bubblesort'"
d366 :: Integer -> T152 -> Integer -> T152
d366 v0 v1 v2
  = let v3 = d22 (coe v0) (coe v2) in
    if coe v3
      then coe v1
      else coe
                d366 (coe v0) (coe d338 (coe v0) (coe v1))
                (coe addInt (coe (1 :: Integer)) (coe v2))
name392 = "demo.bubblesort"
d392 :: Integer -> T152 -> T152
d392 v0 v1 = coe d366 (coe v0) (coe v1) (coe (0 :: Integer))
```

Go:

```
func Fbubblesortiter(A interface{}) func(Vec) Vec {
  return func(B Vec) Vec {
    C := B
    switch type_B := B.(type) {
    case Vec_appendV:
      D := type_B.A
      _ = D
      E := type_B.B
      _ = E
      if helper.MoreOrEquals((A), (big.NewInt(1))) {
        F := B
        switch type_E := E.(type) {
        case Vec_appendV:
          G := type_E.A
          _ = G
          H := type_E.B
          _ = H
          if helper.MoreOrEquals((A), (big.NewInt(2))) {
            I := F_u62_(D)(G)
```

```
                switch type_I := I.(type) {
                case Agda_Builtin_Bool.Bool_false:
            return helper.Id(Vec_appendV{D, Fbubblesortiter(helper.Minus((A),
                (big.NewInt(1))))(Vec_appendV{G, H})}).(Vec)
                case Agda_Builtin_Bool.Bool_true:
            return helper.Id(Vec_appendV{G, Fbubblesortiter(helper.Minus((A),
                (big.NewInt(1))))(Vec_appendV{D, H})}).(Vec)
                default:
                   _ = type_I
                   panic("Unreachable")
                }
             } else {
                return helper.Id(B).(Vec)
             }
      case Vec_emptyV:
         return helper.Id(F).(Vec)
            default:
               _ = type_E
               panic("Unreachable")
            }
         } else {
            return helper.Id(B).(Vec)
         }
   case Vec_emptyV:
      return helper.Id(C).(Vec)
      default:
         _ = type_B
         panic("Unreachable")
      }
   }
}
func Fbubblesortu39(A interface{}) func(Vec) func(interface{}) Vec {
   return func(B Vec) func(interface{}) Vec {
      return func(C interface{}) Vec {
         D := F_eq_(A)(C)
         switch type_D := D.(type) {
         case Agda_Builtin_Bool.Bool_false:
      return
          helper.Id(Fbubblesortu39(A)(Fbubblesortiter(A)(B))(helper.Add((big.NewInt(1)),
          (C)))).(Vec)
         case Agda_Builtin_Bool.Bool_true:
      return helper.Id(B).(Vec)
         default:
```

```
            _ = type_D
            panic("Unreachable")
        }
    }
    }
}
func Fbubblesort(A interface{}) func(Vec) Vec {
    return func(B Vec) Vec {
        return helper.Id(Fbubblesortu39(A)(B)(big.NewInt(0))).(Vec)
    }
}
```