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

Projektinis darbas

Atliko: 4 kurso studentas
Justas Tvarijonas (parašas)

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

Vilnius – 2021

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 [Hau15].

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 gražinti.Tai labai abstraktus apibrėžimas, ką ši programa atlieka, tačiau tai suteikia tam tikrą informaciją apie šios programos veikimą.

Priklausomų tipų sistemos [Zha92] leidžia tipams būti priklausomais nuo konkrečių reikšmių. Programavimo kalbos [Sit18; Stu16], 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 galimybe 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 [Lip11] sintaksę. Dabartinė Agda implementacija tipų patikrinimais siekia užtikrinti, kad programos bei įrodymai būtų teisingi. Tuo tarpu Go programavimo kalba [WP14] 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 kodą, kurį jau galėtų panaudoti egzituojantis Go kodas.

Taigi, šio darbo tikslas yra išanalizuoti galimybe generuoti Go programavimo kalbos kodą iš Agda kalbos kodo.

Uždaviniai

1. Pasirinkti pagrindinius Agda programavimo kalbos konstruktus
2. Rasti pasirinktų Agda konstruktų atitikmenis Go programavimo kalboje
3. Pasiūlyti galimą Agda duomenų struktūrų transliavimą į Go duomenų struktūras
4. Pasiūlyti galimą Agda funkcijų transliavimą į Go funkcijas

1. Priklausomi tipai

Priklausomi tipai yra tokie tipai, kurie priklauso nuo tipų reikšmių [BD08]. 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 [BD08].

1.1. Programavimo kalbos su priklausomais tipais

Šiuo metu yra sukurta nemažai programavimų kalbų, kurios palaiko priklausomus tipus, dalis jų, pavyzdžiui Coq [HKM13], 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 [Stu16], gali būti naudojamos ne tik kaip įrodymų asistentai, bet taip pat ir programinės įrangos kūrimui. Egzistuoja ir atvirščias variantas, kur kalbą yra įvardijama kaip bendrinės paskirties programavimo kalba, tačiau taip pat gali būti naudojama kaip įrodymų asistentas, pavyzdžui Idris [Sit18]. Šiame poskyryje bus trumpai apžvelgtos kelios iš aukčiau 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 [Lip11], taigi turi nemažai panašumų su Haskell, ypač sintaksėje bei tipuose, nors Idris turi labiau pažengusią tipų sistema [Sit18]. 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) [BDN09]. 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 sintaksė 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) [Nor07], kuri apjungia Martin-Löf loginę sistemą su tipų visatomis bei Coquand-Huet kuntrukcijų skaičiavimą [Zha92]. 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 gražina 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) bet dvigubo neiginio teoremos [DP20].

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 apsibrėžti 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 bandis 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). Natūrialiuosius skaičius būtų galima apirėžti 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 gražinti 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 gražina paskutinį 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 gražina paskutinį jo elementą. Kadangi vektorius tikrai yra ne tuščias, nėra privaloma tikrinti to varianto, kai jis neturi elementų. Ši funkcija priima sarašą, 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

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

Ši funkcija veiks taip pat, kaip ir aukščiau aprašyta funkcija, tačiau bandant perduoti sąraša su kitokio tipo elementais, bus gauta klaida.

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.

Pasinaudojamę 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 [WP14]. Vertas paminėta punktas, kad Go programavimo kalba šiuo metu nepalaiko bendrinių (*angl.* generic) tipų.

3.1. Sintaksė

Galime išskirti šiuos Go programavimo kalbos sintaksinius bruožus:

- Operacijas galima atskirti naujos eilutės simboliu arba kabliataškiu
- Identifikatorių vardai turi parasidėti raide arba apatiniu brūkšniu (_), po kurių gali sekti skaičiai, 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 reikšė 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.

- „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
}
```

Norint sukurti naują sukurtos struktūros tipo kintamąjį tai atliekama užrašius struktūros vardą ir riestiniuose skliaustuose nurodant kintamųjų reikšmes, pavyzdžiui:

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

Taip pat yra galimybė sukurtį 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
}
```

Kuriant naują metodą galime nurodyti apsirašytą sąsaja kaip to metodo priimamo argumento tipą, tuo atvėju 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, išnaudojant tokią sąsają galima gauti panašų veikimą į bendrinius sudėtinius tipus, bet apie tie bus išsiplėta vėlesniame skyriuje.

3.5. Funkcijos

Paminėti apie funkcijų exportinimą Kuriant Go funkcijas yra privaloma nurodyti kokio tipo argumentus ši funkcija priima, bei kokio tipo reikšmę gražina (jeigu gražina). Pasinaudodami aukščiau aprašyta struktūra bei Go masyvo tipu galime aprašyti alternatyvias funckijas aprašytas Agda skyriuje:

```
func head(elementai interface{}) interface{} {
   switch nt := elementai.(type) {
      case []zmogus:
```

```
        return nt[len(nt)-1]
      default:
        panic("Nezinomas tipas")
   }
}
func paskutinisZmogus(zmones []zmogus) zmogus {
   return zmones[len(zmones)-1]
}
```

Funckija, kuri priima ir gražina zmogus tipo struktūrą yra paprasta, tačiau kadangi Go kalboje nėra bendrinių tipų, negalime aprašyti šios funkcijos nenurodant konkretaus masyvo elementų tipo, nebent jos viduje išrašysime visus tipus, kuriuos ji priima. Taip pat, kadangi funkcija head gražina 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 naudojant bendrinį tipą priimančia funkciją, jos rezultatas iš karto gaunamas su tuo tipu, kuris buvo paduotas, tuo tarpu Go kalboje jis yra interface tipo, iš kurio jį reikia paversti atgal į norimą tipą.

3.6. Generics in go

aprašyti generics ir patį proposal

4. Susiję darbai

Šiuo metu Agda repositorijoje egzistuoja 2 transliatoriai:

- MAlonzo, kuris Agda programinį kodą paverčia į Haskell, iš kurio jis yra sukompiliuojamas naudojant Glasgov Haskell kompiliatorių (*angl.* Glasgov Haskell Compiler, GHC)
- JavaScript, kuris Agda programinį kodą paverčia į JavaScript kodą, šio transliatoriaus pagrindinė paskirtis yra internetinių programų kūrimas [Jef13]

Taip pat egzistuoja ir daugiau transliatorių, kurie nėra pridėti prie oficialios Agda repositorijos, tai tokie transliatoriai, kaip Utrecht bei Epic [GF11; HDS15]. 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 [GF11; HDS15]. Vienas minusas, kurį galima išskirti apie šį transliatoriu yra tai, kad norint, kad programa praeitų GHC kompiliatorių, transformuojant kodą MAlonzo posistemė tu panaudoti nesaugius prievartinius tipų vertimus (*angl.* unsafeCoersion, tai metodas Haskell kalboje) prieš kiekvieną funkcijos rezultatą bei argumentą, kas 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 kopiliatoriaus(*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 [HDS15].

Kaip ekperimentinis 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 repositorijoje. Agda kodą transliuoja į netipizuotą lambda skaičiavimą (*angl.* untyped lambda calculus) ir tada transformuoja į ECMASript (dar žinomą kaip JavaScript) [Jef13]. 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 [HDS15].

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. Šis transliavimo aprašymas apima keletą pagrindinių Agda kalbos konstruktų, kiti kalbos kontruktai turėtų būti įtraukti į ateities darbus.

5.1. Egzituojančių transliatorių panašumai

Apžvelgus šiuo metu esančius Agda kalbos tranliatorius 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 žingsius 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 [Jef13])

5.2. Agda programavimo kalbos tarpinė reprezentacija

Kompiliuojant agda failus jie praeina pro kelis žingsius, 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. Žingsiai kol Agda kodas yra paverčiamas kitos kalbos vykdomuoju failu [NAD+]

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 aprašyme duomenų tipas nėra kairėje konstruktoriaus tipo pusėje, tai padeda apsisaugoti nuo darbo nebaigiančių funkcijų.
  - 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 formas (*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.

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 atvėju „demo“ yra modulio pavadinimas)

5.3. Agda Duomenų tipų konvertavimas į Go

konvertavimas į structs ir interfaces. Struktūros kaip list/map naudoja interface, todėl casting reiktų po to

5.4. Agda funkcijų konvertavimas į Go

5.4.1. Basic konvertavimas

.

5.4.2. builtins

primitive int type

5.5. Agda įrašų konvertavimas į Go

todo iš kodo pusės, bet matyt turėtų pakankamai normaliai verstis į struct, labai panašus formatas

5.6. Įrodymai

Galbūt pridėt kažkur atskirai, bet verta paminėt, kad įrodymai yra trinami. TBD ar iš vis netransliojam, ar turbūt kuriam funkciją, kuri iš kart panic meta.

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 varda 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, jeigu yra poreikis jį vėliau kompiliuoti į veikiantį Go kodą.

Rezultatai

1. Parinkti pagrindiniai Agda programavimo kalbos konstruktai
2. Surasti pasirinktų Agda konstruktų atitikmenys Go programavimo kalboje
3. Pasiūlytas galimas Agda duomenų struktūros konvertavimas į Go programavimo kalbą
4. Pasiūlytas galimas Agda funkcijos konvertavimas į Go programavimo kalbą

Išvados

1. Darbe aprašytus Agda duomenų tipus bei funckijas galima transliuoti į Go sąsajas bei struktūras
2. Šiuo metu esantys Agda transliatoriai taikosi į funkcines arba labda skaičiavimo kalbas, taigi transliatorius į Go programavimo kalbą būtų pirmasis kompiliuojantis į kitokio tipo kalbą.

Literatūra

[BD08] A. Bove ir P. Dybjer. Dependent types at work. *LerNet ALFA Summer School*, 2008.

[BDN09] A. Bove, P. Dybjer ir U. Norell. A brief overview of agda – a functional language with dependent types, 2009.

[DP20] P. Dybjer ir E. Palmgren. Intuitionistic type theory. Metaphysics Research Lab, Stanford University, 2020.

[GF11] D. Gustafsson ir O. Fredriksson. A totally epic backend for agda. 2011.

[Hau15] P. Hausmann. The agda uhc backend. 2015.

[HDS15] P. Hausmann, A. Dijkstra ir W. Swierstra. The utrecht agda compiler. 2015.

[HKM13] G. Huet, G. Kahn ir C. P. Mohring. The coq proof assistant a tutorial, 2013.

[Jef13] A. Jeffrey. Dependently typed web client applications - frp in agda in html5. *PADL*, 2013.

[Lip11] M. Lipovača. Learn you a haskell for great good!, 2011.

[NAD+] U. Norell, A. Abel, N. A. Danielsson, M. Takeyama ir C. Coquand. Agda readthedocs puslapis. URL: `https://agda.readthedocs.io/en/v2.6.1.1/overview.html`.

[Nor07] U. Norell. Towards a practical programming language based on dependent type theory. 2007.

[Sit18] B. Sitnikovski. *Gentle Introduction to Dependent Types with Idris*. Amazon/KDP, 2018.

[Stu16] A. Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery, Morgan ir Claypool, 2016. ISBN: 9781970001273.

[WP14] E. Westrup ir F. Pettersson. Using the go programming language in practice, 2014.

[Zha92] L. Zhaohui. A unifying theory of dependent types: the schematic approach. *Logical Foundations of Computer Science*, p. 293–304. Springer Berlin Heidelberg, 1992.

1 priedas. Agda kodas paverstas į kitas kalbas

Agda sąrašo tipas ir funkcijos, kurios sudeda sąrašo elementus ir gražina 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")
  });
```