Трансформација петљи помоћу Кланга

Семинарски рад у оквиру курса
Конструкција компилатора
Математички факултет, Београд

Јелена Јеремић
mi16062@alas.math.rs

Марија Ерић
m17115@alas.math.rs

Лазар Васовић
mi16099@alas.math.rs

Дарко Нешковић
mi16208@alas.math.rs

23. јун 2020.

Сажетак

Размотрена је употреба Кланга као библиотеке у циљу трансформације свих петљи у *C* коду у жељени тип (*for*, *while*, *do-while*). На основу уводних разматрања имплементирана је апликација, у које сврхе је коришћен Клангов апликативни програмски интерфејс према апстрактном синтаксном стаблу. Саме измене вршене су у тексту кода. Описано је неколико изазова и проблема – како успут решених, тако и отворених – и сви су илустровани пратећим (тест) примерима.

Кључне речи — Кланг (*Clang*), *AST*, језик *C*, петље

Садржај

1 Увод

Кланг (*Clang*) представља предњи део компилатора, са задатком анализе и превођења *C*-оликих језика [1]. Један је од основних потпројеката преводилачке инфраструктуре *LLVM*, па се често спојено назива *Clang/LLVM*. Осим што се може користити као алат за превођење *C/C++/Objective-C* кода на међурепрезентацију, могуће је употребити га и као библиотеку, по чему се издваја међу конкурентима. Та особина следи из чињенице да пружа чист апликативни програмски интерфејс (*API*), тако да га је без проблема могуће у коду користити у другим пројектима. У овом раду је истражен Клангов *API* према апстрактном синтаксном стаблу (*AST*) на примеру језика *C*, и то конкретно на примеру трансформације петљи у жељени тип.

Петље су један од главних конструката виших императивних језика, настао са развојем структурне (пот)парадигме [2]. Њима је уведено понављање (итерација) као нови вид контроле тока извршавања програма. Данас су неизбежан елемент већине процедуралних језика вишег нивоа, па и нешто старијег језика *C*, који је овде размотрен. У њему постоје три типа петљи: *for*, *while*, *do-while* (*do*). Како је конкретан решавани проблем промена петљи из једног типа у други, постоји шест трансформација које су имплементиране. Детаљније информације о примењеном шаблону измена изложене су у телу рада.

Што се тиче самих *C*-овских петљи, главне компоненте сваке јесу услов (један израз типа који се може претворити у целобројни) и тело (једна наредба, али може бити и сложена). Тако се нпр. *for* и *while* извршавају тако што им се прво израчунава услов, а затим и тело уколико је услов испуњен (добијена је целобројна ненула вредност), и тако укруг, све док је услов испуњен или се на неки други начин не искочи из петље. Опционо, *for* омогућава да се пре прве итерације изврши иницијализација (једна наредба), а на крају сваке инкрементација (корак петље, исто једна наредба). Тип *do-while* по свему личи на *while*, осим по томе што му се у првој итерацији тело изврши без евалуације услова. Петље дозвољавају и два додатна механизма контроле тока унутар тела – наредбе *break* и *continue*. Прва искаче из петље, преусмеравајући контролу на излаз из ње, док друга прескаче текућу итерацију, преусмеравајући контролу на крај тела петље.

2 Апликација

У складу са уводним разматрањима проблема, написана је апликација која трансформише петље по корисниковој жељи. Пре превођења (билдовања) програма неопходно је инсталирати шесту верзију *LLVM*-а (*llvm-6.0-dev*) и Кланга (*libclang-6.0-dev*) за развијаоце апликација, као и алат за лепо форматирање кода (*clang-format*). Приложена датотека за аутоматску изградњу извршиве верзије заснована је на систему *CMake*, а за сам процес је дат и *build* директоријум.

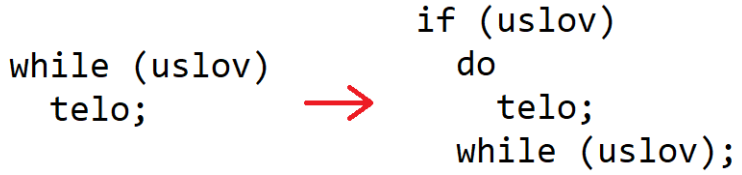
Како је већ напоменуто, три петље са могућношћу промене сваког типа у сваки други дају укупно шест могућих трансформација. Главни изазов тог посла јесте одржавање што веће семантичке (значењске) једнакости са петљом пре транформације. У овиру рада је заправо имплементирано пет трансформација, пошто је једна извршена посредно – зарад лакшег очувања семантике контроле тока, код промене *do* у *while* прво је свако *do* претворено у *for*, па тек онда свако *for* у *while*. Имплементирана је и помоћна трансформација која

припрема *for* петље за даљи рад. Целокупан ток дат је алгоритмом 1. Свака трансформација заснована је на особинама полазне и циљне петље, а имплементирана је прослеђивањем парсеру стабла посебно направљених класа које наслеђују Клангове класе *ASTConsumer* и *RecursiveASTVisitor*, редом задужене за обраду (конзумацију) и рекурзивни обилазак (посету) апстрактног синтаксног стабла. Одговарајуће текстуалне измене кода врши класа *Rewriter* – преписивач.

Алгоритам 1: Трансформација петљи

```
1  if промена на do then
2  |  замени свако while са do
3  else
4  |  замени свако do са for
5  end
6
7  if промена на for then
8  |  замени свако while са for
9  else
10 |  припреми свако for
11 end
12
13 if промена на while then
14 |  замени свако for са while
15 else if промена на do then
16 |  замени свако for са do
```

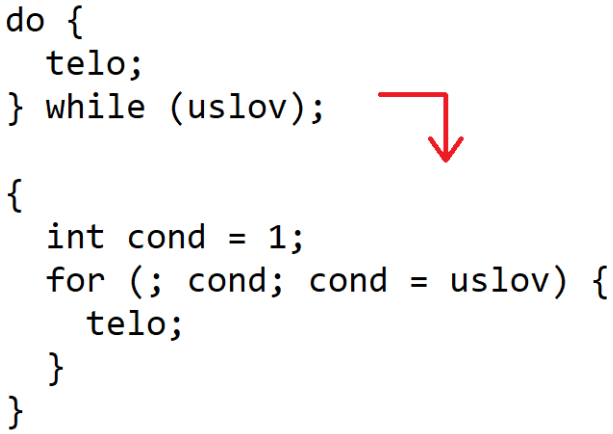
Идући редом, прва конкретна трансформација јесте замена сваког *while* са *do* (линија 2). Ово је одрађено релативно простим преусмеравањем аргумената полазне петље на циљну, уз додатак испитивања услова пре прве иначе безусловне итерације циљне петље. Примењени шаблон измене дат је сликом 1, са разматраном *C*-овском синтаксом.



Слика 1: Замена сваког *while* са *do*

Следећа је замена сваког *do* са *for* (линија 4). Ово је најкомплекснија трансформација, која укључује увођење помоћне променљиве за складиштење евалуираниог услова. На почетку, он је тривијално испуњен, тако да је варијаблина почетна вредност целобројна јединица, док се за корак петље узима израчунавање услова и његова додела. Иницијализација је празна. Схема трансформације дата је сликом 2.

Наредна је замена сваког *while* са *for* (линија 8). Ово је најтривијалнија трансформација, пошто подразумева само преусмеравање



Слика 2: Замена сваког *do* са *for*

тела и услова, док иницијализација и корак циљне петље остају празни. Шаблон ове измене илустрован је сликом 3.

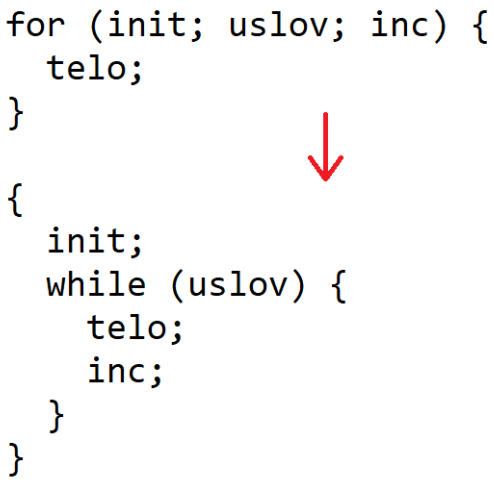
Слика 3: Замена сваког *while* са *for*

Разлог припреме *for* петљи пре даљег рада (линија 10) изложен је у поглављу посвећеном проблемима. Засад је довољно напоменути да је ово урађено тако што је испред сваког *continue* чији је први предак *for* додат корак петље, наравно, уколико петља уопште има корак.

Следи замена сваког *for* са *while* (линија 14). Ова трансформација своди се на преусмеравање тела и услова, уз додатак да се пре циљне петље мора извршити иницијализација полазне, уколико она постоји, док се на крају тела циљне петље евалуира корак полазне, опет уколико постоји. Схема примењене измене дата је сликом 4.

Напослетку преостаје замена сваког *for* са *do* (линија 16). Она је идентична композицији замене *for* са *while*, а затим *while* са *do*, али је засебно имплементирана зарад бољих перформанси (један пролаз уместо два) и бољег уклапања у изложени општи алгоритам. Због те еквивалентности, ипак, за њу није приложена илустрација.

Апликација као аргументе командне линије прима улазну и излазну дадотеку, као и жељени тип петље, који учествује у условима алгоритма. Приликом сваке описане трансформације изнова се формира и обрађује *AST*, док се текстуалне измене кода инкрементално очитавају у излазној датотеци. На крају се лепо форматира нови код, пошто се приликом измена не брине о конзистентном увлачењу линија.



Слика 4: Замена сваког *for* са *while*

3 Проблеми

Упркос великој робусности Кланговог *AST*-а и ширини интерфејса према њему, постоји неколико проблема на које се наишло приликом рада. Неки су настали због ограничења самог *API*-ја, а неки су општи изазови који неминовно следе из рада са (*C*-овским) петљама, односно из посебности њихове семантике. Неки су успешно решени, док неки нису, али су сви издвојени и објашњени пратећим (тест) примерима.

Један од главних јесте имутабилност (непроменљивост) генерисаног *AST*-а, као ограничење *API*-ја. Наиме, чворове Кланговог синтаксног стабла није могуће мењати нити превезивати. Један од разлога за то је чињеница да Кланг не испоручује сирово стабло генерисано у току синтаксне анализе, већ успут врши и семантичку анализу, те генерише цео тзв. *AST* контекст, попуњен подацима о парсираном програму. Анотирано стабло, сем нпр. информација о типовима, садржи и ситне детаље попут линије и индекса почетног и крајњег карактера у коду где је оно што чвор представља реализовано. Овакви подаци би ручном манипулацијом изгубили смисао, те би било неопходно још једном испитати значење. Директна последица овога је да није могуће лако манипулисати угнежђеним петљама. У идеалном случају, било би могуће изменити прво унутрашњу петљу, па одмах затим спољашњу, у једном пролазу нагоре, или обрнуто у једном пролазу надоле. Ипак, како измена стабла није дозвољена, а чворови не виде текстуалне измене, то овде није могуће, али је решено увођењем вишепролазности. Функција обраде заправо је бесконачна петља која у једном пролазу обради само највише спољашње чворове и не иде дубље. Петља се прекида када више нема измена, а гаранцију заустављања пружа чињеница да се хијерархија угнежђених петљи у свакој итерацији смањује. Аналоган проблем је код припреме *for* петљи.

Следећи важан проблем јесте контрола тока специфична за петље. Како је већ наведено, у питању су наредбе *break* и *continue*, од којих прва искаче из петље, преусмеравајући контролу на излаз из ње, док

друга прескаче текућу итерацију, преусмеравајући контролу на крај тела петље. Оне су један од разлога зашто неколико наивних идеја за трансформације – нпр. она према којој је *do* петљу могуће претворити у друге тако што се тело избаци испред петље – није прихватљиво решење. Ипак, уколико тело приликом сваке измене остаје на свом месту, онда *break* не представља проблем, јер на исти начин искаче из петље независно од њеног типа. С друге стране, *continue* има проширено значење у случају *for* петљи, пошто пре преласка на следећу итерацију изврши корак петље, уколико он постоји. Прецизније, значење је и даље исто – преусмерење контроле на крај тела петље – али је додатак што се иза тела налази корак петље пре скока на почетак и проверу услова. Ипак, како ова финеса није видљива преко *C*-овског интерфејса, у раду се сматра да је значење проширено. Проблем неједнаког значења *continue* у *for* и осталим типовима решен је реализацијом припремног корака који испред сваког *continue* чији је први предак *for* додаје корак петље, наравно, уколико петља уопште има корак. Ово можда није најбоље решење, пошто након овог корака ниједна нетрансформисана *for* петља није једнака изворној, али се највише уклапа у претходно разматрану вишепролазност и не представља проблем уколико апликација успешно заврши са радом.

Пример ситног проблема са *API*-јем јесте то што се празна иницијализација или корак код *for* петљи не сматрају празним (нултим) наредбама, већ недостајућим вредностима, па је неопходно пазити на нулте показиваче (*nullptr*). Ово се, додуше, може схватити и као предност, пошто омогућава да нпр. прелазак са *for* на *while* не буде компликован као на слици 4, већ врло једноставан, као инверз слике 3. Ово упрошћавање је и примењено, као у тест примеру... Када је у питању упрошћавање, могло би се помислити и да је трансформацију *do* у *for* са слике 2 лако сажети убацивањем декларације са иницијализацијом условне променљиве у део за иницијализацију петље. То у неку руку јесте истина, али овде наменски није урађено, јер резултујући програм не би био у складу са *ANSI C* стандардом, који многи програми прате, па би таквом изменом били покварени.

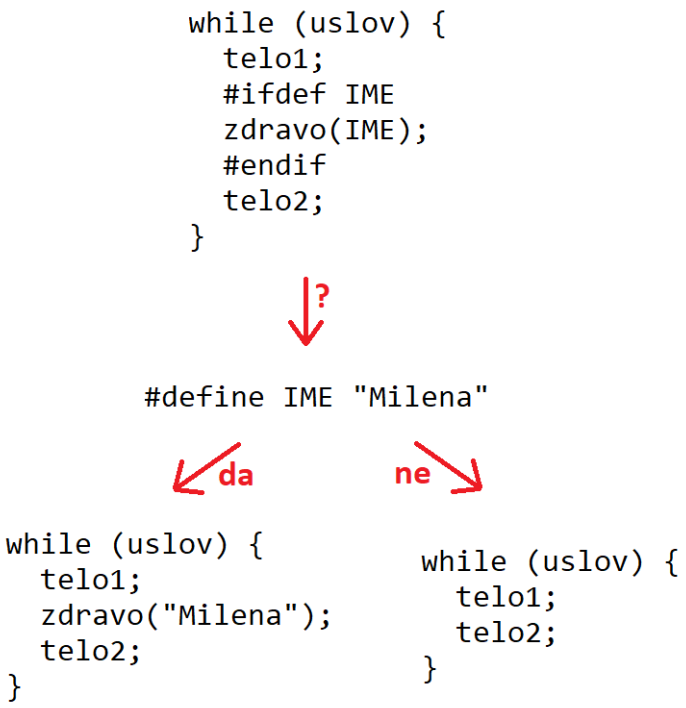
Један ситан семантички проблем који је успешно решен јесте проблем досега иницијализационе променљиве у *for* петљама. Иако се при решавању претходно описаног проблема пазило да све лепо ради са *ANSI C* стандардом, подржани су и *C99* и новији као улаз, тако да је било неопходно обрадити и тај случај, када је пример обрнут. Наиме, иницијализациона променљива има досег који је у хијерархији између блоковског досега саме петље и досега блока изван петље. Да је нпр. при трансформацији *for* у *while* са слике 4 иницијализација просто стављена испред петље, без обазирања на околину, нашла би се у спољашњем досегу. Тиме би потенцијално изазвала конфликт са већ постојећим објектом истог имена. Из тог разлога је направљен нови блок који обухвата иницијализацију и петљу, чиме се одржава полазна семантика, иако на први поглед можда не изгледа смислено. Сличан овоме је проблем са именима при увођењу помоћне условне променљиве, као на слици 2, при чему је неопходно пазити да се не узме неко већ заузето име, било да је глобално или важно локално.

Нешто већи проблем са досегом праве *for* петље које у телу маскирају (енгл. *shadowing*) имена која се користе у кораку. Наиме, у том случају простом припремом у виду додавања корака испред сваког *continue* не постиже се жељени ефекат. Испољавање проблема дато је у тест примеру... Ово, међутим, није могуће решити локално, јер зама-

скирана имена у језику *C* није могуће одмаскирати. Пример решења јесте одустајање од претходно објашњене апроксимације да *continue* има проширено значење у случају *for* петљи и прихватање чињенице да је увек у питању преусмерење контроле на крај тела петље. Тако би се нпр. за претварање *for* у *while* уместо схеме са слике 4 могао користити шаблон са слике 5. То ипак у овом раду није урађено, пошто се безусловним скоковима спушта на ниво који није примерен структурном програмирању, чијег су петље основица.

Слика 5: Замена сваког *for* са *while*

Напослетку, једини примећен нерешени проблем, тачније скуп проблема који нису решени, чине све грешке које настају при раду са претпроцесором. Наиме, језик *C* има веома развијену и често коришћену инфраструктуру која претходи самом процесу превођења. Мада је основна улога претпроцесирања укључивање декларација из заглавља, распрострањена је употреба макроа и великог броја других претпроцесорских директива. Постоји и велики број условних директива, као и неких које значајно утичу на даљи процес превођења (нпр. прагме), чиме претпроцесор подсећа на преводилац у малом. Предњи део компилатора види само резултате претпроцесирања, тако да није лако реконструисати шта је тачно било у изворном коду. За ову проблематику нису формирани тест примери, али је илустрација дата кроз слику 6. Са ње се види да један исечак кода може дати два резултата након претпроцесирања, без икаквих назнака шта је заправо ту писало. Овде спада и проблем са коментарима, које претпроцесор једноставно уклања и није им лако могуће приступити преко *API*-ја за *AST*. Они стога нису укључени у излаз апликације из рада.



Слика 6: Двоструки проблем претпроцесирања – да ли је име дефинисано и која му је вредност ако јесте

4 Закључак

Кланг се, као део преводилачке инфраструктуре *LLVM*, издваја од конкурената по томе што пружа чист апликативни програмски интерфејс (*API*) према апстрактном синтаксном стаблу (*AST*). Управо тај *API* истражен је у овом раду, на примеру језика *C*, и то конкретно на примеру трансформације петљи у жељени тип. За те потребе написана је апликација која умногоме успешно обавља замишљени посао, инкрементално мењајући текст кода све док не остане само жељени тип петље. У разматрању, као и при самом раду на програму, јавило се неколико углавном ситних проблема. Већина је успешно решена, са главним изузетком када је у питању руковање претпроцесорским директивама, на шта би се могло усмерити даље истраживање.

Литература

[1] Clang: a C language family frontend for LLVM. веб-сајт посвећен пројекту и документација, доступно на: https://clang.llvm.org/.

[2] Dexter Kozen and Wei-Lung Tseng. The Böhm–Jacopini Theorem Is False, Propositionally. pages 177–192, 07 2008. доступно на: https://www.researchgate.net/publication/225114059_The_Bohm-Jacopini_Theorem_Is_False_Propositionally.