XXXV Летняя школа-конференция по параллельному программированию 2020

Разработка архитектуры компилятора LuNA

Выполнил: Ижицкий Р. Л.
Руководитель: Перепелкин В. А.
17.07.2020

LuNA

Проект “Технология фрагментированного программирования”

- Система фрагментированного программирования LuNA и язык LuNA
- Программа:
  - множество переменных (фрагменты данных)
  - множество операций (фрагменты вычислений)
  - информационные зависимости

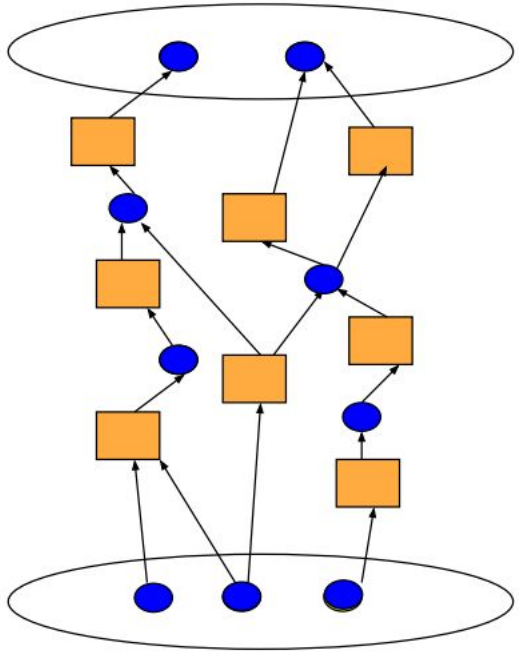
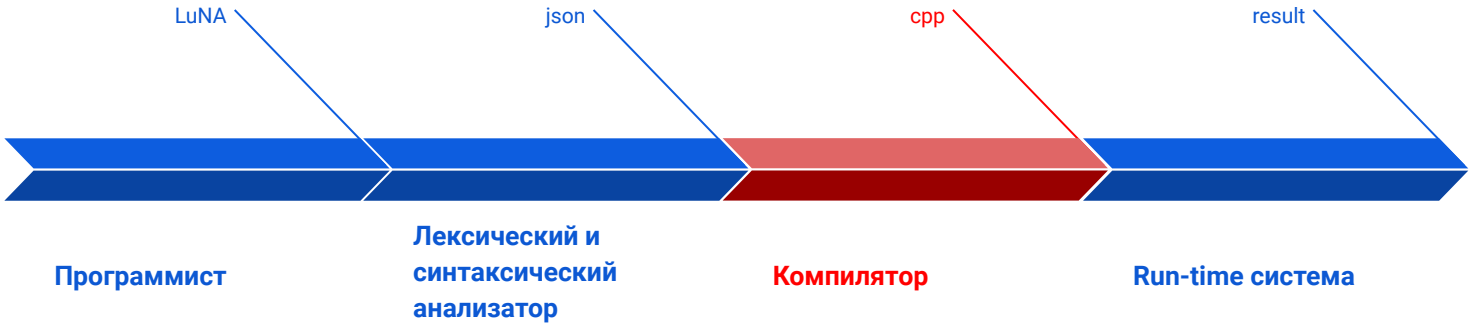


Схема работы



Проблема

На основе полученного опыта в предыдущих реализациях компилятора требуются:

- доработка его архитектуры
- рефакторинг кода

Цель и задачи

Цель: произвести рефакторинг компилятора LuNA

Задачи:

- Разобрать входное и выходное представления компилятора
- Разобрать архитектуру компилятора и алгоритмы компиляции
- Разработать новую архитектуру компилятора

Входное представление

- Было изучено внутреннее входное представление языка
- Была дополнена документация

Ссылка:
https://gitlab.ssd.sscc.ru/luna/lo3/wikis/recommendations

Все рекомендации записываются в следующем синтаксисе:
`sub sub_name() { code } @ { recommendation1; r2; ... }`
или `cf_name() @ { recommendation; r2; ... };`

Все рекомендации могут быть одного из следующих видов:

- **locator_cyclic**
  `locator_cyclic x[i] => i;`
  разместить фрагмент данных `x[i]` на узле номер `i` в виртуальной топологии (в реальной топологии этому узлу будет соответствовать узел номер i % (NODE_NUMBER)
- **request**
  `request N; request x[N];`
  запросить фрагмент данных (т.е. копию с узла, где имеется этот фрагмент данных) `N`, затем `x[N]` перед выполнением соответствующего фрагмента вычисления
  каждое исполнение `request` (а не использование фрагмента данных) увеличивает счетчик соответствующего фрагмента данных на 1 (при достижении `req_count` фрагмент будет удален)
- **req_count**
  `req_count A[i][j]=N;`
  после `N` запросов (может не совпадать с количеством использований фрагмента данных) `request` фрагмент данных `A[i][j]` нужно удалить (т.е. он будет запрошен через `request` максимально столько раз)
- **delete**
  `delete C[i][j];`
  удалить фрагмент данных `C[i][j]` после выполнения соответствующего фрагмента вычисления
  также этот оператор может быть записан в виде `cf(args) --> (C[i][j])`
- **-->**
  см. `delete`
- **>>**
  `empty() >> (N[it]);`
  после выполнения фрагмента вычисления, указанного до оператора, установить значение 1 в фрагмент данных, указанного после оператора (т.е. после выполнения `empty()` сработает `N[it]=1`), это соответствует информационной зависимости между `empty()` и `N[it]`

Выходное представление

- Было изучено внутреннее выходное представление языка
- Была дополнена документация

Ссылка:
https://gitlab.ssd.sscc.ru/luna/lo3/wikis/выходное-представление-компилятора

ФВ/CF - фрагмент вычисления/control fragment (операция)
ФД/DF - фрагмент данных/data fragment (переменная)

- **int NextBlock** - ФВ (с соответствующим номером блока), который будет запущен следующим после текущего

- **ID create_id()**
  создать новый уникальный идентификатор для ФД или ФВ
- **ID id(3)**
  получить уникальный идентификатор для четвертого (3 + 1) фрагмента этого ФВ

- **DF arg(3)**
  доступ к четвертому аргументу текущего ФВ
- **DF argv(ValueType, 3)**
  получить ФД, переданный четвертым аргументом в sub main

- **CF fork(8)**
  создать новый ФВ, соответствующий блоку 8
- **bool migrate(*loc*)**
  возвращает true, если текущий ФВ находится не на узле, соответствующему локатору loc;
  пример использования: `if (self.migrate(CyclicLocator(0))) { return MIGRATE; }`
- **void store(*id*, DF x)**
  присвоить фрагменту, соответствующему *id*, значение x (в текущий ФВ)
- **DF wait(*id*)**
  получить в данном ФВ фрагмент данных, соответствующий *id*, если он доступен;
  пример использования: `if (self.wait(self.id(0)).is_unset()) { return WAIT; }`
- **void destroy(*id*, *loc*)**
  сделать запрос на узел, соответствующий локатору *loc*, на удаление ФД, соответствующего *id*

- **void post(*id*, DF x, *loc*, *req_count*)**
  ФД x отправить на узел, соответствующему локатору loc, с использованием *Id*;
  после *req_count* запросов request ФД станет недоступным (если в качестве *req_count* указать -1, то будет доступно неограниченное количество запросов)
- **void request(*id*, *loc*)**

Преобразования компилятора

- Было изучено внутреннее представление языка
- Была дополнена документация

Ссылка:
https://gitlab.ssd.sscc.ru/luna/lo3/wikis/преобразование-компилятора

LuNA	C++
`@ { locator_cyclic: 2; };`	`if (self.migrate(CyclicLocator(2))) { return MIGRATE; }`
`@ { req_count x=2; };`	`DF posted = self.wait(x_id);` `self.post(x_id, posted, loc, 2);`
`@ { req_unlimited; };`	`DF posted = self.wait(x_id);` `self.post(x_id, posted, loc, -1);`
`@ { delete x[8]; locator_cyclic x[i] => i+1; };`	`self.destroy(x_id[8], CyclicLocator(static_cast<int>((8)+(1))));`

`@ { request N; };`	BLOCK1:
	`self.request(n_id, loc);`
	`self.NextBlock=2;`
	BLOCK2:
	`if (self.wait(n_id).is_unset()) { return WAIT; }`
	`self.NextBlock=3;`

`@ { request N; request x[N]; };`	request N -> wait N -> request x[N] -> wait x[N]
`@ { afterpush x[10] => a; };`	`self.push(x_id[10], self.wait(x_id[10]), a_id, a_loc);`

`cf a : ... @ { wait x[10]; };`	BLOCK1:
	`self.expect_pushes(a_id);`
	`self.NextBlock=2;`
	BLOCK2:
	`if (self.wait(x_id[10]).is_unset()) { return WAIT; }`
	`self.NextBlock=3;`

`sub main(string arg1)`	`self.arg(0) = self.argv(TYPE_STRING, 0);`
`for i=1..5 hello_world();`	`for (int lc_i=1; lc_i<=5; lc_i++) { CF *child=self.fork(3); }`

Алгоритмы преобразования

- Разобраны алгоритмы преобразования в компиляторе
  - Рекурсивный обход
  - Проброс переменной через вложенные пространства имен
- Разработаны отдельные модули компилятора

```
sub main() {
    df z, y, x;
        for i=1..100 {
                print(y);
        }
}
```

```
main->for:
        CF *child=self.fork(3);
        child->id(0)=self.id(1);

for->print:
        CF *child=self.fork(6);
        child->arg(0)=(self.wait(self.id(0))).get_int();
```

Требования

- На основе проведенных исследований сформировалось промежуточное видение новой архитектуры компилятора
  - Использование объектов вместо словарей
  - Другое разбиение на блоки
  - Несколько проходов по коду
  - Обработка рекомендаций в том же компиляторе

Заключение

- Разобрано входное и выходное представления компилятора
- Созданы вики-страницы с разбором выходного представления и соответствием входного и выходного представлений компилятора
- Разработаны промежуточные требования к архитектуре компилятора

Спасибо за внимание