LLVM

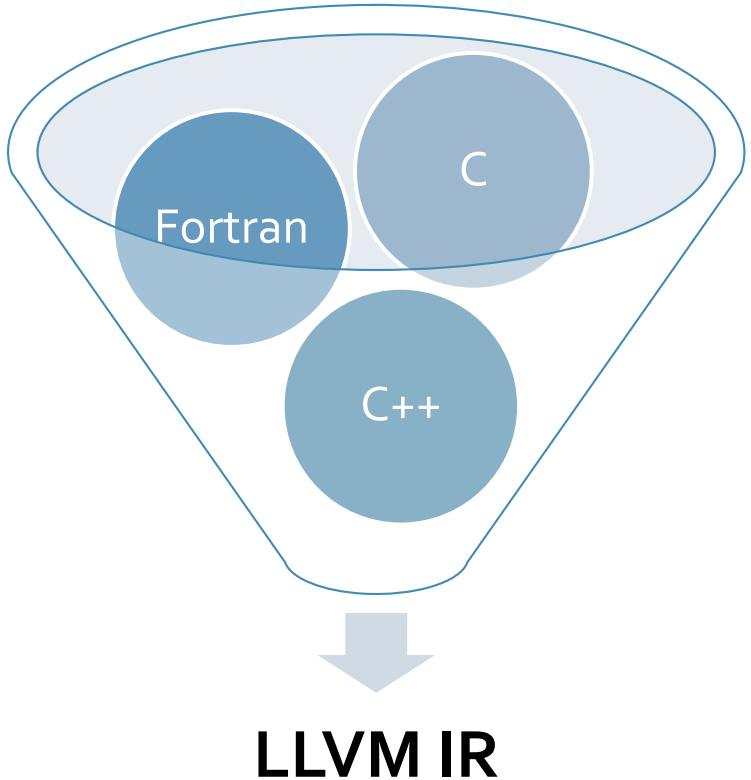
Нетривиальные приложения ООП и шаблонов, итераторов и контейнеров на примере большого проекта

К. Владимиров, Syntacore, 2023
mail-to: konstantin.vladimirov@gmail.com

- Основы LLVM IR и экосистема
- Генерация IR
- RTTI в стиле LLVM
- Оптимизации и опять ParaCL

Разработка и предназначение

- Лицензионная дружественность
- Модульность
- Текстовая читаемость
- Сильная типизация
- IR это виртуальный набор инструкций, подходящий для всех языков
- Более низкие уровни IR играют ту же роль для архитектур



The Design of LLVM

Основные термины LLVM IR

```
define i32 @fib(i32) {
entry:
  %1 = icmp ult i32 %0, 2
  br i1 %1, label %final, label %st

st: ; main recursion entry
  %2 = add i32 %0, -1
  %3 = call i32 @fib(i32 %2)
  %4 = add i32 %0, -2
  %5 = call i32 @fib(i32 %4)
  %6 = add i32 %3, %5
  br label %final

final:
  %7 = phi i32 [%6, %st], [1, %entry]
  ret i32 %7
}
```

- Ключевые слова
- Глобальные символы
- Локальные символы
- Типы
- Метки
- Базовые блоки
- phi-узлы
- Комментарии
- Инструкции

https://godbolt.org/z/ef67bsYTE

Основные термины LLVM IR

```
define i32 @fib(i32) {
; 1:
  %2 = icmp ult i32 %0, 2
  br i1 %2, label %9, label %3

; 3:
  %4 = add i32 %0, -1
  %5 = call i32 @fib(i32 %4)
  %6 = add i32 %0, -2
  %7 = call i32 @fib(i32 %6)
  %8 = add i32 %5, %7
  br label %9                    ; terminator

; 9:
  %10 = phi i32 [%8, %3], [1, %1]
  ret i32 %10
}
```

- Ключевые слова
- Глобальные символы
- Локальные символы
- Типы
- Метки (могут быть опущены)
- Базовые блоки
- phi-узлы
- Комментарии
- Инструкции

https://godbolt.org/z/f11esrYdz

LLVM IR это SSA представление

- Обычное представление

```
x = foo();
y = x;
x = bar();
y = x + y;
```

- SSA

```
x.0 = foo();
y.0 = x.0;
x.1 = bar();
y.1 = x.1 + y.0;
```

- Конечно тут есть проблема. Как представить схождения управления?

- Обычное представление

```
x = foo();
if (x > 5) x = x + 1;
x = x + 2;
```

- SSA

```
x.0 = foo();
if (x.0 > 5) x.1 = x.0 + 1;
x.2 = ? + 2;
```

Явные phi-узлы

- Обычное представление

```
x = foo();
y = x;
x = bar();
y = x + y;
```

- SSA

```
x.0 = foo();
y.0 = x.0;
x.1 = bar();
y.1 = x.1 + y.0;
```

- Конечно тут есть проблема. Как представить схождения управления?

- Обычное представление

```
x = foo();
if (x > 5) x = x + 1;
x = x + 2;
```

- SSA

```
x.0 = foo();
if (x.0 > 5) x.1 = x.0 + 1;
x.2 = phi(x.0, x.1) + 2;
```

Основные термины LLVM IR

```
define i32 @fib(i32) {
; 1:
  %2 = icmp ult i32 %0, 2
  br i1 %2, label %9, label %3

; 3:
  %4 = add i32 %0, -1
  %5 = call i32 @fib(i32 %4)
  %6 = add i32 %0, -2
  %7 = call i32 @fib(i32 %6)
  %8 = add i32 %5, %7
  br label %9                    ; terminator

; 9:
  %10 = phi i32 [%8, %3], [1, %1]
  ret i32 %10
}
```

- Ключевые слова
- Глобальные символы
- Локальные символы
- Типы
- Метки (могут быть опущены)
- Базовые блоки
- phi-узлы
- Комментарии
- Инструкции

https://godbolt.org/z/7KWv3c1E9

Инструкции

- Базовый LLVM IR содержит фиксированное количество платформенно-независимых инструкций

```
<result> = add <ty> <op1>, <op2>
<result> = icmp <cond> <ty> <op1>, <op2>
<result> = phi <ty> [ <val0>, <label0>], ...
br i1 <cond>, label <iftrue>, label <iffalse>
br label <dest>
```

- Добавление новой инструкции крайне болезненно и меняет биткод

https://llvm.org/docs/LangRef.html

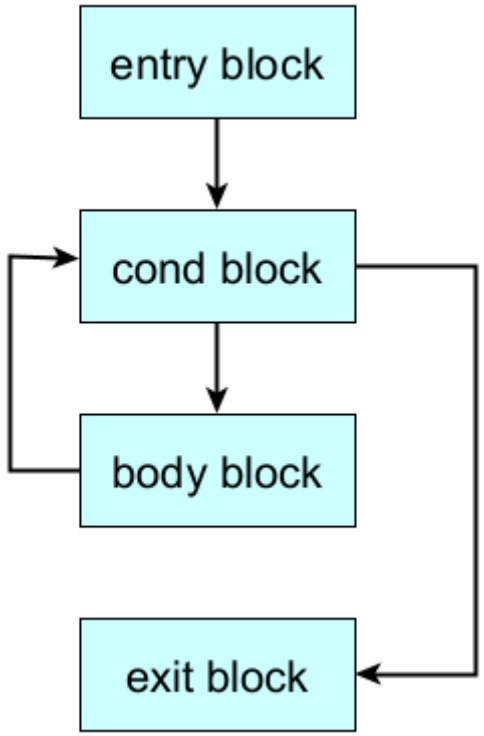
Типы

- Пустой тип: `void`
- Скалярные типы: `i1`, `i8`, `i16`, …, `half`, `float`, `double`
- Векторные типы: `<10 x i32>`
- Указатели: `i32*`, `i32 addrspace(5)*`
- Массивы: `[10 x i32]`, `[12 x [10 x float]]`
- Структуры: `{i32, i32, float, i8}`
- Функции: `i32 (i32, i32)`

https://llvm.org/docs/LangRef.html

Возвращаемся к числам Фибоначчи

```
@fibarr = global [10 x i32] zeroinitializer

.....
  ; fibarr[0] = 0; fibarr[1] = 1;
  br label %for.cond

for.cond:
  %i.0 = phi i64 [2, %entry], [%inc, %for.body]
  %cmp = icmp ult i64 %i.0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:
  ; fibarr[i] = fibarr[i - 1] + fibarr[i - 2]
  %inc = add i64 %i.0, 1
  br label %for.cond

for.end:
```



Обсуждение

- Теперь нам надо построить заполнение массива.
- Наличие в языке таких типов как "указатель" предполагает некую модель памяти.
- Разумно ли вносить на уровень общего для всех IR такие тонкости как alignment, padding, etc?

Симметрия в store & load

- Все операции с памятью происходят по указателю.
- Чтобы загрузить значение, мы указываем его слева.

```
%1 = load i32, i32* %idx2
```

- Чтобы сохранить значение, мы указываем его справа.

```
store i32 %1, i32* %idx2
```

- Откуда приходит указатель?
- Для начала: очень часто он уже есть.

Сослаться на fibarr

- Глобальная переменная это всегда указатель

```
@gv = global i8 0 ;  i8 const *
@gvc = constant i8 42 ; const i8 const *
```

- Это означает, что она может использоваться в load/store напрямую

```
%1 = load i32, i32* gvc
store i32 %1, i32* gv
```

- Но в примере выше было написано

```
@fibarr = global [10 x i32] zeroinitializer
```

- Как достать указатель на его нулевой и первый элементы?

GEP: униформность доступа

- Идея `getelementptr` – одинаковый доступ к массивам и структурам

```
<result> = getelementptr <ty>, <ty>* <ptrval> {, <ty> <idx>}*
```

- Здесь каждый индекс снимает один уровень косвенности

```
; мы хотим написать: fibarr[1] = 1
%fst = i32* getelementptr [10 x i32],
                          [10 x i32]* @fibarr, i64 1 ; FAIL
store i32 1, %fst
```

GEP: униформность доступа

- Идея `getelementptr` – одинаковый доступ к массивам и структурам

```
<result> = getelementptr <ty>, <ty>* <ptrval> {, <ty> <idx>}*
```

- Здесь каждый индекс снимает один уровень косвенности

```
; мы хотим написать: fibarr[1] = 1
%fst = i32* getelementptr [10 x i32],
                          [10 x i32]* @fibarr, i64 0, i64 1 ; OK
store i32 1, %fst
```

GEP для структуры

- Смоделируем своего рода структуру

```
; struct S { int x; double y; float z[10]; };
%struct.S = type { i32, double, [10 x float] }

@x = %struct.S zeroinitializer

define i32 @main() {
  %eltptr = getelementptr %struct.S,
                          %struct.S* @x, i32 0, i32 2, i64 3
  store float 1.000000e+00, float* %eltptr
```

К какому элементу идёт обращение через eltptr?

Пишем голову цикла

```
@fibarr = global [10 x i32] zeroinitializer

define void @fill() {
entry:
  %f0 = getelementptr ([10 x i32], [10 x i32]* @fibarr, i64 0, i64 0)
  %f1 = getelementptr ([10 x i32], [10 x i32]* @fibarr, i64 0, i64 1)
  store i32 1, i32* %f0
  store i32 1, i32* %f1
  br label %for.cond

for.cond:
  %i.0 = phi i64 [2, %entry], [%inc, %for.body]
  %cmp = icmp ult i64 %i.0, 10
  br i1 %cmp, label %for.body, label %for.end
....
```

Как написать тело цикла?

```
....

for.cond:
  %i.0 = phi i64 [2, %entry], [%inc, %for.body]
  %cmp = icmp ult i64 %i.0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:
  ; fibarr[i] = fibarr[i - 1] + fibarr[i - 2]
  %inc = add i64 %i.0, 1
  br label %for.cond

for.end:
....
```


https://godbolt.org/z/q4PqEq5z4

Искомое тело цикла

```
; fibarr[i] = fibarr[i - 1] + fibarr[i - 2]
for.body:
  %sub1 = sub i64 %i.0, 1
  %idx1 = getelementptr [10 x i32], [10 x i32]* @fibarr, i64 0, i64 %sub1
  %0 = load i32, i32* %idx1

  %sub2 = sub i64 %i.0, 2
  %idx2 = getelementptr [10 x i32], [10 x i32]* @fibarr, i64 0, i64 %sub2
  %1 = load i32, i32* %idx2

  %add = add i32 %0, %1
  %idx3 = getelementptr [10 x i32], [10 x i32]* @fibarr, i64 0, i64 %i.0
  store i32 %add, i32* %idx3

  %inc = add i64 %i.0, 1
  br label %for.cond
```


https://godbolt.org/z/GYeb49GW7

Обсуждение

- И для структуры и для массива у нас был gep с первым индексом 0

```
// auto arrayfirst = &fibarr[1]
%arrayfirst =
    i32* getelementptr [10 x i32],
                       [10 x i32]* @fibarr, i64 0, i64 1

// auto eltptr = &S.field3[3]
%eltptr = getelementptr %struct.S,
                        %struct.S* @x, i32 0, i32 2, i64 3
```

- Можно ли придумать когда он будет не ноль?

Обсуждение

- В LLVM IR нет объектов в том смысле, в каком они есть в C.
- Есть SSA-values, которые не перезаписываются.
- Есть локации в "памяти", которые униформны и доступны только по указателю.
- Сложно ли материализовать SSA value?.
- Сложно ли поднять семантическую сеть операций с памятью в SSA?

Фибоначчи без phi: локальные allocas

```
define dso_local i32 @fib(i32) #0 {

  %2 = alloca i32, align 4 ; слоты для материализации
  %3 = alloca i32, align 4 ; SSA значений
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i32, align 4

  store i32 %0, i32* %2, align 4
  store i32 0, i32* %3, align 4
  store i32 1, i32* %4, align 4
  store i32 0, i32* %5, align 4
  br label %7
```

Фибоначчи без phi: локальные allocas

```
store i32 %0, i32* %2, align 4
store i32 0, i32* %3, align 4
store i32 1, i32* %4, align 4
store i32 0, i32* %5, align 4
br label %compare

compare: ; preds = %body, %entry

%8 = load i32, i32* %5, align 4
%9 = load i32, i32* %2, align 4
%10 = icmp ult i32 %8, %9
br i1 %10, label %body,
           label %exit
```

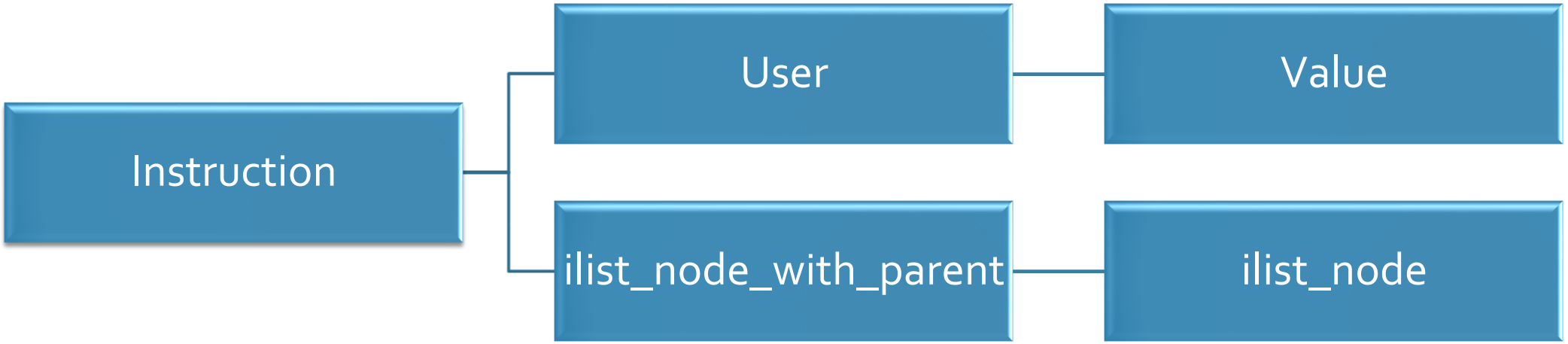
```
body: ; preds: %compare

%12 = load i32, i32* %3, align 4
store i32 %12, i32* %6, align 4

%13 = load i32, i32* %4, align 4
store i32 %13, i32* %3, align 4

%14 = load i32, i32* %6, align 4
%15 = load i32, i32* %4, align 4
%16 = add nsw i32 %15, %14

store i32 %16, i32* %4, align 4
br label %compare

exit: ; preds: %body
```

Обсуждение

- Понятно, что классы всех инструкций (`add`, `sub`, `gep`, `phi`, …) наследуются от базового класса `Instruction`.
- Как бы вы спроектировали этот класс?

Представление инструкции

```
class Instruction : public User,
          public ilist_node_with_parent<Instruction, BasicBlock>
```



Идеология User / Value

```
%1 = add i64 %0, 1  ; value
%2 = add i64 %1, %1 ; user / value
%3 = add i64 %1, %2 ; user
```

- Инструкция, которая порождает Value это и есть Value (по SSA).
- Value знает обо всех своих Users (`Value::use_iterator`).
- User знает о других своих операндах (`User::op_iterator`).

`User::getOperand(i)` вернёт `Value*`

ilist: интрузивные списки

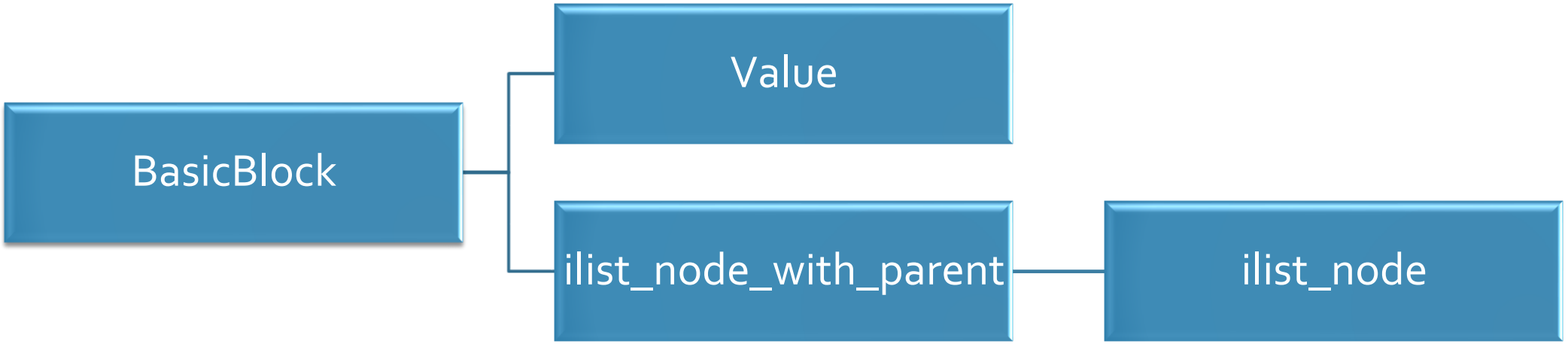
- Вообще-то в идеологии C++ большинство стандартных контейнеров неинтрузивны, то есть объект помещённый в контейнер не знает помещён он в контейнер или нет.
- Увы, на идеологию LLVM лучше легли интрузивные списки. В таком списке сам элемент списка предоставляет механизмы prev/next.
- Поскольку каждая инструкция является узлом интрузивного списка, она должна наследовать от ilist_node.
- Благодаря этому допустимо делать вот так:

```
Instruction *subInst = addInst->getNextNode();
```


http://llvm.org/docs/ProgrammersManual.html

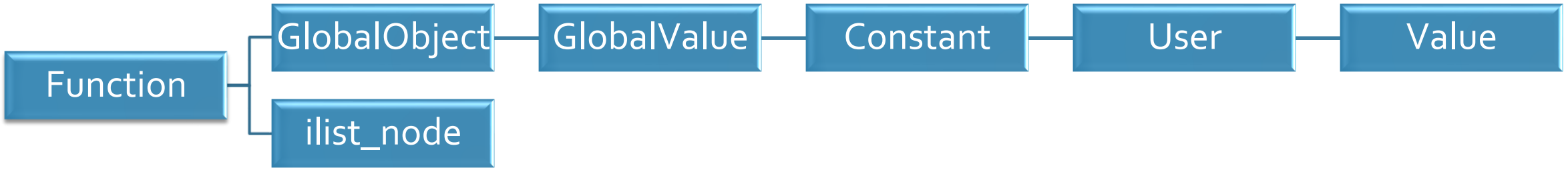
Представление базового блока

```
class BasicBlock : public Value,
          public ilist_node_with_parent<BasicBlock, Function>
```



Представление функции

```
class Function : public GlobalObject,
                 public ilist_node<Function>
```



Функции в LLVM IR

- Чтобы использовать функцию, её следует объявить.

```
declare i8* @malloc(i32) ; вероятно это malloc из C standard library
```

- Далее она используется через `call`.
- Также через `call` заведён инлайн-ассемблер.

```
call void asm sideeffect "mov ax, bx", ""()
```

- Также через `call` можно делать индиректные вызовы.

```
%result = call i64 %binop(i64 %x, i64 %y)
```

- Основы LLVM IR и экосистема
  - Генерация IR
- RTTI в стиле LLVM
- Оптимизации и опять ParaCL

Контекст

- LLVM исходно проектировался как компилятор, способный работать в многозадачных режимах.
- LLVMContext содержит все глобальные сущности, например типы.
- Примитивные типы можно просто получить:

```
llvm::Type::getInt32Ty(*currentContext)
```

- Более сложные типы конструируются из примитивных.

```
// using FTPrint = void (int);
Type *Tys[] = { getInt32Ty(*currentContext) };
FunctionType *FTPrint =
    FunctionType::get(getVoidTy(*currentContext), Tys, false);
```

Владение контекстом

- LLVM исходно проектировался как компилятор, способный работать в многозадачных режимах.
- LLVMContext содержит все глобальные сущности, например типы.
- Любая работа начинается с создания контекста которое обычно тривиально.

```
Context = new llvm::LLVMContext; // можно и без new
```

- У контекста стёрт конструктор копирования, поэтому его нельзя положить в контейнер вроде vector.
- Означает ли это, что у нас нет возможности следить за его временем жизни автоматически?

Модули

- Итак, у нас есть контекст.
- Далее мы используем его чтобы создать модуль.

```
Module = std::make_unique<llvm::Module>("pcl.module", Context);
```

- Модуль обозначает единицу трансляции. Скомпилируем fib.cc и мы увидим:

```
; ModuleID = 'fib.cc'
source_filename = "fib.cc"
target datalayout = какой-то layout
target triple = "x86_64-pc-linux-gnu"
```



```
clang++ -S -memit-llvm fib.cc
```

DataLayout и TargetTriple

- Технически выставить после создания DataLayout и TargetTriple может быть неплохой идеей.

```
Module->setTargetTriple("x86_64-unknown-unknown");
```

- В обычном кланге есть возможность задавать march или mcpu
- Поддержать во фронтенде PCL эти опции может быть хорошей идеей
- Немного странные правила DataLayout строчек можно посмотреть в официальной документации
- Что ещё может быть в модуле?

Модуль как мультиконтейнер

- Интересный способ заглянуть в содержимое это посмотреть итераторы.
- Итератор по функциям: begin / end / rbegin / rend.
- Итератор по глобальным переменным: global_begin / global_end.
- Итератор по алиасам: alias_begin / alias_end.
- Итератор по косвенным вызовам: ifunc_begin / ifunc_end.
- Итератор по метаданным: named_metadata_begin / named_metadata_end.
- Кроме того для каждого поддержан range: aliases / globals, etc…

```
for (auto &&g : Module->globals()) { // нечто с g
```


Module doxygen

Функции

- Функция создаётся после того как создан её тип.

```
auto *Int32Ty = llvm::Type::getInt32Ty(Context);

// using scanty = int ();
auto *ScanTy = llvm::FunctionType::get(Int32Ty, false);

auto *ScanF = Function::Create(ScanTy, ExternalLinkage,
  "__pcl_scan", Module);
```

- Мы создаём "функцию" и назначаем её в модуль.

Функции

- Функция создаётся после того как создан её тип.

```
auto *Int32Ty = llvm::Type::getInt32Ty(Context);

// using scanty = int ();
auto *ScanTy = llvm::FunctionType::get(Int32Ty, false);

auto *ScanF = Function::Create(ScanTy, ExternalLinkage,
  "__pcl_scan");
```

- Мы создаём "функцию" чтобы назначить её в модуль позднее.

```
Module->getFunctionList().push_back(ScanF);
```

- Здесь используется тот факт что функции это интрузивный список.

Базовые блоки

- Чтобы начать вставку нам понадобиться базовый блок.

```
auto *BB = BasicBlock::Create(*Ctx, "entry", CurrentFunction);
```

- Имя и функция – не обязательные параметры.
- Ещё один параметр, четвертый это блок после которого вставлять (по умолчанию в конец).
- Функция является родителем блока, но блок всегда может быть отвязан от функции через removeFromParent и вставлен в другую через insertInto.
- Кроме того у блока есть eraseFromParent которая отвязывает его от родителя и стирает.

Обсуждение

- Допустим базовый блок точно так же наполняется инструкциями

```
auto *Inst = Instruction::Create(*Ctx, ADD, currentBB);
```

- Вы видите здесь некоторую проблему?

IR Builder: общая идея

- Правильный способ это ортогональное создание инструкций через builder.

```
auto Builder = std::make_unique<llvm::IRBuilder<>>(Ctx);
```

- Класс `llvm::IRBuilder<FolderTy, InserterTy>` имеет два шаблонных параметра, здесь оба по умолчанию.
- `InserterTy` это ваш собственный класс, который должен быть наследником класса `IRBuilderDefaultInserter.`
- `FolderTy` это ваш собственный класс, который должен быть наследником класса `IRBuilderFolder` по умолчанию это `ConstantFolder.`
- Такие шаблонные параметры-типы, с помощью которых выполняются действия называются классами политик (policy classes).

IR Builder: inserters

- Единственный метод для вашего переопределения.

```
virtual void InsertHelper(Instruction *I, const Twine &Name,
  BasicBlock *BB, BasicBlock::iterator InsertPt) const;
```

- Реализация по умолчанию.

```
if (BB) BB->getInstList().insert(InsertPt, I);
I->setName(Name);
```

- Обсуждение: давайте попробуем придумать какого-нибудь нетривиального наследника?

IR Builder: folders

- У класса `IRBuilderFolder` довольно много виртуальных методов.
- Переопределение типичного CreateAdd в классе ConstantFolder.

```
Constant *CreateAdd(Constant *LHS, Constant *RHS,
  bool HasNUW = false, bool HasNSW = false) const override
{ return ConstantExpr::getAdd(LHS, RHS, HasNUW, HasNSW); }
```

- Обычный CreateAdd из IRBuilder использует Folder если аргументы константы

```
if (auto *LC = dyn_cast<Constant>(LHS))
  if (auto *RC = dyn_cast<Constant>(RHS)) {
    auto Add = Folder.CreateAdd(LC, RC, HasNUW, HasNSW);
    return Insert(Add, Name);
  }
```


Folder type

Обсуждение

- Выше упомянут dyn_cast а не dynamic_cast.

```
if (auto *LC = dyn_cast<Constant>(LHS))
```

- Но что такое dyn_cast?

- Основы LLVM IR и экосистема
- Генерация IR
- RTTI в стиле LLVM
- Оптимизации и опять ParaCL

Допустим мы не хотим RTTI

- Предположим у нас есть некая иерархия

```
struct Base; // abstract
struct DerivedLeft: Base; // abstract
struct DerivedRight: Base;
struct MostDerivedL1: DerivedLeft;
struct MostDerivedL2: DerivedLeft;
struct MostDerivedR: DerivedRight;
```

- Это нечто вроде узлов в иерархии ParaCL.
- Мы можем завести виртуальные деструкторы. Но для узлов у нас скорее всего уже есть некий enum.

Допустим мы не хотим RTTI

- Предположим у нас есть некая иерархия.

```
struct Base; // abstract
struct DerivedLeft: Base; // abstract
struct DerivedRight: Base;
struct MostDerivedL1: DerivedLeft;
struct MostDerivedL2: DerivedLeft;
struct MostDerivedR: DerivedRight;
```

```
enum BaseId {
  DerivedRightId,
  MostDerivedL1Id,
  MostDerivedL2Id,
  MostDerivedRId
};
```

- Это нечто вроде узлов в иерархии ParaCL.
- Мы можем завести виртуальные деструкторы. Но для узлов у нас скорее всего уже есть некий enum и мы не хотели бы дублировать то же в RTTI.

Заводим волшебный classof

- База это всегда база.

```
struct Base {
  static inline bool classof(Base const*) { return true; }
  BaseId getValueID() const { return Id; }
```

- Теперь чуть сложнее (допускаем что везде есть `getValueId`)

```
struct DerivedRight: Base {
  static inline bool classof(DerivedRight const*) { return true; }

  static inline bool classof(Base const* B) {
    switch(B->getValueID()) {
    case DerivedRightId: case MostDerivedRId: return true;
    default: return false;
```

Теперь будет работать isa, dyn_cast, etc

- Простейший шаблонный метод.

```
template <typename To, typename From>
bool isa(From const& f) { return To::classof(&f); }
```

- В итоге в LLVM (а вы уже поняли что речь про LLVM) работает.

```
if (isa<Function>(myVal)) { // ....
```

- Аналогично работает `dyn_cast`: возвращает либо указатель либо `nullptr`.

```
if (auto *AI = dyn_cast<AllocationInst>(Val)) { // ....
```

- Его контролируемый вариант `cast`: либо кастует либо abort.

Обсуждение

- Обратите внимание: в нетривиальной иерархии у нас нет ни одного виртуального деструктора.
- Как мы будем удалять по указателю на базовый класс?

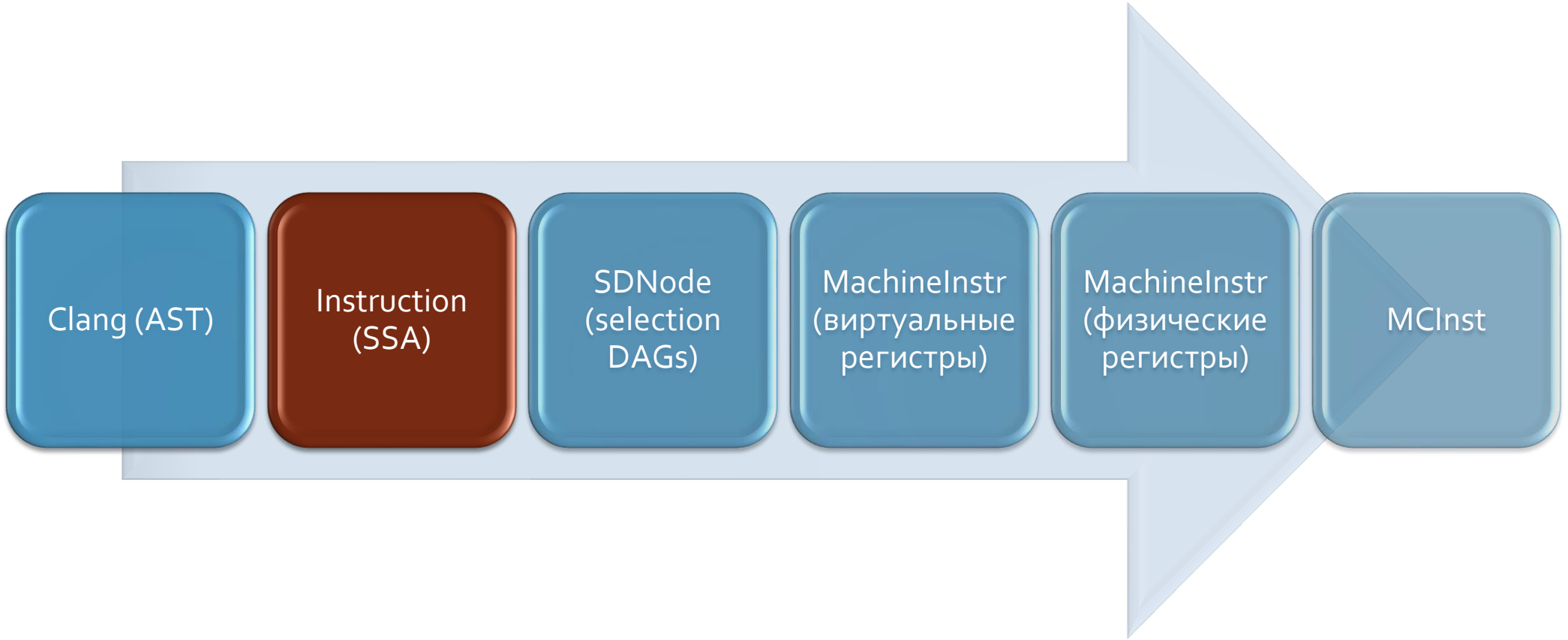
Ответ: ужасно

```
void Value::deleteValue() {
  switch (getValueID()) {

#define HANDLE_VALUE(Name)                                                    \
  case Value::Name##Val:                                                      \
    delete static_cast<Name *>(this);                                         \
    break;

#define HANDLE_MEMORY_VALUE(Name)                                             \
  case Value::Name##Val:                                                      \
    static_cast<DerivedUser *>(this)->DeleteValue(                            \
      static_cast<DerivedUser *>(this));                                      \
    break;

#define HANDLE_CONSTANT(Name)                                                 \
  case Value::Name##Val:                                                      \
    llvm_unreachable("constants should be destroyed with destroyConstant");   \
  break;
// И так далее...
```


Value dtor

- ❑ Основы LLVM IR и экосистема
- ❑ Генерация IR
- ❑ RTTI в стиле LLVM
- ➢ Оптимизации и опять ParaCL

Жизнь инструкции



https://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm

Работа с IR и утилиты llvm toolchain

- Применим вручную оптимизацию tail-call elimination

```
> llvm-as fib_handwritten.ll -o fib_handwritten.bc
> opt -tailcallelim fib_handwritten.bc -o fibopt.bc
> llvm-dis fibopt.bc -o fibopt.ll
```

- Можно подать на утилиту opt различные опции, например `-debug` чтобы посмотреть лог работы или `-dot-cfg`, чтобы сбросить графы
- IR после каждой из оптимизаций на O2 можно посмотреть так:

```
> llc -O2 --print-after-all fib_handwritten.ll
```

- LLVM 11 делает более 100 высокоуровневых оптимизаций

Модульность llvm toolchain: clang

- LLVM это библиотека, проводящая работу над LLVM IR и способная грузить в свою очередь библиотеки бэкендов.
- Для того, чтобы вся конструкция заработала как сишный компилятор, используется фронтенд clang.
- Команда для оптимизатора напрямую:

```
> llc -O2 --print-after-all fib_handwritten.ll
```

- Команда для clang с указанием пробросить оптимизатору.

```
> clang -O2 -mllvm --print-after-all -x ir fib_handwritten.ll
```

Структура

```
> clang -mllvm --debug-pass=Structure -O2 fib.c -S -emit-llvm
```

- Эта команда показывает дерево всех фаз оптимизации и опции для каждой из их групп.

```
ModulePass Manager
   Force set function attributes
   Infer set function attributes
   Interprocedural Sparse Conditional Constant Propagation
   Called Value Propagation
   Global Variable Optimizer
   FunctionPass Manager
      Dominator Tree Construction
      Promote Memory to Register
   Dead Argument Elimination
   FunctionPass Manager
      Dominator Tree Construction
      Basic Alias Analysis (stateless AA impl)
....
```

Домашнее задание: ParaCL compiler

- Разработайте кодогенератор языка ParaCL (далее – парасил) в LLVM IR в объёме арифметика + if + while.
- Скомпилированная программа должна считывать со стандартного ввода всё, что считывается, и печатать на стандартный вывод всё что нужно распечатать.
- Скрафтить хорошие тесты – важная часть задачи.
- Эта домашняя работа следует за симулятором парасила из пятого семинара.
- Надеюсь все дошли в нём до высоких уровней?

IR Builder для ParaCL

- Сначала мы ставим insertion point на текущий базовый блок.

```
auto *BB = llvm::BasicBlock::Create(Ctx, "entry", currentFn);
currentBuilder->SetInsertPoint(BB);
```

- Далее мы используем builder для создания инструкций.

```
case Ops::Less:
  return currentBuilder->CreateICmpSLT(LeftV, RightV);
```

- Полный список того что можно создать, можно посмотреть в справке.
- Спойлер: почти всё.

Секрет изменяемых переменных

- Допустим нам надо поддержать программу на парасиле вроде такой.

```
foo = func(a, b) {
  // много кода
  a = 1;
  b = a + 1;
}
```

- Переменные a и b изменяемые и чтобы не заморачиваться с phi-nodes, можно построить для них аллоки.
- Но, если они поздно, как сместиться в entry block чтобы сделать аллоки там?

Выход: временный второй билдер

- Мы можем одновременно поддерживать сколько угодно билдеров.

```
auto *CreateEntryBlockAlloca(const std::string &varname) {
  IRBuilder<> TmpB(&currentFunction->getEntryBlock(),
                   currentFunction->getEntryBlock().begin());
  auto *Int32Ty = llvm::Type::getInt32Ty(*currentContext);
  return TmpB.CreateAlloca(Int32Ty, 0, varname.c_str());
}
```

- Билдер умирает, но созданная им аллока теперь принадлежит базовому блоку и живёт.
- Кроме того нам нужно будет отображение имени переменной в её аллоку.

Обсуждение

- После того как вы сделали LLVM IR, над ним происходят оптимизации.
- Далее кодогенерация в ассемблер конкретной машины.

Литература

- Information technology – Programming languages – C++, ISO/IEC 14882, 2017
- Bjarne Stroustrup – The C++ Programming Language (4th Edition)
- Davide Vandevoorde, Nicolai M. Josuttis – C++ Templates. The Complete Guide, 2nd edition, Addison-Wesley Professional, 2017
- Chris Lattner – LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, CGO'2004
- Mike Shah – Introduction to LLVM, Fosdem'2018
- Bridgers, Piovezan – LLVM IR Tutorial, EuroLLVM'2019
- Kaleidoscope language tutorial