llvm & clang

или компилятор для людей

TOC

- история
- использование
  - address sanitizer
  - инструментация кода
  - модификация исходников
  - анализатор
  - оптимизируюший JIT
- внутреннее устройство и модификация
  - промежуточное представление
  - plugins API

О чем речь?

llvm - Low Level Virtual Machine
framework для построения компиляторов

clang - C/C++/ObjC компилятор (Frontend)
базируется на llvm

История

Как появился llvm & clang

- 2003-2004 как иследовательский проект университета Иллинойса (University of Illinois)
- Chris Lattner
- Vikram Adve
- Впоследствии стал спонсироваться Apple

Приемущества

- Свободная лицензия (BSD style)
- Модульный и современный дизайн (Написан на C++)
- Быстрее чем gcc (время компиляции и потребляемая память)
- Подробные сообщения об ошибках
- Легок в изучении и модификации

Кто использует

- Apple (Mac OS X 10.7 Lion and iOS5)
- Adobe (Action Script 3 compiler)
- Google (Android RenderScript)
- Nvidia (NVCC Cuda compiler)
- Cray (Cray x86 compiler)
- Iced Tea (Shark JIT)
- FreeBSD (building kernel & system)
- Linux (in progress...)

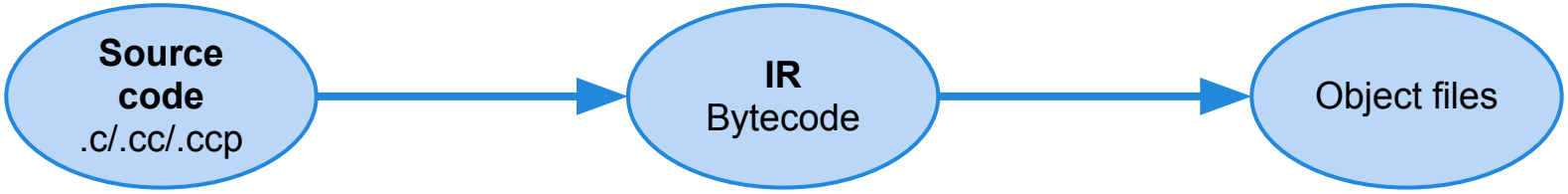
Устройство

Архитектура llvm

ЯВУ в промежуточное представление

Оптимизация

Генерация машинного кода JIT



FrontEnd

- Лексический, семантический анализ
- Построение AST (Abstract syntax tree)
- Генерация машинного кода/промежуточного кода

FrontEnd (язык C)

Два способа использовать как C компилятор

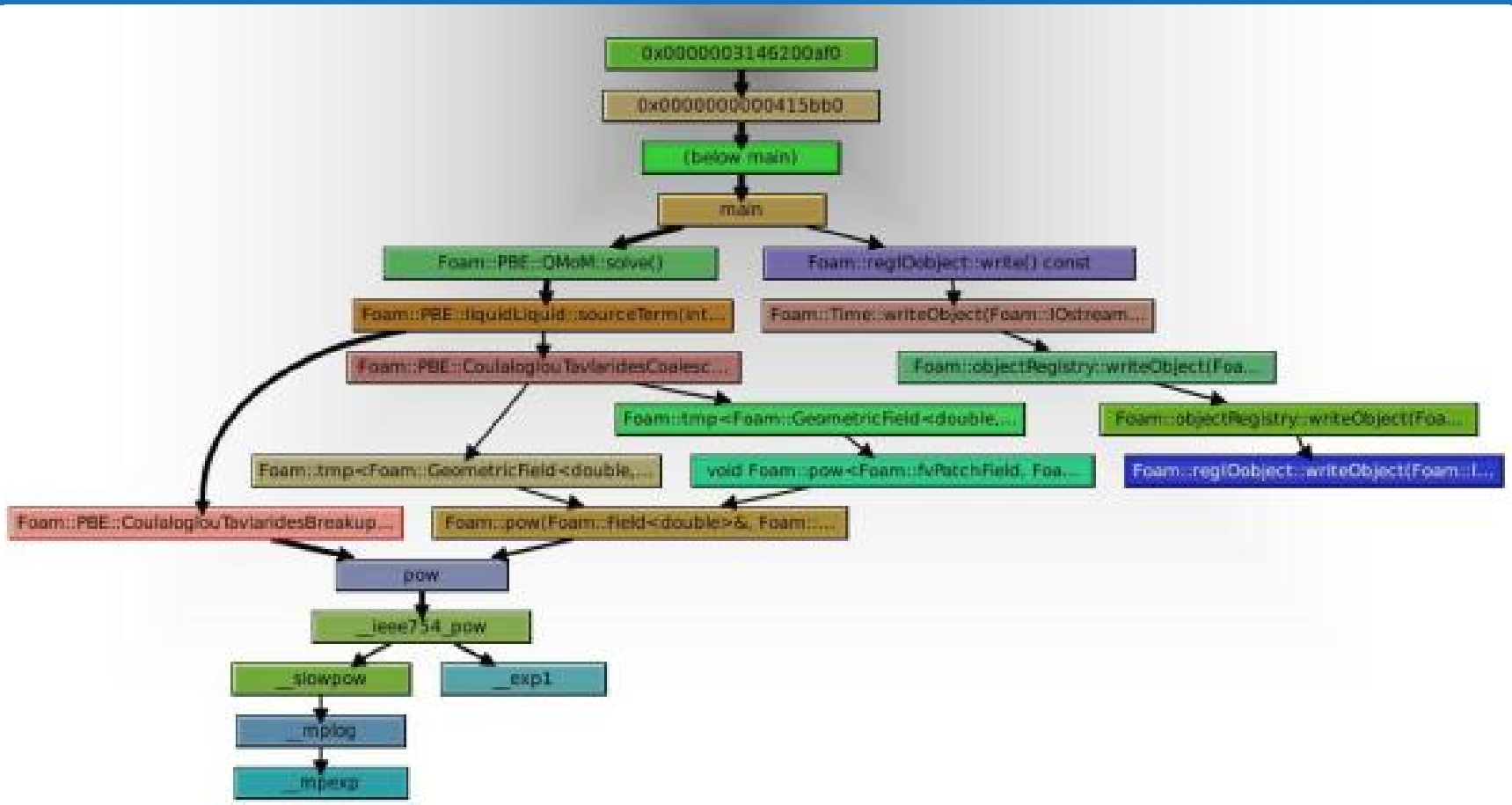
- clang
  Написанный с нуля разработчиками llvm
  - Быстрее чем GCC (2.3 раза gcc 4.2 vs clang 3.1)
  - Подробные сообщения об ошибках

- dragon-egg
  Плагин к GCC позволяющий использовать GCC с llvm
  - GCC и clang имеют расхождение в трактовании стандартов

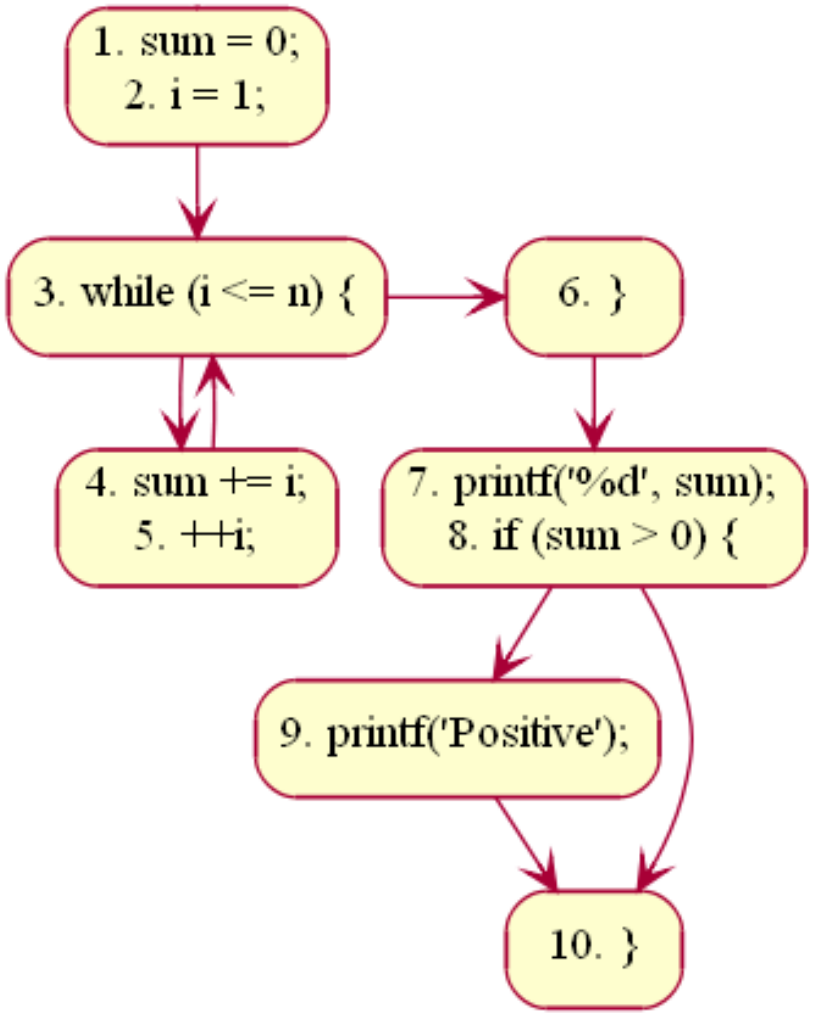
LLVM Framework

- Static Single Assignment (SSA) form
  - SSA originally invented by IBM :-)
- Анализ (всего ~30)
  - Построение Control flow graph (CFG)
  - Построение Call graph
  - Построение Dominators tree
  - Regions
  - Alias Analysis
- Основные оптимизации (всего ~80)
  - Dead code elimination
  - Constant propagation
  - Inline functions
  - Loop unrolling
  - ...

Framework: Call Graph

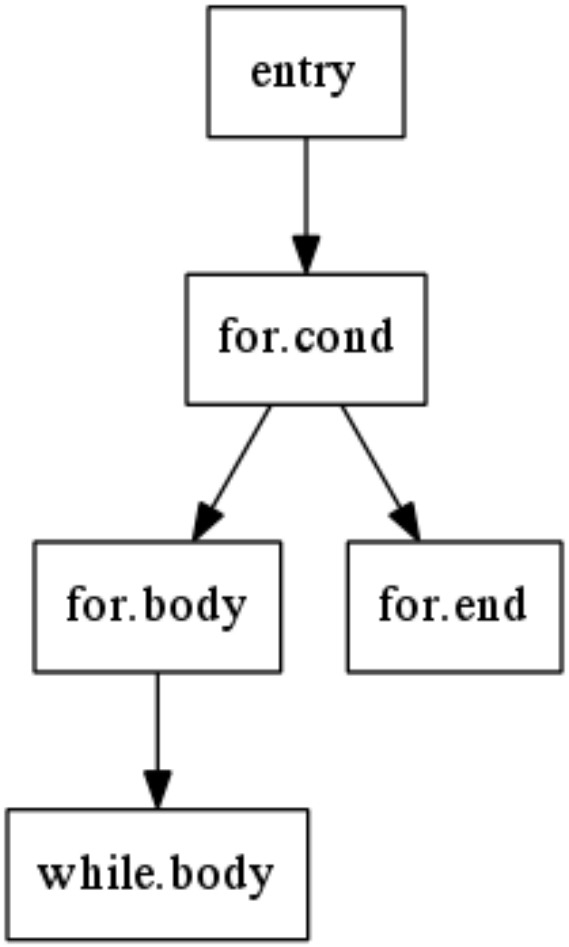


Framework: Control Flow Graph



Framework: Dominators tree

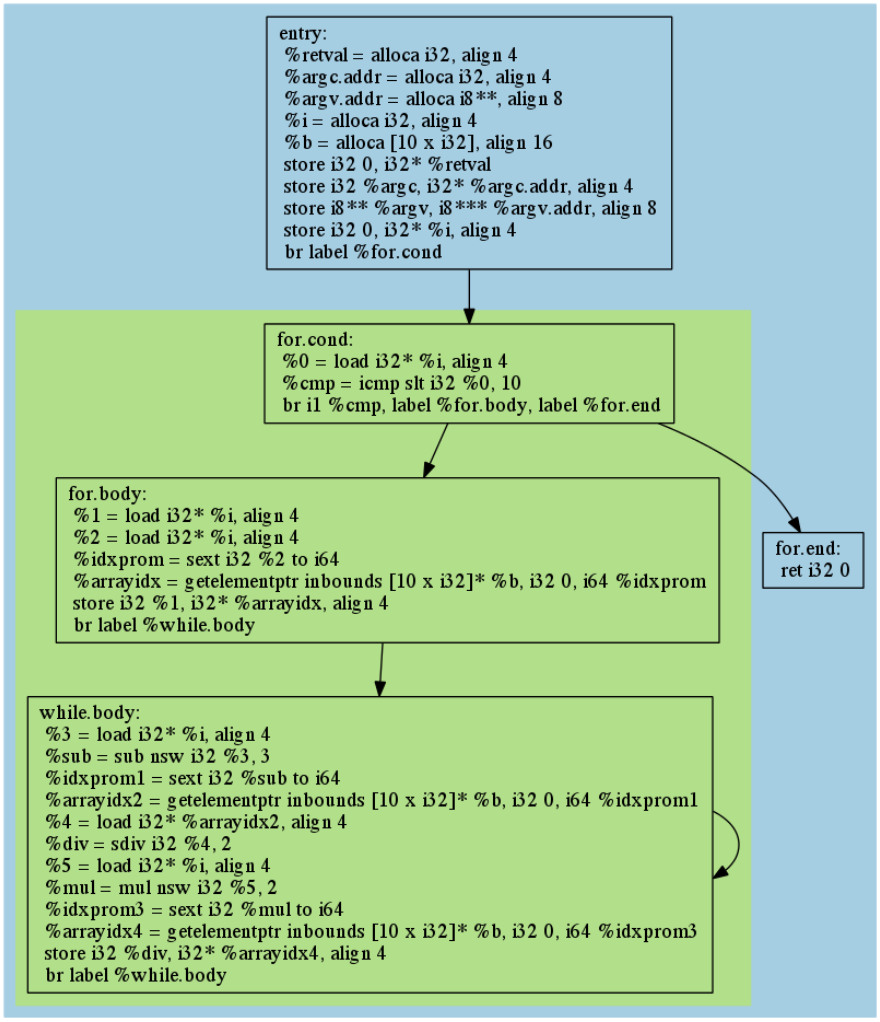
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
  int i;
  int b[10];

  for(i = 0; i<10; i++){
    b[i] = i;
    while(1){
        b[i*2] = b[i-3]/2;
    }
    printf("%x",b[i]);
  }

  return 0;
}
```



Dominator tree for 'main' function

Framework: Regions

Участок программы с одним входом и одним выходом



Region Graph for 'main' function

Framework: Alias Analysis

```
a = &obj;
*a = b;
c = a;
*c = d;

*a = ?
```

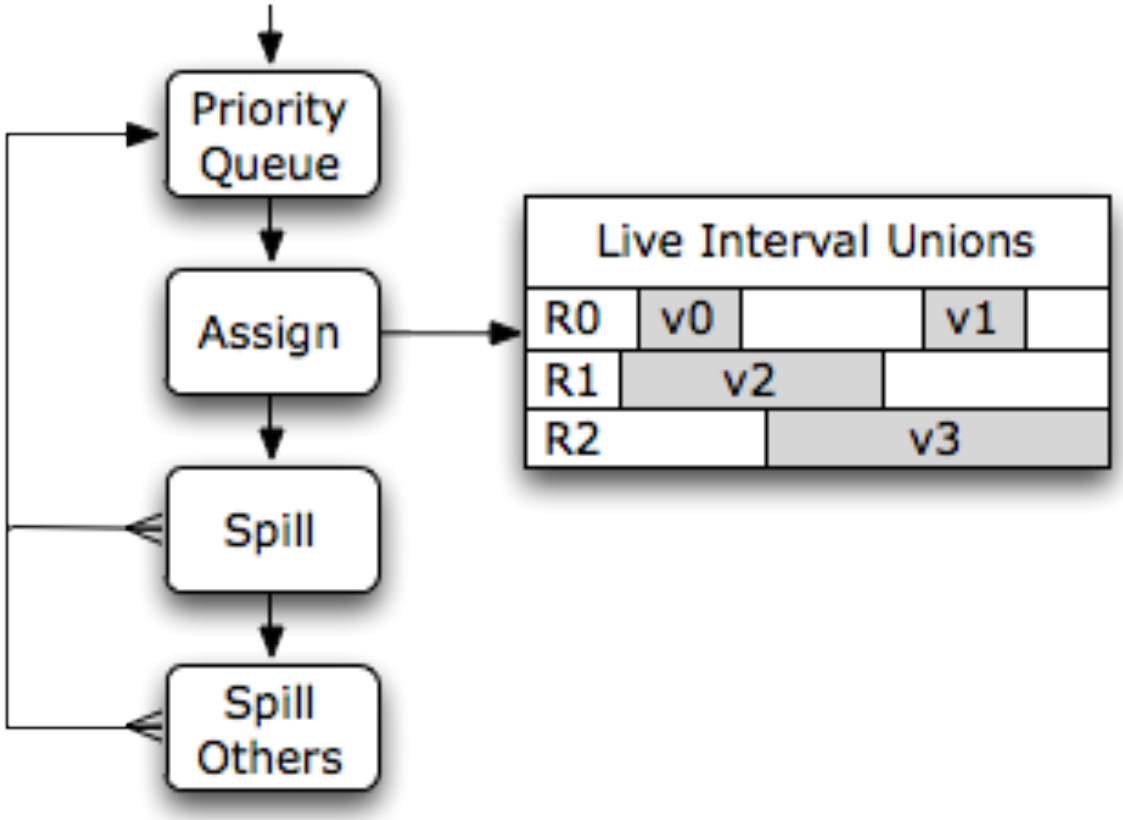
Backend: Code generator

- Архитектурно независимый
  - ISA описывается на универсальном языке (Table gen)
- Функционально независим
  - генерация машинного кода
    - MachO
    - ELF
    - PE
  - генерация ассемблерного листинга
  - Just in time (JIT) compiling
- Register allocation
- Архитектурно-зависимая оптимизация

Register allocation: Liveness

```
A = ...
C = ...
...
B = A+C
... = C
...
 ? = B
```

Register allocation



Поддерживаемые архитектуры

X86/X86_64	Production quality
ARM	Production quality
MIPS/MIPS64	Good quality
PowerPC/PPC64	Good quality
SPARC (v8/v9)	Normal quality
CellSPU	Normal quality
Hexagon	Qualcomm contribute
NVPTX	Nvidia contribute
XCore	unknown quality
MSP430	unknown quality

Использование

Собираем из исходников

git clone https://github.com/sergioche/llvm-workshop.git

- Потребуется
  - git
  - cmake

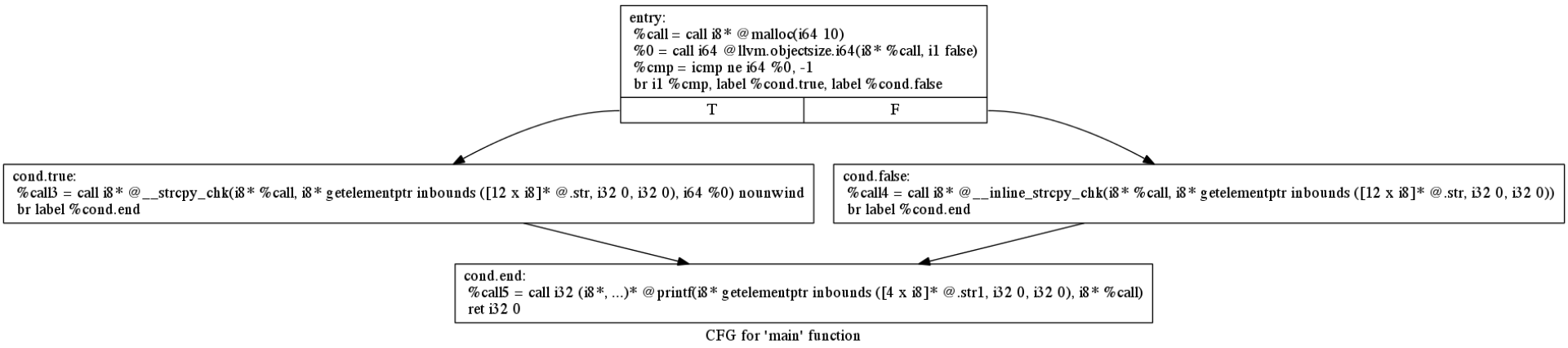
- 01-getting_started
  - build-source.sh
  - clone-source.sh

Простые манипуляции

- Генерация байт кода
  - clang -cc1 -emit-llvm
  - clang -cc1 -emit-llvm-bc
- Дизасемблирование кода
  - llvm-dis
  - llvm-as
- Генерация машинного кода
  - llc -filetype=obj
  - llc -filetype=asm
- Оптимизация
  - opt -load=<plugin> [-optname] in.bc -o out.bc

Отладка

- opt -help | grep dot
  - -dot-callgraph
  - -dot-cfg
  - -dot-dom
  - -dot-postdom
  - -dot-regions
- dot -Tpng cfg.main.dot -o main.png



CFG for 'main' function

Address Sanitizer

- Использовать очень просто
  - clang -faddress-sanitizer
- Ошибки которые можно найти
  - out of bound memory read (stack, heap, global)
  - out of bound memory write (stack, heap, global)
  - use after free
  - use after return
- Падение производительности ~2 раза

Address Sanitizer

	AddressSanitizer	Valgrind/Memcheck	Dr. Memory	Mudflap	Guard Page
technology	CTI	DBI	DBI	CTI	Library
ARCH	x86	x86,ARM,PPC	x86	all(?)	all(?)
OS	Linux, Mac	Linux, Mac	Windows, Linux	Linux, Mac(?)	All (1)
Slowdown	**2x**	20x	10x	2x-40x	?
Detects					
Heap OOB	**yes**	yes	yes	yes	some
Stack OOB	**yes**	no	no	some	no
Global OOB	**yes**	no	no	?	no
UAF	**yes**	yes	yes	yes	yes
UAR	**some**	no	no	no	no
UMR	no	yes	yes	?	no
Leaks	not yet	yes	yes	?	no

Source code manipulation

- libclang
- python binding

Static analysis

- Использование
  - clang --analyze

- Что умеет искать
  - clang -cc1 -analyzer-checker-help
    - 5 unix
    - 9 security
    - 11 debug
    - 15 core
    - 20 osx
    - 23 alpha

Just in time compiler (JIT)

Maybe in the Future ;(

Внутренности

IR

- RISC подобное трех операторное представление
  - **$3** = add i32 **$1**,**$0**
- Бесконечное число регистров (SSA форма)
  - **$val2** = add i32 **$val1**,1
- Типизированное
  - $val2 = add **i32** $val1,1
-

plugins (passes)

Вставка инструкций (IRBuilder)

Отладка