

# Виртуальные машины

## Лекции 3-7: Интерпретация

Олег Плисс  
[Oleg.Pliss@gmail.com](mailto:Oleg.Pliss@gmail.com)

2023



# Определение

- *Простой интерпретатор* — это механизм непосредственного исполнения отдельных языковых элементов программ
  - Примеры языковых элементов: строка, лексема, команда
  - Интерпретатор может быть программным, аппаратным или программно-аппаратным
  - В дальнейшем мы будем рассматривать только программные интерпретаторы

# Интерпретация

- Ядро VM — интерпретатор кода виртуального процессора
  - VM может и не содержать интерпретатора
- Интерпретация исходного текста программы
  - Не эффективна из-за многократного повторения лексического и синтаксического анализов
  - Текстовое представление выбирается для облегчения взаимодействия с человеком
- Интерпретация промежуточного кода
  - Промежуточное представление (ПП) выбирается для облегчения анализа и интерпретации
  - ПП включает в себя промежуточный код (ПК) и, возможно, дополнительные таблицы
    - Например, привязка к исходному тексту
  - Исходный текст компилируется в бинарное ПП
  - Бинарный промежуточный код интерпретируется
  - Компиляция исходного текста в ПП может быть неявной динамической

# Виды интерпретаторов

- Рекурсивный интерпретатор
  - Обходит код программы, рекурсивно вызывая себя для операндов инструкций
  - При вызове подпрограммы вызывает себя с подпрограммой и параметрами её вызова
  - Не требует отдельных стеков
  - Обычно применяется для промежуточного кода, представленного текстами, деревьями или графами
- Итеративный интерпретатор
  - Последовательно в цикле перебирает инструкции программы
  - Параметры и результаты передаются в виртуальных регистрах или на стеке операндов
  - При вызове подпрограммы формирует новую секцию в стеке вызовов, меняет указатель текущей инструкции, продолжает выполнение цикла
  - Применяется для промежуточного кода, представленного потоками, последовательностями или массивами инструкций (*линейными кодами*)

# Компоненты итеративного интерпретатора

- Набор инструкций
  - *add, load, branch...* (может быть расширяемым)
- Виртуальные регистры
  - *ip* — адрес текущей инструкции
  - *sp* — указатель стека
  - *fp* — адрес текущей рамки стека вызовов
- Стеки
  - *Стек вызовов* для организации вызовов подпрограмм, обычно состоит из рамок/секций/записей активации
  - *Стек операндов* для передачи параметров и выдачи результатов виртуальных инструкций (может быть частью стека вызовов или вообще отсутствовать)
- Цикл интерпретатора
  - Включает декодер инструкций и их реализацию

# Распространяемый и выполняемый форматы кода

- Распространяемый формат
  - Формат файла, предназначенный для распространения программ
  - Стандартизован
  - Машинно-независим
  - Расширяем
  - Компактен
  - Пример: *.jar* файлы Java
- Выполняемый формат
  - Представление кода в памяти VM, в основном предназначенное для его выполнения
  - Порождается из распространяемого формата после его **загрузки, линковки и верификации**
    - Может его дополнять или полностью замещать
    - Статически или динамически
    - Преобразование производится на том же или другом устройстве

# Представление виртуальной инструкции

- Токен
    - Целое число, однозначно идентифицирующее инструкцию и, возможно, все или некоторые из ее операндов
    - Остальные операнды могут быть закодированы в потоке инструкций вслед за токеном
    - Токены **не зависят от аппаратуры и размещения кода в памяти**
    - Пример: байтовые коды Smalltalk VM или JVM
  - Адрес подпрограммы
    - Единственная инструкция — вызов подпрограммы, подразумеваемая по умолчанию
    - С другой стороны, расширяемый набор инструкций, представленных адресами их реализаций
    - Пример: шитый код Forth-машины
- Bell, J.K., Threaded code, Communications of the ACM vol 16, nr 6 (Jun) 1973, pp.370-372

# Виды линейных кодов

- Байтовый код
- Шитый код
  - Подпрограммный
  - Прямой
  - Косвенный
- Токен-шитый код
  - Прямой
  - Косвенный



# Байтовый код

*foo (int a, int b)*  
*→ a + b + 1;*

iload	0
iload	1
iadd	
iconst	1
iadd	
ireturn	

1 байт

```
void Interpreter (void) {  
    const byte* ip;  
    ...  
    for (;;) {  
        switch (*ip++) {  
            ...  
            case iadd: {  
                const int a = pop();  
                const int b = pop();  
                push(a+b);  
                break;  
            }  
            ...  
        }  
    }  
}
```

# Ассемблерный интерпретатор байтового кода с явным циклом

```
InterpreterLoop:
```

```
    tmp = *ip++;
```

```
    jmp BytecodeTable[BytesInWord * tmp];
```

```
...
```

```
L_iadd:
```

```
    pop tmp1;
```

```
    pop tmp2;
```

```
    tmp1 += tmp2;
```

```
    push tmp1;
```

```
    jmp InterpreterLoop;
```

```
...
```

```
BytecodeTable: .word ..., L_iadd, ...
```

Здесь *ip*, *tmp*, *tmp1*, *tmp2* — машинные регистры,  
причем *tmp* может совпадать с *tmp1* или *tmp2*.

# Ассемблерный интерпретатор байтового кода с неявным циклом

```
next: macro()  
    tmp = *ip++;  
    jmp BytecodeTable[BytesInWord * tmp];  
endm
```

Interpreter:

```
    next();
```

...

L\_iadd:

```
    pop tmp1;  
    pop tmp2;  
    tmp1 += tmp2;  
    push tmp1;  
    next();
```

...

```
BytecodeTable: .word ..., L_iadd, ...
```

# Подпрограммный шитый код

<i>jsr</i>	iload	0
<i>jsr</i>	iload	1
<i>jsr</i>	<i>iadd</i>	
<i>jsr</i>	iconst	1
<i>jsr</i>	<i>iadd</i>	
<i>ret</i>		

1 слово

*jsr* — машинная инструкция  
вызова подпрограммы

*ret* — машинная инструкция  
возврата из подпрограммы

Цикл интерпретатора отсутствует.

```
.code
...
iadd: pop(tmp1);
      pop(tmp2);
      tmp1 += tmp2;
      push(tmp1);
      ret
```

*iadd* — метка в коде

*tmp1*, *tmp2* — регистры процессора

*pop*, *push* — макросы операций  
со стеком операндов

# Прямой шитый код

<i>jsr</i>	<i>enter</i>	
	iload	0
	iload	1
	iadd	
	iconst	1
	iadd	
	<i>exit</i>	

1 слово

*enter* / *exit* — вход в / выход из интерпретатора **данного фрагмента кода**.

Интерпретатор:

```
next: macro()  
    tmp = *ip++;  
    jmp tmp;  
endm  
enter: rpush(ip);  
        pop ip;  
        next();  
exit:  rpop(ip);  
        next();  
iadd: pop tmp1;  
        pop tmp2;  
        tmp1 += tmp2;  
        push tmp1;  
        next();
```

*pop*, *push* — машинные инструкции  
*rpop*, *rpush* — макросы операций со стеком возвратов  
++ — инкремент на размер слова

# Косвенный шитый код

enter	
iload	0
iload	1
iadd	
iconst	1
iadd	
exit	

Интерпретатор:

```
next:  macro()  
        tmp = [ip++];  
        jmp [tmp];  
        endm  
enter: rpush(ip);  
        ip = ++tmp;  
        next();  
exit:  rpop(ip);  
        next();  
iadd:  .word @iadd  
@iadd: pop tmp1;  
        pop tmp2;  
        tmp1 += tmp2;  
        push tmp1;  
        next();
```

Единственная разновидность шитого кода, не содержащая машинных инструкций.

Dewar, R.B.K., Indirect threaded code.

*Communications of the ACM*, June 1975, pp.330-331

# Прямой токен-шитый код

<i>jsr</i>	Interpreter
iload	0
iload	1
iadd	
iconst	1
iadd	
ireturn	

1 байт

```
next: macro()  
    tmp = *ip++;  
    jmp BytecodeTable[BytesInWord*tmp]  
endm
```

Interpreter:

```
    rpush(ip);  
    pop ip;  
    next();  
L_iadd:  
    pop tmp1;  
    pop tmp2;  
    tmp1 += tmp2;  
    push tmp1;  
    next();
```

Байтовый код встроен в прямой шитый как вариант интерпретации. Инструкция вызова передает управление на начало кода вызываемой функции.

# КОСВЕННЫЙ ТОКЕН-ШИТЫЙ КОД

Interpreter	
iload	0
iload	1
iadd	
iconst	1
iadd	
ireturn	

1 байт

Interpreter:

```
    rpush(ip) ;  
    ip = tmp + BytesInWord;  
    next();
```

L\_iadd:

```
    pop tmp1;  
    pop tmp2;  
    tmp1 += tmp2;  
    push tmp1;  
    next();
```

next: macro()

tmp = \*ip++;

jmp BytecodeTable[BytesInWord\*tmp]  
endm

Инструкция вызова загружает адрес начала кода вызываемой функции в регистр *tmp* и передает управление по хранящемуся там адресу интерпретатора.



# Передача параметров инструкциям

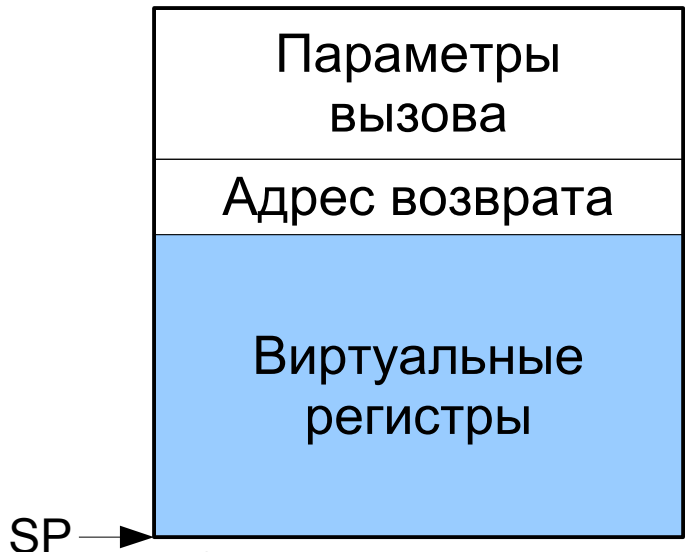
# Параметры виртуальных инструкций

- Неявные на вершине стека операндов
- Явные и неявные в виртуальных регистрах
- Непосредственные операнды
  - Значения разных типов, в т.ч. структурных
  - Номера регистров
    - При явной передаче параметров в регистрах
  - Позиции меток в коде
  - Ссылки на объекты и их элементы
    - Например, класс или поле объекта
  - Внешние символические ссылки

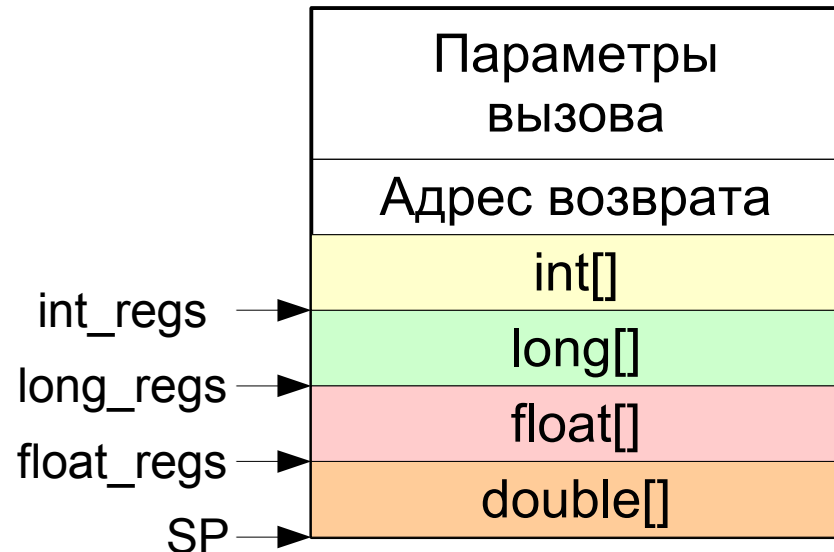
# Явная передача параметров в виртуальных регистрах (1)

- Активация функции фиксированного размера
- Содержит массив бестиповых виртуальных регистров
- ... Или несколько массивов разных примитивных типов: *int*, *long*, *float*, *double*...

Бестиповые  
регистры



Типизированные  
регистры



# Явная передача параметров в виртуальных регистрах (2)

- Число регистров определяется функцией
  - Их должно быть достаточно для размещения всех живых локальных переменных
    - Включая временно отводимые при вычислении выражений и для передачи параметров при вызовах
    - В разные моменты в одном регистре могут располагаться разные локалы
  - Но что, если регистров оказалось недостаточно?
    - Номер регистра ограничен шириной отведенного для него битового поля инструкции
    - Код не обязательно пишется вручную
    - Деление регистров по типам более экономно использует индексы
    - Можно запретить порождать такой код
    - ... или расширить поля номеров регистров
    - ... или завести префикс ширины таких полей
    - ... или использовать кодировку переменной длины

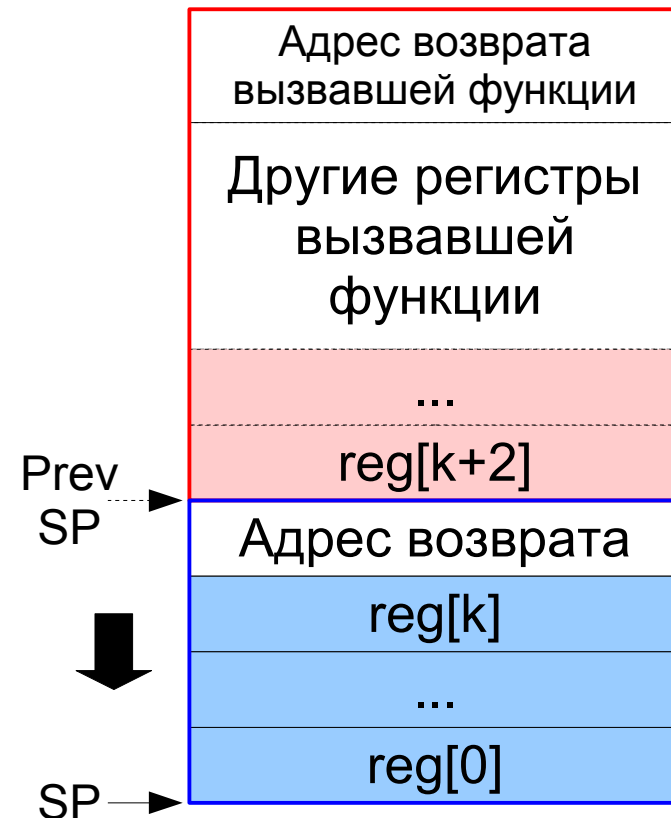
# Явная передача параметров в виртуальных регистрах (3)

- Инициализация регистров
  - В начале функции её регистры не инициализированы, кроме параметров вызова
- Инструкция вызова:
  - копирует параметры из заданных регистров вызвавшей функции в регистры вызываемой
    - Фиксированные или заданные функцией
  - ... а при возврате в заданный регистр вызвавшей функции копируется результат

call	foo	ra	rb	rc
------	-----	----	----	----

# Явная передача параметров в виртуальных регистрах (4)

- Оптимизация: индексация параметров вызова
  - Если выполнены все следующие условия:
    - Виртуальные регистры бестиповые
    - Секция вызвавшей функции непосредственно предшествует секции вызванной в том же стеке
    - Параметры вызова расположены в известных регистрах вызвавшей функции
  - Можно эти регистры **вызвавшей** функции индексировать как регистры **вызванной** функции
    - Индекс регистра — просто индекс массива в памяти



# Явная передача параметров в виртуальных регистрах (5)

- Инструкции длинные
  - Часто 2-3 и более операндов
- Декодирование каждой из них сложное
  - Нужно извлечь индексы всех операндов
- Если сравнить с передачей параметров на стеке, инструкций порождается меньше, но порождённый код длиннее

# Явная передача параметров в виртуальных регистрах (6)

- Усложняется компилятор языка в код VM
  - Нужно минимизировать число виртуальных регистров
    - Сокращение размера записи активации
    - Улучшение пространственной локальности доступа
  - Нужно отслеживать **живость** их значений
    - Число локалов каждого типа определяется при входе в функцию
    - Они не всегда всегда хранят живые значения — например, не инициализированы в начале
    - Это важно для верификатора, сборщика мусора и рефлексии (в частности, отладчика)
    - Таблицы живости большие, их нужно **сжимать** и хранить в метаданных
    - При передаче параметров через стек мёртвые временные значения снимаются со стека, а не меняют таблицы живости
- За счёт этого может упроститься распределение регистров в динамическом компиляторе



# Неявная передача параметров через стек операндов

- Стек операндов неограниченной глубины
  - В конкретной реализации стек всегда конечен
  - Нужно следить за его возможным переполнением
    - И исчерпанием в некорректном коде
  - Вариант:
    - Потребовать, чтобы глубина стека была легко вычислима во время компиляции (в т.ч. запретить слияние потоков управления с разной глубиной стека)
    - Проверять переполнение при входе в функцию
    - Верификация кода исключает исчерпание стека
- Упрощает:
  - Набор виртуальных инструкций
    - Загрузка значений в стек отделена от их обработки
  - Реализацию интерпретатора
  - Компилятор языка в виртуальный код
- Используется в большинстве современных VM

# Сравнение порождённого кода с виртуальными регистрами и стеком

*foo (int a, int b) → a + b + 1;*

- Виртуальные регистры (3 инструкции, 10 байтов)

```
// Входные параметры в регистрах: r0 = a, r1 = b
    iadd r0, r0, r1;           // r0 = r0 + r1;
    iaddi r0, r0, 1;          // r0 = r0 + 1;
    ireturn r0;
```

- Стек операндов (6/4 инструкций, 9/5 байтов)

```
    iload 0;    // Эти 2 инструкции не нужны,
    iload 1;    // если параметры вызова
    iadd;       // передаются на стеке операндов
    iconst 1;
    iadd;
    ireturn;
```

# Сравнение реализаций *iadd* с виртуальными регистрами и стеком

- Виртуальные регистры

```
const int a = *ip++;  
const int b = *ip++;  
const int c = *ip++;  
reg[a] = reg[b] + reg[c];
```

iadd	ra	rb	rc
------	----	----	----

int\* **reg** - массив  
целочисленных  
виртуальных регистров  
текущей активации

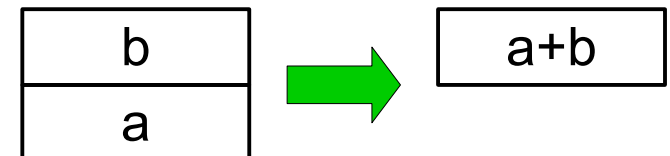
- Стек операндов

```
const int a = pop();  
const int b = pop();  
push(a+b);
```

iadd
------

- Оптимизация для верифицированного кода

```
const int a = pop();  
*sp += a;
```



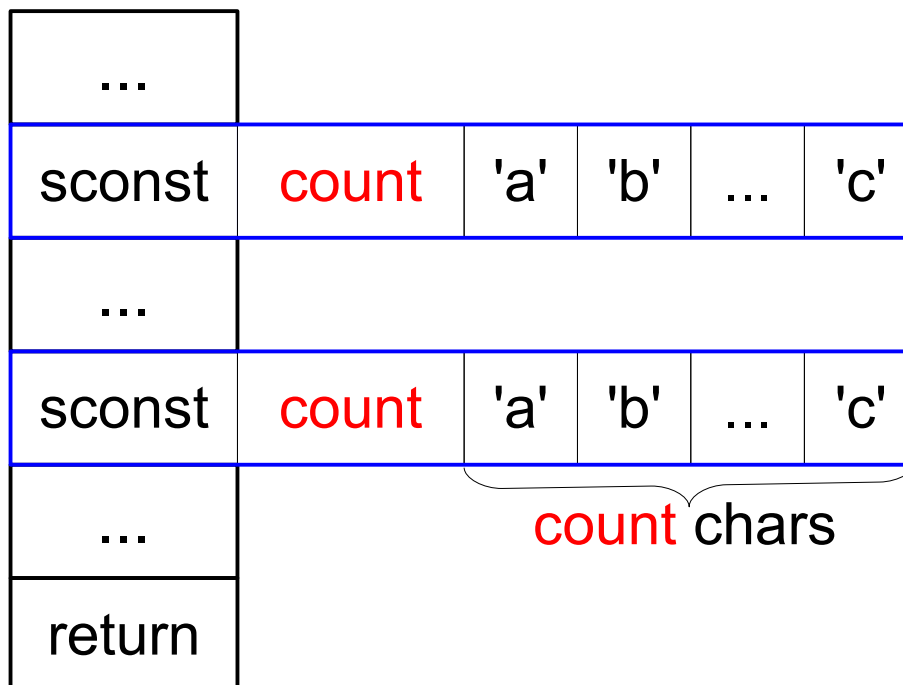
# Доступ к непосредственным операндам инструкций (1)

- Битовые поля самой инструкции
  - Не применимо к шитому коду
- Выборка из потока инструкций при помощи виртуального регистра `ip`  
`const byte arg = *ip++;`
- Доступ к более чем байтовым операндам сложнее
  - Порядок байтов виртуального и аппаратного процессоров может быть различным
    - Его можно поменять во время преобразования распространяемого кода в исполняемый
  - Аппаратура может требовать выравнивания на соответствующую границу

# Доступ к непосредственным операндам инструкций (2)

- Непосредственный операнд может быть переменного размера
  - Сериализованное представление любого типа (например, строка со счетчиком)

Код



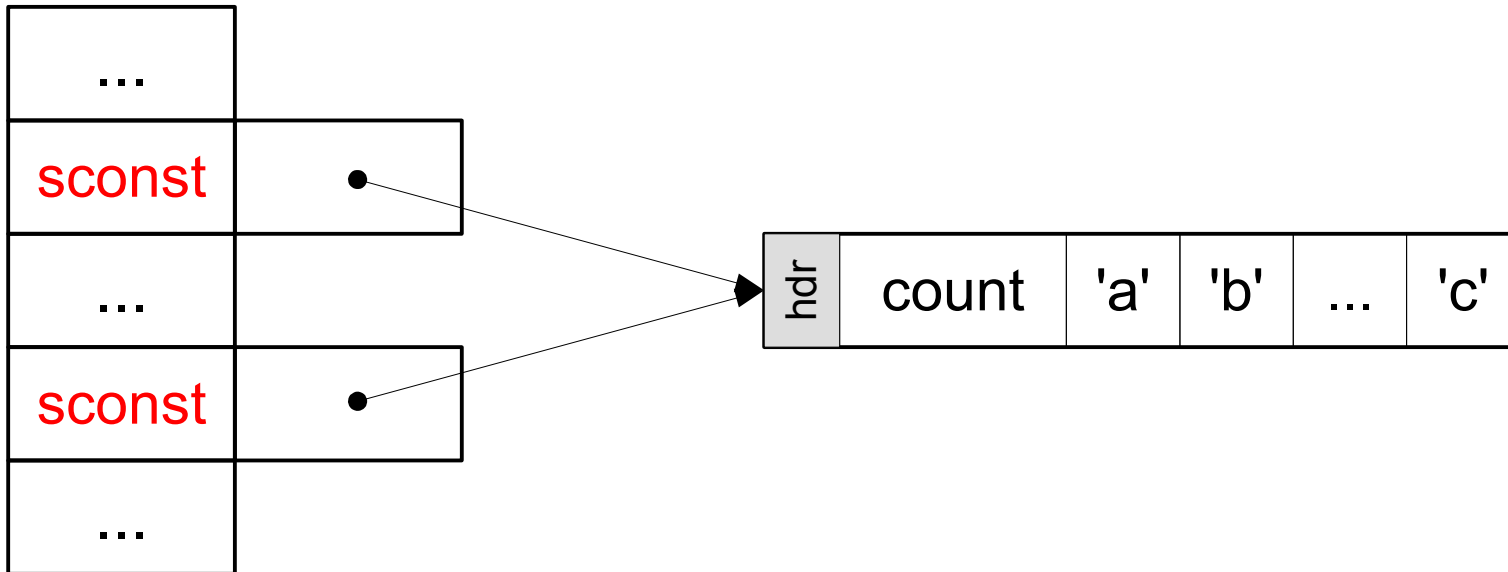
# Литеральный (константный) пул (1)

- Длинные непосредственные операнды:
  - Увеличивают число копий констант
    - При каждом выполнении инструкции из сериализованного представления создается новая константа
  - Замедляют интерпретатор
  - Удлиняют код
- Литеральный (константный) пул
  - Свяжем с кодом область памяти для размещения констант
  - Инструкции могут ссылаться туда по адресам или смещениям констант от его начала
    - Ширина адреса машинно-зависима
  - Инструкции могут совместно использовать константы
  - Десериализация констант может производиться один раз во время **ЛИНКОВКИ**

# Литеральный (константный) пул (2)

Код

Литеральный пул



- Во время линковки кода можно объединять литеральные пулы, устраняя при этом дублирование констант

# Кодирование непосредственных операндов

- Встречающиеся в коде целые числа и смещения **распределены неравномерно**
  - Малые значения встречаются гораздо чаще
- Если отвести для них меньше битов, длина кода сократится
  - Это существенно только для байтового кода — в шитом коде доминируют метки размером в слово
- Что делать, если значение не помещается в отведенные для него биты?
  - Завести варианты инструкций с разной шириной непосредственных операндов
    - Но пространство байткодов ограничено
  - Завести байткод - префикс ширины операндов
    - Меняет ширину **всех** непосредственных операндов
  - Использовать кодировку переменной длины
    - Сравнительно медленное декодирование



# Кодировка переменной длины

- Little Endian Base 128 (LEB128)
  - Используется в отладочном формате DWARF и в исполняемом коде WebAssembly
  - Беззнаковый вариант ULEB128
  - Старший бит текущего байта — признак продолжения значения в следующем байте

```
uint64_t ULEB128_decode (const uint8_t* p) {  
    uint64_t value = 0;  
    for (uint shift = 0;; shift += 7) {  
        const uint64_t next_byte = *p++;  
        value |= (next_byte & 0x7F) << shift;  
        if (!(next_byte >> 7)) return value;  
    }  
}
```

# Ссылки из кода на позиции в коде

- Некоторые инструкции могут ссылаться на *позиции в коде*
  - Например, инструкции локальных переходов
- Представления позиций в коде
  - Адрес позиции
  - Смещение позиции относительно инструкции
    - Позиционно-независимый код
    - Обычно смещения короткие — можно обойтись меньшим числом битов

# Символические ссылки (1)

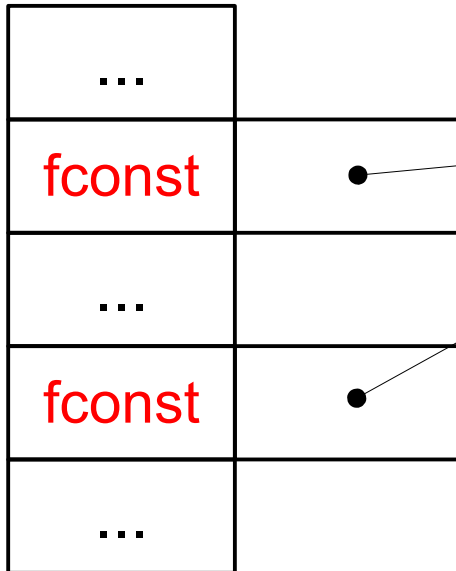
- Код может ссылаться на уникально именованные объекты
  - Такие ссылки называются *символическими*
    - Например, глобальные переменные, классы, их методы (функции)
  - В общем случае эти объекты не константные
  - Могут **перемещаться в памяти**
- Поиск объекта по ссылке называется *разрешением ссылки*
- Ссылку можно разрешить во время выполнения посредством рефлексии, но:
  - В языке может не быть рефлексии
  - Имена фиксированы в коде, неэффективно повторять разрешение одних и тех же имен
    - За исключением языков с *ленивым разрешением ссылок*, где повторное разрешение может выдать другой результат
  - Например, объект может быть создан позднее

# Символические ссылки (2)

- Пусть ссылки разрешает линковщик
  - Замена ссылки на символическое имя адресом найденного объекта
- Представление символических ссылок в коде
  - Непосредственный операнд инструкции
    - Инструкция определяет смысл операнда
    - Битовая ширина операнда машинно-зависима
    - Для нахождения всех ссылок на объект (например, при сборке мусора) нужно просканировать весь код
    - Если это критично, запомним такие инструкции в таблице
  - «Константы» в литеральном пуле
    - В его отдельной части, или специального типа, или просто отличающиеся по значению от обычных констант
    - Могут изменяться при перемещении объектов в памяти

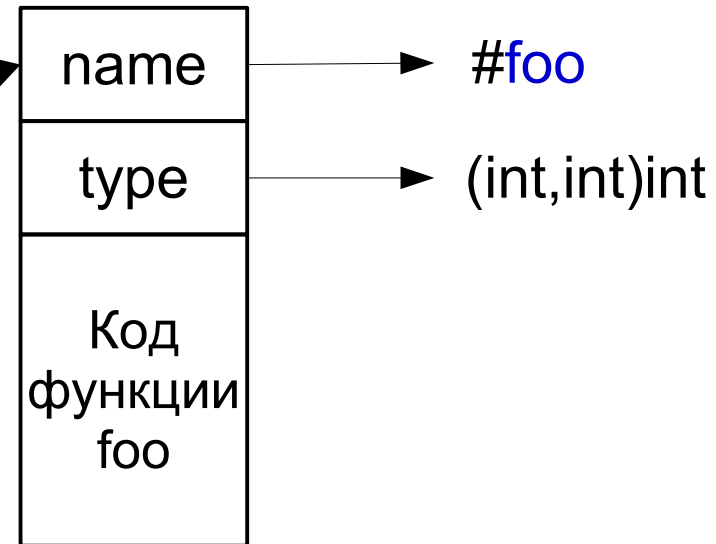
# Непосредственные СИМВОЛИЧЕСКИЕ ССЫЛКИ

Код

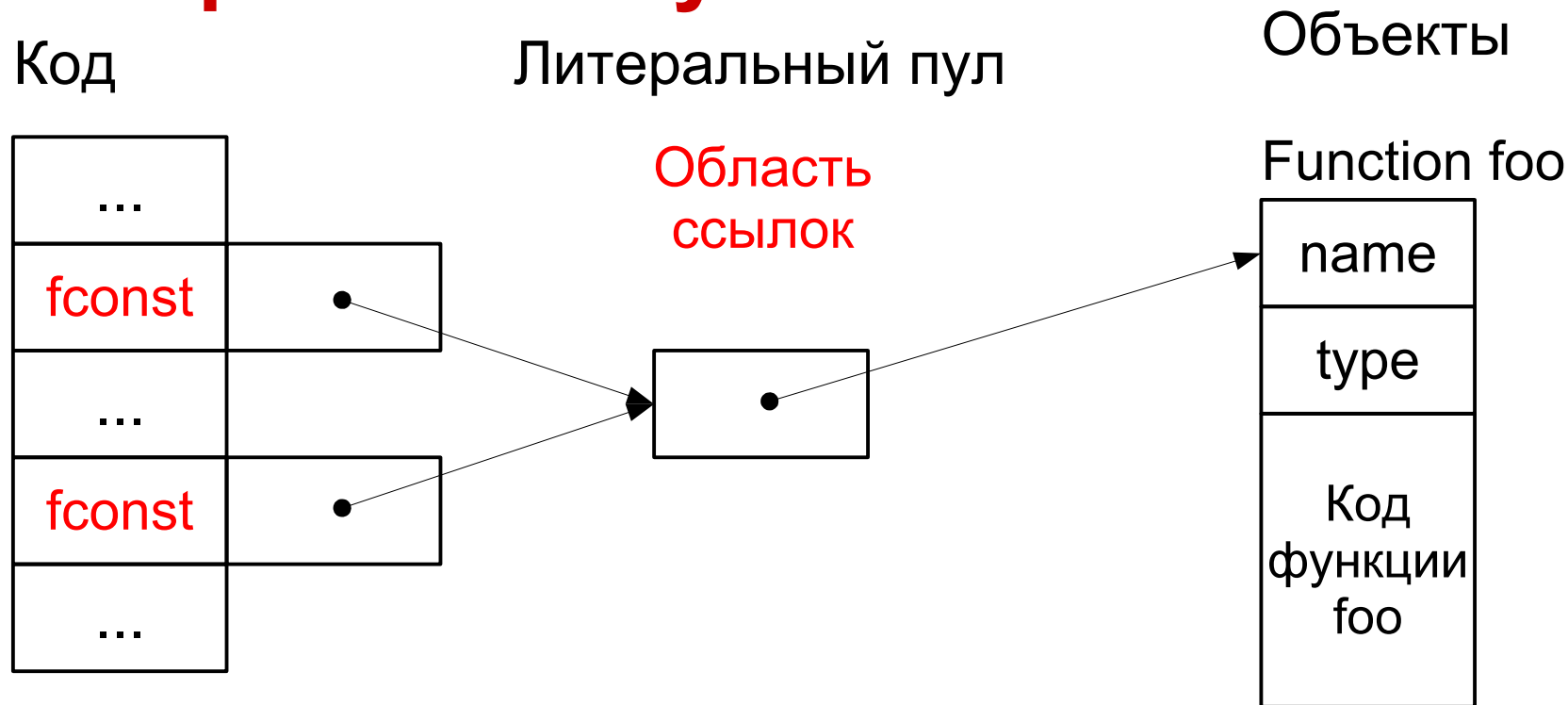


Именованные объекты

Function **foo**



# Символические ссылки в литеральном пуле



- Символические ссылки в литеральном пуле
  - Хранятся в отдельной области
  - Или отличаются диапазоном значений от других ссылок
  - Или разрешаются до выполнения кода
  - Иначе снабжены специальным **тегом** для различения имени объекта и его адреса

# Пример: Байтовый код JVM

- стек-ориентированный набор из  $\approx 200$  инструкций
- Аналогичен коду Smalltalk VM
  - В Java примитивные типы данных не являются объектами
- Четыре виртуальных регистра
  - ip, sp, fp, lp
- По одному стеку вызовов для каждого потока выполнения (thread)
- Стек операндов — часть секции активации
  - Компилятор в виртуальный код должен вычислить требуемую глубину стека операндов
  - Верификатор проверяет достаточность глубины стека операндов

# Пример: Байтовый код JVM (2)

- Большинство инструкций однобайтовые
  - Операнды на вершине стека операндов
- У некоторых инструкций 8- и 16-битовые непосредственные операнды
- Три сложных инструкции переменной длины
  - Длина инструкции вычисляется по операндам
  - *lookupswitch*, *tableswitch*, *wide*
- Несколько кодов зарезервировано для внутреннего использования VM



# Пример: Байтовый код Self

- Всего 8 инструкций:
  - SELF
  - LITERAL <value index>
  - NON-LOCAL RETURN
    - Return from the lexically enclosing method
  - DELEGATEE <parent slot index>
    - Set parent for message delegation
  - SEND <message name index>
  - IMPLICIT SELF SEND <message name index>
  - RESEND
  - INDEX-EXTENSION <index extension>
- 3 бита — код операции, остальные 5 — параметр
- Большинство примитивов вызывается с помощью обычной посылки сообщений

# Записи активации (секции стека)

# Виды записей активации (секций стека)

- Запись активации может быть *фиксированной* или *переменной* длины
  - Размер записи фиксированной длины определяется вызываемой функцией, одинаков для всех ее активаций и иногда называется *аппетитом* данной функции
  - Размер активации переменной длины может меняться в течение ее жизни
    - Например, если в ней можно отводить локальную память
    - Или она заканчивается стеком операндов
- У разных функций могут быть разные виды и разные форматы секций активации

# Секции стека фиксированного размера

- Пролог

```
SP -= LocalSize(foo);
```

- Эпилог

```
SP += LocalSize(foo);
```

- Адресация параметров и локальных переменных

```
reg = [SP+offset];
```



# Секции стека переменного размера

- Пролог

```
push(FP);  
FP = SP;  
SP -= LocalSize(foo);
```

- Эпилог

```
SP = FP;  
FP = pop();
```

- Адресация параметров

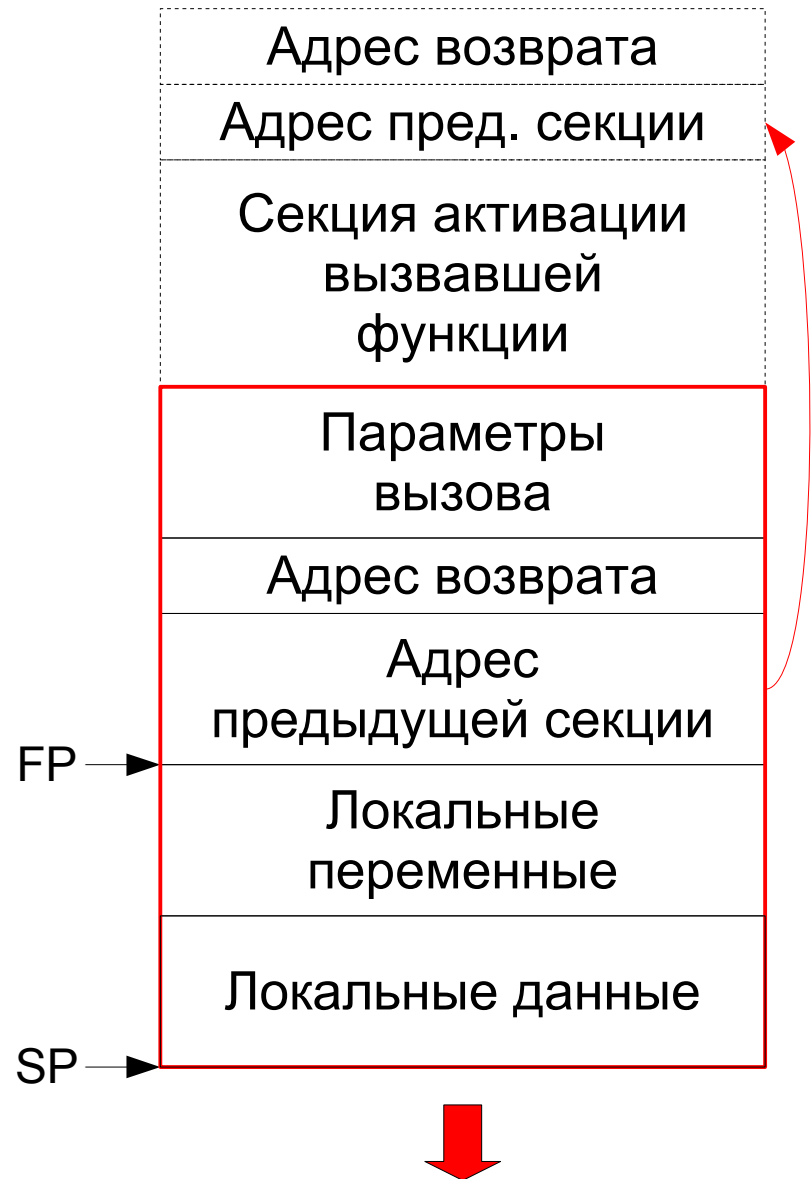
```
reg = [FP+offset];
```

- Адресация локальных переменных

```
reg = [FP-offset];
```

- Отведение локальной памяти

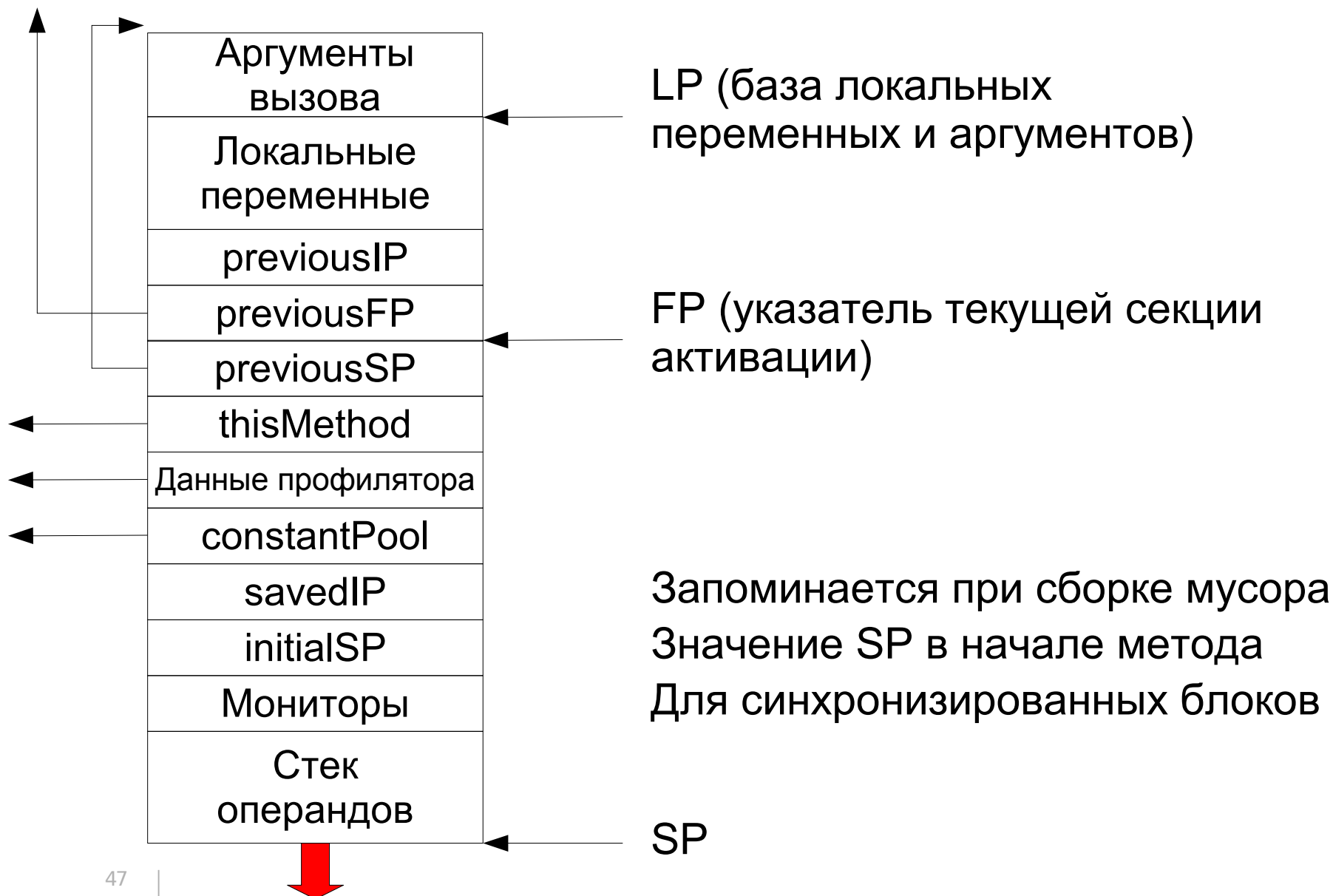
```
alloca(size);
```



# Стек вызовов и записи активации

- Smalltalk и Self: стека вызовов нет
  - Дисциплина вызовов может отличаться от LIFO
  - Записи активации — обычные объекты, отводятся в «куче»
- Forth: стек вызовов есть, записей активации нет
  - Запись активации редуцирована до адреса возврата
  - Поэтому стек вызовов называется стеком возвратов
- Java: свой стек вызовов у каждого потока
  - Запись активации создается при каждом вызове и сбрасывается при каждом возврате
  - Запись активации содержит:
    - аргументы вызова
    - место для локальных переменных
    - место для стека операндов
    - служебные данные

# Пример: Запись активации интерпретатора OpenJDK



# Организация стека вызовов

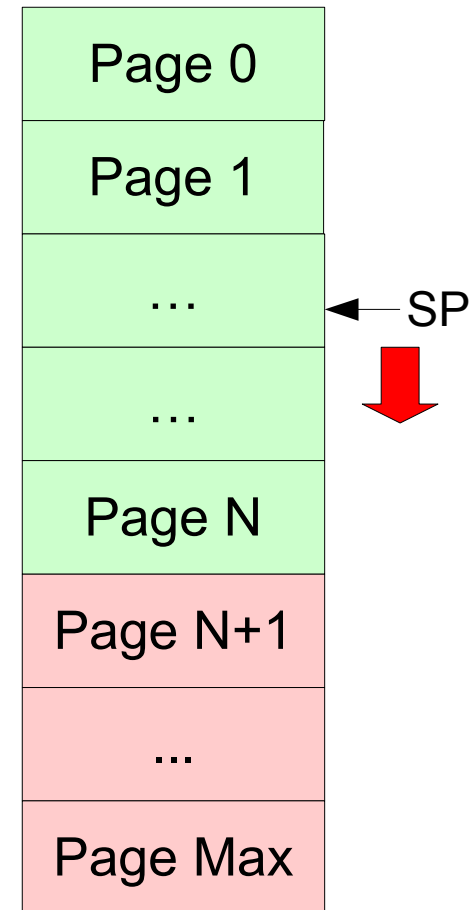


# Организация стека вызовов

- Реализация VM ограничивает
  - Максимальную глубину стека вызовов
    - За исключением очень простых случаев требуемая для выполнения программы глубина не может быть вычислена статически
  - Максимальный размер записи активации
    - Проверяется верификатором до выполнения кода
- Варианты реализации
  - Максимальное резервирование памяти
    - Неэффективно, так как обычно глубина стека существенно меньше максимальной
    - Используется лишь небольшая часть отведенной памяти
  - Максимальное резервирование адресного пространства
  - Кусочный неподвижный стек переменной длины
  - Непрерывный перемещаемый эластичный стек

# Максимальное резервирование адресного пространства

- При наличии страничной адресации зарезервируем адресное пространство
  - Отводим страницы с запасом на одну максимальную секцию
  - По мере необходимости отображаем память
  - Современные ОС поддерживают растущие **вниз** стеки на уровне ядра
    - При резервировании диапазона адресов достаточно указать, что это стек
  - Сокращение размера стеков возможно, но ОС не предоставляет для этого специальной поддержки
    - Заметив, что большая часть стека не используется, VM может вернуть эту память ОС



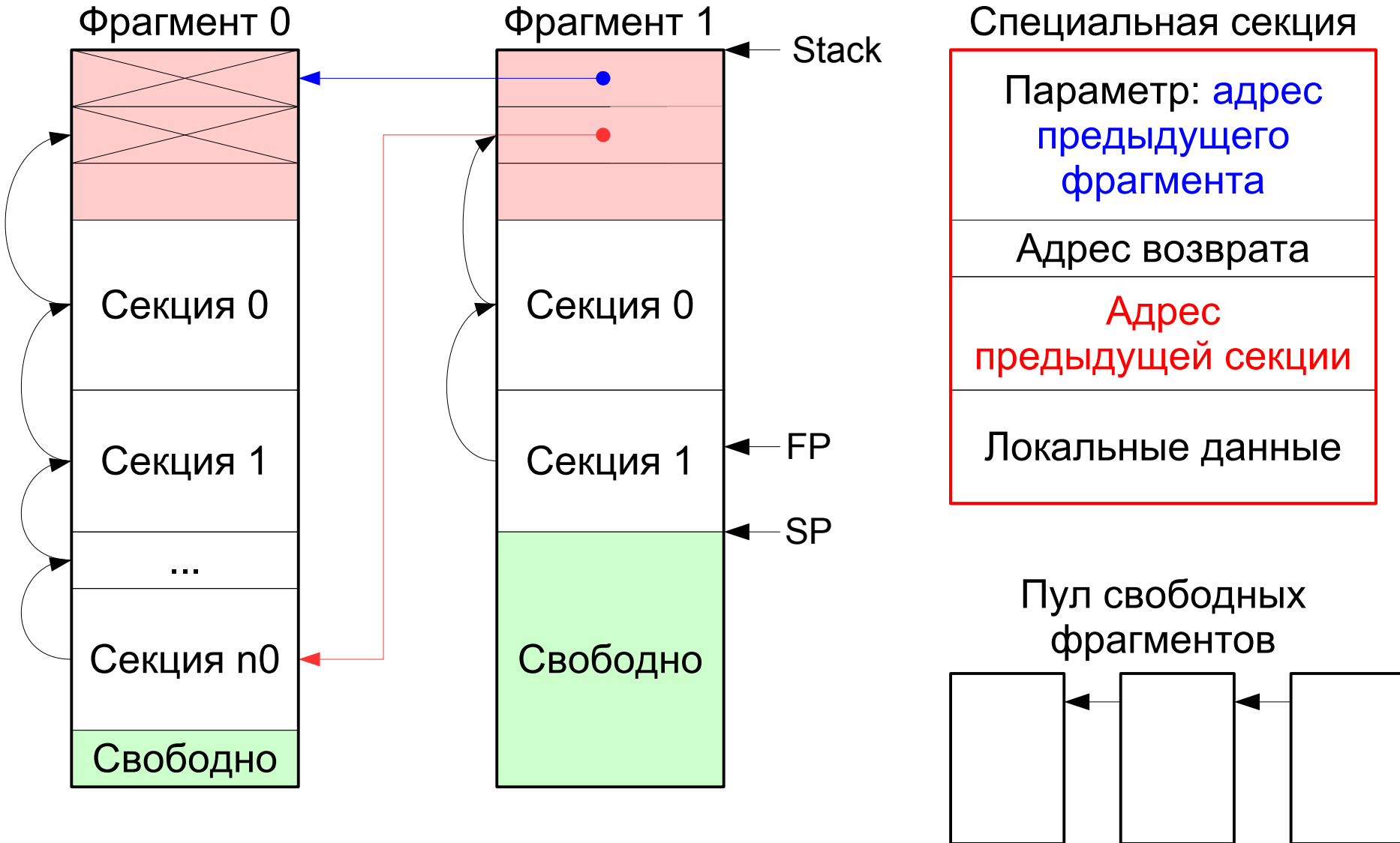
# Недостатки максимального резервирования адресного пространства

- Адресное пространство не бесконечно
  - Ограничения процессора, чипсета и ОС
  - В некоторых языках стеков может быть много
    - Например, каждой сопрограмме нужен свой стек - обычно очень маленький, но максимальный размер не известен
    - В некоторых приложениях могут одновременно существовать сотни тысяч сопрограмм
- Медленный вызов ОС
  - Найти подходящий фрагмент адресного пространства и выставить атрибуты у всех его страниц
    - Можно завести пул зарезервированных фрагментов популярных размеров
- Увеличение расхода памяти на таблицу страниц

# Кусочный неподвижный стек

- Делим стек на фрагменты одинакового размера
  - Не менее максимальной секции активации
- При переполнении стека отводим новый фрагмент
  - Проверка переполнения обычно программная — жалко **каждый** фрагмент дополнять защищёнными страницами по максимальному размеру секции активации
- Для ускорения отведений и освобождений фрагментов заводим для них **пул**
  - Освобождение и отведение фрагмента из пула требует всего нескольких присваиваний
- При выходе из фрагмента возвращаем его в пул
  - Для этого при отведении фрагмента снабдим его искусственной секцией активации
    - При ее выполнении освобождаем текущий фрагмент
    - Переключаемся на предыдущий фрагмент и предшествующую запись активации

# Кусочный неподвижный стек (2)



В многопоточной реализации у каждого потока свой пул.

# Непрерывный перемещаемый эластичный стек

- Проверка переполнения программная или с использованием защиты страниц
  - Программная проверка не требует знания максимальных размеров секции и стека
- При переполнении копируем стек в область памяти большего размера
  - Например, удваиваем размер
- При нехватке памяти в VM сокращаем размеры стеков
  - Копируем в области памяти меньшего размера
- Перемещение стека может требовать **релокации** содержащихся в нем адресов
  - Например, ссылок на предыдущую секцию в динамической цепочке
  - Необходимо знать формат **каждой** секции стека

# **Нативный интерфейс VM: Вызов библиотечных функций**

# Отступление: типизация языков программирования (1)

- В программах на языках программирования случаются ошибки
- Ошибка может быть обнаружена:
  - *Статически* — до начала выполнения программы
    - Во время компиляции, линковки или верификации
  - *Динамически*
- Среди всех ошибок могут выделяться *типовые ошибки (ошибки несоответствия типов)*.
  - В этом случае язык называется *типизированным*, иначе *бестиповым*

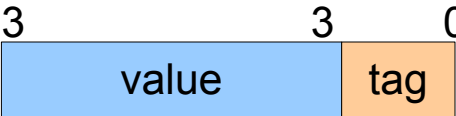


# Отступление: типизация языков программирования (2)

- Типизированные языки классифицируются по времени обнаружения типовых ошибок
  - Со *статическим контролем типов (статически типизированный)* — все типовые ошибки обнаруживаются статически
  - С *динамическим контролем типов (динамически типизированный)* — все типовые ошибки обнаруживаются динамически
  - Иначе это язык со *смешанным контролем типов*
- У динамически типизированных и бестиповых языков с классами много общего
  - Объект принадлежит классу
  - Это отношение неизменно
  - Класс определяет семантику операций над объектом

# Отступление: FixNum'ы

- Популярный способ представления значений в реализациях бестиповых и динамически типизированных языков
- Впервые появились в реализации языка *Scheme* в 1970х
  - Dorai Sitaram. *Teach Yourself Scheme in Fixnum Days*, 1998
- *FixNum* — тегированное значение, занимающее одно слово памяти
- Например, для 64-битовых слов
  - Объекты выровнены в памяти на границу слова
  - 3 младших бита адреса нулевые
  - Запишем в них тег


  - **Ref**, Bool, Byte, Short, Int, Long, Float, Double
  - В long теряем 3 старших бита, в double — 3 младших бита мантиссы (при переполнении можно мигрировать значение в кучу - тег **Ref**)

# Внешние (библиотечные) функции

- Необходимы для связи с ОС и библиотеками
  - Например, чтение из файла или форматный вывод
- Особенности внешних функций
  - Написаны на другом языке
    - Обычно C, C++, asm. Вызовы на прочих языках требуют полиглотных VM
    - Мы их не можем или не хотим конвертировать в функции на нашем языке
  - В них другие типы данных
    - Указатели, объединения, структуры, массивы
  - Другие соглашения о связях
    - Как и в каком порядке передаются параметры, кто их снимает со стека
    - Кто и какие регистры процессора сохраняет
    - Бросает ли исключения, как и какие
    - Форматирование имени (name mangling)
  - Другое использование стека
    - Направление роста, проверка переполнения, устройство секции

# Адаптация вызова внешних функций

- Из-за разницы между внутренними и внешними функциями их вызов требует **адаптации**
- Адаптация вызова
  - Вычисление идентификатора вызываемой функции по имени, типам аргументов и соглашению о связях
  - Преобразование параметров в нативный вид
  - Сохранение регистров
  - Возможно, переключение на нативный стек
    - Если реализация стека не совместима с нативной
  - Нативный вызов в соответствии с соглашением о связях
  - Ловля возможных программных исключений, брошенных вызванной функцией
  - Преобразование результата или исключения из нативного вида в языковой
  - Возможно, обратное переключение стека
  - Восстановление регистров

# Способы адаптации вызовов внешних функций

- Отдельная инструкция интерпретатора
- Нативный адаптер для каждой внешней функции
- Адаптеры на ограниченном низкоуровневом расширении языка

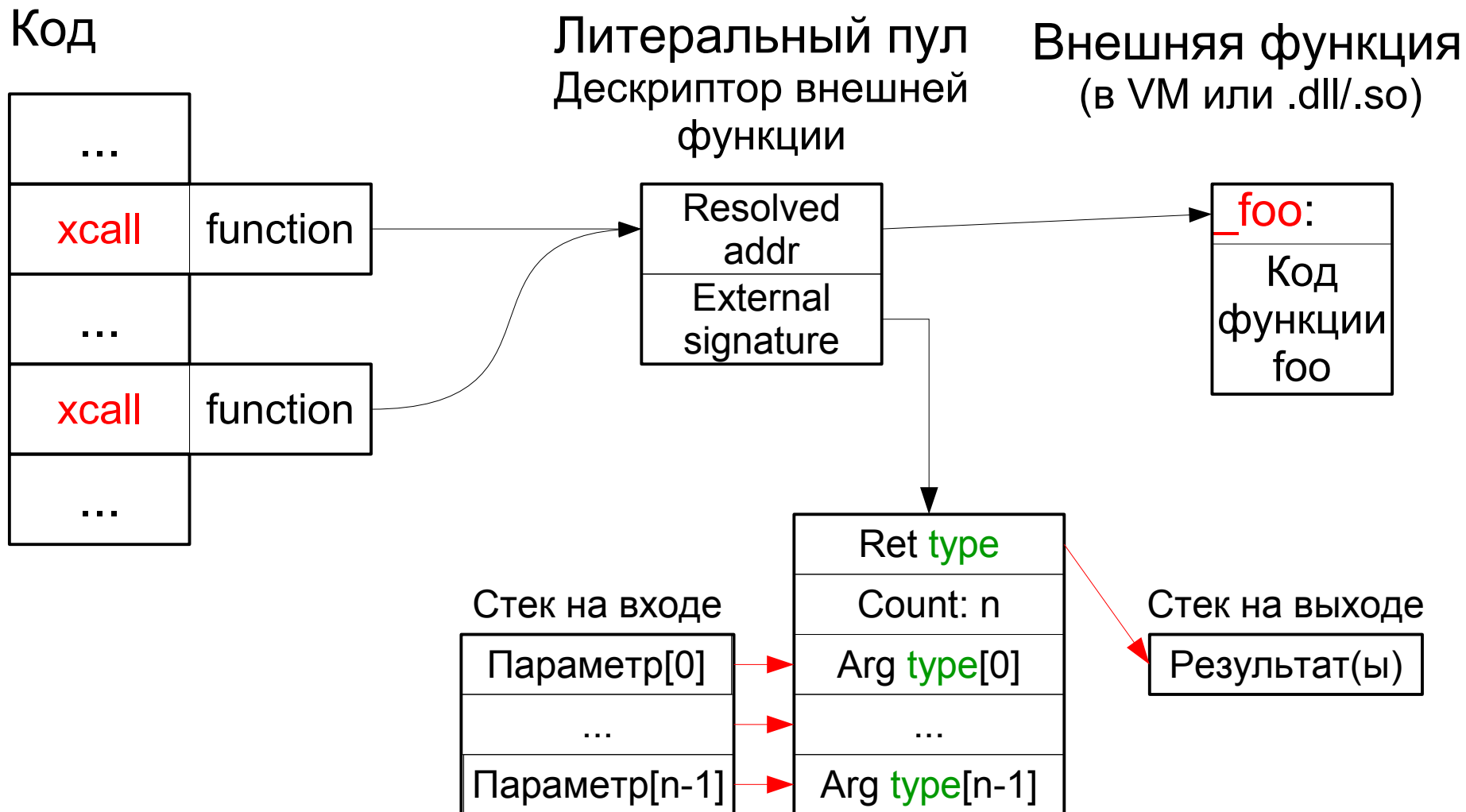
# Вызов внешних функций инструкцией интерпретатора (1)

- Интерпретатор выполняет **универсальную адаптацию** отдельного вызова
  - Инструкция вызова непосредственно параметризована дескриптором внешней функции
    - Дескриптор функции расположен в литеральном пуле и содержит её имя и внешнюю сигнатуру
  - Параметры вызова лежат на стеке
  - Перед вызовом интерпретатор сканирует сигнатуру и преобразует очередное значение на стеке в позиционно соответствующий ему тип сигнатуры
  - Это преобразование определяется парой типов — **внутренним** (значения) и **внешним** (аргумента функции)
    - В динамически типизированном языке каждое значение знает свой тип
    - В статически типизированном языке внутренний тип должен однозначно вычисляться по внешнему
    - В смешанно типизированных языках комбинация этих подходов в зависимости от внешнего типа

# Вызов внешних функций инструкцией интерпретатора (2)

- После вызова интерпретатор преобразует результат из внешнего типа во внутренний
  - Извлекает из сигнатуры **внешний** тип результата
  - Вычисляет про нему **внутренний** тип
  - Производит преобразование полученного результата из внешнего типа во внутренний
- Преобразования типов «защиты» в интерпретатор
  - Если интерпретатор не знает преобразования, это ошибка
  - Не все внешние функции можно таким образом вызвать
  - Можно придумать механизм расширения, но он **опасен** и потому вряд ли будет доступен приложению

# Вызов внешних функций инструкцией интерпретатора (3)





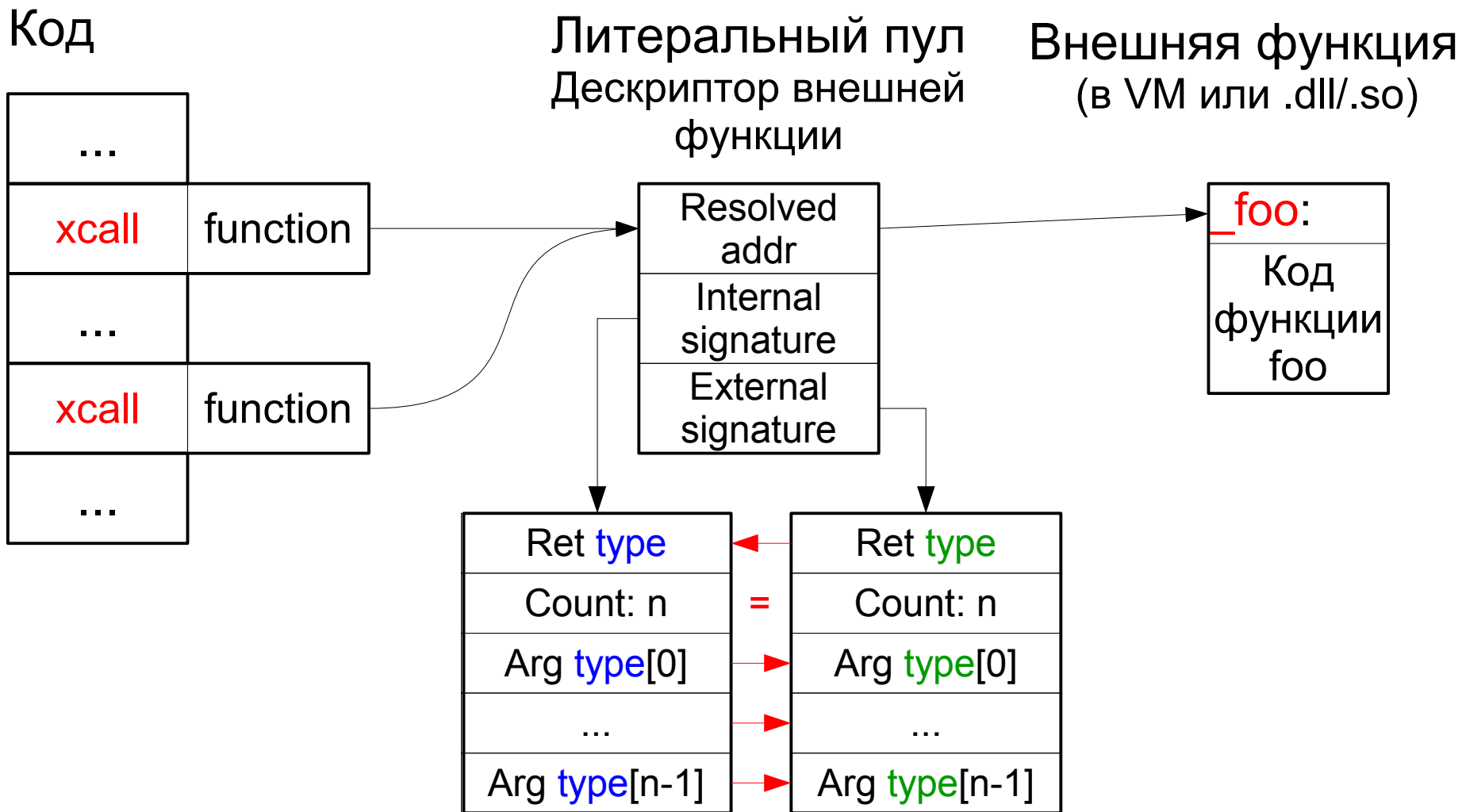
# Вызов внешних функций инструкцией интерпретатора (4)

- Зашитое в интерпретатор преобразование типов не всегда удобно

```
void* memset(void* dest, int c, size_t n)
```

  - Хотелось бы передать байт
  - Во внутренней системе типов байт не обязательно автоматически преобразуется в целое
- Это преобразование может зависеть от платформы
  - Разрядность внутренних типов может быть фиксирована стандартом языка
  - Разрядность внешних типов может меняться
- Иногда удобно внутреннюю сигнатуру не вычислять, а задавать явно

# Вызов внешних функций инструкцией интерпретатора (5)



- Во внутренней сигнатуре **внутренние** типы, во внешней - **внешние**

# Вызов внешних функций инструкцией интерпретатора (6)

- Не все типы легко конвертируются
  - Разная битовая ширина (long и double в fixnum)
  - Структуры (поля должны следовать в том же порядке и иметь тот же размер и смещение)
  - Строки (в разных языках разные представления)
- Легче конвертировать, если внешняя функция не может вызвать перемещение объектов в куче
  - Тогда можно использовать обычные указатели на поля объектов и элементы массивов
  - Для этого необходимы:
    - нативно однопоточная VM
    - отсутствие callback'ов
    - запрет на отведение памяти из внешней функции при помощи сервисных функций VM

# Вызов внешних функций инструкцией интерпретатора (7)

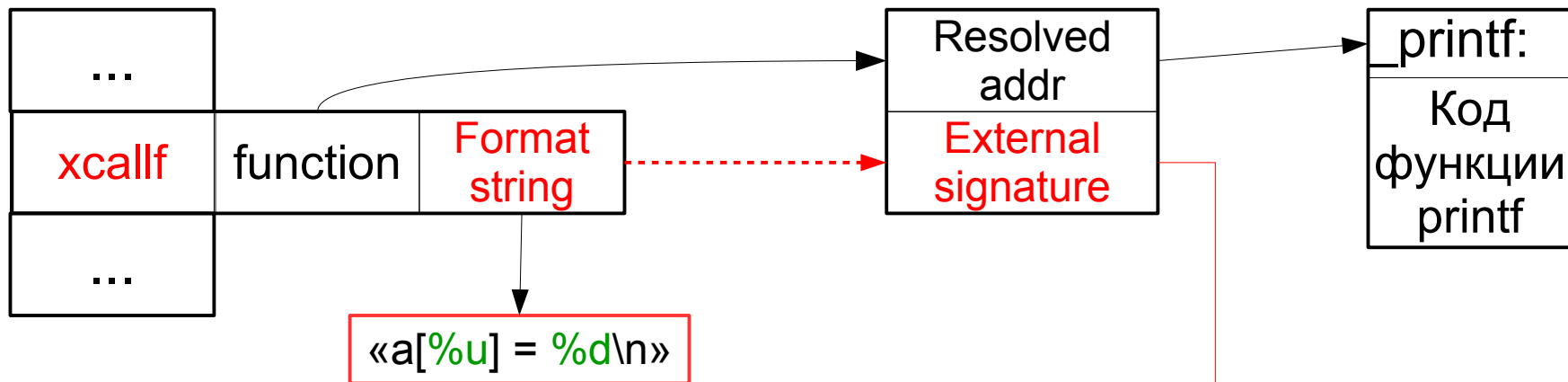
- Отдельная трудность — функции с переменным числом параметров (`printf`)
  - Чтобы узнать внешние типы, интерпретатор должен разобрать форматную строку
  - Еще сложнее верификатору — он должен это сделать до запуска программы
  - К счастью, форматная строка обычно константа времени компиляции
  - Пусть наш умный **линковщик** не только разрешит ссылку на внешнюю функцию, но и **вычислит внешнюю сигнатуру** по форматной строке
  - Внутренняя сигнатура вычисляется по внешней
- Общего решения нет
  - По существу форматная строка — это интерпретируемая программа на другом языке
  - Но можно завести **отдельную инструкцию** вызова функций с переменным числом параметров с **известным синтаксисом форматной строки**

## инструкцией интерпретатора (8)

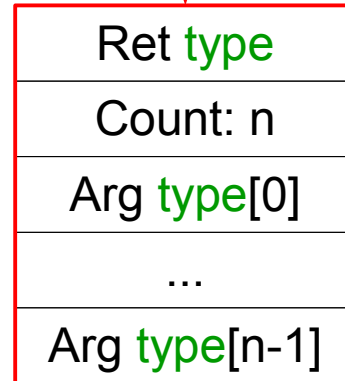
## Код

# Литеральный пул

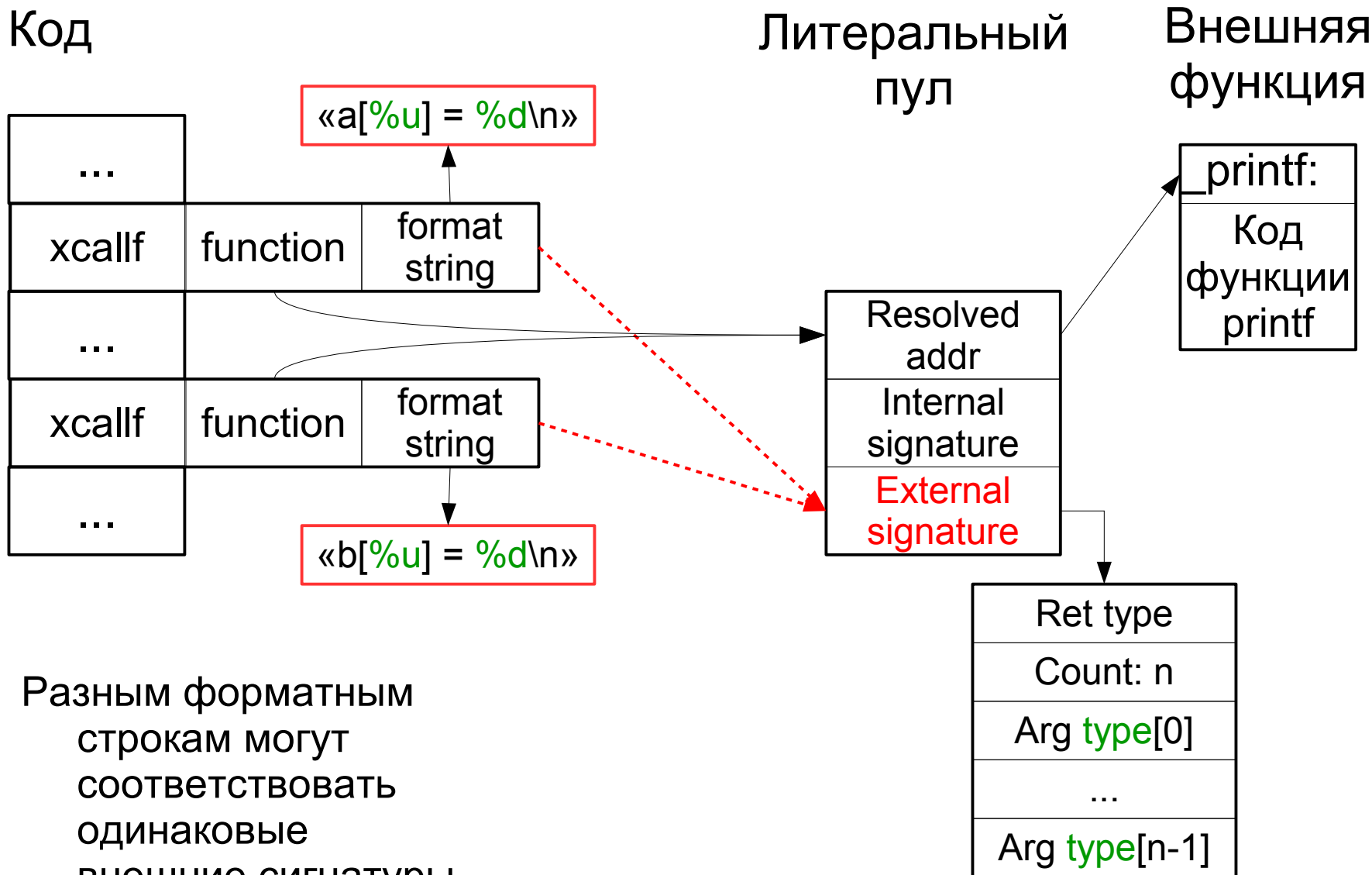
## Внешняя функция



- Линковщик вычисляет внешнюю сигнатуру вызова по его параметру - форматной строке
- Верификатор для проверки соответствия внутренних типов параметров вызова форматной строке вычисляет внутреннюю сигнатуру по внешней



# Вызов внешних функций инструкцией интерпретатора (9)



- Разным форматным строкам могут соответствовать одинаковые внешние сигнатуры

# Вызов внешней функции из нативного адаптера (1)

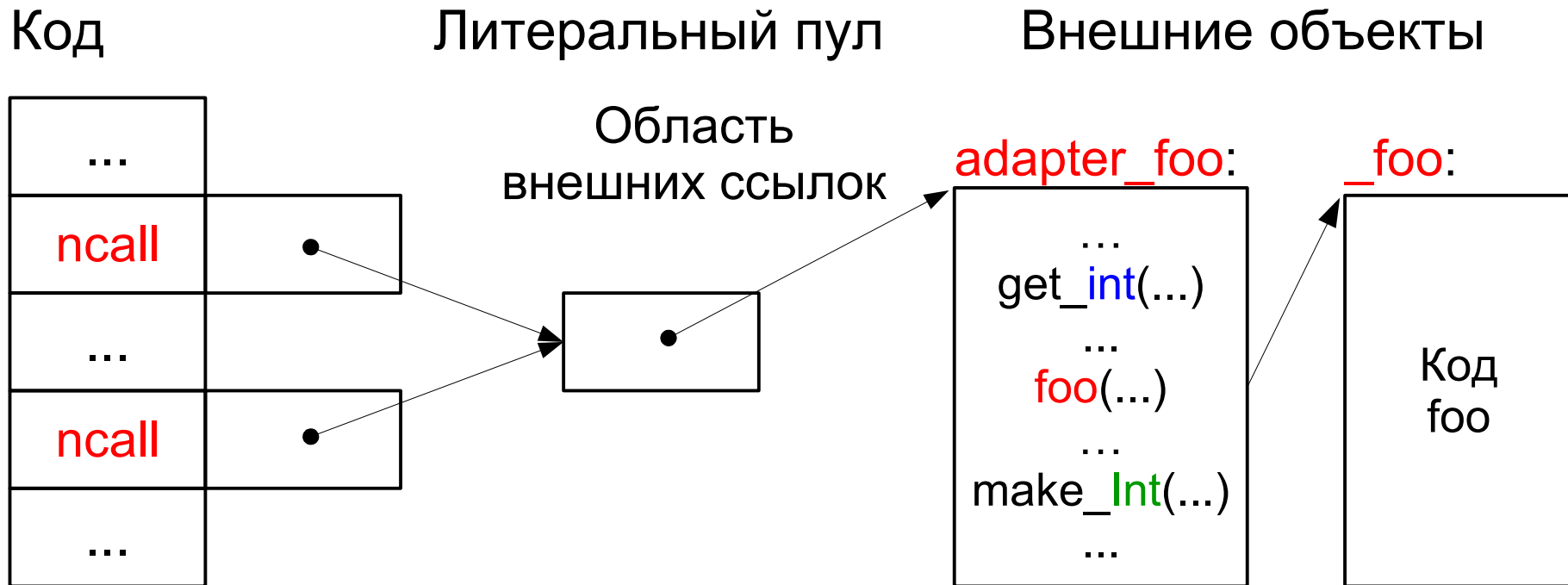
- Интерпретатор умеет вызывать нативный код
  - Отдельной инструкцией вызова нативного кода или обычной инструкцией с флагом в заголовке функции
  - Этот вызов выполняет все адаптации, за исключением преобразования типов параметров и результата
  - Могут быть специальные соглашения о связях для облегчения вызова и возврата
    - Например, параметры и результат остаются на стеке параметров, а не перекладываются на нативный стек
    - Например, нативный код должен положить результат на стек параметров и закончиться не возвратом, а переходом в интерпретатор
- VM обеспечивает набор нативных функций:
  - Доступ к стеку и полям объектов, создание объектов, бросание исключений...
  - Преобразования внутренних типов значений во внешние и наоборот

# Вызов внешней функции из нативного адаптера (2)

- Нативный адаптер
  - Вызовами этих функций преобразует параметры из внутренних типов во внешние
  - Вызывает внешнюю функцию
  - Ловит исключения
  - Преобразует полученный результат или пойманное исключение из внешних типов во внутренние
  - Возвращает результат или бросает исключение
  - Этот код скомпилирован
    - Не нужно во время выполнения в цикле сопоставлять внутреннюю и внешнюю сигнатуры и выбирать преобразования
  - Корректность кода нативного адаптера не проверяется верификатором
- Внутренняя сигнатура вызова — это сигнатура самого адаптера
  - Она используется при верификации вызова



# Вызов внешней функции из нативного адаптера (3)



- Каждой внешней функции нужен свой адаптер
  - В простых случаях исходный код адаптеров можно породить автоматически (импорт заголовков)

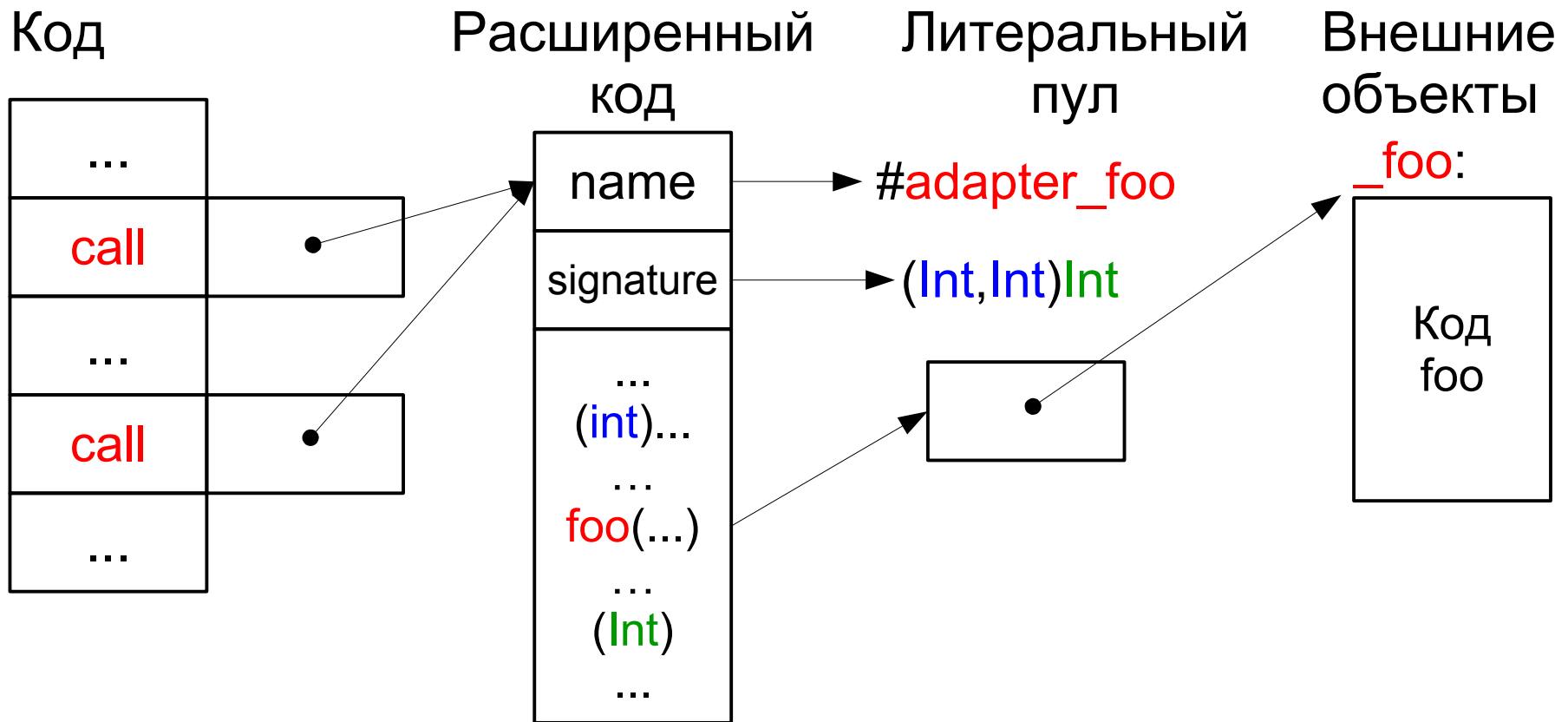
## низкоуровневое расширение языка (1)

- Расширим язык низкоуровневыми типами
  - Современные ABI (Application Binary Interface) определяются в типах языка C
    - Типы данных и функции системных библиотек
- Разрешим использовать это расширение только в синтаксически ограниченных контекстах
  - Низкоуровневых структурных типах и массивах
  - Низкоуровневых функциях
    - Для них потребуется новый верификатор и компилятор или интерпретатор
- Обычный язык может вызвать функцию расширенного языка, если **в ее сигнатуре нет расширенных типов**

# Ограниченное низкоуровневое расширение языка (2)

- Функция расширенного языка:
  - Может свободно использовать типы и вызывать функции обычного языка
  - Может конвертировать типы обычного языка в низкоуровневые и обратно
    - Как примитивные, так и структурные
  - Может вызывать внешние функции
- Будем порождать или писать адаптеры на этом расширенном языке
  - Код проще и короче
  - Устраняются типичные ошибки
    - Например, регистрация ссылок в «кучу» для GC
  - Ниже накладные расходы
    - Свой компилятор может макроподставить вызов адаптера и оптимизировать его код
  - Верификация адаптеров

# Вызов внешней функции из низкоуровневого расширения



- При удачном построении системы типов расширенного языка в большинстве случаев компилятор сможет сам породить преобразования типов

# Нативный интерфейс VM: Callback-функции

# Отступление: Обработка исключений в Java (1)

```
try {  
    ... Сделать что-нибудь ...  
} catch (IOException e1) {  
    ... Обработка ошибок ввода-вывода ...  
} catch (MyException e2) {  
    ... Обработка определенных пользователем  
        исключений ...  
} finally {  
    ... Гарантированно исполняющийся код ...  
}
```

- Тяжеловесная монолитная конструкция
- При компиляции метода в дополнение к его коду создается таблица try-блоков
  - Диапазон индексов блока
  - Для каждого обработчика класс ловимого исключения
  - ... и индекс обработчика
  - Специальный обработчик непоиманных исключений

# Обработка исключений в Java (2)

- При возникновении исключения секции активации последовательно сканируются от вершины до дна стека (от вызванной секции к вызвавшей)
- Из каждой секции имя вызванного метода и позиция в нем сохраняются в *трассе стека*
- Если метод текущей секции содержит try-блок вокруг заданного адреса с обработчиком подходящего типа исключений, все верхние секции сбрасываются и этот обработчик вызывается с трассой стека как параметром
- Иначе процесс продолжается для вызвавшей секции и ее адреса вызова
- При сбрасывании синхронизированных секций снимаются замки синхронизации

# Callback-функции

- Интерпретируемый код вызывает нативный
- Вызванный нативный код может захотеть вызвать интерпретируемую функцию
  - Эта функция может быть известна по имени
  - Или ее адрес передан параметром
- Такие функции называются *коллбэками*
- Особенности реализации коллбэков:
  - Нужен возврат в нативный код, а не в интерпретатор
  - Вызов должен быть адаптирован
    - Аналогично вызову внешних функций, но в обратную сторону
    - Адаптер может быть написан в нативном коде или в ограниченном низкоуровневом расширении
  - Необходимо взаимно-однозначное соответствие между нативными стеками и потоками VM
  - Необходимо уметь **перебрасывать** исключения
    - Иначе трасса стека теряется при переходе



# Реализация возврата в нативный код (1)

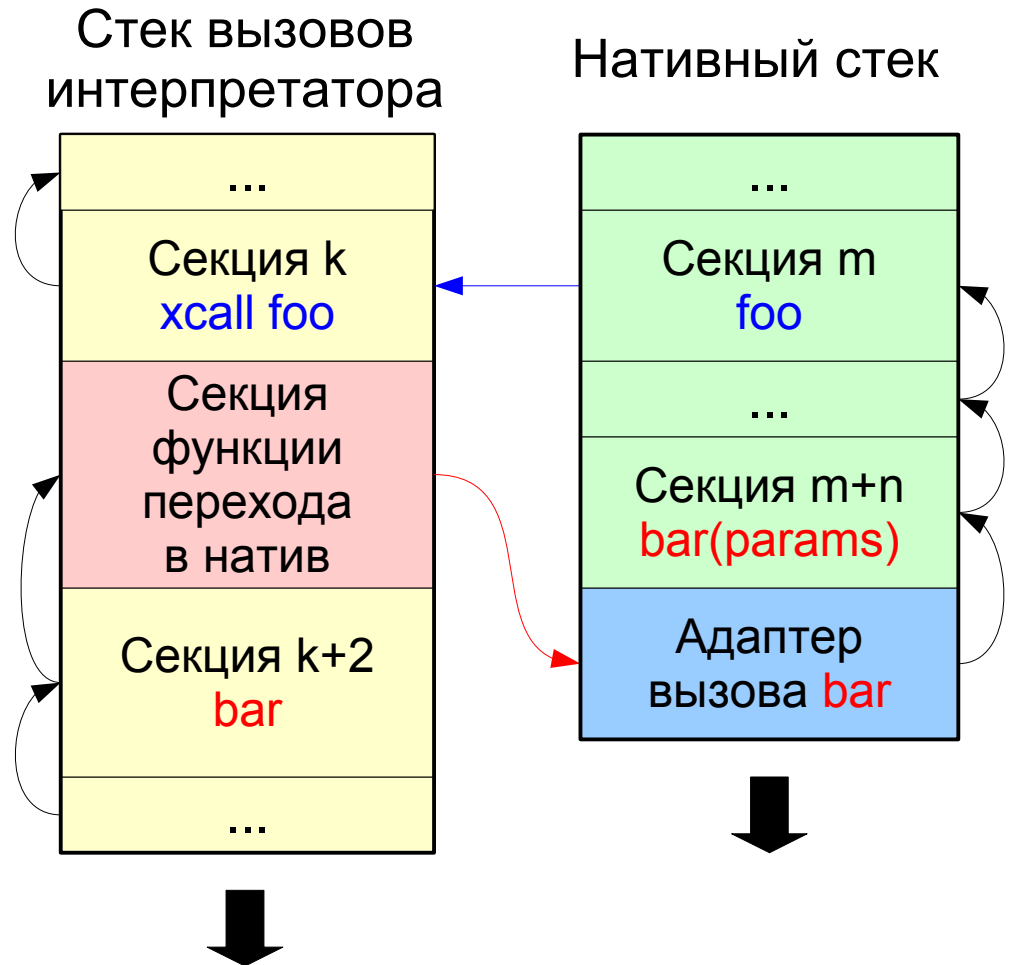
- Интерпретируемая функция может быть вызвана из интерпретируемого и нативного кода
- Инструкция возврата из этой функции сама не может знать, **куда** происходит возврат
- Нужен промежуточный вызов *функции перехода* из нативного кода в интерпретируемый
  - Вызывается из нативного кода адаптера вызова
  - Получает параметрами интерпретируемую функцию для вызова и набор параметров для ее вызова
  - Производит вызов
  - Ловит исключения
    - Адаптер не может этого делать, потому что форматы интерпретируемой и нативной секции различны
    - При бросании исключения ищется секция интерпретируемого стека с обработчиком
  - Возвращается в нативный код адаптера вызова с результатом вызванной функции или исключением

# Реализация возврата в нативный код (2)

- В бестиповых и динамически типизированных языках достаточно единственной функции перехода
- В статически и смешанно типизированных языках нужна отдельная функция для **каждой сигнатуры** интерпретируемой функции, вызываемой из натива
- К счастью, адаптер может симулировать эффект вызова функции перехода вопреки системе типов языка
  - Создать и правильно инициализировать секцию перехода на стеке вызовов интерпретатора
  - Достаточно отдельной функции перехода для **каждого типа возвращаемого значения**

# Реализация возврата в нативный код (3)

- Интерпретатор в секции  $k$  некоторой функции вызывает внешнюю функцию  $foo$
- Нативная функция  $foo$  или какая-либо вызванная из нее нативная функция в секции  $m+n$  вызывает нативный адаптер интерпретируемой callback-функции  $bar$
- Адаптер преобразует параметры, создает секцию перехода, вызывает интерпретируемую функцию  $bar$
- При возврате из  $bar$  секция перехода получит результат или поймает исключение, переключится на нативный стек и вернется в адаптер



# Реализация возврата в нативный код (4)

- Как передать управление из интерпретируемого кода в нативный?
- В байткоде:
  - Завести **инструкции** возврата в натив с выдачей результата и перебрасывания в натив исключения
  - Или нативные функции с аналогичным эффектом
- В подпрограммном шитом коде:
  - Произвольный машинный код вместо очередного вызова
- В прямом шитом коде:
  - Адрес следующего слова, затем произвольный машинный код по этому адресу

# Нативный интерфейс VM: Доступ к данным

# Доступ к данным VM из нативного кода

- Динамическая нативная рефлексия
  - Неэффективна, зато всегда работает
  - Требуется хранения иначе не нужных имен
    - Они могут занимать существенный объем памяти
  - Затрудняет анализ ссылок из нативного кода
- Статическая структурная нативная рефлексия
  - Эффективна, но применима только к замкнутой модели мира
  - Не всем структурам VM можно найти нативный аналог
    - Например, поля по обе стороны заголовка объекта
- Эти способы можно комбинировать
  - Статическая рефлексия для статически известной части Вселенной
  - Динамическая для всего остального
    - Например, динамически загруженных классов
    - В частности, определенных пользователем в процессе выполнения программы

# Динамическая нативная рефлексия

- Позднее связывание по имени (строке или символу)
  - Символ (атом) — зарегистрированная строка
    - При повторной регистрации выдается тот же адрес
  - Поиск по адресу строки быстрее поиска по ее значению
  - Но нужно где-то запомнить символ

```
ClassHandle klass = find_class(«MyClass»);  
FieldHandle field = klass.find_field(«myField»);  
int value = field.read_int(obj);
```

# Статическая структурная нативная рефлексия

- Статически порождаем заголовочные файлы с нативными структурами, эквивалентными представлению объектов в памяти
  - **struct** в C
  - **class** с аксессорами полей в C++
    - Аксессоры удобнее — в них можно завернуть семантики, навешанные на доступ к полям определенных типов
    - Например, сборщику мусора могут быть интересны изменения ссылочных полей

```
class MyClass: Object { // Наследуем заголовок
    int _myField;
public:
    int myField() const { return _myField; }
    int setMyField(int val) { _myField = val; }
};
```



# Домашнее задание №2

- Написать простой итеративный интерпретатор стековой машины языка *Lama*
  - Проверить правильность на тестах
  - Сравнить по производительности с существующим рекурсивным интерпретатором
  - Исходный код, тесты корректности и производительности: <https://github.com/JetBrains-Research/Lama>
  - Компилятор содержит в себе рекурсивный интерпретатор и запускает его с опцией *-i*
  - Компилятор порождает байткод с опцией *-b*
  - Байтовый код стековой машины см. в *src/SM.ml*
  - Дизассемблер байткода см. в директории *byterun*
  - **Подсказка:** сборщик мусора считает, что границы стека операндов [`__gc_stack_top`, `__gc_stack_bottom`)

# Ускорение интерпретации

# Производительность интерпретатора

- По разным исследованиям интерпретатор в 2.5-50 раз медленнее соответствующего нативного кода, порожденного оптимизирующим компилятором
  - Менее предсказуем
  - При той же работе потребляет больше энергии
- Замедление определяется в основном реализацией VM, а не приложением и не компилятором в виртуальный код
  - Повышением производительности в основном должен заниматься разработчик VM

# Накладные расходы интерпретатора

- Декодирование виртуальных инструкций
  - Косвенный переход к множеству меток в `switch(*ip++)` плохо совместим с предсказанием переходов в современных процессорах (макрос `next` чуть лучше)
- Доступ к виртуальным регистрам
  - Обычная переменная или элемент массива в памяти
- Вызов примитивов
  - Распаковать операнды (например, `Int`  $\rightarrow$  `int`)
  - Сохранить состояние интерпретатора
  - Вызвать функцию на отдельном нативном стеке
  - Восстановить состояние
  - Заpackовать результат, возможно, бросить исключение
- Лишние проверки при выполнении инструкций
  - Инициализированность классов, проверка на `null...`
  - Контекст интерпретатора **слишком узкий**, чтобы их избегать

# Ускорение интерпретации

- Шитый код декодируется быстрее байтового
  - Но метки менее удобны в обработке, чем токены
    - Например, можно написать switch по токенам, но нельзя по меткам (адресам функций)
- Переписать цикл и критические инструкции на ассемблере
  - Ускорение в  $\approx 2$  раза
  - Небольшое сокращение размера
  - Выше трудоемкость разработки и сопровождения
  - Ухудшение портируемости и безопасности
    - Распространенные статические анализаторы сдаются при виде ассемблерного кода
- Если виртуальных регистров немного, отобразить их в физические
  - На ассемблере или при помощи специальных директив компилятора
  - Ухудшение портируемости

# Вынесение редко используемых сложных инструкций из цикла интерпретатора

- Сложным инструкциям нужно больше регистров
- От этого может пострадать распределение регистров в реализациях простых инструкций
- Простые инструкции выполняются гораздо чаще, но компилятор об этом не знает
  - Умный компилятор может знать при наличии **профиля выполнения** программы
- Вынесем сложные инструкции в отдельную функцию
- Вызовем ее в ветке *default*
  - Распределение регистров может улучшиться

# Кэширование стека операндов

# Ассемблерный интерпретатор байткода без кэширования стека

```
next: macro() // 1 обращение к памяти, 1 косвенный переход
    tmp = *ip++;
    jmp bytecode_table [BytesInWord * tmp];
endm
```

```
L_iadd: // 3 обращения к памяти, 1 регистровая операция
    pop tmp1;
    pop tmp2;
    tmp1 += tmp2;
    push tmp1;
    next();
```

*foo (int a, int b)*  
*→ a + b + 1;*

```
L_iloc_0: // 2 обращения к памяти
    tmp1 = local(0);
    push tmp1;
    next();
```

```
L_iconst_1: // 1 обращение к памяти
    push 1;
    next();
```

iloc_0
iloc_1
iadd
iconst_1
iadd
ireturn

Обращений к  
памяти: **11**

Регистровых  
операций: **2**

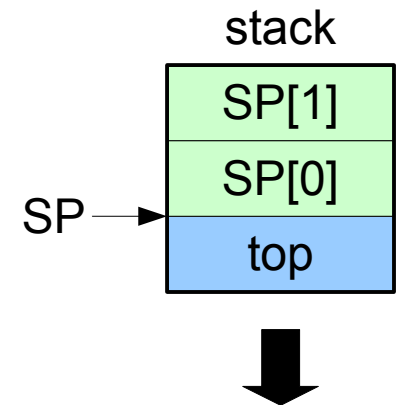
next(): **5**

«Циклов»:  
 $11 \cdot 3 + 2 \cdot 1$   
 $+ 5 \cdot 11 = \mathbf{90}$



# Ассемблерный интерпретатор байткода с кэшированием вершины стека операндов в регистрах

```
L_iadd:      // 1 обращение к памяти, 1 регистровая операция
    pop tmp1;
    top += tmp1;
    next();
L_iloadd_0:  // 2 обращения к памяти
    push top;
    top = local(0);
    next();
L_iconst_1:  // 1 обращение к памяти, 1 регистровая операция
    push top;
    top = 1;
    next();
```



iloadd_0
iloadd_1
iadd
iconst_1
iadd

Обращений к памяти: 7

Регистровых операций: 3

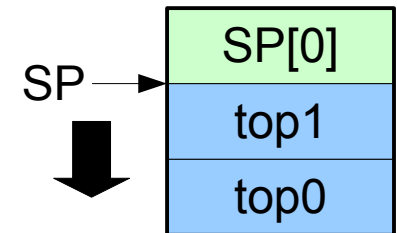
next(): 5

«Циклов»:  $7*3 + 3*1 + 5*11 = 79$

# Ассемблерный интерпретатор байткода с кэшированием 2 верхних элементов стека операндов

```
L_iadd:      // 1 обращение к памяти, 1 регистровая операция
             top0 += top1;
             pop top1;
             next();
```

```
L_ildload_0: // 2 обращения к памяти, 1 регистровая операция
             push top1;
             top1 = top0;
             top0 = local(0);
             next();
```



```
L_iconst_1: // 1 обращение к памяти, 2 регистровых операции
             push top1;
             top1 = top0;
             top0 = 1;
             next();
```

ildload_0
ildload_1
iadd
iconst_1
iadd

Обращений к памяти: 7

Регистровых операций: 6

`next()`: 5

«Циклов»:  $7*3 + 6*1 + 5*11 = 82$

# Эффекты кэширования верхних элементов стека операндов

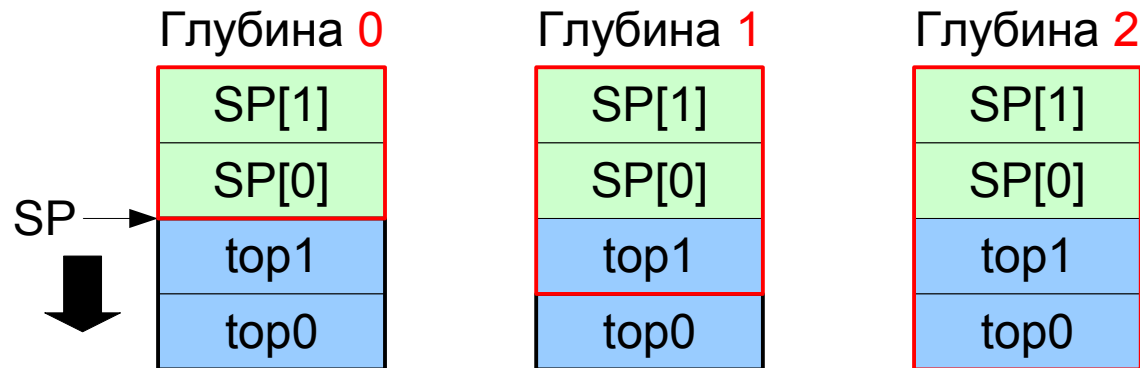
Глубина кэширования	iadd	iload(0)	iconst(1)
0	<pre>pop(tmp1); pop(tmp2); tmp2 += tmp1; push(tmp2);</pre>	<pre>tmp1 = local(0); push(tmp1);</pre>	<pre>push(1);</pre>
1	<pre>pop(tmp1); top += tmp1;</pre> <p>Экономия 2 инструкций доступа к памяти</p>	<pre>push(top); top = local(0);</pre> <p>Эффект нулевой</p>	<pre>push(top); top = 1;</pre> <p>1 дополнительная регистровая пересылка</p>
2	<pre>top0 += top1; pop(top1);</pre> <p>Экономия 2 инструкций доступа к памяти</p>	<pre>push(top1); top1 = top0; top0 = local(0);</pre> <p>1 дополнительная регистровая пересылка</p>	<pre>push(top1); top1 = top0; top0 = 1;</pre> <p>2 дополнительные регистровые пересылки</p>

- Наблюдения:
  - Оптимальная глубина кэширования зависит от виртуальной инструкции
  - С глубиной растёт число регистровых пересылок

# Кэширование верха стека на переменную глубину (1)

- Наблюдения:
  - Чем больше элементов кэшировано, тем больше регистров сохранять и восстанавливать при вызове
    - Стек операндов локален в секции активации
  - Накладные расходы растут вместе с глубиной
- Будем кэшировать стек на **переменную** и притом небольшую глубину
- Породим отдельные циклы интерпретатора для глубин 0-2 с взаимными переходами
  - Для каждой глубины кэширования — своя таблица адресов реализаций байткодов
  - Следующая глубина — параметр макроса **next**
    - По существу глубина - состояние конечного автомата
  - К шитому коду этот способ **не применим** — в нём таблица не используется

# Кэширование верха стека на переменную глубину (2)



- Реализации виртуальных инструкций могут быть объединены и повторно использованы

```
L0_iadd: // Входная глубина 0
        pop(top1); // Регистры top0 и top1 свободны
L1_iadd: // Входная глубина 1
        pop(top0); // Регистр top0 свободен
L2_iadd: // Входная глубина 2
        top1 += top0;
        // Выходная глубина 1
        goto Loop1; // Или next(1)
```

# Ассемблерный интерпретатор байткода с переменной глубиной кэширования стека операндов (1)

```
L0_iadd:      // 2 обращения к памяти, 1 регистровая операция
              pop top1;
L1_iadd:      // 1 обращение к памяти, 1 регистровая операция
              pop top0;
L2_iadd:      // 1 регистровая операция
              top1 += top0;
              next(1);
L0_ildload_0: // 1 обращение к памяти
              top1 = local(0);
              next(1);
L2_ildload_0: // 2 обращения к памяти, 1 регистровая операция
              push top1;
              top1 = top0;
L1_ildload_0: // 1 обращение к памяти
              top0 = local(0);
              next(2);
```

# Ассемблерный интерпретатор байткода с переменной глубиной кэширования стека операндов (2)

```
L0_iconst_1:  // 1 регистровая операция
               top1 = 1;
               next(1);
```

```
L2_iconst_1:  // 1 обращение к памяти, 2 регистровых операции
               push top1;
               top1 = top0;
```

```
L1_iconst_1:  // 1 регистровая операция
               top0 = 1;
               next(2);
```

```
foo (int a, int b)
    → a + b + 1;
```

iload_0
iload_1
iadd
iconst_1
iadd

Обращений к памяти: 2

Регистровых операций: 3

next() : 5

«Циклов»:  $2*3 + 3*1 + 5*11 = 64$

# Кэширование верха стека на переменную глубину (2)

- Увеличение размера кода интерпретатора
  - Иногда размер кода и констант критичен
  - Больше вероятность промахов мимо кэша кода
- Увеличение числа таблиц косвенных переходов
  - Увеличение нагрузки на предсказание переходов
- Нарушение выравнивания меток переходов
  - Для некоторых процессоров это важно
- Полезный эффект ограничен фиксированными расходами на декодирование

Глубина	Полезная работа	Общая работа	Декодирование
0	35 (100%)	$35+55 = 90$ (100%)	55 ( $55/90 = 61\%$ )
1	24 ( $24/35 = 69\%$ )	$24+55 = 79$ ( $79/90 = 88\%$ )	55 ( $55/79 = 70\%$ )
2	27 ( $27/35 = 77\%$ )	$27+55 = 82$ ( $82/90 = 91\%$ )	55 ( $55/82 = 67\%$ )
переменная	9 ( $9/35 = 26\%$ )	$9+55 = 64$ ( $64/90 = 71\%$ )	55 ( $55/64 = 86\%$ )



# Многоходовый косвенный шитый код

&L0_enter	
&L1_enter	
&L2_enter	
iload	0
iload	1
iadd	
iconst	1
iadd	
exit	

```
exit: .word L0_exit, L1_exit, L2_exit
```

```
L0_exit: Exit(0)
```

```
L1_exit: Exit(1)
```

```
L2_exit: Exit(2)
```

```
L0_enter: Enter(0)
```

```
L1_enter: Enter(1)
```

```
L2_enter: Enter(2)
```

```
Next: macro(i)
    tmp = [ip++];
    jmp [tmp + i * BytesInWord];
endm

Enter: macro(i)
    rpush(ip);
    ip = tmp + (3-i) * BytesInWord;
    Next(i);
endm

Exit: macro(i)
    rpop(ip);
    Next(i);
endm
```

```
iadd: .word L0_iadd, L1_iadd, L2_iadd
```

```
L0_iadd: pop top1;
```

```
L1_iadd: pop top0;
```

```
L2_iadd: top1 += top0;
```

```
Next(1);
```

# Оптимизация интерпретируемого кода

- Компилятор в виртуальный код
- Конвертор в исполняемый код
- Интерпретатор

# Оптимизации в компиляторе в виртуальный код (1)

- Компилятор обладает значительными вычислительными ресурсами
- Полезные оптимизации
  - Свёртка константных выражений
  - Протяжка констант и присваиваний
  - Упрощение выражений
  - Изведение общих подвыражений
  - Понижение динамики вызовов
  - Закрытая макроподстановка вызовов (*inlining*)
  - Изведение мёртвого кода, в т.ч. более не используемых локальных переменных
  - Расцикливание
  - Передача результатов анализов конвертеру и интерпретатору в виде аннотаций кода
  - Но **возможности применения** этих оптимизаций **крайне ограничены**

# Оптимизации в компиляторе (2)

- Раздельная компиляция модулей
  - Может ли компилятор использовать константу, импортированную из другого модуля?
  - Нет, мы не узнаем, если она там изменится
    - Она должна быть прочитана во время линковки или исполнения
    - Вариант: приписать модулям **версии** и следить за зависимостями на уровне версий (реестр модулей, подгрузка обновлений...)
    - Вариант: оптимизированный код — необязательное дополнение к неоптимизированному, **импортированные данные сохраняются вместе с кодом**, проверяются на совпадение **до его исполнения**, при несовпадении используется неоптимизированный код
  - То же верно для типов и кода
    - Класс может быть подклассом другого класса или реализовать другой интерфейс
    - В классе могут быть другие поля и методы
    - Тип или смещение поля могут измениться...

# Оптимизации в компиляторе (3)

- Языковая рефлексия
  - Иногда может изменять произвольные языковые элементы
    - Значения констант (например, `System.out` в Java)
    - Реализацию классов (например, `redefineClasses`)
  - Внутренние и внешние зависимости кода могут измениться **после его загрузки**
- Анализ, отладка и профилирование
  - Выполненные оптимизации не должны быть заметны для средств разработки
    - Трассы стека брошенных исключений
    - Привязка точек останова к исходному коду
    - Пошаговое исполнение
    - Переменные и их значения...
- Компилятор не знает, где будет запускаться порожденный код
  - В идеале «Write once, run everywhere» (James Gosling)

# Оптимизации в конверторе (1)

- Конвертор может обладать или **не** обладать значительными вычислительными ресурсами
  - Например, если конверсия производится при загрузке распространяемого кода в небольшое устройство
- Знает VM и устройство, на которых будет выполняться код
  - Может знать весь присутствующий на устройстве код
- Менее ограничен раздельной компиляцией
  - Достаточно локально хранимых зависимостей
  - При обновлении распространяемых модулей нужно заново конвертировать их и все зависимые модули
- Больше полезных оптимизаций и возможностей их применения, чем у компилятора
  - Например, оптимизация приложения целиком
  - Для этого должен распространяться высоко- (HIR) или среднеуровневое (MIR) представление программы
    - Обычный байтовый код слишком низкоуровневый

# Оптимизации в конверторе (2)

- Полезные оптимизации
  - Языко-зависимые оптимизации
    - Например, выборочная инициализация классов в Java
    - Многие инструкции бросают исключения, если класс отсутствует, и вызывают его статический инициализатор, если он еще не инициализирован
    - После этого выполняется простая семантика самой инструкции
  - Специализация инструкций по набору непосредственных параметров
  - Замена идиом супер-инструкциями
  - Закрытая макроподстановка (*inlining*) простых вызовов
- Конвертор может выполнять оптимизации интерпретатора
  - Если ему известны инварианты состояния программы во время ее будущего выполнения
    - Например, конвертор инициализировал класс и поэтому знает, что он останется инициализированным
  - Снижение нагрузки на интерпретатор

# Оптимизации в конверторе (3)

- Альтернативные версии сконвертированного кода
  - Для разных наборов опций запуска интерпретатора
  - Например, отладочная версия без оптимизаций



# Специализация инструкций по непосредственным параметрам

- Малые константы и малые смещения локальных переменных и полей встречаются гораздо чаще, чем большие
  - Заведём для них *специализированные* инструкции
    - `<t>const_<value>`
    - `<t>{load, store}_<offset>`
    - `<t>{get, set}field_<offset>`
  - Короче код, быстрее выполнение
    - Смещение зашито в реализацию, не нужно его вычитывать из потока инструкций или дескриптора в литеральном пуле
  - Некоторые из этих инструкции полезны интерпретатору для выполнения его оптимизаций
  - Но свободных инструкций мало (кроме шитого кода)
    - Можно первый байт считать признаком расширения кода, а второй - кодом инструкции, но такие инструкции декодируются вдвое медленнее

# Замена идиом (1)

- *Идиома* — часто встречающаяся последовательность виртуальных инструкций
  - Например, прочитать поле текущего объекта *this*:  
`aload_0; getfield(cp_index)`
  - Нет переходов снаружи внутрь последовательности
- Известно, что текущий объект не *null*
  - Интерпретатор не смог бы вызвать текущий метод
  - Язык не разрешает присваивания *this*
    - Зато **разрешает JVM** — в виртуальных методах можно использовать `astore(0)`, поэтому конвертор должен проверить весь метод
- Заведем оптимизированную *супер-инструкцию* для всей идиомы `aload_0_<т>get_field_<offset>`
  - Если класс загружен и верифицирован, мы знаем **тип** и **смещение** поля, поэтому можем выполнить замену
  - Иначе перепишем `aload_0` в `quick_aload_0`, чтобы интерпретатор попытался сделать **аналогичную замену** при выполнении этой инструкции

# Замена идиом (2)

- Супер-инструкции можно использовать для словарного сжатия виртуального кода
  - Напишем автоматический анализатор кода
  - Зададим ему ограничения на длину и число идиом
  - Подадим на вход код одного или нескольких приложений или библиотек
  - Пусть он найдет идиомы, замена которых на супер-инструкции наиболее сокращает код приложения
  - ... и породит оптимизированный код для реализации этих супер-инструкций
  - Удобнее всего это делать с шитым кодом
    - Набор инструкций неограниченно расширяем
  - Сжатый таким образом код не требует распаковки перед выполнением
- Супер-инструкции и отладка программ
  - Супер-инструкции могут быть заметны при отладке
    - Нельзя остановиться посреди супер-инструкции
    - Другие индексы в трассе стека

# Макроподстановка вызовов простых функций (1)

- Многие функции **простые** и **короткие**
  - Например, аксессоры полей
  - Их код *позиционно-независим*
  - Не содержит сложных инструкций
    - Например, со ссылкой к чужому пулу констант или *try-catch* блоков
    - Является ли сложным бросание исключений?
  - Его можно просто копировать
    - Особенно при отсутствии локальных переменных
    - Например, если параметры функции передаются на стеке операндов

# Макроподстановка вызовов простых функций (2)

- Подставим тело такой функции вместо ее прямого вызова
  - Сокращение размера кода
  - Если все вызовы функции подставлены, можно ее удалить
    - Мешает рефлексия, в т.ч. динамическая нативная
    - Функцию можно искать по имени
  - Ускорение кода
    - Особенно если после подстановки выполнить замену образовавшихся идиом

Ian Piumarta, Fabio Ricardi. *Optimizing Direct Threaded Code By Selective Inlining*. ACM SIGPLAN Notices, June 1998

# Макроподстановка вызовов простых функций (3)

- Макроподстановка может быть заметна при:
  - Бросании исключений из подставленного кода
    - В трассе отсутствует секция подставленного вызова
    - Вариант: хранить в таблицах привязки не одну позицию в вызывающей функции, а позиции во всём стеке подставленных в данном месте вызовов
  - Исчерпанию стека
    - Подставленный вызов может быть выполняться при некотором статически невычислимом условии
    - При слиянии секций суммарный размер может (но не обязан) увеличиться
    - Проверка переполнения происходит при входе в вызывающую функцию
    - Если размер секции увеличивается незначительно, изменение поведения не заметно
    - Если спецификация VM не задаёт размеры стека и секции вызова, момент исчерпания стека не задан
  - Профилировании, отладке и инструментировании кода

# Оптимизации в интерпретаторе (1)

- Интерпретатор знает опции запуска VM
  - Например, запущена ли VM в отладочном режиме
- Знает текущее динамическое состояние программы
  - Например, набор и состояние загруженных классов
  - Может оптимизировать код **для данного выполнения**
- Не обладает ресурсами для анализа и преобразования программ
  - Но может использовать данные, подготовленные для него компилятором и конвертором

# Оптимизации в интерпретаторе (2)

- Итеративный интерпретатор может выполнять только **локальные оптимизации, не меняющие длины кода**
  - Интерпретатор не должен перемещать код в памяти, корректировать таблицы и адреса переходов...
  - **Это ограничение линейных кодов** — рекурсивный интерпретатор может свободно менять деревья
  - *Code Quickening* - локальная (само-)модификация кода для ускорения его будущих выполнений
  - В VM: замена интерпретатором текущей инструкции или их короткой последовательности на быстрее выполняемую версию
- Оптимизации
  - Специализация инструкций по инвариантам состояния непосредственных параметров
  - Макроподстановки вызовов простых функций
  - Замена оставшихся идиом супер-инструкциями



# Специализация инструкций по инвариантам состояния непосредственных параметров (1)

- Жизненный цикл некоторых языковых объектов представляет собой небольшой набор состояний с однонаправленными переходами между ними
- Например, при ленивой линковке:
  - Класс может быть один раз найден и загружен, после чего один раз инициализирован
  - Неразрешенные ссылки на класс, поле и метод могут быть один раз разрешены
- При разрешении ссылки для ускорения последующих выполнений в литеральном пуле кэшируются **неизменные атрибуты** объекта
  - Например, для поля — тип и смещение
  - Для статического поля — тип и адрес

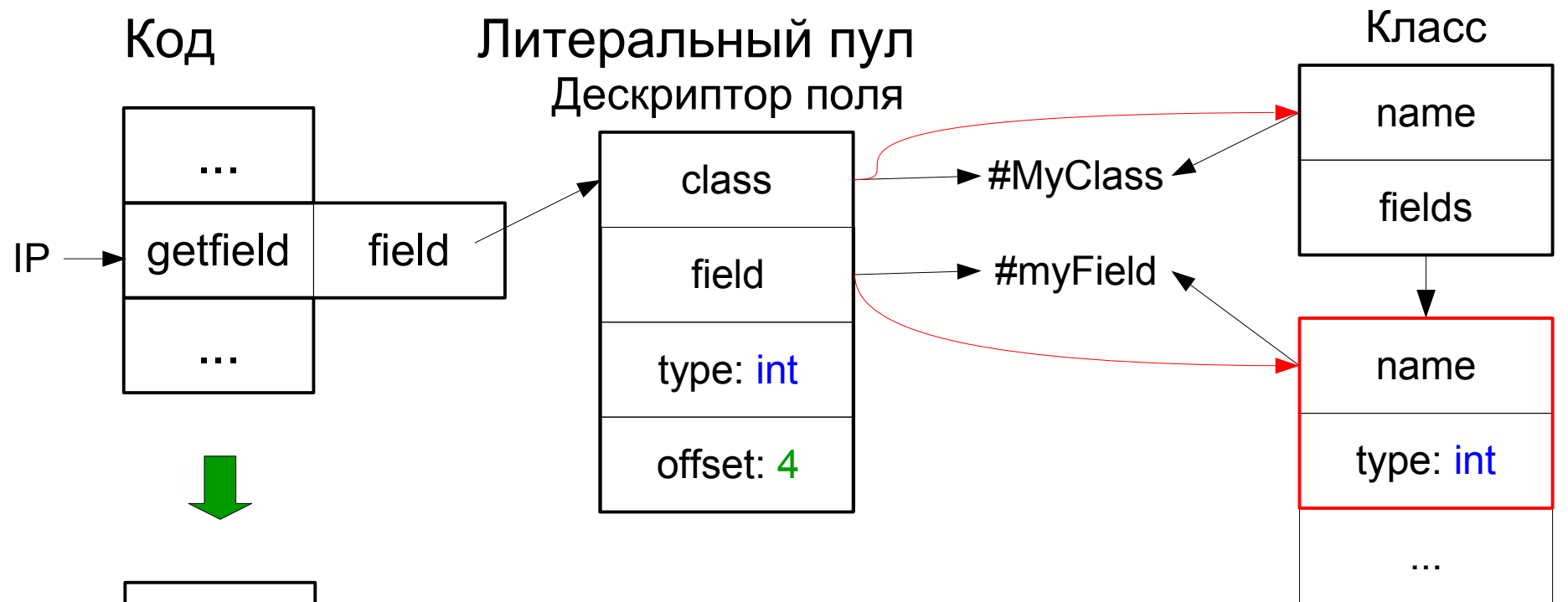
# Специализация инструкций по инвариантам состояния непосредственных параметров (2)

- При каждом исполнении инструкция доступа к полю `{get|set}field(cp_index):`
  - Проверяет, разрешена ли ссылка в литеральном пуле
  - Если нет, пытается ее разрешить и запомнить тип и смещение
  - Иначе использует тип и смещение для чтения или записи поля
- Повторные проверки избыточны
  - После успешного разрешения ссылки на поле его тип и смещение не могут измениться
- Нельзя ли избежать повторения проверок и чтения атрибутов?

# Специализация инструкций по инвариантам состояния (3)

- Заведем внутренние ускоренные инструкции, специализированные по **типу поля**  
`<t>{get|set}field(field_offset)`
- Можно дополнительно специализировать эти инструкции по **смещениям**  
`<t>getfield_<field_offset>`
  - Но размер кода должен сохраняться...
  - И проверка на null ограничивает полезный эффект
  - Можно использовать аппаратную защиту страниц
- После успешного разрешения ссылки заменим общую инструкцию специализированной
- Возможности такого переписывания кода ограничены
  - Мало свободных инструкций (кроме шитого кода)

# Специализация инструкций по инвариантам состояния (4)



- После разрешения ссылки на поле ранее не известного класса интерпретатор переписывает дескриптор, чтобы больше не заглядывать в класс...
- ...и инструкцию, чтобы больше **В ЭТОМ МЕСТЕ** не заглядывать в дескриптор

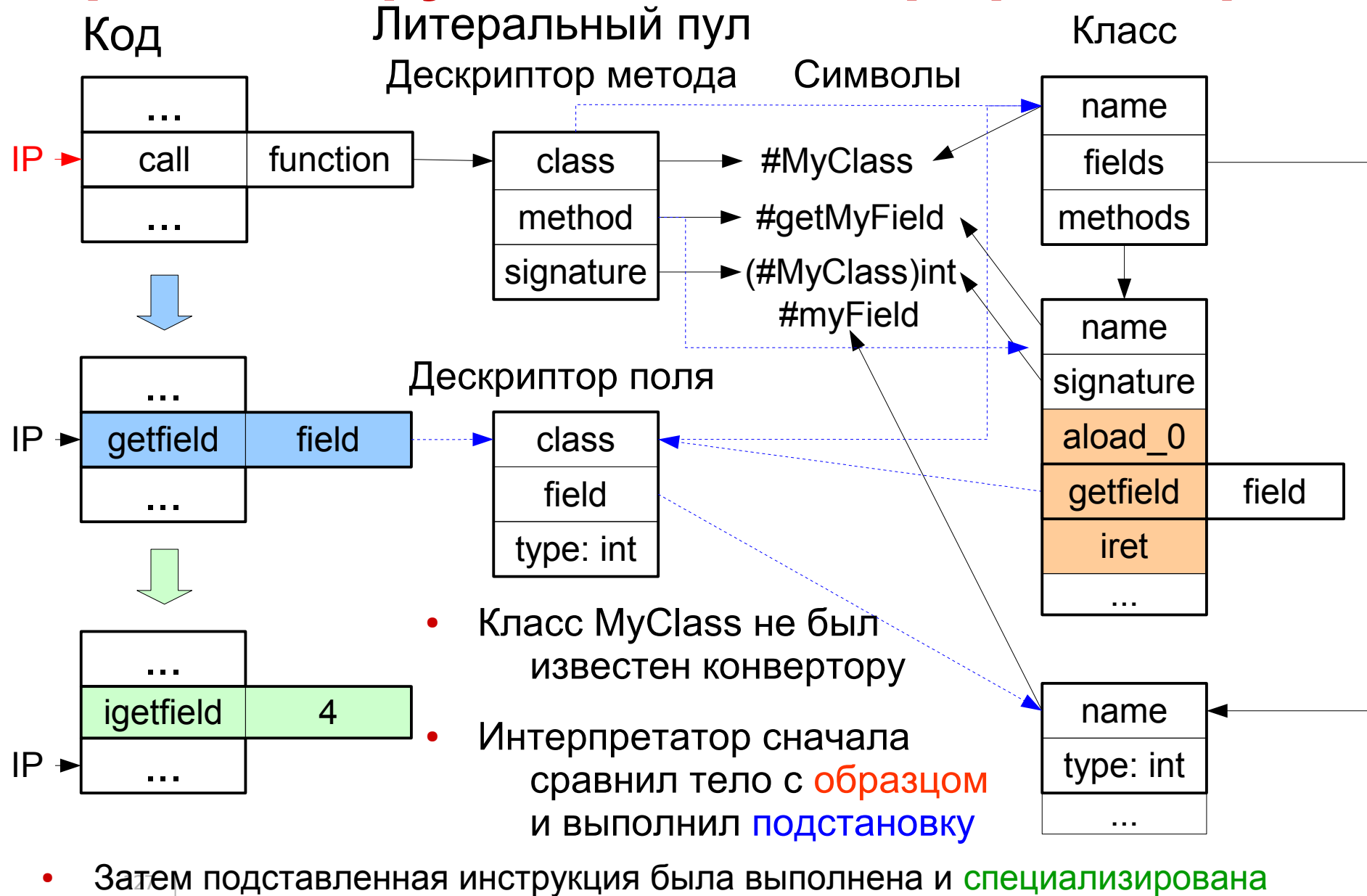
# Макроподстановка вызовов простых функций в интерпретаторе (1)

- Конвертор не подставил вызов, потому что код вызываемой функции не был ему известен
- Интерпретатор не может анализировать код и должен сохранять его длину
- Возможности подстановки **крайне ограничены**
- Выберем несколько коротких образцов
- Если вызываемая функция стала известна, сравниваем ее тело с образцами
  - При совпадении выполняем замену
  - Если совпадение в будущем невозможно, чтобы не повторять попытки, перепишем инструкцию вызова
  - Кроме тела функции, могут быть важны:
    - Атрибуты функции (например, *synchronized*)
    - Её метаданные (таблицы обработки исключений)
    - Исключения, бросаемые **инструкцией вызова**

# Макроподстановка вызовов простых функций в интерпретаторе (2)

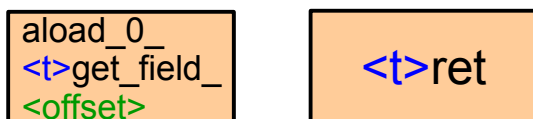
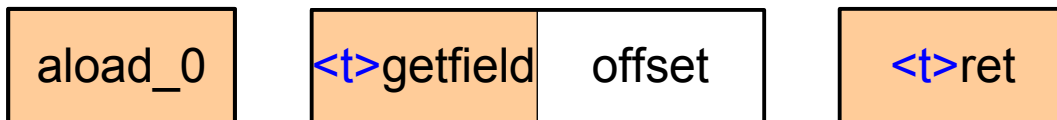
- Подстановка заметна при бросании исключений
  - При наивной реализации в трассе стека отсутствуют секции подставленных функций
  - Но стандарт языка может не определять поведение с такой точностью
  - Можно не подставлять непосредственно или косвенно бросающий исключения код (**но бывает ли такой?**)
  - Лучше записать подстановки в *журнал замен*
    - Интерпретатор добавляет записи в конец таблицы фиксированного размера, отведенной конвертором
    - По таблице узнаем, брошено ли исключение в подставленном коде
    - Если да, то было бы оно брошено инструкцией вызова функции или в ее теле
    - Если в теле, вычисляем смещение, по которому было бы брошено исключение в вызванной функции
    - И просим эту функцию дописать в трассу ее секцию (или секции, если в функции были подстановки)
    - После этого добавляем в трассу свою секцию

# Макроподстановка вызовов простых функций в интерпретаторе

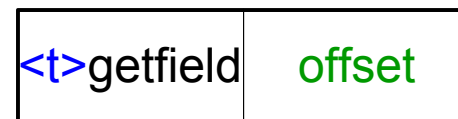
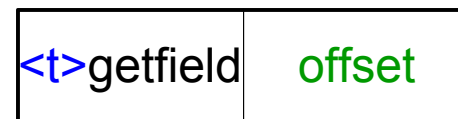
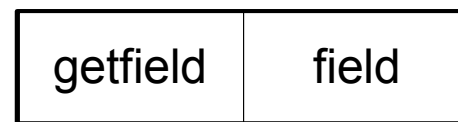


# Варианты образцов

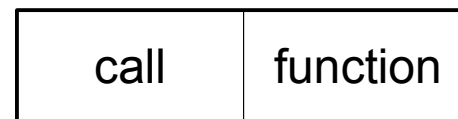
## Варианты образцов тела функции



## Варианты замены



↑  
**Заменяемый вызов**



- Набор образцов должен включать варианты, появляющиеся в результате частичного переписывания кода конвертором и интерпретатором
- Замена должна бросать все исключения инструкции вызова

- Размер кода обязан сохраняться



# Замена оставшихся идиом

## супер-инструкциями

- Конвертор проанализировал код, нашел начало похожей на идиому последовательности
- В ней оказалась ссылка на пока не известный элемент (например, класс, поле, метод...)
  - Замена не может быть выполнена, пока эта ссылка не разрешена
    - Неизвестно, существует ли это поле, каково его смещение, правильный ли у поля тип
- Интерпретатор не может анализировать код, поэтому не может делать такие замены **сам**
  - В частности, не знает, нет ли перехода внутрь этой последовательности
    - Если метка — это инструкция, анализ не нужен, но пропуск таких меток замедляет интерпретатор
  - Конвертор помогает интерпретатору, переписывая инструкцию в начале идиомы
  - Замена не обязательно идентична конверторной — интерпретатор должен сохранять длину кода

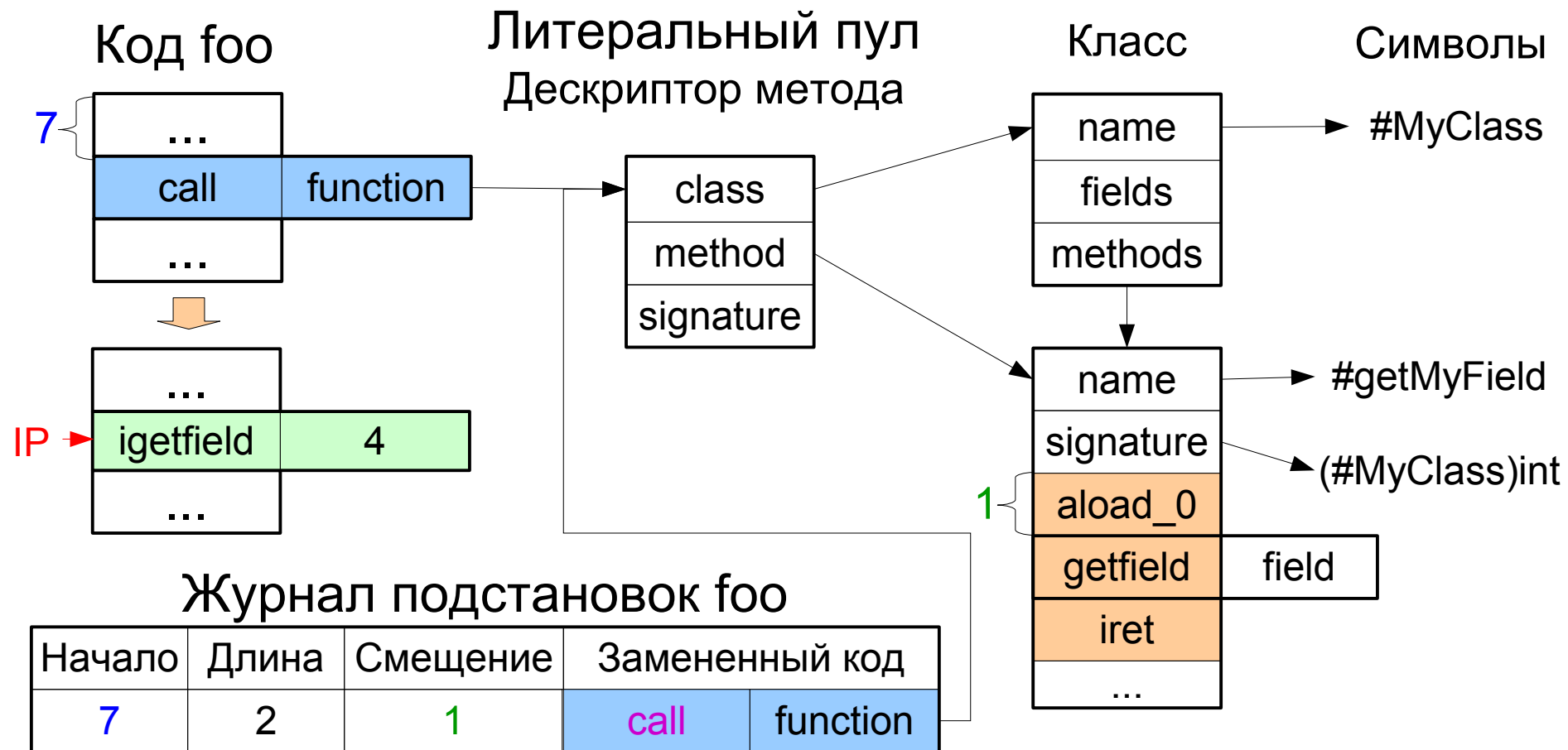
# Соккрытие изменений кода (1)

- Конвертор и оптимизирующий интерпретатор переписывают код
- Средства разработки (инспекторы, отладчики, профиляторы) не должны этого замечать
  - Они работают в терминах исходного или стандартного промежуточного кода
- Для этого проще всего глобально выключать все или наиболее сложные оптимизации
- Часто можно **транслировать** термины средств разработки в термины интерпретатора
  - Таблицы привязки адресов исполняемого кода к позициям исходного и/или распространяемого кода
  - Таблицы локальных переменных

# Соккрытие изменений кода (2)

- Соккрытие переписывания кода с сохранением размера
  - Делаем вид, что код не менялся
  - Или восстанавливаем переписанные идиомы
    - Переписывание обратимо, можно его откатывать лениво по мере необходимости
- Сложнее скрыть макроподстановки вызовов
  - Они необратимы - нужно в таблице запомнить выполненные замены и исходные фрагменты кода
    - Замена идентифицируется началом и длиной
    - Таблица отсортирована по началам
    - При вложенных подстановках следующий элемент попадает в диапазон предыдущего
  - По этой таблице можно восстановить и трассу стека
    - По заданному адресу извлекаем из таблицы стек подстановленных вызовов и добавляем их в трассу
  - **Интерпретатор** не может создавать сложные таблицы, но и **не делает сложные замены**

# Соккрытие макростановок



- Предположим, на стеке null, нужно бросить *NPE* с правильной трассой стека. По текущему адресу находим запись в журнале. По замененной инструкции видим, что *NPE* нужно бросать в подставленной функции. Вычисляем в ней смещение, пусть она добавит свою трассу для этого смещения, потом добавим свой элемент.

Восстановленная трасса стека *NPE*

MyClass.getMyField	1
foo	7
...	

$$IP - (\&foo + 7) + 1 = 1$$

# Ускорение разработки итеративных интерпретаторов

- Использовать C/C++
  - Если понадобится, потом напомним ассемблерные вставки, используем прагматы и т.д.
- Написать свой DSL, порождающий ассемблерный или бинарный код
- Использовать генератор интерпретаторов

M. A. Ertl, D. Gregg, A. Krall, and B. Paysan, “*vmgen - A Generator of Efficient Virtual Machine Interpreters*”, *Software - Practice and Experience*, vol. 32, no. 3, 2002.

  - Реализации всех виртуальных инструкций на C
  - Цикл интерпретатора или прямой шитый код
    - Шитый код использует расширение GCC
  - Дизассемблер, отладчик и профилятор виртуального кода
  - Некоторые важные оптимизации
    - Кэширование стека, суперинструкции...

# Домашнее задание №3

- Написать статический анализатор частот вхождений параметризованных байткодов в программах на *Lama*
  - Анализатор читает байткод из файла
  - Считает вхождения параметризованных байткодов и выводит их отсортированными по убыванию частоты
  - Анализатор проверяется на скомпилированных в байткод тестах производительности
    - Сейчас там единственный тест
  - Исходный код и тест(ы) производительности: <https://github.com/JetBrains-Research/Lama>
  - Если задание покажется слишком легким, проанализировать частоты вхождений идиом ограниченной длины

# Вопросы?