

Распределение регистров и решение паззлов

Register Allocation by Puzzle Solving

Косарев Дмитрий

матмех СПбГУ

18 ноября 2019

- 1 Задача распределения регистров
- 2 Особенности, осложняющие задачу
- 3 Терминология
- 4 Стандартные алгоритмы (prior work)
- 5 Построение элементарных программ
- 6 Отображение элементарных программ в фигурки пазлов
- 7 Решаем пазлы

- 1 **Задача распределения регистров**
- 2 Особенности, осложняющие задачу
- 3 Терминология
- 4 Стандартные алгоритмы (prior work)
- 5 Построение элементарных программ
- 6 Отображение элементарных программ в фигурки пазлов
- 7 Решаем пазлы

Задача распределения регистров (register allocation)

- Назначение физических локаций переменным в программе
 - регистры быстрые, но их мало
 - памяти много, но она медленная
- Ограничение: переменные, которые *живы одновременно*, должны быть назначены в разные регистры
- Если регистров не хватает, то некоторые переменные должны храниться в памяти

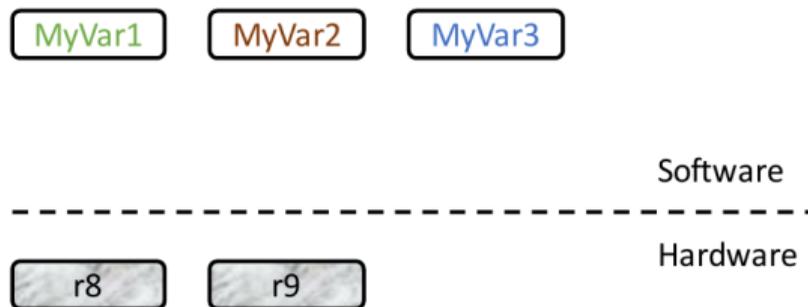
Оглавление

- 1 Задача распределения регистров
- 2 Особенности, осложняющие задачу**
- 3 Терминология
- 4 Стандартные алгоритмы (prior work)
- 5 Построение элементарных программ
- 6 Отображение элементарных программ в фигурки пазлов
- 7 Решаем пазлы

Распределение регистров: что происходит?

Исходная программа:

```
MyVar1 = 2
MyVar2 = 40
MyVar3 = 0
MyVar3 += MyVar1
MyVar3 += MyVar2
print(MyVar3)
```

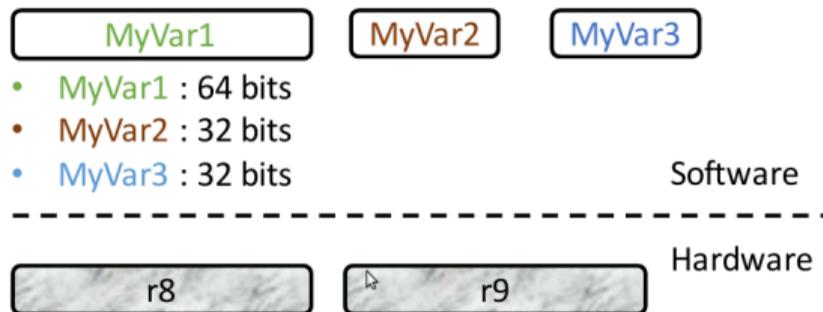


Результат:

```
MyVar1 -> stack
MyVar2 -> r8
MyVar3 -> r9
```

Исходная программа:

```
MyVar1 = 2
MyVar2 = 40
MyVar3 = 0
MyVar3 += MyVar1
MyVar3 += MyVar2
print(MyVar3)
```



Иерархия регистров (aliasing):

- r8 может хранить одно 64-битное число или два 32-битных
- r9 может хранить 64-битное число

Нерегулярные архитектуры. Pre-coloring

Вызов функций (PowerPC):

```
a := 10;  
b := 2;  
R0 := a;  
R1 := b;  
call(R0, R1);
```

Деление (x86)

```
a := 10;  
b := 2;  
AX := a;  
(AL, AH) := DIV AX, b;  
d := AL; // частное  
r := AH; // остаток
```

Некоторые переменные должны быть связаны с конкретными регистрами из-за соглашений о вызовах, деления и т.п.

Нерегулярные архитектуры. Иерархия регистров (aliasing)

Вложенные регистры могут использоваться как независимо, так и в комбинации с другими

Встречается в архитектурах: x86, Sun SPARC, MIPS у чисел плавающей точкой, и т.д.

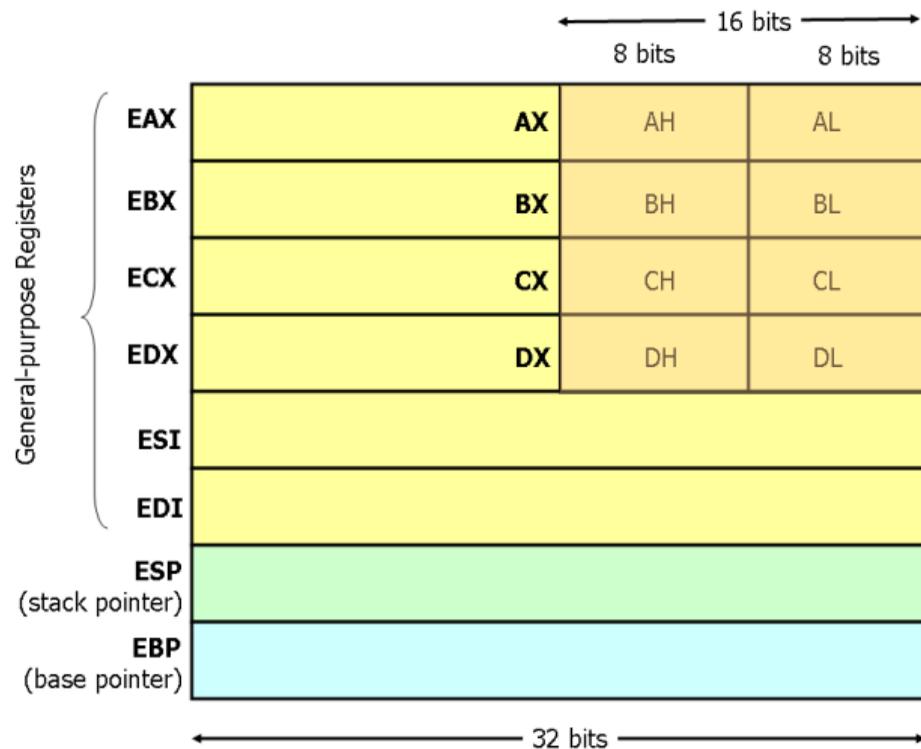


Рис.: Пример: иерархия регистров у Pentium

- 1 Задача распределения регистров
- 2 Особенности, осложняющие задачу
- 3 Терминология**
- 4 Стандартные алгоритмы (prior work)
- 5 Построение элементарных программ
- 6 Отображение элементарных программ в фигурки пазлов
- 7 Решаем пазлы

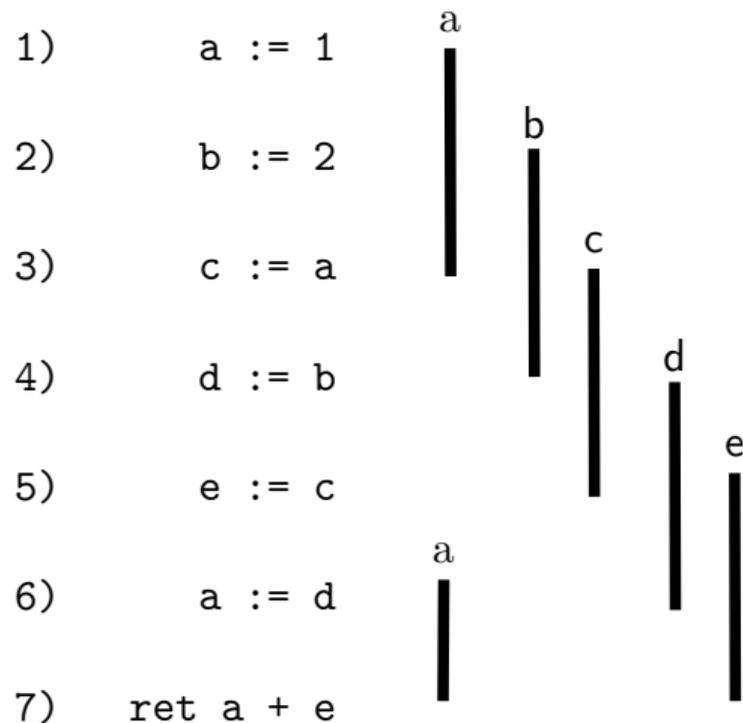
Терминология (1/3). Spilling & coalescing

Spilling

Если регистров не хватает для всех переменных, то некоторые должны храниться в памяти. Обращение к этим переменным **неэффективно**

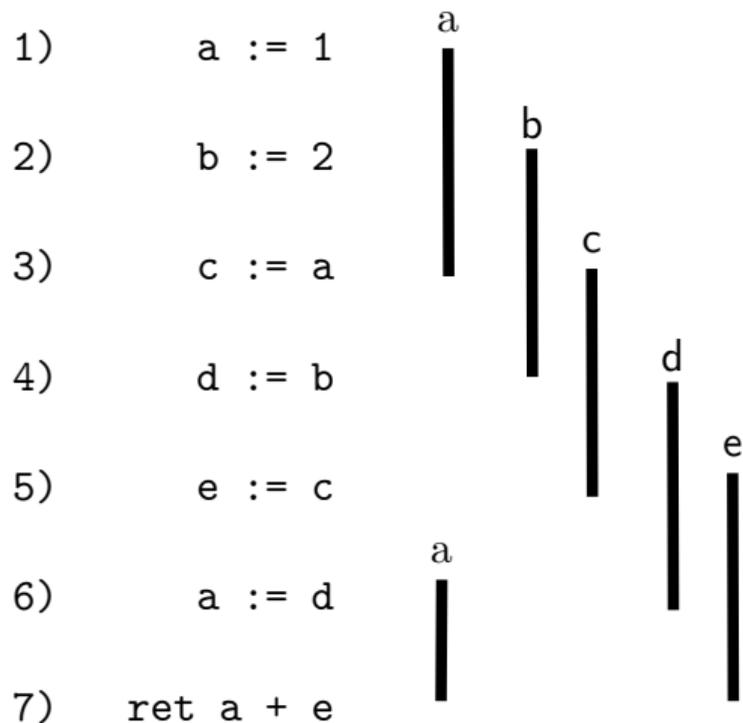
Если области использования переменных *не пересекаются* и они *связаны инструкцией копирования* $v_1 = v_2$, то их следует хранить в одном регистре, и это называется *coalescing* (объединение).

Терминология (2/3). Период жизни переменной



- Переменная *живая*, если она может быть использована в будущем
- *Период жизни (live range)* переменной – это коллекция точек программы, где она жива.

Распределение регистров и графы. Пример (1/3)



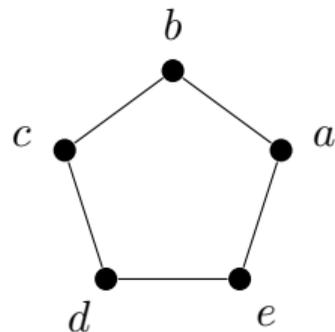
Вопросы:

- Сколько нужно регистров для этой программы?
- Существует ли универсальный алгоритм?
- P или NP?

Распределение регистров и графы. Пример (2/3)

1)	<code>a := 1</code>	a				
2)	<code>b := 2</code>		b			
3)	<code>c := a</code>			c		
4)	<code>d := b</code>				d	
5)	<code>e := c</code>					e
6)	<code>a := d</code>	a				
7)	<code>ret a + e</code>					

Граф несовместимости (interference graph)



Spill-free Register Allocation = SFRA
SFRA \sim Graph Coloring.
SFRA – NP-полна

Распределение регистров и графы. Пример (3/3)

a := 1

b := 2

c := a

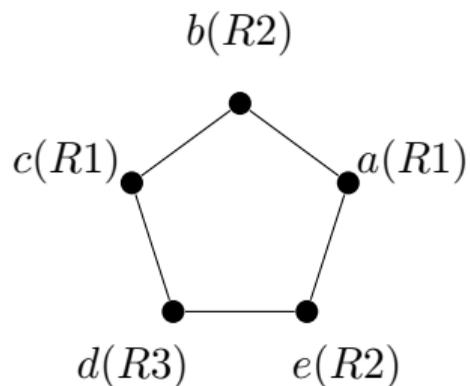
d := b

e := c

a := d

ret a + e

Нужно три регистра: R1, R2 и R3



R1 := 1

R2 := 2

R1 := R1

R3 := R2

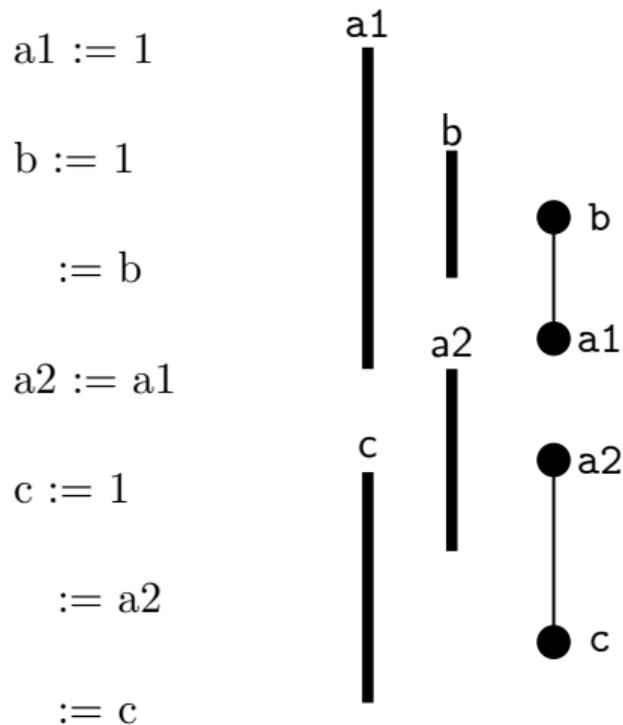
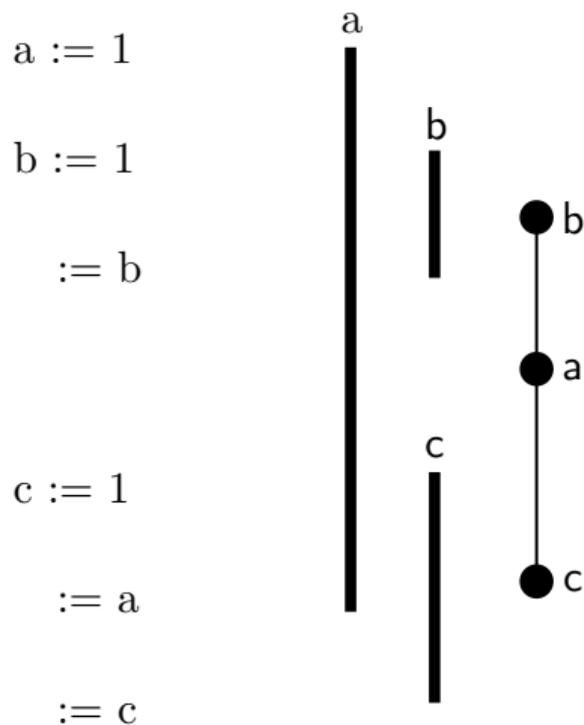
R2 := R1

R1 := R3

ret R1 + R2

Терминология (3/3). Live range splitting

Вставка инструкций копирования для упрощения interference graph



- 1 Задача распределения регистров
- 2 Особенности, осложняющие задачу
- 3 Терминология
- 4 Стандартные алгоритмы (prior work)**
- 5 Построение элементарных программ
- 6 Отображение элементарных программ в фигурки пазлов
- 7 Решаем пазлы

Стандартные алгоритмы (1/2)

- Раскраска вершин графа в k цветов (Chaitin et al. 1982)
 - NP-полная задача
 - требует live range splitting, иначе spillится слишком много
 - переменная, которая не spilled, всегда находится в одном регистре
 - алгоритмы для оптимизации coalescing слишком консервативны (Briggs)
- Linear Scan Register Allocation (2я половина 1990х) – жадный алгоритм появился из-за критики консервативных подходов к coalescing
 - Изначально работал существенно быстрее, но выдавал не такой хороший код
 - Позже это улучшили с помощью binpacking (Traub et al., 1998)
 - Дальнейшее: выбор более оптимальных место для live range splitting
 - *Extended linear scan* – вставка дополнительных инструкций для уменьшения spilling

Стандартные алгоритмы (2/2)

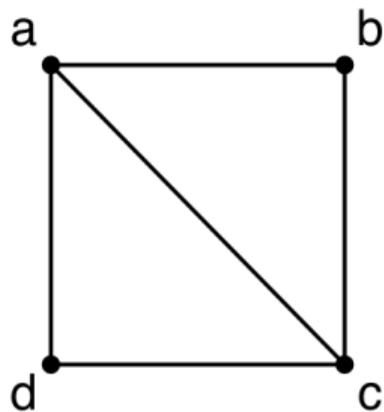
- Linear Programming – слишком медленный, даже по сравнению с раскраской графа
- Register allocation via Partitioned Quadratic Programming
дает оптимальное решение (если оно есть) за $O(|V| \cdot K^3)$, где V – переменные, а K – количество регистров
- Register allocation via Multi-Flow of Commodities
Умеет порождать более маленький код, чем раскраска графа
- Распределение регистров на основе SSA представления (2005)
- Распределение регистров с помощью решения паззлов (2008)

Распределение регистров на основе SSA представления (2005)

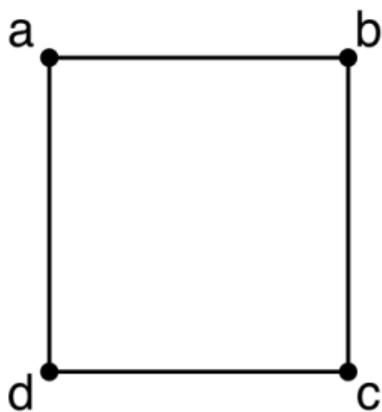
- Существенный прогресс
- Из-за SSA представления программ *interference graph* можно сделать хордовым
- Тогда алгоритм раскраски графа становится полиномиальным

Определение (Хордальный граф (*chordal graph*))

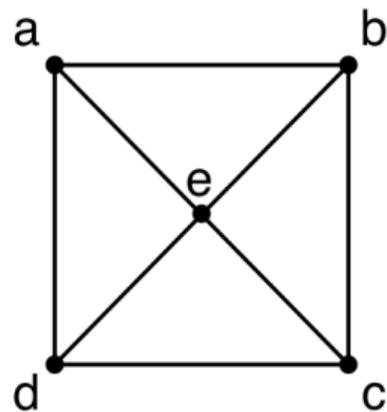
Граф называется *хордовым* (*chordal*), если каждый из его циклов, имеющих четыре ребра и более, имеет хорду (ребро, соединяющее две вершины цикла, но не являющееся его частью)



(a)



(b)



(c)

Рис.: (a) Хордовый граф. (b-c) Нехордовые графы

Ходовые графы имеют хорошие свойства. Задачи *минимальной раскраски, поиска максимальной клики и минимального покрытия кликами* NP-полны, но решаются за полиномиальное время для хордовых графов.

- 1 Задача распределения регистров
- 2 Особенности, осложняющие задачу
- 3 Терминология
- 4 Стандартные алгоритмы (prior work)
- 5 Построение элементарных программ**
- 6 Отображение элементарных программ в фигурки пазлов
- 7 Решаем пазлы

Из программ в кусочки паззлов

1. Преобразуем в программу в более простой вид
 - A. Преобразуем исходную программу в SSA
 - B. Преобразуем A в SSI форму
 - C. Вставляем в B параллельное копирование между каждой парой инструкций
2. Отображаем элементарные программы в кусочки паззлов.

Static Single Assignment (SSA)

Определение (SSA форма)

Программа находится в SSA форме, если для всякой переменной только одна инструкция (statement) присваивания в теле программы присваивает значение этой переменной

Плюсы:

- ссылочная прозрачность (referential transparency)
- значение переменной не зависит от позиции вхождения её идентификатора в программу

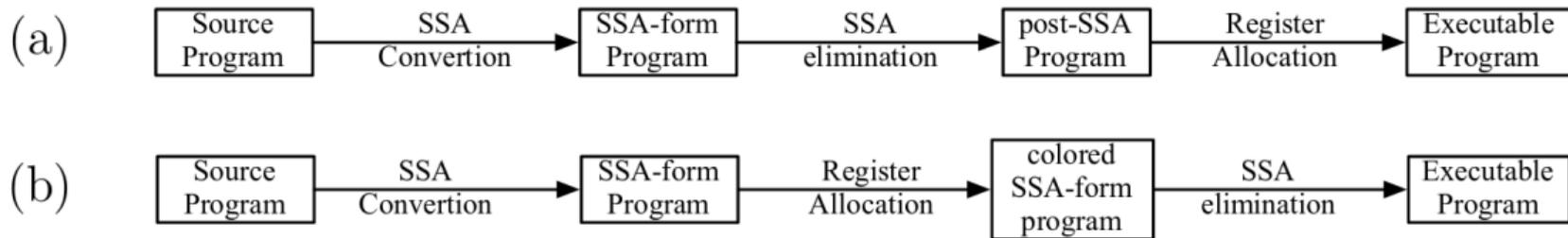


Рис.: (a) Классическое распределение регистров, (b) Распределение регистров на основе SSA

Static Single Assignment (SSA). Пример

Переменной присваивается значение только один раз

```
/* not a SSA */
```

```
x = 1;
```

```
y = x + 1;
```

```
x = 2;
```

```
z = x + 1;
```

```
/* SSA */
```

```
x1 = 1;
```

```
y = x1 + 1;
```

```
x2 = 2;
```

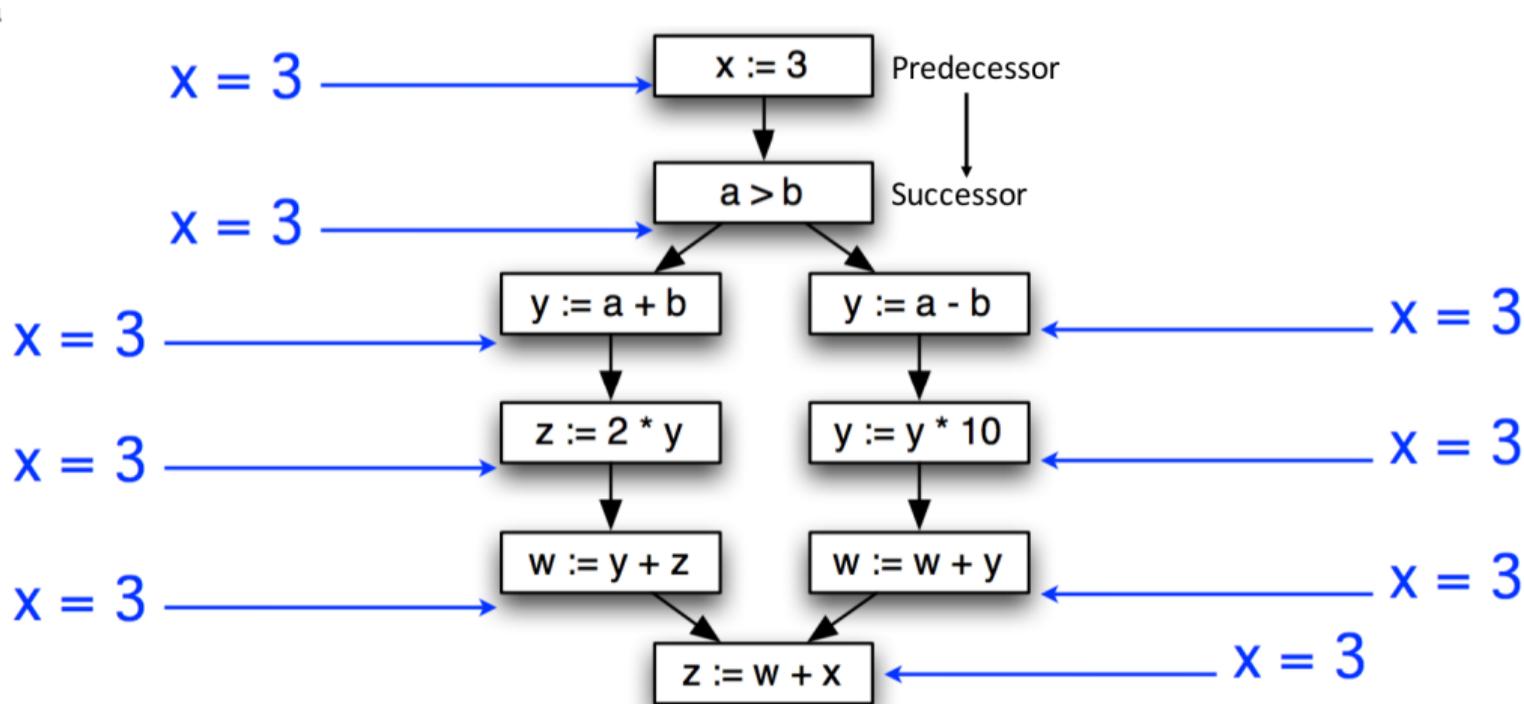
```
z = x2 + 1;
```

В примере выше хочется оптимизировать и написать $z = y$; так как правые части синтаксически одинаковые.

Но это сделать нельзя так как в x присваивается новое значение, т.е. содержимое переменной z зависит от контекста.

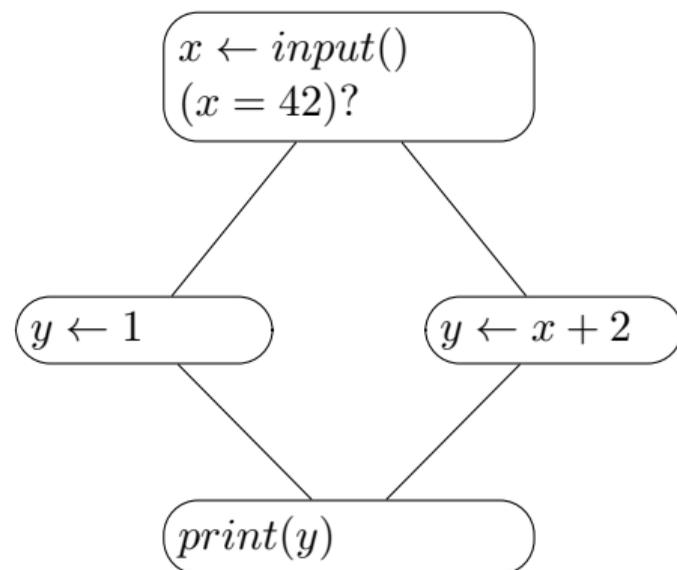
Мотивация: хотим *хранить* некоторую информацию в каждой точке программы

Пример: храним в каждой точке программы информацию о константах



Разветвление потока управления (1/2)

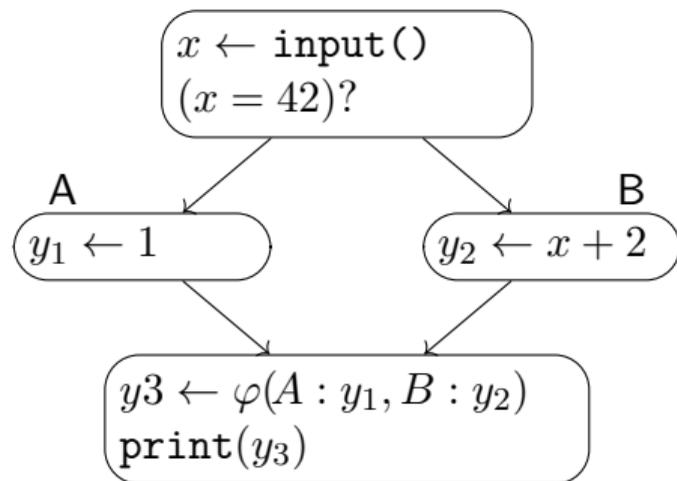
```
x = input();  
if (x==42)  
then  
  y = 1; /* A */  
else  
  y = x+2; /* B */  
end  
print(y);
```

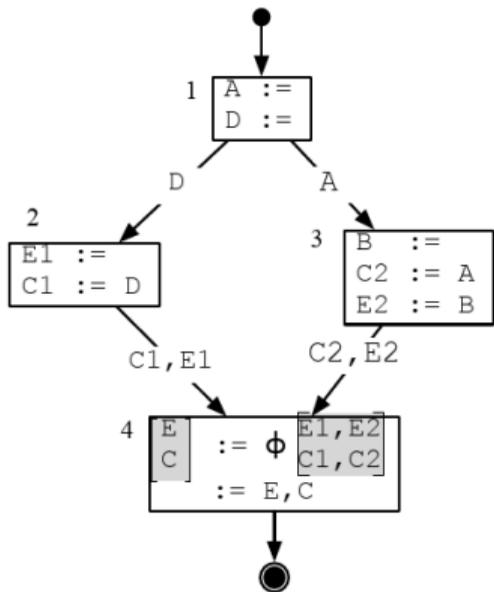


Разветвление потока управления (2/2)

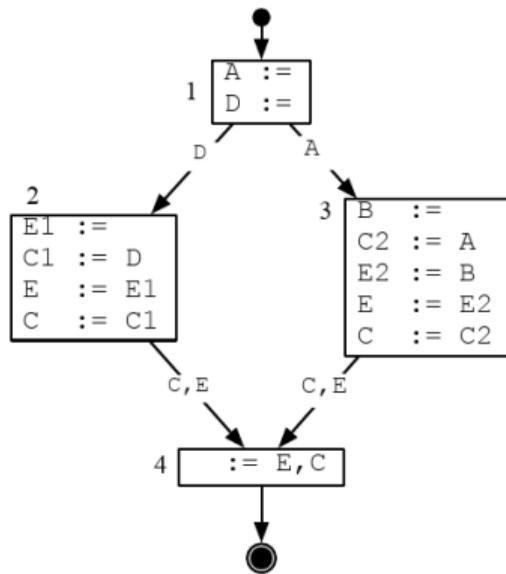
- Добавляем φ -функции (φ -узлы), чтобы смоделировать слияние потоков управления
- Операционно: выбираем один из аргументов в зависимости откуда пришли
- Несколько присваиваний φ -функций должны выполняться *параллельно* (одновременно)
- При кодогенерации от φ надо будет избавляться

```
x = input();  
if (x==42)  
then  
    y1 = 1; /* A */  
else  
    y2 = x+2; /* B */  
end  
y3 =  $\varphi(y_1, y_2)$ ;  
print(y3);
```

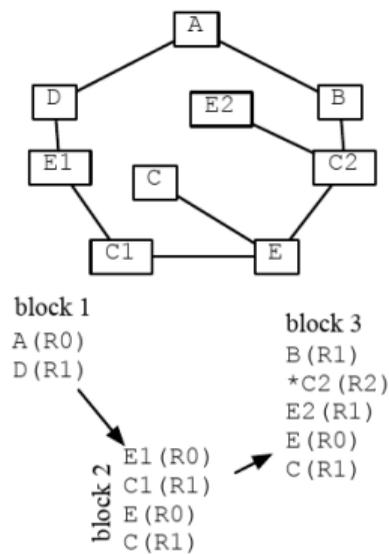




(a)



(b)



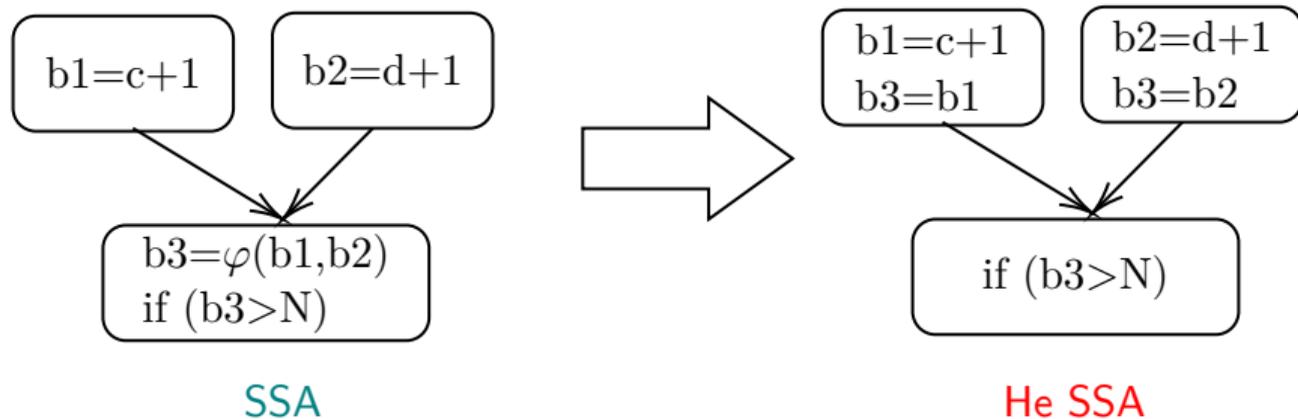
(c)

Рис.: (a) SSA-форма программы . (b) Программа после избавления от SSA. (c) Граф несовместимости и последовательность присвоений в регистры

Избавляемся от φ -функций

Основная идея: φ обозначает тот факт, что значение при слиянии потоков управления может прийти по различным путям управления

- Сделаем присваивание на каждом пути управления
- Несколько присваиваний φ -функций выполняется одновременно



Избавляемся от φ на практике

- Копирования из-за φ могут быть бесполезны
- Объединенное (joined) значение может быть не нужно далее в программе (зачем его оставлять?)
- Воспользуемся dead code elimination, чтобы убрать бесполезные φ
- Затем будет заниматься распределением регистров

Задача Spill Free Register Allocation (SFRA) для SSA формы имеет полиномиальное решение, но NP-полно в общем случае.

Большое количество полезных задач NP-полно для SSA формы, а следовательно NP-полно для всех программ

- Spill minimization
- (Optimal) Coalescing
- (Optimal) Live range splitting
- (Optimal) aliasing
- Pre-coloring

Можно попробовать использовать более простое представление программ, чтобы получались полиномиальные алгоритмы

Static Single Information (SSI)

По умолчанию SSA связывает некоторую информацию с парой переменная \times позиция в программе – это *плотный (dense) анализ*.

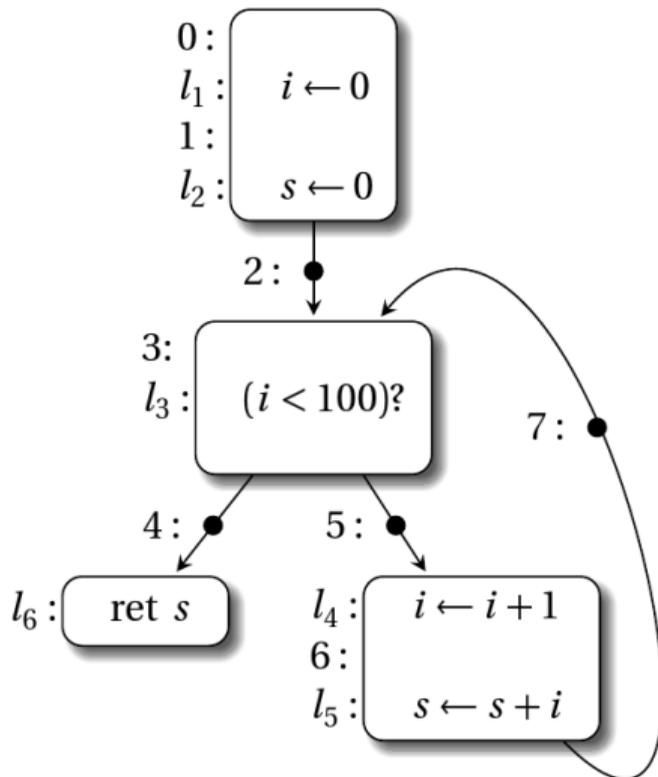
Для некоторых видов анализа будет удобнее связывать иметь общую информацию по всей программе с конкретной переменной, а не каждым её вхождением (*sparse анализ*).

Это удобно для таких анализов:

- вывод классов с виртуальными методами по классам-таблицам (Python, Javascript, Ruby)
- Null pointer анализ
- интервалы значений (далее будет пример)
- live range splitting (важно в контексте выделения регистров)

Пример плотного data-flow анализа: интервальный анализ

```
1  $i \leftarrow 0$ 
2  $s \leftarrow 0$ 
3 while  $i < 100$ 
  do
4    $i \leftarrow i + 1$ 
5    $s \leftarrow s + i$ 
6 return  $s$ 
```

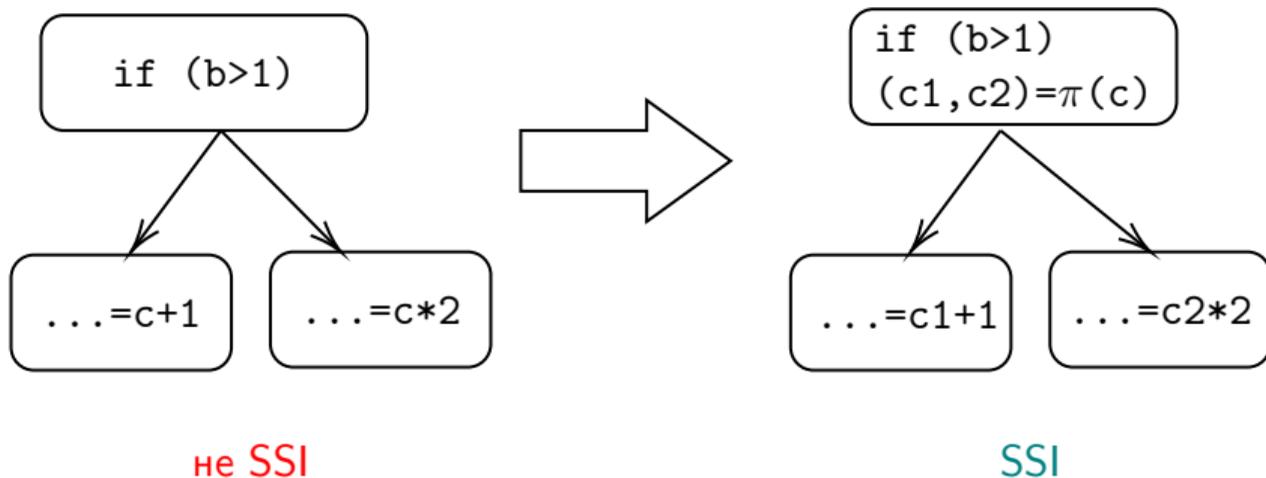


<i>prog. point</i>	[<i>i</i>]	[<i>s</i>]
0	top	top
1	[0,0]	top
2	[0,0]	[0,0]
3	[0,100]	[0,+∞[
4	[100,100]	[0,+∞[
5	[0,99]	[0,+∞[
6	[0,100]	[0,+∞[
7	[0,100]	[0,+∞[

Static Single Information (SSI) форма

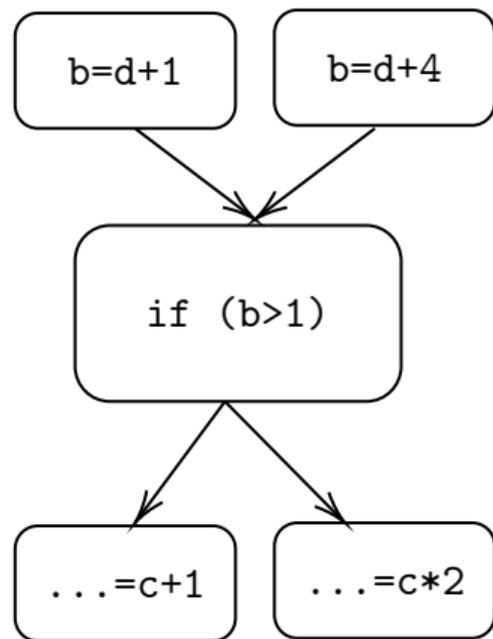
Программа в SSI форме:

- Каждый базовый блок заканчивается π -функцией, которая переименовывает переменные, которые “живы” на выходе из базового блока

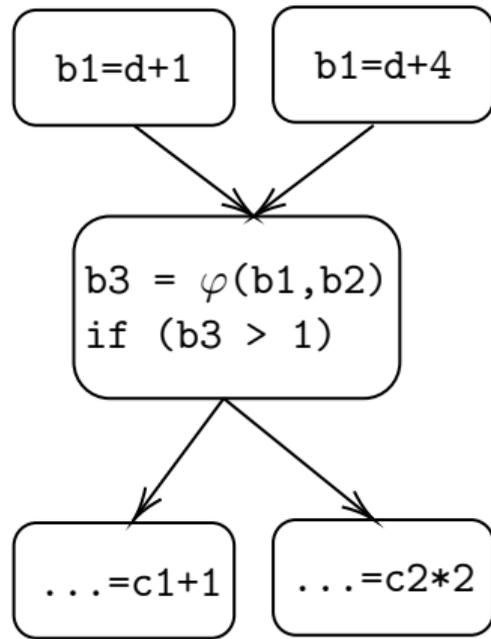


Базовый блок – это последовательность инструкций с только одной точкой входа и только одной точкой выхода.

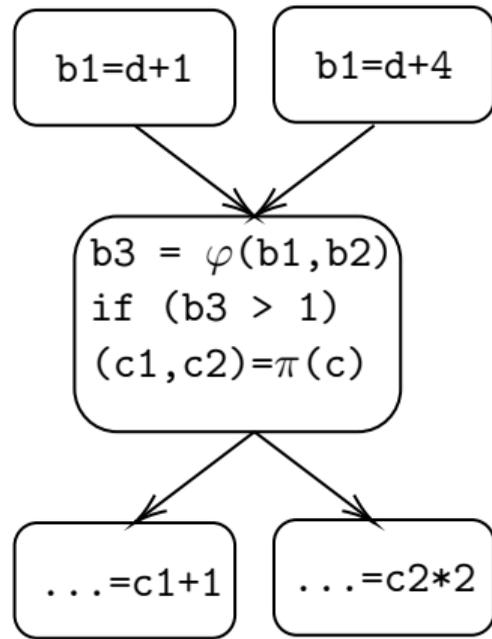
Примеры SSA и SSI



Не SSA и не SSI



SSA, но не SSI

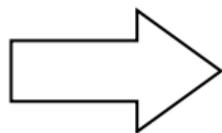


SSA и SSI

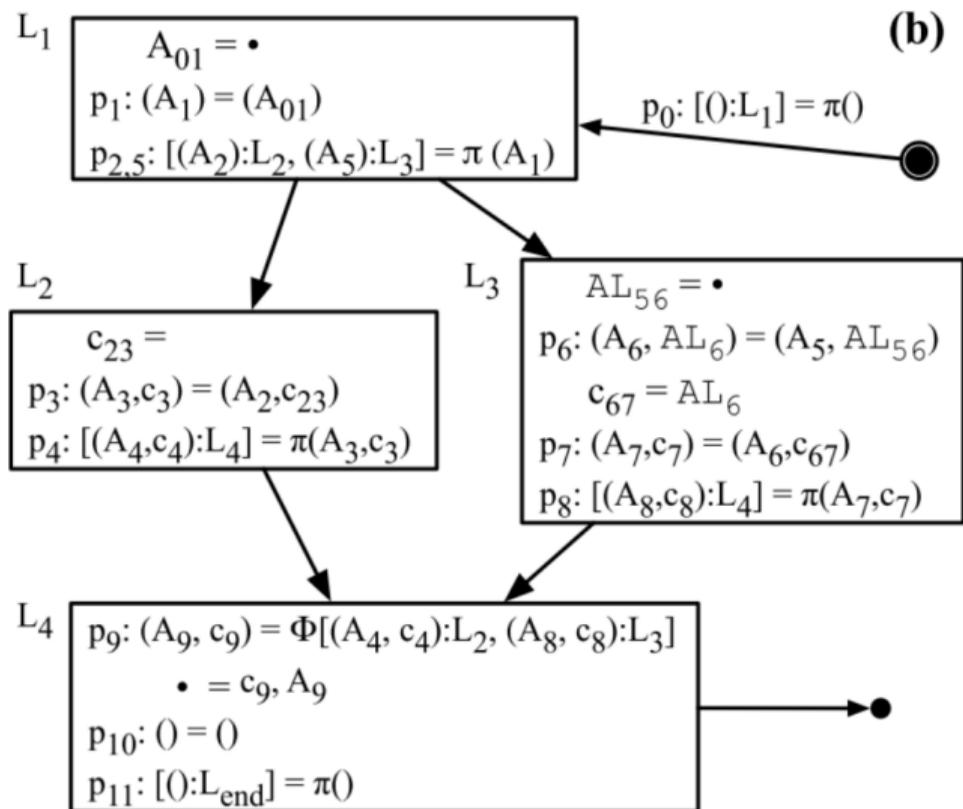
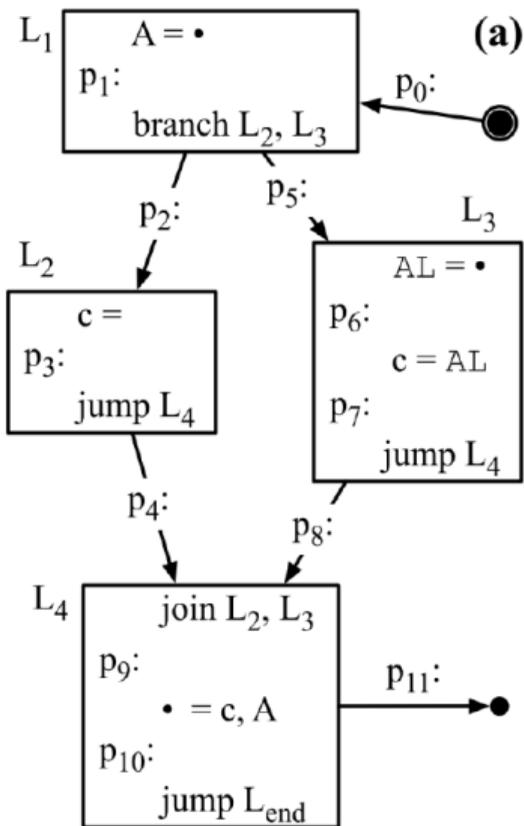
Параллельное копирование

По тем же причинам, по которым несколько φ -функций должны исполняться параллельно, копирование тоже должно происходить параллельно

$$\begin{aligned} V &= X + Y \\ Z &= A + B \end{aligned}$$



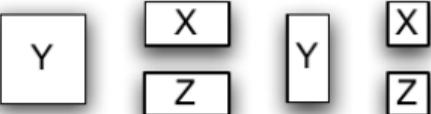
$$\begin{aligned} (V_1, X_1, Y_1, Z_1, A_1, B_1) &= (V, X, Y, Z, A, B) \\ V_1 &= X_1 + Y_1 \\ (V_2, X_2, Y_2, Z_2, A_2, B_2) &= (V_1, X_1, Y_1, Z_1, A_1, B_1) \\ Z_2 &= A_2 + B_2 \end{aligned}$$



Оглавление

- 1 Задача распределения регистров
- 2 Особенности, осложняющие задачу
- 3 Терминология
- 4 Стандартные алгоритмы (prior work)
- 5 Построение элементарных программ
- 6 Отображение элементарных программ в фигурки пазлов**
- 7 Решаем пазлы

Виды паззлов

	Board	Kinds of Pieces
Type-0		
Type-1		
Type-2		

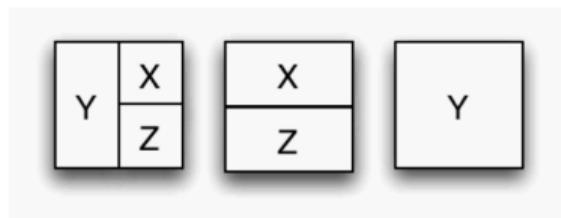
- Тип 0: PowerPC & ARM integers
- Тип 1: ARM float
- Тип 2: SPARC V8
- Гибрид 0+1: x86
- Гибрид 1+2: SPARC V9

Абстракция паззлов

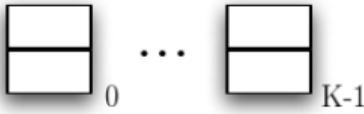
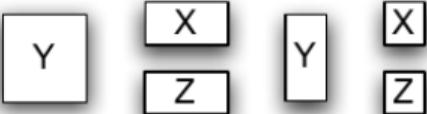
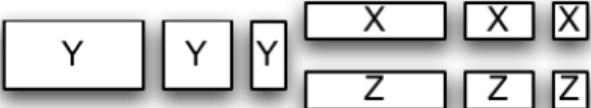
- Пазл = доска (зоны = регистры) + фигурки (переменные)



- Фигурки не могут перекрываться
- Некоторые фигурки могут быть уже выложены на доску
- Задача: уместить оставшиеся фигурки на доску (**register allocation**)



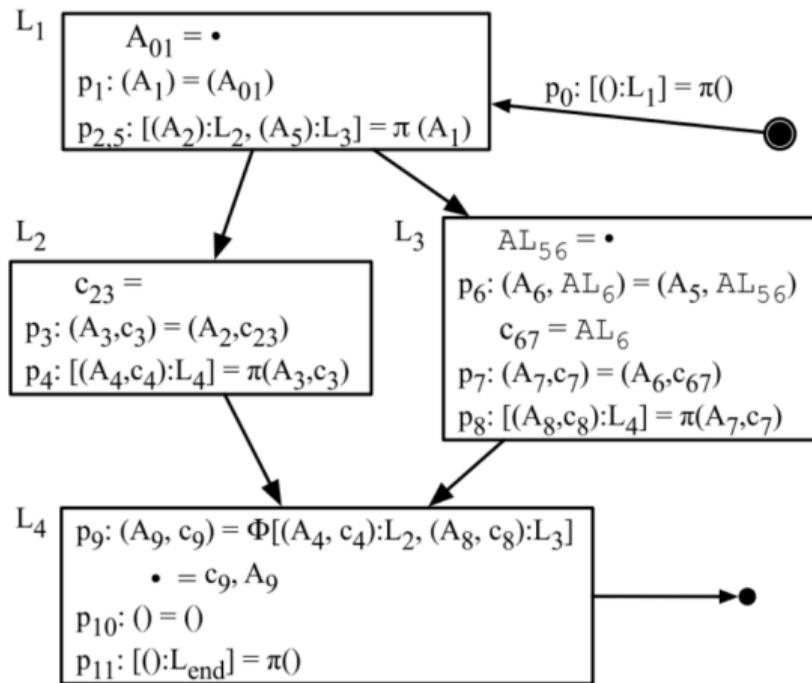
Создание фигурок пазла

	Board	Kinds of Pieces
Type-0		
Type-1		
Type-2		

Для каждой инструкции в программе i :

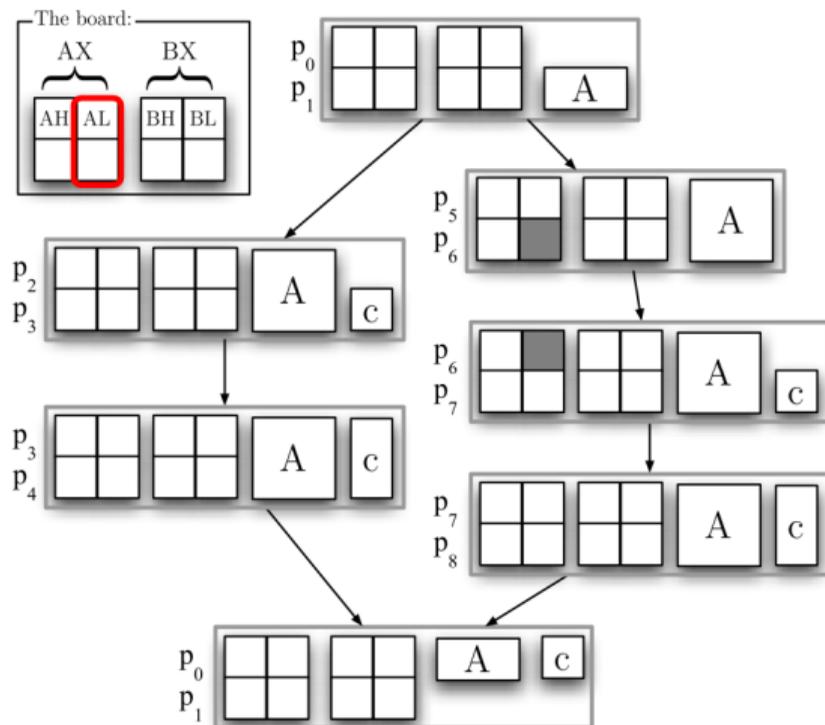
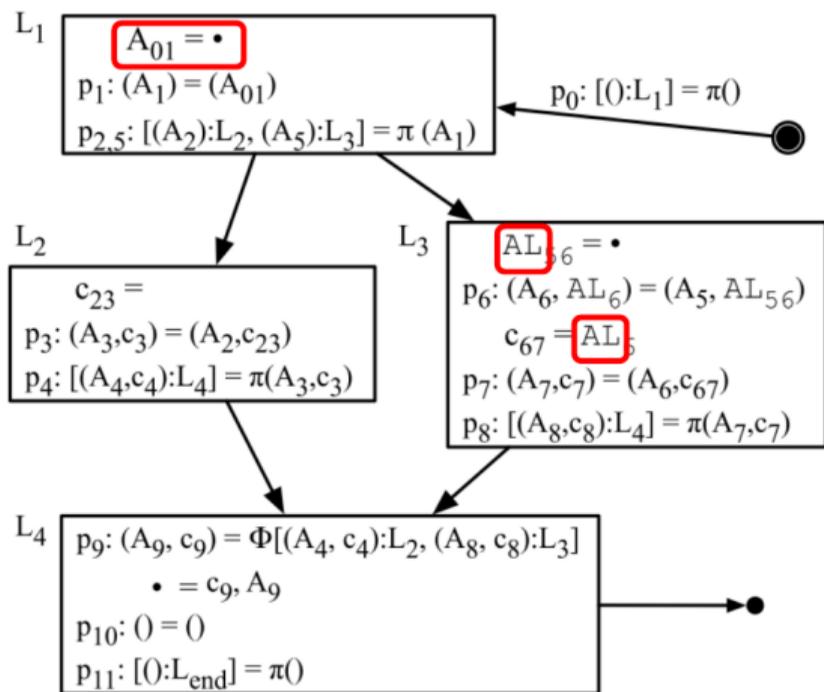
- Создать по одной фигурке для live-in и live-out переменных
- Если переменная перестает быть живой ниже i , тогда тип фигурки X
- Если переменная начинает быть живой в i , тогда Z
- Иначе Y

Пример. Создание фигурок



Variables	$p_x: (C, d, E, f, g) = (C', d', E', f')$ $A, b = C, d, E$ $p_{x+1}: (A'', b'', E'', f'', g'') = (A, b, E, f)$
Live Ranges	
Pieces	

Пример. Padding



Оглавление

- 1 Задача распределения регистров
- 2 Особенности, осложняющие задачу
- 3 Терминология
- 4 Стандартные алгоритмы (prior work)
- 5 Построение элементарных программ
- 6 Отображение элементарных программ в фигурки пазлов
- 7 Решаем пазлы**

Решение паззлов типа 1

- Предлагаемый подход: заполнять по одной клетке за раз
- Для каждого квадрата:
 - Заполняем фигурками X или Z пока не заполнится вся доска
- Решить задачу

Решаем пазлы типа 1 визуальным языком (1/3)

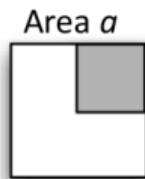
(Program) $p ::= s_1 \dots s_n$

(Statement) $s ::= r \mid r : s$

(Rule) $r ::=$ 



- Правило = как заполнять зоны
- Правилл состоит из
 - шаблона: что уже заполнено (подходит/неподходить под зону)
 - стратегии: какие виды фигурок класть и куда
- Правило r можно применить на зоне a iff
 - r подходит к a
 - доступны фигурки для данной стратегии



Решаем пазлы типа 1 визуальным языком (2/3)

(Program) $p ::= s_1 \dots s_n$

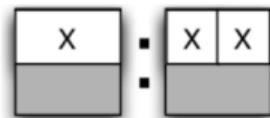
(Statement) $s ::= r \mid r : s$

(Rule) $r ::=$ 



Условие успешного завершения:

- Программа успешно завершается iff все утверждения завершаются
- Правило r можно применить на зоне a iff
 - r подходит к a
 - доступны фигурки для данной стратегии
- Условие $(r : s)$ успешно завершается iff
 - r успешно завершается or
 - s успешно завершается



Решаем пазлы типа 1 визуальным языком (3/3)

(Program) $p ::= s_1 \dots s_n$

(Statement) $s ::= r \mid r : s$

(Rule) $r ::=$ 

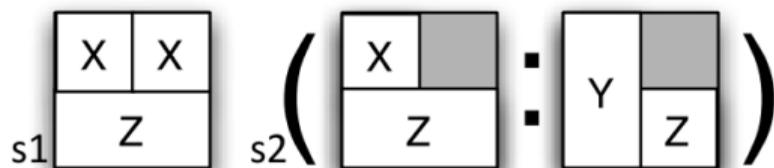


Исполнение решателя пазлов

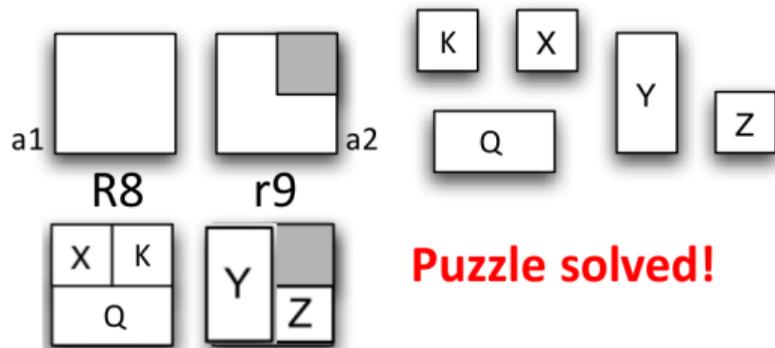
- Для каждой инструкции s_1, \dots, s_n
 - Для каждой зоны a , такой что паттерн s_i подходит к a
 - Применить s_i к a
 - Если s_i закончилось и ошибкой, прерваться и сообщить об ошибке

Исполнение решателя паззлов: пример

- Визуальная программа-решатель



- Паззл



- $s1$ подходит только к $a1$
- Применение $s1$ к $a1$ завершается успешно и выдает результат



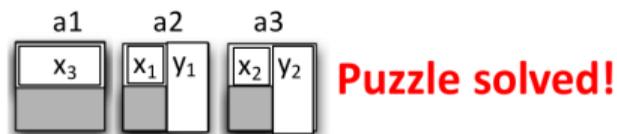
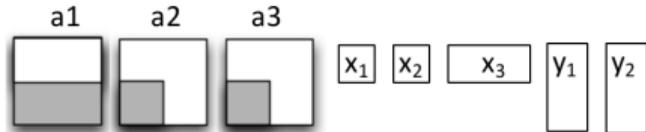
- $s2$ подходит только к $a2$
- Применяем $s2$ к $a2$
 - Применение первое правило $s2$: неудача
 - Применим второе правило $s2$: успех

Исполнение решателя паззлов: другой пример

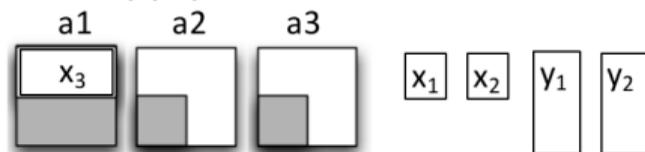
- Визуальная программа-решатель



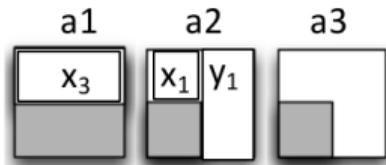
- Паззл



- s1 подходит только к a1
- Применяем s1 к a1
 - Применяем первое правило к s1:
победа



- s2 подходит и к a2, и к a3
- Применяем s2 к a2



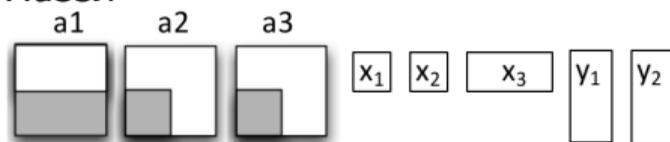
- Применяем s2 к a3

Исполнение решателя паззлов: ещё один пример

- Визуальная программа-решатель



- Паззл

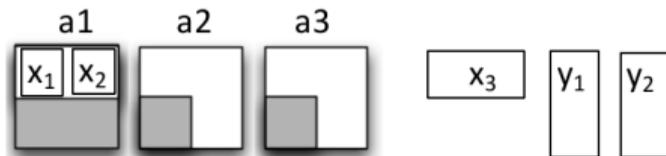


Построение правильной визуальной программы-решателя существенно!

- $s1$ подходит только к $a1$

- Применяем $s1$ к $a1$

- Применяем первое правило к $s1$: победа

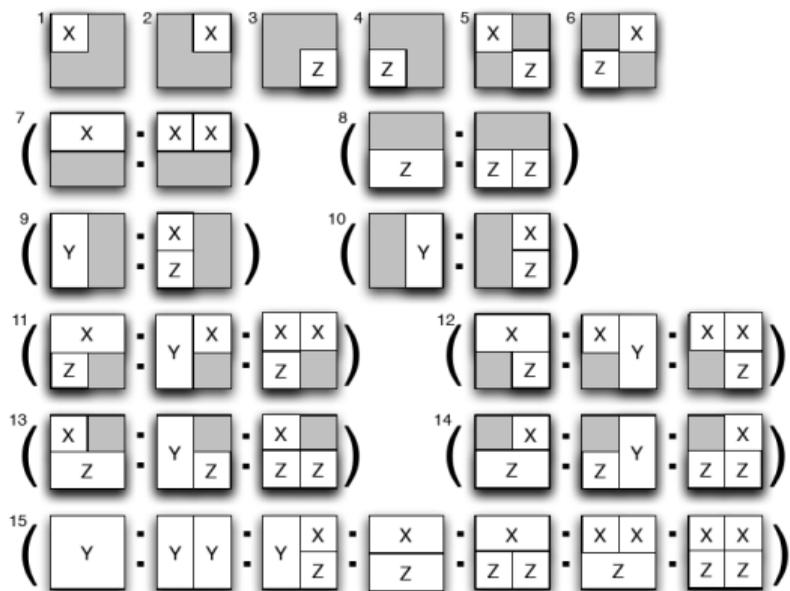


- $s2$ подходит и к $a2$, и к $a3$

- Применяем $s2$ to $a2$: **неудача**

Не осталось фигур типа X размера 1: мы использовали их всех

Программа-решатель паззлов типа 1



Теорема

Задача типа 1 решается iff эта программа завершается успешно

Не решили ли мы NP задачу за полиномиальное время?

Выделение регистров \sim заполнение всех клеток.

Решена более простая задача: заполнение одной клетки за единицу времени

Решение паззлов типа 1: сложность

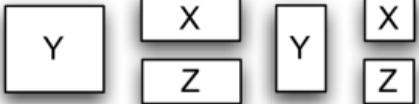
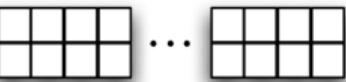
Теорема

Задача *spill-free register allocation (SFRA)* с предсраскраской для элементарной программы P и K регистров решается за $O(|P| \times K)$ времени.

Для одной инструкции из P :

- Применение правила к зоне: $O(1)$
- Применяется константное количество правил к каждой зоне
- Исполнение на поле с K зонами занимает $O(K)$ времени

Решаем пазлы типа 0 (SFRA)

	Board	Kinds of Pieces
Type-0		
Type-1		
Type-2		

- Уложить все Y-фигурки



- Затем уложить все X и Z фигурки

Spilling в общем случае

- Если алгоритм для решения SFRA не может решить пазл, (т.е. количество имеющихся регистров не достаточно) \Rightarrow spill
- Наблюдение: преобразование программы в элементарную форму создаёт семейства переменных для каждой оригинальной переменной
- Чтобы сделать spill:
 - Выбираем переменную v в оригинальной программе
 - Spillим все переменные из элементарной формы, которые относятся к тому же семейству, что и v
 - Для пазлов это будет означать выкидывание фигурок пазла

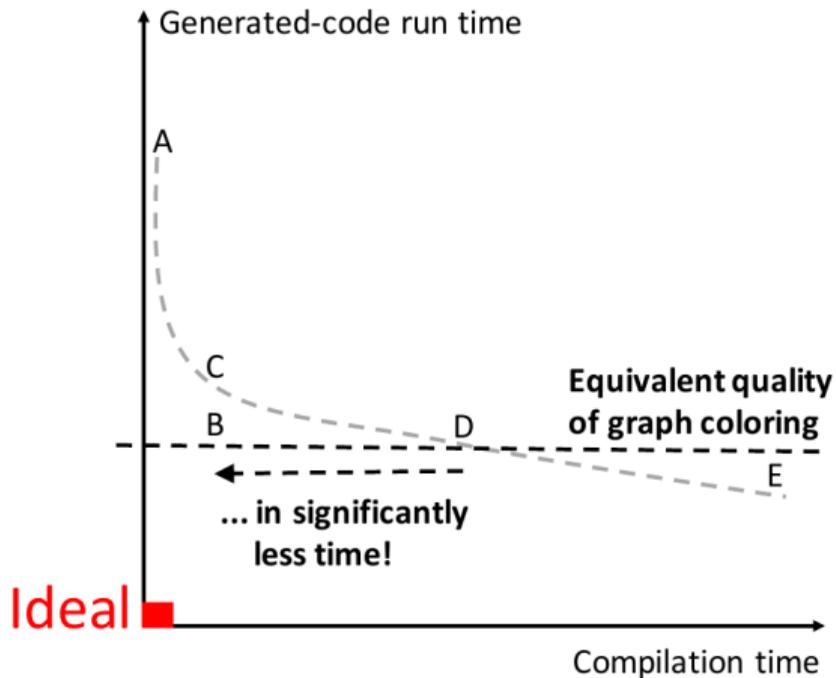
Теорема (Сложность)

Выделение регистров в присутствии предраскрашивания (pre-coloring) и spilling семейства переменных для элементарного представления – NP-полная задача.

Ещё один метод выделения регистров

Генерирует сходный по качеству код, но существенно быстрее, чем использовавшийся до этого в LLVM graph coloring with iterated coalescing.

Конец



 [Register Allocation by Puzzle Solving \(PLDI 2008\)](#)
Fernando Magno Quintão Pereira & Jens Palsberg
PDF

 [PhD thesis](#)
Fernando Magno Quintão Pereira

 [Slides from Compiler Construction course of Northeastern University](#)
PDF

 [SSA book PDF](#)