Формальное доказательство эквивалентности программ, скомпилированных в ABI со статическим и динамическим связыванием

Пьянкова Ю. А.
Научный руководитель: Малаховски Я. М.

17 июня 2015

Введение

- Цена ошибки в программе пропорциональна:
  - частоте проявления,
  - сложности поиска,
  - сложности исправления.
- Вполне вероятно, что на данном этапе развития технологий не про все программы экономически целесообразно доказывать их корректность.
- Однако про инструменты разработки ПО уже давно целесообразно, поскольку сложность поиска и исправления ошибок в них чрезвычайно высоки, а многие полезные теоремы уже доказаны.

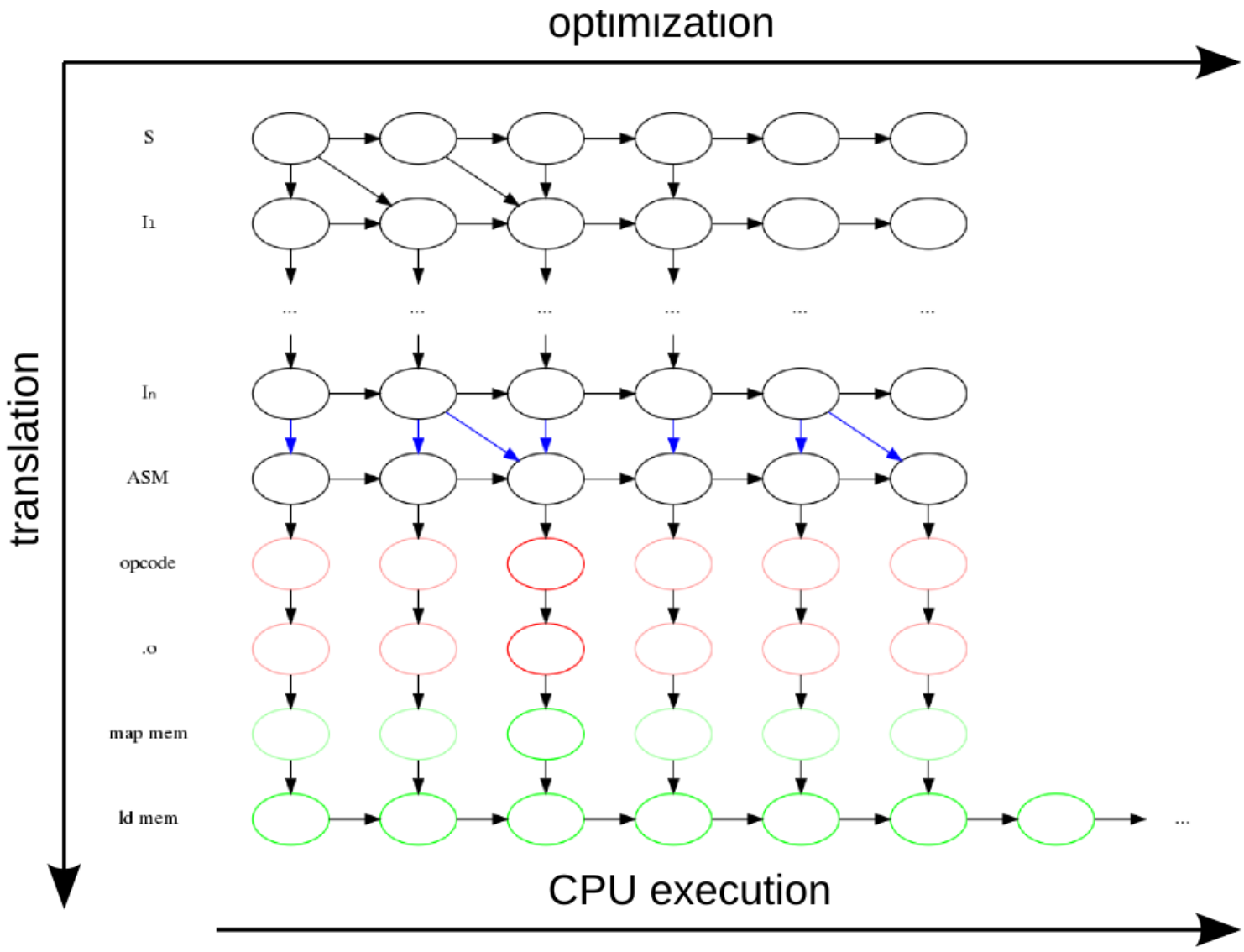
optimization
translation
S
I1
...
In
ASM
opcode
.o
map mem
ld mem
...
CPU execution

Рис. 1:

- S — изначальный язык программирования
- I — промежуточные языки, в которые транслируется код
- ASM — язык ассемблера
- opcode — машинный код
- .o — скомпилированные файлы с исполняемым кодом
- map mem — код, загруженный в память
- ld mem — пролинкованный код в памяти

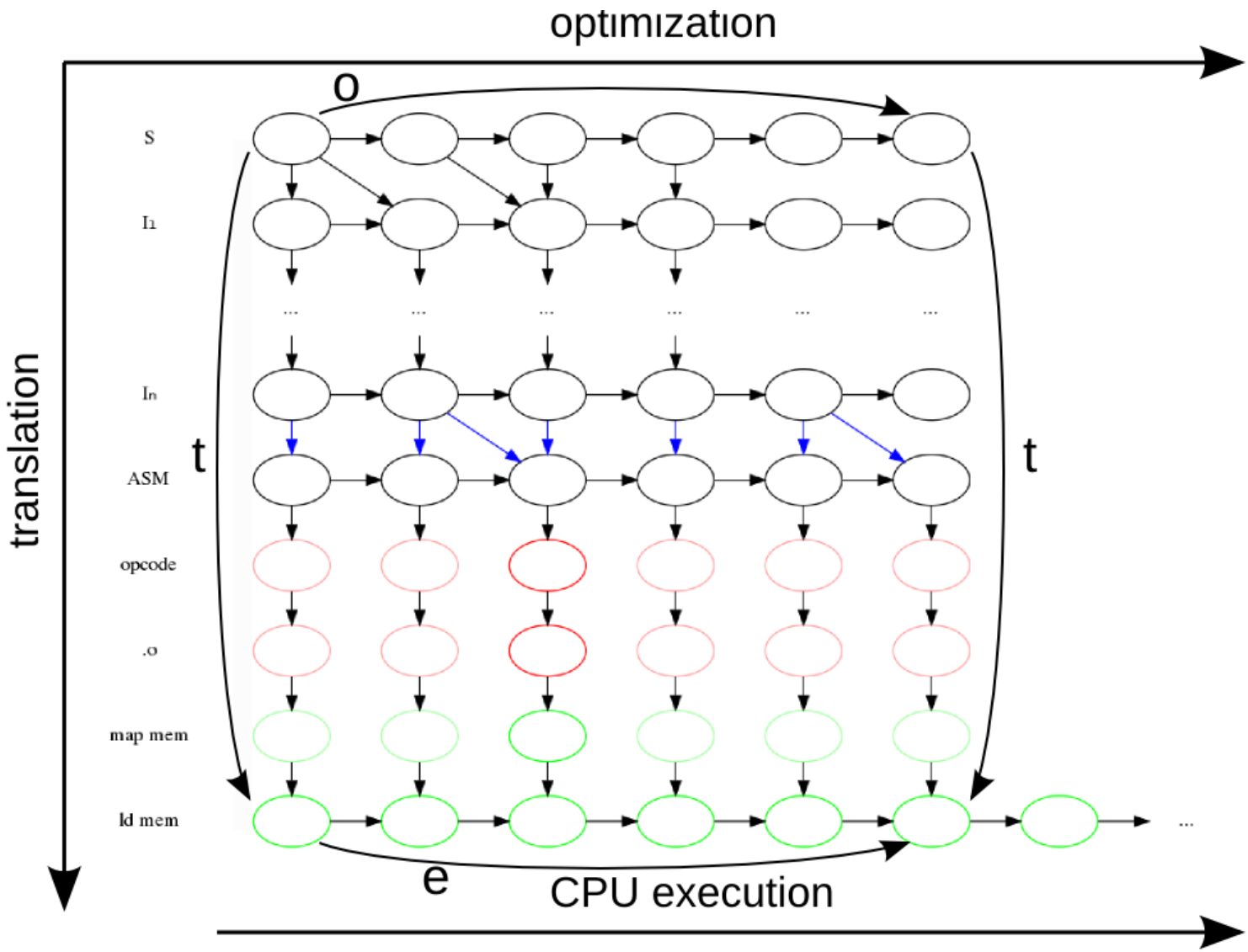

Рис. 2:

В идеальном мире диаграмма коммутативна: $t \circ o \equiv e \circ t$

Идеальный компилятор

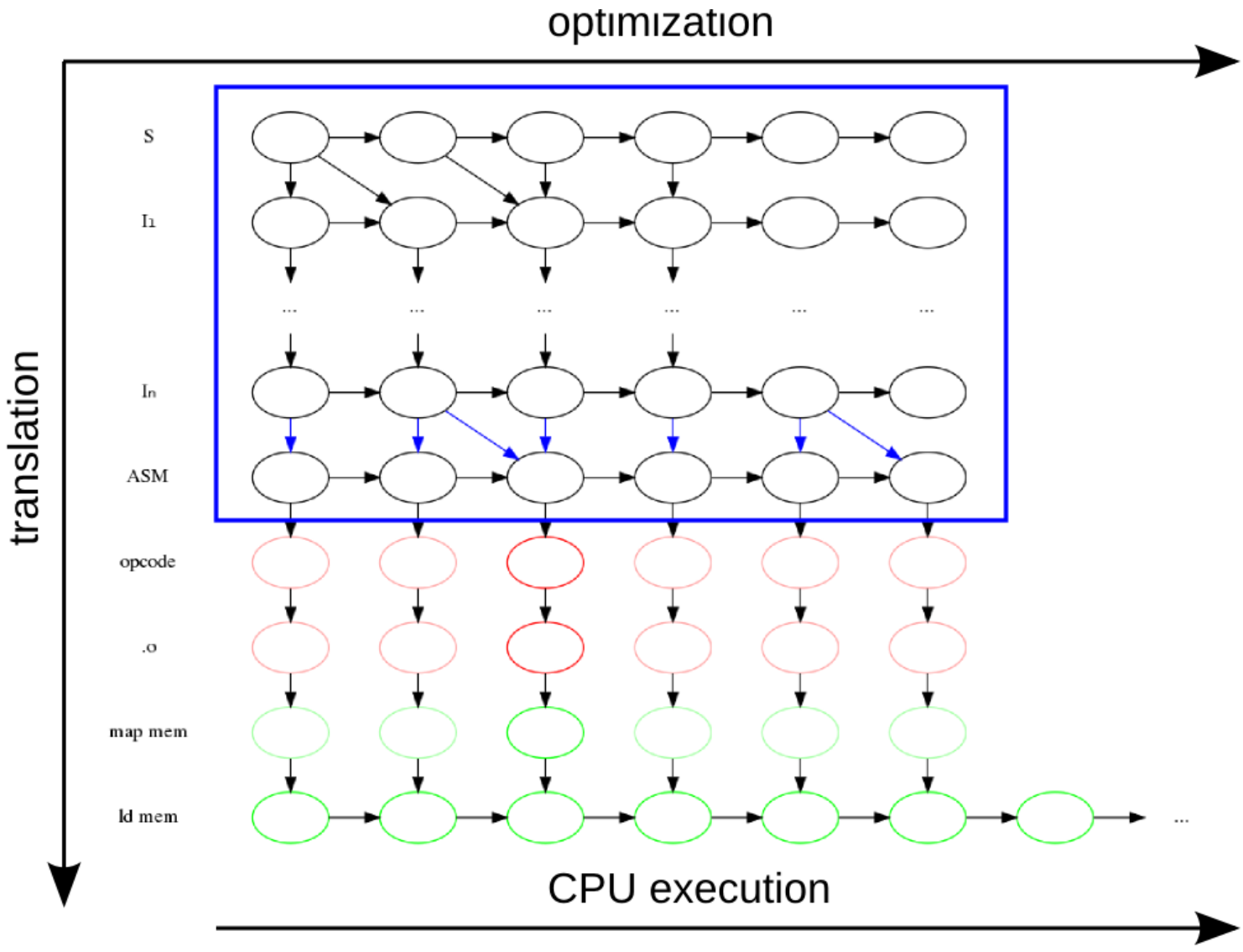

Рис. 3:

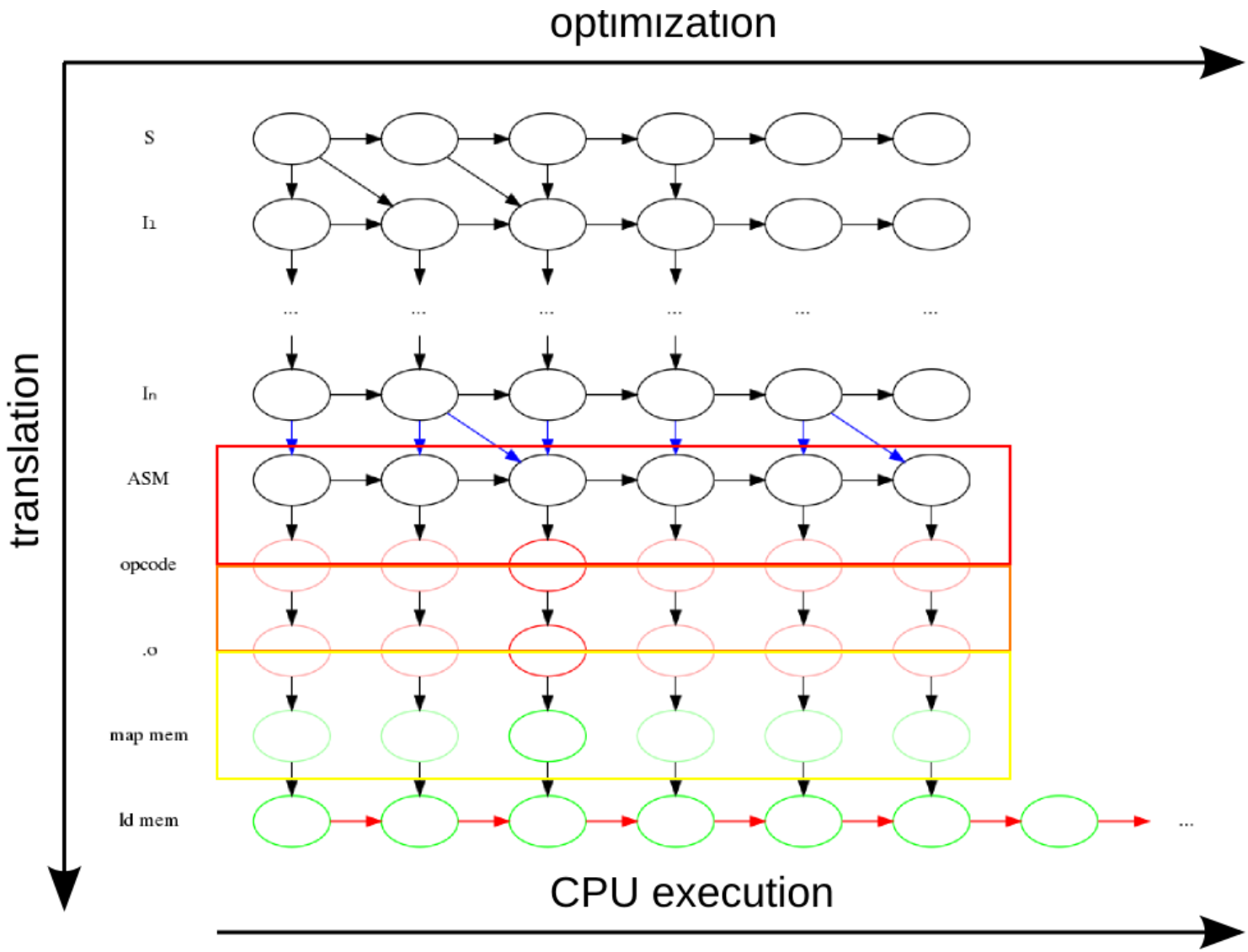
optimization
translation
S
I1
...
In
ASM
opcode
.o
map mem
ld mem
...
CPU execution

Рис. 4:

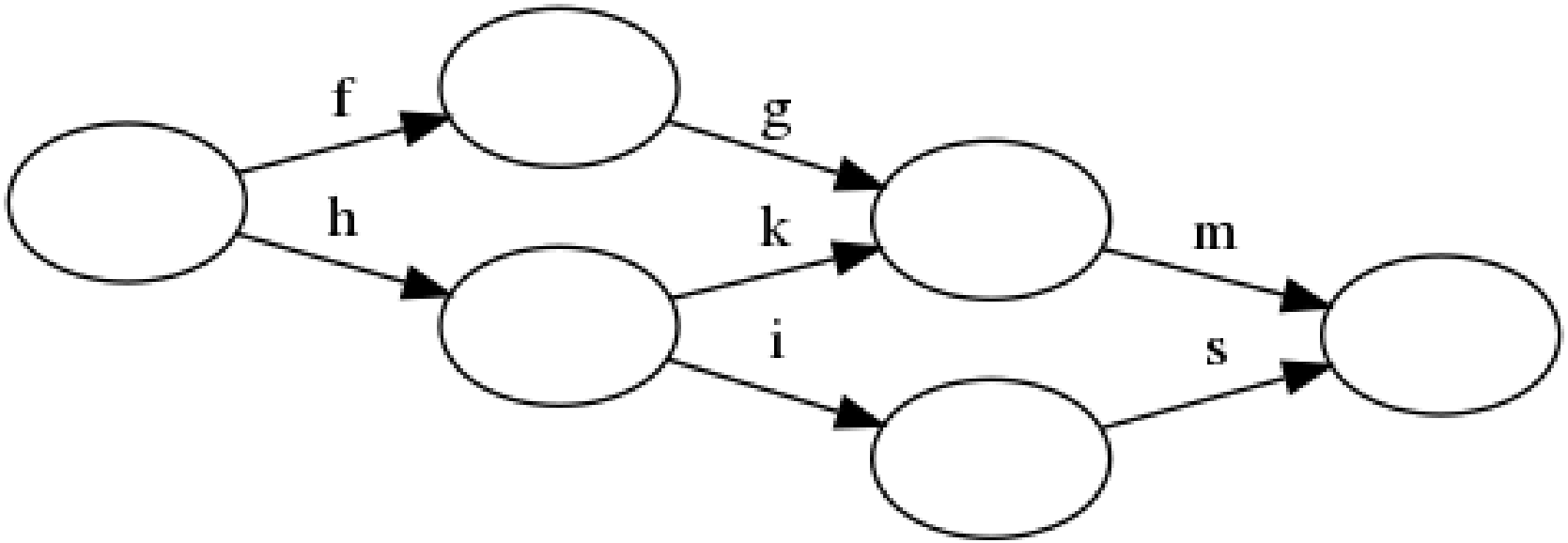

Рис. 5:

$$(g \circ f \equiv k \circ h) \to (m \circ k \equiv s \circ i) \to (m \circ g \circ f \equiv s \circ i \circ h)$$

- Выбор опкодов:
  - можно верифицировать, если иметь модель CPU;
- Генерация объектных файлов:
  - можно верифицировать, если формализовать формат ELF с его семантикой и реализовать доказанно корректный линковщик для него;
- Загрузка объектных файлов в память:
  - необходимо верифицированное ядро ОС;
  - верифицированные менеджеры памяти — большая открытая проблема;
- Техпроцесс:
  - можно только поверить.

Существующие решения

- Software Foundations
  - учебник с формализациями простых исчислений и доказательств о них на Coq
- Iron Lambda
  - продолжение Software Foundations на реально использующиеся исчисления
  - доказаны все стандартные теоремы про все стандартные лямбда-исчисления
  - реально используется в компиляторе Disciple
- CompCert
  - компилятор подмножества C с доказанно корректными оптимизациями
- VeLLVM
  - транслятор SSA ассемблера с доказанно корректными оптимизациями

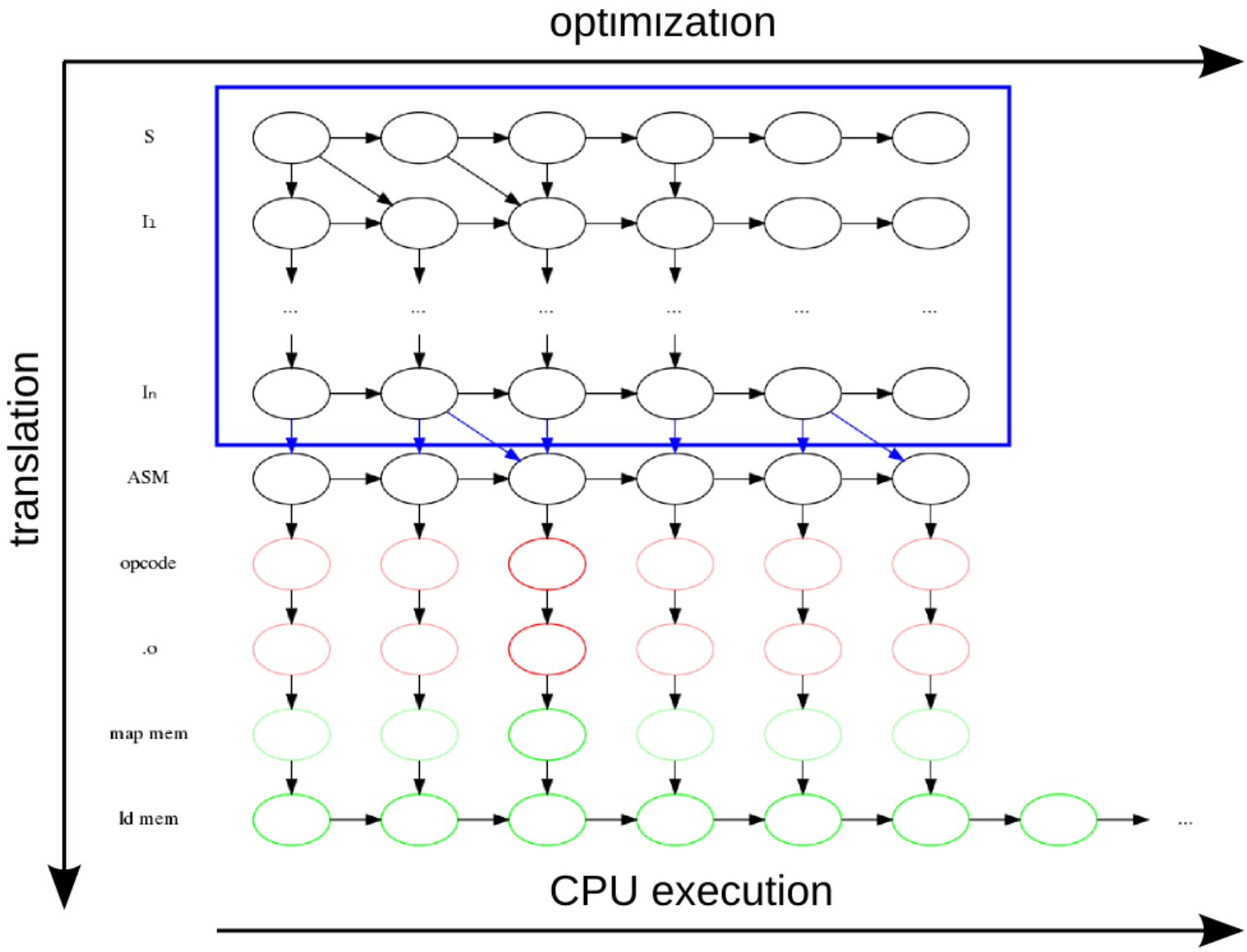
optimization
translation
S
I1
...
In
ASM
opcode
.o
map mem
ld mem
...
CPU execution

Рис. 6:

VeLLVM

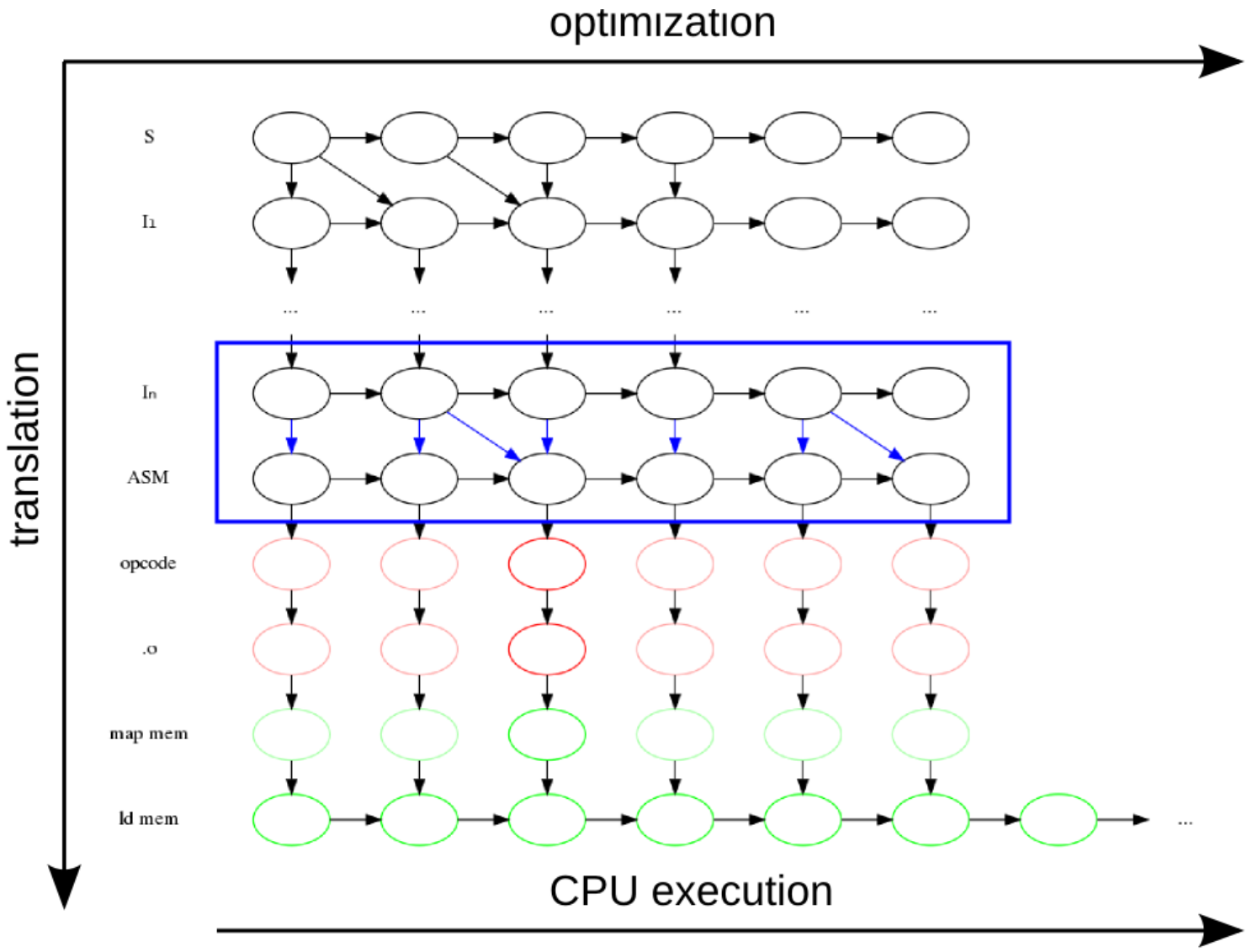
optimization
translation
S
I1
...
In
ASM
opcode
.o
map mem
ld mem
...
CPU execution

Рис. 7:

Цель работы

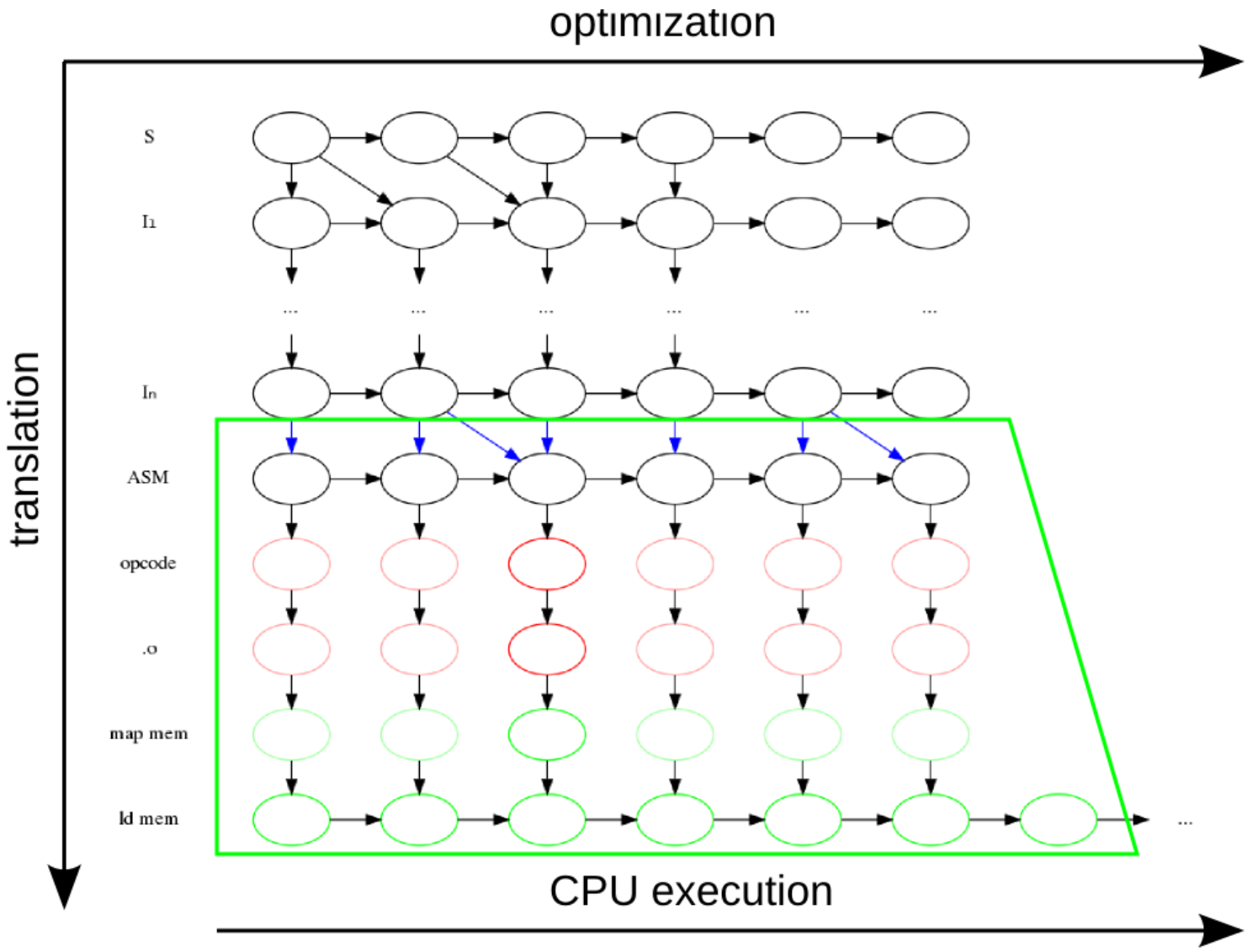

Рис. 8:

Цель работы

- Почему задача решаема:
  - мы верим:
    - соответствию ассемблера спецификации;
    - корректности работы с объектными файлами;
    - разработчикам ядра ОС;
    - производителям CPU;
  - не нужна работа с динамической памятью, что не требует решения больших открытых проблем.
- Проблемы:
  - программы соответствуют некоторому ABI, и надо доказывать сохранение семантики линковщиком с точностью до заданного ABI;
  - в области формальных доказательств существуют серьезные проблемы с переиспользованием доказательств для слегка измененных определений.

Основные сущности и типы

- Тип, значение которого может находиться в регистрах, и тип, описывающий состояние всех регистров:

```
data RegType : Set
RegTypes = List RegType
```

- Тип произвольного размера и тип, описывающий состояние памяти:

```
data Type : Set
DataType = List Type
```

- Стек вызовов и стек данных:

```
DataStackType = List RegType
CallStackType = List (RegTypes × DataStackType)
```

Основные сущности и типы

- Типы данных:

```
data RegType where
    _*  : Type → RegType
    int : RegType

data Type where
    atom : RegType → Type
    block : RegTypes
        → DataStackType
        → CallStackType
        → Type
```

Инструкции

- Управляющая инструкция:

```
data ControlInstr (S : StateType)
    : Maybe (CallStackChg S) → Set
```

- Не-управляющая инструкция:

```
data Instr (S : StateType) : SmallChg S → Set
```

Мета-ассемблер

- Обычно при небольшом изменении основных определений все доказательства приходится менять
- Общую для всех языков ассемблера часть можно определить независимо от конкретного языка ассемблера

```
module Blocks
    (ControlInstr : (S : StateType)
        → Maybe (CallStackChg S)
        → Set)
    (Instr : (S : StateType) → SmallChg S → Set)
    where
    data Block (S : StateType) : Diff S → Set
```

Мета-ассемблер

- Определения, относящиеся к мета-ассемблеру, можно получать по минимальному описанию конкретного языка ассемблера:
  - тип управляющей инструкции;
  - тип не-управляющей инструкции;
  - функция, определяющая результат исполнения управляющей инструкции;
  - функция, определяющая результат исполнения не-управляющей инструкции.

Ассемблер: примеры инструкций

- Непрямой jmp:

```
jmp[_] : (ptr : atom
    (block
    (StateType.registers S)
    (StateType.datastack S)
    (StateType.callstack S) *)
    ∈ StateType.memory S)
    → ControlInstr S nothing
```

- push значения из регистра:

```
push : ∀ {r}
    → r ∈ StateType.registers S
    → Instr S (onlystack (StackDiff.push r))
```

Линковка

- При динамической линковке в память добавляются таблицы GOT и PLT:

  $$\text{pltize} : \text{DataType} \to \text{DataType}$$

- Элемент таблицы PLT выглядит следующим образом:

  $$\begin{aligned}&\text{plt-stub} : \forall\ \{\text{R H DS CS}\} \to \text{atom}\ (\text{block R DS CS}\ *) \in \text{H}\\ &\quad \to \text{Block}\ (\text{statetype R H DS CS})\ \text{dempty}\\ &\text{plt-stub got} = \rightsquigarrow \text{jmp[got]}\end{aligned}$$

Леммы

- Состояние исполнителя в момент непосредственного вызова функции эквивалентно состоянию исполнителя после исполнения непрямого jmp по указателю на ее тело:

```
∀ {ST} → (S : State ST)
  → (p : atom (block
      (StateType.registers ST)
      (StateType.datastack ST)
      (StateType.callstack ST)
*) ∈ StateType.memory ST)
  → exec-block S (⇝ jmp[ p ])
  ≡ S
  , loadblock
      (State.memory S)
      (loadptr (State.memory S) p)
```

Леммы

- Состояние исполнителя в момент непосредственного вызова функции эквивалентно состоянию исполнителя после исполнения соответствующего этой функции элемента PLT при условии корректно заполненного GOT:

$$
\begin{aligned}
&\forall\ \{\mathrm{R\ H\ DS\ CS}\} \\
&\quad \rightarrow (\mathrm{f} : \mathrm{block\ R\ DS\ CS} \in \mathrm{H}) \\
&\quad \rightarrow (\mathrm{S} : \mathrm{State\ (statetype\ R\ (pltize\ H)\ DS\ CS)}) \\
&\quad \rightarrow \mathrm{GOT[\ f\ ]\text{-}correctness\ (State.memory\ S)} \\
&\quad \rightarrow \mathrm{exec\text{-}block\ S\ (plt\text{-}stub\ (got\ f))} \\
&\quad \equiv \mathrm{S\ ,\ loadblock\ (State.memory\ S)\ (func\ f)}
\end{aligned}
$$

Эквивалентность блоков

- Блок A в состоянии исполнителя S_1 и блок B в состоянии исполнителя S_2 считаются эквивалентными, если конструктивно существует блок C в состоянии исполнителя S_C, достижимый из A в состоянии S_1 и из B в состоянии S_2 (частный случай bisimulation).

```
data BlockEq
  : {ST₁ ST₂ : StateType}
  → {d₁ : Diff ST₁} {d₂ : Diff ST₂}
  → (S₁ : State ST₁) (S₂ : State ST₂)
  → (A : Block ST₁ d₁) → (B : Block ST₂ d₂)
  → Set
```

Эквивалентность программ

- Программа A — набор блоков памяти, подмножеством которого является набор блоков кода, среди которых указан стартовый блок $main_A$
- Две программы A и B считаются эквивалентными, если для любого корректного состояния исполнителя S_{start} эквивалентны $main_A$ в состоянии S_{start} и $main_B$ в состоянии S_{start}
- Отношение bisimulation является отношением эквивалентности, следовательно, подстановочным. Тогда если для любого состояния S блок A эквивалентен блоку B, то замена блока A на блок B в программе не влияет на результат исполнения.

Теорема

- При корректно заполненном GOT верна внешняя эквивалентность блока PLT, использующего соответствующий функции элемент GOT, и самой функции:

$$
\begin{aligned}
&\forall\ \{\mathrm{R\ H\ DS\ CS}\} \\
&\quad \to (\mathrm{f} : \mathrm{block\ R\ DS\ CS} \in \mathrm{H}) \\
&\quad \to (\mathrm{S} : \mathrm{State}\ (\mathrm{statetype\ R}\ (\mathrm{pltize\ H})\ \mathrm{DS\ CS})) \\
&\quad \to \mathrm{GOT[\ f\ ]\text{-}correctness}\ (\mathrm{State.memory\ S}) \\
&\quad \to \mathrm{BlockEq\ S\ S} \\
&\qquad (\mathrm{plt\text{-}stub}\ (\mathrm{got\ f})) \\
&\qquad (\mathrm{proj_2}\ \$\ \mathrm{loadblock}\ (\mathrm{State.memory\ S})\ (\mathrm{func\ f}))
\end{aligned}
$$

Эквивалентность программ, слинкованных статически и динамически

- В предположении, что добавление новых элементов в релоцируемый код не изменяет семантику программы, эквивалентны:
  - оригинальная программа (статически слинкованная программа);
  - программа с добавленными таблицами GOT и PLT;
  - программа с добавленными таблицами GOT и PLT с заменой вызовов функций на вызовы соответствующих элементов PLT.
- Последний пункт после загрузки в память по определению является динамически слинкованной программой с точностью до перестановки данных, расположенных в памяти.

Результаты

- Реализован фреймворк (мета-ассемблер), обобщающий определения основных сущностей и эквивалентности блоков для различных языков ассемблера.
- Формализовано подмножество ассемблера x86_64 на базовых блоках со стеком и типизированной статической памятью:
  - подмножество, необходимое для описания элементов PLT;
  - основные управляющие инструкции, совершающие безусловные переходы;
  - загрузка значения из памяти в регистр;
  - основные инструкции, работающие со стеком данных.
- Формализовано подмножество ABI, касающееся компоновки со статическим и динамическим связыванием.

Результаты

- Доказана (интенсиональная) эквивалентность состояний исполнителя в момент непосредственного вызова функции и после исполнения непрямого jmp по указателю на тело функции
- Доказана (интенсиональная) эквивалентность состояний исполнителя в момент непосредственного вызова функции и после исполнения соответствующего этой функции элемента PLT при условии корректно заполненного GOT
- Доказана внешняя эквивалентность (эквивалентность вызовов) некоторой функции и соответствующего ей элемента PLT при условии корректно заполненного GOT
- В указанном ранее предположении доказана эквивалентность статически и динамически слинкованных программ

Вопросы?